CS161 - String Operations

David Kauchak

• Basic string operations

Let $\Sigma$ be an alphabet, e.g. $\Sigma = (a, b, c, ..., z)$

A string is any member of $\Sigma^*$, i.e. any sequence of 0 or more characters of $\Sigma$

Given strings $s_1$ of length $n$ and $s_2$ of length $m$, here are some string functions we might use:

- Equality - Is $s_1 = s_2$ (can also consider case insensitive). $O(n)$ where $n$ is the length of the shortest string.
- Concatenate (append) - Create string $s_1s_2$. $\Theta(n + m)$
- Substitute - Exchange all occurrences of a particular character with another character. For example $\text{SUBSTITUTE}('\text{this is a string}', i, x) = '\text{thxs xs a strxng}'$. $\Theta(n)$
- Length - return the number of characters in the string. $\text{LENGTH}(s_1) = n - \Theta(1)$ or $\Theta(n)$ depending on how the string is stored.
- Prefix - Get the first $j$ characters in the string. $\text{PREFIX}(\text{this is a string'}, 5) = '\text{this '}. \Theta(j)$
- Suffix - Get the last $j$ characters in the string. $\text{SUFFIX}(\text{this is a string'}, 6) = '\text{string'}. \Theta(j)$
- Substring - Get the characters between $i$ and $j$ inclusive. $\text{SUBSTRING}(\text{this is a string'}, 4, 8) = 's is ' . \Theta(j - i)$

• Edit Distance (Levenshtein distance)

The edit distance between two strings is the minimum number of insertions, deletions and substitutions required to transform string $s_1$ into string $s_2$. 

1
Insertion: $ABACED \rightarrow ABACCED$

Deletion: $ABACED \rightarrow ABAED$

Substitution: $ABACED \rightarrow ABADED$

Some examples:

- $\text{Edit}(\text{Kitten, Mitten}) = 1$
- $\text{Edit}(\text{Happy, Hilly}) = 3$
- $\text{Edit}(\text{Banana, Car}) = 5$
- $\text{Edit}(\text{Simple, Apple}) = 3$

Edit distance is symmetric, that is:
$\text{Edit}(s_1, s_2) = \text{Edit}(s_2, s_1)$

Why?

Calculating the edit distance is similar to LCS.

$$\text{Edit}(X, Y) = \min \begin{cases} 
1 + \text{Edit}(X_{1\ldots n}, Y_{1\ldots m-1}) & \text{insertion} \\
1 + \text{Edit}(X_{1\ldots n-1}, Y_{1\ldots m}) & \text{deletion} \\
\text{Diff}(x_n, y_m) + \text{Edit}(X_{1\ldots n-1}, Y_{1\ldots m-1}) & \text{equal/substitution}
\end{cases}$$

where $\text{Diff}$ returns 1 if the characters are different and 0 if they are the same.
\begin{verbatim}
EDIT(X, Y)
1  m ← length[X]
2  n ← length[Y]
3  for i ← 0 to m
4      d[i, 0] ← i
5  for j ← 0 to n
6      d[0, j] ← j
7  for i ← 1 to m
8      for j ← 1 to n
9          d[i, j] = min(1 + d[i - 1, j],
10             1 + d[i, j - 1],
11             DIFF(x_i, y_j) + d[i - 1, j - 1])
10  return d[m, n]
\end{verbatim}

- Is it correct?
- Runtime?
  \( \Theta(nm) \)

Variants:
- Only include insertions and deletions
- Include swaps, e.g. swapping two adjacent characters counts as one edit
- weight insertion, deletion and substitution operations differently
- weight specific insertions, deletions and substitutions differently
- Length normalized

• String Matching
  contains, grep, search, find ...

Given a string pattern \( P \) of length \( m \) and a string \( S \) of length \( n \), find all the locations where \( P \) occurs in \( S \).

Example

• Naive method
Naive-String-Matcher($S, P$)
1. $n \leftarrow \text{length}[S]$
2. $m \leftarrow \text{length}[P]$
3. for $s \leftarrow 0$ to $n - m$
4. \hspace{1em} if $S[1...m] = T[s + 1...s + m]$
5. \hspace{1em} print “Pattern at $s$”

- Is it correct?
- Runtime?
  How long does the test for equality take?

  Best case: $O(1)$
  Worst case: $O(m)$

What is the best case for the algorithm?

The first character of the pattern does not occur in the string.
$\Theta(n - m + 1)$

What is the worst case?
The pattern occurs at every location, e.g.

$P = aaaa$
$S = aaaaaaaaaaaaaaaaaaa$

$O((n - m + 1)m)$

- String matching with finite state automata (FSA)

A FSA is defined by 5 components

- $Q$ is a the set of states
- $q_0$ is the start state
- $A \subseteq Q$ is a set of accepting states where $|A| > 0$
- $\Sigma$ is the input alphabet
- $\delta$ is the transition function from $Q \times \Sigma$ to $Q$
A finite state machine begins at state $q_0$ and reads the characters of the input string one at a time. If the automaton is in state $q$ and reads character $a$, then it transitions to state $\delta(q, a)$. If the FSA reaches an accepting state $q \in A$, then the FSA accepts the string read so far. A string that is not accepted is rejected by the FSA.

