Nondeterministic Finite Automata

Bakhadyr Khoussainov

Computer Science Department, The University of Auckland, New Zealand
bmk@cs.auckland.ac.nz

1 Definitions and Examples

We start with an example. For a fixed natural number $n \geq 1$, consider the following language

$$L = \{uav \mid u \in \{a, b\}^\star \text{ and the length of } v \text{ is } n\}.$$

Thus, L consists of all strings w that have an a in the $(n+1)^{th}$th position from the right of w. Assume that we are asked to design a DFA recognizing this language and draw its transition diagram. Trying to design a DFA recognizing this language, one can soon realize that such a DFA should have many states. We will prove later on that there exists a DFA with 2^n that recognizes this language. Moreover, no DFA with less that 2^n states recognizes L. An interesting observation, however, is this.

Imagine, we are reading an input string w from $\{a, b\}^\star$ to determine whether w is in L. We read w until we see an a. If no a is found then we reject w. Let's assume that we have just seen the first a in w. We can make one of the following two *guesses*:

1. The symbol a we have just seen is in the $(n+1)^{th}$ position from the right of w.
2. The symbol a we have just seen is *not* in the $(n+1)^{th}$ position from the right of w.

Suppose that we select the first guess. We change our state, and try to verify our guess. Let's call this state the *verification* state. In order to verify the correctness of our guess we just need to read the rest of the string and check if the length of the rest of the string is n. Verifying that the rest of the string has length n is easy. We set the variable $length$ to 0 and increment the variable by 1 each time after we read the next input symbol. If the length of the rest of the string is strictly less than n, then the guess is incorrect. Now assume that $length = n$. If the string does *not* have any more symbols left then the guess is correct. Otherwise, the guess is incorrect. In this case, we need to go back to the position of w where we made the guess, and select the second guess.

Suppose we select the second guess. Let's call this state the *passive* state. This means we can continue on with the rest of the string just passing through the bs until we encounter the next a. When we see an a, we again repeat the same guessing process either by selecting the *verification* state or the *passive* state. Thus, while reading w, once we see an a we can either stay in the *passive* state or we can select the *verification* state and verify the correctness of our guess.

An important note here is this. If $w \in L$ then w is of the form uav where the length of v is n. Therefore, we can act as follows. We stay in the *passive* state until we read through the

entire string u. Once we finish reading u, we select the *verification* state and then verify the correctness of our guess which clearly tells us that our guess after processing u was correct. Now suppose that $w \notin L$. Then *all* our guesses when we select the *verification* state will never verify that our guess was correct.

The *passive* state is thus the **nondeterministic state**. The reason for that is this. When we are in the *passive* state and read the symbol a then it is *not* determined which transition we should make. Namely, we can make one of the following two transitions. One is to stay in the *passive* state, and the other is to move to the *verification* state. In Figure 1 we represent our analysis above by a finite state machine for the case when $n = 3$.

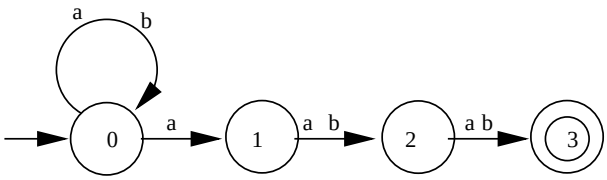

Fig. 1. An example of an NFA

In this figure the initial state plays the role of the *passive state* and the state after the initial state plays the role of the *verification state*. Keeping this example in mind we now give the following definition, central to this lecture:

Definition 1. *A* **nondeterministic finite automaton** *is a* 5*-tuple* (S, I, T, F, Σ)*, where*

1. S *is the set of* **states**.
2. $I \subseteq S$ *is the set of initial states.*
3. T *is the* **transition function** $T : S \times \Sigma \to P(S)$*, where* $P(S)$ *is the power set of* S.
4. F *is a subset of* S *called the set of* **accepting states**.
5. Σ *is a given alphabet.*

We abbreviate nondeterministic finite automata as NFA*. We often write* (S, I, T, F) *instead of* (S, T, I, F, Σ) *as it will be clear from* T *which alphabet* Σ *is used. We use letters* $\mathcal{A}$*,* $\mathcal{B}$*, ... to denote NFA. We also often say* **automata** *for short instead of saying nondeterministic finite automata.*

As for DFA from the previous lecture, we can visualize an NFA (S, I, T, F) as a labeled graph. The states of the NFA are vertices of the graph. We put an edge from state s to state p and label it with σ if $p \in T(s, \sigma)$. These are called σ**-transitions**. The initial states are the vertices that have in-going arrows without the source. The final states are doubly circled. We call this visual presentation a **transition diagram** of the automaton. Since $T : S \times \Sigma \to P(S)$ it may well be the case that $T(s, \sigma) = \emptyset$ for some $s \in S$ and $\sigma \in \Sigma$.

