Algorithms in Comp. Bio.

Today: (Semi-) practical stuff - BLAST + FASTA

Recap

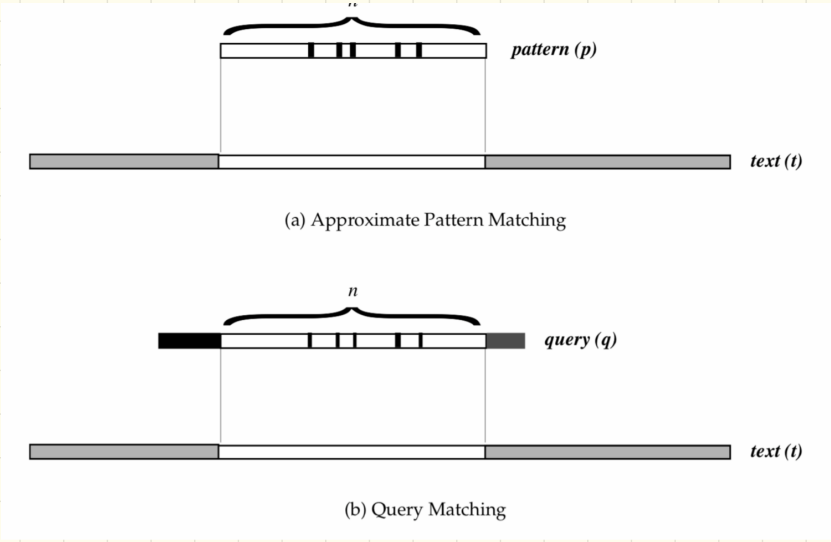
- How to find papers:
  call the library!
- Essay - by next Tuesday
- Next HW - look for it tomorrow

Back to our pattern matching in bacteria:

Approximate Pattern Matching Problem:
Find all approximate occurrences of a pattern in a text.

Input: A pattern $\mathbf{p} = p_1 p_2 \dots p_n$, text $\mathbf{t} = t_1 t_2 \dots t_m$, and parameter k, the maximum number of mismatches.

Output: All positions $1 \leq i \leq m - n + 1$ such that $t_i t_{i+1} \dots t_{i+n-1}$ and $p_1 p_2 \dots p_n$ have at most k mismatches (i.e., $d_H(\mathbf{t}_i, \mathbf{p}) \leq k$).



Query Matching Problem:
Find all substrings of the query that approximately match the text.

Input: Query $\mathbf{q} = q_1 \dots q_p$, text $\mathbf{t} = t_1 \dots t_m$, and integers n and k.

Output: All pairs of positions (i, j) where $1 \leq i \leq p - n + 1$ and $1 \leq j \leq m - n + 1$ such that the n-letter substring of $\mathbf{q}$ starting at i approximately matches the n-letter substring of $\mathbf{t}$ starting at j, with at most k mismatches.

Last week, we saw several $O(km)$ approaches to approximate pattern matching.

(brute force took $O(mn)$)

k = # of mismatches

Query matching:

Same sorts of trade-offs.

However: for both, can be faster to do filtering.

Idea:

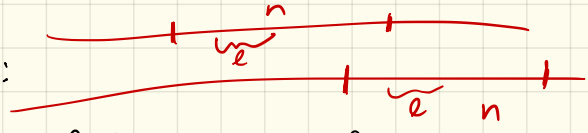
- Select candidates which are "likely"
- Focus on them + verify

Trade-off:

How few candidates can we get?

What do we miss?

Example:



ℓ-mer filtration: if an n-letter P substring matches an n-letter substring of T, then some ℓ-mer is identical.

Key: What is ℓ?

Note: ℓ-mers in common can be found by hashing:

Pre-hash everything in data base, T.

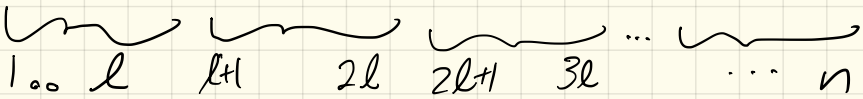
(This is part of software).

If there aren't many - use these to isolate likely matches.

Pinning this down:

Theorem 9.1 *If the strings $x_1 \dots x_n$ and $y_1 \dots y_n$ match with at most k mismatches, then they share an l-mer for $l = \lfloor \frac{n}{k+1} \rfloor$, that is, $x_{i+1} \dots x_{i+l} = y_{i+1} \dots y_{i+l}$ for some $1 \le i \le n - l + 1$.*

Pf: Partition 1...n into k+1 groups:



↑ k+1 buckets

K+1 groups, k mismatches

one bucket contains 0 mismatches.