Example

We define the suffix function, $\sigma(x, y)$ to be the longest suffix of $x$ that is also a prefix of $y$, that is

$$\sigma(x, y) = \max_i(x_{m-i+1...m} = y_{1...i})$$

For example

- $\sigma(\text{abcdab}, \text{ababcd}) = 2$
- $\sigma(\text{daabac}, \text{abacac}) = 4$
- $\sigma(\text{dabb}, \text{abacd}) = 0$
- $\sigma(\text{daba}, \text{abacd}) = 3$

Why do we care about this function?

Consider trying to find the pattern “ababaca” in the string “abababaca”.

**Building a string matching automata**

Given a pattern $p_{1...m}$

- The set of states $Q$ is $0, 1, ..., m$
- The start state $q_0 = 0$
- The set of accept states $A = (q_m)$
- The vocab $\Sigma$ is all characters in the pattern plus an extra symbol for any character not in the pattern
- The transition function for $q \in Q$ and $a \in \Sigma$ is defined as:
  $$\delta(q, a) = \sigma(p_{1...q}a, P)$$
For example, given \( P = \text{ababaca} \)

<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></td>
</tr>
</tbody>
</table>

Given this finite automata, we then process the input string. Every time we reach state \( m \), then we know that there is a match.

- Is it correct?
- Runtime
  - Creating the automata:  
    - What is the best case? \( \Omega(|\Sigma|) \)
  
  Naive implementation (pg. 922 of [1]) - \( O(m^3|\Sigma|) \)
  
  Fast implementation \( O(m|\Sigma|) \)

  Overall runtime:

  - Preprocessing: \( O(m|\Sigma|) \)
  - Matching: \( \Theta(n) \)

- Rabin-Karp algorithm
  
  High-level idea: Given a pattern \( p_{1...m} \), create a hash function \( T \) that hashes \( m \) characters, such that given a \( T(s_{1...m}) \) we can efficiently calculate \( T(s_{2...m+1}) \). We can then compare the hash of the pattern with the hash of each \( m \) character string for a match.

  For simplicity, we’ll assume \( \Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \) (in general, we can use a base larger than 10 to suit our purposes). A string can then be viewed as a decimal number.
Given a pattern \( p \), we can calculate this number using Horner’s rule:

\[
d = p_m + 10(p_{m-1} + 10(p_{m-2} + \ldots + 10(p_2 + 10p_1)))
\]

in time \( \Theta(m) \)

Given a string \( s \), we would like to compute the decimal values at each location.

Example

We do this by first calculating it at the first position \( t_1 \) as above. To calculate the remaining positions we do the following:

\[
t_{i+1} = 10(t_i - 10^{m-1}s_i) + s_{i+m+1}
\]

that is, we subtract out the higher order digit, shift everything up a digit and add in the lowest order digit.

What is the cost of this operation? If we precompute \( 10^{m-1} \) then it is \( \Theta(1) \)

To calculate all of the matches we compare \( d \) to each \( t_i \) from \( i = 1 \) to \( n - m \). If \( d = t_i \) then it is a match.

- Is it correct?
- Runtime
  
  Preprocessing: \( \Theta(m) \)
  
  Matching: \( \Theta(n - m + 1) \)

  Is this right?

This assumes that we can calculate \( d = t_i \) in \( \Theta(1) \) time.

To get around this, we’ll calculate our functions modulo \( q \) so that the result fits in memory and we can calculate \( d \bmod q = t_i \bmod q \) in constant time.
We define \( d' = d \mod q \) and \( t'_i = t_i \mod q \)

We now use these values instead of \( d \) and \( t_i \) to check for equality.

The only challenge is *spurious hits* that is if \( d' = t'_i \) does not imply that \( d = t_i \). So, if we do get a hit, we must explicitly check if the pattern is actually equal.

- Is it correct?
- Runtime
  
  Preprocessing: \( \Theta(m) \)

  Best case: \( \Theta(n - m + 1) \)

  Worse case: \( \Theta(n - m + 1)m \)

  Average case:
  
  \( v \) is the number of valid hits

  How many spurious hits? probability of a spurious hit: \( 1/q \)

  \( O(n/q) \) spurious hits

  Preprocessing: \( \Theta(m) \)
  
  Matching: \( O(n - m + 1) + O(m(v + n/q)) \)

• Summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Preprocessing time</th>
<th>Matching time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>0</td>
<td>( O((n - m + 1)m) )</td>
</tr>
<tr>
<td>FSA</td>
<td>( \Theta(m</td>
<td>\Sigma</td>
</tr>
<tr>
<td>Rabin-Karp</td>
<td>( \Theta(m) )</td>
<td>( O((n - m + 1)m) )</td>
</tr>
<tr>
<td>Knuth-Morris-Pratt</td>
<td>( \Theta(m) )</td>
<td>( \Theta(n) )</td>
</tr>
</tbody>
</table>

(adapted from 32.2 pg. 907 from [1])

These notes are adapted from material found in chapters 32 of [1].

References