Every DFA can be counted as an NFA but the difference between DFA and NFA is this. Every state in a DFA always has *exactly* one σ-transition for every $\sigma \in \Sigma$. Contrary to this, in

an NFA there could be states that have none, one, two or more σ-transitions. We call a state s of an NFA a **nondeterministic state** if the state has more than one σ-transitions for some $\sigma \in \Sigma$. In Figure 2 we present another NFA with 6 states.

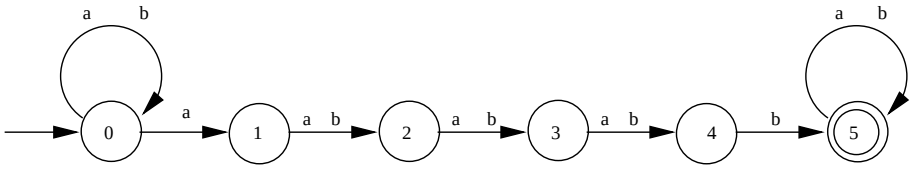

Fig. 2. Another example of an NFA

Formally, this automaton is defined as follows:

1. $S = \{0, 1, 2, 3, 4, 5, 6\}$.
2. $I = \{0\}$.
3. The table of the transition function T is this:

	a	b
0	$\{0, 1\}$	$\{0\}$
1	$\{2\}$	$\{2\}$
2	$\{3\}$	$\{3\}$
3	$\{4\}$	$\{4\}$
4	$\emptyset$	$\{6\}$
5	$\{6\}$	$\{6\}$

4. $F = \{6\}$.

Let $\mathcal{A} = (S, I, T, F, \Sigma)$ be an NFA and w be a string $\sigma_1\sigma_2 \ldots \sigma_n$. How does the automaton $\mathcal{A}$ run on the input string w? We define this below and then explain the definition in more detail.

Definition 2. *A* **run** *of the automaton $\mathcal{A}$ on the string $\sigma_1\sigma_2 \ldots \sigma_n$ is a sequence of states*

$$s_1, s_2, \ldots, s_n, s_{n+1}$$

such that $s_1 \in I$ and $s_{i+1} \in T(s_i, \sigma_i)$ for all $i = 1, \ldots, n$.

A run of $\mathcal{A}$ on string $w = \sigma_1 \ldots \sigma_n$ can be thought of as the execution the following randomized algorithm $Run(\mathcal{M}, w)$:

1. Initialize $i = 0$ and randomly choose an $s \in I$.
2. Print s.

3. *While* $i \leq n$ do
   (a) Set $\sigma = \sigma_i$.
   (b) Randomly choose a state q from $T(s, \sigma)$.
   (c) Set $s = q$ and print s.
   (d) Increment i

This algorithm outputs the states of the DFA $\mathcal{M}$ as it reads through the string w. First, the algorithm prints some of the initial states. It then reads the first input σ_1, makes a random transition from the initial state to one of the states q in $T(q_0, \sigma_1)$, prints q , reads the next symbol σ_2, makes a random transition from q, and so on. The most important thing to remember is that there could be *several runs* of the automaton on the input w.

Another way of looking at how $\mathcal{A}$ computes on $w = \sigma_1\sigma_2 \ldots \sigma_n$ is this. Say, $\mathcal{A}$ has k initial states $q_1, \ldots, q_k$. Before the automaton starts reading w, it splits itself into k many copies where the i^{th} copy is at state q_i, $i = 1, \ldots, k$. Each copy of $\mathcal{A}$ now reads the first symbol σ_1. Assume that the copy is at state q, and there are t transitions from q labeled with σ. The copy now splits itself into t many copies where each moves along one of the transitions. This continues as before. Thus, at any given stage of the computation there are several copies of the automaton reading the same symbol in *parallel*. These copies all make independent transition by splitting themselves and thus making more copies if that is dictated by the transition function. If a copy of the automaton is in a state where there are *no* transitions, then the copy dies.

