CS481: Bioinformatics Algorithms

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

http://www.cs.bilkent.edu.tr/~calkan/teaching/cs481/
EXACT STRING MATCHING
Fingerprint idea

- Assume:
  - We can compute a fingerprint $f(P)$ of $P$ in $O(m)$ time.
  - If $f(P) \neq f(T[s..s+m-1])$, then $P \neq T[s..s+m-1]$
  - We can compare fingerprints in $O(1)$
  - We can compute $f' = f(T[s+1..s+m])$ from $f(T[s..s+m-1])$, in $O(1)$
Algorithm with Fingerprints

- Let the alphabet $\Sigma = \{0,1,2,3,4,5,6,7,8,9\}$
- Let fingerprint to be just a decimal number, i.e., $f(\text{“1045”}) = 1\times10^3 + 0\times10^2 + 4\times10^1 + 5 = 1045$

**Fingerprint-Search(T,P)**

01 $fp \leftarrow \text{compute } f(P)$
02 $f \leftarrow \text{compute } f(T[0..m-1])$
03 for $s \leftarrow 0$ to $n-m$ do
04 \hspace{1em} if $fp = f$ return $s$
05 \hspace{1em} $f \leftarrow (f - T[s]\times10^{m-1})\times10 + T[s+m]$
06 return $-1$

- Running time $2O(m) + O(n-m) = O(n)$
Using a Hash Function

- **Problem:**
  - we cannot assume we can do arithmetics with m-digits-long numbers in O(1) time

- **Solution:** Use a hash function \( h = f \mod q \)
  - For example, if \( q = 7 \), \( h(“52”) = 52 \mod 7 = 3 \)
  - \( h(S1) \neq h(S2) \Rightarrow S1 \neq S2 \)
  - But \( h(S1) = h(S2) \) does not imply \( S1 = S2 \)
    - For example, if \( q = 7 \), \( h(“73”) = 3 \), but “73” ≠ “52”

- **Basic “mod q” arithmetics:**
  - \( (a+b) \mod q = (a \mod q + b \mod q) \mod q \)
  - \( (a*b) \mod q = (a \mod q)*(b \mod q) \mod q \)
Preprocessing and Stepping

- **Preprocessing:**
  - \( fp = P[m-1] + 10*(P[m-2] + 10*(P[m-3]+ \ldots + 10*(P[1] + 10*P[0])\ldots)) \mod q \)
  - In the same way compute \( ft \) from \( T[0..m-1] \)
  - Example: \( P = "2531" \), \( q = 7 \), \( fp = ? \)

- **Stepping:**
  - \( ft = (ft – T[s]*10^{m-1} \mod q)*10 + T[s+m]) \mod q \)
  - \( 10^{m-1} \mod q \) can be computed once in the preprocessing
  - Example: Let \( T[\ldots] = "5319" \), \( q = 7 \), what is the corresponding \( ft \)?
Stepping

- $T = 25319446766\ldots$, $m = 4$, $q=7$
- $T_0 = “2531”$
  - $ft = 2531 \mod 7 = 4$
- $T_1 = “5319”$
  - $ft = ((ft – T[0]*(10^3 \mod 7)) * 10 + T[0+4]) \mod 7$
    - $= ((4 - (2*1000 \mod 7)) * 10 + T[4]) \mod 7$
    - $= ((4-(2*6))*10+6) \mod 7 = (-8*10+9) \mod 7$
    - $= -71 \mod 7 = 6$
  - $5319 \mod 7 = 6$
Rabin-Karp Algorithm

Rabin-Karp-Search(T,P)
01 q ← a prime larger than m
02 c ← $10^{m-1}$ mod q // run a loop multiplying by 10 mod q
03 fp ← 0; ft ← 0
04 for i ← 0 to m-1 // preprocessing
05 fp ← (10*fp + P[i]) mod q
06 ft ← (10*ft + T[i]) mod q
07 for s ← 0 to n - m // matching
08 if fp = ft then // run a loop to compare strings
09 if P[0..m-1] = T[s..s+m-1] return s
10 ft ← ((ft - T[s]*c)*10 + T[s+m]) mod q
11 return -1
Analysis

