1 Introduction

I've been circling around various ideas for proving how some languages such as $\{a^n b^n : n \geqslant 0\}$ are not regular. It's time to bite the bullet and prove that. In doing so we'll come across a really useful characterisation of regular languages which inspires an algorithm for minimising DFAs.

2 Suffix languages

As always, let Σ be an alphabet and $L \subseteq \Sigma^*$ a language. For $w \in \Sigma^*$, define:

$$\mathrm{Suff}(w, L) = \{v \in \Sigma^* : wv \in L\}.$$

This is just the set of suffixes that can occur following w in L. It might well be empty of course, for instance if L is the language NO-ba, then $\mathrm{Suff}(ba, L) = \emptyset$.

Now, associated to any language L let's define an equivalence relation $\sim_L$ on Σ^* as follows:

$$w \sim_L u \quad \text{if and only if} \quad \mathrm{Suff}(w, L) = \mathrm{Suff}(u, L).$$

Since the condition defining $\sim_L$ is an "is the same as" condition it's clear that this is an equivalence relation. Another way to put this is: $w \sim_L u$ if and only if for every word $v \in \Sigma^*$, either both wv and uv are in L or neither is.

3 The Myhill-Nerode theorem

Suppose that L is a regular language and is accepted by some DFA, M. What is $\mathrm{Suff}(w, L)$? It is the language accepted by the DFA which is identical to M except that the start state is the one reached by first processing w through M. In particular, there are only finitely many different possibilities for the set $\mathrm{Suff}(w, L)$ (one per state of M) or, what is the same thing, for the number of equivalence classes of the relation $\sim_L$. That's one half of the *Myhill-Nerode* theorem, whose full form is:

Theorem 3.1. *A language $L \subseteq \Sigma^*$ is regular if and only if the relation $\sim_L$ has only finitely many equivalence classes.*

Proof. One half of the theorem was already proven in the previous paragraph. If L is regular, then there is a DFA, M, that accepts it and then the number of equivalence classes of $\sim_L$ is at most the number of states of M (it might be less if two different states of M would accept the same language as starting states).

For the other half, suppose that L is a language and $\sim_L$ has only finitely many equivalence classes. Each equivalence class is associated with a particular language $\mathrm{Suff}(w, L)$ for some (any) w in the equivalence class. Let these languages be S_0, S_1, $\dots S_{k-1}$ where, for convenience, we choose $S_0 = \mathrm{Suff}(\epsilon, L) = L$. Define an automaton M as follows:

- Its states are (labelled by) S_0, S_1, $\dots$, S_{k-1}.
- S_0 is the initial state.
- For any state S_i choose any w so that $S_i = \mathrm{Suff}(w, L)$. Now for $x \in \Sigma$, define $T(S_i, x) = \mathrm{Suff}(wx, L)$.
- Define the accepting states to be those S_i such that $\epsilon \in S_i$.

As I am fond of saying "this just works".

Suppose that $w \in \Sigma^*$ and let's trace the computation carried out by M on input $w = w_0 w_1 \cdots w_{n-1}$. The first transition takes us to the state $\mathrm{Suff}(w_0, L)$, the second to $\mathrm{Suff}(w_0 w_1, L)$ and so on. When the computation is complete we are in the state $\mathrm{Suff}(w, L)$. Under what circumstances do we accept? Exactly if $\epsilon \in \mathrm{Suff}(w, L)$, i.e., exactly if $w \in L$. So indeed M accepts (exactly) the language L.

The only slight gap at this point is in the definition of T – as presented it's not entirely clear that it might not depend on our choice of w (the mathematical terminology is that it might not be *well-defined*). However, suppose that $w \sim_L v$. I claim that, for any $x \in \Sigma$, $wx \sim_L vx$. This is because:

$$\begin{aligned}
\mathrm{Suff}(wx, L) &= \{y \,:\, wxy \in L\} \\
&= \{y \,:\, xy \in \mathrm{Suff}(w, L)\} \\
&= \{y \,:\, xy \in \mathrm{Suff}(v, L)\} \quad \text{(since } \mathrm{Suff}(w, L) = \mathrm{Suff}(v, L)\text{)} \\
&= \{y \,:\, vxy \in L\} \\
&= \mathrm{Suff}(vx, L)
\end{aligned}$$

Thus, where we "choose w" or "choose v" does not change the value we want to assign to $T(S_i, x)$, ensuring that this value is *well-defined*. □

The power of the Myhill-Nerode theorem is twofold: it gives a great tool to prove that languages are non-regular, but it also shows how to build the smallest possible automaton recognising a given language.