Another helpful way to think about how $\mathcal{A}$ computes on w (of length n) is to represent this as a tree. The root of the tree is the start of the computation. Every branching node of the tree represents the fact that $\mathcal{A}$ splits itself at that node by reading the next symbol from w. The height of the tree is at most $n + 1$. The paths of length $n + 1$ from the root to the leaves correspond to runs of the automaton on w. For example, consider the automaton represented in Figure 1. Then its computation on *ababaab* can be represented by the tree in Figure 3. In this figure we can see that the two rightmost paths die out because at state 3 the automaton can not make any transition. The other paths produce runs of the automaton on the input. One of these runs is such that the end state of the run is the accepting state. Now we are ready to the next definition:

Definition 3. *We say that a NFA* $\mathcal{A}$ **accepts** *a string* $w = \sigma_1\sigma_2 \ldots \sigma_n$ *if the automaton has a run* $s_0, s_1, s_2, \ldots, s_{n+1}$ *on* w *such that the state* s_{n+1} *is an accepting state. In this case we say that the run* $s_0, s_1, s_2, \ldots, s_{n+1}$ *is an* **accepting run**.

Thus, the automaton $\mathcal{A}$ **does not accept** w if *all* runs of $\mathcal{A}$ are not accepting runs. Informally, $\mathcal{A}$ accepts w if in the transition diagram of $\mathcal{A}$ there exists a path from an initial state to a final state such that the path is labeled with the string w. The acceptance of a given string w by the automaton $\mathcal{A}$ is equivalent to saying that at least of the runs of $\mathcal{A}$ (among possibly many) on w must be an accepting run.

Definition 4. *Let* $\mathcal{M} = (SI, T, F, \Sigma)$ *be an NFA. The* **language accepted by** $\mathcal{M}$*, denoted by* $L(\mathcal{M})$*, is the following language:*

$$\{w \mid \text{ the automaton } \mathcal{M} \text{ accepts } w\}.$$

A language $L \subseteq \Sigma^\star$ *is* **NFA recognizable** *if there exists an NFA* $\mathcal{M}$ *such that* $L = L(\mathcal{M})$.

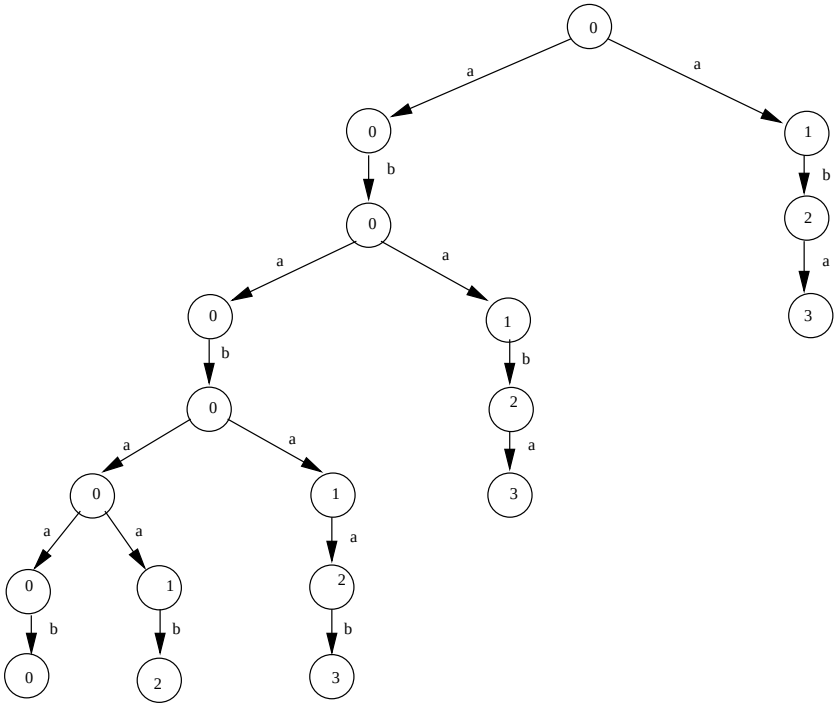

Fig. 3. A tree representation of a computation

Now we give several simple examples:

1. Consider the NFA with exactly one state with no transitions. If the state is accepting then the automaton accepts the language $\{\lambda\}$. Otherwise, the automaton accepts the language $\emptyset$.
2. Consider the automaton in Figure 1. This automaton accepts all the strings w of the alphabet $\{a, b\}$ such that w has an a at the third position from the right.
3. Consider the automaton in Figure 2. This automaton accepts all the strings w of the alphabet $\{a, b\}$ such that w has a substring aub where u is a string of length 3.
4. Every DFA is an NFA. Therefore every DFA recognizable language is NFA recognizable.

2 The subset construction

It is clear that every DFA recognizable language is NFA recognizable. A natural question is whether every NFA recognizable language is DFA recognizable. In this section we give a positive answer to this question. Our interest is thus in the following problem. Design an algorithm that, given an NFA $\mathcal{A}$, produces a DFA $\mathcal{B}$ such that $L(\mathcal{A}) = L(\mathcal{B})$.

