

# CS421 Topic 14: Derivatives of Regular Expressions & DFA Minimization<sup>1</sup>

Sameer Sundresh  
sundresh@uiuc.edu

University of Illinois at Urbana-Champaign

June 18, 2007

---

<sup>1</sup>Based on “Derivatives of Regular Expressions” by Janusz Brzozowski, JACM 11.4, October, 1964, pp. 481-494

## Introduction

The regular expressions from last time didn't allow intersections and complements. This means we have to take a roundabout way to express simple concepts like *"all strings that are not valid floating-point numbers"* (complement) or *"all variable names that match this user-supplied regular expression"* (intersection).

The method of derivatives of regular expressions solves this problem, and also lets us skip the step of going through an NFA to create a DFA.

We'll also look at how to minimize a DFA, *i.e.*, given a DFA, create an equivalent DFA with the minimum possible number of states.

# Regular Expressions

Review from last time: regular expressions represent sets of strings on some alphabet  $\Sigma$ , as follows.

- ▶ Let  $a \in \Sigma$ , while  $P$  and  $Q$  are regular expressions.
- ▶  $\epsilon$  is the empty string.

$$[[\epsilon]] = \{\epsilon\}$$

$$[[a]] = \{a\}$$

$$[[PQ]] = [[P]] \times [[Q]]$$

$$[[P^*]] = \{\epsilon\} \cup [[P]] \times [[P^*]]$$

$$[[P + Q]] = [[P]] \cup [[Q]]$$

## Enhanced Regular Expressions

Why can't we do more? Let's add intersection and complementation to our regular expressions.

$$[\epsilon] = \{\epsilon\}$$

$$[\emptyset] = \{\}$$

$$[a] = \{a\}$$

$$[PQ] = [P] \times [Q]$$

$$[P^*] = \{\epsilon\} \cup [P] \times [P^*]$$

$$[P + Q] = [P] \cup [Q]$$

$$[P \& Q] = [P] \cap [Q]$$

$$[\bar{P}] = \Sigma^* \setminus [P]$$

## Enhanced Regular Expressions

Why can't we do more? Let's add intersection and complementation to our regular expressions.

Notice how now we can express arbitrary Boolean operations on regular expressions (we have **or**, **and**, **not**).

But building NFAs to represent regular expressions just became harder. **We can't just compose the NFAs, we have to take intersections and complements, both of which are more involved operations.** Intersection and complement may be easier with DFAs than NFAs, but that doesn't help us much.

# Derivatives of Regular Expressions

There is an algorithm (*Brzozowski, 1964*) based on the notion of a **derivative** of a regular expression which we can use to construct a DFA directly from a regular expression

Brzozowski's algorithm provides natural support for intersections and complements of regular expressions.

## Definition: Derivative of $R$

Let  $R$  be a regular expression and  $s$  a finite string. Define the **derivative** of  $R$  with respect to  $s$  as follows:

$$D_s R = \{t \mid st \in R\}$$

In other words,  $D_s R$  tells us if we've already seen  $s$  what additional input  $t$  we need to match regular expression  $R$ .

**How do we compute the derivative of a regular expression?**

*Note: the  $|$  is part of the set definition, not part of a regular expression; we're using  $+$  as the or operator in our regular expressions.*

## Definition: $\delta(R)$

Given a regular expression, we're going to need an algorithm to construct its derivative. Before we do that, let's define something simpler.

$$\delta(R) = \begin{cases} \epsilon & \text{if } \epsilon \in R \\ \emptyset & \text{if } \epsilon \notin R \end{cases}$$

In other words,  $\delta$  tells us whether or not  $R$  accepts the empty string. Pretty simple, but it will come in handy.

## $R$ in Terms of its Derivatives

Notice that you can always write a regular expression in terms of its first derivatives, as follows.

$$R = \delta(R) + \sum_{a \in \Sigma} aD_aR$$

Where the  $\delta(R)$  term is for the case that  $R$  accepts  $\epsilon$ , while the  $aD_aR, bD_bR, \dots$  terms are for the cases that  $R$  accepts a string starting with the character  $a, b, \dots$

## Computing $\delta(R)$

Ok... so how do you compute  $\delta(R)$ ?

Let  $a \in \Sigma$  and  $P$  and  $Q$  be regular expressions.

$$\delta(a) = \emptyset$$

$$\delta(\epsilon) = \epsilon$$

$$\delta(\emptyset) = \emptyset$$

$$\delta(P^*) = \epsilon$$

*(more on next slide)*

## Computing $\delta(R)$

And here are the more complicated cases.

Again, let  $P$  and  $Q$  be arbitrary regular expressions.

$$\delta(PQ) = \delta(P) \& \delta(Q)$$

$$\delta(P + Q) = \delta(P) + \delta(Q)$$

$$\delta(P \& Q) = \delta(P) \& \delta(Q)$$

$$\delta(\overline{P}) = \begin{cases} \epsilon & \text{if } \delta(P) = \emptyset \\ \emptyset & \text{if } \delta(P) = \epsilon \end{cases}$$

## Computing $D_a R$

Now how do you actually compute the derivative of a regular expression,  $D_a R$ ? Let  $a, b \in \Sigma$ , where  $a \neq b$ , and let  $P$  and  $Q$  be regular expressions. Let's start with just the derivative with respect to a single character,  $a$ .

