# CSE 584A Class 4

Jeremy Buhler

February 1, 2016

## 1 Getting *sp* values without $Z$-boxes

We've seen that we can use the Z-box algorithm to get *sp* values, but this is not actually how it's done in practice.

- Suppose we know $sp_{i-1}$ for a string $P$. How can we get $sp_i$?
- If $P[i] = P[sp_{i-1} + 1]$, then we have $sp_i = sp_{i-1} + 1$.

- But what if this doesn't hold?
- Consider *next* longest proper suffix of $P[1..i-1]$ that matches a prefix of $P$.
- How long is this next suffix? If $j = sp_{i-1}$, we claim that next suffix has length $sp_j$.
- Can prove by transitivity of matching:

- If $P[sp_j + 1] = P[i]$, then we have $sp_i = sp_j + 1$.
- Else, try the next longest matching proper suffix of $P[1..i-1]$...
- Eventually, we will either find a suffix whose corresponding prefix can be extended by $P[i]$, or we will discover that no suffix, not even the empty one, can be so extended. In latter case, $sp_i = 0$.

This idea leads to the following algorithm to get all *sp* values:

```
COMPUTESP(P)
    sp_1 ← 0                                              ▷ by definition
    for i = 2..|P| do
        j ← sp_{i−1}
        while j > 0 and P[j + 1] ≠ P[i] do
            j ← sp_j                                      ▷ length of next longest matching suffix

        if P[j + 1] = P[i]
            sp_i ← j + 1
        else
            sp_i ← 0                                      ▷ suffix = ϵ and P[i] unmatched
```

**Example**:

How fast is this algorithm?

- Not immediately obvious: inner while loop may run many times per value of $i$.
- But let's consider the extent of the matching suffix that the algorithm is trying to extend.
- On any given entry to the while loop, we're trying to extend suffix $P[i-j..i-1]$
- If extension fails (i.e. loop does not terminate), we move the start $I-J$ of this suffix forward by at least one character (because $j$ gets strictly smaller!)
- If extension succeeds, the start of the matching suffix stays put (both $i$, $j$ increase by 1 at start of next for loop iter).
- If $j$ goes to 0 and extension *still* fails, $j$ remains 0, and $i$ increases by 1, at start of next for loop iter, so $i-j$ increases.
- Hence, number of passes through while loop over all values of $i$ is bounded by $|P|$, since matching suffix cannot fall off end of $P$.
- Conclude that entire procedure takes time $\Theta(|P|)$.

## 2 An alternative view of KMP

We can also think of the execution of KMP graphically.

- Let $P$ be a pattern string of length $n$.
- Define the $i$th *failure link* $f_i$ for $P$ to be a pointer to the first character of $P$ that KMP compares to $T$ after seeing a mismatch at $P[i]$.

- Recall KMP: if we mismatch at $P[i], i > 1$, then we resume matching at $P[sp_{i-1}+1]$. Hence, for $i > 1$, we have
$$f_i = sp_{i-1} + 1.$$
(Hence, by definition, $f_2 = 1$.)
- If we mismatch at $P[1]$, then we resume at $P[1]$, so
$$f_1 = 1.$$
- KMP says that after matching *all* of $P$, we resume at $P[sp_n + 1]$.
- If we imagine that every pattern ends with a special $n + 1$st character \$ not found in the text, then we can define $f_{n+1}$ according to the general case, and all is well.

- How far do we advance in the text when following a failure link?
- Refer to behavor of KMP algorithm.
- If we mismatch at $P[i], i > 1$, we do not advance in the text but rather try to match the mismatched text character again (to a different pattern character)
- In special case of a successful pattern match, we advance in the text by 1 character to move past the match.
- If we mismatch at $P[1]$, we always advance in the text by 1 character.
- *Note*: we may follow multiple failure links on a mismatch in order to find next match to the pattern, if any. However, each link corresponds to shifting the pattern forward in the text by at least one character, so as we proved earlier, the *total* number of links followed by KMP is $O(|T|)$.

