Module 9: Tries and String Matching

Search for a string (pattern) in a large body of text

- \( T[0..n-1] \) – The text (or haystack) being searched within
- \( P[0..m-1] \) – The pattern (or needle) being searched for
- Strings over alphabet \( \Sigma \)
- Return the first \( i \) such that
  \[
  P[j] = T[i+j] \quad \text{for} \quad 0 \leq j \leq m-1
  \]
- This is the first occurrence of \( P \) in \( T \)
- If \( P \) does not occur in \( T \), return FAIL
- Applications:
  - Information Retrieval (text editors, search engines)
  - Bioinformatics
  - Data Mining

Example:
- \( T = \) “Where is he?”
- \( P_1 = \) “he”
- \( P_2 = \) “who”

Definitions:
- Substring \( T[i..j] \) \( 0 \leq i \leq j < n \): a string of length \( j - i + 1 \) which consists of characters \( T[i], \ldots, T[j] \) in order
- A prefix of \( T \):
  a substring \( T[0..i] \) of \( T \) for some \( 0 \leq i < n \)
- A suffix of \( T \):
  a substring \( T[i..n-1] \) of \( T \) for some \( 0 \leq i \leq n - 1 \)

General Idea of Algorithms

Pattern matching algorithms consist of guesses and checks:
- A guess is a position \( i \) such that \( P \) might start at \( T[i] \).
  Valid guesses (initially) are \( 0 \leq i \leq n - m \).
- A check of a guess is a single position \( j \) with \( 0 \leq j < m \) where we compare \( T[i+j] \) to \( P[j] \). We must perform \( m \) checks of a single correct guess, but may make (many) fewer checks of an incorrect guess.

We will diagram a single run of any pattern matching algorithm by a matrix of checks, where each row represents a single guess.
Brute-force Algorithm

Idea: Check every possible guess.

```plaintext
Brute-forcePM(T[0..n - 1], P[0..m - 1])
T: String of length n (text), P: String of length m (pattern)
1. for i ← 0 to n - m do
2. match ← true
3. j ← 0
4. while j < m and match do
5. if T[i + j] = P[j] then
6. j ← j + 1
7. else
8. match ← false
9. if match then
10. return i
11. return FAIL
```

Example

- Example: \( T = \text{abbbababbab} \), \( P = \text{abba} \)

```
  a b b b a b a b b a b
  a
  a
  a
  a
  a
  a
  a b b
  a
  a b b a
```

- What is the worst possible input?
  \( P = a^{m-1}b \), \( T = a^n \)
- Worst case performance \( \Theta((n - m + 1)m) \)
- \( m \leq n/2 \) \( \Rightarrow \Theta(mn) \)

Pattern Matching

More sophisticated algorithms
- Deterministic finite automata (DFA)
- KMP, Boyer-Moore and Rabin-Karp
- Do extra preprocessing on the pattern \( P \)
- We eliminate guesses based on completed matches and mismatches.

String matching with finite automata
There is a string-matching automaton for every pattern \( P \). It is constructed from the pattern in a preprocessing step before it can be used to search the text string.

Example: Automaton for the pattern \( P = \text{ababaca} \)
String matching with finite automata

Let $P$ the pattern to search, of length $m$. Then

- the states of the automaton are $0, \ldots, m$
- the transition function $\delta$ of the automaton is defined as follows, for a state $q$ and a character $c$ in $\Sigma$:
  \[
  \delta(q, c) = \ell(P[0..q-1]c),
  \]

where

- $P[0..q-1]c$ is the concatenation of $P[0..q-1]$ and $c$
- for a string $s$, $\ell(s) \in \{0, \ldots, m\}$ is the length of the longest prefix of $P$ that is also a suffix of $s$.

Graphically, this corresponds to

$$q \xrightarrow{c} \delta(q, c)$$

Idea of proof: the state after reading $T[i]$ is $\ell(T[0..i])$.

