Finite Automata

Mahesh Viswanathan

In this lecture, we will consider different models of finite state machines and study their relative power. These notes assume that the reader is familiar with DFAs, NFAs, and regular expressions, eventhough we recall DFAs and NFAs, and some classical results concerning them.

1 Deterministic Finite Automata (DFA)

Deterministic finite automata are the simplest formal model of a machine that has finitely many states, and processes an input string symbol-by-symbol to solve a decision problem. These machines are also *deterministic* in that their behavior is completely determined by the input string.

Definition 1. *A* deterministic finite automaton (DFA) *is a tuple* $M = (Q, \Sigma, \delta, s, F)$*, where*

- Q *is a finite set of* states*,*
- Σ *is a finite set called the* input alphabet *that is used to encode the input,*
- $\delta : Q \times \Sigma \to Q$ *is the* transition function*, that determines how the automaton changes its state in response to a new input symbol,*
- $s \in Q$ *is the* initial state *in which the automaton begins its computation, and*
- $F \subseteq Q$ *is the set of* accept *or* final *states.*

The behavior of a DFA M on an input string $x \in \Sigma^*$ is as follows. It reads the symbols in x one at a time, from left to right. At each point in time, the automaton has a *current state*. Initially, the automaton is in state s. If the current state of the automaton is q, and the next symbol in x to be read is a, then the automaton on reading a transitions to the state $\delta(q, a)$, and continues processing the remaining symbols of x. After all the symbols in x have been read by the automaton, if the current state of the automaton is in F then the input is said to have been *accepted* (or the automaton returns the Boolean answer *true* on the input x), and if the current state is not in F then input is *rejected* (or the automaton returns *false*).

The above informal description of computation can be convenient captured by the notion of a *run*. A run of M starting from state q on input $x = a_1 a_2 \cdots a_n$, where $a_i \in \Sigma$, is a sequence of states $q_0, q_1, \ldots q_n$ such that

- $q_0 = q$, and
- $q_{i+1} = \delta(q_i, a_i)$ for every $i \geq 0$.

Such a run corresponds to an *accepting* computation, if it starts at the initial state s and the last state in the run belongs to F. Based on the definition of a DFA, one can see that given the starting state q, and the input word x, the sequence of states in the run is *uniquely* determined.

We will also find it convenient to extend the transition function $\delta : Q \times \Sigma \to Q$ to a function $\widehat{\delta} : Q \times \Sigma^* \to Q$. The function $\widehat{\delta}$ is defined inductively as follows

$$\widehat{\delta}(q, x) = \begin{cases} q & \text{if } x = \epsilon \\ \delta(\widehat{\delta}(q, y), a) & \text{if } x = ya \end{cases}$$

In the above definition, we assume that $x, y \in \Sigma^*$, while $a \in \Sigma$. Intuitively, $\widehat{\delta}(q, x)$ is current state of the DFA after reading the string x, provided its state was q before it started reading the symbols in x. We will sometimes use $\widehat{\delta}_M$ (instead of $\widehat{\delta}$) to make explicit that we are running the DFA M.

Definition 2. *A DFA $M = (Q, \Sigma, \delta, s, F)$* accepts *input $x \in \Sigma^*$ if $\widehat{\delta}(s, x) \in F$. The* language *accepted or recognized by M is*

$$\mathbf{L}(M) = \{x \in \Sigma \mid \widehat{\delta}(s, x) \in F\}.$$

Intuitively, a DFA gives a boolean answer on any input string x; it accepts (or answers true) if the state after reading x is in F and rejects (or answers false) otherwise. The language of a DFA is the decision problem it solves, namely, the collection of inputs for which it answers true.

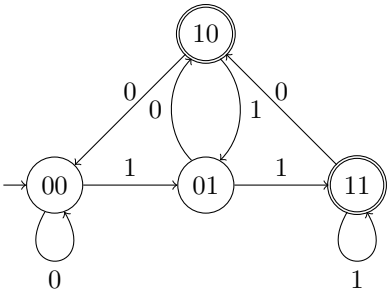

Figure 1: Transition Diagram of M_2

Example 3. *For a string $x \in \{0, 1\}$, let us define* $\text{last}_2(x)$ *to be the last two symbols in x. It can be defined as follows.*

$$\text{last}_2(x) = \begin{cases} x & \text{if } |x| < 2 \\ ab & \text{if } x = yab \text{ where } y \in \{0,1\}^*,\ a, b \in \{0,1\} \end{cases}$$

Consider the DFA $M_2 = (\{00, 01, 10, 11\}, \{0, 1\}, \delta, 00, \{10, 11\})$, where

$$\delta(u, a) = \text{last}_2(ua).$$

DFAs are often shown pictorially as a labeled directed graph, where vertices correspond to states of the DFA, and edges are transitions. DFA M_2 is shown in Figure 1.