4 Non-regular languages

Let's start with an example we've considered many times now. I claim that the language $L = \{a^n b^n \, : \, n \geqslant 0\}$ is not regular. To see this consider, distinct natural numbers n and m. Since $b^n \in \mathrm{Suff}(a^n, L) \setminus \mathrm{Suff}(a^m, L)$, $a^n \not\sim_L a^m$. Thus, $\sim_L$ has infinitely many equivalence classes and so L cannot be regular.

Many texts, in place of the Myhill-Nerode theorem, prove a weaker statement called the pumping lemma. For completeness, and because it will feature again in the next set of languages we consider here's that result:

Theorem 4.1 (Pumping lemma for regular languages)**.** *Let L be a regular language. There is a positive integer k such that if $w \in L$ and $|w| \geqslant k$ then there are $x, u, y \in \Sigma^*$ with $w = xuy$, $|x| + |u| \leqslant k$, $u \neq \epsilon$, and for all non-negative integers n, $xu^n y \in L$.*

That is, for any sufficiently long word in a regular language we can find a non-empty segment in the middle that can be "pumped" an arbitrary number of times (including removing it entirely) while remaining in the language.

Proof. Choose k to be any positive integer greater than the number of $\sim_L$-equivalence classes. Consider the first k prefixes of w. Two of them, say x and xu belong to the same $\sim_L$-equivalence class. Write $w = xuy$. Since $uy \in \mathrm{Suff}(x, L)$ and $\mathrm{Suff}(x, L) = \mathrm{Suff}(xu, L)$, $uy \in \mathrm{Suff}(xu, L)$, i.e., $xu^2 y \in L$. Inductively, $xu^n y \in L$ for all $n > 0$. But also $y \in \mathrm{Suff}(xu, L) = \mathrm{Suff}(x, L)$ so $xy \in L$ which completes the result. □

Let's use the pumping lemma to reprove that $L = \{a^n b^n \, : \, n \geqslant 0\}$ is not regular. Suppose it were, and choose k as guaranteed by the pumping lemma. Consider $w = a^k b^k$. The appropriate x and u would have to consist entirely of a's (and u is non-empty). But then deleting u (or adding any number of new copies) destroys the balance between the number of a's and the number of b's.

Why do I (strongly) prefer the Myhill-Nerode theorem to the pumping lemma? For several reasons – the theorem really is "about" automata and provides both a necessary and a sufficient condition for regularity. On the other hand the pumping lemma is just "one weird trick". It can only be used to prove that a language is not regular, and the proof technique needed is always proof by contradiction. Finally, if we look again at the proof of the Myhill-Nerode theorem we will discover a fascinating idea about how to discover the smallest possible DFA to recognise a language. That's the topic of the next section.

5 DFA minimisation

Let L be any regular language and let M be any DFA that recognises L. The proof of the Myhill-Nerode theorem implies that M must have at least as many states as the number of equivalence classes of $\sim_L$. Since every state of a DFA can be associated with the suffix language it accepts, and since the transitions between suffix languages are fixed, it also shows that there is a unique *minimal* automaton recognising L having exactly that number of states.

But what if we are just given M? Is there a mechanism for efficiently constructing the minimal DFA accepting the same language as L does? Indeed there is, in fact there are several. I'm going to concentrate on one, Moore's algorithm.

In Moore's algorithm, prior to completion, we will have constructed an equivalence relation (or partition), $\approx$ on the states of the automaton that is (or might be) *coarser* than $\sim_L$. Here we mean by $\sim_L$ the relation on states that holds if the languages accepted from those states are the same (which corresponds to the $\sim_L$ relation on words that get us to those states). That means that if two states are $\sim_L$-equivalent then they will be guaranteed to be $\approx$-equivalent, but there might be $\approx$-equivalent states that are not $\sim_L$-equivalent. We will check this and, if necessary *refine* the relation $\approx$ until we're sure that it agrees with $\sim_L$.

