

Brzozowski's Automata Algorithms

Incrementality and Parallelism

Bruce W. Watson
with Tinus Strauss, Derrick G. Kourie and Loek Cleophas

FASTAR Research Group
Stellenbosch University, South Africa
bruce@fastar.org

Brzozowski 80, University of Waterloo, 24 June 2015

Application area

Scanning network traffic

- ▶ Very large, or endless
- ▶ Little ability to back-up
- ▶ Many *streams*
- ▶ Packetized
- ▶ Patterns usually expressed as regex's
- ▶ > 3000 patterns combined into one
- ▶ Most implementations based on finite automata

Done roughly two dozen such implementations

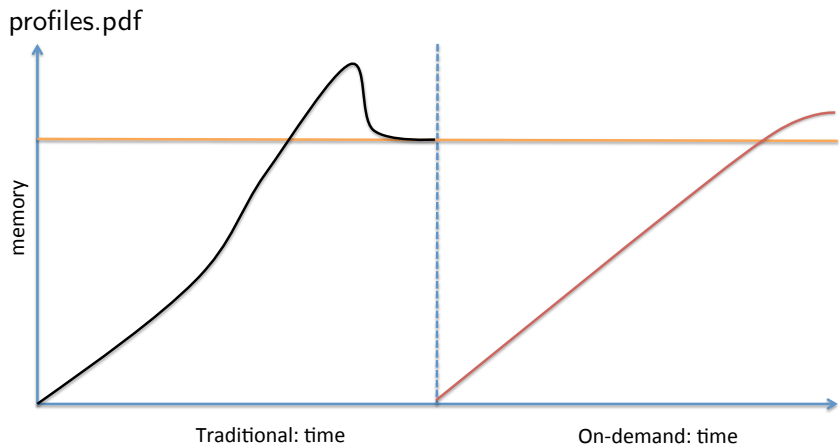
Construct an FA from a regular expression

Lots of solutions, most with some problems:

- ▶ Large intermediate data-structures (data bump)
- ▶ Unable to deal with *extended* regular operators
- ▶ Additional minimization step required
- ▶ States are information poor
- ▶ Not suitable pedagogically
- ▶ Nonincremental (batch)
- ▶ Sequential (not parallel)

Solution turns out to be incremental and parallel versions of Brzozowski's construction

Why incrementality?



Continuations (enroute to derivatives)

For language L and symbol a , define

$$c(L, a) = \{y : ay \in L\}$$

Some of you might write $a^{-1}L$

We can use this to test $x \in L$

Computing continuations

What if we knew L is regular?

$$c(\emptyset, a) = \emptyset$$

$$c(\{\varepsilon\}, a) = \emptyset$$

$$c(\{b\}, a) = \begin{cases} \{\varepsilon\} & \text{if } a = b \\ \emptyset & \text{otherwise} \end{cases}$$

$$c(L_0 \cup L_1, a) = c(L_0, a) \cup c(L_1, a)$$

$$c(L_0 L_1, a) = c(L_0, a) L_1 \cup \begin{cases} c(L_1, a) & \text{if } \varepsilon \in L_0 \\ \emptyset & \text{otherwise} \end{cases}$$

$$c(L_0^*, a) = c(L_0, a) L_0^*$$

$$c(L_0 \cap L_1, a) = c(L_0, a) \cap c(L_1, a)$$

Language membership algorithm

To test $x \in L$, we have a simple algorithm

LANGUAGE MEMBER(x, L)

```
1  while  $x \neq \varepsilon$ 
2  do
3       $L \leftarrow c(L, x[0])$ 
4       $x \leftarrow x[1 \dots]$ 
5  return  $\varepsilon \in L$ 
```

Obviously impractical

(Could have been written as a functional program)

Brzowski's derivatives

For regex E and symbol a , *derivative* of E w.r.t. a is

$$d(0, a) = 0$$

$$d(1, a) = 0$$

$$d(b, a) = \begin{cases} 1 & \text{for } a = b \\ 0 & \text{otherwise} \end{cases}$$

$$d(E_0 + E_1, a) = d(E_0, a) + d(E_1, a)$$

$$d(E_0E_1, a) = d(E_0, a)E_1 + \begin{cases} d(E_1, a) & \text{if } E_0 \text{ is 'nullable', } n(E_0) \\ 0 & \text{otherwise} \end{cases}$$

$$d(E_0^*, a) = d(E_0, a)E_0^*$$

$$d(E_0 \& E_1, a) = d(E_0, a) \& d(E_1, a)$$

Similarly, nullability can also be done inductively.

