Note 2: Pattern Matching

Pattern matching is one of the most ubiquitous problems in computer science, and shows up not only in obvious applications such as text processing but also in other areas such as DNA sequence analysis. In this problem, we are given as input two strings x, y over a common alphabet Σ: x, of length n is the *text* and y, of length $m \ll n$ is the *pattern*. Our goal is to determine whether y occurs as a contiguous substring of x and, if so, to output the location of the first such occurrence.

The obvious algorithm for pattern matching simply "slides" y along x, one symbol at a time, and in each position checks whether the pattern matches the substring of the text starting at that position. This algorithm takes time $O(nm)$ in the worst case, since we may have to perform $O(m)$ checks at each of the $O(n)$ possible positions. If we imagine a typical biological application of searching for a motif in a genome sequence, where $n \approx 100M \approx 2^{27}$, $m \approx 2^8$ and the alphabet is $\{A, C, T, G\}$, then we may have to perform up to about 2^{35} character comparisons!

Exercise: If $x = AAAAAAAAAAAAAAAAB$ and $y = AAAB$, how many comparisons are performed to detect the pattern?

One of the most important applications of finite automata is the so-called Knuth-Morris-Pratt (KMP) pattern matching algorithm from the early 1970s, which lowers the complexity to $O(n + m)$, which is linear in the size of the input (and hence optimal, since we need to read both the pattern and the text). We will sketch the operation of the KMP algorithm informally with the aid of a concrete toy example. Suppose we are given the text $x = ABACCABABACA$ and the pattern $y = ABACA$. The naive algorithm will start matching y from position 1, and will detect a mismatch when it sees the C in position 5. It will then start matching again from position 2, then position 3 and so on. In contrast, the KMP algorithm uses the fact that it has already read the first five symbols $ABABC$ to deduce that it can safely skip ahead and start matching again at position 6. The next mismatch detected is at position 9, when a B is encountered instead of the desired C; in this case, the algorithm can move the pattern forward to position 8 and start matching its third symbol at position 10 (since it knows it has already matched the prefix AB in positions 8 and 9). Note that, under this scheme, each symbol in the input string is read only once!

To keep track of the relevant symbols it has read, and the various skips that are available, the KMP algorithm uses a DFA. This DFA depends only on the pattern (not on the text), and is built ahead of time. Figure 1 illustrates the state diagram of the DFA for our example pattern $y = ABACA$ (over alphabet $\{A, B, C\}$).

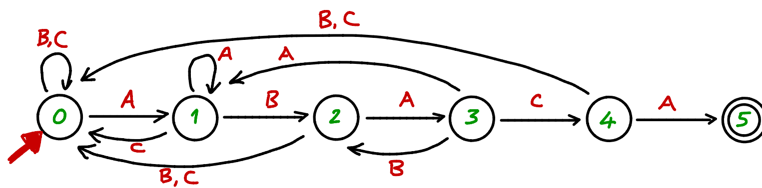

Figure 1: DFA for the pattern $ABACA$.

There are six states, labeled $0, \dots, m$ (one for each initial segment of the pattern—including the empty initial segment). The main "spine" of transitions (from state i to state $i + 1$) corresponds exactly to the pattern itself. The initial state is 0 and the only accepting state lies at the end of this spine; it is reached if and only if the pattern is found in the text. All other transitions correspond to a mismatched symbol, and

take the DFA back to an earlier state in the spine that represents the longest matched prefix of the pattern ending at the current position in the text.

To specify this construction more formally, we introduce the notation $y[k]$ to denote the prefix consisting of the first k symbols of y, for $0 \leq k \leq m$. We then define the transition function of the automaton as follows:

$$\delta(k, a) = \max\{j : y[j] \text{ is a suffix of } y[k]a\} \quad \text{for } 0 \leq k \leq m \text{ and } a \in \Sigma. \tag{1}$$

In other words, upon reading the new symbol a in the text, we transition to the state corresponding to the largest prefix of the pattern that is matched by a suffix of the text read so far. You should check carefully that you understand this construction.

Exercise: Hand-turn the above DFA on the input text $x = ABACCABABACA$ from the above example.

Given the above finite automaton, the pseudo-code for the overall matching algorithm is as follows (where δ denotes the transition function of the DFA, and x_i denotes the symbol at position i of x):

$q := 0; \quad i := 1$
repeat
  $q := \delta(q, x_i)$
  $i := i + 1$
until $q = m$ or $i = n + 1$
if $q = m$ **then** output "match found at position $i - m$"
else output "no match found"

Note that this code simply runs the DFA on the text as input, halting when the first occurrence of the pattern is found. We leave a formal verification of this algorithm to the reader.

The running time of this algorithm, given the DFA, is clearly $O(n)$, since it attempts to match each text symbol only once. However, there is additional overhead involved in constructing the DFA. If we do this naively, following the definition (1) above, it will take time $O(m^3|\Sigma|)$, or $O(m^3)$ for a fixed alphabet. (Exercise: Why?). This is not too bad if m is small or if we are searching for the same pattern in multiple texts (as the DFA only needs to be computed once). However, the cost of this preprocessing step can be reduced to $O(m|\Sigma|)$ (i.e., $O(m)$ for a fixed alphabet) using a cleverer strategy for constructing the DFA, thus bringing the overall running time down to the value $O(n + m)$ claimed earlier. The KMP algorithm itself actually circumvents this overhead entirely by avoiding the explicit construction of the DFA and instead computing the transition function "on the fly." We omit the details.

Exercise: The DFA in Figure 1 has no transitions out of its accepting state (implying that it halts when a match is found). How would you add transitions out of the accepting state so that the DFA can be used to search for additional occurrences of the pattern after one is found?

Exercise: Construct the DFA for the pattern $y = AAAB$ from the exercise above (over alphabet $\{A, B\}$). How many comparisons are performed by the KMP algorithm on the text in that exercise?

Notes: The KMP algorithm itself is a bit more involved than the sketch above, mainly because it avoids explicitly constructing the finite automaton

An alternative classical linear-time pattern-matching algorithm is due to Boyer and Moore, a few years later than KMP. Unlike KMP, the Boyer-Moore algorithm starts matching the pattern from the last symbol rather than the first, allowing it to make big "jumps" if no match is found. Boyer-Moore is more complicated than KMP, and is usually a little faster in practice, though KMP is often the method of choice for small alphabets such as $\{A, C, T, G\}$. A very simple and efficient *randomized* algorithm due to Karp and Rabin also runs in linear time with a lower overhead (but has a very small probability of falsely detecting a match).