The behavior of M_2 on a few example strings is given below.

$$\begin{array}{lll} \widehat{\delta}(00, \epsilon) = 00 & \widehat{\delta}(00, 0) = 0 & \widehat{\delta}(00, 01) = 01 \\ \widehat{\delta}(00, 011) = 11 & \widehat{\delta}(00, 0110) = 10 & \widehat{\delta}(00, 01101) = 01 \\ \widehat{\delta}(11, \epsilon) = 11 & \widehat{\delta}(11, 0) = 10 & \widehat{\delta}(11, 01) = 01 \end{array}$$

In general, we can prove by induction that, for any state $u \in \{00, 01, 10, 11\}$ and string $x \in \{0, 1\}^*$,

$$\widehat{\delta}(u, x) = \text{last}_2(ux).$$

And the decision problem solved by M_2 is

$$\mathbf{L}(M_2) = \{x \in \{0,1\}^* \mid \text{last}_2(00x) \in \{10, 11\}\} = \{x \in \{0,1\}^* \mid \text{last}_2(x) \in \{10, 11\}\}.$$

A language/decision problem L is said to be *regular* [1] if there is a DFA M that recognizes it, i.e., $\mathbf{L}(M) = L$. Since the collection of DFAs is countable and the set of languages is uncountable (see lecture on

[1] The name is derived from the fact that regular expressions describe the same class of languages as those that can be recognized by a DFAs.

infinite cardinals), there are many languages that are not regular. Standard proofs of non-regularity involve using arguments like the fooling set method or the pumping lemma; see examples in Lectures 11 and 12 of "Automata and Computability" by Dexter Kozen.

It is useful to recall a couple of classical automata constructions that allow us to conclude that the class of regular languages is closed under all Boolean operations. For a DFA $M = (Q, \Sigma, \delta, s, F)$, define the DFA $\overline{M} = (Q, \Sigma, \delta, s, \overline{F})$. In other words, $\overline{M}$ has exactly the same states, input alphabet, transition function, and initial state as M, and only the set of final states are flipped in $\overline{M}$.

Proposition 4. $\mathbf{L}(\overline{M}) = \overline{\mathbf{L}(M)}$.

Proof. Observe that since M and $\overline{M}$ have the same transition function, we have that for any state $q \in Q$ and $x \in \Sigma^*$,

$$\widehat{\delta}_M(q, x) = \widehat{\delta}_{\overline{M}}(q, x).$$

Therefore, the following reasoning establishes the proposition.

$$\begin{array}{lll} x \in \mathbf{L}(\overline{M}) & \text{iff} & \widehat{\delta}_{\overline{M}}(s, x) \in \overline{F} \quad \text{(by defn. of acceptance)} \\ & \text{iff} & \widehat{\delta}_M(s, x) \in \overline{F} \quad \text{(since } \widehat{\delta}_{\overline{M}} = \widehat{\delta}_M\text{)} \\ & \text{iff} & \widehat{\delta}_M(s, x) \notin F \quad \text{(by defn.)} \\ & \text{iff} & x \notin \mathbf{L}(M) \quad \text{(by defn. of acceptance)} \end{array}$$

□

The second important construction we will recall is the *cross-product* construction that runs two DFAs concurrenly on a common input string. Consider DFAs $M_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, s_2, F_2)$. Define the DFA $M_1 \times M_2 = (Q_1 \times Q_2, \Sigma, \delta, (s_1, s_2), F)$ where

$$\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a)).$$

We intentionally leave F unspecified for now.

Proposition 5. *If we take* $F = F_1 \times F_2$ *then* $\mathbf{L}(M_1 \times M_2) = \mathbf{L}(M_1) \cap \mathbf{L}(M_2)$. *On the other hand, if* $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$, *then* $\mathbf{L}(M_1 \times M_2) = \mathbf{L}(M_1) \cup \mathbf{L}(M_2)$.

Proof. No matter what we take F to be, we can show by induction (left as exercise), for any $(q_1, q_2) \in Q_1 \times Q_2$ and $x \in \Sigma^*$

$$\widehat{\delta}_{M_1 \times M_2}((q_1, q_2), x) = (\widehat{\delta}_{M_1}(q_1, x), \widehat{\delta}_{M_2}(q_2, x)).$$

The proposition then follows by unrolling the definition of acceptance; again it is left as an exercise for the reader. □

2 Nondeterministic Finite Automata (NFA)

Nondeterminism is an important computational abstraction that is used to model situations where computation *is not uniquely determined* by the input. It can be used to describe scenarios where we have incomplete information about external factors that influence a computation (like which process among concurrently executing processes is scheduled, or what an intruder sniffing on a network might do). It is also used as an algorithmic paradigm to search for "proofs" that establish certain properties about the input. Studying the computational power of nondeterminism has remained a central goal in computer science ever since it was first introduced by Rabin and Scott.