Language membership algorithm revisited

If we now test language membership against a regex, we have

MEMBER(x, E)

```
1  while  $x \neq \varepsilon$ 
2  do
3       $E \leftarrow d(E, x[0])$ 
4       $x \leftarrow x[1 \dots]$ 
5  return  $n(E)$ 
```

Improving performance:

- ▶ Thanks to finiteness of derivatives, we can *memoize* d .
- ▶ Lazily builds a deterministic finite automaton (DFA).
- ▶ Derivatives are states *and* their right languages.

This is 'lazy-Brzozowski'.

Properties

A regex has a finite number of derivatives, provided we recognize some identities

$$E_0 + E_0 \sim E_0 \quad \text{idempotence}$$

$$E_0 + E_1 \sim E_1 + E_0 \quad \text{commutativity}$$

$$E_0 + (E_1 + E_2) \sim (E_0 + E_1) + E_2 \quad \text{associativity}$$

These are the *bare-bones* Brzozowski simplification rules.

Example of memoization

Invoke $\text{MEMBER}((a + b)^*b, bab)$

Processing first b yields memo d :

$$d(\underbrace{(a + b)^*b}_{\text{state A}}, \underbrace{b}_{\text{state B}}) \mapsto \underbrace{(0 + 1)(a + b)^*b + 1}_{\text{state B}}$$

Second loop iteration, for a , adds to memo:

$$d(\underbrace{(0 + 1)(a + b)^*b + 1}_{\text{state B}}, \underbrace{a}_{\text{state C}}) \mapsto \underbrace{0(a + b)^*b + (1 + 0)(a + b)^*b + 0}_{\text{state C}}$$

Third loop iteration, for the second b adds:

$$d(\underbrace{0(a + b)^*b + (1 + 0)(a + b)^*b + 0}_{\text{state C}}, \underbrace{b}_{\text{state C}}) \mapsto \underbrace{0(a + b)^*b + (1 + 0)(a + b)^*b + 0}_{\text{state C}}$$

We have the beginnings of a 4-state DFA.

Example, redone with simplification

What if we have some regex identities? (for all regex's E)

$$0E, E0 \mapsto 0 \quad 0 + E, E + 0 \mapsto E \quad 1E, E1 \mapsto E$$

Invoke `MEMBER(($a + b$)* b , bab)` Processing first b yields memo d :

$$d(\underbrace{(a + b)^* b}_{\text{state A}}, b) \mapsto \underbrace{(a + b)^* b + 1}_{\text{state B}}$$

Second loop iteration, for a , adds to memo:

$$d(\underbrace{(a + b)^* b + 1}_{\text{state B}}, a) \mapsto \underbrace{(a + b)^* b}_{\text{state A}}$$

Third loop iteration, for the second b , directly goes from state A to B .

Simplification allows *merging* equivalent states.

Implementation choices

Usually

1. Regex's (derivatives) are stored as strings
States are information rich
2. Derivatives are
mapped to integers, then
eventually thrown away
States are information poor
3. Fast transition lookup based on integers

Enriched representations

Is there no way to be *information rich* **and** *efficient*?

Let's have a look at a graph representation of d for a few cases

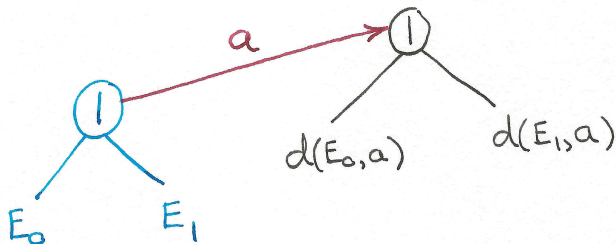
$$d(E_0 + E_1, a) = d(E_0, a) + d(E_1, a)$$

$$d(E_0 E_1, a) = d(E_0, a) E_1 + \begin{cases} d(E_1, a) & \text{if } E_0 \text{ is 'nullable', } n(E_0) \\ 0 & \text{otherwise} \end{cases}$$

