CDM

Algebra of Regular Languages

Klaus Sutner

Carnegie Mellon University

30-kleene 2017/12/15 23:19

1 Kleene's Theorem

- Conversion To Regular Expression
- Equations*
- Conversion To Machine
- Realistic Regular Expressions

The Structure of Finite State Machines

- We have a number of good algorithms to manipulate FSMs and to test their properties.
- Alas, these machines have one serious drawback: they don't have any nice internal structure and they are somewhat difficult to describe (other than by a brute force table).
- It would be nice if we could somehow build machines in some systematic way, using only trivial basic components and a handful of reasonably simple operations.
- Is there any hope for this?

For the Record: Krohn-Rhodes

There is a basic result from 1962, the Krohn-Rhodes Theorem, arguably one of the most important results in automata theory.

The theorem says, in essence, that every deterministic automaton can be decomposed into simple components.

Alas, the argument uses algebra quite heavily (the simple components are machines corresponding to finite simple groups plus a 2-state flip-flop automaton).

Recently, some actual implementations have been developed, but overall the theorem seems to be mostly of theoretical value.

BTW, this is one of the rare cases where 2 PhDs were granted for one result, Krohn at Harvard and Rhodes at MIT.

Kleene's theorem

Here is another, much easier result that has many direct practical applications.

Theorem (Kleene 1956)

Every regular language over Σ can be constructed from $\emptyset$ and $\{a\}$, $a \in \Sigma$, using only the operations union, concatenation and Kleene star.

Regular languages are closed under other operations such as intersection and complement, but these are not needed to construct a regular language from the basic ones.

Before we sketch the proof, let us introduce the notation system for regular languages suggested by Kleene's result.

Regular Expressions

Definition

A regular expression is a term constructed as follows:

- Basic expressions: $\underline{\emptyset}$, $\underline{a}$ for all $a \in \Sigma$
- Operators: $(E_1 + E_2)$, $(E_1 \cdot E_2)$, $(E^\star)$.

For example,

$$((\underline{a} \cdot (\underline{b}^\star)) + \underline{c})$$

is a regular expression. While correct, this is too clumsy for words: as usual in arithmetic, one uses precedence ordering to avoid parentheses and drops the dot for concatenation.

One often allows $\underline{\varepsilon}$ as a primitive denoting the empty word (though this is technically redundant since $\underline{\emptyset}^\star$ denotes the same language).

The Corresponding Languages

We can associate a language with each regular expression:

$$\begin{aligned}
\mathcal{L}(\underline{\emptyset}) &= \emptyset \\
\mathcal{L}(\underline{a}) &= \{a\} \\
\mathcal{L}(E_1 + E_2) &= \mathcal{L}(E_1) \cup \mathcal{L}(E_2) \\
\mathcal{L}(E_1 \cdot E_2) &= \mathcal{L}(E_1) \cdot \mathcal{L}(E_2) \\
\mathcal{L}(E^\star) &= \mathcal{L}(E)^\star
\end{aligned}$$

So by Kleene's theorem, for every regular language L there is a regular expression α such that $\mathcal{L}(\alpha) = L$.

Lastly, one avoids the underlines and writes things like $ab^\star + c$, it's always clear from context what is meant. One should also be relaxed about identifying $\{x\}$ and x whenever convenient.

Regex Example

Example

All words containing bab: $(a + b)^\star bab(a + b)^\star$.

All words containing 3 a's: $b^\star ab^\star ab^\star ab^\star$

All words not containing aaa: $(\varepsilon + a + aa)(b + ba + baa)^\star$

Exercise

Construct a regex for all words with an even number of a's and b's.

Exercise

Construct a regex for all words with an odd number of a's and b's.

Proof Sketch Kleene

One direction is easy given the results with already have.

To show that $\mathcal{L}(\alpha)$ is regular for α all we need is induction on the build-up of α:

- The claim is trivial for atomic regular expressions.
- For compound regular expressions use closure under union, concatenation and Kleene star.

More on realistic implementations later.

Proof Sketch Kleene, the Hard Part

Suppose we have an NFA that accepts some regular language L. Assume $Q = [n]$. For p, q in Q define

$$L_{p,q} = \mathcal{L}(\langle Q, \Sigma, \delta; \{p\}, \{q\} \rangle)$$

Then $L = \bigcup_{p \in I, q \in F} L_{p,q}$ and it suffices to construct regular expressions for the $L_{p,q}$.

In order to obtain an inductive argument, define a run from state p to state q to be k-bounded if all intermediate states are no greater than k. Note that p and q themselves are not required to be bounded by k.

Now consider the approximation languages:

$$L^k_{p,q} = \{\, x \in \Sigma^\star \mid \text{ there is a } k\text{-bounded run } p \xrightarrow{x} q \,\}.$$

Note that $L^n_{p,q} = L_{p,q}$.

Proof Sketch, cont'd.

One can build expressions for $L^k_{p,q}$ by induction on k.

