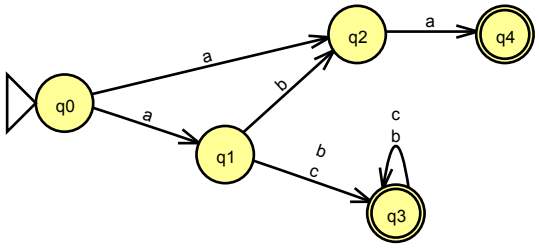
COMP 314 Class 19: Nondeterministic finite automata (NFAs)

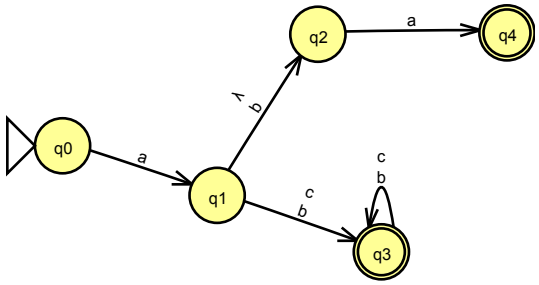
1 Definition of an nfa

An *nfa* (*non-deterministic finite automaton*) is the same as a dfa, except:

- the same label can occur on multiple edges leaving the same vertex. Example 1:



- edges can be labeled with the empty string λ. Example 2:



Note that we are defining nfas informally, via transition graphs. They can be defined more formally, using transition functions, but we won't pursue that approach here.

The execution of an nfa is similar to the nondeterministic programs and nondeterministic Turing machines we saw in the previous lecture. If, while consuming an input string, an nfa finds that it has two possible transitions, it can clone itself and allow each clone to follow a different transition. For example, in the nfa of Example 1 above, if the first symbol in the input string is "`a`", the nfa immediately clones itself, with one clone entering state q_1 and the other entering state q_2. We can imagine the clones continuing to operate in parallel ("real" multitasking) or taking turns ("fake" multitasking), just as with the nondeterministic programs and machines in the previous lecture.

Transitions labeled with the empty string λ immediately trigger a cloning operation: one of the clones stays in the current state, and the other clone follows the λ-transition. For example, in the nfa of Example 2 above, if the first symbol in the input string is "`a`", the nfa transitions to q_1, and then immediately clones itself, with one of the clones following the transition to q_2 and the other remaining at q_1.

2 How does an nfa accept a string?

The definition of acceptance for an nfa is the same as for nondeterministic Turing machines. Specifically, the nfa

- *accepts* if *any* possible path accepts
- *rejects* if *all* possible paths reject

Example 3: Consider the nfa of Example 1 above, with input string "ab". There are three possible paths compatible with this string: $(q_0, q_2, q_{\text{trap}}), (q_0, q_1, q_2), (q_0, q_1, q_3)$. (Note that q_{trap} is a trap state that isn't explicitly shown on the transition graph, but is implicitly the target of any transition that doesn't have a labelled arrow.) The first two of these paths end in rejecting states—q_{trap} and q_2. But the third path ends in the accepting state q_3, and this is sufficient for us to declare that the nfa as a whole accepts the string "ab".

Exercise: Describe the languages accepted by each of the above nfas.

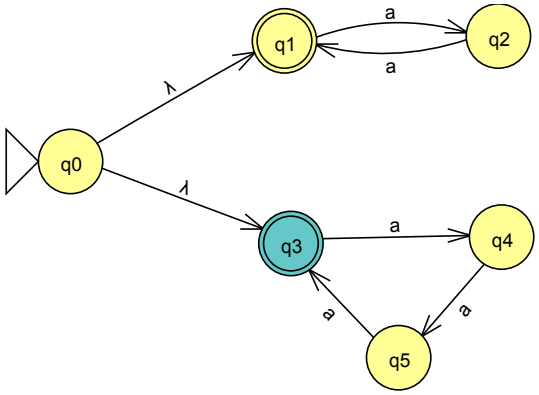
3 Sometimes nfas make things easier

One reason for working with nfas instead of dfas is that they can be simpler, easier, and more natural. Sometimes, a language is most easily described with a nondeterministic decision.

