Finite State Automata

CS 440: Programming Languages and Translators, Spring 2020

A. From Regular Expression searches to Finite State machines

- Take the regular expression `(ab)*$` ($ means end-of-input) and imagine matching it against an input string like `abab$`.
    - You'd probably think in terms of "looking for an `a`", then "looking for a `b`", then back to "looking for an `a`" and so on.
    - If we find a symbol that isn't we're looking for, the match fails, but instead of stopping immediately, we'll handle this by simply gobbling up symbols until we get to the end of the input.
    - At the end of the input, if we were "looking for an `a`" then the input was acceptable; if not, the input should be rejected.
- We can number the different states of the search:
    - 0: Looking for `a`; if we see one, go to state 1; if we see a `b`, go to state 2.
    - 1: Looking for a `b`; if we see one, go to state 0; if we see an `a`, go to state 2.
    - 2: No matter what symbol we see, stay in state 2.
- You'll most likely recognize this as the general structure of a **Finite State Machine** a.k.a. **Finite (State) Automaton**. (Automaton just means machine.)

B. Finite State Machines

- A finite state machine has a *state*; it processes a string of symbols one-by-one with each state / symbol combination leading to another state. The alphabet (set of symbols) and the set of state values are both fixed and finite (hence Finite State Machine) and so is the set of transitions from state and symbol to next state.
- **Recognizer**: The simplest kind of FSM is a recognizer: It only produces 1 bit of output ("Yes" or "No"), after all the input is seen. It **accepts** the input if it says "Yes," and **rejects** the input if it says "No."
- As an example, you can build an FSM that takes a sequence of 0's and 1's and accepts strings that include exactly two occurrences of 1.
- **To specify a recognizer FSM** you specify
    - The **state set** Q (the finite set of all possible states) and alphabet Σ of symbols.
    - The **start state** (the one the FSM begins execution in); some $q_0 \in Q$.
    - The set of **accepting states** (when the FSM ends computation, it accepts the input iff it ends in an accepting state). $Accept \subseteq Q$. The start state can $\in Accept$ but isn't required to.
    - A **transition function** of type $Q \times \Sigma \to Q$ that describes how the FSM executes. It takes the current state and input symbol and produces the next state for the machine (which may or may not be the same as the current state).

- **FSM Execution**:

```
Initialize state ← start state;
while there exists more input {
        Read next symbol of input;
        state ← transition_function(input symbol, state) (i.e., set state to new state)
}
// At end of input
if state ∈ Accepting States then output "Accept" else output "Reject"
```

FSM Example:

- **Machine M_1** reads strings of `a`'s and `b`'s and accepts a string iff it contains exactly two occurrences of `b`. M_1 has four states $Q = \{q_0, q_1, q_2, q_3\}$ where we're in states q_0, q_1, or q_2 if we've seen exactly 0, 1, or 2 `b`'s. We'll be in state q_3 if we've seen three *or more* `b`'s. State q_0 is the initial state, state q_2 is the (only) accepting state. For transitions, regardless of the state, if we see an `a`, we stay in that state. If we see a `b` and we're in state q_k (where k is 0, 1, or 2), then we go to state q_{k+1}. If we're in state q_3 and the input is `b`, we stay in q_3.
- Execution of M_1: One way to trace the execution of an FSM uses two rows: The top row holds symbols of input; the bottom row holds the states that the FSM is in as it reads the input. The leftmost column holds the start state; the rightmost column shows the final state of the execution (we accept iff it is an accepting state) and as we go across the rows, we stagger the columns for visibility.
- Below a run of M_1 with input `abaabaa`. Since we end in state q_2, we accept the input.

	a		b		a		a		b		a		a	
q_0		q_0		q_1		q_1		q_1		q_2		q_2		q_2

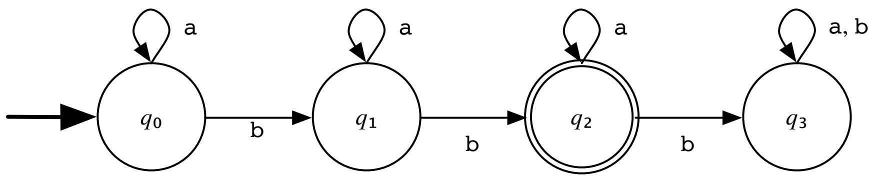
- (Quick question: What strings would we accept if we change the accepting state from q_2 to q_3? What if we make every state accepting *except for* q_2?)

