Notes on Theory of Computation Last updated: August, 2021

Languages, regular languages, finite automata

Content largely taken from Richards [1], Sipser [2], and Dov Gordon's notes

1 Languages

An *alphabet* is a finite set of characters, which we will often denote by Σ. For example, $\Sigma = \{0, 1\}$ is an alphabet that we will frequently use. A *language* over some alphabet Σ is a set of strings made up of characters from Σ. For example, $L_1 = \{0, 1, 00, 11\}$ is a language over the alphabet $\Sigma = \{0, 1\}$. An English dictionary is also a language, over the alphabet $\{a, \ldots, z, A, \ldots, Z\}$. However, languages need not be finite size: "the set of all binary strings ending in 0" is a language over $\Sigma = \{0, 1\}$. Clearly such a language is not as easy to formally describe, but we will address that issue later on.

It is useful to define a few set operators for languages. The *union* operator, $\cup$, is defined as with any other collection of sets. The *concatenation* operator, $||$, is so natural, we will often omit the operator all together: $L_1||L_2 = L_1L_2 = \{xy \mid x \in L_1 \wedge y \in L_2\}$. For any language L, we define $L^0 = \{\Lambda\}$, where Λ is a special string, called the empty string. In other words, L^0 is the language consisting only of the empty string. For $k > 0$, we recursively define $L^k = LL^{k-1} = \{xy \mid x \in L \wedge y \in L^{k-1}\}$. That is, we concatenate L with itself k times (and also include the empty string). Finally, we define the closure operator (sometimes called *star* operation): $L^* = \cup_{i=0}^{\infty} L^i = L^0 \cup L^1 \cup L^2 \cup \ldots$

At a high level, the fundamental question of theory of computation is the following: Given a language L and some string x, how hard is it to determine whether $x \in L$? (Or, as we will often phrase this question, how hard it is to *decide* the language L?) We all have some intuition of what this means. For example, given a string of english characters, one can determine whether it is a valid English word by scanning through the English dictionary, one word at a time. But anyone that still uses a paper dictionary knows that they can do better using binary search, and we may even have seen a proof that you require $O(\log n)$ comparison for a dictionary of size n. Such algorithmic questions aren't really our focus in this class, though. Rather, we are interested in characterizing whole *classes* of languages.

A language that can be decided in polynomial time on a Turing machine is said to be in a class of languages that we call $\mathcal{P}$ (more on this later). But are there languages that are fundamentally easier to decide than these? Are there languages that can be proven to be strictly harder to decide than those in $\mathcal{P}$? And, furthermore, why is the Turing machine the right model for computation? What if we consider other models? And why is time the right metric: how do memory and communication constraints impact what we can compute? Does access to good random sources help us to decide more languages? Moreover, why is deciding whether some string $x \in L$ the right place to focus our attention? We will look at almost all of these questions, and more, with the aim of gaining a deeper fundamental understanding of what is possible in our field and what is not.

2 Finite Automata

2.1 Regular Languages

To begin, we start with a very simple model of computation, and a simple class of languages, which are called the regular languages. The *regular languages* correspond to languages generated by regular expressions. Jumping ahead, this is one way to define regular languages, we will see later in class that regular languages can be defined with respect to deterministic finite automata. We formalize the class of regular languages recursively, as follows. Let $\mathcal{R}$ denote the set of all regular languages over some alphabet, Σ. Then, we have:

1. $\emptyset \in \mathcal{R}$ and $\{\Lambda\} \in \mathcal{R}$.
2. $\forall \sigma \in \Sigma : \{\sigma\} \in \mathcal{R}$.
3. If $L \in \mathcal{R}$ then $L^* \in \mathcal{R}$.
4. If $L_1 \in \mathcal{R}$ and $L_2 \in \mathcal{R}$ then $L_1 L_2 \in \mathcal{R}$.
5. If $L_1 \in \mathcal{R}$ and $L_2 \in \mathcal{R}$ then $L_1 \cup L_2 \in \mathcal{R}$.

We note here that $\emptyset$ is the empty set with cardinality zero, while $\{\Lambda\}$ is the set that contains the empty string, and as a result, it has cardinality one. Property 4 indicates that the class of regular languages in closed under the concatenation operation, whereas, property 5 indicates that the it is closed under the union operation.

Regular languages are so common and useful, that we frequently use a special notation, called *regular expressions* in order to define these languages. A useful analogy is the following, operations '+' and '$\times$' and '$-$' are used to define mathematical expressions where the output is a number, while operations '$\cup'$ and '$||'$ and '$*$' are used to define regular expressions where the output is a language. For brevity, when defining regular expressions, the '{' and '}' are dropped, and union is denoted by '+'.

For example, let's break down the regular language that comes from the regular expression $(ab + c)^*$. Let L be the regular language that is defined by the regular expression $(ab + c)$. First we have $L^0 = \{\Lambda\}$. Then, we have $L^1 = LL^0 = L = \{ab, c\}$. Then, we have $L^2 = LL^1 = \{abab, abc, cab, cc\}$. Then, we have $L^3 = LL^2 = \{ababab, ababc, abcab, abcc, cabab, cabc, ccab, ccc\}$. Thus, we define $(ab + c)^* = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \ldots$

2.2 Deterministic Finite Automata (DFAs)

A *deterministic finite automaton* is a state machine that takes an input string and either accepts or rejects that string. It makes this decision by transitioning through a sequence of states, making exactly one transition for each character of the input string in a deterministic way. After the transitions are complete, it accepts if it has terminated in a state marked "accept", and it rejects if it has stopped in a state marked "reject". We will formalize this model of computing in a moment, but it is helpful to first demonstrate it by example. In the first state diagram in Figure 1, there is a special start state, labeled 'A', and the state labeled 'C' has a circle around it, denoting that it is an accept state. We note that there can be multiple accept states, though that isn't the case in these two examples.

