CS 315 - Day 34, Pattern Matching

The broad topic we begin discussing today are String algorithms.

In particular, we will discuss:

- String and Pattern Matching algorithms
- Text ***Similarity*** testing (used in DNA sequencing)
- File compression
- Data structures and algorithms used in search engines

Document processing is one of the dominant functions of technology.

We start with pattern matching, which formally defined is:

Given a ***pattern*** and a ***text***, we want to find all occurrences of the pattern in the text. Many applications, including searching for text in a document, finding web pages relevant to queries, and searching for patterns in DNA sequences.

Formally:

Input: The text is an array $T[1:n]$, and the pattern is an array $P[1:m]$, where $m \le n$. The elements of P and T are drawn from a finite ***alphabet*** Σ. Examples: $\Sigma = \{0,1\}$, Σ is the ASCII characters, or $\Sigma = \{\mathtt{A},\mathtt{C},\mathtt{G},\mathtt{T}\}$ for DNA matching. Call the elements of P and T ***characters***.

Output: All amounts that we have to shift P to match it with characters of T. Say that P ***occurs with shift s in T*** if $0 \le s \le n-m$ and $T[s+1:s+m] = P[1:m]$. We want to find all valid shifts.

This is the naïve/brute-force approach.

Σ is the alphabet of all the possible characters, T is the input string to search, and P is the pattern searched for.

Example: $\Sigma = \{\mathtt{A},\mathtt{C},\mathtt{G},\mathtt{T}\}$, $T = \mathtt{GTAACAGTAAACG}$, $P = \mathtt{AAC}$.

Just try each shift.

$\boxed{\text{NAIVE-STRING-MATCHER}}(T, P, n, m)$
for $s = 0$ **to** $n - m$
  if $P[1:m] == T[s+1:s+m]$
    print "Pattern occurs with shift" s

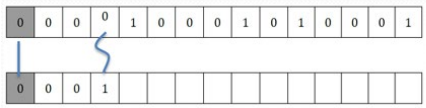
The shift s is the number of characters to "skip over," so P starts at $s+1$ in T. As defined above, P is m characters long and T is n characters long.
In the example above, we have shifts $s = 2$ and $s = 9$.

G	T	A	A	C	A	G	T	A	A	A	C	G
1	2	3	4	5	6	7	8	9	10	11	12	13

Time: Tries $n-m+1$ shift amounts, each taking $O(m)$ time, so $O((n-m+1)m)$. This bound is tight in the worst case, such as when the text is n As and the pattern is m As. No preprocessing needed.

$\boxed{\text{This algorithm is not efficient. It throws away valuable information.}}$

Brute Force?



It gets expensive if you "almost" find the string, over and over again.

So, can you describe the ***worst-case*** for the Naïve-String-Matcher algorithm?
Almost finding the string each loop, and the length of the Pattern P is $m = \left\lfloor \frac{n}{2} \right\rfloor$

So, we will explore two algorithms which are better because they pre-process or save information learned during the search. The first algorithm makes use of a structure we explored in MA 116 – Discrete Structures, a Finite-State-Automaton (FSA).

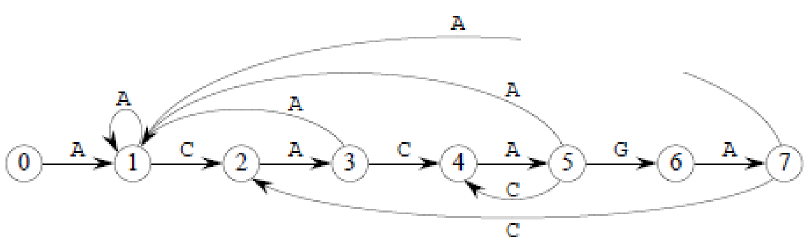
Formally: A finite automaton M is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$ where

- Q is a finite set of ***states***,
- $q_0 \in Q$ is the ***start state***,
- $A \subseteq Q$ is a distinguished set of ***accepting states***,
- Σ is a finite ***input alphabet***,
- δ is a function $Q \times \Sigma \to Q$, called the ***transition function*** of M.

The FA begins in state q_0 and reads the characters of the input string one at a time. When in state q and reading character a, it moves to state $\delta(q, a)$. If the current state is in A, then the FA has ***accepted*** the string read so far. An input that is not accepted is ***rejected***.