The simplest context in which one can understand nondeterminism is that of *nondeterministic finite automata (NFA)*. These are finite state machines that generalize DFAs by allowing the next state of the machine to be not determined by the current state and input symbol being read. Formally they are defined as follows.

Definition 6. *A* nondeterministic finite automaton (NFA) *is a tuple* $N = (Q, \Sigma, \Delta, S, F)$*, where*

- Q, Σ, $F \subseteq Q$*, are the set of states, input alphabet, and final states, respectively, as before,*
- $\Delta : Q \times \Sigma \to 2^Q$ *is transition function, which given a current state, and input symbol, determines the set of* possible *next states of the automaton, and*
- $S \subseteq Q$ *is set of possible initial/start states in which the machine could begin.*

2.1 Computation and Acceptance

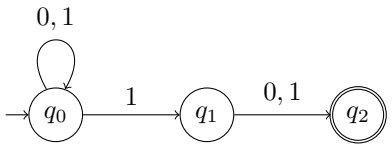

Figure 2: Transition Diagram of N_2

It useful to explain the behavior of an NFA through an example. Consider the NFA N_2 shown pictorially in Figure 2. The formal definition of $N_2 = (\{q_0, q_1, q_2\}, \{0, 1\}, \Delta, \{q_0\}, \{q_2\})$ where

$$\begin{array}{lll} \Delta(q_0, 0) = \{q_0\} & \Delta(q_0, 1) = \{q_0, q_1\} & \Delta(q_1, 0) = \{q_2\} \\ \Delta(q_1, 1) = \{q_2\} & \Delta(q_2, 0) = \Delta(q_2, 1) = \emptyset & \end{array}$$

There are two useful ways to think about nondeterministic computation. Both these views are mathematically equivalent, and in certain contexts, one view maybe more convenient than the other.

Parallel Computation View. At any given time, the machine has a few active threads which could have different current states. Initially, the machine starts threads corresponding to each of the initial states. At each step, each (currently active) thread reads the next input symbol, and "forks" a thread corresponding to each of the possible next states, given its current state. If from the current state of a thread there is no transition on the current input symbol, then the thread dies. After reading all input symbols, if there is some active, live thread of the machine that is an accepting state, the input is accepted. If none of the active threads (or if there are no active threads) is in an accept state, the input is rejected. This view is shown in Figure 3 which describes the computation of the NFA in Figure 2 on inputs 0100 and 0110.

Guessing View. An alternate view of nondeterministic computation is that the machine *magically* chooses the next state in manner that leads to the NFA accepting the input, unless there is no such choice possible. For example, again for the NFA in Figure 2 and input 0110, the machine (in this view) will magically choose to transition from q_0 to q_1 on the second 1 (and not on the first 1).

Like we did for DFAs, we can define the notion of the run of an NFA on a given input string. The definition is (almost) identical, but now there could be multiple runs on a given input, which correspond to different (complete) paths in the parallel computation view (see Figure 3). A run of NFA $N = (Q, \Sigma, \Delta, S, F)$ on input $x = a_1 a_2 \cdots a_n$ starting from state q is a sequence of states $q_0, q_1, \ldots q_n$ such that

- $q_0 = q$, and
- $q_{i+1} \in \Delta(q_i, a_i)$ for every $i \geq 0$.

Again an accepting run is one that starts in some $q \in S$ and ends in some state $q' \in F$. And an input x is accepted if N has *some* accepting run.

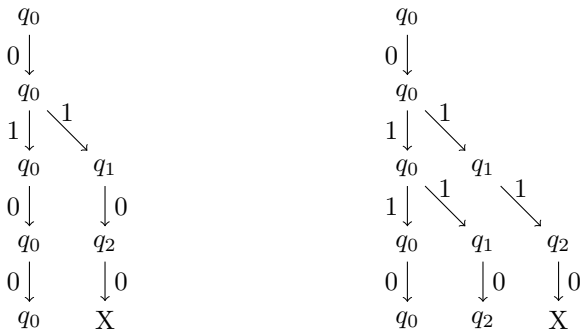

Figure 3: Computation of NFA N_2 in Figure 2. X here denotes a thread dying because of the absence of any transitions. The left tree shows the behavior on input 0100; this is not accepted since the only active thread is in state q_0 which is non-accepting. The right tree shows the behavior on input 0010. One of the threads is in an accepting state q_2 at the end, and so 0110 is accepted.

