Combinatorial Pattern Matching

CS 466
Saurabh Sinha
Genomic Repeats

• Example of repeats:
  – ATGGTCTAGGTCCTAGTGGTC

• Motivation to find them:
  – Evolutionary annotation
  – Diseases associated with repeats
Genomic Repeats

• The problem is often more difficult:
  – ATGGTCTAGGACCTAGTGTTC
• Motivation to find them:
  – Evolutionary annotation
  – Diseases associated with repeats
$\ell$-mer Repeats

- Long repeats are difficult to find
- Short repeats are easy to find (e.g., hashing)
- Simple approach to finding long repeats:
  - Find exact repeats of short $\ell$-mers ($\ell$ is usually 10 to 13)
  - Use $\ell$-mer repeats to potentially extend into longer, maximal repeats
\( \ell \)-mer Repeats (cont’d)

- There are typically many locations where an \( \ell \)-mer is repeated:

\[
\text{GCTTTACAGATTTCAGTCTTACAGATGGT}
\]

- The 4-mer \textcolor{red}{\text{TTCAC}} starts at locations 3 and 17
Extending \( \omega \)-mer Repeats

GCTTACAGATTCCAGTCTTTACAGATGGT

- Extend these 4-mer matches:

GCTTTACAGATTCCAGTCTTTACAGATGGT

- Maximal repeat: CTTACAGAT
Maximal Repeats

• To find maximal repeats in this way, we need ALL start locations of all \( \ell \)-mers in the genome

• Hashing lets us find repeats quickly in this manner
Hashing: Maximal Repeats

• To find repeats in a genome:
  – For all ℓ-mers in the genome, note the start position and the sequence
  – Generate a hash table index for each unique ℓ-mer sequence
  – In each index of the hash table, store all genome start locations of the ℓ-mer which generated that index
  – Extend ℓ-mer repeats to maximal repeats
Pattern Matching

• What if, instead of finding repeats in a genome, we want to find all sequences in a database that contain a given pattern?

• Why? There may exist a library of known “repeat elements” (strings that tend to occur as repeats); we may scan for each such repeat element rather than finding them \textit{ab initio}

• This leads us to a different problem, the \textit{Pattern Matching Problem}
Pattern Matching Problem

• **Goal**: Find all occurrences of a pattern in a text

• **Input**: Pattern $p = p_1 \ldots p_n$ and text $t = t_1 \ldots t_m$

• **Output**: All positions $1 \leq i \leq (m - n + 1)$ such that the $n$-letter substring of $t$ starting at $i$ matches $p$

• **Motivation**: Searching database for a known pattern
Exact Pattern Matching: Running Time

• Naïve runtime: $O(nm)$

• On average, it’s more like $O(m)$
  – Why?

• Can solve problem in $O(m)$ time?
  – Yes, we’ll see how (later)
Generalization of problem: Multiple Pattern Matching Problem

- **Goal**: Given a set of patterns and a text, find all occurrences of any of patterns in text

- **Input**: $k$ patterns $p^1, \ldots, p^k$, and text $t = t_1 \ldots t_m$

- **Output**: Positions $1 \leq i \leq m$ where substring of $t$ starting at $i$ matches $p_j$ for $1 \leq j \leq k$

- **Motivation**: Searching database for known multiple patterns

- **Solution**: $k$ “pattern matching problems”: $O(kmn)$

- **Solution**: Using “Keyword trees” => $O(kn+m)$ where $n$ is maximum length of $p_i$
Keyword Trees: Example

• **Keyword tree:**
  – Apple
Keyword Trees: Example (cont’d)

• **Keyword tree:**
  – Apple
  – Apropos
Keyword Trees: Example
(cont’d)

• **Keyword tree:**
  – Apple
  – Apropos
  – Banana
Keyword Trees: Example
(cont’d)

• **Keyword tree:**
  – Apple
  – Apropos
  – Banana
  – Bandana
Keyword Trees: Example (cont’d)

- **Keyword tree:**
  - Apple
  - Apropos
  - Banana
  - Bandana
  - Orange
Keyword Trees: Properties