$$d(E_0^*, a) = d(E_0, a) E_0^*$$


$$d(E_0 + E_1, a) = d(E_0, a) + d(E_1, a)$$

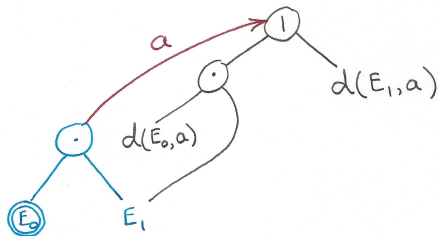
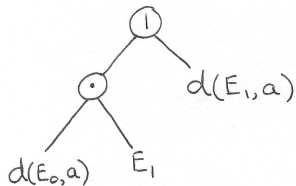
$$d\left(\begin{array}{c} \textcircled{1} \\ / \quad \backslash \\ E_0 \quad E_1 \end{array}, a\right) = \begin{array}{c} \textcircled{1} \\ / \quad \backslash \\ d(E_0, a) \quad d(E_1, a) \end{array}$$



$$d(E_0 E_1, a) = d(E_0, a) E_1 + \begin{cases} d(E_1, a) & \text{if } E_0 \text{ is 'nullable', } n(E_0) \\ 0 & \text{otherwise} \end{cases}$$

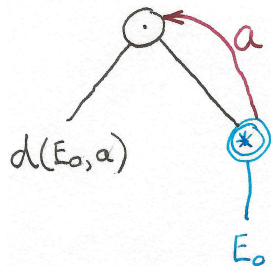
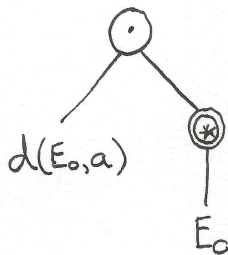
For simplicity $n(E_0)$ case only:

$$d(\text{node}, a) =$$




$$d(E_0^*, a) = d(E_0, a)E_0^*$$

$$d\left(\begin{array}{c} \textcircled{*} \\ | \\ E_0 \end{array}, a\right) =$$



New implementation

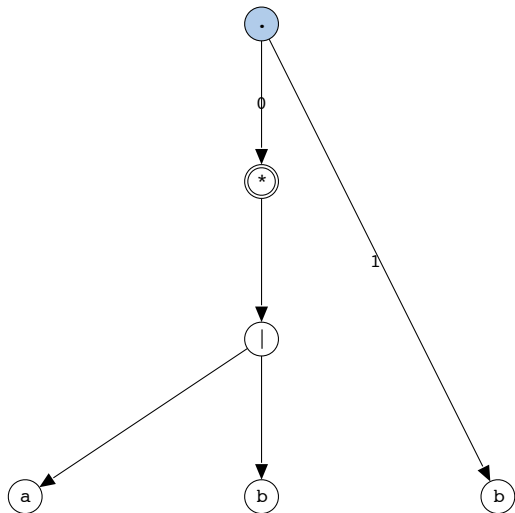
Information rich *and* efficient?

1. **Change representation:** Parse trees for derivatives
 - ▶ Parse DAG nodes *are* states
 - ▶ d is a *node operation*
2. **New invariant:** No duplicate expressions
 - ▶ RHS's of d are full of common subexpressions
 - ▶ Use these to further compress away redundancy

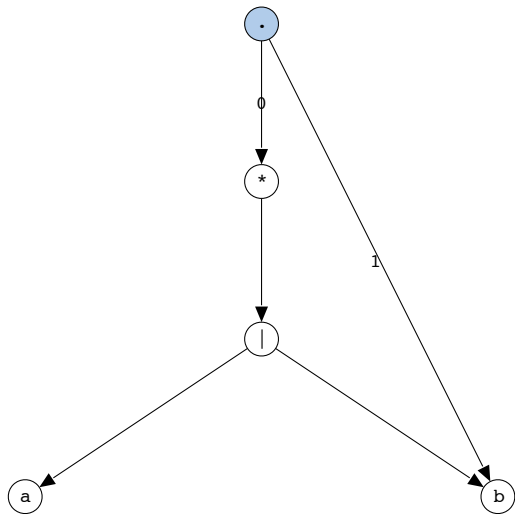
With node sharing, we have parse *DAG's*

3. **Optimize:** Use simplification rules
 - ▶ *Never* build ugly regex's
 - Get arbitrarily & asymptotically close to *minimal*

