Contents

Contents 1

0 Notations 1

0.1 Whats this all about? 1

1 Regular Languages 3

1.1 Deterministic Finite Automata 4
1.2 Regular Languages 8
1.3 The Generalization of Nondeterminism 11
1.4 Formal Definition 12
1.5 Coping with Nondeterminism 13
1.6 Examples 13
1.7 Comparison with DFAs 15
1.8 The Limitation of DFAs 20
1.9 The Pumping Lemma 20
1.10 Formula 21
1.11 Examples 22
1.12 A note on choosing a bad s 25

Notations

0.1 Whats this all about?

CS 4510: Automata & Complexity Theory. This course is primarily the study of two questions:

1. What are the limits of computation? (Computability Theory: Approx. 70% of course)
2. What makes some problems easy and others hard? (Complexity Theory: Approx. 30% of course)

Fundamentally about theoretical computer science—what is a computer and how can we effectively describe its limits and capacities? Automata are tools that we can use to reason about more powerful versions.

Formal Language Theory

The entire subject is formalized upon the notation used in *formal language theory*

Definition 0.1 (Alphabet)**.** An alphabet is a finite non-empty set of distinct symbols/glyphs/characters.

We most commonly will denote an alphabet with Σ. Some common alphabets can include $\{a, b\}, \{1\}, \{0, 1\}, \{a, \ldots, z, A \ldots, Z\}$. Let $\Sigma = \{a, b\}$ and consider the cartesian product $\Sigma^2 = \Sigma \times \Sigma$. It is conventionally has its elements represented in tuple form, such as (a, a). Here, we will drop the cumbersome paranthesi and commas and let $\Sigma^2 = \{aa, ab, ba, bb\}$. These are called strings or words.

Definition 0.2 (String, word)**.** A string or word is a finite sequence of letters from some alphabet. The length of the string is the number of symbols it contains.

Generalizing the previous example, Σ^n is all possible strings of length n. The set Σ^0 is defined as $\Sigma^0 = \{\varepsilon\}$, where ε is a special string of length zero called the empty string, $\varepsilon =$ "". It is different from the empty set $\emptyset$. It is analogous to the difference between an array of no elements, and a string of no length. They are of different types.

more string math

Definition 0.3 (Language)**.** A language is a selection of words $L \subseteq \Sigma^*$

Here are some examples of languages

- $L = \{aa, bb, abab, aaa, b\}$
- $L = \{w \in \Sigma^* \mid w \text{ begins with } a\}$
- $L = \{w \in \Sigma^* \mid \#a(w) \text{ is even}\}$
- $L = \{a^n \mid n \text{ is even}\}$
- $L = \{w \in \Sigma^* \mid \#a(w) \equiv 3, 4 \pmod 7\}$

- $L = \{w \in \Sigma^* \mid w \text{ is an encoding of a prime number}\}$

Its important you understand which way "the infiniteness" of an infinite language can go. There are no infinite length strings, each string must eventually terminate and have a specific length. But an infinite language has infinitely many words, and the length of words can be increasing, but each word is itself finite. It is analogous to how $\mathbb{N}$ is infinite, yet each number itself may be written with only finitely many digits. Convince yourself a language is infinite if and only if it has no longest word.

Automata

An automata is a hypothetical model of a computer. We may study the limitations of certain automata, or contrast them to one another. We do not really care about the automata themselves, but what they can tell us about the kinds of problems they can solve.

We need the ability to first discuss what it means to solve a problem, and here we borrow tools from formal language theory. A decision problem is a distinguishing, a partition of Σ^* into the "good" and the "bad". We give an automata a word, and it will either accept the string or reject it. We say that an automata M decides a language L if:

$$M \text{ on input } w \textit{ accepts} \iff w \in L$$
$$M \text{ on input } w \textit{ rejects} \iff w \notin L$$

We are concerned with what kinds of automata can decide what kinds of languages.

There are two perspectives. First fix the machine, and note that each machine must define some language. Every machine has some behavior. Next is to fix the language and consider all possible machines which may decide it correctly. We are more concerned with the second perspective than the first.

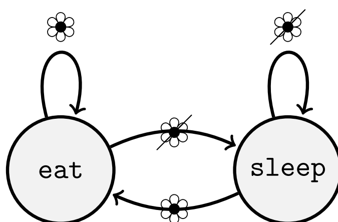
Definition 0.4 (Decision Problem)**.** A Decision problem is a computational problem which can be phrased as a yes or no question

Phrasing everything as decision problems simplifies the mechanics immensely, and lets us fully use the power of set theory. A decision problem can easily be formalized in set theory, since for any set, a word is either in the set, or not in the set. This is a binary relationship. Consider the language $\{w \in \Sigma^* \mid w \text{ is an encoding of a prime number}\}$ A machine to decide this language would take on input a word and simply have to accept or reject. We could then remark that this kind of automata would have the power of understanding prime numbers. A more cumbersome way would be to try to use *search problems*, which could be phrased like *on input n, output the nth prime.* If such a machine could perform such a task, we could remark that it too could comprehend prime numbers. But there is a diversity of ways different machines could output the nth prime number. By restricting the output of automata to be a simple boolean, we only need to ensure they have two small wired lights. This will help us compare and contrast them effectively.

Regular Languages

1.1 Deterministic Finite Automata

Many systems and processes can be represented as a finite collection of modes or steps or “states of mind”. Even processes which are assumed continuous can be arbitrarily discretized, such as the phases of the moon. Lets consider an example. My rabbit has exactly two braincells, and they have to take turns. The first braincell, she uses when she eats, and the second braincell, she uses when she sleeps. We may represent these two states of mind as two labelled nodes. We may then define *transitions* between those states of mind as outgoing arrows, on which a transition is acted upon sensory input. There is either food, or no food. She will sense food, wake up, and continue eating until there is none. We could represent the relationship between her states of mind with the following *state diagram*.



Many complex systems could be primitively modeled this way, such as the water cycle, the economy, and carbohydrate metabolism. Our first automata is motivated by this simple design.

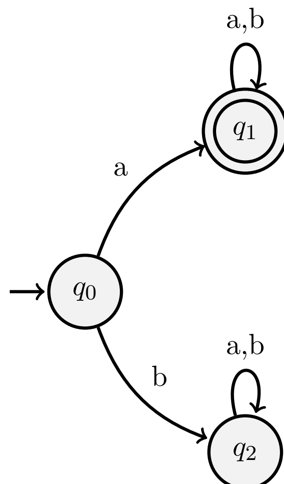
Definition 1.1. A Deterministic Finite Automata (DFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$:

- $Q = \{q_0, \ldots, q_k\}$ is a non-empty finite set of *states*
- Σ is the non-empty finite *alphabet*, usually $\Sigma = \{a, b\}$ or $\{0, 1\}$
- $\delta : Q \times \Sigma \to Q$ is the *transition function*. It is a well-defined finite function. Every state symbol pair in the input has a single output.
- $q_0 \in Q$ is the designated *start state*. We have to start somewhere.
- $F \subseteq Q$ is the set of *acceptance*, or final states. If a state is not final, we may say it is rejecting.

We define a computation of a DFA on a word to be a repeated sequence of applications of the transition function, one per letter in the word sequentially. We say the DFA accepts the word if the computation terminates on a final state, and the DFA rejects if the computation terminates on a non-final state.

Lets do several examples.

Example 1.1. $L_1 = \{w \in \Sigma^* \mid w \text{ begins with } a\}$



Before discussion of this specific DFA, we note the notation of DFAs in general. The previous formal definition can be cumbersome, so it is better to give a *state diagram.* A state diagram is what you see above, sort of like a graphical programming language. We denote the start state as q_0 and with a tiny arrow from nothing. We denote an accepting states as those with a double circles, and rejecting as exactly those without. Note that our transition function is the edges, and the function is well-defined when each state has exactly $|\Sigma|$ outgoing transitions, one per symbol. This diagram clearly communicates all five parts of the tuple, but if we were to state it explicitly, it would look like

- $Q = \{q_0, q_1, q_2\}$
- $\Sigma = \{a, b\}$
- The start state will always be q_0.
- The transition function δ can be encoded as the following table:

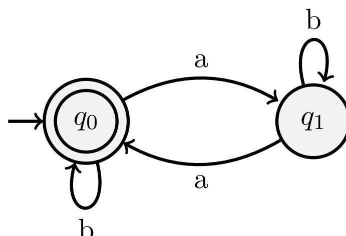
Q	Σ	Q
q_0	a	q_1
q_0	b	q_2
q_1	a	q_1
q_1	b	q_1
q_2	a	q_2
q_2	b	q_2

- $F = \{q_1\}$

The state diagram communicates all five parts more effectively, but know we may delegate to the formal tuple definition during proofs.