For $k = 0$ the expressions are easy:

$$L^0_{p,q} = \begin{cases} \sum_{\tau(p,a,q)} a & \text{if } q \neq p, \\ \sum_{\tau(p,a,q)} a + \varepsilon & \text{otherwise.} \end{cases}$$

So suppose $k > 0$. The key idea is to use the equality

$$L^k_{p,q} = L^{k-1}_{p,q} + L^{k-1}_{p,k} \cdot (L^{k-1}_{k,k})^\star \cdot L^{k-1}_{k,q}$$

Done by induction hypothesis. $\square$

Pseudo Code

```
foreach p = 1,..,n do
foreach q = 1,..,n do
    initialize  A[p,q,0];

foreach k = 1,..,n do
foreach p = 1,..,n do
foreach q = 1,..,n do
    A[p,q,k] = A[p,q,k-1] + A[p,k,k-1].A[k,k,k-1]*.A[k,q,k-1];

return sum(  A[p,q,n] : p in I, q in F );
```

Feasibility

The critical line is

```
A[p,q,k] = A[p,q,k-1] + A[p,k,k-1].A[k,k,k-1]*.A[k,q,k-1];
```

and assumes that we have overloaded the arithmetic operators appropriately.

Note that the expression on right is about 4 times bigger than its components, so we are dealing with expressions of exponential size.

In reality, these expressions quickly become unmanageable even if one attempts to simplify terms and keep things concise and short.

Finding the shortest regular expression is unfortunately PSPACE-hard.

Aside 1: Algebra

Note that Kleene's theorem really establishes an algebraic result. Define the language semiring over Σ to be

$$\mathcal{L}(\Sigma) = \langle\, \mathfrak{P}(\Sigma^\star), \cup, \cdot, {}^\star, \emptyset, \varepsilon \,\rangle$$

This is a type of algebra with 3 operations and two constants.

Then Kleene's theorem can be interpreted thus: the least subalgebra generated by the singletons $\{a\}$, $a \in \Sigma$, consists precisely of the regular languages. More later.

Aside 2: Déjà Vu, All Over Again

This should look eminently familiar: logically, Floyd-Warshall's all-pairs shortest path algorithm is essentially the same.

The underlying algebra, the min-plus semiring, is different and simpler (loops are irrelevant for shortest paths). The recursion for dynamic programming looks like this:

$$d^k_{p,q} = \min(d^{k-1}_{p,q}, d^{k-1}_{p,k} + d^{k-1}_{k,q}).$$

And, of course, we are calculating with rational numbers here, not with formal expressions. There is no danger of expressions blowing up.

Aside 3: Simple Application

When arguing about properties of regular languages it is sometimes easier to use regular expressions rather than machines.

For example, consider closure under homomorphisms.

Definition

A homomorphism is a map $f : \Sigma^\star \to \Gamma^\star$ such that

$$f(x_1 x_2 \dots x_n) = f(x_1) f(x_2) \dots f(x_n)$$

where $x_i \in \Sigma$. In particular $f(\varepsilon) = \varepsilon$.

Note that a homomorphism can be represented by a finite table: we only need $f(a) \in \Gamma^\star$ for all $a \in \Sigma$.

Closure under Homomorphisms

Claim

Regular languages are closed under homomorphisms: $f(L)$ is regular whenever L is for any homomorphism f.

Given a homomorphism f (a finite table) define a regular expression α^f over Γ for each α over Σ:

$$\begin{array}{ll} \emptyset \mapsto \emptyset & \alpha + \beta \mapsto \alpha^f + \beta^f \\ \varepsilon \mapsto \varepsilon & \alpha \cdot \beta \mapsto \alpha^f \cdot \beta^f \\ a \mapsto f(a) & \alpha^\star \mapsto (\alpha^f)^\star \end{array}$$

Then $\mathcal{L}(\alpha^f) = f(\mathcal{L}(\alpha))$.

Exercise

Give a machine based proof of the claim.

- Kleene's Theorem
2. Conversion To Regular Expression
- Equations*
- Conversion To Machine
- Realistic Regular Expressions

Machine to Regex

There are three basic approaches to converting a finite state machine to a regular expression:

- Kleene's State elimination method
  The proof of Kleene's theorem provides a dynamic programming algorithm.
- Linear systems of equations
  The machine is converted into a system of linear equations over the language semiring. The system can then be solved using Arden's lemma.
- Eilenberg's prefix/monoidal decomposition
  An essentially unique decomposition based on prefix and monoidal languages.

Converting a finite state machine to a regular expression is a bit of an academic exercise. The key problem is that for all approaches to yield manageable expressions one needs to simplify the intermediate results. There are heuristics to do that, but in general simplification is computationally hard.

Kleene's Method

The proof of Kleene's theorem provides a direct dynamic programming algorithm: construct expression $\alpha(p, q, k)$ in stages $k = 0, \dots, n$ for $1 \leq p, q \leq n$, assuming the state set is $[n]$.

There are two problems this algorithm:

- There are $n^2(n+1)$ expressions to construct, which is bad but not fatal.
- The expressions roughly quadruple in size in the step from level k to level $k+1$. This is a disaster.

In practice, one has to simplify the expressions to make them smaller and choose the elimination order cleverly so that some of the expressions are not needed (top-down versus bottom-up dynamic programming).