For string matching, the FA has $m + 1$ states, numbered 0 to m. The FA starts in state 0. When it's in state k, the k most recent text characters read match the first k characters of the pattern. When the FA gets to state m, it has found a match.

Example: Use the alphabet $\Sigma = \{\text{A}, \text{C}, \text{G}, \text{T}\}$ for DNA sequencing. If P = ACACAGA with $m = 7$, then the FA is *[leave this figure on the board]*



State-Transition-Diagram

The horizontal spine has edges labeled with P. Whenever P occurs in the text, the FA moves right, to the next state along the spine. When it reaches the rightmost state, it has found a match.

The transition function δ is defined for all states $q \in Q$ and all characters $a \in \Sigma$. Missing arrows are assumed to be transitions to state 0.

When a character from the text does not make progress toward a match, the FA moves to a lower-numbered state or stays in the same state. The only arrows pointing to the right are along the spine.

We'll see how to compute the transition function δ later. Assuming that we have it, here's how to perform string matching.

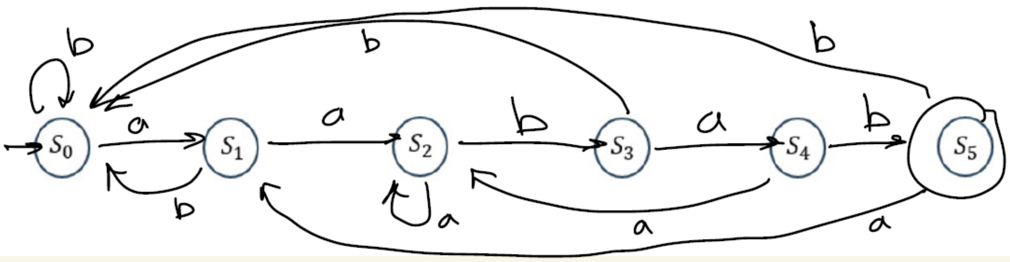
```
FA-MATCHER(T, δ, n, m)
  q = 0
  for i = 1 to n
      q = δ(q, T[i])
      if q == m
          print "Pattern occurs with shift" i − m
```

In-class w/***assigned groups of four (4)***, write 0. and 1. on board as ground rule(s):

0. No devices.
1. Intro: Name, POE, Icebreaker.
2. Pick someone on the group who will report out.
3. Create the State-Transition-Diagram for $\Sigma = \{a, b\}$, $P = aabab$

Hint: $m + 1$ states

4. Stretch goal, create the Next-State-Table.



So, we could just create a Next-State-Table directly from the State-Transition-Diagram.

It would have $m + 1$ rows and $|\Sigma|$ columns, and is called $\delta(\,\boldsymbol{current - state}\,, \boldsymbol{input - char})$, and outputs the next state:

State Transition Table

	Input		
State	**a**	**b**	**P**
0	δ(0,a) = σ(P_0a) = σ(a) = 1	δ(0,b) = σ(P_0b) = σ(b) = 0	a
1	δ(1,a) = σ(P_1a) = σ(aa) = 2	δ(1,b) = σ(P_1b) = σ(ab) = 0	a
2	δ(2,a) = σ(P_2a) = σ(aaa) = 2	δ(2,b) = σ(P_2b) = σ(aab) = 3	b
3	δ(3,a) = σ(P_3a) = σ(aaba) = 4	δ(3,b) = σ(P_3b) = σ(aabb) = 0	a
4	δ(4,a) = σ(P_4a) = σ(aabaa) = 2	δ(4,b) = σ(P_4b) = σ(aabab) = 5	b
5	δ(5,a) = σ(P_5a) = σ(aababa) = 1	δ(5,b) = σ(P_5b) = σ(aababb) = 0	

σ(P_ia) = "length of the longest prefix of P which is a suffix of P_ia"

But we should be able to automate the creation of the table:

How do we build the finite automaton? We already know that it has states 0 through m, the start state is 0, and the only accepting state is m. The hard part is the transition function δ. The idea:

> When the FA is in state k, the k most recent characters read from the text are the first k characters in the pattern.

A ***prefix*** $P[:i]$ of a string P is a substring consisting of the first i characters of P. A ***suffix*** is a substring consisting of characters from the end. For a string X and a character a, denote concatenating a onto X by Xa.

So, in state k, we've most recently read $P[:k]$ in the text. We look at the next character a of the text, so now we've read $P[:k]a$. How long a prefix of P have we just read?

> Find the longest prefix of P that is also a suffix of $P[:k]a$. Then $\delta(k, a)$ should be the length of this longest prefix.

