Notes on Regular Language Transformations

Pranay Midha

CS374 – Spring 2026

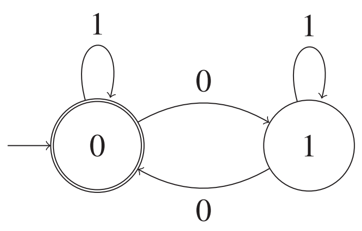
Students typically find language transformation problems to be challenging since their solutions are more abstract than direct DFA/NFA constructions we've encountered thus far in the course. In these notes, we aim to demystify the process of developing solutions to these problems by describing some useful tools and illustrating their use in example problems. We'll begin by reviewing product construction; then, we'll introduce the structure of language transformation problems and the approach we recommend to solving them along with some tips. We'll then illustrate these tips in some examples.

1 Review: Product Construction

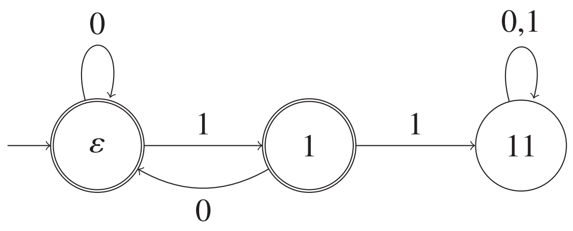
Recall that when we wanted to combine two DFAs, we used product construction to run the two DFAs *simultaneously*. For example, consider the two languages

$$L_1 \coloneqq \{w \in \{0,1\}^* \mid \#(0,w) \equiv 0 \pmod 2\}$$
$$L_2 \coloneqq \{w \in \{0,1\}^* \mid w \text{ does not contain } 11 \text{ as a substring}\}$$

We can construct a DFA for $L_1 \cap L_2$ by creating a new DFA that combines the DFAs for L_1 and L_2 using product construction.



(a) M_1: a DFA recognizing L_1



(b) M_2: a DFA recognizing L_2

Figure 1: Individual DFA constructions for L_1 and L_2

In $M_1 = (Q_1, s_1, A_1, \delta_1)$, shown in Figure 1a, we see that each state represents the parity of the number of 0s read. In $M_2 = (Q_2, s_2, A_2, \delta_2)$, shown in Figure 1b, each state represents the "progress" made towards reading substring 11.

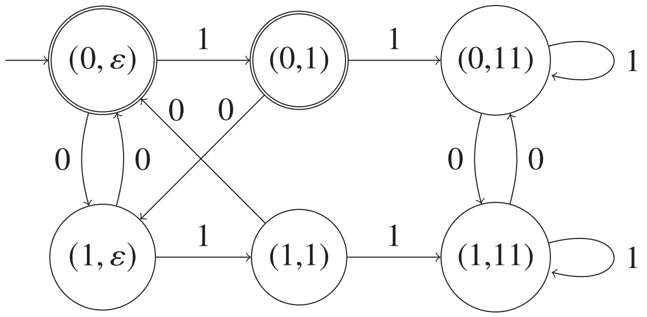


Figure 2: M: a DFA recognizing $L_1 \cap L_2$

Figure 2 describes a product DFA $M = (Q, s, A, \delta)$ where each state is a pair (p, q). Inputting a string w to M is equivalent to running M_1 and M_2 on w simultaneously, where p is the current state in M_1 and q is the current state in M_2.

Another interpretation of the product DFA is that it keeps track of *both* the parity of the number of 0s read *and* the progress made towards reading the substring 11 in the states using a tuple. Each component of the tuple corresponds to a different property of the language $L_1 \cap L_2$ that we want to keep track of. We'll use this idea to modify DFAs for arbitrary regular languages to solve these types of problems as well.

2 Problem Structure

Language transformation problems are typically structured as follows: Given a regular language L and a transformation[1] $f\colon \Sigma^* \to \Sigma^*$, we want to prove that either

$$\textsc{Transformed} := \{f(w) \mid w \in L\} \qquad \text{or} \qquad \textsc{UnTransformed} := \{w \mid f(w) \in L\}$$

are regular.

These problems can be challenging since there is no one-size-fits-all approach to solving them, but there are some strategies that can be helpful, and we will describe some of these in the subsequent sections.

Assumptions

- As is standard in this course, we'll assume $\Sigma = \{0, 1\}$, but many techniques we discuss here can be generalized to other finite alphabets.
- We'll also assume that all NFAs have ε-transitions and a single start state, and all unspecified transitions go to $\varnothing$ (for NFAs) or a dead state (for DFAs).

[1]While f may sometimes output a set of strings rather than a single string, the fundamental strategies remain the same. When actually solving problems, this distinction is not usually very important.

3 General Strategy

In general, we want to prove that some language L' that is derived from applying some transformation f to a regular language L is also regular.

Since we want to prove L' is regular, it is sufficient to construct an NFA N that recognizes L'. To help us, we can assume there is some DFA M that recognizes L since L is regular. The key strategy we'll employ is to modify M to create N that recognizes L'.

Remark 1: Why this strategy?