C. Representing Transition Functions

- **State Transition Table:** A state transition table has three columns: The first two columns list a state and input symbol. The third column lists the state we go to if we see the given state and input symbol. The start state and accepting states must be given separately from the table.
- **State Transition Diagram**: A state transition diagram is a labeled directed graph, with a node for each state and an arc for each transition. The transition arc goes from the old state to the new state and is labeled with the input symbol that causes this transition. In a state transition diagram, the start state and accepting states

are shown in the diagram itself: The start state is pointed to by a tail-less arrow; accepting states are drawn with a doubled border.

- **Example 1**: The table and diagram here both describe machine M_1 (which accepts a string of `a`'s and `b`'s iff it has exactly two `b`'s).



Machine M_1

State	*Input*	*New State*
q_0	a	q_0
q_0	b	q_1
q_1	a	q_1
q_1	b	q_2
q_2	a	q_2
q_2	b	q_3
q_3	a	q_3
q_3	b	q_3

Initial state: q_0
Accepting state: q_2

D. *Nondeterministic Finite Automata (NFA)*

- In a **Deterministic Finite State Automaton (DFA)**, the transition table holds exactly one new state for each state/symbol combination: δ: Q × Σ → Q is the transition function, then it's fully defined (every δ(q, x) is defined and leads to exactly one state. The transition diagram has exactly one outgoing labeled arrow for each symbol in the alphabet. (For this purpose, an arrow with two labels is counted as two arrows.)
- In a general **Nondeterministic Finite Automaton (NFA)**, each state/symbol combination leads to a set of next states, so δ(q, x) ⊆ Q, not ∈ Q. If δ(q, x) has exactly 1 state, that's the state we go to.
- **Nondeterministic choice or backtracking execution**: There are a couple of different ways to explain what happens if δ(q, x) has > 1 state; we'll address nondeterministic choice in a bit, but for now the intuition is that we can try them all, one-by-one, backtracking if necessary.
- **Stuck computations**: It's possible for δ(q, x) = ∅. In that case, execution is said to be **stuck**: There's no way to continue execution from state q on input x. We have to backtrack to some previous position and try another possible choice. If we run out of states to choose, then that state/symbol combination is stuck and we have to backtrack from there, and so on.
- **ε-transitions**: There's also one new column in the transition table, labeled ε (the empty string). If q is the current state and a current input symbol x (i.e., we're not at the end of the string), then in addition to going to any state in δ(q, x) (and consuming x), we can take an ε-transition (a.k.a. ε jump) to any state in δ(q, ε) without using x. If we're at the end of the input, we can still do ε-transitions.
  - It's possible to have an **ε-transition cycle**, where every transition is an ε-jump. In that case, execution can cycle through these states without ever terminating. (Hopefully there's some alternative transition (ε- or not-ε-) that breaks the cycle.
- To summarize: In a DFA, δ(q, x) is always uniquely defined and there is no δ(q, ε) defined. In an NFA, δ(q, x) can be any set of states ⊆ Q. If δ(q, x) is empty, computation can't continue and we have to backtrack. If δ(q, x) has > 1 member, then we can choose any state in δ(q, x) as our next state. In addition, defined is δ(q, ε) ⊆ Q, the states we can ε-jump to without consuming input.
- **NFAs and state transition diagrams**: For each state node q and symbol x ∈ Σ, there can be zero, one, or more than one labels of x on an arrow leaving q; the number is the size of δ(q, x). When executing, we can choose any such arrow and follow it in the diagram. In addition, there can be zero or more arrows from q labeled ε, and we're also allowed to follow one of them out of q, without using an input symbol.

- **Example 2**: How can a finite automaton handle the regular expression `(ab)*a` ? Machine M_2 shows one way.
  - (Start) State 0: Either we're looking for the `a` in `ab` or we're looking for the final `a`. If we see an `a`, go to states 1 or 2. If we see a `b`, we're stuck.
  - State 1: We've seen the `a` in `ab` and are looking for the `b`; if we see a `b`, go to state 0. If we see an `a`, we're stuck.
  - State 2: We've seen what might have been the final `a`. There are no transitions: If we see an `a` or `b`, we're stuck. State 2 is the accepting state, so we'll accept exactly when we get to state 2 at the end of the input.