Top Down Elimination

Here is a reasonable method for small machines.

- Add two new states b (begin) and e (exit) to the machine. Add ε-transitions from b to all initial states, and from all final states to e.
- Think of the edge labels as regular expressions. We will successively remove all states other than b and e.
- To this end pick some state $r \in Q$. For all incoming transitions $p \xrightarrow{\alpha} r$, outgoing transitions $r \xrightarrow{\gamma} q$, and self-loops $r \xrightarrow{\beta} r$, add a new transition
  $$p \xrightarrow{\alpha\beta^\star\gamma} q$$
  In the end, remove r and all incident transitions.
- Repeat until only b and e remain.

Result

After all states in Q are eliminated we are left with

$$b \xrightarrow{\alpha} e$$

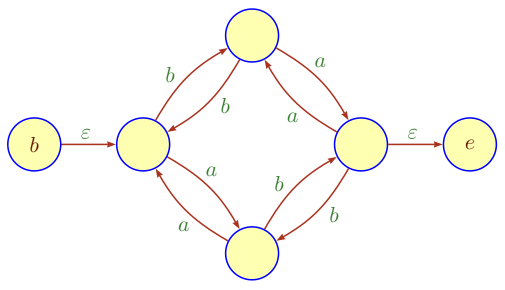
where α denotes the language of the machine.

The proof is to show by induction that, at any point during the construction, the "automaton" is equivalent to the original one. Here we need to generalize the notion of automaton a little: allow regular expressions as labels rather than just letters.

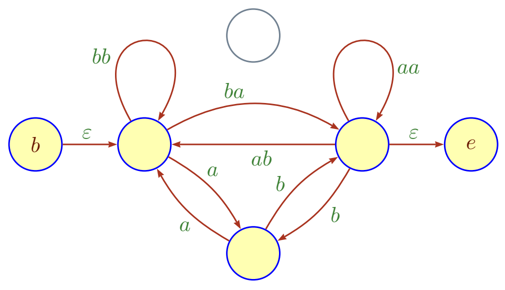
Exercise

Figure out the details.

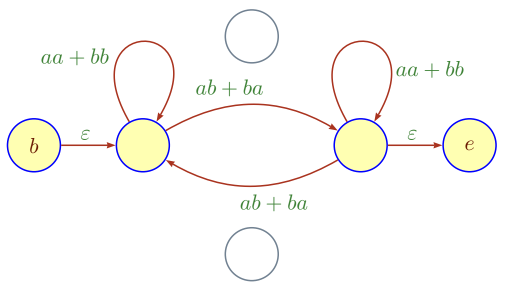
Odd/Odd, Step 0



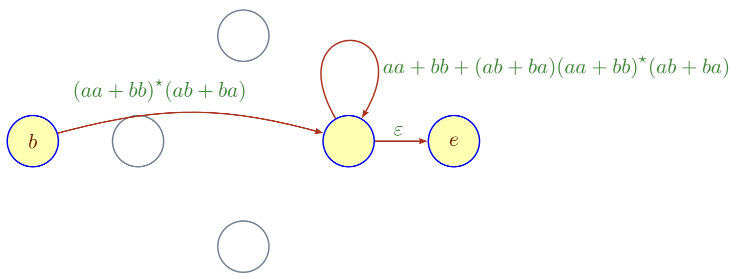
Odd/Odd, Step 1



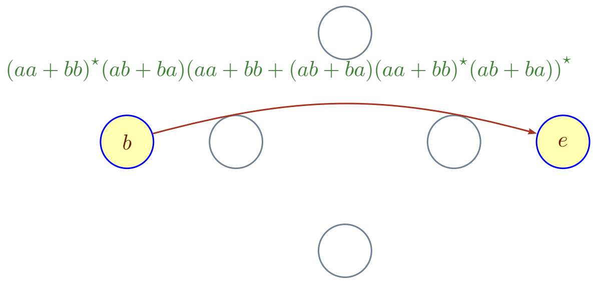
Odd/Odd, Step 2



Odd/Odd, Step 3



Odd/Odd, Step 4


$(aa+bb)^\star(ab+ba)(aa+bb+(ab+ba)(aa+bb)^\star(ab+ba))^\star$
b
e

Final Result

The final regex is

$$\alpha = (aa+bb)^\star(ab+ba)(aa+bb+(ab+ba)(aa+bb)^\star(ab+ba))^\star$$

It's not completely clear that this is correct.

A little argument shows that the last part

$$(aa+bb+(ab+ba)(aa+bb)^\star(ab+ba))^\star$$

denotes all even/even words.

Then α must be correct, too: all odd/odd words consist of a (possibly empty) prefix $(aa+bb)^\star$, followed by $(ab+ba)$, followed by an even/even word.

Simplification

To keep the size of the expressions small it is important to apply various simplifications. For example, the following rules seem reasonable:

$$\begin{gathered}\emptyset \cdot \alpha \mapsto \emptyset \\ \varepsilon \cdot \alpha \mapsto \alpha \\ \alpha + \alpha \mapsto \alpha \\ \varepsilon + \alpha\alpha^\star \mapsto \alpha^\star\end{gathered}$$

Unfortunately, the algebra of regular expressions is complicated and there is no simple set of rules that would produce reasonable expressions.