(pigeonhole principle.)

Algorithm:

- Find all matches of ℓ-mers, for $\ell = \lfloor \frac{n}{k+1} \rfloor$.

How? Hashing.

- For each potential match, expand to left + right until $k+1$ mismatches are found.

How? suffix trees (see last week)

In practice: What the heck do you use??

An example: BLAST
("basic local alignment seach tool")

Use: Approximate local alignment + local similarity

Note: not what CS people call approx.

We'll discuss high level idea of which algorithms it uses.

BLAST: - fast in practice!

- also outputs range of solutions
- each match is accompanied by estimate of statistical significance.

BLAST concentrates on finding high local similarity:

Fundamental objects:

- Segment pairs: equal length substrings aligned with no spaces
- locally maximal segment pairs: one whose alignment score (no space) would drop by expanding or shortening
- maximal segment pair: pair w/ maximum score over all segment pairs

MSP →

- BLAST compares all sequences in a database w/ a fixed query P.

- Goal is MSP above some cut off C.

  ↳ look for lowest C s.t. MSP w/ score above C is unlikely to occur by chance.

  So, any sequence w/ MSP score $\geq C$ is "significant"

  How to find C?

  [Karlin-Altschul 1990]

  [Dembo-Karlin 1991]

Proteins: To find MSP above C in a protein database:

- For fixed length w & threshold t,

  find all length w substrings of S that align to length w substring of P w/ alignment score $\geq t$.

  These are hot-spots (or hits).

  Then extend to see if these are contained in a segment pair w/ score $\geq C$.

To find hits:

- Basically variation on filtering.
- Then try to extend.

↑ very greedy

BLAST trade-offs:

- no guarantees
  (b/c truncates early if score is bad)
- no dynamic programming
  (since no spaces)
- choice of w, t, C, & scoring matrix are critical.

  For example:

  Lowering t reduces chance of misses, but increases computation time.

- Lots of empirical investigation into what parameters should be, as well as comparison to FASTA.

A note on FASTA:

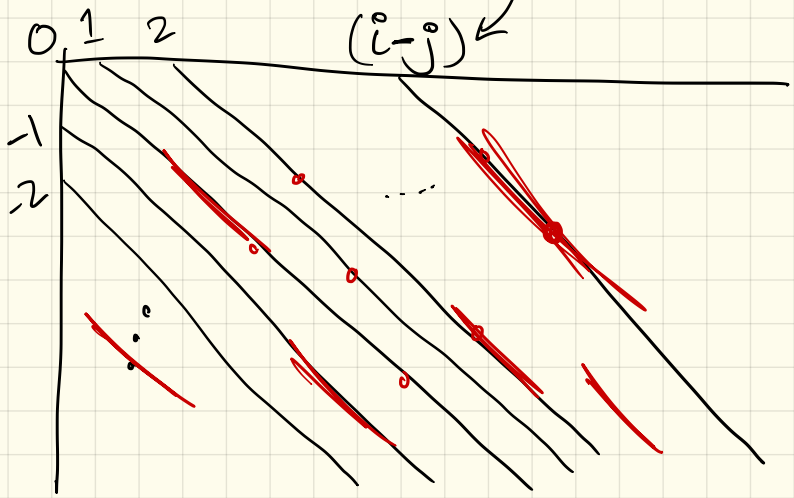
- More dynamic programming based (although faster!)

Idea:

- Find hot-spots (or hits), usually by pre-hasing entire database

  (so exact matching for query can be run).

- Result is pairs (i, j) of exact matches.



Next:

- find 10 best diagonal runs:

  sequence of consecutive hot-spots in a single diagonal.

  (need not contain all the hot spots)

  How? FASTA gives each hot spot a positive score, & gives space between hot spots a negative score that decreases as distance increases

Key: pre-hash & use local expanding on diagonals.

FASTA (cont):

- Each diagonal run now specifies a possible aligned substring.

Step1:

- both matches & mismatches (gaps b/t consecutive (i,j)s picked)
- but no spaces (these move off diagonal)

Step2: Each of these alignments is examined to find best subalignment

- Score is now determined by using (ie) an amino acid substitution matrix.

Step3: Take the best of these (above some cutoff) & combine into larger, high scoring alignment with spaces.

(uses graph-based collapsing like) DeBruijn approach).

option 1 b

Step 4: Also computes alternative local alignment score (with output from step 2 as input).

This is dynamic programming, but not on full ~~sequences~~ database.

Finally: Given best scores to identify promising database sequences,

run full DP local alignment.

Recap:

- Trade-off is always DP or not.

These are pretty good.