Recall that Kleene's Theorem states for a language L,

$$L \text{ is regular} \iff \exists \text{ a DFA } M \text{ that recognizes } L \iff \exists \text{ a NFA } N \text{ that recognizes } L$$

Thus, if we want to prove $L' := \textsc{Transformed}(L)$ (or $\textsc{UnTransformed}(L)$) is regular for any regular L, we can assume there is a DFA M that recognizes L, and it is sufficient to construct an NFA N that recognizes L'.

We usually start with a DFA for L because DFAs have a more rigid structure than NFAs, which makes it easier to modify them. On the same note, we construct an NFA for L' because NFAs are more flexible than DFAs, which makes it easier to construct them. Often the ability for NFAs to make guesses and the power of nondeterminism will be useful.

This is not the only strategy to approach these types of problems — we give an example of another approach using regular expressions in Remark 4.

3.1 Concrete Tips and Strategies

❄ Note 1: Make sure you're going in the right direction!

One of the most common mistakes we encounter is that students construct an NFA that recognizes the *opposite* of what they want. For example, consider our previous two definitions:

$$\textsc{Transformed} := \{f(w) \mid w \in L\} \qquad \text{and} \qquad \textsc{UnTransformed} := \{w \mid f(w) \in L\}$$

If we want to make an NFA for $\textsc{Transformed}$, we want to *read and accept* the transformed string $f(w)$ into our NFA N, but *check* that the original string w is in L (i.e., check that w is accepted by M).

A concrete example: Suppose $L = \{001\}$ and let $f(w) = w1$.

Now, observe that $\textsc{Transformed}(L) = \{f(w) \mid w \in L\} = \{0011\}$. Thus, when N is given the input $f(w) = 0011$, N needs to simulate the string $w = 001$ in M to check that the original string $w \in L$.

Analogously, we have $\textsc{UnTransformed}(L) = \{w \mid f(w) \in L\} = \{00\}$. Here, when N is given the input $w = 00$, N needs to simulate the string $f(w) = 001$ in M to check that $f(w) \in L$.

There are several common techniques we can use to modify M to create N, and we'll see some of these in the examples later. This list is not exhaustive, but it should give a good starting point for thinking about how to solve these problems.

- The primary question to ask when modifying M is: *What additional information do we need to keep track of in order to recognize L'?* This can guide us in constructing N since we usually want to encode this additional information using a product construction.

- Use nondeterminism to your advantage! As we discussed in Remark 1, NFAs are more flexible than DFAs, so we should use this extra power to our advantage.
  - If the transformation f involves inserting or deleting symbols, we can use ε-transitions to "guess" where these insertions or deletions occur.
  - We can also use nondeterminism to branch into multiple paths to try out different possibilities by either using ε-transitions or multiple destination states on the same symbol from a state.
  - Sometimes it is useful to "guess" the start or end of a certain substring using ε-transitions or multiple transitions as well.
  - We can interpret many transformations as breaking the input string into "phases" (e.g., before deletions, deletions, after deletions).

- If the transformation f involves reordering symbols, it can be helpful to think about how to store (or remember) symbols temporarily in the states of N until they are needed again. This can be interpreted as using product construction as memory within the NFA.

- Sometimes it is useful to use the extended transition function[2] δ^* of M to your advantage if we want to simulate reading strings in M, as we'll see in some examples.

- In any construction, we should be able to clearly describe the *meaning* of each state in N in terms of the states of M and any additional information you are keeping track of; the transitions in N should then follow from these descriptions.

Remark 2: Looking Ahead

We'll see that many of these tips will be useful later in the algorithms part of the course as well, specifically when solving graph problems — DFAs and NFAs are just graphs after all!

4 Examples

Below, we'll explore some examples that illustrate and combine some of the techniques discussed above. Note that the solutions presented are more detailed than we would expect for homework problems or exams; on assignments, we only require the formal construction of N and an explanation.

In these examples, we'll focus on transformations that delete, insert, or rearrange characters. This taxonomy is not meant to be comprehensive — there are transformations that combine these or do not fit neatly into any of these categories. However, these examples showcase the main ideas and key techniques that can be used to solve many of the problems you will encounter in this course. Several of the techniques we will

[2]Recall that δ^* is a generalization of the standard transition function δ. $\delta(q, c)$ outputs (for DFAs) the destination state (or for NFAs, the set of destination states) when reading the *character* c from state q. $\delta^*(q, w)$ outputs the destination state (or for NFAs, the set of destination states) when reading the *string* w from q.

explore can be applied to problems outside of these categories; the purpose of providing this (long) list of examples is to understand some of the tools we can use to approach challenging problems.

4.1 Deleted Characters

In this section, we explore transformations where characters are removed: we want to accept strings that are missing symbols that strings accepted by M contain. To account for this, N can simulate re-adding these missing symbols using nondeterminism even if they are not present in the input string.

First, let's consider the case where a specific substring[3] is deleted from strings in L.

