CMPSCI 250: Introduction to Computation

Lecture #23: Finishing Kleene's Theorem

David Mix Barrington

25 April 2013

Finishing Kleene's Theorem

- A Normal Form for λ-NFA's
- Building λ-NFA's by Induction
- Why is this Construction Correct?
- A New Model: R.e.-NFA's
- State Elimination
- The Validity of State Elimination
- Three Examples of State Elimination

Review: Parts of Kleene’s Theorem

- Our goal in Kleene’s Theorem is to be able to convert regular expressions to DFA’s and vice versa.
- In the last lecture we’ve provided two of the three pieces of the transformation from regular expressions to DFA’s. We defined ordinary NFA’s and λ-NFA’s, then presented the Subset Construction to turn ordinary NFA’s to DFA’s, then presented the Killing λ-Moves Construction to turn λ-NFA’s to ordinary NFA’s.
- Today we will see how to convert regular expressions to equivalent λ-NFA’s. This will complete the steps needed to go from regular expressions to DFA’s.
- In this lecture we will finish also Kleene’s Theorem by presenting the State Elimination Construction to convert DFA’s (or NFA’s, or λ-NFA’s) to regular expressions.

Review: Induction on Regular Expressions

- We want to prove that for every regular expression R, we can construct a λ-NFA N such that L(N) = L(R).

- The way to prove a proposition P(R) for all regular expressions R is to use induction on the definition of regular expressions. We must prove the two base cases, (1) P(∅) and (2) P(a) for every letter a ∈ Σ. Then we must prove the three inductive cases. If P(R) and P(S) are true, we must (3) prove P(R + S), (4) prove P(RS), and (5) prove P(R*).

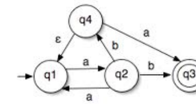
- Here P(R) is "there exists N with L(N) = L(R)". As with our other inductive proofs on regular expressions, we will actually define a recursive algorithm that will take R as input and return N as output. We could code this algorithm in pseudo-Java using class definitions for `RegExp` and `LambdaNFA`, but we will stick with an informal description here.

A Normal Form for λ-NFA's

- Since we want to actually carry out this construction by hand on examples, we're going to make it a little more complicated than it would need to be just to prove that a valid construction exists. We'll produce λ-NFA's in a particular **normal form** -- they will satisfy three rules that will allow us to make simpler λ-NFA's in most cases.
- Rule (1) says that the λ-NFA has exactly one final state, which isn't the start state.
- Rule (2) says that no transitions go into the start state.
- Rule (3) says that no transitions go out of the final state.
- Similar rules will also show up later in the State Elimination Construction.

iClicker Question #1: λ-NFA Normal Form

- Here is a λ-NFA (though it denotes the empty string as ε instead of λ) that is *not* in the normal form we just specified. The four statements below are all true -- **which one is a violation** of the normal form?



- (a) There are two b-moves out of state q2.

- (b) There are no moves out of the final state q3.

- (c) There is exactly one final state which is not the start state.

- (d) There is a move into the start state q1.

The Construction

- (1) For ∅, we need a λ-NFA with a start state and a final state. That's all we need -- if it has no transitions, it accepts no strings and its language is ∅.

- (2) For a, we can again have a start state i and a final state f, with a single transition (i, a, f). The rules are satisfied, and the language is {a} as it should be.

- (3) Now assume, as our IH, that we have constructed λ-NFA's M and M' for our two regular expressions R and R', and that M and N follow the three rules. We need to build a new λ-NFA M'' such that $L(M'') = L(M) \cup L(M') = L(R + R')$. M'' will have copies of all the states of M and M', but we will *merge* the two initial states, and *merge* the two final states.

- (4) To make M''' with $L(M''') = L(M)L(M') = L(RR')$, we instead merge the final state of M with the initial state of M', making the new state non-final.

The Star Case and the Proof

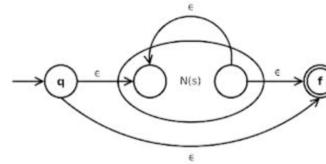
- (5) Finally we want to build a λ-NFA N such that L(N) = L(M)* = L(R*). Assume that M has start state i and final state f. N's states will be M's states plus two more, a new start state s and a new final state t. We then add four new λ-moves: (s, λ, i), (i, λ, f), (f, λ, i), and (f, λ, t). We make f now a non-final state.

- Now we want to prove by induction on all regular expressions that this construction is correct -- if N is the λ-NFA made from R, then L(N) = L(R). This is pretty obvious for the two base cases as we can check the languages of the λ-NFA's directly. So we must check the three inductive cases.