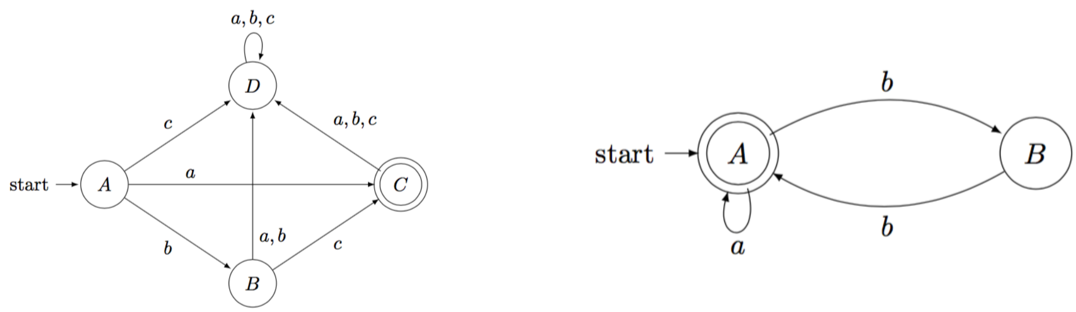

Figure 1: Two examples of DFAs (taken from Richards [1])

Consider input string ‘bc’: the DFA reads the first character, and transitions from state ‘A’ to state ‘B’. It then reads the second character and transitions into the accept state, ‘C’. Because this is the last character of input, the machine terminates in the accept state, and we say that the machine accepts input ‘bc’. Equivalently, we say that ‘bc’ is in the language of this DFA.

Consider now input ‘bcc’: the last character causes a transition out of the accept state and into state ‘D’, where the machine now terminates. Because ‘D’ is not marked as an accept state, we say that the DFA rejects this input, or that the input is not in the language of this DFA. Looking more closely at state ‘D’, you can see that it is a sink: there are no transitions out of ‘D’, so no input that leads to state ‘D’ will ever be accepted. This is sometimes called a *trap state*, and usually we will leave such states out of our diagrams in order to simplify them. Removing ‘D’ and all of its edges, we will sometimes find that while processing an input string, there is no transition that can be made. In this case, we interpret this though we had transitioned to a trap state, and say that the machine rejects the input.

The language of the second DFA in Figure 1 is the language of the regular expression $(a + bb)^*$. (Written as a regular language, this would be L^*, where $L = \{a\} \cup \{bb\}$.) Note that Λ is in this language, by the definition of the closure operator; to see why Λ is accepted by the DFA, note that the start state is also an accept state. We will shortly show that all regular languages can be decided by DFAs. Actually, we will show a stronger statement: that the class of regular languages is *equivalent* to the class of languages decided by DFAs.

2.3 Formally Defining DFAs

Formally, a deterministic finite automaton M is defined by an *alphabet* Σ, a finite set of *states*, Q, a *start state*, $S \in Q$, a set of *accept states*, $\mathcal{A} \subseteq Q$, and a *transition function* $\delta : Q \times \Sigma \to Q$. Putting this together, $M = (\Sigma, Q, S, \mathcal{A}, \delta)$. Returning to the first example in Figure 1 (and ignoring the trap state D), this DFA can be formally described by $(\Sigma = \{a, b, c\}, Q = \{A, B, C\}, A, \mathcal{A} = \{C\}, \delta)$, where δ is defined as:

$$\delta = \begin{array}{c|c|c|c|} & a & b & c \\ \cline{2-4} A & C & B & \perp \\ \cline{2-4} B & \perp & \perp & C \\ \cline{2-4} C & \perp & \perp & \perp \\ \cline{2-4} \end{array}$$

Notice here that the transition function outputs $\perp$ if the DFA does not consider the corresponding transition, e.g., $\delta(A, c) = \perp$, in which case the state machine rejects the input.

The formalism will help us to prove things about DFAs and the languages that they decide. Consider a language L that contains a single element of the alphabet, we will show that there exists a DFA that decides L. Formally, for any alphabet Σ, and any $x \in \Sigma$, we can see that there exists a DFA M deciding language $L = \{x\}$ that is defined as $M = (\Sigma, Q = \{A, B\}, A, B, \delta)$, where $\delta(A, x) = B$, and δ otherwise has output $\perp$. Informally, the DFA has a start state A that is non-accepting, and a single accept state B. The only transition allowed goes from the start state A to the accept state B on input x. The notation $L(M)$ denotes that state machine M decides language L. We can also define a DFA for $\{\Lambda\}$: it has a single state that is both the start state and an accept state, and it does not allow any transitions. So we can see that the languages defining the "base-cases" of the regular languages are all decidable by DFAs.

We now show that if a language L_1 is decided by DFA M_1, and a language L_2 is decided by DFA M_2, then there exists a DFA M that decides $L_1 \cup L_2$. Intuitively, we construct M so that it tracks the movement of the input through *both* M_1 and M_2, simultaneously. To do that, we create $|Q_1| \cdot |Q_2|$ states, and label each with a pair of names, one from Q_1 and one from Q_2. For example, if $A \in Q_1$ and $B \in Q_2$ then we create a state '(A, B)' for DFA M. If M is in state (A, B), we can think of this as indicating that M_1 would currently, on this input, be in state A, while M_2 would currently be in state B. If M halts in state (A, B), we want to accept if either A is an accept state for M_1, or if B is an accept state for M_2.