To carry out the refinement procedure we look at each $\approx$-equivalence class and, for each element, q, of that class, and each letter, a, of Σ determine the $\approx$-equivalence class of $T(q, a)$. We associate to q the ordered tuple consisting of those classes. Now we break up the $\approx$-equivalence class into smaller ones determined by those tuples. That is, if the tuples for q and r are the same, they remain $\approx$-equivalent, while if they differ, they are no longer $\approx$-equivalent.

The algorithm terminates when no equivalence class gets broken up (i.e., all the tuples are the same within every equivalence class).

We have as an invariant that two states which are *not* $\approx$-equivalent cannot be $\sim_L$ equivalent (i.e., they have different associated suffix languages). This invariant is preserved by the refinement procedure since splitting two states q and r in the refinement implies that for some letter a, $T(q, a)$ and $T(r, a)$ are not $\approx$-equivalent, hence have different suffix languages. But then the part of q's suffix language consisting of words beginning with a is different from that of r, and hence the suffix language of q and r are not the same.

How does the algorithm begin? Well, there are two types of states that we know have different suffix languages without any further information: the accepting states (their suffix language includes ϵ) and the non-accepting states (their suffix language doesn't). So, that's what we take to be the initial $\approx$ relation.

How can we be sure that when the procedure terminates $\approx$ and $\sim_L$ will be the same relation? Suppose they were not. Since our invariant guarantees that $\sim_L$-equivalent states are $\approx$-equivalent it must be that there are two states q and r that are $\approx$-equivalent but not $\sim_L$-equivalent. Without loss of generality, q accepts some word $w = w_0 w_1 \dots w_{k-1}$ that r rejects. Certainly w couldn't be the empty word otherwise q and r are not even initially $\approx$-equivalent. Now $q_1 = T(q, w_0)$ and $r_1 = T(r, w_0)$ must be $\approx$-equivalent, otherwise q would have been split from r. But q_1 accepts $w_1 \dots w_{k-1}$ and r_1 doesn't. So we can continue this procedure finding that $q_2 = T(q_1, w_1)$ and $r_2 = T(r_1, w_1)$ must be $\approx$-equivalent. And so on inductively – eventually we find that q_k, the state we reach from q by processing all of w and r_k the one we reach from r are supposed to be $\approx$-equivalent. But q_k is accepting and r_k is not! So they aren't! So there's a contradiction![1] And so when the algorithm terminates, $\approx$ is the same as $\sim_L$.

Moore's algorithm is not just theoretically interesting, it's actually quite practical. How-

[1] I hope you appreciate how exciting this is!

ever, Hopcroft's algorithm has better performance guarantees. The reason for these is that it changes how the refinement part of the algorithm works. Underlying Hopcroft's algorithm is a current equivalence relation $\approx$ that is coarser than $\sim_L$ and a collection of $\approx$-classes that I'll call the *splitters*. Initially, as in Moore's algorithm $\approx$ just has two classes - the accepting and non-accepting states, and both these classes are splitters.

Now, so long as the collection of splitters is not empty, we take one, B, and then check for each current $\approx$-class A and each letter a whether the images of A under a are split by B. In the event that they are split, we refine the $\approx$-relation so that A is split into its two parts A_1 (say elements that are taken into B on an a-transition) and A_2 (elements that are taken elsewhere). Then, we might think we need to add both the newly-created classes as possible splitters for later, but it turns out this is only necessary if A was already in the list of splitters prior to this step (and in that case, we can replace A in the list of splitters by both A_1 and A_2). Otherwise, it suffices to add the smaller of the two classes as a potential splitter.

Using the *partition refinement* data structure (which is a dual version of the *union find* data structure) Hopcroft's algorithm can be implemented with a run-time bound of $O(n|\Sigma| \log n)$ where n is the number of states of the original automaton M. Since the space requirement for even representing such an automaton is $\Theta(n|\Sigma|)$ that's more than satisfactory.