CS481: Bioinformatics Algorithms

Can Alkan
EA509
calkan@cs.bilkent.edu.tr

http://www.cs.bilkent.edu.tr/~calkan/teaching/cs481/
COMBINATORIAL PATTERN MATCHING
Genomic Repeats

- Example of repeats:
  - ATGGTCTAGGTCCTAGTGGTC

- Motivation to find them:
  - Genomic rearrangements are often associated with repeats
  - Trace evolutionary secrets
  - Many tumors are characterized by an explosion of repeats
Genomic Repeats

- The problem is often more difficult:
  - ATGGTCTAGGACCTAGTGGTTC

- Motivation to find them:
  - Genomic rearrangements are often associated with repeats
  - Trace evolutionary secrets
  - Many tumors are characterized by an explosion of 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
There are typically many locations where an \( \ell \)-mer is repeated:

GCT\textcolor{red}{TTAC}CATTCAGTCT\textcolor{red}{TTAC}CAGATGGT

The 4-mer \textcolor{red}{TTAC} starts at locations 3 and 17
Extending ℓ-mer Repeats

GC TTAC AGAT TTCAGTGCT TTAC AGAT GGT

- Extend these 4-mer matches:

GC TTAC AGAT TTCAGTGCT TTAC AGAT GGT

- Maximal repeat: TTAC AGAT
Maximal Repeats

- To find maximal repeats in this way, we need ALL start locations of all ℓ-mers in the genome.

- **Hashing** lets us find repeats quickly in this manner.
Hashing DNA sequences

- Each ℓ-mer can be translated into a binary string (A, T, C, G can be represented as 00, 01, 10, 11)
- After assigning a unique integer per ℓ-mer it is easy to get all start locations of each ℓ-mer in a genome
Hashing: Maximal Repeats

To find repeats in a genome:
- For all $\ell$-mers in the genome, note the start position and the sequence
- Generate a hash table index for each unique $\ell$-mer sequence
- In each index of the hash table, store all genome start locations of the $\ell$-mer which generated that index
- Extend $\ell$-mer repeats to maximal repeats
Hashing: Collisions

- Dealing with collisions:
  - “Chain” all start locations of $\ell$-mers (linked list)
Hashing: Summary

- When finding genomic repeats from ℓ-mers:
  - Generate a hash table index for each ℓ-mer sequence
  - In each index, 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?

- This leads us to a different problem, the *Pattern Matching Problem*
Pattern Matching Problem

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

- **Input**: Pattern $p = p_1...p_n$ and text $t = t_1...t_m$

- **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: A Brute-Force Algorithm

**PatternMatching**\((p, t)\)

1. \(m \leftarrow \text{length of pattern } p\)
2. \(n \leftarrow \text{length of text } t\)
3. for \(i \leftarrow 1 \text{ to } (n - m + 1)\)
4. if \(t_i \ldots t_{i+m-1} = p\)
5. output \(i\)
Exact Pattern Matching: An Example

- **PatternMatching** algorithm for:
  - Pattern **GCAT**
  - Text **CGCATC**
**Exact Pattern Matching: Running Time**

- *PatternMatching* runtime: $O(nm)$

- Better solution: **suffix trees**
  - Can solve problem in $O(n)$ time
  - Conceptually related to **keyword trees** (=trie)
    - Multiple T, single P; or
    - Single T, multiple P
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
Multiple Pattern Matching: Straightforward Approach

- Can solve as \( k \) “Pattern Matching Problems”
  - Runtime:
    
    \[ O(kmn) \]
    
    using the \textit{PatternMatching} algorithm \( k \) times
  - \( m \) - length of the text
  - \( n \) - average length of the pattern
Multiple Pattern Matching: Keyword Tree Approach

- Or, we could use keyword trees:
  - Build keyword tree in $O(N)$ time; $N$ is total length of all patterns
  - With naive threading: $O(N + nm)$
  - Aho-Corasick algorithm: $O(N + m)$
Keyword Trees: Example

- **Keyword tree:**
  - Apple

Also known as “trie”
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
Keyword Trees: Threading (cont’d)

- Thread “appeal”
  - appeal
Keyword Trees: Threading (cont’d)

- Thread “appeal”
  - appeal
Keyword Trees: Threading (cont’d)

- Thread “appeal”
  - appeal
Keyword Trees: Threading (cont’d)

- Thread “appeal”
  - appeal
Keyword Trees: Threading (cont’d)

- Thread “apple”
  - apple
Keyword Trees: Threading (cont’d)

- Thread “apple”
  - apple
Keyword Trees: Threading (cont’d)

- Thread “apple”
  - apple
Keyword Trees: Threading  (cont’d)

- Thread “apple”
  - apple
Keyword Trees: Threading (cont’d)

- Thread “apple”
  - apple
Keyword Trees: Threading

- To match patterns in a text using a keyword tree:
  - Build keyword tree of patterns
  - “Thread” the text through the keyword tree

```
t = “mr and mrs dursley of number 4 privet drive were proud to say that they were perfectly normal thank you very much”
```
Keyword Trees: Threading (cont’d)

- Threading is “complete” when we reach a leaf in the keyword tree.