To simplify the formal exposition, we'll assume M_1 and M_2 share the same alphabet; it is easy to see that this isn't necessary. Let $M_1 = (\Sigma, Q_1, S_1, \mathcal{A}_1, \delta_1)$ and let $M_2 = (\Sigma, Q_2, S_2, \mathcal{A}_2, \delta_2)$. We'll also assume, without loss of generality, that both machines have a trap state. Then $M = (\Sigma, Q, S, \mathcal{A}, \delta)$ is defined as follows: $Q = \{(A, B) \mid A \in Q_1 \wedge B \in Q_2\}$, $S = (S_1, S_2)$, $\mathcal{A} = \{(A, B) \mid A \in \mathcal{A}_1 \vee B \in \mathcal{A}_2\}$, and $\delta((A, B), x) = (\delta_1(A, x), \delta_2(B, x))$.

To prove that $L(M) = L_1 \cup L_2$, we must show two things, the first direction (i.e., $L(M) \Rightarrow L_1 \cup L_2$) is to prove that that if $w \in L(M)$, then $w \in L_1 \cup L_2$ and the other direction (i.e., $L(M) \Leftarrow L_1 \cup L_2$) is to prove that if $w \in L_1 \cup L_2$ then $w \in L(M)$.

First, we prove that if M accepts w, then $w \in L_1 \cup L_2$. We will write $w = w_1 \cdots w_k$, letting w_i denote the ith character of w. Note that $w \in L(M)$ implies that there is a sequence of states in M, $(S_1, S_2), (A_1, B_1), (A_2, B_2), \ldots, (A_k, B_k)$ such that $\delta((S_1, S_2), w_1) = (A_1, B_1)$, $\delta((A_i, B_i), w_{i+1}) = (A_{i+1}, B_{i+1})$, and either $A_k \in \mathcal{A}_1$, or $B_k \in \mathcal{A}_2$. Without loss of generality, let's assume that $A_k \in \mathcal{A}_1$. It follows, by the way M was constructed, that $\delta_1(S_1, w_1) = A_1$, and, for $i \in \{1, \ldots, k-1\}$, $\delta_1(A_i, w_{i+1}) = A_{i+1}$. Since $A_k \in \mathcal{A}_1$, it follows that M_1 accepts w, and that $w \in L_1 \cup L_2$. Secondly, we must show that if $w \in L_1 \cup L_2$, then M accepts w. We leave this direction as an exercise. We will later come back to the other regular operators, closure and concatenation.

2.4 Non-deterministic Finite Automata (NFAs)

We consider a very useful relaxation in how we model finite automata. Although it was not made explicit, we previously did not allow any ambiguity in how our transitions were to be made: for any state A and any input character x, we have, so far, allowed only a *single* transition from A to be labeled with x. Relaxing that gives us a lot more flexibility in our design. Consider the two examples in Figure 2, again taken from Richards [1]. Both machines decide the same language: $\{w \in \{a, b\}^* \mid w \text{ ends in } ab\}$. The first one is a deterministic finite automation. The second example, which is non-deterministic, has an ambiguous transition out of the start state: on input 'a', the machine has a choice to make. It could either transition to state q_1, or it could stay in

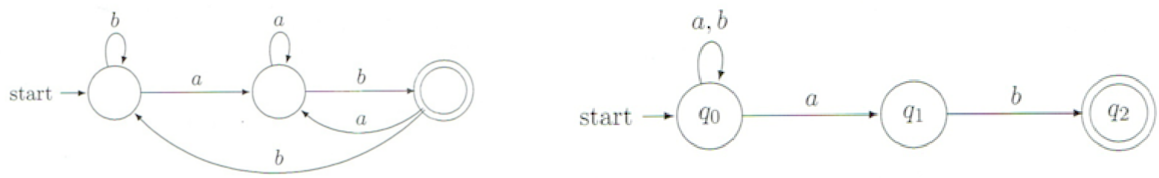

Figure 2: An DFA and an NFA that decide the same language (taken from Richards [1])

the start state. We say that this machine accepts an input if there exists some sequence of allowable transitions that ends in an accept state. Importantly, we only require *the existence* of some such sequence of transitions: we do not require that all allowable transitions result in acceptance, and we do not care about *how* one might find such a sequence.

An even better example of where non-determinism helps ease the design of a finite automata is the following language: $L = \{x \in \{a,b\}^* \mid \text{the } k\text{th symbol from the last is `a'}\}$, where k is some fixed integer. We described an NFA for this language in class, and we will design a DFA for this language in the homework.

To formally define NFAs, we have to change the definition of our transition function. Whereas in DFAs, we have $\delta : Q \times \Sigma \to Q$, we now have to allow δ to map the same domain to a *set of states*, rather than to a single state. Formally, $\delta : Q \times \Sigma \to 2^Q$, where 2^Q denotes the power-set. Looking again at example 2 in Figure 2, we have

$$\delta = \begin{array}{c|c|c|} & a & b \\ \hline q_0 & \{q_0, q_1\} & \{q_0\} \\ \hline q_1 & \perp & \{q_2\} \\ \hline q_2 & \perp & \perp \\ \hline \end{array}$$

Additionally, it is helpful to allow Λ transitions. These transitions allow the machine to move from one state to another without using up any of the input string. We don't bother to formalize this.

2.5 Equivalence of DFAs and NFAs

How much additional power does this non-determinism give us? It seems to make machine design a lot simpler, but does it allow us to decide a larger class of languages? It turns out that it does not: the set of languages decidable by NFAs is exactly the regular languages, just as for DFAs. We prove now that the two models are equivalent in this sense.

