Overview

- **Pattern Matching**
  - Exhaustive Search
  - DFA-Based Search Algorithm
  - KMP Algorithm
- Multiple Pattern Matching
  - AC Algorithm
  - Weiner’s Algorithm
- Approximate Pattern Matching
Pattern Matching

- Definition
  - Given a text (string), finding all occurrences of a pattern (substring)
  - Given a DNA, RNA, or protein sequence, finding all occurrences of a specific repeat

- Examples
  - A T G G T C T A G G T C C T A G T G G T C

- Applications
  1. Homolog search in BLAST
  2. Sequence motif search
     - Repeats (substrings, patterns) often represent sequence motifs
     - Functional domains are often associated with repeats
     - Evolutionary path can be traced by repeats

Terminology

- Prefix
  - $S[1..j]$ is a prefix of a string $S[1..n]$ where $j \leq n$
  - $X$ is a prefix of $Y$ if $X \cdot Z = Y$ for some string $Z$

- Suffix
  - $S[i..n]$ is a suffix of a string $S[1..n]$ where $1 \leq i$
  - $X$ is a suffix of $Y$ if $Z \cdot X = Y$ for some string $Z$

- Substring
  - A string of consecutive letters from $S$
  - $S[i..j]$ is a substring of a string $S[1..n]$ where $1 \leq i$ and $j \leq n$
  - A substring of $S$ is a prefix of a suffix of $S$

- Empty String
  - $S[i..j]$ is an empty string where $i > j$
Properties

➢ Proper Prefix, Proper Suffix, Proper Substring
  ▪ The proper prefix, suffix, or substring of a string S is a prefix, suffix, or substring that is not the empty string nor S itself

➢ Main Properties
  ▪ Reflexivity (But, not for proper prefix, proper suffix, proper substring)
  ▪ Anti-symmetry (But, not for proper prefix, proper suffix, proper substring)
  ▪ Transitivity

➢ Other Properties
  ▪ If X is a suffix of Y, then X*Z is a suffix of Y*Z for some string Z
  ▪ If X is a suffix of Z, Y is a suffix of Z, and |X| ≤ |Y|, then X is a suffix of Y

Formulation of Pattern Matching Problem

➢ Goal
  ▪ Finding all occurrences of a substring (length-m) in a string (length-n)

➢ Input
  ▪ A substring \( P = p_1 \cdot p_2 \cdot ... \cdot p_m \) and a string \( T = t_1 \cdot t_2 \cdot ... \cdot t_n \)

➢ Output
  ▪ All positions \( 1 \leq i \leq (n-m+1) \) such that the substring of \( T \) starting at \( i \) matches \( P \)
Naïve Approach

- Algorithm
  - Exhaustive search

  \[
  \text{NaïveMatching}(T, P) \\
  n \leftarrow \text{length}(T) \\
  m \leftarrow \text{length}(P) \\
  \text{for } i \leftarrow 1 \text{ to } n - m + 1 \\
  \quad \text{if } P[1..m] = T[|..(i+m)] \\
  \quad \quad \text{then print } i \\
  \]

- Example
  - \( T = \text{CTGCATC} \)
  - \( P = \text{GCAT} \)

  \[
  \begin{array}{ccccccc}
  \text{CTGCATC} & \text{CTGCATC} & \text{CTGCATC} & \text{CTGCATC} & \text{CTGCATC} & \text{CTGCATC} & \text{CTGCATC} \\
  \text{GCAT} & \text{GCAT} & \text{GCAT} & \text{GCAT} & \text{GCAT} & \text{GCAT} & \text{GCAT} \\
  \end{array}
  \]

- Runtime ?

Pattern Matching Using DFA

- Pattern Matching by DFA
  1. Constructs an automaton for the substring (pattern) \( P \)
  2. Searches \( P \) by reading the string (text) \( T \) on the automaton

- Suffix Function
  - Suffix function \( \sigma(X) \) for \( P \): a mapping to the length of the longest prefix of \( P \) that is a suffix of \( X \)
  - e.g., \( P = \text{"abc"}, \sigma(\text{"cbaca"}) = ? \), \( \sigma(\text{"ccab"}) = ? \)
Constructing DFA

- Process
  - Given a substring (pattern) $P$ with length $m$
  - Makes the set of states $Q = \{0, 1, \ldots, m\}$, with the state 0 as $q_0$, and the state $m$ as the only accepting state
  - Defines the transition function $\delta$ as
    \[ \delta(q, a) = \sigma(P[1..q]a) \]

- Example
  - $P=\text{“ababaa”}$, $\Sigma=\{a, b\}$

- Runtime ?