The above notions of runs and acceptance can also be conveniently captured by extending the transition function as we did for DFAs. Let $\widehat{\Delta} : Q \times \Sigma^* \to 2^Q$ be the function inductively defined as follows. (As before $x, y \in \Sigma^*$ and $a \in \Sigma$.)

$$\widehat{\Delta}(q, x) = \begin{cases} \{q\} & \text{if } x = \epsilon \\ \bigcup_{q_1 \in \widehat{\Delta}(q,y)} \Delta(q_1, a) & \text{if } x = ya \end{cases}$$

$\widehat{\Delta}(q, x)$ is the set of states in which at least one active thread of N is after reading x, provided we start with a single thread in state q. Given a set of states $A \subseteq Q$, we will take

$$\widehat{\Delta}(A, x) = \bigcup_{q \in A} \widehat{\Delta}(q, x)$$

As in the case of DFAs, we will sometimes write $\widehat{\Delta}_N$ to emphasize that we are looking at the extended transition function of NFA N.

Definition 7. *For an NFA $N = (Q, \Sigma, \Delta, S, F)$ and string $x \in \Sigma^*$, we say N* accepts *x iff $\widehat{\Delta}(S, x) \cap F \neq \emptyset$.*

The language accepted or recognized *by NFA N is $\mathbf{L}(M) = \{x \in \Sigma^* \mid N \text{ accepts } x\}$. A language L is said to be* accepted/recognized *by N if $L = \mathbf{L}(N)$.*

Every DFA is a special kind of NFA, where the transition function provides *exactly* one next state, given a current state and symbol. Thus, if a language is recognized by (some) DFA, then it can also be recognized by an NFA. It turns out the converse of this statement is also true. Thus, any language recognized by an NFA is regular. This is achieved by the so called *subset construction*, which we revist here.

In order to construct a DFA M that is equivalent to an NFA N, the DFA will "simulate" the NFA N on the given input. The computation of N on input w is completely determined by the threads that are active at each step. While the number of active threads can grow exponentially as the N reads more of the input, since the behavior of two active threads in the same state will be the same in the future, the DFA does not need to keep track of how many active threads are in a particular state; the DFA only needs to track whether there is an active thread in a particular state. Thus, to simulate the NFA, the DFA only needs to maintain the current set of states of the NFA.

The formal construction based on the above idea is as follows. Let us consider an NFA $N = (Q, \Sigma, \Delta, S, F)$. Define the DFA $2^N = (2^Q, \Sigma, \delta, S, F')$, where

$$\delta(A, a) = \cup_{q \in A} \Delta(q, a)$$

and $F' = \{A \subseteq Q \mid A \cap F \neq \emptyset\}$.

Proposition 8. *For any NFA N,* $\mathbf{L}(N) = \mathbf{L}(2^N)$.

Proof. The proposition is proved by inductively establishing the following claim.

$$\forall A \subseteq Q.\, \forall x \in \Sigma^*.\, \widehat{\Delta}_N(A, x) = \widehat{\delta}_{2^N}(A, x)$$

Proving the above claim is left as an exercise. The equivalence of the languages of N and 2^N then follows from the way we defined the set of final states F' of 2^N. □

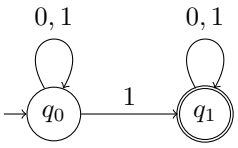

Figure 4: NFA N

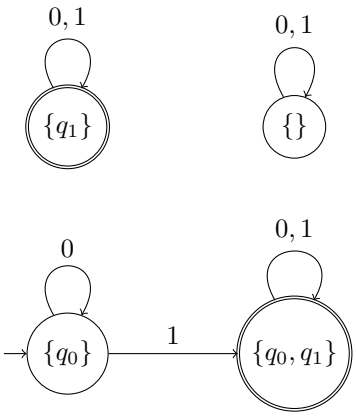

Figure 5: DFA 2^N equivalent to N

We conclude this section with an example NFA N (Figure 4) and its equivalent DFA 2^N (Figure 5).

3 Universal Finite Automata (UFA)

NFAs generalize DFAs by allowing for multiple computations on a given input string. But they also make a choice in terms of when an input is accepted — an NFA accepts an input string if *some* execution ends in a final state. One could consider other modes of acceptance, in particular, the one where an input is accepted if *every* computation ends in a final state. This leads us to *universal finite automata*.

Definition 9. *A* universal finite automaton (UFA) *is a tuple $U = (Q, \Sigma, \Delta, S, F)$ where Q, Σ, S, and F are the set of states, input alphabet, initial states, and final states, respectively, just like for NFAs (Definition 6). The transition function $\Delta : Q \times \Sigma \to (2^Q \setminus \{\emptyset\})$ maps a current state and input symbol to a* non-empty *set of next states.*

Remark. The requirement that the transition function Δ map a state and input symbol to a non-empty set of next states is a technical condition that we impose, which ensures that in the parallel computation view of such machines, *no thread dies* in the middle of processing the input. This makes defining the notion of acceptance cleaner and less challenging.