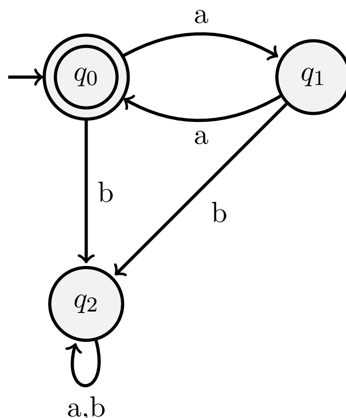
Consider a computation of this DFA on any word. It branches to two different states on the first letter. Once you enter states q_1 or q_2, you may not leave. Once you enter either of these two purgatories, the rest of the letters of the word are ignored. We denote q_1 as the good purgatory by making it a final state, and q_2 as the bad purgatory by making it a rejecting state.

Example 1.2. $L_2 = \{w \in \Sigma^* \mid \#a(w) \text{ is even}\}$



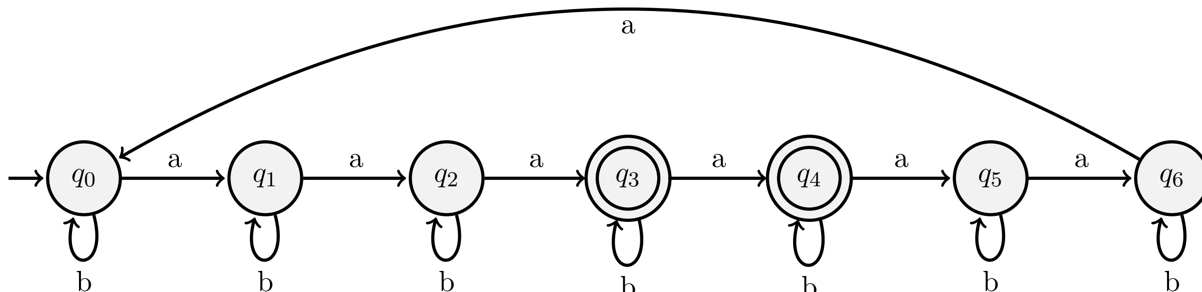
Here we have transitions to keep track modulo two the number of a's we have seen. Any computation of ε on a DFA will end on the start state. Therefore, a DFA accepts the empty string if and only if its start state is also a final state. We have self loops for the b's because we want to ignore them. What if we wanted to reject whenever we saw any b's, and not ignore them?

Example 1.3. $L = \{(aa)^n \mid n \in \mathbb{N}\}$



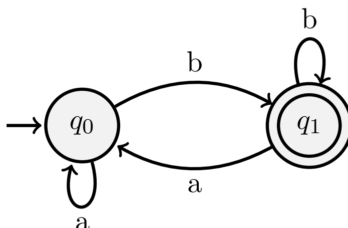
Now as soon as we see a b, we immediately enter purgatory and can never leave.

Example 1.4. $L = \{w \in \Sigma^* \mid \#a(w) \equiv 3, 4 \pmod 7\}$



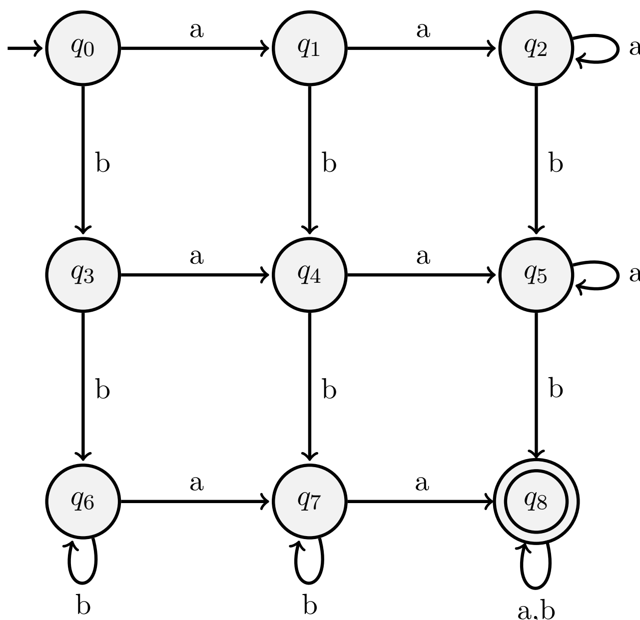
There is nothing too special about the number two. We may generalize the previous example to keep track of residues modulo any other number. Create one state per equivalence class, and simply transition between them on seeing an a, and ignore all b's. Going from one state to the next means you have seen an additional a. Going around the clock means you have seen a seven times.

Example 1.5. $L = \{w \in \Sigma^* \mid w \text{ ends with } b\}$



The DFA does not have any ability to rewind its input, jump around, or do any post processing. It halts exactly when it runs out of symbols to read, but it doesn't know when that will happen. Like death, it must be vigilant and prepared for it at any moment. In this DFA, if we see a b, we transition to an accept state, but if we see an a, we must transition away.

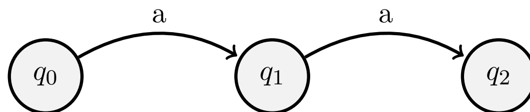
Example 1.6. $L = \{w \in \Sigma^* \mid w \text{ has atleast two a's and atleast two b's}\}$



To each state, we may correspond a semantic meaning. Here, there are several paths to the accept state. The rows keep track of how many a's you have seen, and the columns keep track of how many b's you have seen. In fact, the unique paths to the accept state correspond to in which order you saw your required two a's and two b's. The fact that the shortest path to the accept state is of length four can also tell us that the shortest strings in this language must be of length four.

Programming Advice

A DFA is essentially a very limited kind of finite space program. There are only finitely many things it can keep track of at once, and it can also only interact with its input in a "read once only" way. When designing DFAs, it is helpful to think that each state has assigned to it a semantic meaning, and that a state can only be reached by certain strings which satisfy certain properties. For example, suppose we have a portion of a DFA which looks like the following



You know you may only enter q_1 upon seeing an a. You know you may only enter q_2 not only upon seeing an a, but seeing an a from q_1, which you could only enter upon seeing an a. So q_2 may be entered only upon seeing aa. Each state corresponds this way like a line of code. You know a certain line of code may be hit by the control flow only if certain conditions are met. Our portion of a DFA could correspond to a portion of a program as:

```
if w[i] == 'a':
    if w[i+1] == 'a':
        *
```

You know the line with the $*$ will only be executed if certain conditions are met. Analogously, q_2 can only be entered by a string if a prefix of it meets certain conditions.

1.2 Regular Languages

What kinds of languages can DFAs decide? We don't yet know the problems they are capable of solving or not solving. We say a language is regular if and only if it is decided by a DFA.

Definition 1.2. We write the class of languages decidable by a DFA as $\mathscr{L}(DFA)$. These are called the regular languages.

Note that a word is a finite sequence of symbols, a language is a (possibily infinite) set of words, and a class is a (possibily infinite) set of languages. A class is a set of sets of strings. We want to study the regular languages, and the only way for us to do so is by studying DFAs. What properties to the regular languages have?

Theorem 1.1. If $L \in \mathscr{L}(DFA)$, then $\overline{L} \in \mathscr{L}(DFA)$. The regular languages are closed under complement.

A DFA is a machine to tell you exactly what strings to accept, but by doing so, it also tells you exactly what strings to reject.