It is clear from the definitions that every DFA is also an NFA, so we only need to show that for every NFA, $M = (\Sigma, Q, q_0, \mathcal{A}, \delta)$, there exists a DFA, $M' = (\Sigma, Q', S', \mathcal{A}', \delta')$, such that $L(M') = L(M)$. The intuition is similar to the one above for showing a DFA that decides the union of two languages. We will create a new state for every possible subset of Q. For example, if we have $|Q| = k$ states in the NFA, then we create 2^k states for its equivalent DFA. To illustrate this point, let's define DFA's state '$\{q_1, q_5, q_7\}$'; this state captures the fact that M with input x can follow paths that lead to state q_1, q_5, or q_7. We emphasize here that a single state in this DFA essentially represents multiple states from the NFA. In this way, our DFA will keep track of all the possible places we could currently be in the NFA, given the input string seen so far. Furthermore, the state $\{q_1, q_5, q_7\}$ is an accept state if *any* of q_1, q_5, or q_7 are accept states for M. Formally, for DFA M'

we have $Q' = 2^Q$, $S' = \{q_0\}$, $\mathcal{A}' = \{T \in Q' \mid \exists t \in T \text{ s.t. } t \in \mathcal{A}\}$, and $\delta'(T,x) = \bigcup_{q\in T} \delta(q,x)$. Remember here that T is a single state of the DFA that represents a set of states from the NFA, thus, the expression $\delta'(T,x) = \bigcup_{q\in T} \delta(q,x)$ is a transition from a single state (i.e., T) of the DFA to a single state (i.e., $\bigcup_{q\in T} \delta(q,x)$) of the DFA.

We need to prove two things: 1) For $w = w_0 \cdots w_{k-1}$, if $w \in L(M)$, then $w \in L(M')$, and 2) if $w \in L(M')$, then $w \in L(M)$. We start with the first statement, suppose M accepts w, i.e., $w \in L(M)$. Because M is an NFA, we know that there exists some sequence of states, $q_0, q_1, \ldots, q_k$, such that, $q_{i+1} \in \delta(q_i, w_i)$, and the last state is an accept state, i.e., $q_k \in \mathcal{A}$. We switch our attention now to the corresponding DFA M' with the same input w. Let $T_0, T_1, \ldots, T_k$, be the states in DFA M' such that for each $i \in \{0 \ldots, k\}$, $T_{i+1} = \delta'(T_i, w_i)$. We want to show that these transitions lead to an accept state of the DFA, i.e., $T_k \in \mathcal{A}'$. To show this, we first argue that $q_i \in T_i$, that is, the state from the NFA is an element from the 'set-state' of the DFA. We proceed with an inductive argument. This clearly holds for q_0, since $T_0 = S' = \{q_0\}$ (base case). Suppose that indeed $q_i \in T_i$ (inductive hypothesis), and recall that by the definition of δ', $T_{i+1} = \bigcup_{q\in T_i} \delta(q, w_i)$. From the definition of T_{i+1} we have:

$$T_{i+1} = \delta'(T_i, w_i) = \bigcup_{q\in T_i} \delta(q, w_i) = \delta(q_i, w_i) \cup \left\{ \bigcup_{q\in T_i} \delta(q, w_i) \right\} = q_{i+1} \cup \left\{ \bigcup_{q\in T_i} \delta(q, w_i) \right\},$$

where for the third equality we used the inductive hypothesis (that is $q_i \in T_i$) and for the fourth equality we used the definition of q_{i+1} (that is $q_{i+1} \in \delta(q_i, w_i)$). The above steps show that $q_{i+1} \in T_{i+1}$. Finally, since $q_k \in T_k$, and $q_k \in \mathcal{A}$, it follows that $T_k \in \mathcal{A}'$.

We now need to prove that if $w \in L(M')$, then $w \in L(M)$. Using the same notation, let $T_0, \ldots, T_k$ be the sequence of states of the DFA such that $T_{i+1} = \delta'(T_i, w_i)$. We have to show that given input w there exists some sequence of states from Q of the NFA, $q_0, \ldots, q_k$, such that $q_{i+1} \in \delta(q_i, w_i)$, and $q_k \in \mathcal{A}$. To show this, we will use the 'set-states' $T_0, \ldots, T_k$ of the DFA M' and the fact that M' accepts w. We start by fixing q_k and work backwards to q_0. To choose q_k, we note that because $T_k \in \mathcal{A}'$, then by definition of $\mathcal{A}'$, there is some $q_k \in T_k$ such that $q_k \in \mathcal{A}$. Let's pick any such q_k. For $i < k$, assume q_{i+1} has already being fixed. Since $T_{i+1} = \bigcup_{q\in T_i} \delta(q, w_i)$, there exists some $q_i \in T_i$ such that $q_{i+1} \in \delta(q_i, w_i)$. Choose any such q_i, and repeat. Since $T_0 = \{q_0\}$, we can (and must) choose q_0 as our start state. This concludes the proof.

Actually, technically, we also have to show how to handle Λ-transitions when constructing M'. This is easily done. For each $q \in Q$, let $E(q)$ be the set of states that is reachable using only Λ-transitions. Then, instead of defining $\delta'(T,x) = \bigcup_{q\in T} \delta(q,x)$, we define it as $\delta'(T,x) = \bigcup_{q\in T} (\delta(q,x) \cup E(q))$. The rest of the proof would proceed as before.