The way to think of computation of a UFA on an input string x is similar to the parallel view of nondeterministic computation. Initially, the machine starts with a one thread in each of the initial states in S. At each step, each thread reads the next input symbol, and "forks" a thread corresponding to each of the possible next states, given its current state. After reading the entire input, if the state of *every* thread is final, then the input is accepted. If some thread is an non-final state, the input is rejected.

The above intuitive view computation can be captured using runs. Runs for UFAs are defined in exactly the same manner as for NFAs, and its definition is skipped. Similarly, we can extend the transition function of a UFA in exactly the same manner. An input x is accepted if the last states of all runs on x end in a final state. We can restate this definition as follows.

Definition 10. *A UFA $U = (Q, \Sigma, \Delta, S, F)$ is said to* accept *an input x iff $\widehat{\Delta}(S, x) \subseteq F$. The* language accepted/recognized *by U is $\mathbf{L}_\forall(U) = \{x \in \Sigma^* \mid U \text{ accepts } x\}$.*

Once again since DFAs are a special class of UFAs, we can conclude that every regular language is recognized by UFAs. In addition, like for NFAs, we can also prove the converse — any language recongnized by a UFA is regular.

Proposition 11. *For any UFA U, $\mathbf{L}_\forall(U)$ is regular, i.e., can be recognized by a DFA.*

Proof. There are a couple of ways we can argue this. First, we can observe that the DFA 2^U with final states $F' = 2^F$, where F is the set of final states of U, accepts the same language as U.

The second proof relies on observing the correspondence between UFAs and NFAs. Let the UFA $U = (Q, \Sigma, \Delta, S, F)$. Consider the NFA $\overline{U} = (Q, \Sigma, \Delta, S, \overline{F})$ [2]. One can prove that

$$\mathbf{L}_\forall(U) = \overline{\mathbf{L}_\exists(\overline{U})}.$$

Here we are using $\mathbf{L}_\exists(\overline{U})$ (instead of $\mathbf{L}(N)$) to emphasize the fact that $\overline{U}$ is being interpreted an an NFA, which accepts strings when *some* run is accepting. The proof of this fact is left as an exercise. We know that $\mathbf{L}_\exists(\overline{U})$ is regular (Proposition 8) and regular languages are closed under complementation (Proposition 4). These facts together imply that $\mathbf{L}_\forall(U)$ is regular. □

NFAs and UFAs correspond to two extremes of defining acceptance for a machine that can have multiple computations on a given input string. NFAs accept if *some* computation/run/thread accepts, while UFAs accept if *all* computations/runs/threads accept. One can imagine more complicted conditions on the set of runs on an input to define when it is accepted. A general form of acceptance can be defined through the notion of *alternation*, which we will introduce in the context of Turing machines. To see how alternating finite automata work, and their computational power, you can read miscellaneous exercise 59 of "Automata and Computability" by Dexter Kozen.

4 2-way Deterministic Finite Automata (2DFA)

All the finite automata we have introduced so far (DFA/NFA/UFA) read the symbols of the input from left to right. None of these machines have the ability to *go back* and re-read an input symbol they have already processed. In this section, we introduce two-way automata that can read the input string in either direction, and can choose to re-read some symbols they have already seen. Such two-way machines can be either *deterministic* or *nondeterministic*. We will only focus our attention on the deterministic model.

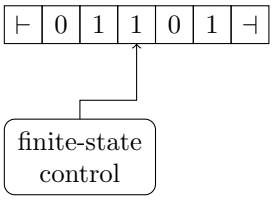

Figure 6: Two way deterministic finite automata

A two way deterministic finite automata is schematically shown in Figure 6. The symbols of the input are thought of occupying cells of a finite tape. The leftmost and rightmost cells of this tape are assumed to contain a left ($\vdash$) and right ($\dashv$) endmarker. These endmarkers enable the machine to know when the left and right end of the input string have been reached, and they are assumed to be not part of the alphabet Σ used to encode the input.

[2] NFAs and UFAs are in some sense the same kind of machine. The only difference is in the way we define acceptance.

The machine starts in its initial state s with its tape head pointing to the left endmarker. At any point in time, the machine reads the symbol written on the tape cell currently being scanned by the head, and based on this symbol and the current state of the machine, it moves its tape head either left or right, and changes its state. It accepts the input if it reaches a special accept state t and rejects if it reaches the reject state r. In either of these cases the computation *halts*. However, it is possible that the 2DFA may never halt and loop. In this case, the input is assumed to be *rejected* as well.