Proof. Let L be a regular language, then there exists a DFA $(Q, \Sigma, q_0, \delta, F)$ to decide L. Consider the DFA $(Q, \Sigma, q_0, \delta, Q \setminus F)$. It is identical to the first DFA, except that every previously accepting state is now rejecting, and every previously rejecting state is now accepting. If $w \in L$ then the first DFA will accept w, so the second DFA will reject w. If $w \notin L$, then the first DFA will reject w, so the second DFA will accept w. Therefore, the second DFA will except exactly and only the strings not in L, which is the complement $\overline{L}$. Since the second DFA is one which decides $\overline{L}$, then $\overline{L}$ has a DFA to decide it, and is therefore, regular. $\square$

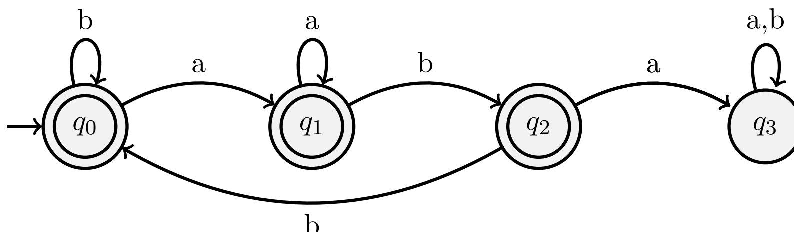
Often times, the core idea of such proofs are a construction of an automata of some sort. For some complex constructions, a proof of correctness would be done by induction on the length of the input. The correctness of the automata is often obvious, and such a wordy proof is often unnecessary. Analogously, many primitive algorithms also have omitted proofs of correctness, when the algorithm is simple enough that it is obvious. We don't really understand theorems until we

understand their proofs, but computation is such a natural, human cognitive process, an example is almost as instructive.

Suppose we wanted to create a DFA for the language

$$\{w \in \Sigma^* \mid w\text{does not contain the substring } aba\}$$

Rather than try to find some kind of positive characterization of strings with this property, lets just check if the string does contain aba as a substring. We reject those strings, and then accept everything else.



Theorem 1.2. Let $L_1, L_2 \in \mathscr{L}(DFA)$. Then $L_1 \cap L_2 \in \mathscr{L}(DFA)$. The regular languages are closed under intersection.

We may use one DFA to simulate two other DFAs simultaneously, and have our DFA accept only if the two DFAs it is simulating accept. Consider how DFAs are analogous to a very limited kind of program. Among its other limitations, it only uses constant memory. We may combine two constant memory programs into one (bigger) constant memory program. This is the intuition. Each state of our new DFA will correspond to a pair of possible states in two different DFAs. Computation on our DFA will correspond to computation on two other DFAs in parallel. We shall make our DFA accept only if the two DFAs it is simulating accept. For simplicity, suppose they have the same fixed input alphabet.

Proof. Let L_1 be decided by DFA $(Q_1, \Sigma, q_0^1, \delta_1, F_1)$ and L_2 be decided by DFA $(Q_2, \Sigma, q_0^2, \delta_2, F_2)$. We program a DFA called the cartesian product DFA $(Q, \Sigma, q_0, \delta, F)$ to decide $L_1 \cap L_2$ as follows:

1. $Q = Q_1 \times Q_2$
2. Σ is the same
3. $\delta : (Q_1 \times Q_2) \times \Sigma \to (Q_1 \times Q_2)$ such that

   $$\delta((q_i, q_j), a) = (\delta_1(q_i, a), \delta_2(q_j, a))$$

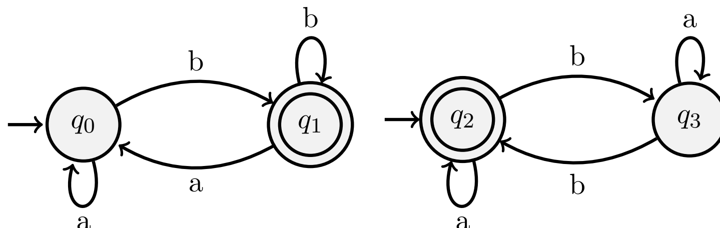
   for $q_i \in Q_1$ and $q_j \in Q_2$. The first DFA is simulated in the first coordinate, and the second DFA in the second coordinate.
4. $q_0 = (q_0^1, q_0^2)$
5. $F = F_1 \times F_2$

Let $w \in L_1 \cap L_2$, then $w \in L_1$ and $w \in L_2$. So both DFAs will accept w. Then the computation of this DFA on w will end on some state (q_i, q_j) where q_i is accepting in the first DFA, and q_j is accepting in the second DFA. By definition, this is an accepting state of the cartesian product DFA, and thus the DFA accepts w. Similarly, if $w \notin L_1 \cap L_2$, then the cartesian product DFA will not reach an accept state on w, and it will reject it. Thus, this DFA decides $L_1 \cap L_2$. $\square$

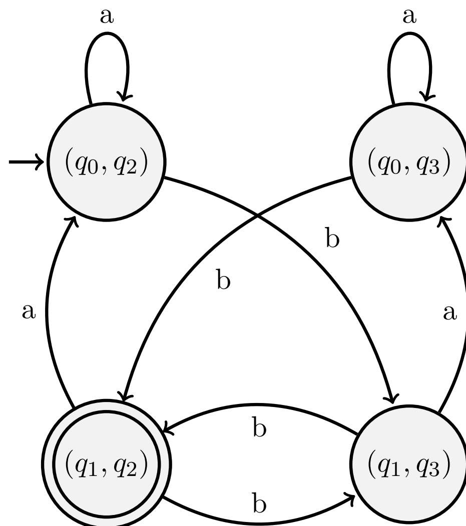
To demonstrate this constructionLets proceed with an example.

$$L_1 = \{w \in \Sigma^* \mid w \text{ ends with a } b\}$$
$$L_2 = \{w \in \Sigma^* \mid \#b(w) \text{ is even}\}$$

Lets make two DFAs for these languages.



Out cartesian product DFA then looks like the following:



We may assign meaning to the states. State (q_1, q_2) means being in state q_1 in the first DFA and state q_2 in the second DFA simultaneously. You may only end on state q_1 if your string ends with a b, and you may only end on state q_2 if you have seen an even number of b's at that point. So strings which end on state (q_1, q_2) are those which both end with b and have seen an even number of b's. If a string lands on states (q_0, q_2) or (q_1, q_3), then it is accepted by one DFA but not the other. What if you wanted the simulator DFA to accept if either DFA accepted?

Theorem 1.3. Let $L_1, L_2 \in \mathscr{L}(DFA)$. Then $L_1 \cup L_2 \in \mathscr{L}(DFA)$. The regular languages are closed under union.

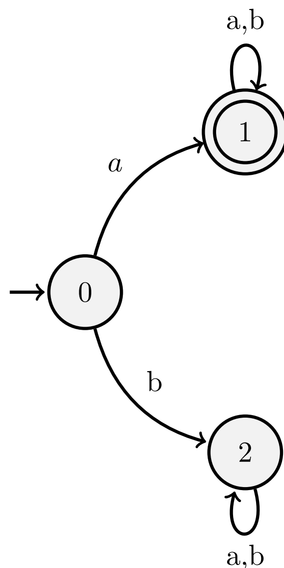
Proof. Simply modify the construction from the previous proof, where the only difference is that the final states are constructed as $F = (Q_1 \times F_2) \cup (F_1 \times Q_2)$. Then our cartesian product DFA would accept if any of its two simulated two DFA accepted, and it would reject if both DFAs rejected. □

In the previous example, our accepting states would then be $F = (q_0, q_2), (q_1, q_2), (q_1, q_3)$.

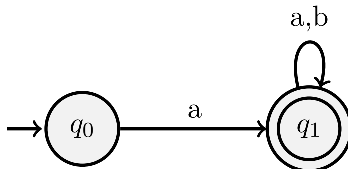
1.3 The Generalization of Nondeterminism

We noted that DFAs are weak. Let's try to generalize them. Recall that a DFA can be represented as a tuple $(Q, \Sigma, \delta, q_0, F)$. Given this definition, we wish to modify it to hopefully extend its power. The only useful thing we can extend is the way in which states interact with each other; the transition function δ. The rest of the device is static. We extend δ in the following three ways:

Implicit Rejection We allow transitions to be undefined, and it is understood that undefined transitions implicitly reject. As an example, recall the following DFA which decides the language $\{w \in \Sigma^* \mid w \text{ begins with } a\}$.

