1 Combinatorial Pattern Matching

(Oct. 9)

Over the years the expression combinatorial pattern matching has become a synonym for the field of theoretical computer science concerned with the study of combinatorial algorithms on strings and related structures.

The term *combinatorial* emphasizes that these are algorithms based on mathematical properties and a deep understanding of the individual problems, in contrast to statistical or machine learning approaches, where general frameworks are often applied to model and solve the problems.

Work in this field began in the 1960s with the study of how to efficiently find all occurrences of a pattern string p in a text t. We, too, take this as our starting point…

But before looking at the highlight of this section, the Knuth–Morris–Pratt algorithm for string matching, we'll get a taste of the core computational model used by the algorithm, namely *deterministic finite automata* (DFAs for short).

1.1 Finite Automata and Regular Expressions

Finite automata are the most basic models of computation. What can such a basic model of computation do? Many useful things! In fact, we interact with such computers all the time, as they lie at the heart of various electromechanical devices.

Before studying finite automata, let's set up some terminology:

Definition. An *alphabet* is a finite set of symbols or letters. A *word* or *string* over an alphabet Σ is a finite sequence of symbols from Σ.

Definition. The empty string (the string with no symbols) is denoted by ϵ.

The empty string ϵ is to strings what 0 is to the integers. This'll become clear in a second:

Definition. Given two strings $x = x_1 \dots x_n$ and $y = y_1 \dots y_m$, the *concatenation* of x and y is denoted by xy and is the string $x_1 \dots x_n y_1 \dots y_m$.

Thus, for any string x, we have $x\epsilon = \epsilon x = x$. Some more definitions follow.

Definition. Given a string $x = x_1 \dots x_n$ over Σ (i.e. $x_i \in \Sigma$)we define:

1. The *length* of x is written $|x|$ and is the number of characters in x (here, $|x| = n$).
2. A *prefix* of x is a string $x_1 \dots x_i$ for some $1 \leq i \leq n$.
3. A *suffix* of x is a string $x_j \dots x_n$ for some $1 \leq j \leq n$.

And finally, we have:

Definition. A *language* is a set of strings over some alphabet Σ.

1.1.1 Operations on languages

Since a language is, in some sense, nothing but a set, we can define operations on languages just as we do with sets.

1. The union, intersection, and complement of a language are defined just as they are for sets.
2. Given two languages L_1 and L_2 over Σ, we define the language $L_1 \circ L_2 = \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\}$, that is, $L_1 \circ L_2$ is the language obtained by concatenating all words in L_1 with all words in L_2.
3. Let $L^0 = \{\epsilon\}$. Define $L^i = L \circ L^{i-1}$ for $i \geq 1$. The *Kleene closure* of a language L over alphabet Σ is defined the language $L^* = L^0 \cup L^1 \cup L^2 \ldots = \bigcup_{i=0}^{\infty} L^i$.
4. Note that $\epsilon \in L^*$, but sometimes we will not want to include the empty string in the Kleene closure, leading us to define the *positive closure* of a language. Given a language L, it's positive closure, denoted L^+, is given by $L^+ = L^1 \cup L^2 \ldots = \bigcup_{i=1}^{\infty} L^i$.

In words, the Kleene closure of a set of strings is a new set containing all possible combinations of concatenations of strings from the original language. The Kleene closure of a language is also called the *Kleene star* of a language, and sometimes we'll be lazy and refer to it directly as just the *star* of a language.

Here's some examples:

1. $0^* = \{\epsilon, 0, 00, 000, \ldots\}$.
2. $0^* \cup 1^* = \{\epsilon, 0, 1, 00, 11, 000, 111, \ldots\}$.
3. $\{0 \cup 1\}^* =$ all strings over the alphabet $\{0, 1\}$.

1.1.2 Regular Expressions

In arithmetic, we can use the operations $+$ and $\times$ to build up expressions such as $(5+3) \times 4$. Similarly, we can use *regular operations* to build up expressions describing languages, which are called *regular expressions*. An example is $(0 \cup 1)0^*$.

This'll become very clear in a second.

Definition. The *regular expressions over* Σ and the languages they denote are defined recursively as follows:

1. $\emptyset$ is a regular expression denoting the empty language $\emptyset$.
2. ϵ is a regular expressions and denotes the language $\{\epsilon\}$.
3. For each symbol $a \in \Sigma$, a is a regular expression and denotes the language $\{a\}$.
4. If p and q are regular expressions denoting the languages P and Q respectively, then the following are regular expressions:
   (a) $(p+q)$ corresponding to the language $P \cup Q$
   (b) pq corresponding to the language $P \circ Q$
   (c) p^* corresponding to the language P^*
   (d) p^+ corresponding to the language P^+

Here are some examples of regular expressions and the languages that they correspond to:

Examples. Let $\Sigma = \{0, 1\}$.