Definition 12. *A 2DFA is a tuple $M = (Q, \Sigma, \vdash, \dashv, \delta, s, t, r)$ where*

- *Q is the (finite) set of states,*
- *Σ is the input alphabet,*
- *$\vdash$ is the left endmarker, with $\vdash \notin \Sigma$,*
- *$\dashv$ is the right endmarker, with $\dashv \notin \Sigma$,*
- *$\delta : (Q \setminus \{t, r\}) \times (\Sigma \cup \{\vdash, \dashv\}) \to (Q \times \{\mathsf{L}, \mathsf{R}\})$ is the transition function which given a current state and symbol being read, describes what the next state and direction (left/right) in which the input head is moved; we assume that no transition is defined if the current state is t or r,*
- *$s \in Q$ is the start state,*
- *$t \in Q$ is the accept state, and*
- *$r \in Q$ is the reject state, with $t \neq r$.*

We also assume that the transition function is such that the 2DFA always moves right on reading $\vdash$ and left on reading $\dashv$. In other words, for every $q \in Q \setminus \{t, r\}$,

$$\begin{aligned} &\delta(q, \vdash) = (p, \mathsf{R}) \quad \text{for some } p \in Q, \\ &\delta(q, \dashv) = (p, \mathsf{L}) \quad \text{for some } p \in Q. \end{aligned}$$

Example 13. *Let us design a 2DFA for the language accepted by the DFA M_2 (Example 3 and Figure 1). Recall that the language we are interested in recognizing is*

$$L_2 = \{x \in \{0,1\}^* \mid \text{last}_2(x) \in \{10, 11\}\}$$

The 2DFA M_* will work as follows. It will move the input head right until the right endmarker is reached. Then it will move it's head 2 steps to the left. If the symbol read is 1 then it will halt and accept. Otherwise it will halt and reject.

Formally, $M_* = (\{s, p, q, t, r\}, \{0, 1\}, \vdash, \dashv, \delta, s, t, r)$ where δ is defined as follows.

$$\begin{aligned} &\delta(s, a) = (s, \mathsf{R}) \text{ for any } a \in \{\vdash, 0, 1\} && \delta(s, \dashv) = (p, \mathsf{L}) \\ &\delta(p, a) = (q, \mathsf{L}) \text{ for any } a \in \{0, 1, \dashv\} && \delta(p, \vdash) = (q, \mathsf{R}) \\ &\delta(q, 0) = (r, \mathsf{R}) && \delta(q, 1) = (t, \mathsf{R}) \\ &\delta(q, \vdash) = (q, \mathsf{R}) && \delta(q, \dashv) = (q, \mathsf{L}). \end{aligned}$$

4.1 Configurations and Computations

Recall that we defined the run of DFA to be a sequence of states that is consistent with the symbols in the input and the transition function. While states are sufficient to determine what the next step of a DFA will be, for 2DFA we need to know both the state and the position of the tape head so that we know which symbol is being read. Thus, computations or runs of a 2DFA will be a sequence of pairs that consist of the state and input head position at that step. We will define this precisely.

Let us fix a 2DFA $M = (Q, \Sigma, \vdash, \dashv, \delta, s, t, r)$ and an input $x = a_1 a_2 \cdots a_n$. Recall that the 2DFA is started with the input between the endmarkers. Thus, taking $a_0 = \vdash$ and $x_{n+1} = \dashv$, the 2DFA is executed on a tape

containing the string $a_0a_1a_2\cdots a_na_{n+1} = \vdash x \dashv$. The tape head, at any given point, maybe scanning one of the cells between 0 and $n+1$.

A *configuration* of M on input x is a pair (q,i) where $q \in Q$ and $0 \leq i \leq n+1$. Since initial M is in state s and is scanning the leftmost cell, the *start configuration* is $(s,0)$. The *next configuration relation*, $\xrightarrow[x]{1}$, which describes step of the machine on input x, is defined as follows.

$$\delta(p,a_i) = (q,\mathsf{L}) \Rightarrow (p,i) \xrightarrow[x]{1} (q,i-1),$$
$$\delta(p,a_i) = (q,\mathsf{R}) \Rightarrow (p,i) \xrightarrow[x]{1} (q,i+1).$$

Using the next configuration relation, we can define the notion of a run. A *run* of M on input x from configuration c is finite or infinite sequence of configurations $c_0, c_1, \ldots$ such that

- $c_0 = c$,
- $c_i \xrightarrow[x]{1} c_{i+1}$ for every $i \geq 0$, and
- if the sequence is finite, then the last configuration is either (t,j) or (r,j) for some j.

In the above definition, the possibility that the run is an infinite sequence accounts for the fact that M may loop and never halt. The last condition says that M halts only if either reaches the accept state t or the reject state r. Since M is deterministic, it has a *unique* run starting from any configuration c. The input is *accepted* if M's run on x starting from $(s,0)$ reaches a configuration where the state is t.