KMP Algorithm

- Knuth-Morris-Pratt algorithm (1977)
- Compares the pattern to the text in left-to-right
- Shifts the pattern more intelligently than the brute-force algorithm
- When a mismatch occurs, how much can we shift the pattern (reusing knowledge from previous matches)?

$$T = \begin{array}{cccccccc}
  a & b & c & d & c & a & b & c \\
  a & b & c & d & c & a & b & a \\
  \hline
  a & b & c & d & c & a
\end{array}$$

- KMP Answer: this depends on the largest prefix of $P[0..j]$ that is a suffix of $P[1..j]$
KMP Failure Array

Suppose we have a match up to position $T[i-1] = P[j-1]$, but not at the next position.

Define $F[j-1]$ as the index we will have to check in $P$, after we bring the pattern to its next possible position (previous example: $j = 6$, $F[5] = 2$).

This can be computed by trying all sliding positions until finding the first one matching the text (as in previous example). We can do better:

- any possible sliding position corresponds to a prefix of $P[0..j-1]$ that is also a strict suffix of it, a suffix of $P[1..j-1]$
- the next possible sliding position corresponds to the largest such prefix / suffix
- we let $F[j-1]$ be the length of this prefix / suffix.

Schematically:

<table>
<thead>
<tr>
<th>$j$</th>
<th>$P[1..j]$</th>
<th>$P$</th>
<th>$F[j]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>—</td>
<td>abacaba</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
<td>abacaba</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>ba</td>
<td>abacaba</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>bac</td>
<td>abacaba</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>baca</td>
<td>abacaba</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>bacab</td>
<td>abacaba</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>bacaba</td>
<td>abacaba</td>
<td>3</td>
</tr>
</tbody>
</table>

Haque, Irvine, Smith (SCS, UW)  
CS240 - Module 9  
Spring 2017  
13 / 43
Computing the Failure Array

failureArray(P)
P: String of length m (pattern)
1. F[0] ← 0
2. i ← 1
3. j ← 0
4. while i < m do
   5. if P[i] = P[j] then
      6. F[i] ← j + 1
      7. i ← i + 1
      8. j ← j + 1
   9. else if j > 0 then
      10. j ← F[j − 1]
   11. else
      12. F[i] ← 0
      13. i ← i + 1

Haque, Irvine, Smith (SCS, UW) CS240 - Module 9 Spring 2017 17 / 43

KMP Algorithm

KMP(T, P), to return the first match
T: String of length n (text), P: String of length m (pattern)
1. F ← failureArray(P)
2. i ← 0
3. j ← 0
4. while i < n do
   5. if T[i] = P[j] then
      6. if j = m − 1 then
         7. return i − j //match
      8. else
         9. i ← i + 1
        10. j ← j + 1
   11. else
      12. if j > 0 then
         13. j ← F[j − 1]
      14. else
         15. i ← i + 1
   16. return −1 // no match

Haque, Irvine, Smith (SCS, UW) CS240 - Module 9 Spring 2017 18 / 43

KMP: Example

P = abacaba
j 0 1 2 3 4 5 6
F[j] 0 0 1 0 1 2 3
T = abaxyabacabbaababacaba

Exercise: continue with T = abaxyabacabbaababacaba

Haque, Irvine, Smith (SCS, UW) CS240 - Module 9 Spring 2017 19 / 43

KMP: Analysis

failureArray
At each iteration of the while loop, at least one of the following happens:
1. i increases by one, or
2. the index i − j increases by at least one (F[j − 1] < j)
There are no more than 2m iterations of the while loop
Running time: Θ(m)

KMP
failureArray can be computed in Θ(m) time
At each iteration of the while loop, at least one of the following happens:
1. i increases by one, or
2. the index i − j increases by at least one (F[j − 1] < j)
There are no more than 2n iterations of the while loop
Running time: Θ(n)

Haque, Irvine, Smith (SCS, UW) CS240 - Module 9 Spring 2017 20 / 43
Boyer-Moore Algorithm

Based on three key ideas:
- **Reverse-order searching**: Compare $P$ with a subsequence of $T$ moving backwards
- **Bad character jumps**: When a mismatch occurs at $T[i] = c$
  - If $P$ contains $c$, we can shift $P$ to align the last occurrence of $c$ in $P$ with $T[i]$
  - Otherwise, we can shift $P$ to align $P[0]$ with $T[i + 1]$
- **Good suffix jumps**: If we have already matched a suffix of $P$, then get a mismatch, we can shift $P$ forward to align with the previous occurrence of that suffix (with a mismatch from the suffix we read). If none exists, look for the longest prefix of $P$ that is a suffix of what we read. Similar to failure array in KMP.
- Can skip large parts of $T$