2.6 Equivalence of Regular Languages and DFAs

Using NFAs, it becomes much easier to show that all *regular* languages can be decided by a DFA, i.e. the direction of the proof regular⇒DFA. We leave it as a homework problem to show that

1. if a language L is decidable by a DFA, then L^* is decidable by some NFA, M.
2. if L_1 and L_2 are decided by DFAs M_1 and M_2, then $L = L_1 L_2$ is decidable by some NFA, M.

To complete the equivalence proof (that is, the class of regular languages and the class of languages decidable by DFAs are the same) we must also show that every language that is decidable by a DFA is regular, i.e., the direction DFA⇒regular. This is not a difficult proof, but we omit it in this class so that we can move on to other interesting things.

2.7 Some Languages Are Not Regular

There are some languages that cannot be decided by any finite automaton. To demonstrate this, we first prove a useful lemma that is famously known as the pumping lemma.

Lemma 1 ***(Pumping Lemma)*** *If L is a regular language, then there exists a number p (the pumping length) such that for any $w \in L$ with $|w| > p$, w can be divided into 3 strings, $w = xyz$ such that:*

1. $\forall i \geq 0, xy^i z \in L$
2. $|y| > 0$
3. $|xy| \leq p$

Proof Since L is regular, we know that there exists some finite automaton M that decides L. We define p to be the number of states in M. Let n be the total length of the input string. For $w = w_1, \ldots, w_n$, let $q_0, \ldots, q_n$ be the sequence of states that lead from start to accept on string w. Because $n > p$ (by assumption in the lemma statement), there must be some state in this sequence that is repeated (by the pigeonhole principal). We'll call the first such state q^*. Let q_s be the first appearance of q^* in our state sequence, and let t be the index of the first repetition of q^*. That is, $q_s = q_t = q^*$: they represent the same states, but appear in different places on this list. Then we define $x = w_1 \cdots w_s$, $y = w_{s+1} \cdots w_t$, and $z = w_{t+1} \cdots w_n$. An example of this partition is illustrated in Figure 3.

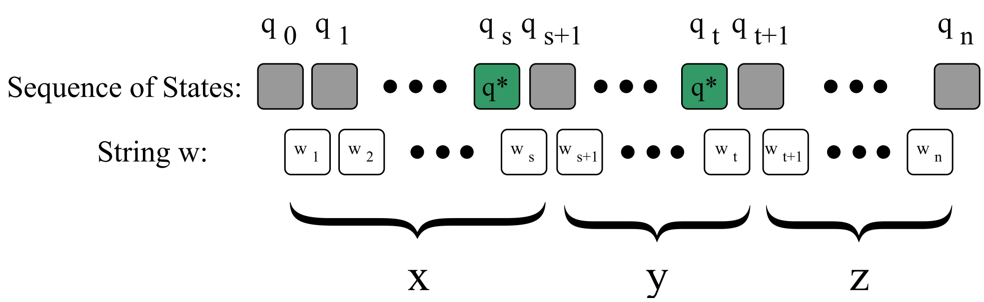

Figure 3: An illustration of how we define the three strings x, y, and z using the fact that a state q^* is repeated.

We now show that the three properties of the lemma are satisfied. For the first property, let's consider $i = 0$, so that we have input string $xz = w_1, \ldots, w_s, w_{t+1}, \ldots, w_n$. We know that using the first s characters the automaton will reach state q_s, since these are the same characters in the original input, w. Furthermore, since $q_s = q_t$, we know that the string $w_{t+1}, \ldots, w_n$ will transition the automaton through states $q_{t+1}, \ldots, q_n$, and will eventually reach the same accept state that results from processing the original string w. For $i = 1$, it is true by assumption that $xyz \in L$. For $i > 1$, we claim that at the end of each repetition of string y, we end in state q_t. This is certainly true at the end of the first repetition of string y, by the way we defined state q_t. Since $q_t = q_s$, the next repetition of string y transitions through states $q_{s+1}, \ldots, q_t$, just as the first occurrence of string y did. Since the last repetition of string y leaves us in state q_t, it follows that string z transitions the automaton to accept state q_n.

For the second property, it follows immediately that $|y| > 0$ from the definition of y. For the third property, recall that q_t is the first state to be repeated in the transition sequence. This implies that all t states, i.e., $q_0, \ldots, q_{t-1}$ are unique. Suppose, for the sake of contradiction, that $t > p$, then it follows that there must be more than p states in automaton M, which violates our definition of p, contradiction. ∎

We now use the pumping lemma to show that $L = \{a^n b^n | n \geq 0\}$ is not regular. Suppose, for the sake of contradiction, that L is regular, and let M be the deterministic finite automaton that decides it. Since we assumed that L is regular, from Lemma 1, we know that there exists a pumping length p for which the pumping lemma holds. By the pumping lemma, we know that *any* string $w \in L$ with $|w| > p$ can be written as xyz such that y can be "pumped". The important part here is the word "any"; in the rest of the proof we will come up with a string that is longer than p but violates the pumping lemma. We define the value j as $j = \lceil p/2 \rceil$ and we will show that $a^j b^j$ cannot be pumped. Specifically, regardless of how y is chosen for this string, $xy^2z \notin L$. To prove this we have to perform a case analysis and consider the following three cases:

1. String y is defined so as it contains only a values. In this case, xy^2z has more as than bs. Therefore xy^2z is not in language L.

2. String y is defined so as it contains only b values. In this case, xy^2z has more b values than as. Therefore xy^2z is not in language L.

3. String y is defined so as it contains a number of as and a (potentially different) number of bs. Then note that xy^2z has a substring in which some as come after some bs. Therefore xy^2z is not in language L.