- With the two λ-NFA's connected in **parallel** in step (3), a path from the start to final state of M'' must either pass through only states of M or only states of M'. This is because the first move must be a move of either one machine or the other, and from that point we must stay in that machine until we finish, since we can't return to the start or continue past the finish, due to the rules. The path has *either* read a string in L(M) *or* read a string in $L(M')$.

iClicker Question #2: A Different Star Construction

- Here, again with ε in place of λ, is a λ-NFA M constructed from an existing λ-NFA called N(s). The intent is that if N(s) has the language of some regular expression s, then L(M) = s*. Could we have used this construction?



- (a) No, because M is not in our normal form.
- (b) No, because L(M) does not include all the strings in s*.
- (c) No, because L(M) includes strings not in s*.
- (d) Yes.

Finishing the Correctness Proof

- In step (4) we created M''' by connecting M and M' in **series**, and we must show that $L(M''') = L(M)L(M')$. How could a path get from the start state of M''' (which is the start state of M) to the final state of M''' (which is the start state of M')? The first transition has to be in M, then the path must stay in M until it reaches the final state of M'. The only way out of that state is into M', where it must stay until it reaches the final state and then stops. So the path reads a string in L(M) followed by a string in $L(M')$, as it should.

- In step (5) we created N by adding two new states and four new λ-moves to N. First note that we can read any sequence of zero or more strings in L(M) by going to i, reading each string going from i to f, returning to i each time, then winding up in t. Furthermore, any path from s to t must consist of some combination of trips from i through M to f, and uses of the new λ-moves. So the string we read is the concatenation of zero or more strings in L(M), and thus is in $L(M)^*$.

Some Notes on the Construction

- The construction makes use of the normal form constantly -- if we could not assume that the input λ-NFA's followed the rules, we would need to introduce new states and new λ-moves in steps (3) and (4) as well as in (5). We pay for the normal form in step (5). We need to connect the start and final states, but then to obey the rules we need to put in new start and final states.

- We only create λ-moves when we do step (5). Thus if R has few or no stars, we will get a λ-NFA with few or no λ-moves, which can be good because making an ordinary NFA is more complicated the more λ-moves there are.

- We can sometimes see ways to simplify the λ-NFA without changing the language. But we need to be careful that our simplification is correct.

- It can be shown that the number of states in the λ-NFA is about the same as the length of the regular expression. So the only big blowup is NFA's to DFA's.

An Example: (ab + ba)* + bb

- Let's see how the construction works on a fairly complicated regular expression. (There are diagrams of this example in the text.) We can think of the construction either top-down or bottom-up -- let's try bottom-up.

- The three regular expressions "ab", "ba", and "bb" each get three-state λ-NFA's, with letter moves from the start state to a middle state and from that middle state to a final state.

- The λ-NFA for "ab + ba" has four states, three each for "ab" and "ba" minus two when we merge the two start states and two final states. To get a λ-NFA for (ab + ba)* we add a new start and final state, plus four new λ-moves, to get a six-state λ-NFA with four letter moves and four λ-moves. Finally, we place this six-state machine in parallel with the three-state machine for "bb", getting a seven-state machine with six letter moves and four λ-moves.

Taking This Example to a Minimal DFA

- Killing the λ-moves in this seven-state λ-NFA gives us a seven-state ordinary NFA with state set {i, p, q, r, s, t, f}, start state i, final state set {i, f}, and fourteen transitions: (i, a, q), (i, b, r), (i, b, t), (p, a, q), (p, b, r), (q, b, p), (q, b, f), (q, b, s), (r, a, p), (r, a, s), (r, a, f), (s, a, q), (s, b, r), and (t, b, f).

- We could potentially get 128 states in our DFA, but fortunately the process stops with only seven. State {i} goes to {q} on a and {r,t} on b, state {q} goes to ∅ on a and {p,s,f} on b, state {r,t} goes to {p,s,f} on a and {f} on b, state ∅ stays at ∅ on both, state {p,s,f} goes to {q} on a and to {r} on b, state {f} goes to ∅ on both, and state {r} goes to {p,s,f} on a and to ∅ on b.

- This DFA is minimal. The three final states are {i}, {p,s,f}, and {f}. String bb separates {i} from {p,s,f}, and ab separates these two from {f}. The four non-final states are {q}, {r,t}, {r}, and ∅, and we can separate these pairwise as well.