Basic Equations

Disregarding Kleene star, the basic rules are not too bad:

$$\begin{gathered}(\alpha+\beta)+\gamma = \alpha+(\beta+\gamma) \\ \alpha+\beta = \beta+\alpha \\ \emptyset+\alpha = \alpha \\ \alpha+\alpha=\alpha \\ (\alpha\cdot\beta)\cdot\gamma = \alpha\cdot(\beta\cdot\gamma) \\ \varepsilon\cdot\alpha = \alpha\cdot\varepsilon = \alpha \\ \emptyset\cdot\alpha = \alpha\cdot\emptyset = \emptyset \\ \alpha\cdot(\beta+\gamma) = \alpha\cdot\beta+\alpha\cdot\gamma \\ (\beta+\gamma)\cdot\alpha = \beta\cdot\alpha+\gamma\cdot\alpha\end{gathered}$$

Alas ...

Rules for Kleene

But Kleene star causes huge problems. Here are some (natural?) rules:

$$
\begin{aligned}
(\alpha + \beta)^\star &= (\alpha^\star \beta)^\star \alpha^\star \\
(\alpha\beta)^\star &= \varepsilon + \alpha(\beta\alpha)^\star \beta \\
(\alpha^\star)^\star &= \alpha^\star \\
\alpha^\star &= (\varepsilon + \alpha + \ldots + \alpha^{n-1}) \cdot (\alpha^n)^\star
\end{aligned}
$$

Note that the last equation is actually an infinite family of equations; it is known that no finite family will do.

Not to mention that for simplification we need rewrite rules, not equations.

Horror Example

Here is the result running a conversion algorithm with some degree of simplifications on the even/even example (which is easier than odd/odd):

$$
\begin{aligned}
&\epsilon + b(bb)^\star b + (a + b(bb)^\star ba)\,(aa + ab(bb)^\star ba)^*\,(a + ab(bb)^\star b) + \\
&\quad \big(b(bb)^\star a + (a + b(bb)^\star ba)\,(aa + ab(bb)^\star ba)^*\,(b + ab(bb)^\star a)\big) \\
&\quad \big(a(bb)^\star a + (b + a(bb)^\star ba)\,(aa + ab(bb)^\star ba)^*\,(b + ab(bb)^\star a)\big)^* \\
&\qquad \big(a(bb)^\star b + (b + a(bb)^\star ba)\,(aa + ab(bb)^\star ba)^*\,(a + ab(bb)^\star b)\big)
\end{aligned}
$$

It takes quite a bit of effort just to check that this expression describes the even/even language.

Eilenberg's Method

The dynamic programming approach to the construction of an equivalent regular expression has the disadvantage that the choice of the next vertex to be eliminated is arbitrary, leading to many possible expressions.

However, there is another approach due to Eilenberg (1973) that produces an expression that is essentially unique (up to associativity and commutativity).

The key is a complexity measure that is slightly different from ordinary state complexity. Suppose M is the minimal DFA for some language L. If M has a sink, remove it; call the resulting automaton the *minimal partial DFA (MPDFA)*.

$$\gamma(L) = \#\text{ transitions in MPDFA for } L$$

Note that this counts transitions, not states as in ordinary state complexity.

Unitary Languages

A language is unitary if it's minimal DFA has exactly one final state.

Clearly, every regular language has a unique decomposition as a disjoint union of unitary languages.

$$L = \bigcup L_i$$

Just pick a single final state in the minimal DFA for each component.

Prefix Languages

A language L is prefix (or prefix-free) if no proper prefix of a word in L belongs to L.

Clearly, "prefix-free" makes sense and "prefix" does not, but there is nothing you can do about established terminology.

What can we say about the DFA for a prefix language?

There cannot be a path from any final state back to a final state. So in the MPDFA there are no transitions with source a final state. Note that this is if and only if.

A prefix language is either empty or unitary.

Monoidal Languages

A language L is monoidal if $L^\star = L$.

Proposition

A language L is monoidal iff $\varepsilon \in L$ and $L\,L \subseteq L$.

Proof. Assume L is monoidal. Then $\varepsilon \in L^\star = L$ and $L\,L \subseteq L^\star = L$.

For the opposite direction note that $L\,L = L$ since $\varepsilon \in L$. By induction $L^k = L$ for all $k \geq 1$, so $L^\star = \varepsilon + \sum_{i \geq 1} L = \varepsilon + L = L$ (super idempotency).

$\square$

More Monoidal Languages

Proposition

A monoidal language L can be written uniquely as $L = K^\star$ where K is prefix.

Proof. Consider the minimal PDFA M for L: the unique final state must also be the initial state: $M = \langle Q, \Sigma, \delta; p, \{p\} \rangle$.

Define a new PDFA M' as follows:

- Remove p from Q and add q_0 and q_1.
- q_0 is initial and q_1 is final.
- For $p \xrightarrow{a} q$ introduce $q_0 \xrightarrow{a} q$.
- For $q \xrightarrow{a} p$ introduce $q \xrightarrow{a} q_1$.
- For $p \xrightarrow{a} p$ introduce $q_0 \xrightarrow{a} q_1$.