Problem 1: DeleteSixSeven

For a language $L \subseteq \{0,1\}^*$, prove that if L is regular, then

$$\textsc{DeleteSixSeven}(L) := \{xy \mid x1000011y \in L\}$$

is also regular. For example, if $L = \{\varepsilon, 000, 1\underline{1000011}0, \underline{1000011}\,\underline{1000011}\}$, then we have

$$\textsc{DeleteSixSeven}(L) = \{10, 1000011\}$$

Solution 1: Some Assembly Required

Given a regular language L, let $M = (Q, s, A, \delta)$ be a DFA that recognizes L. We'll construct an NFA $N := (Q', s', A', \delta')$ that recognizes $\textsc{DeleteSixSeven}(L)$.

What direction? We read a string z into N and check if simulating re-adding 1000011 exactly once gives a string w that is accepted by M.

What information do we need to keep track of? Since we don't know where the substring 1000011 was deleted from z, we can use nondeterminism to "guess" where this deletion occurred. At any point when reading z, we can guess that the next 7 characters might correspond to the deletion. We only want to simulate re-adding 1000011 once, so we need to keep track of whether we have done this or not.

How do we implement this plan? We define N as follows:

$$\begin{aligned}
Q' &:= Q \times \{0,1\} \\
s' &:= (s, 0) \\
A' &:= \{(q,1) \in Q' \mid q \in A\} = A \times \{1\} \\
\delta'((q,k),a) &:= \{(\delta(q,a),k)\} && \text{for } q \in Q, k \in \{0,1\}, a \in \{0,1\} \\
\delta'((q,0),\varepsilon) &:= \{(\delta^*(q,1000011),1)\} && \text{for } q \in Q
\end{aligned}$$

Why does this solution work? Given an input string z, we want to accept z if re-adding 1000011 to z any exactly once in a string that is accepted by M. N simulates this by reading z and at any point non-deterministically choosing to re-add 1000011 by taking an ε-transition that simulates adding 1000011 to the string passed into M. We keep track of whether we have simulated 1000011 yet or not with the 0 and 1 product phases in the NFA (i.e., the 0 phase means we have not simulated re-adding 1000011 to z, and the 1 phase means we have).

[3] The author extends not-so-sincere apologies to the reader for the choice of substring.

In Problem 1 and its corresponding solution, the transformation deleted a specific substring 1000011. However, we can also encounter more general cases. In Problem 2, we'll see an example where an *arbitrary* substring is removed. Here, (in contrast to Solution 1) ε-transitions will be used to simulate re-adding *any* substring (rather than the specific substring we needed to simulate in Problem 1).

Problem 2: DELETEPROPERMID

For a language $L \subseteq \{0,1\}^*$, we define

$$\textsc{DeleteProperMid}(L) := \{xz \in \{0,1\}^* \mid \exists y \in \{0,1\}^* \text{ such that } |y| \geq 2 \text{ and } xyz \in L\}$$

Prove that if L is regular, then $\textsc{DeleteProperMid}(L)$ is also regular.

Solution 2: Filling in the Blanks

Given a regular language L, let $M = (Q, s, A, \delta)$ be a DFA that recognizes L. We'll construct an NFA $N := (Q', s', A', \delta')$ that recognizes $\textsc{DeleteProperMid}(L)$.

What direction? We want to read a string xz into N and check if there is some y so that inserting y between x and z gives a string in L. Notice that we don't know what x and z are ahead of time, but we can try all possibilities for where x ends and z begins using nondeterminism.

What information do we need to keep track of? One way to approach this problem is to break it into three phases: x, y, and z. In the first phase, we read x into M as normal. In the second phase, we want to simulate reading y into M (the part that was deleted). However, since we don't know what y is, we can use nondeterminism to guess when y starts and ends. Finally, in the third phase, after guessing y, we read z into M as normal. To implement this, we can add one additional piece of information to the states of M to keep track of which phase we are in.

Remark 3: Alternate Solution

Another way to approach this problem is to break it into two phases: before and after simulating inserting the characters in y. Once again, we don't know where y starts and ends, so we can use nondeterminism to guess when to insert and what characters to read for y. δ^* can come in handy when implementing this solution since $|y| \geq 2$. We leave the details as an exercise to the reader.

How do we implement this plan? We define N as follows:

$$\begin{aligned}
Q' &:= Q \times \{x, y, z\} \\
s' &:= (s, x) \\
A' &:= A \times \{z\} \\
\delta'((q,x),a) &:= \{(\delta(q,a),x)\} && \text{for } q \in Q, a \in \{0,1\} \\
\delta'((q,x),\varepsilon) &:= \{(\delta(q,0),y),(\delta(q,1),y)\} && \text{for } q \in Q \\
\delta'((q,y),\varepsilon) &:= \{(\delta(q,0),y),(\delta(q,1),y),(\delta(q,0),z),(\delta(q,1),z)\} && \text{for } q \in Q \\
\delta'((q,z),a) &:= \{(\delta(q,a),z)\} && \text{for } q \in Q, a \in \{0,1\}
\end{aligned}$$

Why does this solution work? The NFA N simulates reading a string xyz into M by breaking it down into three phases; a state (q,x) indicates that we are reading the x part of the string, and we read characters normally into M. We guess when to transition into the y phase — states of the form (q,y) — and in this phase we use ε-transitions to simulate reading characters into M from the deleted middle part of the string. Finally, we guess when to transition into the z phase, where we once again we read characters normally into M. Notice that we enforce $|y| \geq 2$ since we simulate one character when we transition $x \longrightarrow y$ and at least one character when we transition $y \longrightarrow z$. The idea behind this construction is shown in Figure 3.