- When threading is “complete,” we’ve found a pattern in the text.

Problem: High memory requirement when N is large
Suffix Trees = Collapsed Keyword Trees

- Similar to keyword trees, except edges that form paths are collapsed
  - Each edge is labeled with a *substring* of a text
  - All internal edges have at least two outgoing edges
  - Leaves labeled by the index of the pattern.
Suffix Tree of a Text

- Suffix trees of a text is constructed for all its suffixes

```
ATCATG
TCATG
CATG
ATG
TG
G
```
Suffix Tree of a Text

- Suffix trees of a text is constructed for all its suffixes

How much time does it take?
Suffix Tree of a Text

- Suffix trees of a text is constructed for all its suffixes

ATCATG
TCATG
CATG
ATG
TG
G

Time is linear in the total size of all suffixes, i.e., it is quadratic in the length of the text
Let $s=abab$, a suffix tree of $s$ is a compressed trie of all suffixes of $s=abab$.

\{ 
  $ 
  \text{b}$ 
  $\text{ab}$ 
  $\text{bab}$ 
  $\text{abab}$ 
\}
Trivial algorithm to build a Suffix tree

Put the largest suffix in

Put the suffix $bab$ in
Put the suffix $ab$ in
Put the suffix b$ in
Put the suffix $ in
We will also label each leaf with the starting point of the corresponding suffix.

Trivial algorithm: $O(n^2)$ time
Suffix Trees: Advantages

- Suffix trees of a text is constructed for all its suffixes
- Suffix trees build faster than keyword trees
Use of Suffix Trees

- Suffix trees hold all suffixes of a text
  - i.e., ATCGC: ATCGC, TCGC, CGC, GC, C
  - Builds in $O(m)$ time for text of length $m$
- To find any pattern of length $n$ in a text:
  - Build suffix tree for text
  - Thread the pattern through the suffix tree
  - Can find pattern in text in $O(n)$ time!
- $O(n + m)$ time for “Pattern Matching Problem”
  - Build suffix tree for T and look up P
Pattern Matching with Suffix Trees

**SuffixTreePatternMatching**(p, t)
1. Build *suffix tree* for text t
2. Thread pattern p through *suffix tree*
3. if threading is complete
   4. output positions of all p-matching leaves in the tree
4. else
   5. output “Pattern does not appear in text”
Suffix Trees: Example

Figure 9.6  Threading the pattern ATG through the suffix tree for the text ATGCATA-CATGG. The suffixes ATGCATA and ATGG both match, as noted by the gray vertices in the tree (the p-matching leaves). Each p-matching leaf corresponds to a position in the text where p occurs.
Generalized suffix tree

Given a set of strings \( S \) a generalized suffix tree of \( S \) is a compressed trie of all suffixes of \( s \in S \).

To make these suffixes prefix-free we add a special char, say \( \$ \), at the end of \( s \).

To associate each suffix with a unique string in \( S \) add a different special char to each \( s \).
Generalized suffix tree (Example)

Let $s_1=abab$ and $s_2=aab$ here is a generalized suffix tree for $s_1$ and $s_2$

$$\{\$
\begin{align*}
&\text{a} & \text{b} & \text{a} & \text{b} \\
&\text{b} & \text{a} & \text{b} & \text{a} \\
&\text{ab} & \text{b} & \text{a} & \text{a} \\
&\text{bab} & \text{b} & \text{a} & \text{b} \\
&\text{abab} & \text{a} & \text{b} & \text{b} \\
&\text{babab} & \text{b} & \text{a} & \text{b} \\
\end{align*}
\}$$

Matching a pattern against a database of strings
Longest common substring of two strings

Every node with a leaf descendant from string $s_1$ and a leaf descendant from string $s_2$ represents a common substring.

Find such node with largest “string depth”
Multiple Pattern Matching: Summary

- Keyword and suffix trees are used to find patterns in a text
  - **Keyword trees:**
    - Build keyword tree of patterns, and *thread text* through it
  - **Suffix trees:**
    - Build suffix tree of text, and *thread patterns* through it
Slides from Charles Yan

AHO-CORASICK
Search in keyword trees

- Naïve threading in keyword trees do not *remember* the partial matches
- \( P = \{ \text{apple, propos} \} \)
- \( T = \text{appappapp} \)
- When threading
  - *app* is a partial match
  - But naïve threading will go back to the root and re-thread *app*
- Define *failure links*
v: a node in keyword tree K
L(v): the label on v, that is, the concatenation of characters on the path from the root to v.
lp(v): the length of the longest proper suffix of string L(v) that is a prefix of some pattern in P. Let this substring be $\alpha$.