Then $K = \mathcal{L}(M')$ is prefix and $K^\star = L$. □

Prefix/Monoidal Decomposition

Proposition

A unitary language L can be written in the form $L = AB$ where A is prefix and B is monoidal. Moreover, this decomposition is unique.

Proof. Let M be the minimal PDFA for L.

Let A be the language accepted by the partial DFA obtained by removing all transitions coming out of the final state.

Let B be the language accepted by the partial DFA obtained by making the final state also the initial state.

Uniqueness can be verified. □

Now What?

Using both results we have a representation as a disjoint union

$$L = \bigcup A_i B_i$$

where A_i and B_i both are prefix.

We need to further decompose these pieces.

A priori, it is not clear how decompose prefix languages. Note that we have to make sure γ-complexity does not increase.

Decomposing Prefix Languages

We have already seen left quotients $a^{-1}L$: remove a prefix a whenever possible.

By symmetry, we could also remove a suffix a. Let's write L/a for these right quotients.

Claim

Let L be prefix. Then

$$L = \bigcup_{a \in \Sigma} (L/a) \cdot a$$

and the decomposition is unique.

Proof. Well, if you knock of an a you've got to put it back. □

Pulling Things Together

It is a labor of love to check that in each of our decomposition steps the new languages have smaller γ-complexity than the old ones except for the monoidal-to-prefix step: there the complexity stays the same.

For example, for L prefix we have

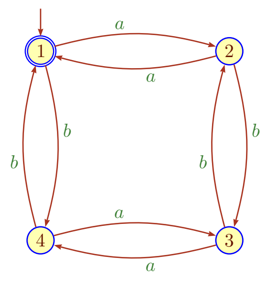
$$\gamma(L/a) < \gamma(L)$$

for all $a \in \Sigma$.

Exercise

Check that γ is decreasing in our decomposition.

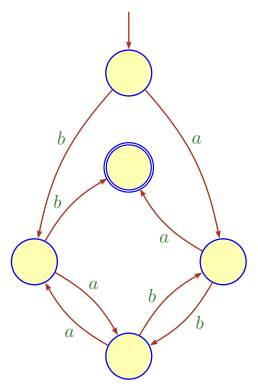
Example: Even/Even



The minimal DFA for the even/even language L.

Note that L is already monoidal, so the first step is to find a prefix L_1 such that $L = {L_1}^\star$.

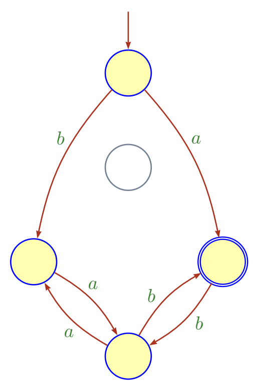
Prefix L_1



Splitting the initial/final state and redirecting transitions.

The new language is duly prefix. Next step is to decompose $L_1 = L_2 \cdot a + L_3 \cdot b$.

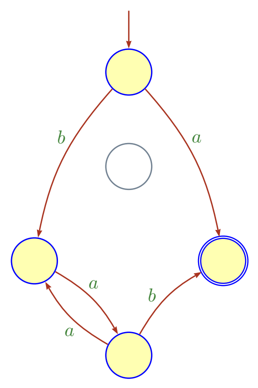
First Quotient of L_1



The quotient $L_2 = L_1/a$.

L_2 is unitary, so the next step is to decompose $L_2 = L_3 \cdot L_4$.

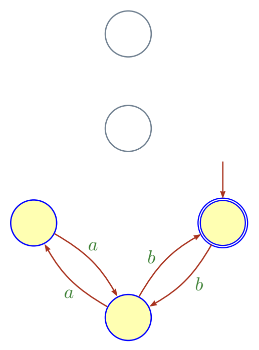
Prefix Part of L_2



The prefix part L_4 of L_2.

Note that $L_4/a = \{\varepsilon\}$, so the end is in sight.

Monoidal Part of L_2



The monoidal part L_5 of L_2.

It is clear that the prefix part of L_5 is $b(aa)^\star b$.

Putting it All Together

After a few more steps we find the regular expression

$$\big((a + ba(aa)^\star b)(b(aa)^\star b)^\star a + (b + ab(bb)^\star a)(a(bb)^\star a)^\star b\big)^\star$$

Compare this to the first expression we have for L. There is no similarity whatsoever.

$$(aa + bb + (ab + ba)(aa + bb)^\star(ab + ba))^\star$$

In light of examples like this it is not too surprising that equivalence of regular expressions is PSPACE-hard.

- Kleene's Theorem
- Conversion To Regular Expression

3 Equations*

- Conversion To Machine
- Realistic Regular Expressions

Arden's Lemma

Yet another conversion algorithm can be based on the algebra of regular languages. Consider a linear language equation of the form

$$X = A \cdot X + B.$$

where $A, B \subseteq \Sigma^\star$ are arbitrary languages, though we will be mostly interested in the case where A and B are finite or perhaps regular. We are looking for a solution $X_0 \subseteq \Sigma^\star$.