- If q is a prime, the hash function distributes m-digit strings evenly among the q values
  - Thus, only every q\textsuperscript{th} value of shift s will result in matching fingerprints (which will require comparing strings with O(m) comparisons)

- Expected running time (if q > m):
  - Preprocessing: O(m)
  - Outer loop: O(n-m)
  - All inner loops: \( \frac{n - m}{q} \)
  - Total time: O(n-m)

- Worst-case running time: O(nm)
Rabin-Karp in Practice

- If the alphabet has d characters, interpret characters as radix-d digits (replace 10 with d in the algorithm).
- Choosing prime q > m can be done with randomized algorithms in O(m), or q can be fixed to be the largest prime so that 10*q fits in a computer word.
Searching in n comparisons

- The goal: each character of the text is compared only once!
- Problem with the naïve algorithm:
  - Forgets what was learned from a partial match!
  - Examples:
    - T = “Tweedledee and Tweedledum” and P = “Tweedledum”
    - T = “pappappapppar” and P = “pappar”
Finite automaton search

<table>
<thead>
<tr>
<th>state</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>a</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>b</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>a</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>c</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>a</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>a</td>
</tr>
</tbody>
</table>

T[i] -- a b a b a b a c a b a

state $\phi(i)$ 0 1 2 3 4 5 4 5 6 7 2 3

Processing time takes $\Theta(n)$.

But have to first construct FA.

**Main Issue:** How to construct FA?
Need some Notation ...

\( \phi(w) = \) state FA ends up in after processing \( w \).

**Example:** \( \phi(abab) = 4 \).

\( \sigma(x) = \max\{k: P_k \text{ suf } x\} \). Called the **suffix function**.

**Examples:** Let \( P = ab \).

\[ \begin{align*}
\sigma(\varepsilon) &= 0 \\
\sigma(ccaca) &= 1 \\
\sigma(ccab) &= 2
\end{align*} \]

**Note:** If \( |P| = m \), then \( \sigma(x) = m \) indicates a match.

\( T: \quad a \ b \ a \ b \ b \ a \ b \ a \ c \ ... \)

States: \( 0 \quad 1 \ldots m \ldots m \ldots m \ldots \)

\[ \begin{array}{c}
\text{match} \\
\text{match}
\end{array} \]
FA Construction

**Given:** \( P[1..m] \) \ Let \( Q = \text{states} = \{0, 1, \ldots, m\} \).

Define transition function \( \delta \) as follows:

\[
\delta(q, a) = \sigma(P_q a) \quad \text{for each} \ q \ \text{and} \ a.
\]

**Example:** \( P = \text{ababaca} \)

\[
\delta(5, b) = \sigma(P_5 b) \\
= \sigma(\text{ababab}) \\
= 4
\]

**Intuition:** Encountering a ‘b’ in state 5 means the current substring doesn’t match. But, you know this substring ends with “abab” -- and this is the longest suffix that matches the beginning of \( P \). Thus, we go to state 4 and continue processing “abab...” .
\( P = \text{ababaca} \)

\[ m = 7; \quad Q = \{0, 1, 2, 3, 4, 5, 6, 7\} \]

Prefixes:
- a
- ab
- aba
- abab
- ababa
- ababac
- ababaca
P = ababaca

\[ \delta(1, a) = \sigma(P_1a) = \sigma(aa) = \sigma(a) = 1 \]

Prefixes:
- a
- ab
- aba
- abab
- ababa
- ababac
- ababaca
$P=\text{ababaca}$

$\delta(1, a) = \sigma(P_1a) = \sigma(aa) = \sigma(a) = 1$
$\delta(1, c) = \sigma(P_1c) = \sigma(ac) = 0$

