Database Search

Lecture 5
Prof. Piotr Indyk

Previous lectures

- Lecture -3:
  - Global alignment in $O(mn)$
  - Dynamic programming
- Lecture -2:
  - Local alignment, variants, in $O(mn)$
- Lecture -1:
  - Exact string matching in $O(n)$
  - Hashing: number $\text{mod } q$

Quiz: What do these problems have in common?
Answer: they enable comparison of two sequences

The Big Picture

Database search

- Database search:
  - Database
  - Query: 
  - Output: sequences similar to query

What does “similar” mean?

- Simplest idea: just count the number of common amino-acids
  - E.g., RGRKW matches RGIKW at 4 positions, or $\text{idperc} = 80\%$
- Not all matches are created equal - scoring matrix
- In general, we should allow insertions and deletions as well

How to answer the query

- We could just scan the whole database
- But:
  - Query must be very fast
  - Most sequences will be completely unrelated to query
  - Individual alignment needs not be perfect. Can fine-tune
- Exploit nature of the problem
  - If you’re going to reject any match with $\text{idperc} < 90\%$, then why bother even looking at sequences which don’t have a fairly long stretch of matching a.a. in a row.
W-mer indexing

- Preprocessing: For every W-mer (e.g., \( W=3 \)) store every location in the database where it occurs. (can use hashing if \( W \) is large)
- Query: Generate W-mers and look them up in the database.
- Process the results
- Running time benefits:
  - For \( W=3 \), if the sequences are "random", then roughly one W-mer in \( 2^3 \) will match, i.e., one in a ten thousand
  - We hit only a small fraction of all sequences

AIKWQPRSTW….
IKMQRHIKW….
HDLFWHLWH….
……………………
RGIKW
……
IKW
IKZ
……
……
IKW
…...

BLAST

- Specific (and very efficient) implementation of the W-mer indexing idea
- How to generate W-mers from the query
- How to process the matches

Blast Algorithm Overview

- Receive query
  - Split query into overlapping words of length \( W \)
  - Find neighborhood words for each word until threshold \( T \)
  - Look into the table where these neighbor words occur: seeds
  - Extend seeds until score drops off under \( X \)
  - Evaluate statistical significance of score
  - Report scores and alignments

Extending the seeds

- Extend until the cumulative score drops

Length and Percent Identity
Why this works. I.e., what do we miss?

- In the worst case
  - W-mer: \( W=3 \)
  - No neighborhoods

Pigeonhole principle

- Pigeonhole principle
  - If you have 2 pigeons and 3 holes, there must be at least one hole with no pigeon

Pigeonhole and W-mers

- Pigeonholing mis-matches
  - Two sequences, each 9 amino-acids, with 7 identities
  - There is a stretch of 3 amino-acids perfectly conserved
- In general:
  - Sequence length: \( n \)
  - Identities: \( t \)
  - Can use W-mers for \( W = \left\lfloor \frac{n}{n-t+1} \right\rfloor \)

True alignments: Looking for K-mers

Personal experiment run in 2000.

- 850Kb region of human, and mouse 450Kb ortholog
- Blasted every piece of mouse against human (6,50)
- Identify slope of best fit line

Extensions

- Ideas beyond W-mer indexing?
  - Faster
  - Better sensitivity (less false negatives)
**Extensions: Filtering**
- Low complexity regions can cause spurious hits
  - Filter out low complexity in your query
  - Filter most over-represented items in your database

**Extensions: Two-hit blast**
- Improves sensitivity for any speed
  - Two smaller W-mers are more likely than one longer one
  - Therefore it’s a more sensitive searching method to look for two hits instead of one, with the same speed.
- Improves speed for any sensitivity
  - No need to extend a lot of the W-mers, when isolated

**Extensions: beyond W-mers**
- W-mers (without neighborhoods):
  - RGIKW → RG, GI, IK, BW
- No reason to use only consecutive symbols
- Instead, we could use combs, e.g.,
  - RGIKW → R*IK*, RG**W, ...
- Indexing same as for W-mers:
  - For each comb, store the list of positions in the database where it occurs
  - Perform lookups to answer the query
- How to choose the combs?
  - Choose the positions of * at random
  - Analyze false positives and false negatives

**Combs and Random Projections**
- Assume we select k positions, which do not contain *, at random with replacement
- What is the probability of a false negative?
  - At most: 1-idperck
  - In our case: 1-(7/9)k = 0.63...
- What is we repeat the process l times, independently?
  - Miss prob. = 0.63l
  - For l=5, it is less than 10%

**Suffix trees**
- Great tool for text processing
  - E.g., searching for exact occurrence of a pattern
- Suffix tree for: xabxac

**Suffix tree definition**
- Definition: Suffix tree ST for text T[1..n]
  - Rooted, directed tree T, n leaves, numbered 1..n
  - Text labels on the edges
  - Path to leaf i spells out the suffix S[i..], by concatenating edge labels
  - Common prefixes share common paths, diverge to form internal nodes
Properties of suffix trees

- How much space do we need to represent a suffix tree of $T[1..n]$?
- Only $O(n)$
  - At most $O(n)$ edges
  - Each edge label can be represented as $T[i..j]$

Exact string matching with suffix trees

- Given the suffix tree for text $T$
- Search for pattern $P[1..m]$
  - For every character in $P$, traverse the appropriate path of the tree, reading one character each time
  - If $P$ is not found in a path, $P$ does not occur in $T$
  - If $P$ is found in its entirety, then all occurrences of $P$ in $T$ are exactly the children of that node
    - Every child corresponds to exactly one occurrence
    - Simply list each of the leaf indices
- Time: $O(m)$

Suffix Tree Construction

- Running time: $O(n^2)$
- Can be improved to $O(n)$

Today

- Search among many database sequences
  - W-mer indexing
  - BLAST
  - Combs and random projections
  - Suffix trees