Machine M_2

State	*Input*	*New States*
0	a	1, 2
0	b	∅
1	a	∅
1	b	0
2	a	∅
2	b	∅

Initial state: 0
Accepting state: 2

- **Example 3**: Machine M_3 also handles `(ab)*a` but uses an ε-transition to jump from "looking for the `a` in `ab`" to "looking for the final `a`".
  - (Start) State 0: We're looking for the `a` in `ab`. On ε, go to state 2 to look for the final `a`; on `a`, go to state 1. On `b`, we're stuck.
  - State 1: (As in M_2) We've seen the `a` in `ab` and are looking for the `b`; if we see a `b`, go to state 0. On `a`, we're stuck.
  - State 2: We're looking for the final `a`. On `a`, go to state 3; on `b`, we're stuck.
  - State 3: (Like state 2 in M_2) This is the accepting state; if we see any input, we're stuck.

Machine M_3

State	*Input*	*New States*
0	ε	2
0	a	1
0	b	∅
1	ε	∅
1	a	∅
1	b	0
2	ε	∅
2	a	3
2	b	∅
3	ε	∅
3	a	∅
3	b	∅

Initial state: 0
Accepting state: 3

Understanding how an NFA executes

- There are various ways to think of how an NFA executes; backtracking search is the most intuitive, nondeterministic execution is more unfamiliar.
- In any case, the question is, what do we do if $\delta(q, x)$ has more than one member?

1. Backtracking search:

- Choose a member of $\delta(q, x)$ and try that as the new state. If the automaton reaches end of input in an accepting state, the original input is accepted and we're all done. If the automaton reaches end of input in a non-accepting state or it gets stuck along some execution path, then backtrack to the most recent choice point and an alternative choice. If there is none, then execution along this path is considered to be fail and we have to backtrack even further back.
- Note: If any execution path leads to end of input in an accepting state, we accept. We only reject if all possible execution paths either get stuck or are in a non-accepting state at end of input.

2. Use a set of possible current states.

- Here, we're always in one member of a set of states but don't know which one. Initially, the set is $\{q_0\}$ where q_0 is the start state. More generally, say we're in one of states of R ($\varnothing \neq R \subseteq Q$).

- First, expand R as much as possible by adding in all states in $\delta(q, \varepsilon)$ to R (for all $q \in R$). This process is called "Finding the ε-closure of R" and corresponds to checking for all ε-transitions from states in R.
- For each input symbol x, the transition on x from R can go to any state we can reach from a state in R, on input x. Collect all the states in each set $\delta(q, x)$ (for all $q \in R$). If we're in one of the states of R and follow input x, we'll end up in one of these states.
- When executing, if we reach the end of input and any of the states in R are accepting, then accept the input. Only reject if every state in R rejects. Here, the idea is that if we might be in any state in R, then clearly we want to be an accepting state if we can, and if we end execution in an accepting state, then we should accept. Notice if R contains more than one accepting state, that's okay.

3. Be in multiple states simultaneously.

- This is just like the previous situation except that we think of control being "at" every state in R simultaneously. If we're asked "What state is the automaton in?" then instead of saying "Some state in R but I don't know which one," we say "All the states in R." In practical terms, there's no real difference, but some people prefer to think of things this way.

E. Implementing an R.E. Using an NFA

- It's fairly easy to take a regular expression and encode its search as a transition graph for an NFA. We initialize the NFA to just have one state marked start and accept simultaneously. As we go through the expression, we extend the graph and/or join it with other graphs to form larger and larger graphs.
- For a constant x, build a graph with four nodes: Node 1 is the start state and has an ε-transition to node 2, which has an arc labeled x to node 3, which has an ε-transition to a node 4, our accept state.

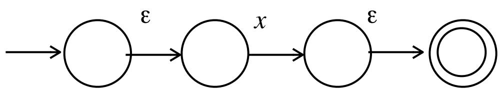


Figure 1: NFA for a Symbol

- For an alternation $R_1 \mid R_2$, recursively generate graphs for R_1 and R_2 and add two more nodes. The accept state of our current graph has an ε-transition to the first new node, and this new node has ε-transitions to the start states of the graphs for R_1 and R_2. The second new node has ε-transitions from the accept states for R_1 and R_2 and it is the accept state of the new graph.