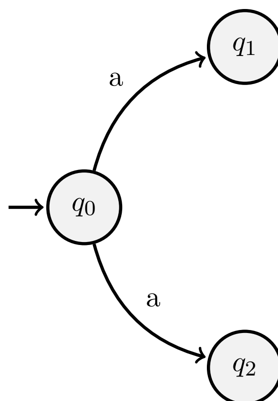


With implicit rejection, we could represent equivalently as



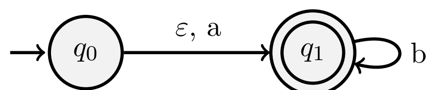
If we are at q_0 and we see b we would reject. This can be helpful for programming. Consider how well-defined a DFA is. Its like it has every edge case covered with all that try-catch nonsense. Implicit rejection allows us to lazily construct only the parts that we care about. Then "undefined behavior" results in immediate rejection. We can have an "if" without having to have a matching "else". Note that when we perform a complement of the accept states in a DFA that decides a language L, we get the complement of the language. The same does not hold here due to implicit rejection.

Nondeterministic Transitions We allow transitions of more than one of the same type. This means that you can have multiple outgoing transitions with the same input. For example

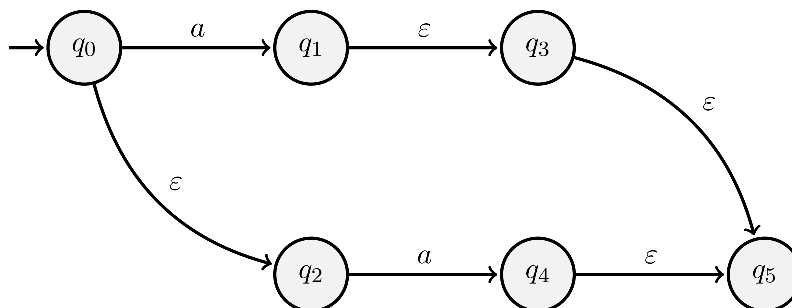


Consider the computation on a word beginning with an a. Which state are you in? q_1? q_2? You are in both! We define a nondeterministic computation to accept *if there exists* an accepting computation. For all possible states you could end up in, if at least one of them is accepting, then the NFA accepts the string. An NFA rejects a string if it doesn't accept, and this happens *if for all* computations, none are accepting. We shall expand on nondeterminism soon.

Epsilon Transitions We define "ε-transitions", which can be taken for free. For example



a, ab, abb are some strings which are accepted. But now that we allow ε-transitions, b, bb, ε are also accepted. While normally, each transition "costs" the next letter of the input, an ε-transition costs nothing. You may take it for free. It is important to know that the choice to take it is not forced. A nondeterministic computation may choose not to take it. For example



On input of ε, this NFA will end on state q_0 and q_2 simultaneously. On input of a, it will end on states q_1, q_3, q_4, q_5. On input of aa, it would implicitly reject.

1.4 Formal Definition

Definition 1.3. A Nondeterminisitic Finte Automata (NFA) can be represented by a 5-tuple $(\Sigma, Q, q_0, \delta, F)$ where:

- $Q = \{q_0, \ldots, q_k\}$ is a non-empty finite set of *states*
- Σ is the non-empty finite *alphabet*, usually $\Sigma = \{a, b\}$ or $\{0, 1\}$

- $\delta : Q \times (\Sigma \cup \{\varepsilon\} \to \mathcal{P}(Q)$ is the *transition function.* It is a well-defined finite function. Every state symbol pair in the input has a single output.
- $q_0 \in Q$ is the designated *start state.* We have to start somewhere.
- $F \subseteq Q$ is the set of *acceptance*, or final states. If a state is not final, we may say it is rejecting.

1.5 Coping with Nondeterminism

With these three new relaxations, we have defined a new kind of automata, the nondeterministic finite automata (NFA). On input a word, there may be multiple different possible computations, and we say an NFA accepts some string if there exists atleast one computation to an accepting state. It does not matter how many more rejecting computations there are.

Its important to understand nondeterminism and not just have deterministic coping strategies. Nondeterminism isn't real. You could not build a nondeterministic computer, but it doesn't matter. We may still study this unrealizable machine as a purely theoretical device. The following analogies may help in visualizing this power.

1. **Graph Search** An NFA or DFA can just be thought of as a graph, and a word computed by an NFA or DFA can be thought of as a path in the graph. You could easily determine if a DFA accepts a word by using the word as instructions on which path to take in the DFA, but to determine if an NFA accepts a word, you may have to employ a graph search algorithm, such as breadth first search or depth first search. Coming to a fork in the road, you explore down both paths until you find an accept state. Under this view, the power of nondeterminism is only how time is measured. The DFA on computation of a word of length n takes exactly n steps. An NFA also takes exactly n steps, but epsilon transitions take no time, and certain paths of the same depth are computed "in parallel", and their time is not double counted. You and I are deterministic. In order to determine if an NFA accepts a word, using pen and paper, necessarily may take more than n steps.

2. **Lucky Coin** During your computation you come to a nondeterministic transition. Imagine you flip a lucky coin that tells you exactly which path to take. Through a purely imaginary way, you have divine information on which path will correctly lead you to an accept state. You have faith in the coin, so you follow it. Somehow, you have the precognition to know where the answer is.

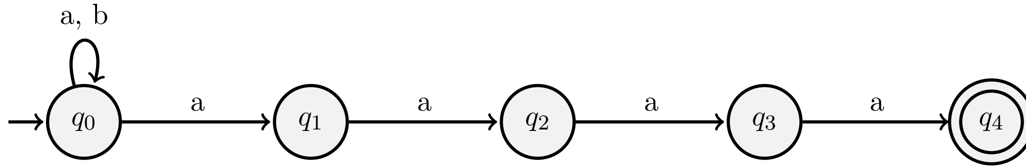
3. **Alternate Timelines**[1] For each nondeterministic action, create multiple timelines. Each timeline consists of the what-if for each possible choice. As long as in one timeline you reach an accept state, then the computation is accepting.

1.6 Examples

Lets show a few examples

Example 1.7. $L_1 = \{w \in \Sigma^* \mid w \text{ ends with } aaaa\}$

[1]Different science fictions have different rules for how time travel works. I am going off of the episode Remedial Chaos Theory from Community.

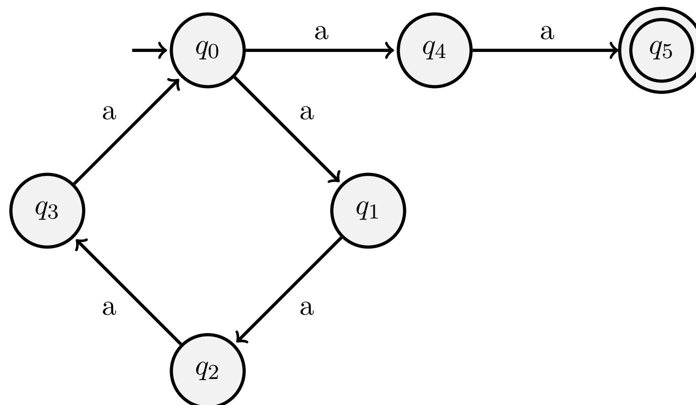


Consider the computation of this machine on input $aaaaaaaaa$. If you are at q_0 and you read an a, you may choose to either stay at q_0 or move on to q_1. Note that this word is accepted by the NFA because it may correctly guess exactly when it is four a's from the end and then choose to leave q_0. Another way is to consider all possible guesses of when to go to q_1 on seeing an a. If we guess too late, we will terminate on one of q_0, q_1, q_2, q_3 and not accept. If we guess too early, we will reach q_4, but then have more input to read, and must implicitly reject since q_4 has no outgoing transitions. Most of the computations will be rejecting but it doesn't matter, as there is atleast one accepting computation, one correct guess.