1. $0^* = \{\epsilon, 0, 00, 000, \ldots\}$
2. $0^*1 = \{1, 01, 001, 0001, \ldots\}$
3. $(0 + 1) = \{0, 1\}$
4. $(0 + 1)^* = \Sigma^* =$ all strings over Σ
5. $1(0 + 1)^*1 + 1 =$ all strings starting and ending with 1

Those languages that correspond to some regular expression have several *nice* properties (which we'll see when we learn about finite automata), and thus have a special name:

Definition. A language is said to be a *regular language* if and only if it is denoted by a regular expression over a finite alphabet.

1.1.3 Deterministic Finite Automata

There exist two types of finite state automata: *deterministic* and *non-deterministic*. We'll see that these two are actually equivalent from a computational point of view, and each of them is in turn equivalent to a regular expression.

Informally, a deterministic finite automaton (DFA) is a machine with a control unit, an input tape, and a head that processes the input tape from left to right, such that:

- The control unit consists of a number of *states*, and the DFA is in one of these states at any given instance.
- Each time the head reads an input, the DFA can transition from one state to another.

Finite automata and their probabilistic counterpart Markov chains are useful tools when we are attempting to recognize patterns in data. These devices are used in speech processing and in optical character recognition. Markov chains have even been used to model and predict price changes in financial markets.

Here's a more formal definition:

Definition. A *deterministic finite automaton* (DFA) is a machine $M = (S, \Sigma, \delta, s_0, F)$ comprising of a control unit, a tape for input, and a head that reads the tape left to right, where:

1. S is a *finite* set of states.
2. Σ is a finite alphabet for the symbols on the tape.
3. $\delta : S \times (\Sigma \cup \epsilon) \to S$ is the *state transition function*
4. s_0 is the initial state of the DFA.
5. $F \subseteq S$ is the set of *final* or *accepting* states.

In order to describe the state of a DFA at some point during computation, we define the following:

Definition. An *instantaneous description* (ID) of a DFA is a pair (s, w) where $s \in S$ represents the current state, and $w \in \Sigma^*$ represents the unused portion of the input tape, i.e. the symbol under the tape head followed by the rest of the string to the right.

Note that the *initial* ID is the ID given by (s_0, w) and an *accepting* ID is an ID of the form (s, w) for $s \in F$.

Having defined the notion of an ID, we'll set up notation to talk about how these IDs change throughout the computation by means of a binary relation.

Definition. If $\delta(s, a) = s'$, then we write $(s, aw) \vdash (s', w)$ for all $w \in \Sigma^*$, where $a \in \Sigma \cup \epsilon$.

Note that if, in the above definition, $a = \epsilon$, then we transition from state s to s' without reading any input.

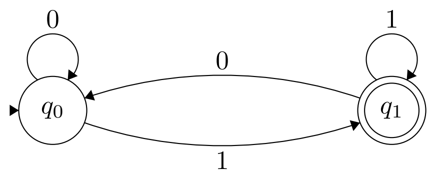
Definition. We use $\vdash^*$ to denote the reflexive, transitive closure of $\vdash$.

And finally:

Definition. The language $L(M)$ of a DFA M is the set of strings *accepted* by M, that is:

$$L(M) = \{w \in \Sigma^* \mid (s_0, w) \vdash^* (s, \epsilon) \text{ for some } s \in F\}$$

This was not done in class, but here's an example of a simple DFA. Suppose we have the alphabet $\Sigma = \{0, 1\}$. Consider:



In the picture above, q_0 is the initial state of the DFA, and q_1 is an accepting state (commonly denoted using the double circles in the above diagram).

The transition function is represented as arrows, e.g. $\delta(q_0, 1) = q_1$, and $\delta(q_0, 0) = q_0$ are represented by an arrow from $q_0 \to q_1$ for "input" 1, and similarly for the self-loop from $q_0 \to q_0$.

What's the language of this DFA? Indeed, the DFA accepts all strings that end in 1. Here's an example of a computation by this DFA on input 10110:

$$(q_0, 10110) \vdash (q_1, 0110) \vdash (q_0, 110) \vdash (q_1, 10) \vdash (q_1, 0) \vdash (q_0, \epsilon)$$

and as $q_0 \notin F$, we say that the DFA *rejects* the string 10110.