Let $\mathcal{A} = (S, I, T, F, \Sigma)$ be an NFA. We want to construct a DFA $\mathcal{B}$ such that $L(\mathcal{A}) = L(\mathcal{B})$. Let's use our method of pretending to be a machine and try to build a DFA recognizing the

language of $\mathcal{A}$. Let $w = \sigma_1\sigma_2 \ldots \sigma_n$ be an input for the NFA $\mathcal{A}$. We proceed as follows. Initially we mark *every* initial states of $\mathcal{A}$. Thus, our current memory is the states that are marked. We read the first symbol σ_1 of w. This activates *every marked* state s as follows.

1. Make s unmarked.
2. Mark all states q if there is a transition from s to q labeled with σ_1.

It may be the case that s can be marked again (because there could be a transition from s to itself labeled with σ_1). The above process marks several states of the automaton. Our current memory is thus all the states that are now marked. We repeat the same process to the next symbol σ_2, then to σ_3, and so on. Thus, all we need to remember in order to process a given input σ is the states that are now.

The above reasoning can be made more formal as follows. Let $w = \sigma_1 \ldots \sigma_n$ be a string and $\mathcal{A} = (S, I, T, F)$ be an NFA. We define the sequence $Q_0, Q_1, \ldots, Q_{n+1}$ of subsets of S inductively as follows:

$$Q_0 = I,\ Q_1 = \bigcup_{p \in Q_0} T(p, \sigma_1), \ldots,\ Q_{i+1} = \bigcup_{p \in Q_i} T(p, \sigma_i), \ldots,\ Q_{n+1} = \bigcup_{p \in Q_n} T(p, \sigma_n).$$

Each Q_i is a subset of S. Now we reason as follows.

Assume that w is accepted by $\mathcal{A}$. Then there must exists an accepting run $q_0, q_1, \ldots, q_{n+1}$ of $\mathcal{A}$ on w. Note that each q_i belongs to Q_i. Therefore Q_{n+1} contains the accepting state q_{n+1}.

Consider the sequence $Q_0, Q_1, Q_2, \ldots, Q_n, Q_{n+1}$. Note the following. For every state $q \in Q_{i+1}$ there exists a state $p \in Q_i$ such that $q \in T(p, \sigma_i)$, for all $i = 1, \ldots, n$. Therefore for every state $q_{n+1} \in Q_n$ we can produce a sequence $q_0, \ldots, q_{n+1}$ such that $q_{i+1} \in T(q_i, \sigma_i)$, for all $i = 1, \ldots, n$. Hence, if Q_{n+1} contains an accepting state then $\mathcal{A}$ must have an accepting run on w.

The discussions above suggests the following construction of a machine $\mathcal{B} = (S', q_0', T', F')$:

1. The set of states S' of $\mathcal{B}$ is $P(S)$.
2. The initial state q_0' of $\mathcal{B}$ is I.
3. The transition function T' is defined as follows. For any state $Q \in S'$ (that is a subset of S) and any symbol $\sigma \in \Sigma$,
$$T'(Q, \sigma) = \bigcup_{p \in Q} T(p, \sigma).$$
4. The set F' contains all states $Q \in S'$ such that Q' has a final state of $\mathcal{A}$.

The machine $\mathcal{B}$ is clearly a DFA. We have already shown that $L(\mathcal{A}) = L(\mathcal{B})$.

The construction above is known as the **subset construction** that converts NFA to DFA.

As an application of this construction consider the language L_n=$\{uav \mid u \in \{a, b\}^\star$ and the length of v is $n\}$ discussed at the beginning of this lecture.There exists an NFA with $n+1$ states that recognizes L. By the theorem above L is DFA recognizable.

Comment: Since NFA and DFA recognizability are equivalent notions, we often use the term finite automata (FA) recognizable to mean either of these notions.

3 NFA with silent moves

An NFA with silent moves is an NFA equipped with λ-transitions. Recall that λ represents the empty string. The meaning of λ-transitions is the following. Assume that the machine is in state s and decides to move along a transition from state s to a state q labeled with λ. The machine, instead of reading the next symbol σ, moves to q and thus changes its state from s to q. This move of the machine is called a *silent move or transition.* In state q the machine can either make a σ-transition or again repeat its silent move (if there is a silent transition from q). In the first case, the machine acts as an NFA, that is it changes its state and reads the symbol and moves to the next symbol of the input string. In the second case, it makes a silent move by changing its state only. Informally, silent moves are internal transitions of the machine. With these transitions the machine does not process the input signal but rather prepares itself to process it. Formally, an NFA with silent moves is defined as follows:

Definition 5. *Let Σ be an alphabet. Denote by $\Sigma_\lambda = \Sigma \cup \{\lambda\}$. An* **NFA with silent moves** *is a tuple $(S, I, T, F, \Sigma_\lambda)$ such that:*

1. *S is a finite set of states of $\mathcal{A}$.*
2. *$I \subset S$ is the set of initial states.*
3. *T is a transition function $S \times \Sigma_\lambda \to P(S)$.*
4. *$F \subset S$ is the set of accepting states.*

On input $w = \sigma_1 \dots \sigma_n \in \Sigma^\star$ the automaton $\mathcal{A}$ starts its *run* as follows. It nondeterministically selects an initial state, call it q_1. At state q_1, the automaton may have one of the following two possibilities:

1. Select nondeterministically a transition from q_1 labeled with σ_1.
2. Select nondeterministically a silent transition without reading the symbol σ_1.

In the first case, the automaton is ready to start processing the next input symbol σ_2. In the second case, the automaton still needs to process the first symbol σ_1. This continues on. Assume that the automaton has arrived to state s and has processed $\sigma_1 \dots \sigma_{i-1}$ part of the input string. At state s, the automaton has two possibilities as above:

1. Select nondeterministically a transition from s labeled with σ_i.
2. Select nondeterministically a silent transition without reading the symbol σ_i.

Again, in the first case, $\mathcal{A}$ is now ready to processing the next input symbol σ_{i+1}. In the second case, the automaton still needs to process the first symbol σ_i. Once the entire string w is processed, the automaton can still continue on running using its silent transitions. We say that the run is accepting if the automaton can enter an accepting state after processing the entire string w. In this case, we say that $\mathcal{A}$ **accepts** the string w.

Our goal is to show that every nondeterministic automaton $\mathcal{A} = (S, I, T, F)$ with silent moves can be simulated by an NFA (without silent moves). Let s be a state of $\mathcal{A} = (S, I, T, F)$. Define $SilentMoves(s)$ recursively as follows:

Base case. $SilentMoves_0(s) = \{s\}$.

Inductive Step $n+1$. Assume we have defined $SilentMoves_n(s)$. Set $SilentMoves_{n+1}(s) = SilentMoves_n(s) \cup \{q \mid \text{there is a silent transition from some } p \in SilentMoves_n(s) \text{ to } q\}$.

Define

$$SilentMoves(s) = SilentMoves_0(s) \cup SilentMoves_1(s) \cup SilentMoves_2(s) \cup \dots.$$

Thus, $SilentMoves(s)$ is the collection of all states that are reachable from s using the λ-transitions. If the automaton is in state s and the next symbol is σ, then the automaton, by moving along the λ-transitions, can select any of the states q in $SilentMove(s)$ and move along a transition labeled by σ. Therefore, we can modify the transition function T of $\mathcal{A}$ as follows:

$$T'(s,\sigma) = \bigcup_{q \in SilentMoves(s)} T(q,\sigma).$$

We now construct the following automaton $\mathcal{B} = (S', I', T', F')$:

1. $S' = S$.
2. $I' = I$.
3. The transition function T' is defined above.
4. $F' = \{q \mid SilentMoves(q) \cap F \neq \emptyset\}$.

Clearly $\mathcal{B}$ is an NFA that does not have λ-transitions. We want to show that the automaton $\mathcal{A}$ with silent moves and the automaton $\mathcal{B}$ are equivalent.

Assume that $w = \sigma_1 \dots \sigma_n$ is accepted by the NFA $\mathcal{B}$. We want to show that the automaton $\mathcal{A}$ accepts w. Take an accepting run $s_1, \dots, s_{n+1}$ of $\mathcal{B}$ on w. Then, by the definition of I', s_1 is an initial state of $\mathcal{A}$. By the definition of T', the automaton $\mathcal{A}$ makes 0 or more silent transitions to a state s_1' at which $\mathcal{A}$ selects a σ_1-transition to state s_2. Again, using the definition of T', the automaton $\mathcal{A}$ makes 0 or more silent transitions to state s_2' at which $\mathcal{A}$ selects a σ_2-transition to state s_3. This continues on. Thus, we can build a run of $\mathcal{A}$:

$$s_1, \dots, s_1', s_2, \dots, s_2', s_3, \dots, s_3', s_4, \dots s_4', \dots, s_{n-1}, \dots, s_{n-1}', s_n.$$

Since s_n is the final state of $\mathcal{B}$, this means $\mathcal{A}$ can make 0 or more silent transitions to reach a final state in $\mathcal{A}$. This shows that $\mathcal{A}$ accepts the string w.