In our example, $\delta(q, input - char) = \sigma(\, P[:q] \mid input - char\,)$

State, $q = 2, input - char = a$ → $\delta(2, a) = P[:2] \mid a = aa \mid a = aaa$
$q = 2, input - char = b$ → $\delta(2, b) = P[:2] \mid b = aa \mid b = aab$

COMPUTE-TRANSITION-FUNCTION(P, Σ, m)

```
for q = 0 to m
    for each character a ∈ Σ
        k = min{m, q + 1}
        while P[:k] is not a suffix of P[:q]a
            k = k − 1
        δ(q, a) = k
return δ
```

Preprocessing time:

- The outermost **for** loop iterates $m + 1$ times.
- The middle loop iterates $|\Sigma|$ times.
- The innermost **while** loop iterates at most $m + 1$ times.
- Each suffix check in the **while** loop test examines at most m characters of the pattern (since $k \le m$). Therefore, each suffix check takes $O(m)$ time.
- Total preprocessing time: $O(m^3 |\Sigma|)$.

Therefore, with a finite automaton, we can perform string matching with preprocessing time $O(m^3 |\Sigma|)$ and matching time $\Theta(n)$. It's possible to get the preprocessing time down to $O(m |\Sigma|)$.

The downside is the expense to pre-process and create the State Transition Table.

(# symbols in alphabet) * (length of ***P***)^3

But we can do even better, reducing the pre-processing time to $\theta(m)$ and keeping the matching time at $\theta(n)$. It is the Knuth-Morris-Pratt (KMP) algorithm, and it uses an auxillary array $\pi[1..m]$, which allows re-use of comparisons.

This array encapsulates knowledge about how the pattern matches against shifts of itself. We avoid (1) testing useless shifts and (2) pre-computing the Next-State-Table δ.

T b a c b | a b a b a | a b c b a b

s → | a b a b a | c a P

$\underbrace{\text{a b a b a}}$

first $q=5$ characters match

Using only our knowledge of the 5 matched characters, we know a shift of $s+1$ would be invalid. A shift of $s+2$ <u>might</u> work

b a c b a b a b a a b c b a b

$\xrightarrow[s+2]{}$ a b a b a c a

$k=3$

In the example above, the longest prefix of P that is also a <u>proper</u> suffix of P_5 is P_3

a b a b a

a b a P_k

$\pi[5] = 3$

$\{a\} \subseteq \{a$

$\{a\} \not\subset \{a$

$\subseteq$ vs $\subset$

prefix func

$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$

$\pi[q]$ is the length of the longest prefix of P that is a proper suffix of P_q.

$k<q$ not $k \le q$

If q characters have successfully matched at shift s, the next potentially valid shift is

$s + (q - \pi[q])$

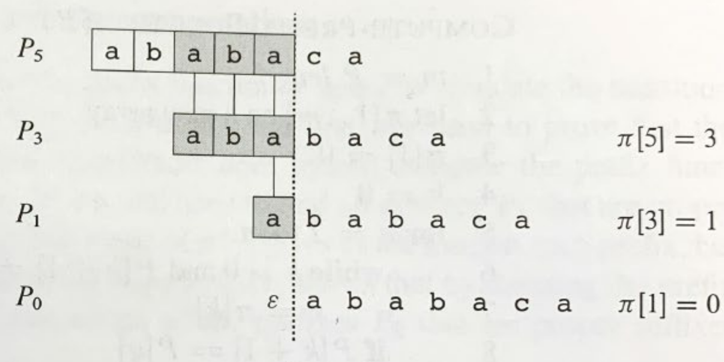
ex:

i	1	2	3	4	5	6	7	8	9	10
$P[i]$	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4	5	6	0	1

i	1	2	3	4	5	6
P[i]	a	b	a	c	a	b
π[i]	0	0	1	0	1	2

i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

(a)



(b)

Compute Prefix Function

```
Compute-Prefix-Function (P)
m = P.length
Π[1] = 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++
   Π[q] = k
return Π
```

KMP Matcher

```
KMP-Matcher (T,P)
n = T.length
m = P.length
Π = Compute-Prefix-Function
q = 0
for i=1 to n
   while q>0 and P[q+1] != T[i]
      q = Π[q]
   if P[q+1] = T[i]
      q++
   if q = m
      print "Pattern occurs w/shift" i-m
      q = Π[q]    // look for the next match
```