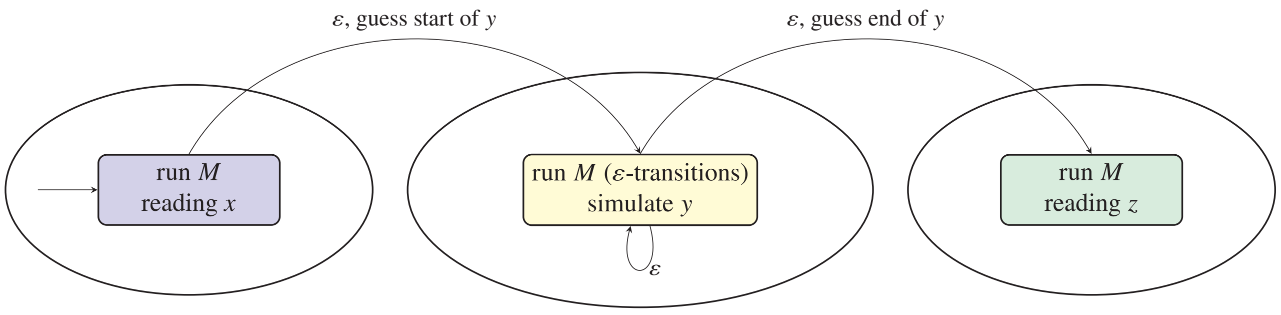

ε, guess start of y
ε, guess end of y
run M reading x
run M (ε-transitions) simulate y
ε
run M reading z

Figure 3: NFA construction described in Solution 2

Figure 3 gives a high-level illustration of the three phases in N described in Solution 2. ε-transitions allow us to guess when to transition between these phases *and* support simulating the characters in the deleted substring y into M.

Both Problem 1 and Problem 2 demonstrate a standard strategy to handle deletion transformations: using nondeterminism to guess the deleted part of the input string. We now proceed to the inverse category: transformations where the characters are added to the input string rather than removed.

4.2 Inserted Characters

In this section, we'll explore transformations that are the inverse of those we saw in Section 4.1: those that insert new characters into strings in L. Here, we'll see that N needs to simulate the "removal" of these inserted characters in M, since we'll read a string $f(w)$ with characters inserted, and we will want to check if the original w is accepted by M.

Problem 3: AnotherOne

For a string $w \in \{0,1\}^*$, we define *AnotherOne*(w) as the string obtained by inserting the character 1 at the beginning of w. For example, *AnotherOne*$(0010) = 10010$. Formally, for $w = w_1 w_2 \dots w_n$, we define *AnotherOne*$(w) := 1w_1w_2 \dots w_n$.

For a language $L \subseteq \{0,1\}^*$, prove that if L is regular, then $\textsc{AnotherOne}(L) := \{\textit{AnotherOne}(w) \mid w \in L\}$ is also regular.

Solution 3: Skipping a Character

Given a regular language L, let $M = (Q, s, A, \delta)$ be a DFA that recognizes L. We'll construct an NFA $N := (Q', s', A', \delta')$ that recognizes $\textsc{AnotherOne}(L)$.

What direction? We want to construct N to accept strings of the form $z = 1z_2 \cdots z_n$ if the string $z_2 \cdots z_n$ is accepted by M.

What information do we need to keep track of? This means we need to pass the string $z_2 \cdots z_n$ through M, and we can do this by skipping the initial 1 in z. Notice that we should *not* accepts strings that start with 0 since we know those cannot be in $\textsc{AnotherOne}(L)$. We can implement this by transitioning to M if we read an initial 1 and failing if we do not.

How do we implement this plan? We define N as follows:

$$\begin{aligned} Q' &:= Q \cup \{start\} \\ s' &:= start \\ A' &:= A \\ \delta'(start, 1) &:= s \\ \delta'(q, a) &:= \{\delta(q, a)\} && \text{for } q \in Q, a \in \{0, 1\} \end{aligned}$$

Why does this solution work? We need to simulate the input string with the starting 1 removed in M, and we do this by "ignoring" the initial 1 we read from the dummy start state. Notice that if we do not read 1 as the first character, the transition set is $\varnothing$, and the input string is rejected. After the initial 1, the rest of the string is read into M as normal in the *not start* phase of N.

Remark 4: Proof Using Regular Expressions

A clean solution to Problem 3 follows from using regular expressions. Let R be a regular expression for a regular language L. Then, $1R$ is a regular expression for $\textsc{AnotherOne}(L)$.

While it is unlikely we will *require* proofs using regular expressions, this simple and elegant solution demonstrates that this approach can be helpful when solving these types of problems.