$$D_\epsilon P = P$$

$$D_a \epsilon = \emptyset$$

$$D_a \emptyset = \emptyset$$

$$D_a a = \epsilon$$

$$D_a b = \emptyset$$

*(more on next slide)*

## Computing $D_a R$

And here are the more complicated cases. Again, let  $P$  and  $Q$  be arbitrary regular expressions and  $a \in \Sigma$ .

$$\begin{aligned}
 D_a P^* &= (D_a P)P^* \\
 D_a(PQ) &= (D_a P)Q + \delta(P)D_a Q \\
 D_a(P + Q) &= (D_a P) + (D_a Q) \\
 D_a(P \& Q) &= (D_a P) \& (D_a Q) \\
 D_a \overline{P} &= \overline{D_a P}
 \end{aligned}$$

*Note that this isn't the **definition** of  $D_a$ , so we need to prove that the above is actually consistent with the definition.*

## Computing $D_s R$

Ok, so we can compute derivatives with respect to a single character; what about arbitrary strings of characters, e.g.,  $s = a_1 a_2 \dots a_n$ ?

$$D_{a_1 a_2} R = D_{a_2}(D_{a_1} R)$$

$$D_{a_1 a_2 a_3} R = D_{a_3}(D_{a_1 a_2} R) = D_{a_3}(D_{a_2}(D_{a_1} R))$$

...

$$D_s R = D_{a_1 \dots a_n} R = D_{a_n}(D_{a_1 \dots a_{n-1}} R) = D_{a_n}(\dots (D_{a_1} R) \dots)$$

*Again, this isn't a definition, it's a theorem.*

## Characteristic Derivatives

We say that two regular expressions  $P$  and  $Q$  are **distinct** if they accept different sets of strings.

### Theorem:

- (a) Every regular expression  $R$  has a finite number  $d_R$  of distinct derivatives, and
- (b) At least one of each of the distinct **characteristic derivatives** occurs amongst the derivatives with respect to strings of length less than  $d_R$ .

*This is more obvious than it looks: it's just another way of looking at a regular expression as a finite automaton.*

## $D_s R$ in Terms of its Derivatives

Just like we wrote  $R$  in terms of its derivatives, we can also write  $D_s R$  in terms of *its* derivatives.

$$D_s R = \delta(D_s R) + \sum_{a \in \Sigma} a D_{u_a} R$$

Where  $D_{u_a} R$  is a derivative **equivalent** to  $D_{sa} R$ . **In other words, the string  $u_a$  might be shorter than  $sa$ .**

## From Derivatives to DFAs

Based on the preceding two slides, we can build up a DFA for  $R$  by examining finitely many derivatives, and creating a state  $q_s$  for each distinct derivative  $D_S R$ .

1. Begin with  $D_\epsilon R = R$ , and create a corresponding new state  $q_\epsilon$
2. Given a new state  $q_s$ , for each  $a \in \Sigma$ :
  - ▶ If there already exists a state  $q_t$  such that  $D_{sa} R$  is equivalent to  $D_t R$ , just add a transition  $q_s \xrightarrow{a} q_t$
  - ▶ Else create a new state  $q_{sa}$  and add the transition  $q_s \xrightarrow{a} q_{sa}$
3. Repeat until no new states are added.

This algorithm creates a minimal DFA for  $R$ .

# Distinguishing Regular Expressions

The only problem is there are many different ways to write regular expressions recognizing the same set of strings.

In general, how do we check whether two derivatives are equivalent?

## Distinguishing Regular Expressions

**Short answer:** We don't have to *perfectly* recognized equivalent regular expressions if all we care about is *finiteness*.

**Definition:** Two regular expressions are **similar** if one can be transformed into the other via the following identities:

$$\begin{aligned}R + R &= R && \text{(idempotence)} \\P + Q &= Q + P && \text{(commutativity)} \\(P + Q) + R &= P + (Q + R) && \text{(associativity)}\end{aligned}$$

**Theorem:** Every regular expression has only a finite number of dissimilar derivatives (*Brzowski, 1964*).

# Distinguishing Regular Expressions

## Long answer:

1. Construct the DFAs for regular expressions  $P$  and  $Q$ .
2. Minimize these DFAs.
3. Check whether the two DFAs are **isomorphic**.
  - ▶ This is a (directed) **graph isomorphism** problem, which is computable, but in general, is not known to be computable in polynomial time (*i.e.*, in P). Curiously, graph isomorphism is also not known to be NP-complete!

So maybe it's best to just create a DFA, only merging *similar* states, and then separately minimize the DFA. Note that a DFA created via the subset construction from an NFA also generally is not minimal.

# Goal

Given a DFA, find an equivalent DFA with the minimum possible number of states.

Minimal DFAs are unique up to isomorphism, while minimal NFAs are not guaranteed to be unique.

## DFA Minimization Algorithm

Given a DFA  $(\Sigma, Q, \Delta, q_0, F)$ ,

1. Remove all states not reachable from  $q_0$  from  $Q$ .
2. Define a set  $S$  of pairs of distinct states  $(p, q)$ :
  - ▶  $(p, q) \in S$  if  $p \in F$  and  $q \notin F$
  - ▶  $(p, q) \in S$  if  $(\Delta(p, a), \Delta(q, a)) \in S$  for some  $a \in \Sigma$
3. Merge states: if  $(p, q) \notin S$ , replace all occurrences of  $q$  by  $p$ .

*Note the similarity to the unification algorithm: transitively build up a substitution, then apply it. Key difference: state graphs can have cycles, while unification fails if  $x = t$  where  $x$  appears in  $t$ .*