Eaxmple 4: Consider the language

$$L = \{\mathtt{a}^n : n \text{ is a multiple of 2 or 3}\}.$$

Here's an nfa that accepts L:

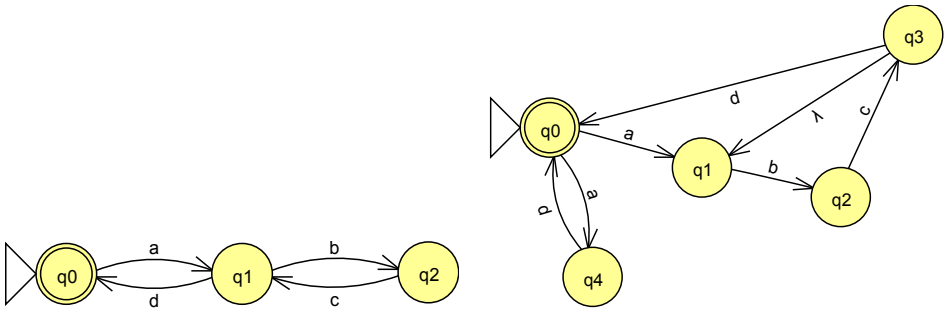


Notice how this nfa decomposes naturally into two possibilities: multiples of 2, and multiples of 3. The nfa can choose nondeterministically between the two possibilities, processing both simultaneously. We will see below that we can also come up with a *deterministic* finite automaton that accepts L, but the deterministic version is more difficult to understand.

4 Equivalence of finite automata

Two or more nfas and/or dfas are *equivalent* if they accept the same language.

Example 5: The two automata below both accept the language `(a(bc)*d)*`, and are therefore equivalent.

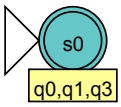


Claim: Every nfa is equivalent to some dfa.

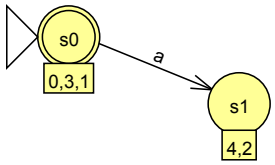
Proof of the claim: This proof is a persuasive sketch, not a formal mathematical argument. Given an nfa, we construct a dfa whose states consist of *subsets* of the nfa states. Transitions between these dfa states are computed by examining all possible transitions in the original nfa. Initial and final states are computed similarly. The examples below show how this is done in practice. The key point is that the original nfa has a finite number of states (say, N), so the constructed dfa also has a finite number of states (at most, the number of subsets of N elements, which is 2^N). The number of transitions in the nfa is also finite, so computing any transition in the constructed dfa is just a matter of checking finitely many transitions in the original nfa. ■

Example 6: Let's convert the nfa of Example 4 above into a dfa.

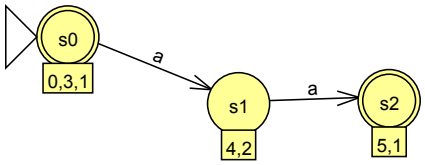
Step 1: The initial state of the nfa is q_0, but because of the λ-transitions out of q_0, the nfa can clone itself and start in any one of the three states $\{q_0, q_1, q_3\}$ before reading any characters of input. Therefore, we create an initial state for our dfa, call it s_0, and label it with "0,1,3" to remind us that it represents the subset $\{q_0, q_1, q_3\}$:



Step 2: Given that the nfa is in one of the states $\{q_0, q_1, q_3\}$, which states could transition to after reading an `a`? From q_1, it can go to q_2; from q_3, it can go to q_4; and these are the only possibilities. So we create a new state for our dfa, call it s_1, and label it with "2,4" to remind us that it represents the subset $\{q_2, q_4\}$:

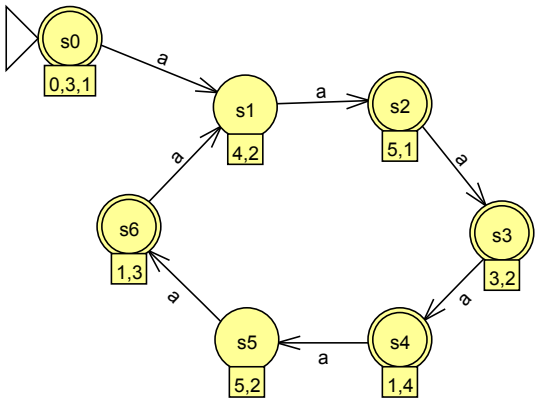


Step 3: Consuming another `a` of input moves us to either q_5 or back to q_1, giving:

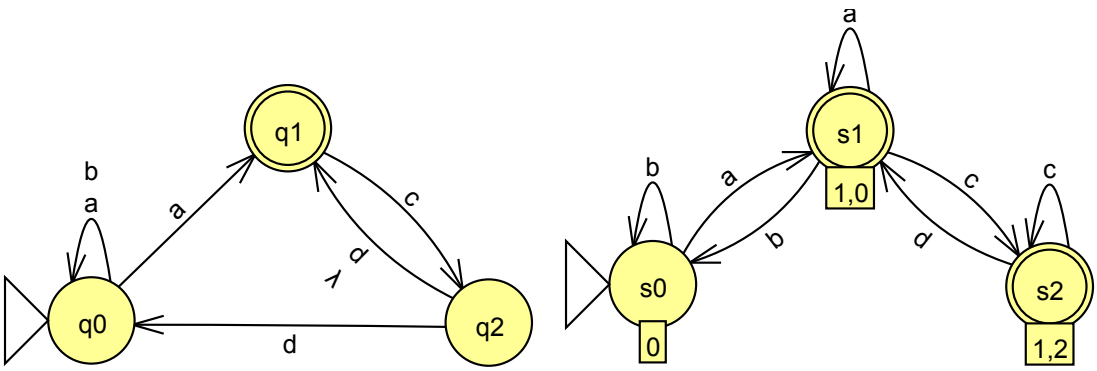


However, there is some additional analysis to be done here. Note that in the original nfa, q_1 is an accepting state, but q_5 is not. So, should our new s_2, representing the subset $\{q_1, q_5\}$, be an accepting state or not? The answer lies in the definition of acceptance for nfas: an nfa accepts a string if *any* of the possible paths for that string end in an accepting state. Therefore, we declare a newly-created dfa state to be accepting if any of the representatives in its corresponding subset are accepting. So in this particular case, s_2 is an accepting state.

The remaining steps continue in similar fashion, finally resulting in:



Example 7: On the left below is another nfa. An equivalent dfa, generated by the algorithm described above, is shown on the right.



4.1 Practicality of converted nfas

When we convert an nfa into an equivalent dfa, the result is often a compact, practical dfa that could be used in real computer programs. In principle, however, an nfa with N states can result in an equivalent dfa of up to 2^N states. For large values of N, this kind of exponentially large dfa is not practical. Therefore, just as with Turing machines, it is clear that adding nondeterminism to dfas does not change computability, but it does have an effect on practicality.

It's worth noting that almost everywhere else in this book, we are equating "practicality" with "reasonable running time". In the case of dfas and nfas, however, the running time is always $O(n)$—the automaton reads each character of the input exactly once, then halts. So the running time of a dfa is always "reasonable". In the previous paragraph, therefore, we instead equated "practicality" with "reasonable size".

5 Why study nondeterminism?

Why do we study nondeterminism as a theoretical concept? Because any computer system (even a large data center) can run only a fixed, maximum number of threads simultaneously, it might appear foolish to study nondeterministic computations, where there is no maximum on the number of simultaneous threads. Despite this, there are several good reasons for studying nondeterminism:

- As shown above (Example 4 with $L = \{\mathtt{a}^n : n \text{ is a multiple of 2 or 3}\}$), it is sometimes more natural to analyze a problem from a nondeterministic point of view.

- Is sometimes easier to prove results using nondeterministic Turing machines and automatons. For example, now that we know nfas and dfas recognize the same set of languages (i.e. regular languages), it is trivial to prove that whenever L is regular, so is L^*. To do this, take the transition graph of a dfa that accepts L, and add a λ-transition from every accepting state back to the initial state. Also, convert the

initial state into an accepting state (if it isn't already). The result is an nfa that accepts L^*.

- To a certain extent, nondeterminism is a reasonable model of how modern computers work. For example, when you submit a query to a search engine such as Google, your query may be simultaneously processed by hundreds of computers—this is a vivid and realistic example of nondeterministic computation in action. On the other hand, any given computer system (even the entire collection of computers controlled by Google), can handle only a finite number of simultaneous threads. Therefore, the theoretical definition of nondeterministic computation, which allows us to create as many threads as desired, is not completely realistic.

- The theory of nondeterministic computation has led to numerous results in complexity theory that have practical implications. We will be studying some of these later in the course. But as a preview, consider the fact that large numbers can be factorized efficiently using a nondeterministic Turing machine, but no one knows how to perform factorization efficiently on a deterministic machine. Many modern cryptography systems depend on the (unproven) assumption that factorization cannot be done efficiently in practice. Therefore, theoretical results that shed light on the distinction between deterministic and nondeterministic computation can have implications for practical applications such as whether or not our cryptography is secure.

6 Summary of nondeterminism

Here are the main facts you should know about nondeterminism:

- Given any nondeterministic Turing machine M, there exists a deterministic Turing machine M' that computes the same function.

- It is widely believed that for certain nondeterministic machines M, the equivalent deterministic machine M' will, in general, take exponentially longer to compute the function. (For example, it is widely believed that factoring requires exponential time on a deterministic machine; but factoring requires only polynomial time on a nondeterministic machine.) However, there is no proof that this exponential blowup must exist.

- Given any nondeterministic finite automaton A, there exists a deterministic finite automaton A' that accepts the same language.

- The number of states in A' can be exponentially larger than the number of states in A.

- There are other models of computation (not studied in this course) where nondeterminism *does* affect computability. For example, there is a type of automaton known as a push down automaton (pda). The deterministic and nondeterministic versions of pdas recognize *different* classes of languages.