N in Solution 3 was nicely structured because the insertion defined in Problem 3 occurred in a fixed location in the string, so we knew where to expect the extra characters (in fact, the regular expression strategy illustrated in Remark 4 can be similarly applied to many transformations which insert characters at the beginning and end of input strings). Of course, this is not true for all insertions. We'll see in Problem 4 how to handle insertions into arbitrary locations in the input string by using nondeterminism to guess the start and end of the insertions.

Problem 4: $\textsc{UnluckyInsertion}$

For a string $w \in \{0, 1\}^*$, define $\mathit{UnluckyInsertion}(w)$ to be the set of strings formed by inserting the substring 1101 to w exactly once. For a language $L \subseteq \{0, 1\}^*$, prove that if L is regular, then $\textsc{UnluckyInsertion}(L) := \{z \in \mathit{UnluckyInsertion}(w) \mid w \in L\}$ is also regular.

For example, if $L = \{\varepsilon, 0, 01\}$, then we have

$$\textsc{UnluckyInsertion}(L) = \{\underline{1101}, 0\underline{1101}, \underline{1101}0, \underline{1101}01, 0\underline{1101}1, 01\underline{1101}\}$$

Solution 4: Skipping a Substring

Given a regular language L, let $M = (Q, s, A, \delta)$ be a DFA that recognizes L. We'll construct an NFA $N := (Q', s', A', \delta')$ that recognizes $\textsc{UnluckyInsertion}(L)$.

What direction? Given an input string z to N, we want to check whether a string w resulting from removing a substring 1101 exactly once from z is accepted by M (if this is the case, then accepting z is the result of inserting 1101 to a string $w \in L$). 1101 could appear multiple times in z, so we can use nondeterminism to guess which instance of 1101 to ignore in this case.

What information do we need to keep track of? When we read z into N, we don't know when we will encounter the substring 1101, so we can use nondeterminism to guess this. We can keep track of the progress in reading the substring 1101, and simulate M before and after we read this substring.

How do we implement this plan? We define N as follows:

$$\begin{aligned}
Q' &:= Q \times \{\mathit{starting}, 1, 11, 110, \mathit{after}\} \\
s' &:= (s, \mathit{starting}) \\
A' &:= A \times \{\mathit{after}\} \\
\delta'((q, \mathit{starting}), 0) &:= \{(\delta(q, 0), \mathit{starting})\} && \text{for } q \in Q \\
\delta'((q, \mathit{starting}), 1) &:= \{(\delta(q, 1), \mathit{starting}), (q, 1)\} && \text{for } q \in Q \\
\delta'((q, 1), 1) &:= \{(q, 11)\} && \text{for } q \in Q \\
\delta'((q, 11), 0) &:= \{(q, 110)\} && \text{for } q \in Q \\
\delta'((q, 110), 1) &:= \{(q, \mathit{after})\} && \text{for } q \in Q \\
\delta'((q, \mathit{after}), a) &:= \{(\delta(q, a), \mathit{after})\} && \text{for } q \in Q, a \in \{0, 1\}
\end{aligned}$$

Why does this solution work? N guesses when to skip reading the substring 1101 nondeterministically (notice that if we read 1 in the *starting* phase, this can either be the start of 1101 or read normally into M) and only accepts if removing this substring from the input string z is accepted by M. Notice that if 1101 does not appear as a substring of z, N will get stuck in the *starting* phase, so z will not be accepted. It is also important to see that N only transitions to the *after* phase if the entire substring 1101 is read.

In Problem 3 and Problem 4, we saw examples of inserting fixed substrings. However, sometimes insertion transformations can be more general. In Problem 5, we'll see an example of such a transformation that depends on the input string. Here, we can use product construction to remember the relevant parts of the input string that encode the transformation.

Problem 5: Stutter

For a string $w \in \{0, 1\}^*$, we define $\mathit{Stutter}(w)$ as follows

$$\mathit{Stutter}(w) := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ aa \cdot \mathit{Stutter}(x) & \text{if } w = ax \text{ for some } a \in \{0, 1\} \end{cases}$$

Intuitively, *Stutter* duplicates every character in w. For example $\mathit{Stutter}(10100) = 1100110000$.

For a language $L \subseteq \{0, 1\}^*$, prove that if L is regular, then $\textsc{Stutter}(L) := \{\mathit{Stutter}(w) \mid w \in L\}$ is also regular.

Solution 5: Parity Tracking

Given a regular language L, let $M = (Q, s, A, \delta)$ be a DFA that recognizes L. We'll construct an NFA $N := (Q', s', A', \delta')$ that recognizes $\textsc{Stutter}(L)$.

What direction? For strings $w \in L$, N should accept $\mathit{Stutter}(w)$. This means, for an input string

Stutter(w) to N, we need check if w is accepted by M.

What information do we need to keep track of? Based on the description above, N needs to do two things (for an input *Stutter*(w)):

1. Verify that the input is a valid *Stutter*ed string.
2. Run the original string w in M. This means, assuming the input is valid, we need to read only *every other* character of the input into M.

To accomplish both of these, we can remember the previously read character (or ε if we have read an even number of characters) and simulate the *second* instance of a given character.