We note that we could have also considered $j = p$, and the argument would have been simpler. But this argument is more "interesting", and demonstrates a proof by *case analysis*.

An important thing to pay attention to in the proof above is the *ordering of quantifiers*. The pumping lemma says that if L is regular, then $\exists p$ such that $\forall w, |w| > p$, $\exists xyz = w$ where x, y,and z satisfy the conditions of the lemma. To show that a language is NOT regular, we have to show that this statement is NOT true. That is, we have to show that $\forall p$, $\exists w, |w| > p$ such that $\forall xyz = w$, x, y, and z fail to satisfy the criteria of the lemma.

If you revisit the proof you will see that we did not structure the proof argument based on a specific value of p. The pumping lemma says that there exists a p but does not provide a way to find its value. Therefore, in order to prove that the statement is NOT true, we have to build an argument that holds for *every possible* value of $p \geq 1$. Notice that no matter what value p is given to us, in our proof we construct a w, i.e., $w = a^{\lceil p/2 \rceil} b^{\lceil p/2 \rceil}$, such that *every possible* splitting up of w contradicts one of the properties of Lemma 1.

3 Pushdown Automata

Pushdown automata are very similar to finite automata, but we equip the state machine with a *stack* for reading and writing data. The pushdown automaton still operates by scanning the input, left to right, one character at a time. The automaton terminates when it has read the last character of the input. Transitions are labeled with expressions of the form "$a, b/c$" where $a \in \Sigma$ is a value of the input, and $b, c \in \Gamma$, where Γ, called the "tape alphabet", is the set of characters that can pushed and popped from the stack. It is reasonable to assume that $\Sigma \subseteq \Gamma$. The notation b/c means that you can take this transition if the character at the top of the stack is b, and, in doing so, you replace character b with character c. Note that you can only take a transition $a, b/c$ if the next character of the input is a AND the character at the top of the stack is character b. If we don't wish to put anything new onto the stack, we can use a transition of the form $a, b/\Lambda$. In this case, we would pop character b from the stack, and the number of elements on the stack would be reduced by one. If we wish to only add something to the stack, we can use a transition of the form $a, \Lambda/c$ and the number of elements on the stack would be increased by one. We can also ignore the input and just operate on the stack, which is denoted by $\Lambda, a/b$. We can ignore both the input and also not pop anything, denoted by $\Lambda, \Lambda/c$. Such transitions can be taken regardless of the values of the next input character and the character at the top of the stack, i.e., they only push a character. We also allow to push multiple characters onto the stack at once (though we do not allow multiple pops at once). For example, $a, b/bc$ would pop 1 b, push 1 b, and then push 1 c. The top (resp. bottom) character of the stack is the leftmost (resp. rightmost) character when the contents of the stack are represented as a sequence. For example, when we say that the content of the stack is abc, then a is at the top of the stack, right under it is b, and at the bottom of the stack we have c.

Empty stack: We don't have any explicit mechanism for testing the stack to see if it is empty. Instead, if that is something we care to do, we can create a transition at the start that pushes a special symbol onto the stack, and we can later interpret as an indicator that the stack is empty. This can be seen in Figure 4 below, where \$ plays that role. Note that we do not read an input character in that transition and we do not pop anything; we only start processing input after we've initialized our stack by pushing \$.

Termination: The automata terminate when the last character of the input is read. It accepts if and only if it terminates in an accept state. Just as with finite state automata, we assume there is a trap state for rejecting that is not made explicit: if it is ever impossible to make a transition, and there is still input that hasn't been processed, then the machine is assumed to transition into a reject state and to stay there. Note that in the case where we ignore the input tape, we also delay termination by one transition. So, we can take many transitions of the form $\Lambda, a/b$, and these transitions do not "consume" any of the input. We will next walk through the example in Figure 4 below, which will demonstrate this point.

Example: Much like the previously defined finite automata which can be categorized as DFA and NFA, the pushdown automata can also be deterministic (denoted as DPDA) or non-deterministic (denoted as NPDA). We walk through the NPDA in Figure 4, using input $abba$.

- There is only one transition we can take from the start state. We transition to state *Even*, the input tape still holds $abba$, and the stack now holds \$.
- Since the first input character is a, the only legal transition is to $>_a$. The input tape holds bba and the stack holds $a\$$.
- Since the top of the stack is now a, the only legal transition is $b, a/\Lambda$, which leaves us in the same state. (Note we cannot take the transition labeled $\Lambda, \$/\$$ yet, because the top of the

stack we had character a.) This transition removes the a at the top of the stack. The input tape now holds ba, and the stack now holds \$.

- The only legal transition is the one labeled $\Lambda, \$/\$$, to state *Even*. The remaining input is *still* ba, since the Λ in that transition does not use up an input character. This transition pops and pushes \$, so the stack still holds \$.
- As a next step, we transition to state $>_b$, the input tape holds a and the stack holds $b\$$.
- We transition using $a, b/\Lambda$, remaining in the same state. The input tape is now empty, and the stack now holds \$.
- We now have a choice to make. We can terminate and reject, or we can transition one more time to state *Even* using $\Lambda, \$/\$$ and then accept. Recall that the definition of non-determinism says that a string is in the language as long as there *exists some sequence of choices* that leads to accept. So, in this case, the string is in the language.