DFA-Based Search (1)

- Process
  - Given an input string $T$ having the letters in $\Sigma$,
  - Starts at the state $q_0$
  - Reads the string $T$, character by character, changing state after each character read

- Pattern Matching
  - Automaton finds the substring $P$ from $T$ if it reaches an accepting state

- Example
  - $P=\text{“ababaa”}$, $\Sigma=\{a, b\}$
  - $T=\text{“aababaababaabababaa”}$
DFA-Based Search (2)

- Algorithm

```plaintext
AUTOMATAMATCHING(T, P, M)

n ← length(T)
m ← length(P)
q ← q₀
for i ← 1 to n
q ← δ(q, T[i])
if q ∈ A
 then print (i - m + 1)
```

- Runtime ?

- Total Runtime of DFA-Based Search ?

Pattern Shifting

- Backgrounds

```
  b a c b a b a b a c b a b  T
  s  a b a b a c a

  b a c b a b a b a b a b c b a b  T
  s'=s+2  a b a b a c a
```

- \( T[1..n] \), \( P[1..m] \)
- Given \( P[1..q] \) (where \( q ≤ m \)) matches \( T[(s+1)..(s+q)] \), what is the least shift \( s' \) (where \( s'>s \)) such that

\[
P[1..k] = T[(s'+1)..(s'+k)] \quad \text{where} \quad s'+k = s+q
\]
Prefix Function

- Prefix Function
  - Prefix function \( \pi(q) \) for \( P \): a mapping to the length of the longest prefix of \( P \) that is a proper suffix of \( P[1..q] \)

Example

<table>
<thead>
<tr>
<th>( i )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P[i] )</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>( \pi(i) )</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Knuth-Morris-Pratt (KMP) Algorithm (1)

- Algorithm

  ```
  KMP-MATCHING(T, P, \pi)
  n ← length(T)
  m ← length(P)
  q ← 0
  for \( i ← 1 \) to \( n \)
      while \( q > 0 \) and \( P[q+1] \neq T[i] \)
          \( q ← \pi[q] \)
      if \( P[q+1] = T[i] \)
          then \( q ← q + 1 \)
      if \( q = m \)
          then print \((i − m + 1) \) and \( q ← \pi[q] \)
  ```

- Runtime ?
Knuth-Morris-Pratt (KMP) Algorithm (2)

- Algorithm of Prefix Function

```plaintext
prefixFunction(P)
    m ← length(P)
    π[i] ← 0
    k ← 0
    for q ← 2 to m
        while k > 0 and P[k+1] ≠ P[q]
            k ← π[k]
        if P[k+1] = P[q]
            k ← k + 1
            π[q] ← k
    return π
```

- Runtime ?

- Total Runtime of KMP Algorithm ?

Overview

- Pattern Matching
  - Exhaustive Search
  - DFA-Based Search Algorithm
  - KMP Algorithm

- Multiple Pattern Matching
  - AC Algorithm
  - Weiner’s Algorithm

- Approximate Pattern Matching
Multiple Pattern Matching

- **Motivation**
  - Finding matches of multiple patterns from a text at the same time
  - Finding all occurrences of multiple patterns at the same time in a DNA or protein sequence improves efficiency for homolog search

- **Examples**
  - ATGGTCTAGGTCCTAGTG
  - \( P = \{ GGTC, CTAG, TGGT \} \)

Formulation of Multiple Pattern Matching Problem

- **Goal**
  - Finding all occurrences of any in a set of substrings (length-\( m \)) in a string (length-\( n \))

- **Input**
  - A set of \( k \) substrings \( P_1, P_2, \ldots, P_k \) and a string \( T = t_1 \cdot t_2 \cdot \ldots \cdot t_n \)

- **Output**
  - All positions \( 1 \leq i \leq n \) such that a substring of \( T \) starting at \( i \) matches \( P_j \) where \( 1 \leq j \leq k \)
Extension of Pattern Matching

➢ Extension of Naïve Approach
  ▪ Naïve string matching k times
  ▪ Runtime ?

➢ Extension of Other String Matching Algorithms
  ▪ KMP string matching k times
  ▪ Runtime ?

➢ Direction
  ▪ Advanced data structure
  ▪ Advanced algorithm

Prefix Tree (1)

➢ Prefix Tree
  ▪ Data structure to manage a set of substrings (patterns), \( P \)
  ▪ Each path from the root represents each pattern
  ▪ Also called "keyword tree" or "trie"

➢ Features
  ▪ Each edge is labeled with a character
  ▪ Any two or more edges to child nodes from a parent node have different labels
  ▪ Each node \( v \) is labeled as the concatenation of edge labels on the path from the root to \( v \) (the node label is denoted by \( L(v) \))
  ▪ For each \( P_i \in P \), there is a node \( v \) such that \( L(v)=P_i \)
  ▪ \( L(v) \) for any leaf node \( v \) equals some \( P_j \) where \( P_j \in P \)
Prefix Tree (2)

- Example
  - $P = \{ TG, ATG, TCA, TGAC \}$

- Runtime of Prefix Tree Construction?