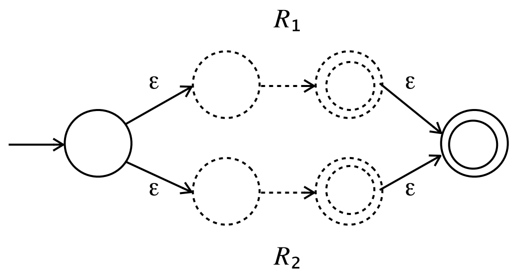


Figure 2: NFA for Alternation

- For a sequence R_1 R_2, we ε-link the accept state of R_1 to the start state for R_2. The start state of R_1 is our start state and the accept state for R_2 is our accept state.

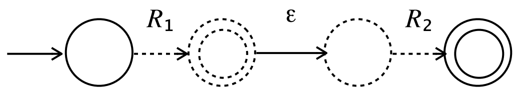


Figure 3: NFA for Concatenation

- For a Kleene * expression R^*, we take the graph for R and add new start and accept nodes and add an ε-transition from the end of R to the beginning of R (this gives us the repeating part of R^*) and also an ε-transition from our start to accept states (this is the zero occurrences of R part of R^*).

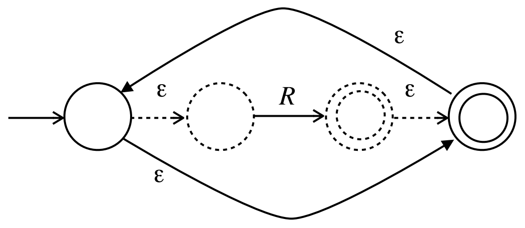


Figure 4: NFA for Kleene Star

- As an example, Figure 5 is a graph for $a\ (b \mid c)^*$ that follows the algorithm in the book. (It contains even more ε-transitions than the transformations from Figures 1 – 4.)
  - There are two sorts of nondeterminism in Figure 5: Arcs labeled by ε and decisions (two arcs from the same node with the same label).
- To make it easier to see what the NFA does, Figure 6 shows the result of eliminating some redundant ε-arrows from Figure 5.
- Not only does there exist an NFA that accepts the same language as a given regular expression, there exists a regular expression for each NFA. (This is not as obvious.) In that sense, NFAs have the same **expressiveness** as regular expressions.