Assume now that $w = \sigma_1 \dots \sigma_n$ is accepted by the NFA $\mathcal{A}$. Take an accepting run

$$(\star) \qquad s_1, \dots, s_1,' s_2, \dots, s_2', s_3, \dots, s_3', s_4, \dots s_4', \dots, s_{n-1}, \dots, s_{n-1}', s_n, \dots, s_n'.$$

of $\mathcal{A}$ on w. For this run we have the following properties:

1. s_1 is an initial state of $\mathcal{A}$.
2. For each $i = 1, \dots, n$, $s_{i+1} \in T(\sigma_i, s_i')$.
3. For each $i = 1, \dots, n, n+1$, the part $s_i, \dots, s_i'$ of the run in $(\star)$ is due to silent moves of the automaton $\mathcal{A}$.

4. $s'_n \in F$.

From the properties above and the construction of $\mathcal{B}$ we have the following. The state s_1 is an initial state of $\mathcal{B}$. For each i, $s_{i+1} \in T'(s_i, \sigma_i)$. The state $s'_n \in F'$. Thus, $\mathcal{B}$ has an accepting run on w. Thus, we have shown that NFA with silent moves are equivalent to NFA without silent moves.

Gathering the results of this and the previous section, for a given language L, the following are all equivalent:

1. There exists a DFA recognizing L.
2. There exists an NFA recognizing L.
3. There exists an NFA with silent moves recognizing L.

4 Algorithmic problems for finite automata

In this section we would like to present algorithms for solving several natural problems about finite automata. We have already dealt with several algorithms manipulating automata. In the previous lecture we provided the minimization algorithm. In the two sections above we dealt with construction of automata such as the subset construction, and the construction that transforms NFA with silent moves to NFA without silent moves. In this section we describe some other interesting problems about automata and their solutions. We recall that we have constructions for the intersection and union of FA recognizable languages from the previous lecture. Also, recall that a state s in an automaton is *non-deterministic* if there are more than 1 transitions from s labeled with the same input symbol.

The emptiness problem. This problem asks to design an algorithm that, given a finite automaton $\mathcal{A}$, decides whether $\mathcal{A}$ accepts a string. Note that $\mathcal{A}$ is not restricted to be deterministic. This problem is interesting because many problems in verification of correctness of programs and program design can be reduced to the emptiness problem for finite automata.

The problem is easy to solve by noticing the following. Let $\mathcal{A} = (S, I, T, F)$ be a finite automaton. If $\mathcal{A}$ accepts a string w then $\mathcal{A}$ has an accepting run s_1, s_2, ..., s_n on w. The accepting run is simply a path from the initial state s_1 to the final state s_n. Conversely, suppose that there exists a path from some initial state to some final state:

$$s_1, s_2, \ldots, s_{n+1}$$

Let σ_1, ..., σ_n be the labels of the edges (s_1, s_2), ..., (s_n, s_{n+1}). Then $\sigma_1 \ldots \sigma_n$ is a string accepted by $\mathcal{A}$. Thus, $\mathcal{A}$ accepts a string if and only if there exists a path from some initial state in I to a final state in F. Here is now a description of the algorithm that solves the emptiness problem.

Remove the labels from the edges of the transition diagram of the automaton $\mathcal{A}$. This produces a directed graph G. For states $s \in I$ and $t \in F$ check if there is a path from s to t (This can be done using the $PathExistence(G, s, t)$ algorithm from Lecture 3). If for some $s \in I$ and

$t \in F$ there is a path from s to t then $\mathcal{A}$ accepts a string, and otherwise $\mathcal{A}$ accepts no string. This solves the emptiness problem for automata.

The equivalence problem. This problem asks to design an algorithm that, given finite automata $\mathcal{A}$ and $\mathcal{B}$, decides whether these automata are equivalent. If we think of $\mathcal{A}$ and $\mathcal{B}$ as finite state description of two programs then this problem asks whether these two programs simulate each other.

There are several algorithms to solve this problem. Here we describe one of the algorithms called the $Equvalence(\mathcal{A}, \mathcal{B})$ algorithm:

1. If $\mathcal{A}$ has nondeterministic states then apply the subset construction to $\mathcal{A}$. If $\mathcal{B}$ has nondeterministic states then apply the subset construction to $\mathcal{B}$. The subset construction produces deterministic finite automata.
   Comment: From now on we assume that $\mathcal{A}$ and $\mathcal{B}$ are deterministic.
2. Construct two deterministic finite automata $\mathcal{A}^{(c)}$ and $\mathcal{B}^{(c)}$ that recognize the complement of $L(\mathcal{A})$ and $L(\mathcal{B})$ respectively.
3. Construct deterministic finite automata $\mathcal{A}_1$ and $\mathcal{B}_1$ that recognize the languages $L(A) \cap L(\mathcal{B}^{(c)})$ and $L(\mathcal{B}) \cap L(\mathcal{A}^{(c)})$, respectively.
   Comment: $\mathcal{A}_1$ recognizes the language $L(\mathcal{A}) \setminus L(\mathcal{B})$ and $\mathcal{B}_1$ recognizes $L(\mathcal{B}) \setminus L(\mathcal{A})$.
4. If $L(\mathcal{A}_1) \neq \emptyset$ then output *"$\mathcal{A}$ and $\mathcal{B}$ are not equivalent"*.
5. If $L(\mathcal{B}_1) \neq \emptyset$ then output *"$\mathcal{A}$ and $\mathcal{B}$ are not equivalent"*. Otherwise, output *"$\mathcal{A}$ and $\mathcal{B}$ are equivalent"*.

This algorithm is correct because $\mathcal{A}$ and $\mathcal{B}$ are equivalent if and only if $L(\mathcal{A}) \setminus L(\mathcal{B}) = \emptyset$ and $L(\mathcal{B}) \setminus L(\mathcal{A}) = \emptyset$. This is exactly what the algorithm above detects.

One comment about this algorithm is the following. The first step of the algorithm uses the subset construction. If an NFA has n states then the subset construction produces a DFA with 2^n states. Therefore it takes an exponential amount of time with respect to the numbers of states of the input automata $\mathcal{A}$ and $\mathcal{B}$ to execute the first step of the algorithm. If $\mathcal{A}$ and $\mathcal{B}$ are deterministic then the algorithm is quite efficient because there is no need to run the subset construction.

The universality problem. This problem asks to design an algorithm that, given a finite automaton $\mathcal{A}$, decides whether the automaton $\mathcal{A}$ recognizes $\Sigma^\star$.

As the previous problems, this problem has a straightforward solution described as follows. If $\mathcal{A}$ has nondeterministic states then we apply the subset of construction to $\mathcal{A}$. So, we can assume that $\mathcal{A}$ is a deterministic finite automaton. We construct the automaton $\mathcal{A}^{(c)}$ that recognizes the complement of the language $L(\mathcal{A})$. Finally, we check if $L(\mathcal{A}^{(c)}) = \emptyset$. If $L(\mathcal{A}^{(c)}) = \emptyset$ then $\mathcal{A}$ recognizes $\Sigma^\star$, and otherwise not.

As in the solution of the equivalence problem, our solution here involves the subset construction that slows down the efficiency of the algorithm significantly for input NFA with many of states.

The infinity problem This problem asks to design an algorithm that, given a finite automaton $\mathcal{A}$, decides whether the automaton accepts infinitely many strings.

In order to solve this problem we need to have a finer analysis of the transition diagram of the automaton. We solve this problem in the next section after explaining a folklore result known as the pumping lemma.

5 The pumping lemma

The essence of the pumping lemma is that any sufficiently long run of an automaton must have repeated states during the computation. The reason for this is that the automaton has a fixed amount of memory, that is a finite number of states. Hence while making a long run, the automaton shall exhaust its memory capacity and has to reuse one of its states that has already been used. Therefore the segment of the computation between any two repeated states can be performed any number of times without effecting the outcome of the run.

Lemma 1 (The Pumping Lemma). *Let $\mathcal{A} = (S, I, T, F, \Sigma)$ be an NFA with exactly n states. Let $w = \sigma_1 \dots \sigma_m$ be a string from $\Sigma^\star$ such that $m \geq n$. If $\mathcal{A}$ accepts the string w then there exists a nonempty substring $v = \sigma_i \dots \sigma_{i+k}$ of the string w such that for all $t \geq 0$, the automaton $\mathcal{A}$ accepts the string*

$$\sigma_1 \dots \sigma_{i-1} v^t \sigma_{i+k+1} \dots \sigma_m.$$

Here is why the lemma is true. Since $\mathcal{A}$ accepts w, there exists a run

$$s_1,\ s_2,\ \dots,\ s_m,\ s_{m+1}$$

of the automaton on the input $w = \sigma_1 \dots \sigma_m$ such that $s_{m+1} \in F$. The key point here is that at least two states in the sequence

$$s_1,\ s_2,\ \dots,\ s_m,\ s_{m+1}$$

must be the same since n is the number of states and $m + 1 > n$. Therefore there must exist a state in this run that is repeated. Hence there exist i and $i + k$ both less than or equal to $m + 1$ such that $s_i = s_{i+k}$ and $k \neq 0$. Let $\bar{s}$ be the sequence of states:

$$s_{i+1},\ \dots,\ s_{i+k}.$$

Recall that $\bar{s}^t$ denotes the concatenation of the sequence $\bar{s}$ to itself t times. We now can conclude that for any integer $t \geq 0$ the sequence

$$s_1,\ \dots,\ s_{i-1},\ s_i,\ \bar{s}^t,\ s_{i+k+1},\ \dots,\ s_{m+1}$$

is an accepting run of the automaton $\mathcal{A}$ on the input

$$\sigma_1 \dots \sigma_{i-1} v^t \sigma_{i+k+1} \dots \sigma_m,$$

where v is $\sigma_i \dots \sigma_{i+k}$. This proves the lemma.