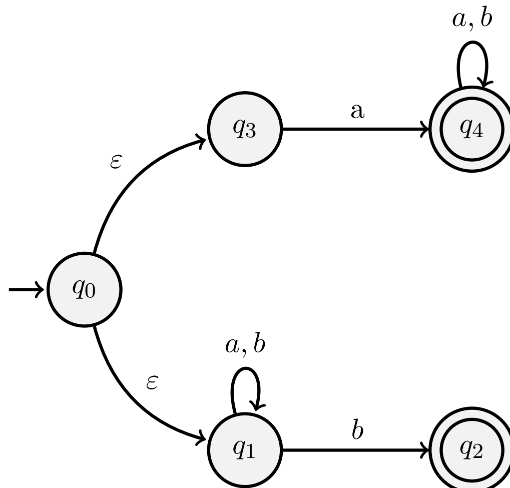
Let the delimiter symbolize when you non deterministically guess to go from q_0 to q_1. Consider the following computations

$\|a$	a	a	a	a	a	a	too early, implicit rejection from q_4
a	$\|a$	a	a	a	a	a	too early, implicit rejection from q_4
a	a	$\|a$	a	a	a	a	too early, implicit rejection from q_4
a	a	a	$\|a$	a	a	a	accepts
a	a	a	a	$\|a$	a	a	too late, rejects from q_3
a	a	a	a	a	$\|a$	a	too late, rejects from q_2
a	a	a	a	a	a	$\|a$	too late, rejects on q_1
a	a	a	a	a	a	a	too late, rejects on q_0

Example 1.8. $L = \{a^{3n+2} \mid n \in \mathbb{N}\}$
Lengths of the strings in this language form an arithmetic progression. You non deterministically choose how many times you would go around the loop, determined by every time you reach q_0, do you choose to go to q_1 or q_4.



Example 1.9. $L = \{w \in \Sigma^* \mid w \text{ begins with } a \text{ or ends with } b\}$



We need to accept strings if they accept any of two conditions. We will nondeterministically guess which condition to check. If the string begins with a, then there is a computation on the above branch which accepts. If the string ends with b, then there is a computation on the below branch which accepts, where the prefix of the string is nondeterministically guessed.

1.7 Comparison with DFAs

We don't really care about comparing the automata themselves, but comparing their *power*. Let $\mathscr{L}(NFA)$ represent the class of languages which are decidable by an NFA. We don't care to compare NFAs and DFAs, but $\mathscr{L}(DFA), \subseteq \mathscr{L}(NFA)$. What is the pow

Theorem 1.4. $\mathscr{L}(DFA) \subseteq \mathscr{L}(NFA)$

Every DFA is an NFA. An NFA has all these super powers, but there is no requirement to use them. Though it may be obvious just from the generalization that is nondeterminism, for exercise, we prove $\mathscr{L}(DFA) \subseteq \mathscr{L}(NFA)$.

Proof. Let $L \in \mathscr{L}(DFA)$. Then there exists a DFA to decide L. Note that this DFA is also an NFA, so there exists an NFA to decide L. Then $L \in \mathscr{L}(NFA)$. Since this is true for all $L \in \mathscr{L}(DFA)$, we see that $\mathscr{L}(DFA) \subseteq \mathscr{L}(NFA)$. □

Theorem 1.5. $\mathscr{L}(NFA) \subseteq \mathscr{L}(DFA)$

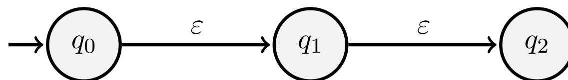
This should surprise you! We gave a normal computation device all this unrealistic unrealizable power. Yet, this power can be simulated using realizable methods. For any NFA, we will show how to simulate it on a DFA. This means that $\mathscr{L}(NFA) \subseteq \mathscr{L}(DFA)$. Combining the aforementioned point, we get $\mathscr{L}(DFA) = \mathscr{L}(NFA)$ We simulate an NFA on a DFA. Although an NFA may be in many states at once, it can only be in finitely many. This is the key idea behind the simulation. To each possible set of state the NFA could be in, we assign one state of our DFA to represent each subset of the NFA. Then the NFA going between subsets of states can be simulated by our DFA going from just one state to another. This is called the powerset construction. Proven by Michael O. Rabin and Dana Scott in 1959, this work earned them the Turing award in 1976. There is also a small comment on economy. NFAs can be smaller. There are languages which have NFAs of n states, but require DFAs of 2^n states. We do not care about the efficienty, rather if these structures exist at all to decide. The simulation of an NFA by a DFA works since 2 to the power of a finite

number is still a finite number. There is exponentially more to keep track of, but that is still only a finite amount.

There is also the issue of these epsilon transitions. We define the concept of reach.

$$reach(q_i) = \{q_i \text{ and any state reachable from } q_i \text{ by } \varepsilon\text{-transitions}\}$$

For example



Then $reach(q_0) = \{q_0, q_1, q_2\}$.

Proof. Let N be any NFA with $N = (\Sigma, Q, q_0, \delta, F)$. We construct an equivalent DFA $D = (\Sigma', Q', q_0', \delta', F')$ so that $L(N) = L(D)$.

- $Q' = \mathcal{P}(Q)$ For each possible subset of the states of the NFA, we creat one state of our DFA.
- $\Sigma' = \Sigma$
- $q_0' = reach(q_0)$ If there is an ε-transition from the start state of the NFA, then the computation need not necessarily begin at q_0 if this ε-transition is taken first. Then the start state of our DFA corresponds to the set of possible states in which the computation could begin in the NFA, which is those states reachable from q_0 in the NFA.
- For $S \subseteq Q$ any subset of states of the NFA and $a \in \Sigma$, we define

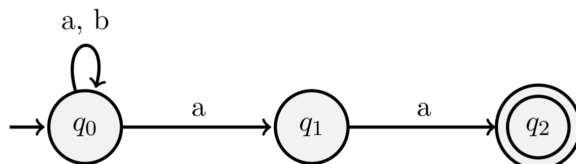
  $$\delta'(S, a) = \bigcup_{q \in S} reach(\delta(q, a))$$

  For S a state of the DFA, its outgoing transitions are defined to be the state corresponding exactly and only to the set of states of the NFA which you can go to on viewing the same symbol.
- $F' = \{S \subseteq Q \mid S \cap F \neq \emptyset\}$ Recall than an NFA accepts if there exists a computation which reaches an accept state. After computation on a word, you may be in several states at once, but if atleast one is accepting, the machine accepts. We set the accepting states of the DFA to be those which contain any accept state of the NFA.

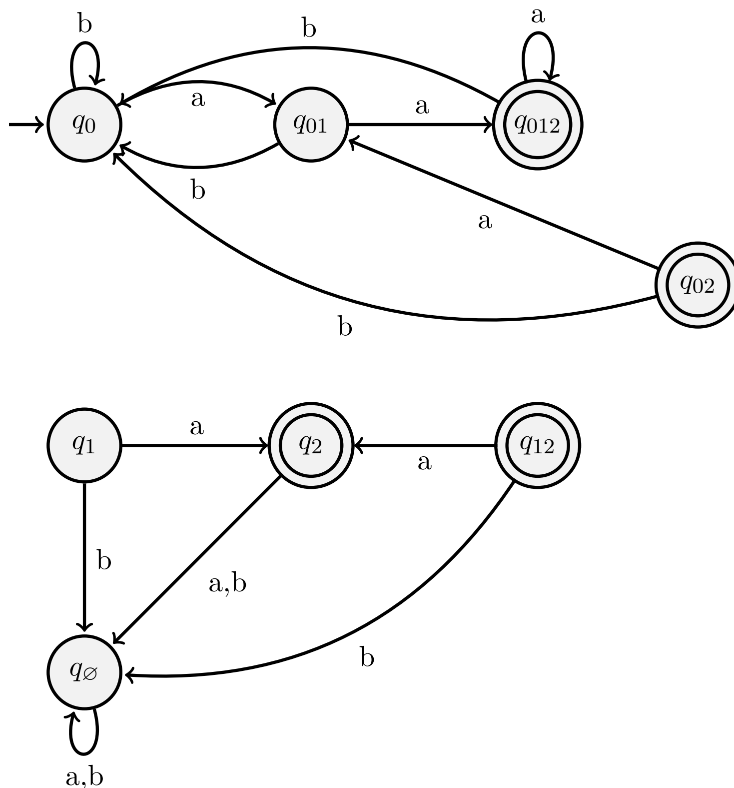
□

Example

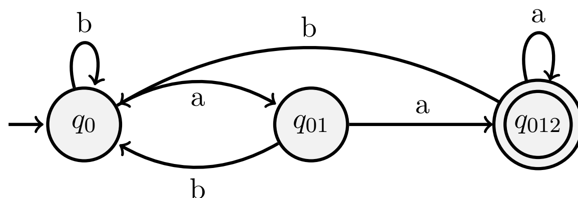
$L_2 = \{w \in \Sigma^* \mid w \text{ ends with } aa\}$