As it turns out, linear equations are rather easy to solve.

Lemma (Arden's Lemma)

Let A and B be languages over Σ. Then the equation $X = A \cdot X + B$ has a solution $X_0 = A^\star B$. Moreover, if $\varepsilon \notin A$, then this solution is unique. In any case, X_0 it is the smallest solution (with respect to set-theoretic inclusion).

Proof

To see that $X_0 = A^\star B$ is a solution note that

$$AX_0 + B = AA^* B + B = (A^+ + \varepsilon)B = A^* B = X_0.$$

Now let Z be any solution, so $Z = AZ + B$. Then for all $k \geq 0$:

$$Z = A^{k+1} Z + (A^k + \ldots + \varepsilon)B.$$

Hence $X_0 \subseteq Z$.

Lastly suppose $\varepsilon \notin A$ and let $x \in Z$; set $k = |x|$. Then $x \notin A^{k+1}Z$, whence necessarily $x \in (A^k + \ldots + \varepsilon)B \subseteq X_0$ and we are done. $\square$

In the applications of interest to us $\varepsilon \notin A$ so that the solution is unique.

Fixed Point Perspective

Recall the Knaster-Tarski theorem: a monotonic map on a complete lattice has a fixed point (in fact, the collection of fixed points is a complete sublattice).

The lattice we are interested in here is the powerset of $\Sigma^\star$:

$$\langle \mathfrak{P}(\Sigma^\star), \cup, \cap \rangle$$

Note that this lattice is trivially complete: the union of any family of languages is again a language (though possibly a very complicated one).

Equations and Monotonicity

Now consider the following map on the powerset lattice:

$$f(Z) = A \cdot Z + B$$

f is trivially monotonic, so it must have a fixed point. In fact, we can construct the least fixed point inductively:

$$Z_0 = \emptyset$$
$$Z_{i+1} = A \cdot Z_i + B$$

Then $\bigcup_i Z_i$ is the least fixed point (X_0 on the last slide).

Regularity

Note that Arden's lemma implies that equation $X = A \cdot X + B$ has a regular solution $X_0 = A^* B$ whenever A and B are regular. Moreover, if A does not contain ε this is the only solution. If A and B are given as rational expressions we obtain a rational expression for the solution.

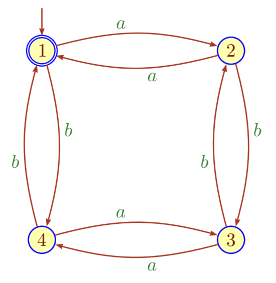
Example

The equation $X = bX + a$ has $b^* a$ as its unique solution.

By contrast, the equation $X = b^* X + a$ has infinitely many solutions $X_k = b^*(a + a^2 + \cdots + a^k)$ and $X_1 = b^* a$ is the least solution. Indeed, there are uncountably many solutions $b^* \cdot L$ where $a \in L \subseteq a^*$ is arbitrary.

Even/Even Example

Let's return to the even/even language. The canonical DFA looks like so:



We convert the DFA into a system of equations.

Corresponding System

$$X_1 = \varepsilon + aX_2 + bX_4 \quad (1)$$
$$X_2 = aX_1 + bX_3 \quad (2)$$
$$X_3 = aX_4 + bX_2 \quad (3)$$
$$X_4 = aX_3 + bX_1 \quad (4)$$

Substituting (2) and (4) into (1) and (3) we get

$$X_1 = \varepsilon + (aa + bb)X_1 + (ab + ba)X_3 \quad (5)$$
$$X_3 = (aa + bb)X_3 + (ab + ba)X_1 \quad (6)$$

Applying Arden's lemma to (6) we get

$$X_3 = (aa + bb)^\star(ab + ba)X_1 \quad (7)$$

The Solution

Substituting (7) into (5)

$$X_1 = \varepsilon + ((aa + bb) + (ab + ba)(aa + bb)^\star(ab + ba))X_1$$

Applying Arden's lemma one more time we get

$$X_1 = (aa + bb + (ab + ba)(aa + bb)^\star(ab + ba))^\star$$

which solution makes intuitive sense.

It is important to note, though, that we tacitly did quite a bit of cleanup along the way, writing the expressions in a simplified form. Remember the horror example, from above.

Another Example

Needless to say, solving systems of equations does not always produce a neat solution either. Consider the system

$$X = aX + bY$$
$$Y = \varepsilon + aX + ba^*$$

If we solve for X in the first equation, $X = a^*bY$, and substitute in the second and then solve for Y and resubstitute we get

$$X = a^*b(a^+b)^*(\varepsilon + ba^*)$$
$$Y = (a^+b)^*(\varepsilon + ba^*)$$

So far, so good.

A Different Approach

However, we could also first substitute the second equation into the first and solve for X.

This leads to solutions

$$X' = (a + b(ab)^*aa)^*b(ab)^*(\varepsilon + ba^*)$$
$$Y' = (ab)^*\,(aa(a + b(ab)^*aa)^*b(ab)^*(\varepsilon + ba^*) + \varepsilon + ba^*)$$

Unless we made a mistake, these expressions must be equivalent to X and Y, but this is certainly not obvious from looking at them.

