CMPSCI 250: Introduction to Computation

Lecture #36: State Elimination: DFA's to Regular Expressions
David Mix Barrington
23 April 2012

State Elimination: DFA's to Regular Expressions

- Kleene's Theorem Overview
- Another New Model: The r.e.-NFA
- Overview of the Construction
- Eliminating a State
- Example: The Language EE
- Example: The Language No-aba
- Example: Number of a's Divisible by 3

Kleene's Theorem Overview

- 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 ∀s: Δ*(s, λ, s), but now we have the rule [Δ*(s, v, u) ∧ Δ(u, R, t) ∧ (w ∈ L(R))] → Δ*(s, vw, t). This rule isn't very useful for computing, as we have no equivalent top-down form for it.

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, αγ*β, 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, δ + αγ*β, r). (Note that if there is no loop on q we can take γ to be ∅ and then γ* = ∅* which is the identity for concatenation, so that αγ*β = αβ.) On HW#9 you will *prove* that eliminating q in this way preserves the language.

- 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.

Example: The Language EE

- In Discussion #8 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 = Σ*abaΣ* 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 edges (i,λ,f), (i,b,3) which becomes (i,λ+b,3), (1,ab,3), and (1,a,f) which becomes (1, λ+a,f). Now if we kill 3 we create (i, λ+b+bb,1), (i, λ+b, f), (1, abb, 1), and (1, λ+a+ab, f).

- Killing 1 gives the final expression λ + b + (λ + b + bb)(abb)*(λ + a + ab).

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?