By following the above process, we get the corresponding DFA

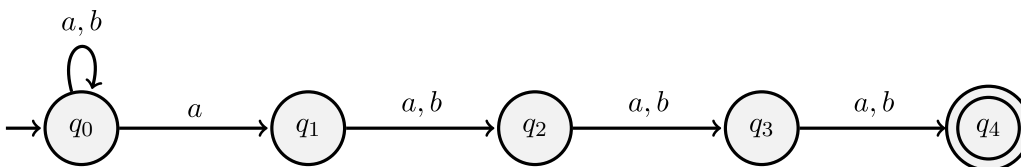


We observe that there are unreachable states like q_{02} and an entire disconnected component. This process does not guarantee to give a minimal DFA, just an equivalent one. On cleaning up these unreachable states, we get the following DFA

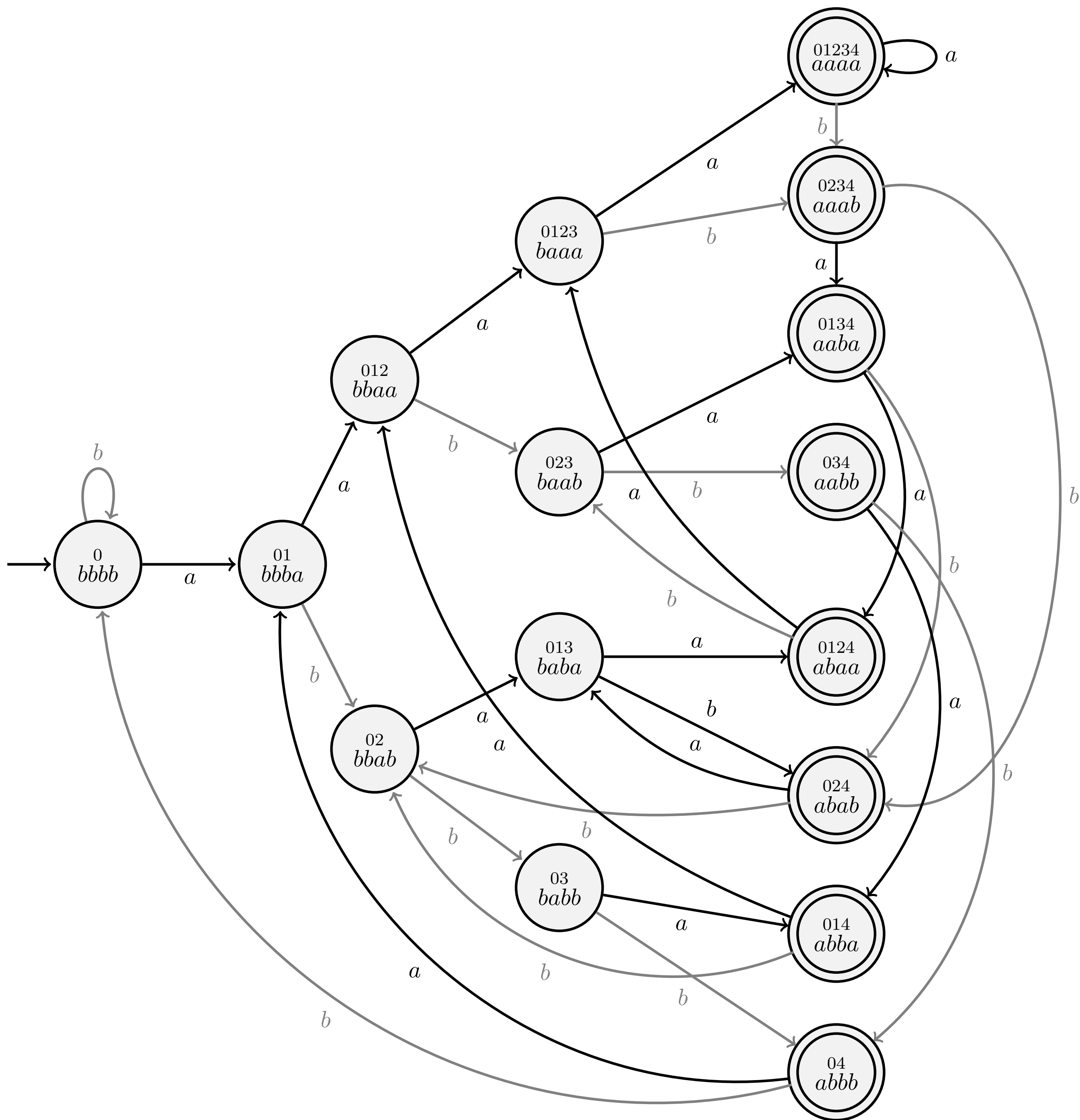


Each state represents a superposition of the states in the NFA. A state being unreachable in the DFA could be interpreted to mean that its exact combination of states in the original NFA was unachievable. You cannot be in q_2 in the NFA without also being in q_0 and q_1.

Lets do another example. $L = \{w \in \Sigma^* \mid w$ has an a as the fourth symbol from the right.$\}$. We could easily constrict the NFA similar as



Following the powerset construction, we get the following DFA.



Additional information has been given in this state diagram. For clarity, b transitions are shaded differently. Each state is not only given a sequence corresponding to the subset of states (023 corresponds to $\{q_0, q_2, q_3\}$), but also a word from Σ^4. Each string corresponds to the last four seen symbols. This is the semantic meaning we assign to each state. A DFA has a memory worse than a goldfish, and can only keep track of where it currently is. Upon a computation of "what symbol did I see three letters ago?", you end up needing one state for every possible word of length four,

of which there are $|\Sigma|^4 = 16$. This DFA is quite large and messy, and you can prove this language cannot be equivalently decided by a smaller DFA.

1.8 The Limitation of DFAs

We previously mentioned that we have some intuition on the limitations of DFAs. Although they seem quite powerful, there are languages which have no DFA to decide them. The goal of today is to prove that.

Consider the language $\{a^n b^n \mid n \in \mathbb{N}\}$. A DFA has a finite amount of states, and is only able to read the string left to right. It cannot do any pre or post processing. It cannot read symbols it has previously. As it reads left to right, it somehow is tasked with memorizing an arbitrarily large amount of information, the number of a's, in order to match them to the number of b's. Note how different this is than $(ab)^*$, or a^*b^*, which can be computed using only a finite number of states. A DFA of say, 20 states may correctly decide if a string has the form $a^{20}b^{20}$, but this DFA must fail on a string of a large enough size, say $a^{100}b^{100}$.

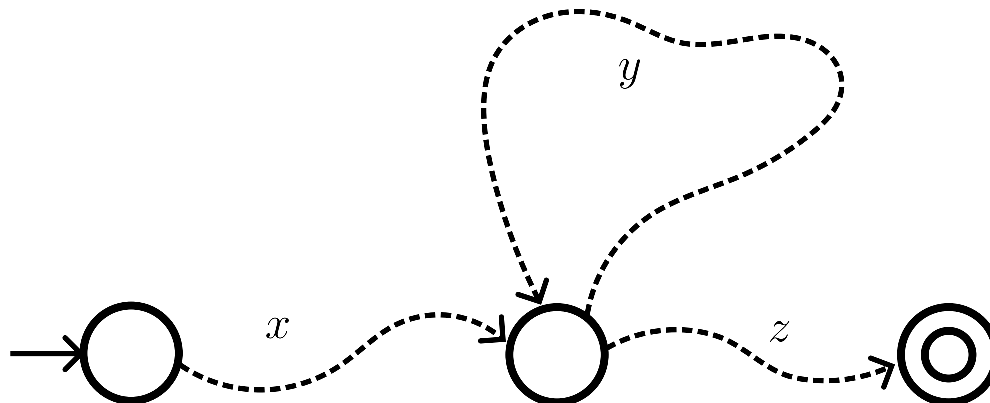
These nonregularity arguments will all follow from a vary simple reason: The pigeonhole principle.

Definition 1.4 (Pigeonhole Principle)**.** If m pigeons are assigned to n holes with $m > n$ then some hole must have more than one pigeon.

If a DFA accepts an infinite language, then there are strings of arbitrarily long length that it must accept, yet it must do so only with finitely many states. The strings the DFA is tasked to accept are much much longer than the size of the DFA itself. The pigeonhole principle will apply to the computation on DFAs, which will show that regular languages have an interesting property. The contrapositive will help us prove a language is not regular.