How do we implement this plan? We define N as follows:

$$\begin{aligned}
Q' &:= Q \times \{\varepsilon, 0, 1\} \\
s' &:= (s, \varepsilon) \\
A' &:= A \times \{\varepsilon\} \\
\delta'((q,\varepsilon), a) &:= \{(q, a)\} && \text{for } q \in Q, a \in \{0,1\} \\
\delta'((q,a), a) &:= \{(\delta(q,a), \varepsilon)\} && \text{for } q \in Q, a \in \{0,1\}
\end{aligned}$$

Why does this solution work? We see that N reads as input *Stutter*(w) and simulates w in M through simulating only the second instance of each repeated character in M. This is done by remembering each odd character a in a state (q, a) in N and transitioning in M on the second read of a in M. Notice that our construction *requires* each character to be repeated, since the only transition from a state (q, a) occurs when we read a again, and we must transition to an ε state to accept the input string.

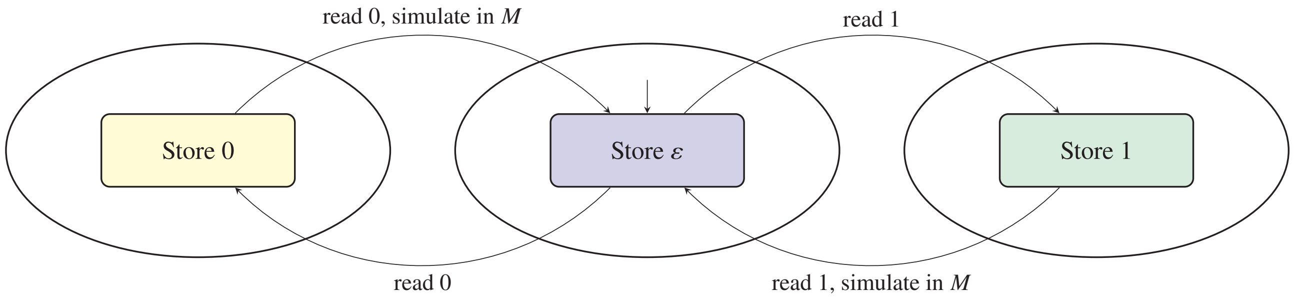


Figure 4: NFA construction described in Solution 5

In contrast to other solutions we have seen where product phases in N are linear (constructed NFAs never re-entered previously exited phases — for example, see Figure 3), Figure 4 illustrates that the application of product construction in Solution 5 is different. Here, each layer[4] "remembers" a specific piece of information, and we update the information we store as we read the input string. This allows us to freely move between layers in N as we update the "memory".

This idea of using product construction to "remember" certain information is also useful when we examine transformations that rearrange characters in input strings. These transformations tend to be more chal-

[4]One interpretation of product construction is creating multiple copies of the original DFA M (in Solution 5, we created three copies: 0, 1, and ε.). Each copy is called a *layer*. This will be a strategy we'll use frequently to solve graph problems later in the course as well.

lenging to reason through conceptually since they often combine the ideas we have seen above and more applications of nondeterminism. We'll explore some examples of these in the next section.

4.3 Rearranged Characters

Thus far, all the transformations we have seen modified the input string, but *preserved* the overall order of characters. The examples we explore in this section break this paradigm, and as alluded to earlier, have solutions that incorporate more nondeterminism as a result.

Problem 6: CycleLeft

For a string w, we define $\mathit{CycleLeft}(w)$ to be the string formed by moving the first character of w to the end of w. For example, $\mathit{CycleLeft}(01101) = 11010$. Formally, for a string $w = w_1 w_2 \cdots w_n$, $\mathit{CycleLeft}(w) = w_2 \cdots w_n w_1$ and $\mathit{CycleLeft}(\varepsilon) = \varepsilon$. We define for any $L \subseteq \{0,1\}^*$,

$$\textsc{CycleLeft}(L) := \{\mathit{CycleLeft}(w) \mid w \in L\}$$

Prove that if L is regular, then $\textsc{CycleLeft}(L)$ is also regular.

Solution 6: Guessing a Character

Given a regular language L, let $M = (Q, s, A, \delta)$ be a DFA that recognizes L. We'll construct an NFA $N := (Q', s', A', \delta')$ that recognizes $\textsc{CycleLeft}(L)$.

What direction? We want to read a string $\mathit{CycleLeft}(w)$ into N and check if the original string w is in L by reading w into M. Since the string we read is the *cycled* string, and we need to check if the *original* string is in L, the first character of w is at the end of $\mathit{CycleLeft}(w)$ (the string we actually read). We can account for this by guessing the first character of w ahead of time.

What information do we need to keep track of? We guess reading the first character at the beginning of the string, and since the last character we read is the first character of w, we don't actually want to read it into M again. Because of this, we need to keep track of our guess to make sure the last character we read actually matches our guess and one additional piece of information to guess whether we are at the end of the string or not.