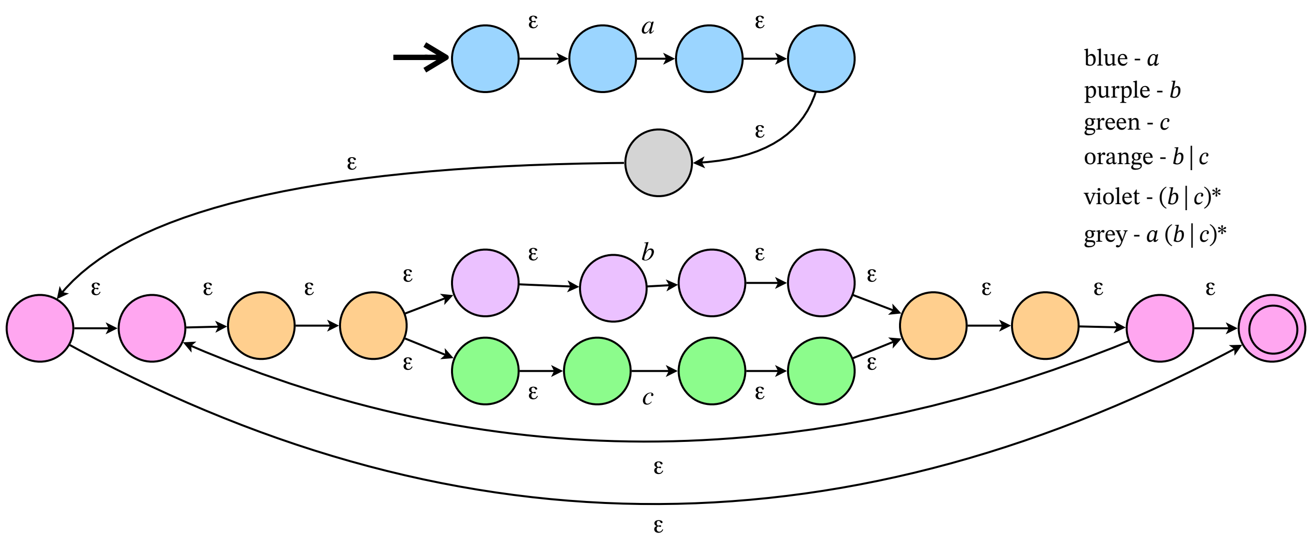


Figure 5: a (b | c) Before Removing ε-Transitions*

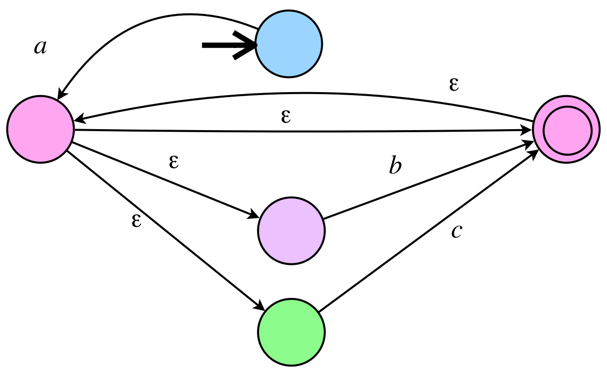


Figure 6: a (b | c) After Removing Some ε-Transitions*

F. Converting from an NFA to an Equivalent DFA

- For every NFA there exists an equivalent DFA; we can build it by tracking all the states the NFA could be in after seeing the current input.
    - Say NFA node q goes to two different states on symbol x. When we use backtracking search, we follow one of those arcs and backtrack to try the other arc if necessary.
    - In the DFA, we'll instead have a single set of states that answers the question "What are all the NFA states I could have gotten to from q with an arc labeled x?"
    - Whenever execution reaches this DFA state we'll be in one of the NFA states but not know which one[*].
- So each DFA state is the set of all current possible NFA states, and to get from one DFA state to another, we have to look at all possible NFA states we can get to from any of the current possible NFA states. This gives us a DFA state = another set of NFA states. If it's a set we've already seen, it's a new state to add to the set of DFA states.
- Note that if there are n states in the NFA then there are 2^n possible DFA states (though not all of them are likely to be in the set of actual needed DFA states).
- ***An Example***: Figure 7 begins with an NFA to transform and goes through the steps of the transformation. The figure shows the results of all the steps; you can follow along as those steps are described below.

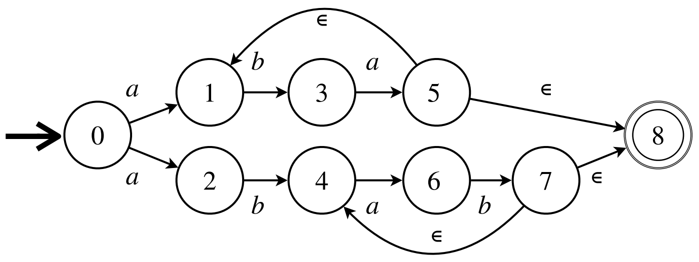
G. Get Rid of Error States

- An error state is any state that has no path that can reach an accepting state. (Once you get to an error state, you're doomed to never accept the input.) Functionally, there's no need for more than one error state, with all transitions from it leading back to it. (If τ is the transition function, then $\tau(\textit{error state}, x) = \textit{error state}$ for all symbols $x \in \Sigma$.)
- An obvious optimization to make is to combine all error states into just one error state. But one can even go further and remove all error states (and transitions to/from them) from the NFA. All transitions in the old machine from some state/symbol combination to an error state turn into undefined transitions on that state/ symbol combination.
- The reason for removing all error states is that since converting from an NFA to a DFA involves looking at sets of NFA states, the fewer the number of starting NFA states, the less exponential blowup occurs.

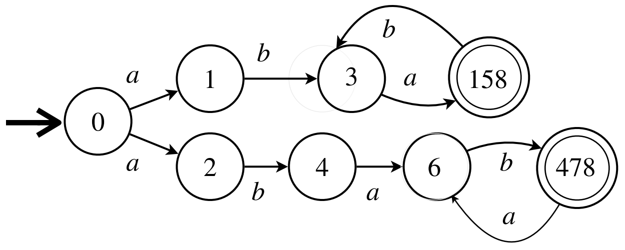
H. Calculate the ε-Closure of a NFA

- "Calculating the ε-closure of an NFA" involves taking an NFA and finding an equivalent NFA that doesn't contain ε-transitions.