What is sorely missing here is a simplification algorithm that brings a regular expression into a normal form. We can check for equivalence by converting to finite state machines and then testing these for equivalence.

Matrix Method

The even/even system from above could also be written as

$$\boldsymbol{X} = A \cdot \boldsymbol{X} + B$$

where

$$A = \begin{pmatrix} 0 & a & 0 & b \\ a & 0 & b & 0 \\ 0 & b & 0 & a \\ b & 0 & a & 0 \end{pmatrix}$$

and $B = (1, 0, 0, 0)$.

Here we have written 0 and 1 instead of $\emptyset$ and $\{\varepsilon\}$ for legibility.

Matrix Method, II

One can then define the Kleene star of A

$$\boxed{A^\star = I + A + A^2 + \ldots + A^n + \ldots}$$

and it is not hard to see that the solution is of the form

$$\boldsymbol{X} = A^\star B$$

Of course, to make this truly useful we need to find a way to compute $A^\star$.

In simple cases one can determine an approximation $\sum_{i \leq n} A^i$ for some small n and then make an educated guess as to what $A^\star$ might be.

More Algebra

A better way is to take a closer look at the underlying algebraic structure:

$$\mathcal{L}(\Sigma) = \langle \mathfrak{P}(\Sigma^\star), \cup, \cdot, ^\star, \emptyset, \varepsilon \rangle$$

This is one example of a closed semiring: intuitively, a semiring that also supports an infinite sum operation:

$$x^* = \sum_{i \geq 0} x^i = x^0 + x^1 + x^2 + \ldots$$

Of course, this is a little problematic: the "..." don't really have any precise meaning.

Closed Semiring Axioms

Let S be an idempotent semiring with an additional infinitary operation $\sum_{i \in I} a_i$. S is a closed semiring if this infinite summation operation behaves properly with respect to finite sums, distributivity and reordering of the arguments. More precisely, we require

$$\sum_{i \in [n]} a_i = a_1 + \ldots + a_n,$$
$$(\sum_{i \in I} a_i)(\sum_{j \in J} b_j) = \sum_{(i,j) \in I \times J} a_i b_j,$$
$$\sum_{i \in I} a_i = \sum_{j \in J} (\sum_{i \in I_j} a_i)$$

Consequences

Note the arbitrary index sets floating around in these axioms. They are inherently more complicated than traditional, purely equational axioms for standard structures such as groups, rings or fields.

At any rate, one can still derive general results from these axioms:

Lemma

In any closed semiring we have

$$(x + y)^* = (x^* y)^* x^*$$
$$(xy)^* = 1 + x(yx)^* y$$

The Language Semiring

It is easy to check that $\mathcal{L}(\Sigma)$ is a closed semiring.

More importantly, the collection of all $n \times n$ matrices over $\mathcal{L}(\Sigma)$ is again a closed semiring:

$$\mathcal{L}(\Sigma)^{n \times n} = \langle \mathfrak{P}(\Sigma^\star)^{n \times n}, \cup, \cdot, ^\star, \mathbf{0}, \mathbf{1} \rangle$$

where $\mathbf{0}$ denotes the null matrix (all entries $\emptyset$) and $\mathbf{1}$ denotes the identity matrix ($\{\varepsilon\}$ along the diagonal).

The additive operation is simply pointwise, and multiplication is standard matrix multiplication. Star is defined as above by an infinite sum.

Computing Star

So far all we have is a clean framework. Algorithmically, we still need to find a way to calculate a star in $\mathcal{L}(\Sigma)^{n \times n}$.

To this end consider a matrix

$$X = \left(\begin{array}{c|c} A & B \\ \hline C & D \end{array} \right)$$

where the pieces have size $n/2$ (put floors and ceilings in the right places).

Then there is a divide-and-conquer algorithm to compute $X^\star$.

Computing Star II

Let $Y = (A + BD^\star C)^\star$ and $Z = (D + CA^\star B)^\star$. Then

$$X^\star = \left(\begin{array}{c|c} Y & YBD^\star \\ \hline ZCA^\star & Z \end{array} \right)$$

Since the dimension of the component matrices shrinks by 1/2 there are only $\log n$ levels in the recursion. Of course, the expressions can still get ugly.

Example

Let's compute the star of

$$X = \begin{pmatrix} 0 & a & 0 & b \\ a & 0 & b & 0 \\ 0 & b & 0 & a \\ b & 0 & a & 0 \end{pmatrix}$$

So the submatrices are

$$A = D = \begin{pmatrix} 0 & a \\ a & 0 \end{pmatrix} \qquad B = C = \begin{pmatrix} 0 & b \\ b & 0 \end{pmatrix}$$

A moment's thought shows

$$\begin{pmatrix} 0 & x \\ x & 0 \end{pmatrix}^\star = \begin{pmatrix} (xx)^\star & x(xx)^\star \\ x(xx)^\star & (xx)^\star \end{pmatrix}$$

Example II

But then

$$Y = Z = (A + BA^\star B)^\star = \begin{pmatrix} b(aa)^\star b & a + ba(aa)^\star b \\ a + ba(aa)^\star b & b(aa)^\star b \end{pmatrix}^\star$$