1.9 The Pumping Lemma

Suppose we have a DFA, D, made up of p states. On input a word w, each letter of the word, will visit a new state. So if $|w| = n$ then on the computation of n, there will be a visited sequence of $n + 1$ states. Consider a word w with $|w| = p$. When we simulate D on input w, it will visit $p + 1$ states. By the pigeonhole principle, some state is repeated twice. our computation path through the DFA must contain a cycle.



Note each letter of our word takes not one state, but one transition. If we have p states, and compute on a word of length $\geq p$, then some state is visited twice in our computation path. We may not know where this loop is or how long it is, but we know that it must exist by the pigeonhole principle. The DFA is such a simple stupid device, if it accepts a long enough string, it must also

accept if you were to take that string and repeat a substring of it arbitrarily many times. The Pumping Lemma is a formalization of this intuitive idea.

Definition 1.5 (Pumping)**.** We say a language can be pumped if there exists a number p for every $w \in L$ with $|w| \geq p$, there exists a partition of $w = xyz$ such that:

- $|xy| \leq p$
- $|y| > 0$
- $\forall i \in \mathbb{N}(xy^iz \in L)$

Theorem 1.6. Let L be an infinite regular language. Then L can be pumped.

Proof. Let L be a regular language. Then there exists a DFA $D = (Q, \Sigma, q_0, \delta, F)$ to decide L. Let $|Q| = p$ and let $w \in L$ be any word with length $|w| = n \geq p$ with $w = w_1w_2...w_n$. Consider the sequence of states visited during the computation of D on w, and let these states be enumerated as $s_1, s_2, ..., s_{n+1}$ with $\delta(s_i, w_i) = s_{i+1}$ for $i \leq 1 \leq n$. Since $n + 1 > p$, by the pigeonhole principle, among the first $p + 1$ visited states $s_1, ..., s_{p+1}$, there must exist a state which has been visited twice. Let the first of these visits be s_i and the second of these visits be s_j. Consider the partition of $w = xyz$ into $x = w_1...w_{i-1}, y = w_i...w_{j-1}, z = w_j...w_n$. We demonstrate these choices of x, y, z satisfy our three conditions which require L to be pumpable.

- By the pigeonhole principle, we know the repeition must occur in the first $p+1$ visited states, so $j \leq p + 1$ and since $|xy| = j - 1$ then $|xy| \leq p$.
- Since $i \neq j$, we know y is never the empty string, thus $|y| > 0$.
- The string x will take D from q_0 to our repeated state q, y will take D from q back to q, and z will take D from q to an accept state, q_a. We may compose these paths and notice that $\forall i \; xy^iz$ will take D from q_0 to q_a. What is a path from the start state to an accept state if not exactly an accepted word, thus we see that $\forall i \; xy^iz \in L$

□

You should think of i here as the number of times you may traverse the loop. Traversing it one time is the original string xyz. You may also traverse it zero times, so $xyz \in L \implies xz \in L$. You may traverse it twice, so $xyz \in L \implies xyyz \in L$, and so on.

The pumping lemma is not itself useful to proving that a language is regular. Instead, we take its contrapositive: If an infinite language cannot be pumped, then it is not regular. Note that this is not an if and only if and there do exist some nonregular languages which can be pumped.

1.10 Formula

The pumping lemma has many moving pieces and can be tricky to apply. There are alternating existential and universal quantifications through out[2]. Of the proof techniques available to you, it certainly is the most cumbersome, and is surprisingly common to make a mistake on what you can or cannot choose. I suggest you use this proof template. Suppose that L is the language we want to prove is not regular.

[2] $\exists p \forall w \exists x, y, z \forall i(...)$, thats four alternating quantifiers!

1. Assume to the contrary, L is regular with pumping length p
2. Choose some $s \in L$ such that $|s| \geq p$
3. For all cases $s = xyz$ such that $|xy| \leq p$ and $|y| > 0$
4. Choose $i \neq 1$ and demonstrate that $xy^iz \notin L$
5. Conclude that L cannot be pumped, and therefore L is not regular

Lets go through the importance of each step.

- First, by assuming to the contrary that L is regular with pumping length p, we are supposing that there exists a DFA of p states. When we reach a contradiction, then no such DFA of p states can exist. Since p is general, this means that no such DFA can exist at all. We cannot fix p. If we did a pumping lemma proof with $p = 5$, this would conclude that there is no DFA of five states. It does not imply there is no DFA at all, as there may exist a DFA with more than five states to decide the language.
- We choose to pump some string in the language. By choosing $s \in L$, we know s brings our assumed DFA to an accept state, like a path in a graph. By requiring $|s| \geq p$, we are enabled to apply the pigeonhole principle, and we know that this computation contains a repeated state. It is not uncommon for us to choose strings with length much much larger than p. The only requirement is that its length is greater than or equal to. Choosing a good s will effect the proof greatly. A poor choice of s may make the proof very long, or even impossible. We will expand upon this later.
- In the computation of s on our assumed to the contrary to exist DFA, we are guaranteed that there exists a loop somewhere by chosing $|s| \geq p$, but we don't know where. So we have to consider all possible cases of where this loop could be. We model this as considering all ways to break up s into the three parts $s = xyz$ subject to our two conditions on each case. Firstly that $|xy| \leq p$. This ensures that the occurance of a repeated state occurs somewhere before the end of what we denote as y. The second condition $|y| > 0$ ensures that this cycle is actually occuring. Note that $|y| = 0$ trivially ensures we could never reach a contradiction.
- For each case, you only need to choose an $i \neq 1$ so that $xy^iz \in L$. Most of the time $i = 2$ works, we will show many examples where it doesn't.
- Since we took a long enough string in the language, showed it was impossible to pump, then there cannot exist a DFA to decide L, and we must conclude that L must not be regular.

1.11 Examples

We apply the five step formula as previously described.

$A = \{0^n1^n \mid n \in \mathbb{N}\}$

Proof. Assume to the contrary, A is regular with pumping length p. Let $s = 0^p1^p$ and notice that $s \in A$ and $|s| = 2p \geq p$. There is only one case since the first p characters in the string are all zeroes. $x = 0^a,\ y = 0^b,\ z = 0^{p-a-b}1^p$ subject to $|xy| = a + b \leq p$ and $|y| = b > 0$. Choose $i = 2$. Then

$$xy^iz = xy^2z = xyyz = 0^a0^b0^b0^{p-a-b}1^p = 0^{p+b}1^p$$

We know that $b > 0$, so the number of 0s does not equal the number of 1s since $p + b > p$. Thus, A cannot be pumped, and as a result, is not regular.

□

Lets annotate this proof. The language 0^n1^n is the canonical example of a non-regular language. We choose s as a function of p so that $|s| \geq p$ is obvious. By choosing a good s, we can ensure that we reduce the number of cases required. The number of cases is technically a function of p, the number of ways $a + b \leq p$ subject to those conditions. We group these all into one case as the contradiction is identical. Note that then the substrings x, y, z also end up being a function of p. We only need to to show that one $i \neq 1$ gives a contradiction, so we choose a smallest and simplest one, that $i = 2$.

Theorem 1.7. $B = \{ww^R \mid w \in \Sigma^*\}$ is not regular.

Note that by w^R, we denote the reveral of the string w. This language, ww^R then consists of the even length palindromes.

Proof. Assume to the contrary, L_2 is regular with pumping length p Let $s = 0^{p-1}110^{p-1}$ (We are choosing a poor s on purpose). Confirm that $s \in L_2$ and $|s| = 2p \geq p$. The first p characters in the string are different, meaning there are several cases:

Case 1, y contains no 1s. Then let $x = 0^a,\ y = 0^b,\ z = 0^{p-1-a-b}110^{p-1}$ subject to $|xy| = a + b \leq p$ and $|y| = b > 0$. Choose $i = 2$. Then $xy^2z = xyyz = 0^a0^b0^b0^{p-1-a-b}110^{p-1} = 0^{p-1+b}110^{p-1}$. Since $b > 0$, we know that $p - 1 + b \neq p - 1$. Therefore, the two sections of 0s are unequal. If b makes $xyyz$ of odd length then we are done, so suppose $xyyz$ is of even length. If we were to split the string in half, the first half contains no 1s, and the second half contains two 1s, implying that this is not a palindrome.