**Bad character examples**

$P = a l d o$

$T = w h e r e i s w a l d o$

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>o</td>
<td></td>
<td></td>
<td>a l d o</td>
</tr>
</tbody>
</table>

6 comparisons (checks)

$P = m o o r e$

$T = b o y e r m o o r e$

|   | (r) | e | (m) | o o r e |

7 comparisons (checks)

**Good suffix examples**

$P = s e l l s$

$s h e i l a s e l l s s h e l l s$

$P = o d e t o f o o d$

$i l i k e f o o d f r o m m e x i c o$

<table>
<thead>
<tr>
<th></th>
<th>o f o o d</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(e)</td>
<td>(o)</td>
</tr>
</tbody>
</table>

Good suffix moves further than bad character for 2nd guess.
Bad character moves further than good suffix for 3rd guess.
This is out of range, so **pattern not found**.

**Last-Occurrence Function**

- **Preprocess** the pattern $P$ and the alphabet $\Sigma$
- Build the **last-occurrence function** $L$ mapping $\Sigma$ to integers
  - $L(c)$ is defined as
    - the largest index $i$ such that $P[i] = c$ or
    - $-1$ if no such index exists
- Example: $\Sigma = \{a, b, c, d\}, P = abacab$

<table>
<thead>
<tr>
<th>c</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>-1</td>
</tr>
</tbody>
</table>

The last-occurrence function can be computed in time $O(m + |\Sigma|)$
In practice, $L$ is stored in a size-$|\Sigma|$ array.
Good Suffix array

- Again, we preprocess $P$ to build a table.
- **Suffix skip array** $S$ of size $m$: for $0 \leq i < m$, $S[i]$ is the largest index $j$ such that $P[i+1..m-1] = P[j+1..m-1-i]$ and $P[j] \neq P[i]$.
- **Note**: in this calculation, any negative indices are considered to make the given condition true (these correspond to letters that we might not have checked yet).
- Similar to KMP failure array, with an extra condition.
- Computed similarly to KMP failure array in $\Theta(m)$ time.

Example: $P = \text{bonobobo}$

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</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>b</td>
<td>o</td>
<td>n</td>
<td>o</td>
<td>b</td>
<td>o</td>
<td>b</td>
<td>o</td>
</tr>
<tr>
<td>$S[i]$</td>
<td>-6</td>
<td>-5</td>
<td>-4</td>
<td>-3</td>
<td>2</td>
<td>-1</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

Computed similarly to KMP failure array in $\Theta(m)$ time.

Boyé-Moore Algorithm

```
boyer-moore(T,P)
1. L ← last occurrence array computed from P
2. S ← good suffix array computed from P
3. $i \leftarrow m-1$, $j \leftarrow m-1$
4. while $i < n$ and $j \geq 0$ do
5.   if $T[i] = P[j]$ then
6.     $i \leftarrow i-1$
7.   else
8.     $i \leftarrow i + m - 1 - \min(L[T[i]], S[j])$
9.     $j \leftarrow j-1$
10. if $j = -1$ return $i + 1$
11. else return FAIL
```

**Exercise:** Prove that $i - j$ always increases on lines 9–10.

Boyé-Moore algorithm conclusion

- **Worst-case running time** $\in O(n + |\Sigma|)$
- This complexity is difficult to prove.
- **Worst-case running time** $O(nm)$ if we want to report all occurrences
- On typical English text the algorithm probes approximately 25% of the characters in $T$
- Faster than KMP in practice on English text.
Rabin-Karp Fingerprint Algorithm

**Idea:** use hashing
- Compute hash function for each text position
- No explicit hash table: just compare with pattern hash
- If a match of the hash value of the pattern and a text position found, then compares the pattern with the substring by naive approach

**Example:**
Hash "table" size = 97
Search Pattern $P$: 5 9 2 6 5
Search Text $T$: 3 1 4 1 5 9 2 6 5 3 5 8 9 7 9 3 2 3 8 4 6
Hash function: $h(x) = x \mod 97$ and $h(P) = 95$.