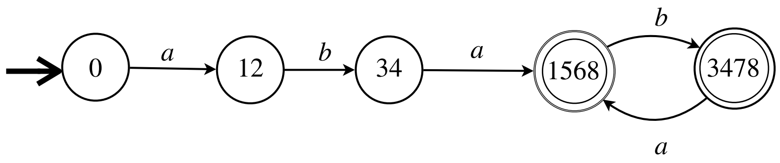
[*] Or if you prefer, we can think of being in all of those states simultaneously by following all possible search paths simultaneously. For example, using threads, whenever we encounter a state with multiple outgoing arcs with the same label, fork off a new thread of control and follow both paths simultaneously.



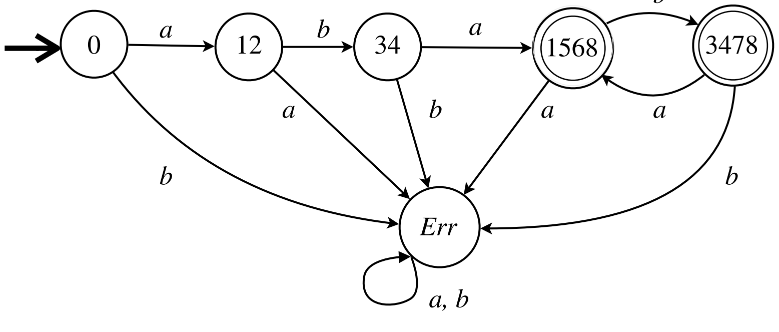
Original NFA



NFA after ∊-closure



Set-of-States DFA



DFA With Error State

Figure 7: A Full Example

- This is useful because an NFA with ε-transitions can be "in" more than one state at a time.
  - If we have a state q_1 connected to q_2 by an ε arc, then we have the choice of staying in q_1 or going to q_2.
  - The choice may be important because we can have arcs from q_1 and q_2 with the same labels but to different target states. Symmetrically, q_1 might not have an arc from it that q_2 does.
- To remove ε-transitions, we can do the conversion by looking at the states reachable by ε-paths from a given state. Here's an algorithm:
- Let NFA$_1$ have states Q_1 and a set of state transitions $\tau_1 \subseteq Q_1 \times (\Sigma \cup \{ \varepsilon \}) \times Q_1$. (Recall that Σ is the alphabet of input characters.) The type of τ_1 can't be $\ldots \rightarrow Q_1$ because τ_1 can map a state/symbol pair to multiple results. You can think of τ_1 as a function if you use the power set of Q_1: τ_1: $Q_1 \times (\Sigma \cup \{ \varepsilon \}) \rightarrow$ $power(Q_1)$
- For Q_2 (the states of our result NFA$_2$; recall that $Q_2 \subseteq power(Q_1)$)
  - Initialize $Q_2 = \varnothing$
  - For each $q_1 \in Q_1$
    - Let $T = \{q_1\} \cup \{q_2 \in Q_1 \mid q_2$ is reachable from q_1 along a path of ε-transitions$\}$
      - (T is the set of ε-path targets from q_1)
    - If $T \notin Q_2$, add it to Q_2.
  - Find all states in NFA$_2$ that contain an initial state in NFA$_1$ and mark them as initial.
  - Find all states in NFA$_2$ that contain an accepting state in NFA$_1$ and mark them as accepting.
- Now to define τ_2 (the transitions for NFA$_2$) where $\tau_2 : power(Q_1) \times \Sigma \rightarrow power(Q_1)$.
- For each $R \in Q_2$ and $x \in \Sigma$, (note $R \subseteq Q_1$)
  - Let $T = \{\tau_1(q_1, x) \in Q_1 \mid q_1 \in R\}$
    - (T = the image of the set R under $\tau_1($ ____, $x)$ treated as a function $Q_1 \rightarrow Q_1$))
  - Add $\tau_2(R, x) = T$ to τ_2.
- It's possible to have one algorithm that combines removing ε-transitions and converting from the NFA-to-DFA, but it's a little messy, so I'm omitting it.

I. Converting From an ε-Transition-Free NFA to an Equivalent DFA

Here's a "set of states" algorithm for the NFA-to-DFA conversion. Let's assume the NFA has no ε-transitions.

- Set the set of DFA states to $\varnothing$.
- Find the set of all initial NFA states, add it to the set of DFA states, mark it as initial and "unprocessed".
- While there exists an unprocessed $Q \in$ the set of DFA states
  - For each symbol $x \in \Sigma$ and each (NFA) state $q \in Q$
    - Let T be the set of all target NFA states from q and x.
    - If T is not already in the set of DFA states, add it and mark it unprocessed.

