CS161 - String Operations

David Kauchak

- Basic string operations

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

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

  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 $s_1 s_2$. $\Theta(n + m)$
  - Substitute - Exchange all occurrences of a particular character with another character. For example $\textsc{Substitute}(\text{'this is a string'}, \text{i}, \text{x}) =$ 'thxs xs a strxng'. $\Theta(n)$
  - Length - return the number of characters in the string. $\textsc{Length}(s_1) =$ n - $\Theta(1)$ or $\Theta(n)$ depending on how the string is stored.
  - Prefix - Get the first j characters in the string. $\textsc{prefix}(\text{'this is a string'}, 5) =$ 'this '. $\Theta(j)$
  - Suffix - Get the last j characters in the string. $\textsc{suffix}(\text{'this is a string'}, 6) =$ 'string'. $\Theta(j)$
  - Substring - Get the characters between i and j inclusive. $\textsc{substring}(\text{'this is a string'}, 4, 8) =$'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.

Insertion: $ABACED \rightarrow ABACCED$

Deletion: $ABACED \rightarrow ABAED$

Substitution: $ABACED \rightarrow ABADED$

Some examples:

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

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

Why?

Calculating the edit distance is similar to LCS.

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

where DIFF returns 1 if the characters are different and 0 if they are the same.

```
EDIT(X, Y)
m ← length[X]
n ← length[Y]
for i ← 0 to m
        d[i, 0] ← i
for j ← 0 to n
        d[0, j] ← j
for i ← 1 to m
        for j ← 1 to n
                d[i, j] = min(1 + d[i − 1, j],
                              1 + d[i, j − 1],
                              DIFF(x_i, y_j) + d[i − 1, j − 1])
return d[m, n]
```

- 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)

$n \leftarrow length[S]$
$m \leftarrow length[P]$
for $s \leftarrow 0$ **to** $n - m$
    if $S[1...m] = T[s + 1...s + m]$
        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.

  $P = aaaa$
  $S = aaaaaaaaaaaaaaaaaa$

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

- 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 $|A| > 0$
  - Σ is the input alphabet
  - δ is the transition function from Q x Σ 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

- σ(abcdab, ababcd) $= 2$
- σ(daabac, abacac) $= 4$
- σ(dabb, abacd) $= 0$
- σ(daba, abacd) $= 3$

Why do we care about this function?

Consider trying to find the pattern "ababaca" in the string "abababacaba".

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 Σ 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 $P = ababaca$

state	a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

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? $\Omega(m|\Sigma|)$

  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 $p_{1\ldots m}$, create a hash function T that hashes m characters, such that given a $T(s_{1\ldots m})$ we can efficiently calculate $T(s_{2\ldots m+1})$. 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 calulate this number using Horner's rule:

$$d = p_m + 10(p_{m-1} + 10(p_{m-2} + ... + 10(p_2 + 10p_1)))$$

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 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 $d' = d \bmod q$ and $t'_i = t_i \bmod q$

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 explicity 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? probabilty 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

Algorithm	Preprocessing time	Matching time
Naive	0	$O((n - m + 1)m)$
FSA	$\Theta(m\vert\Sigma\vert)$	$\Theta(n)$
Rabin-Karp	$\Theta(m)$	$O((n - m + 1)m)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$

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

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

References
[1] Thomas H. Cormen, Charles E. Leiserson Ronald L. Rivest and Clifford Stein. 2007. Introduction to Algorithms, 2nd ed. MIT Press.