How do we implement this plan? We define N as follows:

$$\begin{aligned}
Q' &:= (Q \times \{0,1\} \times \{\mathit{last}, \mathit{not\ last}\}) \cup \{\mathit{starting}\} \\
s' &:= \mathit{starting} \\
A' &:= \begin{cases} \{(q, c, \mathit{last}) \mid q \in A, c \in \{0,1\}\} \cup \{\mathit{starting}\} & \text{if } \varepsilon \in L \\ \{(q, c, \mathit{last}) \mid q \in A, c \in \{0,1\}\} & \text{otherwise} \end{cases}
\end{aligned}$$

δ' is defined as follows for all $q \in Q$ and $a, c \in \{0,1\}$:

$$\delta'(\mathit{starting}, \varepsilon) := \{(\delta(s,0), 0, \mathit{not\ last}), (\delta(s,1), 1, \mathit{not\ last})\}$$

$$\delta'((q, c, \mathit{not\ last}), a) := \begin{cases} \{(\delta(q,a), c, \mathit{not\ last})\} & \text{if } a \neq c \\ \{(\delta(q,a), c, \mathit{not\ last}), (q, c, \mathit{last})\} & \text{if } a = c \end{cases}$$

Why does this solution work? As we discussed above, we read a cycled string into N and need to check if the original string is accepted by M. Since the input to N is cycled, we need to guess the first character of the original string w at the beginning of the input, which we do using the special start state *starting*.

After guessing the first character c of w, we read the input string into M as normal. However, since N has no way to know the end of the string, if we read a character that matches c, we have the option to transition to a state that indicates we are at the end of the string (state of the form (q, c, last) — notice there are *no* outgoing transitions from these states) or proceed as normal if the character we read is not the last. Notice that we only accept if we reach an accepting state of M right after reading the guessed character c *and* that character is the end of the input string.

Finally, we account for the edge case where $\varepsilon \in L$ by allowing the start state *starting* to be accepting as well (otherwise, we would not be able to accept the empty string since we always have to guess a character at the start and expect to read that character later).

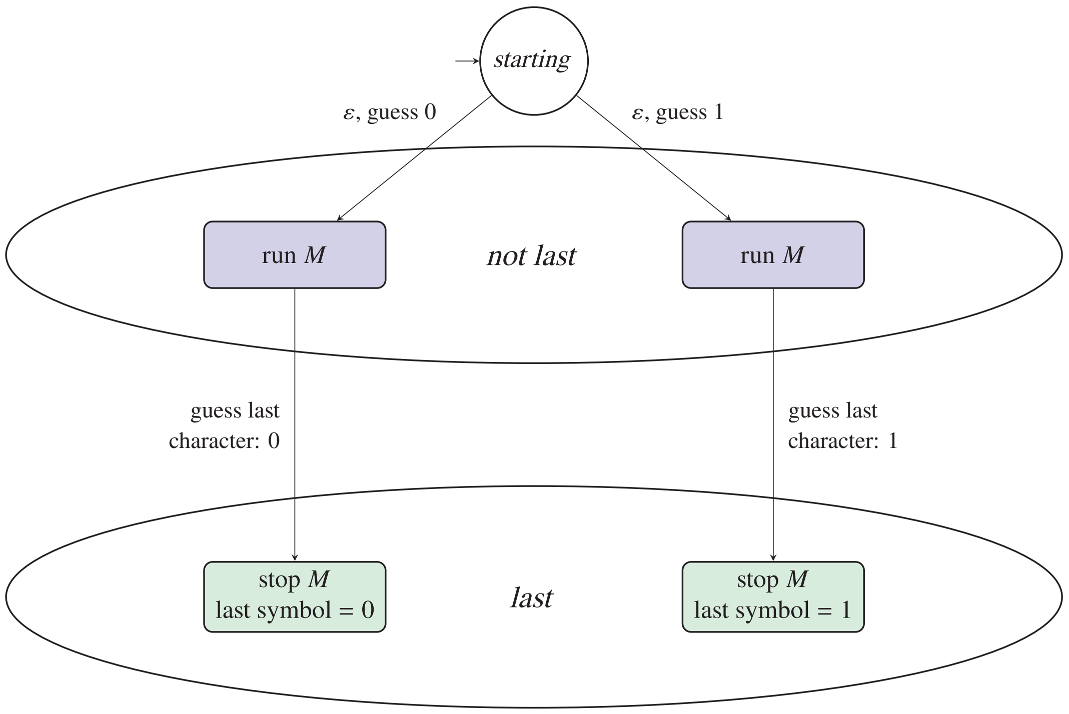


Figure 5: NFA construction described in Solution 6

The construction in Solution 6 is somewhat tedious to account for edge cases, but what is more important than the details is the main idea, which is illustrated in Figure 5. Similar to Solution 2 (and corresponding Figure 3), we see that this construction uses product states to guess the part of the string we are reading (specifically, whether we read the last character or not). However, similar to Solution 5 (and corresponding Figure 4) we also use product states as *memory* to keep track of our guess for the first character.

In Problem 6, we had to guess the ending character of the string, and store the guess ahead in our product construction. Notice that we only accepted in Solution 6 if our initial guess was correct.

In Problem 7, we'll explore the inverse: the cycled character is at the start of the string, and we have to simulate reading it at the end of the string. Here, we don't need to use the product construction to store a guess, rather, we can use it to store the relevant character and read it when we need to.