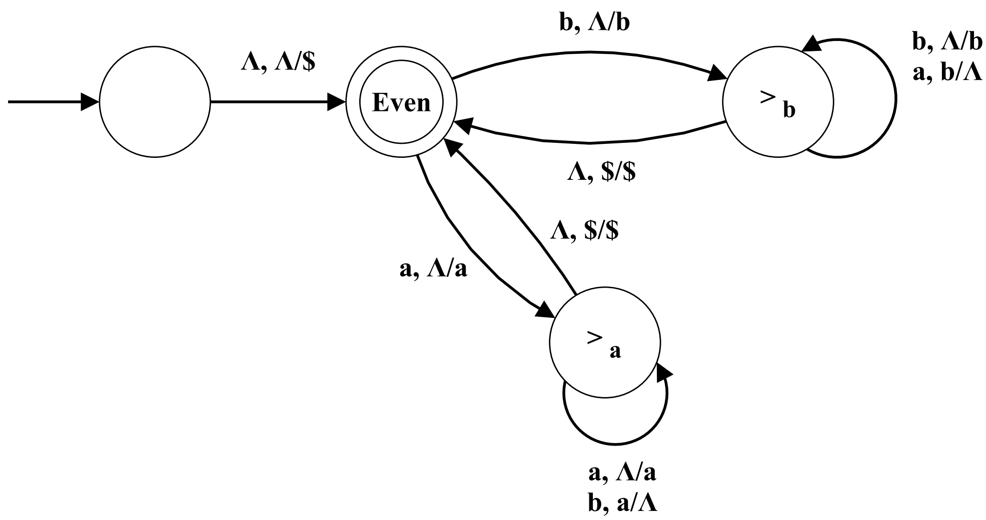

Figure 4: M_L accepting language $L = \{(a+b)^* \mid \text{ there are an equal number of } a\text{s and } b\text{s}\}$

The Power of PDAs: Overall, the above execution of the NPDA shows the power of this new computational model which introduces a stack of infinite memory. It is worth noting here that even though the memory is infinite, we can only access a single location at a time, i.e., the top of the stack. We will later see an even more powerful computational model where we can access more memory. Let's take a step back and see why the stack is crucial in this example. In case we have seen character a more times than b in the input tape (that is, NPDA is in state $>_a$), then the stack acts as a "counter" that keeps track of how many more times we have seen a. We cannot do this "accounting" with the previously introduced finite automata. Thus, with this new model at our disposal, we can even recognize some non-regular languages.

3.1 Formal Notation for Non-deterministic Pushdown Automata

A NPDA can be denoted by $(Q, \Sigma, \Gamma, \delta, q_0, Q_A)$, where Q is the set of states, Σ is the input alphabet, Γ is the tape alphabet (which might contain Σ), δ is a transition function, detailed below, q_0 is a special start state, and $Q_A \subseteq Q$ is a set of accept states. The function δ maps a state, an input character, and a character read from the stack, to a state and a sequence of characters to be written to the stack. However, in the non-deterministic case, note that it might map the same input onto multiple outputs. We therefore let the co-domain be the power set of $Q \times \Gamma^*$. Formally, then, δ is a function $\delta : Q \times \Sigma \times \Gamma \to 2^{Q \times \Gamma^*}$.

A machine accepts string w if and only if w can be written as $w_1 w_2 \cdots w_n$, where each $w_i \in \Sigma \cup \{\Lambda\}$, and there exists a sequence of states $r_0, r_1, \ldots, r_n$, $r_i \in Q$, and a sequence of strings $s_0, \ldots, s_n$, $s_i \in \Gamma^*$, such that

1. $r_0 = q_0$, $s_0 = \Lambda$, and $r_m \in Q_A$.
2. $\forall i \in \{1, \ldots, n\}$, $\exists \alpha, \beta \in \Gamma \cup \{\Lambda\}, \gamma \in \Gamma^*$, such that $s_{i-1} = \alpha\gamma$, $s_i = \beta\gamma$, and $(r_i, \beta) \in \delta(r_{i-1}, w_{i-1}, \alpha)$

Intuitively, $w_1 \cdots w_n$ denote the input string, but possibly "padded" with internal Λ values to account for places that we might take a transition that doesn't read any input. The first condition states that we start in the start state with an empty stack, and we terminate in an accept state. The second condition says that we transition through some valid sequence of states, maintaining valid stack content.

Specifically, we maintain the validity of the stack content by guaranteeing that when we transition from state r_{i-1} to state r_i (via $\delta(r_{i-1}, w_{i-1}, \alpha)$), there is indeed a string $s_{i-1} = \alpha\gamma$ that represents the content of the stack at the moment for which the top of the stack (i.e., the leftmost character of s_{i-1}) matches the expectation of the transition function δ (i.e., its third input). The above notation guarantees that we pop correctly; a similar condition holds for pushing character β to the stack.

3.2 NPDAs and DPDAs Are Not Equivalent

Unlike in the case of DFAs and NFAs, non-determinism in the case of push-down automata does in fact increase the expressiveness of the model. That is, there are language that can be decided by a NPDA that cannot be decided by a DPDA. An example of such a language is $L = \{a^n b^n \mid n \geq 0\} \bigcup \{a^n b^{2n} \mid n \geq 0\}$. We leave it as an exercise to show that L can be decided by an NPDA. Also, we do not prove in this class that there is a pumping lemma, similar to the one for regular languages, which shows that certain languages cannot be decided by NPDAs. One language that cannot be decided by any PDA is $L' = \{a^n b^n c^n \mid n \geq 0\}$.

Returning back to the original point, we want to prove that $L = \{a^n b^n \mid n \geq 0\} \bigcup \{a^n b^{2n} \mid n \geq 0\}$ cannot be decided by any DPDA. To show this we prove that if there was a DPDA that decides L then we can use it to construct a PDA for L', but, as we know, this is impossible; therefore, there is no DPDA that decides L. Since we know that L can be decided by an NPDA, we have shown that the deterministic PDAs are less powerful than nondeterministic PDAs.