Substituting the definition of $f_i$ into our algorithm for computing $sp$ values and simplifying, we can directly compute all failure links for $P$ in time $\Theta(|P|)$ as follows:

COMPUTEF($P$)

```
f_2 ← 1
for i = 2..|P| do
    k ← f_i
    while k > 1 and P[k] ≠ P[i] do
        k ← f_k

    if P[k] = P[i]
        f_{i+1} ← k + 1
    else
        f_{i+1} ← 1
```

## 3 An optimization for small alphabets

- Preprocessing for KMP runs in time independent of the alphabet size and results in exactly one failure link per position of $P$.
- But alphabet independence comes at a cost to running time!
- Consider the pattern $P = aataac$.

- Suppose we match $P$ against $T = aataag\ldots$
- First mismatch is at $P[6]$ and $T[6]$, so KMP next tries to match $T[6]$ against $P[f_6 = 3]$.
- But now we immediately mismatch again! Target of failure link is $t$, but text character is a $g$.
- In fact, the complete set of failure links we will follow on this mismatch is determined *given that the text character is a $g$.* (I.e., we keep following until we either see a $g$ in $P$ or reach the beginning of $P$.)

How can we exploit this observation to reduce the number of character comparisons in our algorithm?

- Suppose we extend failure links from $f_i$ to $g_{i,c}$.
- $g_{i,c}$ points to first occurrence of $c$ reached by following failure links starting from from position $i$.

- Hmmm. What about $g_{6,g}$?
- We can follow the path forever and never see a $g$. In such cases, define $g_{i,c} = 0$.

Why do the $g_{i,c}$'s help us?

- **Claim**: If, on a mismatch between $P[i]$ and a character $T[j] = c$, we resume matching at $P[g_{i,c} + 1]$ rather than $P[f_i]$, then KMP can find all instances of $P$ in $T$ using at most $|T|$ comparisons.
- Notice $g_{i,c}$ either points to an instance of character $c$ in $P$ or to 0.
- If $g_{i,c} > 0$, then in a single step, we shift $P$ so as to match $T[j]$ and so do not look at $T[j]$ again.
- If $g_{i,c} = 0$, then we shift $P$ entirely past the mismatch and so do not look at $T[j]$ again.
- In each case, matching resumes between $T[j + 1]$ and $P[g_{i,c} + 1]$.
- Correctness still follows by correctness of KMP, but no character of $T$ is ever read twice, so we must be doing at most $|T|$ comparisons.

What are we paying for this improved performance?

- Space for $g$ values is now $\Theta(|P||\Sigma|)$, i.e. dependent on alphabet size.
- For a small alphabet (e.g. DNA!), this may be a worthwhile tradeoff.
- Moreover, it takes longer to compute the $g$ table than to get the failure links alone.
- *Exercise*: show how to compute the $g_{i,c}$ values for pattern $P$ in time $\Theta(|P||\Sigma|)$.

# 4 Finite Automata

What have we achieved?

- A *deterministic finite automaton* (DFA) is an abstract machine for reading strings.
- It consists of a set $Q$ of *states* $q_i$ and a *transition function* $\delta : Q \times \Sigma \to q$.
- We also designate an *initial state* $q_0$ and an *accepting state* $q_a$.
- The automaton starts in state $q_0$ and consumes text characters.
- If we are in state $q$ and read character $c$, we *transition* to state $\delta(q, c)$.
- (Transitions are typically drawn as arrows. In general, there *must* be a transition defined for each pair $(q, c)$.)
- Whenever we reach state $q_a$, we *accept* (i.e. report a match).

Here's a suggestive example of a DFA:

This is just the DFA corresponding to the string $P = aataac$ with its extended failure links $g_{i,c}$.

- For a string $P$ of length $n$, we have states $q_0 \ldots q_n$.
- State $q_i$ is intuitively "the state reached after seeing $P[1..i]$." ($q_0$ means we've seen no characters of $P$ yet).
- Naturally, state $q_n$ is accepting!
- The "forward edges" of the DFA correspond to seeing the characters of $P$.
- If we have a failure link $g_{i,c}$, what is the equivalent transition in the DFA?
- Note that the edges out of state $q_{i-1}$ implement the outcomes of testing $P[i]$ against the text.
- Conclude that for $c \neq P[i]$, $\delta(q_{i-1}, c) = q_{g_{i,c}}$.

KMP can be viewed as just a way to construct a DFA that recognizes the pattern string $P$! (Possibly not the minimal DFA, but better than the subset construction.)