- $31415 \mod 97 = 84$
- $14159 \mod 97 = 94$
- $41592 \mod 97 = 76$
- $15926 \mod 97 = 18$
- $59265 \mod 97 = 95$

Guaranteeing correctness
Need full compare on hash match to guard against collisions
- $59265 \mod 97 = 95$
- $59362 \mod 97 = 95$

Running time
- Hash function depends on $m$ characters
- Running time is $\Theta(mn)$ for search miss (how can we fix this?)

Rabin-Karp Fingerprint Algorithm

The initial hashes are called fingerprints.
Rabin & Karp discovered a way to update these fingerprints in constant time.

**Idea:**
To go from the hash of a substring in the text string to the next hash value only requires constant time.
- Use previous hash to compute next hash
- $O(1)$ time per hash, except first one

Example:
Pre-compute: $10000 \mod 97 = 9$
Previous hash: $41592 \mod 97 = 76$
Next hash: $15926 \mod 97 = ??$

Observation:

$$15926 \mod 97 = (41592 - (4 \times 10000)) \times 10 + 6$$
$$= (76 - (4 \times 9)) \times 10 + 6$$
$$= 406$$
$$= 18$$
Rabin-Karp Fingerprint Algorithm

- Choose table size at random to be huge prime
- Expected running time is $O(m + n)$
- $\Theta(mn)$ worst-case, but this is (unbelievably) unlikely

Main advantage:
- Extends to 2d patterns and other generalizations

Suffix Tries and Suffix Trees

- What if we want to search for many patterns $P$ within the same fixed text $T$?
- Idea: Preprocess the text $T$ rather than the pattern $P$
- Observation: $P$ is a substring of $T$ if and only if $P$ is a prefix of some suffix of $T$.

We will call a trie that stores all suffixes of a text $T$ a suffix trie, and the compressed suffix trie of $T$ a suffix tree.

Suffix Trees

- Build the suffix trie, i.e. the trie containing all the suffixes of the text
- Build the suffix tree by compressing the trie above (like in Patricia trees)
- Store two indexes $l, r$ on each node $v$ (both internal nodes and leaves) where node $v$ corresponds to substring $T[l..r]$
Suffix Tree (compressed suffix trie): Example

\[ T = \text{bananaban} \]
\{bananaban, ananaban, nanaban, anaban, naban, aban, ban, an, n\}

Suffix Trees: Pattern Matching

To search for pattern \( P \) of length \( m \):
- Similar to Search in compressed trie with the difference that we are looking for a prefix match rather than a complete match
- If we reach a leaf with a corresponding string length less than \( m \), then search is unsuccessful
- Otherwise, we reach a node \( v \) (leaf or internal) with a corresponding string length of at least \( m \)
- It only suffices to check the first \( m \) characters against the substring of the text between indices of the node, to see if there indeed is a match
- We can then visit all children of the node to report all matches
### Suffix Tree: Example

\( T = \text{bananaban} \)

\( P = \text{nana} \)

### Pattern Matching Conclusion

<table>
<thead>
<tr>
<th>Method</th>
<th>Preproc.</th>
<th>Search time</th>
<th>Extra space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute-Force</td>
<td>( O(m</td>
<td>\Sigma</td>
<td>) )</td>
</tr>
<tr>
<td>DFA</td>
<td>( O(m) )</td>
<td>( O(n) ) (often better)</td>
<td>( O(m) )</td>
</tr>
<tr>
<td>KMP</td>
<td>( O(m) )</td>
<td>( O(n) ) (expected)</td>
<td>( O(m) )</td>
</tr>
<tr>
<td>BM</td>
<td>( O(m +</td>
<td>\Sigma</td>
<td>) )</td>
</tr>
<tr>
<td>RK</td>
<td>( O(m) )</td>
<td>( O(m) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>Suffix trees</td>
<td>( O(m) )</td>
<td>( O(m) )</td>
<td>( O(n) )</td>
</tr>
</tbody>
</table>