    - Create a DFA transition from *Q* to *T* labeled *x*.
    - If *T* contains any NFA-accepting states, mark it as accepting in the DFA.
  - Mark *Q* as being processed.

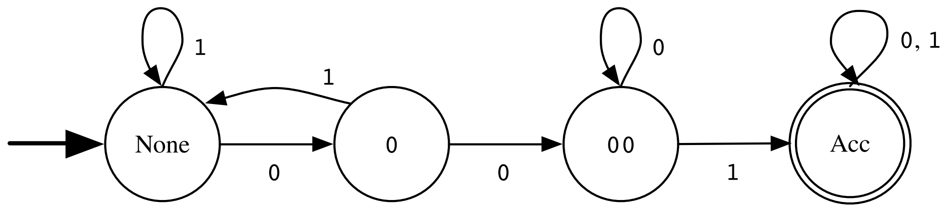
J. Removing Stuck Configurations

- It's possible that the DFA that results from the set-of-states algorithm might be missing possible transitions. (I.e., if τ is the transition function, then $\tau(q, x)$ might be undefined for some q and x.) To get rid of such "stuck" configurations, we can just introduce new state, an error state, and define $\tau(q, x)$ = the error state.
- If there already was an error state, we just reuse that. If there were more than one error state, then a DFA optimization is to combine them all into one error state.

Activity Problems, Lecture 8

Problems

Basics of Finite State Machines



1. Consider the state diagram above for a finite state machine.
    a. What is the start state? The accepting state(s)?
    b. Trace the execution of this machine on the input `0100010`.
    c. What is the pattern of strings accepted by this machine?
    d. Complete the state transition table for this machine.

State	*Input*	*New State*
None	0	0
None	1	None
0		
0		
00		
00		
Acc		
Acc		

2. Design a finite state machine that takes an input string of 0's and 1's and has two states *E* and *O* (for even and odd), which it uses to keep track of whether the input has an even or odd number of 0's. It should accept strings with an even number of 0's.[†]
    a. Draw a state diagram for the machine.
    b. Give a state transition table for the machine.
    c. Give a regular expression for this language.
3. Design a finite state machine that takes an input string of `a`'s and `b`'s and accepts iff every `a` is immediately followed by at least one `b`. E.g., `bbabbab` should be accepted but `aab` and `baab` shouldn't (because of their first `a`'s) and `ababa` shouldn't (because of the final `a`).
    a. Give a state transition diagram for this machine. Hint: You'll need 3 states, one for "ready to see an `a`", one for "just saw an `a`", and one for "reject this string".
    b. Give a state transition table for the machine.
    c. Give a trace of execution for `bbabbab`.
    d. Give a regular expression for the language accepted by this machine.

[†] A side note: It's easy to get a machine or reg expr for the language that has an even number of 1's (just flip the 0 & 1 transitions). If we want a machine that accepts strings that meet both criteria (even number of 0's and even number of 1's), it's straightforward to build a new machine that combines the old machines, but finding a reg expr for the intersection language is definitely more complicated.

NFA to DFA Conversion

1. To practice NFA transformations on small machines,

   a. For the machine M_2 of Example 2, calculate the state transition table you get if we follow the set-of-possible-current states model. You can toss out any states from M_2 that you no longer need.

   b. For the machine M_3 of Example 3, first calculate the ε-closure: Our initial state will be {0, 2} because if we start in state 0, we can ε-jump to state 2, so we can be in either of those two states. Then, follow the set-of-possible-current states model and build the transition table you get from that.

2. Take the NFA described below and find an equivalent DFA.

   a. Identify all the error states and remove them from the NFA and all transitions in the NFA.

   b. Remove ε-transitions.

   c. Convert to a DFA using the set-of-states transformation. (If there are any stuck transitions) add an error state back into the automaton and fill in transitions to the error state from stuck configurations.

Start state = A, Accept state = G

State	ε	`x`	`y`	`z`
A	*B*	*C*	*D, F*	*E, G*
B		*C*	*F*	*G*
C	*H*	*E*	*E*	*G*
D	*E*	*E*		*D*
E		*E*	*D*	
F		*G*	*E*	*B*
G		*H*	*D*	*F*
H	*C*	*D*	*D*	*C*

Solutions to Selected Activity 8 Problems

Basics of Finite State Machines

1. (Basic FSM)

   a. The state state is *None*. The accepting state is *Acc*.

   b.