Problem 7: CycleLeft^{-1}

For a string $w \in \{0,1\}^*$, define $\mathit{CycleLeft}(w)$ as in Problem 6. For a regular language $L \subseteq \{0,1\}^*$, define $\textsc{CycleLeft}^{-1}(L) := \{w \mid \mathit{CycleLeft}(w) \in L\}$. Prove that if L is regular, then $\textsc{CycleLeft}^{-1}(L)$ is also regular.

Solution 7: Remembering a Character

Given a regular language L, let $M = (Q, s, A, \delta)$ be a DFA that recognizes L. We'll construct an NFA $N := (Q', s', A', \delta')$ that recognizes $\textsc{CycleLeft}^{-1}(L)$.

What direction? We want to read a string w into N and check if the cycled string $\mathit{CycleLeft}(w)$ is in L by reading $\mathit{CycleLeft}(w)$ into M. Since the string we read is the *un-cycled* string, and we need to check if the *cycled* string is accepted by M, N needs to shift the first character we read to the end of the string we simulate in M.

What information do we need to keep track of? To implement this, we need to remember the first character we read until the end of the string so that we can simulate reading it into M at the end. We can do this by storing one additional piece of information in the states of N to remember the first character (if any) that we read.

How do we implement this plan? We define N as follows:

$$\begin{aligned}
Q' &:= (Q \times \{0,1\}) \cup \{\mathit{start}\} \\
s' &:= \mathit{start} \\
A' &:= \begin{cases} \{(q,c) \in Q' \mid \delta(q,c) \in A\} \cup \{\mathit{start}\} & \text{if } \varepsilon \in L \\ \{(q,c) \in Q' \mid \delta(q,c) \in A\} & \text{otherwise} \end{cases} \\
\delta'(\mathit{start}, a) &:= \{(s,a)\} \qquad \text{for } a \in \{0,1\} \\
\delta'((q,c),a) &:= \{(\delta(q,a),c)\} \qquad \text{for } q \in Q, a, c \in \{0,1\}
\end{aligned}$$

Why does this solution work? As we discussed above, we want to accept strings $w = w_1 w_2 \cdots w_n$ such that $w_2 \cdots w_n w_1 = \mathit{CycleLeft}(w) \in L$. Thus, we need to simulate $\mathit{CycleLeft}(w)$ in M and see if it is accepted. To do this, when we read w_1, instead of transitioning in M, we store it in N to read into M at the end. For the rest of the characters in w, we transition in M as normal. A' is defined such that N accepts iff the state we reach in M after reading $w_2 \cdots w_n$ transitions to an accepting state when we read w_1 (so we implicitly read w_1 at the end).

Problem 6 and Problem 7 highlight two ways transformations that rearrange characters can be approached. On one hand, we can approach them by guessing the shifted character ahead of time, and on the other hand, we can remember certain patterns and run those substrings in their "shifted" positions.

5 Summary of the Tools

In Section 4, we saw example solutions spanning three *types* of transformations. However, in these solutions there were also recurring themes we used in the NFA constructions across the different types of transformations. These themes align with the general advice provided in Section 3.1, and give us a toolbox that we can use when we encounter other language transformation problems:

1. ***ε-transitions and nondeterminism for guessing:*** We saw these in almost every solution to the example problems, and these are the crux of our NFA approach. We usually have to guess *something* about the input when we are reading it, whether it is the removal of a substring, the insertion of a substring, or the positions of certain characters.

2. ***Product states as phases:*** We saw this in Solution 1, Solution 2, Solution 4, and Solution 6. When the given transformation occurs in stages, we often employed product construction to keep track of these as we read the input string. We could then use the rules of the transformation or nondeterminism to guess when to switch between phases.

3. ***Product states as memory:*** We saw this in Solution 5, Solution 6, and Solution 7. When we needed to remember certain information about the input (for example, a character), we could use product construction to carry this information forward as we read more of the input string.

Of course, many transformations are more complicated than those we covered in these notes. The official course notes have two such examples.

❃ Note 2: Not all transformations preserve regularity

It is important to note that not all transformations necessarily preserve regularity. For example, consider for a regular language L the language $\textsc{Palindromes}(L) := \{ww^R \mid w \in L\}$. Choosing $L = (0+1)^*$ gives $\textsc{Palindromes}(L)$ as the set of all even-length palindromes, which is not regular.

However, the language $\textsc{UnPalindrome}(L) := \{w \mid ww^R \in L\}$ is in fact regular when L is regular, and this is shown in the official course notes.

Nonetheless, attempting to apply and combine the tools described above will often be a good first instinct when approaching many of these problems, and understanding the general strategies can help solve more challenging problems.

Acknowledgments

Problem 2 is from an exam given in Spring 2025. Problem 6 and Problem 7 are from a practice exam given in Fall 2025. The other problems were inspired by lab problems in Fall 2025.

I am grateful to CAs Sarah Dowden, Ananya Kalidindi, Advaita Mamidipudi, Michelle Sun, and Navid Tajkhorshid for reading over an early draft of these notes. I am especially thankful to Navid for his particularly thorough feedback and many helpful suggestions.