We can also define acceptance of inputs by defining the n-fold composition of $\xrightarrow[x]{1}$, which is defined inductively as follows.

$$(p,i) \xrightarrow[x]{0} (p,i) \text{ for every configuration } (p,i)$$
$$\text{If } (p,i) \xrightarrow[x]{n} (q,j) \text{ and } (q,j) \xrightarrow[x]{1} (u,k), \text{ then } (p,i) \xrightarrow[x]{n+1} (u,k)$$

Finally, we will say configuration (q,j) is *reached (in zero or more steps)* from (p,i), if for some n, $(p,i) \xrightarrow[x]{n} (q,j)$; we denote this by $(p,i) \xrightarrow[x]{*} (q,j)$.

Input x is *accepted* by M iff for some i, $(s,0) \xrightarrow[x]{*} (t,i)$. The *language* accepted/recognized by M is

$$\mathbf{L}(M) = \{x \in \Sigma^* \mid M \text{ accepts } x\}.$$

Example 14. *The 2DFA M_* from Example 13 has the following runs.*

$$x_1 = 00110: \quad (s,0) \xrightarrow[x_1]{1} (s,1) \xrightarrow[x_1]{1} (s,2) \xrightarrow[x_1]{1} (s,3) \xrightarrow[x_1]{1} (s,4) \xrightarrow[x_1]{1} (s,5) \xrightarrow[x_1]{1} (s,6) \xrightarrow[x_1]{1} (p,5) \xrightarrow[x_1]{1} (q,4) \xrightarrow[x_1]{1} (t,5)$$
$$x_2 = 00100: \quad (s,0) \xrightarrow[x_2]{1} (s,1) \xrightarrow[x_2]{1} (s,2) \xrightarrow[x_2]{1} (s,3) \xrightarrow[x_2]{1} (s,4) \xrightarrow[x_2]{1} (s,5) \xrightarrow[x_2]{1} (s,6) \xrightarrow[x_2]{1} (p,5) \xrightarrow[x_2]{1} (q,4) \xrightarrow[x_2]{1} (r,5)$$

Given the above runs, M_* accepts 00110 *and rejects* 00100.

4.2 Equivalence between 2DFA and DFA

Eventhough 2DFAs have the ability to re-read parts of the input if needed, they are not more powerful than (1-way) DFAs. Every language recognized by a 2DFA is regular. We will prove this observation by constructing a NFA that recognizes the same language as a given 2DFA; the result will then follow based on the equivalence of NFAs and DFAs (Proposition 8).

The behavior of a 2DFA can be captured by the sequence of states of the machine as it crosses the boundary of each tape cell. For example, consider the computation of M_* (Example 6) on the input 00110.

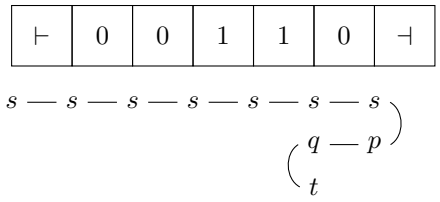

Figure 7: Behavior of 2DFA M_* on input 00110 that illustrates crossing sequences.

This is shown in Figure 7. By convention, we assume that the computation begins by the machine crossing the left boundary of cell 0 in the initial state s. On reading $\vdash$, M_* moves right without changing its state. Thus, the first state when the machine crosses the left boundary of cell 1 is s, and it moves to the right as it crosses this boundary. As the computation proceeds, M_* crosses the left boundary of each cell in state s moving right, until it reads the right endmarker. After reading $\dashv$, it moves left, crossing the left boundary of cell 6 in state p, moving left. It then crosses the left boundary of cell 5 in state q, and after reading 1 in cell 4, it re-crosses the left boundary of cell 5 in state t.

A *crossing sequence* is the sequence of states of a 2DFA as it crosses a cell boundary. Assuming that the input is $x = a_1 a_2 \cdots a_n$, we denote the crossing sequence on the left boundary of cell i ($0 \leq i \leq n+1$) as σ_i. By convention, we take $\sigma_0 = s$, for any input x. For example, in the computation shown in Figure 7, $\sigma_5 = s, q, t$.

The following observations about crossing sequences are useful. If input x is accepted by 2DFA M, then no crossing sequence may have a repeated state with the head moving in the same direction. This is because if we have such a repeated state, then it means that M (because it is deterministic) has entered a loop and will never terminate. Second, the first time a boundary is crossed the head must be moving right. Subsequent crossing must be in alternate directions. Thus, every odd numbered elements of a crossing sequence represent right moves, while every even numbered elements represent left moves. Given the first observation in this paragraph about looping, we have that if x is accepted, no crossing sequence can a state appear twice in odd numbered positions or even numbered positions. Thus the length of any crossing sequence during an accepting computation is at most $2|Q|$.