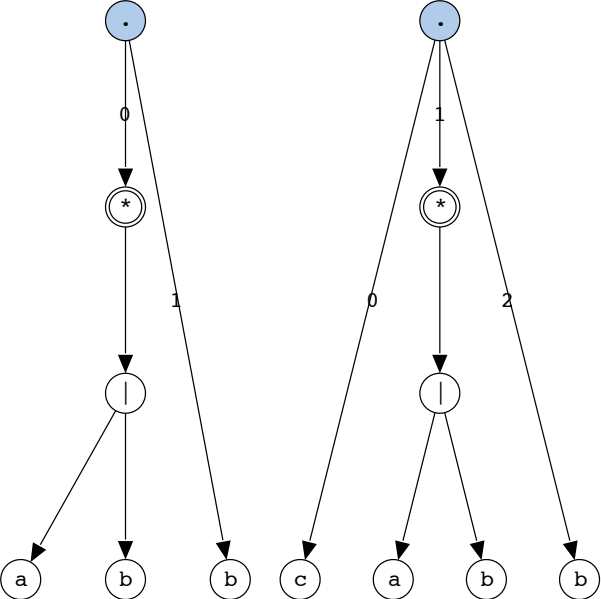
Parse tree for $(a + b)^* b$



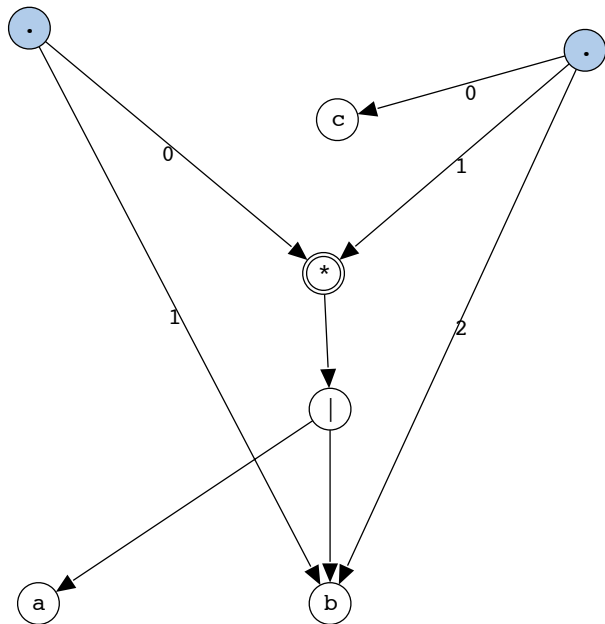
Parse DAG for $(a + b)^* b$



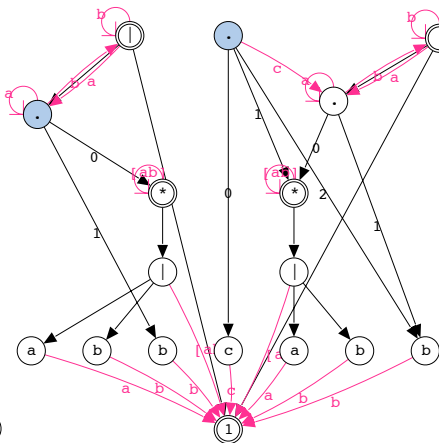
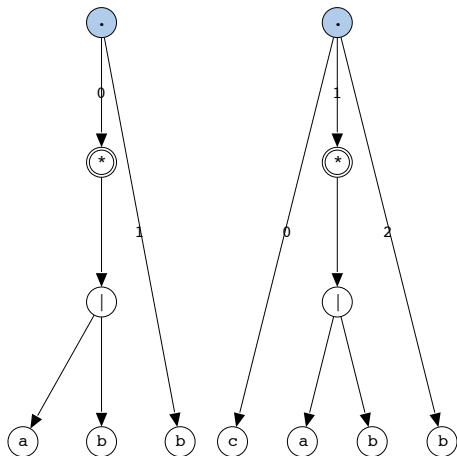
Multiple regex's $(a + b)^*b$ and $c(a + b)^*b$