Lemma. There is a unique node in the keyword tree that is labeled by string $\alpha$. Let this node be $n_v$. Note that $n_v$ can be the root.

The ordered pair $(v, n_v)$ is called a failure link.
Failure Link

\[ P = \{ \text{potato, tattoo, theater, other} \} \]
Failure Link

Failure link computation is $O(n)$
Failure Link

\[ i = 3 \]

\[ k = 8 \]
Failure Link

\[ i = k - l_p(w) = 8 - 3 = 5 \]

\[ k = 8 \]
Failure Link

How to construct failure links for a keyword tree in a linear time?

Let $d$ be the distance of a node $(v)$ from the root $r$.

When $d \leq 1$, i.e., $v$ is the root or $v$ is one character away from $r$, then $n_v = r$.

Suppose $n_v$ has been computed for every node $(v)$ with $d \leq k$, we are going to compute $n_v$ for every node with $d = k + 1$.

$v'$: parent of $v$, then $v'$ is $k$ characters from $r$, that is $d = k$.

Thus the failure link for $v'$ ($n_{v'}$) has been computed.

$x$: the character on edge $(v', v)$
(1) If there is an edge \((n_v', w)\) out of \(n_v'\), labeled with \(x\), then \(n_v = w\).
Failure Link
(2) If such an edge does not exist, examine \( n_{v'} \) to see if there is an edge out of it labeled with \( x \). Continue until the root.
(2) If such an edge does not exist, examine $n_{nv'}$ to see if there is an edge out of it labeled with $x$. Continue until the root.
Failure Link
Failure Link
Output: calculate $n_v$ for $v$

Algorithm $n_v$

$v'$ is the parent of $v$ in $K$

$x$ is the character on edge $(v', v)$

$w = n_v$

while there is no edge out of $w$ labeled with $x$ and $w \neq r$

\[
    w = n_w
\]

If there is an edge $(w, w')$ out of $w$ labeled $x$ then

\[
    n_v = w'
\]

else $n_v = r$
Aho-Corasick Algorithm

Input: Pattern set P and text T
Output: all occurrences in T any pattern from P
Algorithm Aho-Corasick

\( l = 1; \)
\( c = 1; \)
\( w = \text{root of } K \)

Repeat
  while there is an edge \((w, w')\) labeled with \( T(c) \)
    if \( w' \) is numbered by a pattern \( i \) then
      report that \( p_i \) occurs in \( T \) starting at \( l; \)
      \( w = w'; \) \( c++; \)
      \( w = n_w \) and \( l = c - l_p(w); \)
Until \( c > m \)
Slides from Tolga Can

**SUFFIX ARRAYS**
Suffix arrays

- Suffix arrays were introduced by Manber and Myers in 1993
- More space efficient than suffix trees
- A suffix array for a string x of length $m$ is an array of size $m$ that specifies the lexicographic ordering of the suffixes of x.
## Suffix arrays

Example of a suffix array for acaaacatat$

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>aaacatat$</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>aacatat$</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>acaaacatat$</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>acatat$</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>atat$</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>at$</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>caaacatat$</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>catat$</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>tat$</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>t$</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>$</td>
<td>11</td>
</tr>
</tbody>
</table>
Suffix array construction

- Naive in place construction
  - Similar to insertion sort
  - Insert all the suffixes into the array one by one making sure that the new inserted suffix is in its correct place
  - Running time complexity:
    - $O(m^2)$ where $m$ is the length of the string

- Manber and Myers give a $O(m \log m)$ construction.
Suffix arrays

- O(n) space where n is the size of the database string
- Space efficient. However, there’s an increase in query time
- Lookup query
  - Based on binary search
  - O(m log n) time; m is the size of the query
  - Can reduce time to O(m + log n) using a more efficient implementation
Searching for a pattern in Suffix Arrays

def find(Pattern P in SuffixArray A):
    i = 0
    lo = 0, hi = length(A)
    for 0<=i<length(P):
        Binary search for x,y
        lo<=x<=j<y<=hi
        lo = x, hi = y
    return {A[lo], A[lo+1], ..., A[hi-1]}
Search example

Search *is* in *mississippi*$

Examine the pattern letter by letter, reducing the range of occurrence each time.

First letter *i*: occurs in indices from 0 to 3

So, pattern should be between these indices.

Second letter *s*: occurs in indices from 2 to 3

Done.

Output: *issippi*$ and *ississippi*$
Suffix Arrays

- They can be built very fast.
- They can answer queries very fast:
  - How many times does ATG appear?
- Disadvantages:
  - Can’t do approximate matching
    - Except with some heuristics we will cover later
  - Hard to insert new stuff (need to rebuild the array) dynamically.