Extension of Finite Automata (1)

- Multiple String Matching with Finite Automata (Aho-Corasick Algorithm)
  1. Constructs an automaton for the set of substrings (patterns), $P$
  2. Searches all substrings in the string (text) $T$ by the automaton

- Finite Automata, $M=(Q, q_0, A, \Sigma, \delta)$, on a Prefix Tree
  - $Q$: the set of nodes in the prefix tree
  - $q_0$: the root in the prefix tree
  - $A$: the nodes marked in the prefix tree
  - $\Sigma$: the set of all distinct characters in $P$
  - $\delta$: transition functions
    - goto functions ($g$)
    - failure functions ($f$)
Extension of Finite Automata (2)

- **Goto Function**
  - $g(q_i, a)$: a mapping to the state entered from the current state $q_i$ by matching the target character $a$
  - If the edge $(q_i, q_j)$ is labeled by $a$, and $q_i$ is a parent node of $q_j$ in the prefix tree, then $g(q_i, a) = q_j$
  - Otherwise, $g(q_i, a) = \emptyset$, except $g(q_0, a) = q_0$

- **Failure Function**
  - $f(q_i) = \pi(L(q_i))$: a mapping to the state of the longest prefix of some pattern in $P$, which is a proper suffix of $L(q_i)$

Example of DFA-like Structure

- **Example**
  - $P = \{ \text{TG}, \text{ATG}, \text{TCA}, \text{TGAC} \}$

![Diagram of DFA-like Structure]
Searching Multiple Patterns by DFA-like Structure

➢ Process
  ▪ Given an input string $T$ having the letters in $\Sigma$,
  ▪ Starts at the state $q_0$
  ▪ Reads the string $T$, character by character, changing state after each character read

➢ Multiple String Matching
  ▪ Automaton finds a substring $P_j$ in $P$ from $T$
    if it reaches the accepting state corresponding to $P_j$

➢ Example
  ▪ $P = \{ \text{TG, ATG, TCA, TGAC} \}$
  ▪ $T = \text{"ATCATGTGAC"}$

Aho-Corasick (AC) Algorithm (1)

➢ Algorithm

\[
\begin{align*}
\text{AC-MULTIPLE-MATCHING} &: (T, \{ P_1, P_2, \ldots, P_k \}, M) \\
n &\leftarrow \text{length}(T) \\
\text{for } j = 1 \text{ to } k \\
m_j &\leftarrow \text{length}(P_j) \\
q &\leftarrow q_0 \\
\text{for } i = 1 \text{ to } n \\
\text{while } g(q, T[i]) = \emptyset \\
q &\leftarrow f(q) \\
q &\leftarrow g(q, T[i]) \\
\text{if } q = a_j \in A \\
\text{then print } (i - m_j + 1)
\end{align*}
\]

➢ Runtime?
Constructing DFA-like Structure

- Process
  - Constructs the prefix tree for $P$
    - all nodes in the prefix tree $\rightarrow Q$
    - the root node $\rightarrow q_0$
  - Marks all accepting states for $A$
  - Makes goto function for each state
  - Makes failure function for each state as $f(q) = \pi(L(q))$

Aho-Corasick (AC) Algorithm (2)

- Algorithm of Failure Function

```
FAILUREFUNCTION(M, g)
Q ← empty queue
for $a \in \Sigma$
  if $g(q_0, a) = q \neq q_0$
    $f(q) ← \emptyset$ and enqueue($q, Q$)
  while $Q \neq \emptyset$
    $v ←$ dequeue($Q$)
    for $a \in \Sigma$
      if $g(v, a) = u \neq \emptyset$
        enqueue($u, Q$) and $v ← f(v)$
      while $g(v, a) = \emptyset$
        $v ← f(v)$
    $f(u) ← g(v, a)$
return $f$
```

- Runtime ?