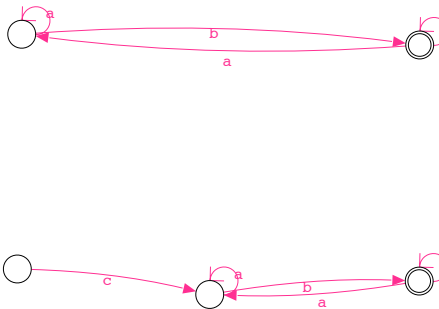
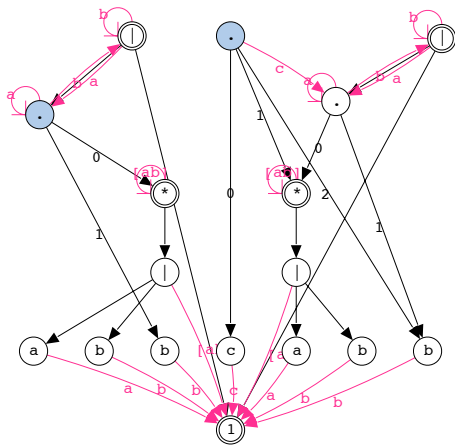
Multiple regex's with sharing



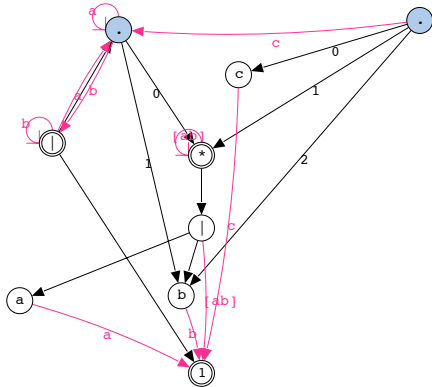
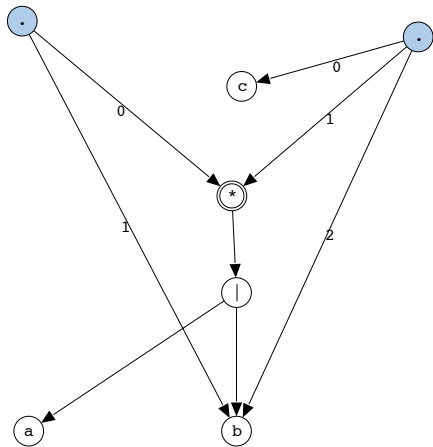
Multiple regex's $(a + b)^*b$ and $c(a + b)^*b$



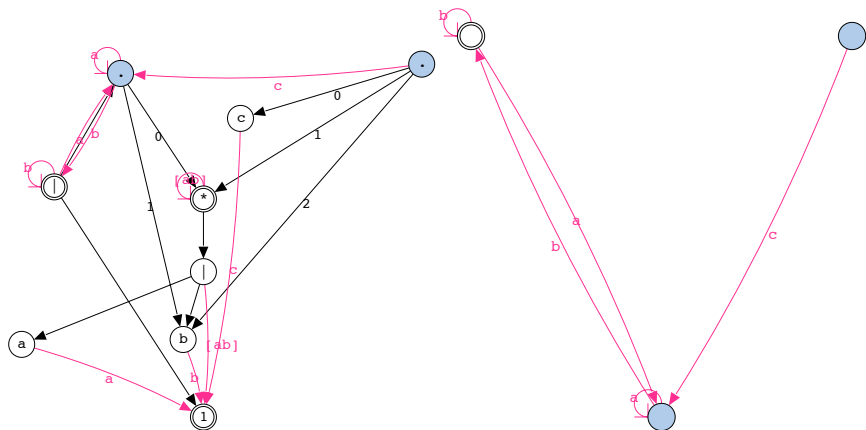
Multiple regex's $(a + b)^*b$ and $c(a + b)^*b$



Multiple regex's with sharing



Multiple regex's with sharing



Properties of the algorithm

Some really exciting properties

- ▶ Annotations of the states is interesting
- ▶ Can get arbitrarily close to minimal DFA
- ▶ Smaller memory consumption 'bump'
- ▶ Global sharing of structures, across many regex's/DFA's
- ▶ Lazy/incremental or eager
- ▶ Shrinking (LRU) is possible by throwing away nodes (and rebuilding later)
- ▶ Approximate DFA is done by aggressive common subexpression elimination

Ongoing and future work

- ▶ We have a CSP specification of Brzozowski's construction
Implementation is efficient
- ▶ Lots of work on combining Brzozowski's two algorithms
Construction & the double-reversal minimization algorithm

derivatives; reverse; determinize; reverse; determinize
done as: *reverse; derivatives; reverse*

- ▶ Extend this to transducers, weighted automata