We now see this proof in more detail. Suppose that there is a DPDA M that decides language L. Let $M_1 = (\{a, b\}, Q_1, q_0, \mathcal{A}_1, \delta_1)$ and $M_2 = (\{a, b\}, Q_2, S_2, \mathcal{A}_2, \delta_2)$ be identical copies of the machine that decides L, but we *relabel* each state so that the copy of any given state from M_1 can be distinguished from the copy of the same state from M_2. The next step is to construct a PDA M' deciding L' (which we know it is not possible). We start with $M' = (\{a, b, c\}, Q_1 \cup Q_2, q_0, \mathcal{A}_2, \delta')$, where, for $x \in \{a, b\}$, $\forall q_1 \in Q_1$, $\delta'(q_1, x) = \delta_1(q_1, x)$, and $\forall q_2 \in Q_2$, $\delta'(q_2, x) = \delta_2(q_2, x)$. Notice that the start state of M' is the start state of M_1 and that the accept states of M_2 are (only) the

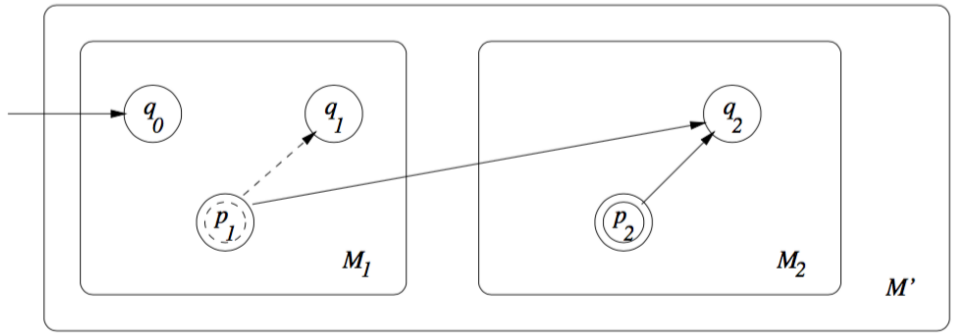

Figure 5: Machine M', deciding language $L' = \{a^n b^n c^n \mid n \geq 0\}$, which we know to be impossible. Figure is taken from the proof of Theorem 14.2 of [1].

accept states of M_2. With respect to the transition, the new transition function δ' imitates (at least for now) the transitions of functions δ_1 and δ_2 on their corresponding and relabeled states. But we are not done with the transition function δ', we need to find a way to connect the two DPDAs. As a next step, we make the following two modifications to δ' in order to define its behavior on inputs of type c.

1. For each accept state $p_1 \in \mathcal{A}_1$, let $q_1 = \delta(p_1, b)$ be the state that the automaton transition to when in p_1 with input b, we ignore the stack symbols on this modification. Suppose that q_2 is the corresponding "twin" state of q_1 but in machine M_2. Then, define the new transition $\delta'(p_1, c) = q_2$.
2. For every $q \in Q_2$, let $t = \delta(q, b)$. Set $\delta'(q, b) = \mathsf{reject}$ and define a new transition instead $\delta'(q, c) = t$ that moves on input c. This last modification essentially changes all transitions that are associated with input b in M_2, to use up a c instead.

To prove that the modifications in M' are enough to decide L', we start by arguing that if $w \in L'$, then M' ends in an accept state. To see this, consider what happens after $a^i b^i$ are processed. Since $a^i b^i \in L$, we know that at this point in the computation, M_1 is in an accept state. Since $w \in L'$, the next i characters are c, and because we're in an accept state of M_1, the first of occurrence of c causes a transition to a state in M_2. From there, the transitions within M_2 follow the *last* i transitions of δ_2 on input $a^i b^{2i}$, i.e., the last stretch of the i long sequence of bs. Since we modified these transitions to use up a c instead of b, the machine M' with input $a^i b^i c^i$ reaches an accept state.

On the other hand, if M' accepts on some string w, then we must argue that $w \in L'$. We first note that if w does not begin either with $a^i b^i c$, or with $a^i b^{2i} c$, then M' must reject. This is because all accept states are in M_2, so w has to touch some state in $\mathcal{A}_1$, and then transition with a c to M_2. Furthermore, if there are any characters other than c after the transition to M_2 is made, then it is easy to see that M' will reject: input b will cause a reject explicitly due to our modification, and input a will cause a reject because M_2 does not allow a character a after the first appearance of a b. At this point in the proof we have established that there is at least one c and there are two more questions to resolve: 1. Is the string $a^i b^i$ or $a^i b^{2i}$ before the first c? 2. How many more cs follow its first occurrence? Suppose w is of the form $a^i b^{2i} c^j$. If M' accepts, it follows that M_2 would accept $a^i b^{2i+j}$, violating our assumption that M_2 decides L. This is because before processing the first c, we transition to M_2 and then process cs as if they were bs. So if w is of the form $a^i b^{2i} c^j$ and is accepted, then M_2 would have accepted $a^i b^{2i+j}$. Therefore, it must be that w is of the form $a^i b^i c^j$. Finally, if $j \neq i$, by the same previous argument, M' must reject, or else M_2 would accept a string $a^i b^{i+j}$, where $0 \neq j \neq i$, violating our assumption about M_2. We conclude that w is of the form $a^i b^i c^i$, as claimed.

References

[1] D. Richards *Logic and Language Models for Computer Science*, third edition. World Scientific Publishing Co., 2018.

[2] M. Sipser. *Introduction to the Theory of Computation* (2nd edition). Course Technology, 2005.