- Total Runtime of AC Algorithm ?
Suffix Tree (1)

- **Suffix Tree**
  - Data structure to manage a string (text), \( T \)
  - Each path from the root represents each suffix of \( T \)
  - Also called “collapsed keyword tree”

- **Features**
  - Each edge is labeled with a string (a substring of \( T \))
  - All internal nodes have at least two outgoing edges
    - Similar to prefix trees, but edges that form a linear path are collapsed
  - Leaf nodes are labeled with the index of the pattern (starting position)

Suffix Tree (2)

- **Examples**
  - \( T = \) ATCATG
    - TCATG
    - CATG
    - ATG
    - TG
    - G

- **Runtime of Suffix Tree**
  - Construction?
    - Naïve approach

(a) Prefix tree        (b) Suffix tree
Constructing Suffix Tree (1)

- Weiner’s Algorithm
  - Linear-time suffix tree construction algorithm

- Substring Function
  - Substring function $\theta(i)$ for $T$: a mapping to the position and length of the substring of $T[i+1..n]$ that matches the longest prefix of $T[i..n]$

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T[i]$</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>A</td>
<td>T</td>
<td>G</td>
</tr>
<tr>
<td>$\theta(i)$</td>
<td>4/2</td>
<td>5/1</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
</tr>
</tbody>
</table>

Constructing Suffix Tree (2)

- Process
  - Read each character in $T$ backwards
  - Attach the suffix $T[i..n]$ to the nodes labeled with the position of $\theta(i)$
  - Converting the edge with the length of $\theta(i)$, and adding new branches
  - Example, $T$="ATCATG"
Multiple Matching with Suffix Tree

- **Process**
  - Build a suffix tree for $T$
  - Thread each pattern $P_i$ where $1 \leq i \leq k$ through the suffix tree
  - If threading is complete, output all labels of leaf nodes

- **Example of Threading**
  - $T = \text{"ATGCATA\text{CAT}\text{GG}"}$
  - $P_i = \text{"ATG"}$

- **Runtime**

Overview

- **Pattern Matching**
  - Exhaustive Search
  - DFA-Based Search Algorithm
  - KMP Algorithm

- **Multiple Pattern Matching**
  - AC Algorithm
  - Weiner’s Algorithm

- **Approximate Pattern Matching**
Exact Matching vs. Approximate Matching

- **Exact Matching**
  \[
  T = \text{agcctccgatcagttactcagatgtaactattcgatgcaccccctattacatctctacgatgtcataca} \\
  P = "cgatgt"
  \]
  \[
  T = \text{agcctccgatcagttactcagatgtaactattcgatgcaccccctattacatctctacgatgtcataca}
  \]

- **Approximate Matching (Inexact Matching)**
  \[
  T = \text{agcctccgatcagttactcagatgtaactattcgatgcaccccctattacatctctacgatgtcataca} \\
  \text{mutations}
  \]
  \[
  P = "cgatgt" \quad P = "cgatgt" \quad P = "cgatgt"
  \]

Formulation of Approximate Matching Problem

- **Goal**
  - Finding all approximate occurrences of a substring (length-\(m\)) in a string (length-\(n\))

- **Input**
  - A substring \(P = \rho_1, ..., \rho_m\), a string \(T = t_1, ..., t_n\), and the maximum number mismatches, \(k\)

- **Output**
  - All positions \(1 \leq i \leq (n-m+1)\) such that \(P\) and the substring of \(T\) starting at \(i\) have at most \(k\) mismatches
Naïve Approach

- Algorithm
  - Exhaustive search

  \begin{algorithm}
  \textbf{APPROXIMATEMATCHING}(T, P, k)
  \begin{align*}
  n & \leftarrow \text{length}(T) \\
  m & \leftarrow \text{length}(P) \\
  \text{for } i & \leftarrow 1 \text{ to } n - m + 1 \\
  \text{mismatch} & \leftarrow 0 \\
  \text{for } j & \leftarrow 1 \text{ to } m \\
  & \quad \text{if } T[i + j - 1] \neq P[j] \\
  & \quad \quad \text{then } \text{mismatch} \leftarrow \text{mismatch} + 1 \\
  & \quad \text{if } \text{mismatch} \leq k \\
  & \quad \quad \text{then print } i
  \end{align*}
  \end{algorithm}

- Runtime ?

Dynamic Programming

- Algorithm
  - Count mismatches, \( D(i, j) \), between \( P[i] \) and \( T[j] \)
  - Find all positions \( i \) such that the number of mismatches between \( P[1..m] \) and \( T[l..(l+m-1)] \) is less than or equal to \( k \)

  \[
  D(i, j) = \begin{cases} 
  D(i-1, j-1) + 0 & \text{if } P[i] = T[j] \\
  D(i-1, j-1) + 1 & \text{otherwise}
  \end{cases}
  \]

- Example
  - \( T = \text{"AGCCTTGAT"}, \ P = \text{"GCAT"}, \ k=2 \)

- Runtime ?
Questions?

- Lecture Slides are found on the Course Website, web.ecs.baylor.edu/faculty/cho/3360