On the other hand, 101101 is accepted by the DFA:

$$(q_0, 101101) \vdash (q_1, 01101) \vdash (q_0, 1101) \vdash (q_1, 101) \vdash (q_1, 01) \vdash (q_0, 1) \vdash (q_1, \epsilon)$$

(Oct. 11) **1.1.4 Nondeterministic Finite Automata**

Put (very) informally, a nondeterministic finite automaton (NFA) is nothing but a DFA where we're allowed to be in multiple states at the same time. Spooky, isn't it? :)

Formally, we have:

Definition. A *nondeterministic finite automaton* (NFA) is a machine $M = (S, \Sigma, \delta, s_0, F)$ comprising of a control unit, a tape for input, and a head that reads the tape from left to right, where:

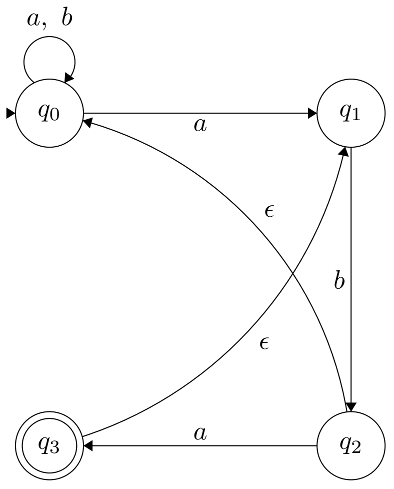
1. S is a *finite* set of states.
2. Σ is a finite alphabet for the symbols on the tape.
3. $\delta : S \times (\Sigma \cup \epsilon) \to \mathscr{P}(S)$ is the *state transition function*
4. s_0 is the initial state of the NFA.
5. $F \subseteq S$ is the set of *final* or *accepting* states.

The *language* of a NFA is defined analogously to what we did for DFAs, and the definitions for IDs carry over as well.

Well, so what's different? Note that the definition above looks very similar to the one for DFAs, with one small (but significant!) change: the transition function has range $\mathscr{P}(S)$, which is the *power set* or the set of all subsets of S.

This makes rigorous our earlier idea of "being in multiple states at the same time".

Here's an example of a NFA:



What's the language of this NFA? Indeed, it accepts all strings that end in *aba*. Let's see a concrete example.

Suppose we have a string *ababa* provided as input to the NFA above. Our NFA starts out in the initial state q_0 as seen from the picture above. We denote one possible series of transitions of IDs:

$$(q_0, ababa) \vdash (q_1, baba) \vdash (q_2, aba) \vdash (q_3, ba) \vdash (q_1, ba) \vdash (q_2, a) \vdash (q_3, \epsilon)$$

and so *ababa* is in the language of the machine above.

Alternatively, another computation history is given by:

$$(q_0, ababa) \vdash (q_0, baba) \vdash (q_0, aba) \vdash (q_0, ba) \vdash (q_0, ba) \vdash (q_0, a) \vdash (q_0, \epsilon)$$

Thus, unlike a DFA, a NFA can have multiple computation histories for the same input string. As long as there exists one computation history in which the given string is accepted, we say that the NFA accepts that string!

1.1.5 Tying it all together

You might expect that being able to be in multiple states at the same time means that NFAs are more *powerful* than DFAs, i.e. can recognize some languages that DFAs cannot recognize. This, however, is not true.

It turns out that every NFA can be converted to a DFA; the issue, however, is that this transformation results in an exponential number of states in the DFA as compared to in the NFA. It also turns out that every language recognized by a DFA is a regular language.

Let's state this formally:

Theorem. Every NFA can be converted to a DFA recognizing the same language.

Theorem. Every language recognized by a DFA is a regular language.

We thus have the following correspondence:

$$\text{NFAs} \iff \text{Regular Expressions} \iff \text{DFAs}$$

Formal proofs were not given in class, but you can find them in Sipser's Introduction to the Theory of Computation (or in CS 1010 ☺).

1.2 The Knuth-Morris-Pratt Algorithm

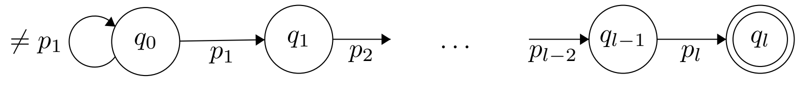
Let's get back to our original problem, namely string matching. Formally, our problem is as follows:

String Matching
Input: A pattern p and a text t, over some alphabet Σ.
Output: The first exact occurrence of p in t.

1.2.1 First steps

Let's get started with constructing M_p.

1. First construct a *skeletal* DFA:



2. Observe that the state q_i of M_p corresponds to the prefix $p_1 \dots p_i$ of p.

3. Our machine M_p will start in state q_0 reading t_1, the first symbol in the text.

However, what do we do in case we're at state q_j with the next input symbol t_i, and $p_{j+1} \neq t_i$? Should we just start over from q_0? But then, this isn't any different from Camillo's algorithm!