A crossing sequence is *valid* if it is a sequence $q_1, q_2, \ldots q_k$ such that no two odd position elements are equal and no two even positioned elements are equal. In an accepting computation, all crossing sequences are valid. Since the length of a valid crossing sequence is bounded by $2|Q|$, there are *finite* number of valid crossing sequences. In fact the number of valid crossing sequences is bounded by $|Q|^{2|Q|}$.

Our strategy for constructing the NFA N equivalent to the 2DFA M will be as follows. N will check if the input x is accepted by M by guessing the crossing sequences σ_i during M's computation on x, as it reads the input; of course, σ_0 is fixed to be s. Now suppose N has guessed the sequence σ_i, and after reading a_i, N guesses σ_{i+1}. N must make sure that the guesses of σ_i and σ_{i+1} (which are the two boundary crossing sequences of cell i) are consistent with the fact that cell i is holding the symbol a_i. Thus we need to carefully define what it means for two crossing sequences to be consistent with respect to the contents of a given cell.

We will say that a sequence $\sigma = p_1, p_2, \ldots p_\ell$ *right matches* a sequences $\sigma' = q_1, q_2, \ldots q_k$ on symbol a, if σ and σ' are consistent provided M enters cell a in state p_1 moving right. Similarly, we will say that a sequence $\sigma = p_1, p_2, \ldots p_\ell$ *left matches* a sequences $\sigma' = q_1, q_2, \ldots q_k$ on symbol a, if σ and σ' are consistent provided M enters cell a in state q_1 moving left. We will define the notion of left and right matching, inductively, as follows.

- $\sigma = \epsilon$ and $\sigma' = \epsilon$ both left and right match on a. That is, if one never reaches the cell containing a, the two boundary crossing sequences must be empty.
- If $p_3, p_4, \ldots p_\ell$ right matches $\sigma' = q_1, \ldots q_k$ and $\delta(p_1, a) = (p_2, \mathsf{L})$ then σ right matches σ' on a. This describes the situation when we cross into cell a in state p_1 and we immediately move left in state p_2.

- If $p_2, p_3, \ldots p_\ell$ *left* matches $q_2, q_3, \ldots q_k$ and $\delta(p_1, a) = (q_1, \mathsf{R})$ then σ right matches σ'. This captures the situation when after entering cell a in state p_1, we move across the right boundary in state q_2, before returning back to the cell from the right boundary of a. Notice, this case is the reason why we need to also define the notion of left matches.

- If $p_1, p_2, \ldots p_\ell$ left matches $q_3, q_4, \ldots q_k$ and $\delta(q_1, a) = (q_2, \mathsf{R})$ then σ left matches σ' on a. The reasoning for this is similar to the first non-base case for right matches.

- Finally, if $p_2, p_3, \ldots p_\ell$ right matches $q_2, \ldots q_k$ and $\delta(q_1, a) = (p_1, \mathsf{L})$ then σ left matches σ'. Again this is similar to the second non-base case for right matching.

Having defined the notion of two crossing sequences right matching on an input symbol, we ready to give the formal definition of the NFA N based on the intuition we outlined above. The NFA $N = (Q', \Sigma, \Delta, \{(s, 0)\}, F')$ where

- $Q' = V \times \{0, 1\}$, where V is the set of all valid crossing sequences. The state of N remembers its current guess for the crossing sequence on the left boundary of the symbol N is going to read, as well as whether N has already guessed a crossing sequence where the last state is the accept state t.

- For a state (σ, b), where σ is a valid crossing sequence, and $b \in \{0, 1\}$,

$$\Delta((\sigma, b), a) = \{(\sigma', b') \mid \sigma \text{ right matches } \sigma' \text{ on } a \text{ and, if } \sigma' = \sigma_1 t \text{ then } b' = 1; \text{ otherwise } b' = b\}$$

  In other words, in the next state (σ', b'), the crossing sequence σ' must be consistent with σ given symbol a, and if the last state of σ' is the accepting state t then $b' = 1$; otherwise b' is the same as b.

- $F' = \{(\sigma, 1) \mid \sigma \in V\}$. So the input is accepted, if at some point before reading all the input symbols, N guessed a crossing sequence whose last state is t.

Now, we need to argue that $\mathbf{L}(N) = \mathbf{L}(M)$. It is easy to see that $\mathbf{L}(M) \subseteq \mathbf{L}(N)$. This is because if input x is accepted by M, then N can guess the crossing sequences in the accepting computation of M on x, and N would accept. To prove the converse, is more challenging, and it is left as a homework problem.

A couple of alternate (simpler?) proofs for this result can be found in the book "Automata and Computability" by Dexter Kozen. See lecture 18 and miscellaneous exercise 61.