The Big Picture of Kleene's Theorem

- We are finally ready to finish Kleene's Theorem, proving that a language has a regular expression if and only if it has a DFA. We have shown how to take a regular expression, produce a λ-NFA from it by the recursive construction, kill the λ-moves to get an ordinary NFA, use the Subset Construction to get a DFA, and then (if we want) minimize that DFA.

- The remaining step is to take a DFA and produce a regular expression for its language. As it turns out, the **State Elimination Construction** works equally well to get a regular expression for the language of any ordinary NFA or λ-NFA as well.

- While the first two steps of converting a regular expression to a DFA roughly preserve the size, the Subset Construction in general takes an NFA with k states to a DFA with 2^k states. Though we won't prove this, State Elimination can also cause a large blowup, creating a long regular expression from a small DFA. (Excursion 14.11 in the text takes a closer look at this.)

Another New Model: The r.e.-NFA

- The State Elimination Construction operates on yet another kind of NFA, which we will call an **r.e.-NFA** because the labels on its moves can be arbitrary regular expressions instead of just letters (as in an ordinary NFA) or either letters or λ (as in a λ-NFA).

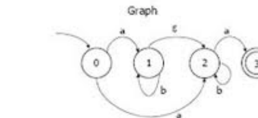
- Not every diagram with regular expressions on its edges is an r.e.-NFA -- we need to satisfy some rules. The first three are the same as the rules in our construction of λ-NFA's from regular expresssions: (1) exactly one final state, not equal to the start state, (2) no moves into the start state, (3) no moves out of the final state. The last rule is new: (4) no **parallel edges**, that is, no two edges with the same start node and end node.

- We have to redefine the Δ^* operation. We still have $\forall s: \Delta^*(s, \lambda, s)$, but now we have the rule $[\Delta^*(s, v, u) \wedge \Delta(u, R, t) \wedge (w \in L(R))] \rightarrow \Delta^*(s, vw, t)$. This rule isn't very useful for computing, as we have no equivalent top-down form for it.

iClicker Question #3: The R.e.-NFA Definition

- Here is another λ-NFA (again with ε in place of λ). **Is it also a legal r.e.-NFA** according to our definition?

NFA that accepts the strings in the language denoted by regular expression ab*a



Example: abba

- (a) No, because there are two moves out of the start state.
- (b) Yes, as long as we consider “ε” to be the regular expression “∅*”.
- (c) No, because the moves have letters and ε rather than regular expressions
- (d) No, because the transition (0, a, 2) is unnecessary.

Overview of the Construction

- The basic idea is to take our original DFA (or NFA, or λ-NFA), modify it so that it obeys the r.e.-NFA rules but still has the same language (how?) and then **eliminate states** one by one until there are only two left. Each elimination will preserve the language of the automaton and ensure that the r.e.-NFA rules still hold.

- An r.e.-NFA with two states must have one of them as the start state and the other as the only final state, by rule (1). By rules (2), (3), and (4), there can be only one edge, going from the start state to the final state, and the only possible path from the start state to a final state has exactly one edge, this one. The edge is labeled by a regular expression R, and the language of the r.e.-NFA is exactly L(R). Thus L(R) is also the language of the original DFA.

- The states we eliminate are every state except the start state and final state. We can eliminate them in any order and get a correct final regular expression, but if we choose the order wisely we may get a simpler regular expression.

Eliminating a State

- Suppose we have a state q that is neither initial nor final, and we want to eliminate it. We don't care about paths that *start* or *end* at q, because the language is defined only in terms of paths that start at the initial state and end at the final state. To safely delete q, we have to *replace* any two-step path, that had q as its middle node, by a single edge.

- If (p, α, q) and (q, β, r) are any two edges, and (q, γ, q) is the loop on q, then when we delete q we must add a new edge $(p, \alpha\gamma^*\beta, r)$. (Here α, β, and γ are regular expressions. Note also that $p = r$ is possible.) If there is already an edge from p to r, though, we add the new edge by changing the existing (p, δ, r) to $(p, \delta + \alpha\gamma^*\beta, r)$. (Note that if there is no loop on q we can take γ to be $\varnothing$ and then $\gamma^* = \varnothing^*$ which is the identity for concatenation, so that $\alpha\gamma^*\beta = \alpha\beta$.)

- When we delete q, we should count all the m edges into q and all the n edges out of q, and make sure that we have added mn new edges. The loop on q, if it exists, does not count toward either m or n.

iClicker Question #4: Eliminating a State

- Suppose we eliminate state 2 from this r.e.-NFA. What is the set of new transitions we must add to replace the state?