	0		1		0		0		0		1		1		0	
None		0		None		0		00		00		Acc		Acc		Acc

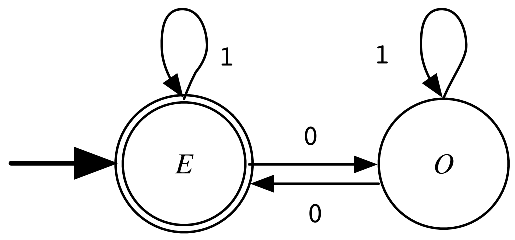
State	*Input*	*New State*
None	0	0
None	1	None
0	0	00
0	1	None
00	0	00
00	1	Acc
Acc	0	Acc
Acc	1	Acc

   c. It accepts any string of 0's and 1's that includes 001 as a substring.

   d. Complete transition table is shown to the right.

2. (Even/odd numbers of 0’s)

   2a. Note that since we’re tracking only the `0`’s, the `1`’s don’t change the state. Each `0` changes the state from even to odd (or vice versa).



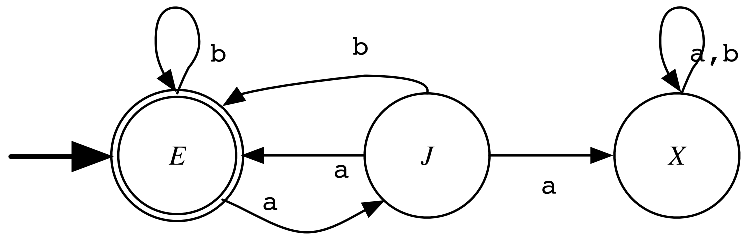
State	*Input*	*New State*
E	0	*O*
E	1	*E*
O	0	*E*
O	1	*O*

   2b. The initial and accepting state is *E*.

   2c. One way to get a regular expression is to ask for all the general ways we can get to the accepting state from the start state. A trip from *E* to *O* and back looks like 0 1* 0; a trip from *E* back to *E* looks like 1. A single trips is one or the other, which gives (1 | 0 1* 0), and we can take as many trips as we like, which gives (1 | 0 1* 0)*. (This is not the only reg expr that's possible.)

3. (Look for each `a` is followed by ≥ one `b`) The three states are *E* = “Ready to see an `a`”, *J* = “Just saw an `a`”, and *X* = “Reject this string”.

   a. (State transition diagram)



   b. State transition table

State	*Input*	*New State*
E	a	*J*
E	b	*E*
J	a	*X*
J	b	*E*
X	a	*X*
X	b	*X*

c. (Execution trace): Omitted.

d. `b*(ab+)*` is a regular expression for this language.

NFA to DFA Conversion

1. (Simple conversions)

1a. (Set-of-states for M_2)

Machine M_2'

State	*Input*	*New States*
0	a	1, 2
0	b	∅
1, 2	a	∅
1, 2	b	0

Initial state: 0
Accepting state: {1, 2}

1b. (ε-closure of M_3)

Machine M_2'

State	*Input*	*New States*
0	a	1, 2
0	b	∅
1, 2	a	∅
1, 2	b	0

Initial state: 0
Accepting state: {1, 2}

2. (Convert NFA with error states)

a. After finding error states *D* and *E* and removing them from the automaton:

Start state = A, Accept state = G

State	ε	x	y	z
A	*B*	*C*	*F*	*G*
B		*C*	*F*	*G*
C	*H*			*G*
F		*G*		*B*
G		*H*		*F*
H	*C*			*C*

b. After removing ε-transitions. **Notation**: *AB* is short for {*A*, *B*}.

Start state = AB, Accept state = G

State	x	y	z
AB	*CH*	*F*	*G*
B	*CH*	*F*	*G*
CH			*CGH*
F	*G*		*B*
G	*CH*		*F*

c. After converting to set-of-states DFA, error state added

Start States = AB, Accept States = BCFGH, CGH, CFGH, G

State	`x`	`y`	`z`
AB	*CH*	*F*	*G*
B	*CH*	*F*	*G*
CH	*err*	*err*	*CGH*
F	*G*	*err*	*B*
G	*CH*	*err*	*F*
CGH	*CH*	*err*	*CFGH*
CFGH	*CGH*	*err*	*BCFGH*
BCFGH	*CGH*	*F*	*BCFGH*
err	*err*	*err*	*err*