Prefixes
a
ab
aba
abab
ababa
ababac
ababaca
$\text{P}=\text{ababaca}$

$\delta(2, \text{b}) = \sigma(P_2\text{b}) = \sigma(\text{aba}) = \sigma(\text{a}) = 1$

$\delta(2, \text{c}) = \sigma(P_2\text{c}) = \sigma(\text{abc}) = 0$

Prefixes:
- a
- ab
- aba
- abab
- ababa
- ababac
- ababaca
P = ababaca (fast forward & simplified)

Prefixes:
- a
- ab
- aba
- abab
- ababac
- ababaca

δ(5, a) = σ(P₅a) = σ(ababaa) = σ(a) = 1
δ(5, b) = σ(P₅b) = σ(ababab) = σ(abab) = 4
P=ababaca (final, simplified)
T = abababacaba

Prefixes
a
ab
aba
abab
ababa
ababac
ababaca
T = abababacaba

Prefixes
a
ab
aba
abab
ababa
ababac
ababaca
T = abababacaba

Prefixes
a
ab
aba
abab
ababa
ababac
ababaca
T = abababacaba

Prefixes:
- a
- ab
- aba
- abab
- ababa
- ababac
- ababaca
- ababacaba
Search

T = abababacaba

Prefixes
- a
- ab
- aba
- abab
- ababa
- ababac
- ababaca
- ababacaba

b,c

a

b

a

abab

aba

ababa

ababac

ababaca
Search

T = abababacaba

Prefixes
a
ab
aba
abab
ababa
ababac
ababaca
T = abababacaba

Prefixes
a
ab
aba
abab
ababa
ababac
ababaca
T = abababacaba

Prefixes
a
ab
aba
abab
ababa
ababac
ababaca
T = abababacaba

Accept state, we are done
Analysis of FA

- Searching: $O(n) \rightarrow \text{good}$
- Preprocessing: $O(m|\Sigma|) \rightarrow \text{bad}$
- Memory: $O(m|\Sigma|) \rightarrow \text{bad}$
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:

\[
\text{GCTTTACAGATTCAGTTACAGATGGT}
\]

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

GCTTTACAGATTTCAAGTCTTTACAGATGGT

- Extend these 4-mer matches:

GCTTTACAGATTTCAAGTCTTTACAGATGGT

- Maximal repeat: TTACAGAT
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 \( \ell \)-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 \( \ell \)-mer it is easy to get all start locations of each \( \ell \)-mer in a genome
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 $\ell$-mers:

- Generate a hash table index for each $\ell$-mer sequence
- In each index, store all genome start locations of the $\ell$-mer which generated that index
- Extend $\ell$-mer repeats to maximal repeats
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 $p = p_1 \ldots p_n$ 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: 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 \( i \leftarrow 1 \) 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: $O(nm)$
  - KMP or BM: $O(n+m)$
- Multiply by $k$ if looking for $k$ different patterns
- Better solution: suffix trees
  - Can solve problem in $O(n)$ time
  - Conceptually related to keyword trees
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 Diagram]
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
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: 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
  - Built from **text**, not patterns
  - 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
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.

Time is linear in the total size of all suffixes, i.e., it is quadratic in the length of the text.
Suffix tree (Example)

Let $s=abab$, a suffix tree of $s$ is a compressed trie of all suffixes of $s=abab$

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

Put the largest suffix in

Put the suffix $\text{bab}$ in
Put the suffix \texttt{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

ATCATG
TCATG
CATG
ATG
TG
G

**Keyword Tree**

**Suffix Tree**

- Quadratic
- Linear (Weiner suffix tree algorithm)
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 and lookup pattern
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
5     else
6       **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 ATGCATACATGG 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 $s_2$

{ 
  $\$
  b$
  ab$
  bab$
  abab$
}

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 maximal common substring and vice versa.

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