Instead, we can harness the fact that DFAs have, in some sense, a *memory*, and use this fact to transition to an *appropriate* state in our machine M_p. This is all very vague, but will become clear soon once we learn about the *failure function*.

(Oct. 18)

1.2.2 The Failure Function

Before proceeding, we recall some facts about our skeletal machine M_p. The state q_j of the skeletal machine M_p corresponds to the prefix $p_1 \dots p_j$ of p. Also, M_p starts in state q_0, and so we need to position it *vis-á-vis* the text. Thus, the head of M_p starts off at t_1 on the input tape.

Suppose after having read $t_1 t_2 \dots t_k$ (the first k characters of the text t), we find that M_p is in state q_j. This implies that:

1. The last j symbols of $t_1 \dots t_k$ are $p_1 p_2 \dots p_j$.
2. The last m symbols of $t_1 \dots t_k$ are *not* a prefix of $p_1 \dots p_l$ for $m > j$, i.e. the suffix of size of j of $t_1 \dots t_k$ is the prefix of size j of p.

Now, if $t_{k+1} = p_{j+1}$, then we move to state q_{j+1} and our tape head moves right by one. But what if $t_{k+1} \neq p_{j+1}$? In this case, where should we transition to?

If you think about it, you'll realize that we want M_p to enter the state q_i for highest i such that $p_1 \dots p_i$ is a suffix of $t_1 \dots t_{k+1}$. In order to determine this i, we construct that we'll call the *failure function* for the pattern p. Formally, we have:

Definition. The function f such that $f(j)$ is the largest integer $s < j$ for which $p_1 \dots p_s$ is a suffix of $p_1 \dots p_j$ is said to be the *failure function* for the pattern p.

What exactly does this function compute? In words, $f(j)$ is the number of positions we can "fall back" to keep looking for the pattern p. Also, note that $s < j \implies p_1 \dots p_s$ is a *proper* prefix of $p_1 \dots p_j$.

Here's an example of the failure function f for the pattern $p = aabbaab$ over $\Sigma = \{a, b\}$.

i	1	2	3	4	5	6	7
$f(i)$	0	1	0	0	1	2	3

More concretely, suppose $i = 2$. Then the prefix we're dealing with is aa, and the only proper prefix of this prefix is a, which happens to be a suffix as well. So $f(2) = 1$.

Similarly aab is the longest proper prefix of $aabbaab$ that is also a suffix of $aabbaab$, and hence $f(7) = 3$.

1.2.3 How do we use the failure function?

We'll now look at an algorithm that utilizes failure function for a pattern p, in order to find an occurrence of p in text t.

First, define f^n recursively as follows: let $f^1(j) = f(j)$. We say $f^n(j) = f(f^{n-1}(j))$.

Thus, from our example in §2.2.2, we have $f^2(6) = 1$. Intuitively, $f^n(j)$ is just f applied n times to j.

Suppose that M_p is in state j having read $t_1 \dots t_k$, and $t_{k+1} \neq p_{j+1}$. At this point, M_p applied the failure function repeatedly to j, and two cases arise:

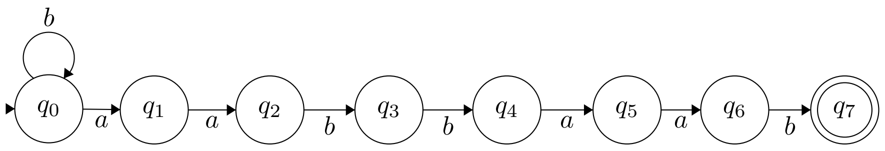
1. $f^m(j) = u$ and t_{k+1} is precisely p_{u+1}, or
2. $f^m(j) = 0$ and t_{k+1} is different from p_i for all $i \in \{1, \dots, j\}$.

In the first case above, we want M_p to enter state q_{u+1}. In the second one, M_p enters the state q_0. But in both cases, the tape head proceeds to the cell on the input tape with t_{k+2}.