Case 2, y contains a single 1. Then let $x = 0^a,\ y = 0^{p-1-a}1,\ z = 10^{p-1}$ subject to $|xy| = p \leq p$ and $|y| = p - 1 - a + 1 > 0$. Choose $i = 0$. Then $xy^0z = xz = 0^a10^{p-1}$ Since there is only a single 1, this is never an even-length palindrome.

For both cases, the language could not be pumped. Therefore, L_2 is not regular. □

Lets annotate this proof as well. We chose a poor s on purpose, resulting in more cases. There were two cases, whether or not xy contained a 1 or not. Had we increased the string length so that the initial block of 0's exceeded p, we would only have one case. For case b, we chose $i = 0$. We call this "pumping down". We could have chosen a worse s as $s = 0^p0^p$. Note that this is a simple even length palindrome, but it is too simple. It can be easily pumped. You want a string so that it is barely in the language, at the extremal conditions. Any small peturbation results in it no longer being in the language. Lets do another example with a better chosen s.

Theorem 1.8. $C = \{\ ww \mid w \in \Sigma^*\}$ is not regular

This language consists of words which are themselves concatenated twice. It is not $\Sigma^*\Sigma^*$, but it contains strings like $abab, abaaba, aabbaa$ and so on.

Proof. Assume to the contrary, L_3 is regular with pumping length p. Let $s = 0^p10^p1$ and notice that $s \in L_3$ and $|s| = 2p + 2 \geq p$. There is only 1 case since the first p characters in the string are all 0s, so let $x = 0^a,\ y = 0^b,\ z = 0^{p-a-b}10^p1$ subject to $|xy| = a + b \leq p$ and $|y| = b > 0$. Consider $i = 2$ so

$$xy^iz = xy^2z = xyyz = 0^a0^b0^b0^{p-a-b}10^p1 = 0^{p+b}10^p1$$

If $xyyz$ is of odd length we are done, so suppose it is of even length. Let $xy^2z = w_1w_2$ with $|w_1| = |w_2|$. Notice that w_1 ends with a 0, but w_2 ends with a 1, therefore, $w_1 \neq w_2$ and $xyyz \notin L$. Thus, L_3 cannot be pumped and is not regular. □

Lets do some unary examples.

Theorem 1.9. $L_5 = \{1^{n^2} \mid n \in \mathbb{N}\}$

Proof. Assume to the contrary, L_5 is regular with pumping length p. Let $s = 1^{p^2}$ and observe that $s \in L_5$ and $|s| = p^2 \geq p$ There is only 1 case since the first p characters in the string are all 1s. Let $x = 1^a,\ y = 1^b,\ z = 1^{p^2-a-b}$ subject to $|xy| = a + b \leq p$ and $|y| = b > 0$. Consider $i = 2$

$$xy^2z = xyyz = 1^a1^b1^b1^{p^2-a-b} = 1^{p^2+b}$$

Since $b > 0$, $p^2 < p^2 + b$, thus $|1^{p^2}| < |1^{p^2+b}|$. Since $a + b \leq p$, $b \leq p$, thus

$$p^2 + b \leq p^2 + p < p^2 + p + (p + 1) = p^2 + 2p + 1 = (p + 1)^2$$

Together, we see that

$$|1^{p^2}| < |1^{p^2+b}| < |1^{(p+1)^2}|$$

Our pumped string xy^2z falls between two squares by length in L_5. Therefore, its length is not some perfect square and is not in L_5. Thus, L_5 cannot be pumped and is not regular. □

Theorem 1.10. $L_6 = \{1^q \mid q \text{ is prime}\}$ is not regular.

Proof. Assume to the contrary, L_6 is regular with pumping length p. Let $s = 1^q$ where q is the next largest prime greater than p. By this definition, $s \in L$ and $|s| = q > p$. There is only 1 case since the first p characters in the string are all 1s. Let $x = 1^a,\ y = 1^b,\ z = 1^{q-a-b}$ subject to $|xy| = a + b \leq p$ and $|y| = b > 0$. Consider at $i = q + 1$. Then

$$xy^{q+1}z = 1^a1^{b(q+1)}1^{q-a-b} = 1^{q+qb} = 1^{q(1+b)}$$

Since $b > 0$, the length $q(1 + b)$ is a product of two numbers. Since it is composite, it is not prime, so we see that $xy^{q+1}z \notin L_6$ and thus, L_6 cannot be regular. □

For this example, how did we know to choose $i = q + 1$? I worked it out before hand, solving for i which would lead to a contradiction. Each pumping lemma proof should be done twice. Once to know the structure of the proof, and the second time formally.

Theorem 1.11. $\{0^n1^m \mid n \neq m\}$ is not regular.

Proof. Assume to the contrary F is regular with pumping length p. Consider $s = 0^p1^{p+p!}$. Observe that $s \in F$ and $|s| = p + p! > p$. We have one case, so let $x = 0^a, y = 0^b, z = 0^{p-a-b}1^{p+p!}$ subject to $a + b \leq p$ and $b > 0$. Consider $i = p!/b + 1$. Since $a + b \leq p$, we know $b \leq p$, and this implies that i is always a natural number. Then

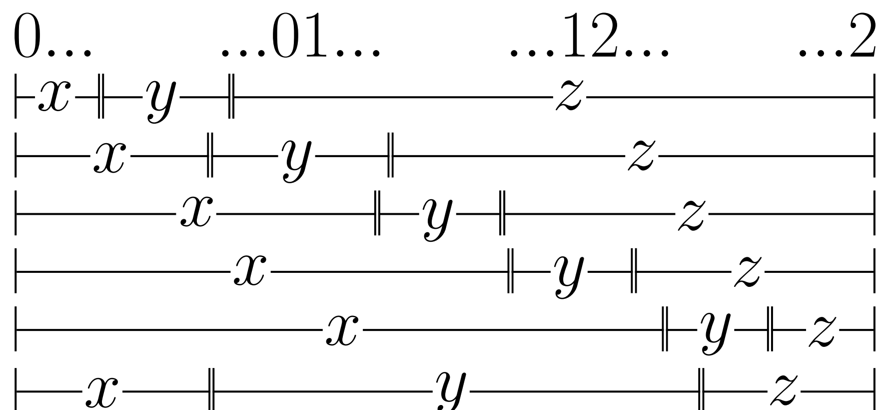
$$xy^iz = xy^{p!/b+1}z = 0^{a+b(p!/b+1)+p-a-b}1^{p+p!} = 0^{p+p!+b-b}1^{p+p!} = 0^{p+p!}1^{p+p!}$$

which is clearly not in F, and therefore, F is not regular. □

1.12 A note on choosing a bad s

You may observe the condition that $|xy| \leq p$ is not actually necessary, and this is correct. On the computation of a word, there could be may repetitions, loops on loops on loops. But we have this condition to restrict ourselves to the first appearance of such a repetition. This condition will be less general, but more useful.

Consider the language $\{0^n 1^n 2^n \mid n \in \mathbb{N}\}$. It is not regular for similar reasons to $a^n b^n$. A good choice of s is $s = 0^p 1^p 2^p$, there is only one case. Suppose we either didn't have the condition $|xy| \leq p$ or didn't choose a good s. What happens if we chose a bad s like $s = 0^{\lfloor p/3 \rfloor + 1} 1^{\lfloor p/3 \rfloor + 1} 2^{\lfloor p/3 \rfloor + 1}$? We would actually end up with six possible messy cases. Note that your proof would be incorrect if you miss enumeration of a case.



By choosing a larger s so that the first block of 0s is length p instead of length $\lfloor p/3 \rfloor + 1$, we can use the condition $|xy| \leq p$ to eliminate the cases $2 - 6$ where y may contain symbols other than 0.

The pumping lemma is not the only way to prove a language is not regular. You may apply closure properties of the regular languages.

Definition 1.6 (Dyck Language)**.** Let the language *Dyck* be over the alphabet $\Sigma = \{(,)\}$ which consists of strings of valid, balanced parenthesis.

Some strings in this language include $(), ()(), ((())()), \varepsilon, ((()())())(())()$, and some strings not in this language include $((), ()(,)(, (((((.$

Theorem 1.12. The Dyck language is not regular.

Proof. Assume to the contrary the Dyck language D was regular. Since the regular languages are closed under intersection, then $D \cap (^*)^* = \{(^n)^n \mid n \in \mathbb{N}\}$ would also be. However, we previously proved this language is not regular, contradiction. □