The Pumping Lemma and its proof can be applied to obtain some interesting results. As an example, we now provide an algorithm that solves the infinity problem.

To design an algorithm that detects whether an automaton $\mathcal{A}$ accepts infinitely many strings we observe the following. We first assume that $\mathcal{A}$ does not have silent transitions (by applying the construction from the previous section that removes all the λ-transitions). Now suppose that $L(\mathcal{A})$ is an infinite language. Then there exists a string w accepted by $\mathcal{A}$ such the length of w is greater than the number of states in $\mathcal{A}$. From the proof of the Pumping lemma, we see that there exists a run $s_1,\ s_2,\ \ldots,\ s_m,\ s_{m+1}$ of $\mathcal{A}$ on w with the following properties:

1. s_1 is the initial state and s_{m+1} is a final state.
2. There exist i and $i+k$ with $k \neq 0$ such that $s_i, \ldots, s_{i+k}$ is a path and $s_i = s_{i+k}$. We call the the sequence $s_i, \ldots, s_{i+k}$ a **loop** and s_i a **looping state**.

Thus, if $L(\mathcal{A})$ is infinite then the automaton $\mathcal{A}$ has the following properties. There exists a path from an initial state to a looping state and from the looping state to a final state.

Conversely, assume that in the transition diagram of the NFA $\mathcal{A}$ there exists a path from an initial state to a looping state, and from the looping state to a final state. Then it is easy to see that $L(\mathcal{A})$ accepts infinitely many strings. Indeed, we take a string u_1 that labels the path from the initial state to the looping state, say q. Since q is a looping there must exist a nonempty string string v that labels a path from q to itself. Now take a string u_2 that labels a path from the looping state to the final state. Thus, we have that the strings $u_1 v^t u_2$ are all accepted by $\mathcal{A}$, where $t \geq 0$. Therefore $L(\mathcal{A})$ is an infinite language. Here is now the $Infinity(\mathcal{A})$ algorithm that solves the infinity problem.

1. Transform $\mathcal{A}$ into an NFA with no silent transitions.
2. Construct the set $X = \{q \mid q \text{ is a looping state }\}$.
   Comment: This set can be constructed using the $PathExistsence(G, s, t)$ algorithm.
3. Check if there exists a path from an initial s state to a state $q \in X$ and from q to a final state. If there is such a path then output *"$L(\mathcal{A})$ is infinite"*. Otherwise, output *"$L(\mathcal{A})$ is finite"*.

The pumping lemma can be applied to prove that certain languages are not FA recognizable. A typical example is an elegant proof that the language

$$L = \{a^n b^n \mid n \in N\}$$

over the alphabet $\{a, b\}$ is not FA recognizable. We already discussed this language in the previous lecture and gave an informal reasoning why L is not FA recognizable. Here is now a formal proof of this fact.

Assume to the contrary that this language L is FA recognizable. Then, of course, there exists a finite automaton $\mathcal{A}$ that recognizes L. Let k be the number of states of the automaton $\mathcal{A}$. Consider the string $u = a^k b^k$. This string satisfies the hypothesis of The Pumping Lemma. Therefore we can split u into three pieces $u = v_1 v v_2$ such that $|v| \geq 1$ and for all $t \geq 0$ the string $v_1 v^t v_2$ is in L. Now there are three cases to be considered.

Case 1. The string v contains a's only. Then, of course, the string v_1vvv_2 has more a's than b's in it. By the definition of L, the string v_1vvv_2 is *not* in L. But by The Pumping Lemma v_1vvv_2 is in L. This is a contradiction.

Case 2. The string v contains b's only. As in *Case 1*, this is again a contradiction.

Case 3. The string v contains both some a's and some b's. Well, then the string v_1vvv_2 must have some b's before a's. Then such a string is not in L by the definition of L. But by The Pumping Lemma the string v_1vvv_2 is in L. Again, we have a contradiction.

All these cases show that $\mathcal{A}$ cannot recognize L. Hence L is not FA recognizable.