- Stores a set of keywords in a rooted labeled tree
- Each edge labeled with a letter from an alphabet
- Any two edges coming out of the same vertex have distinct labels
- Every keyword stored can be spelled on a path from root to some leaf
Multiple Pattern Matching: Keyword Tree Approach

- Build keyword tree in $O(kn)$ time; $kn$ is total length of all patterns

- Start “threading” at each position in text; at most $n$ steps tell us if there is a match here to any $p^i$

- $O(kn + nm)$

- Aho-Corasick algorithm: $O(kn + m)$
Aho-Corasick algorithm

A keyword tree for $\mathcal{P} = \{\text{he, she, his, hers}\}$:
“Fail” edges in keyword tree

Dashed edge out of internal node if matching edge not found
“Fail” edges in keyword tree

• If currently at node q representing word L(q), find the longest proper suffix of L(q) that is a prefix of some pattern, and go to the node representing that prefix

• Example: node q = 5 L(q) = she; longest proper suffix that is a prefix of some pattern: “he”. Dashed edge to node q’=2
Automaton

- Transition among the different nodes by following edges depending on next character seen ("c")
- If outgoing edge with label "c", follow it
- If no such edge, and are at root, stay
- If no such edge, and at non-root, follow dashes edge ("fail" transition); DO NOT CONSUME THE CHARACTER ("c")

Example: search text “ushers” with the automaton
Aho-Corasick algorithm

- $O(kn)$ to build the automaton
- $O(m)$ to search a text of length $m$
- Key insight:
  - For every character “consumed”, we move at most one level deeper (away from root) in the tree. Therefore total number of such “away from root” moves is $\leq m$
  - Each fail transition moves us at least one level closer to root. Therefore total number of such “towards root” moves is $\leq m$ (you can't climb up more than you climbed down)
Approximate vs. Exact Pattern Matching

• So far we’ve seen an exact pattern matching algorithm

• Usually, because of mutations, it makes much more biological sense to find approximate pattern matches
Heuristic Similarity Searches

• Genomes are huge: Dynamic programming-based local alignment algorithms are one way to find approximate repeats, but too slow

• Alignment of two sequences usually has short identical or highly similar fragments

• Many heuristic methods (i.e., FASTA) are based on the same idea of *filtration*: Find short exact matches, and use them as seeds for potential match extension
Query Matching Problem

• **Goal**: Find all substrings of the query that approximately match the text
• **Input**: Query $q = q_1 \ldots q_w$, 
  text $t = t_1 \ldots t_m$, 
  $n$ (length of matching substrings), 
  $k$ (maximum number of mismatches)
• **Output**: All pairs of positions $(i, j)$ such that the 
  $n$-letter substring of $q$ starting at $i$ 
  approximately matches the 
  $n$-letter substring of $t$ starting at $j$, 
  with at most $k$ mismatches
Query Matching: Main Idea

• Approximately matching strings share some perfectly matching substrings.

• Instead of searching for approximately matching strings (difficult) search for perfectly matching substrings (easy).
Filtration in Query Matching

• We want all $n$-matches between a query and a text with up to $k$ mismatches

• **Potential match detection**: find all matches of $l$-tuples in query and text for some small $l$

• **Potential match verification**: Verify each potential match by extending it to the left and right, until $(k + 1)$ mismatches are found
Filtration: Match Detection

- If $x_1...x_n$ and $y_1...y_n$ match with at most $k$ mismatches, they must share an $\ell$-tuple that is perfectly matched, with $\ell = \lceil n/(k + 1) \rceil$

- Break string of length $n$ into $k+1$ parts, each of length $\lceil n/(k + 1) \rceil$
  - $k$ mismatches can affect at most $k$ of these $k+1$ parts
  - At least one of these $k+1$ parts is perfectly matched
Filtration: Match Verification

- For each $\ell$-match we find, try to extend the match further to see if it is substantial.

Extend perfect match of length $\ell$ until we find an approximate match of length $n$ with $k$ mismatches.