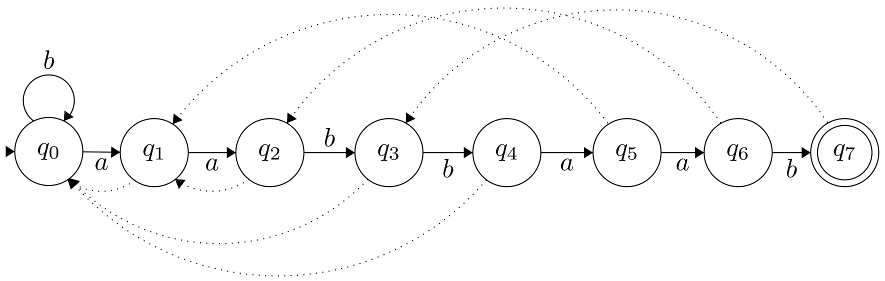
In Case 1, it's easy to see that if $p_1 \dots p_j$ was the longest prefix of p that is a suffix of $t_1 \dots t_k$, then $p_1 \dots p_{f^m(j)+1}$ is the longest prefix of p that is a suffix of $t_1 \dots t_{k+1}$. In Case 2, no prefix of p is a suffix of $t_1 \dots t_{k+1}$.

The algorithm is now clear: M_p proceeds reading t_{k+2} and so on and so forth, operating in the fashion described above, until it reaches the final state q_l, in which case it accepts. If it never reaches the final state, then the given text t has no occurrence of p.

Let's go back to our earlier example from §2.2.2 where $p = aabbaab$. We have a skeletal DFA for p as follows:



Having computed the failure function, we can add in the following arrows:



Let $t = abaabaabbaab$.

Initially M_p is in state q_0. On reading the first symbol of t (i.e. t_1 which is an a), M_p enters q_1. Since there is no transition from q_1 on the second input symbol of t (i.e. t_2 which is b), M_p enters state q_0. More explicitly, M_p goes back to the state given by the output of the failure function from q_1.

Now, $u = 0$ and $p_{u+1} = p_1 = a \neq t_2$, and so Case 2 (from the discussion in § 2.2.3) prevails and M_p remains in state 0. From here M_p continues consuming characters in the input string and following the corresponding arrows. If the machine ever reaches q_7, the pattern p has been found in the text t.

1.2.4 Computing the failure function

Having seen the failure function in action, let's now see how to compute it. That is, given a string $p = p_1 \dots p_l$, how do we compute its failure function?

```
 1: function FAILURE FUNCTION($p = p_1 \dots p_l$)
 2:     $f(1) \leftarrow 0$
 3:     $i \leftarrow 0$
 4:     for $j \in \{2, \dots, l\}$ do
 5:         $i \leftarrow f(j-1)$
 6:         while $p_j \neq p_{i+1}$ and $i > 0$ do
 7:             $i \leftarrow f(i)$
 8:         end while
 9:         if $p_j \neq p_{i+1}$ and $i = 0$ then
10:             $f(j) \leftarrow 0$
11:         else
12:             $f(j) \leftarrow i + 1$
13:         end if
14:     end for
15: end function
```

(Oct. 20)

1.2.5 Proof of correctness for the failure function

How can we be sure our algorithm for constructing the failure function is correct? By proof, of course! We'll refer to the pseudocode above in our proof using line numbers (L#).

Let's prove the definition of the failure function by induction: that for a pattern p of length l, $f(j) = i$ is the largest integer $i < j$ such that $p_1 p_2 \dots p_i = p_{j-i+1} p_{j-i+2} \dots p_j$.

Assume that this induction hypothesis is true for all $f(k)$ such that $k < j$.

Proof. **Base case:** $f(1) = 0$. Empty string equality, if you will, but it checks out.

Induction step: Our algorithm compares p_j with $p_{f(j-1)+1}$ in L6 using the assignment from L5. This splits our analysis into two cases:

- **Case 1:** $p_j = p_{f(j-1)+1}$

  The L6 and L9 blocks do not get executed, and we continue to L12. By the induction hypothesis, $f(j-1) = i$ is the largest i such that $p_1 \dots p_i = p_{j-i+1} \dots p_j$. Since the comparison from L6 ensures that the next character in p matches the character after the $j-1$th prefix, L12 assigns $f(j)$ correctly.

- **Case 2:** $p_j \neq p_{f(j-1)+1}$

The L6 loop finds the largest i such that

$$\begin{cases} p_1 \dots p_i & = p_{j-i} \dots p_{j-1} \quad i\text{th prefix matches } (j-i)\text{th suffix} \\ p_{i+1} & = p_j \quad \text{next characters match} \end{cases}$$

If such an i exists, the L6 loop will consider all possible i which satisfies the above equalities. Note that $f(k)$ is always less than k, so L6 will check these i from greatest to least. Based on this ordering, it finds the largest such i. So L12 assigns $f(j)$ correctly.

If no such i exists, i eventually reaches 0 from L7. So L10 assigns $f(j)$ correctly, as the next characters do not match.

In all cases, our algorithm assigns failure function values correctly. This concludes our proof!