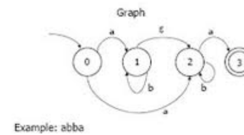
- (a) (0, ab*a, 3) only

- (b) (0, ab*, 2) only

- (c) (0, aa, 3) and (1, a, 3)

- (d) (0, ab*a, 3) and (1, b*a, 3)

NFA that accepts the strings in the language denoted by regular expression ab*a



Example: The Language EE

- In Discussion #11 we designed a regular expression for the language EE of strings over {a, b} that have both a even number of a's and an even number of b's. We'll now use State Elimination to get such an expression from a DFA.

- The DFA has state set {00, 01, 10, 00} -- 00 is the start state and the only final state, a's change the first bit of the state, b's change the second bit. But this DFA violates the rules for an r.e.-NFA -- we have to add a new start state i and a new final state f, and add transitions (i, λ, 00) and (00, λ, f). Now all we have to do is eliminate four states to get our regular expression.

- We begin by killing 01, which has two edges in and two out. We need four new edges: (00, bb, 00), (00, ba, 11), (11, ab, 00), and (11, aa, 11). Next we eliminate 10 (which looks like a good idea as it has no loop and fewer overall edges. Again we get four new edges, each of which is parallel to an existing edge, making (00, aa+bb, 00), (00, ab+ba, 11), (11, ab+ba, 00), and (11, aa +bb, 00). This gives us four states.

Finishing the EE example

- The four remaining states are i, 00, 11, and f. State 11 now has one edge in and one edge out, along with a loop. When we eliminate 11 we create only one edge, (00, (ab+ba)(aa+bb)*(ab+ba), 00). This adds to the existing edge (00, bb+aa, 00), to give us the single edge (00, aa+bb+(ab+ba)(aa+bb)*(ab +ba), 00). Note that this regular expression is exactly what we designed for the language EEP (the nonempty strings in EE that cannot be factored into two other nonempty strings in EE).

- The last state to eliminate is now 00, which also has one edge in, one edge out, and one loop. (Note that a three-state r.e.-NFA must have a form similar to this, maybe with another edge from the initial to final state.) The one edge that we create is (i, [aa+bb+(ab+ba)(aa+bb)*(ab+ba)]*, f), and our final regular expression is the label of this edge.

- We would get a grubbier, equivalent regular expression by eliminating the states in a different order.

Example: The Language No-aba

- We've seen the language Yes-aba = $\Sigma^*aba\Sigma^*$ and its complement No-aba several times now. We have a four-state DFA for No-aba -- let's turn this into a regular expression.

- The state set is {1,2,3,4}, the start state 1, final state set {1,2,3,4}, and edges (1,a,2), (1,b,1), (2,a,2), (2,b,3), (3,a,4), (3,b,1), (4,a,4), and (4,b,4). Again we need new start states i and f, with new edges (i,λ,1), (1,λ,f), (2,λ,f), and (3,λ,f).

- We can just delete 4 and no new edges are needed. Then 2 looks like a good state to kill -- we get the two edges (1, aa*b, 3), and (1, aa*, f), and the latter becomes (1, λ+aa*, f). Now if we kill 3 we create (1, aa*bb, 1) which becomes (1, b + aa*bb, 1) and (1, aa*b, f) which becomes (1, λ + aa* + aa*b, f).

- Killing 1 gives the final expression (b + aa*bb)*(λ + aa* + aa*b).

Example: Number of a's Divisible by Three

- Here's another example (Exercise 14.10.3 in the text). Let D be the language of strings over {a, b} where the number of a's is divisible by 3. It's clear how to make a DFA for this: states {0, 1, 2}, start state and only final state 0, edges (p, b, p) for each state p, and edges (0, a, 1), (1, a, 2), and (2, a, 0). To make an r.e.-NFA, we once again add a new start state i and new final state f, with edges (i, λ, 0) and (0, λ, f). We have five states now and must kill three.

- We first kill 2, creating one new edge (1, ab*a, 0). Then killing 1 creates a new edge (0, ab*ab*a, 0), which adds to the existing (0, b, 0) to get (0, b + ab*ab*a, 0). Finally, killing 0 gives the expression [b + ab*ab*a]*, which makes sense because we can break any string in D into pieces that are either b's or have exactly three a's.

- A more challenging problem is the language of strings where *both* the number of a's and the number of b's are divisible by three. How about the strings where the number of a's and the number of b's are *congruent* modulo 3?