Since we are only interested in the top/left entry of $X^\star$ we can apply the decomposition one more time to obtain yet another regular expression for the even/even language:

$$X^\star(1,1) = \big(b(aa)^\star b + (a + ba(aa)^\star b)(b(aa)^\star b)^\star(a + ba(aa)^\star b)\big)^\star$$

Three Expressions

We now have tree different but equivalent expressions for the even/even language, obtained by different methods:

$$(aa + bb + (ab + ba)(aa + bb)^\star(ab + ba))^\star$$

$$\big((a + ba(aa)^\star b)(b(aa)^\star b)^\star a + (b + ab(bb)^\star a)(a(bb)^\star a)^\star b\big)^\star$$

$$\big(b(aa)^\star b + (a + ba(aa)^\star b)(b(aa)^\star b)^\star(a + ba(aa)^\star b)\big)^\star$$

Exercise

Make sure you understand how these three expressions work.

- Kleene's Theorem
- Conversion To Regular Expression
- Equations*
- 4 Conversion To Machine
- Realistic Regular Expressions

Regex To Machine

We will ignore the issue of parsing a regular expression and simply worry about how to construct the finite state machine.

It is not hard to guess that we will build the machine by induction on the structure of the regular expression, but the question is what kind of machine we should be build.

DFAs are certainly a bad choice since we have to deal with concatenation and Kleene star. It is a fair guess that some kind of NFAE is the right target architecture.

As it turns out, the construction becomes quite a bit easier if insist on very special NFAEs. Of course, the right choice requires a bit of experimentation.

Begin/Exit Automata

Definition

A begin/exit automaton (BEA) is an NFAE that has exactly one initial state b (the begin), exactly on final state e (the exit). Furthermore, no transitions end at b and no transitions start at e.

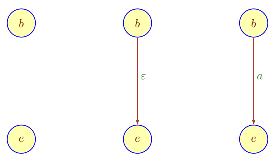
If we assume the state set $[n]$ a BEA is essentially just a list of transitions: by renumbering we can make sure that $b = 1$ and $e = n$.

Hence the data structure representing a BEA is particularly simple.

There are other ways to go about the conversion, but BEAs are probably the most intuitive choice.

The Basic machines

For expressions $\emptyset$, ε and a the corresponding BEAs are as follows.



In practice, the BEA for $\emptyset$ is never used.

Thompson Automata

Definition

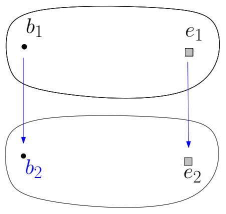
A Thompson automaton is a begin/exit automaton that satisfies the additional condition that no transitions end at the begin, and no transitions start at the exit.

The basic BEAs are all Thompson automata.

As it turns out, the other operations become very natural and efficient for Thompson automata.

Operations: Union

Union for Thompson automata.

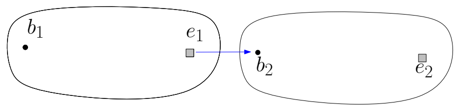


All new transitions are ε-transitions.

Number of states: $n_1 + n_2$, number of transitions: $t_1 + t_2 + 2$.

Operations: Concatenation

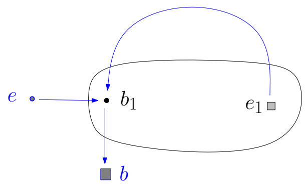
Union for Thompson automata.



Number of states: $n_1 + n_2$, number of transitions: $t_1 + t_2 + 1$.

Operations: Star

A BEA for the Kleene star of a BEA.

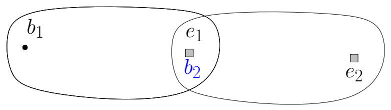


Number of states: $n_1 + 2$, number of transitions: $t_1 + 3$.

Improvements

So far we have hardly used the conditions on begins and exits. They can be exploited to streamline the sum and product operations.

For example, a BEA for the product can be built like so:



The number of states is $n_1 + n_2 - 1$, the number of transitions is $t_1 + t_2$.

Exercise

Figure out how to optimize sums of BEAs.

Correctness

Even without optimization we have a linear time conversion algorithm.

Theorem

A regular expression can be converted in linear time into an equivalent begin/exit automaton.

Proof. Write α, σ, π and κ for the number of atomic symbols, sums, products and Kleene stars in the regular expression, respectively. Then the number of states/transitions in the corresponding BEA is bounded by

$$2\alpha + 2\kappa \qquad\qquad \alpha + 2\sigma + \pi + 3\kappa$$

□

Exercise

Determine the size of a BEA with optimization.

BEA Example

Expression $(\varepsilon + a + aa)(b + ba + baa)^\star$ produces an 8-state BEA with 13 transitions (with brutal optimization):

1	1	1	2	3	4	4	4	4	5	6	7	8
a	a	ε	a	ε	b	b	b	ε	a	a	a	ϵ
2	3	3	3	4	5	6	8	9	8	7	8	4

ε-elimination produces an NFA with initial states $\{1, 3, 4, 9\}$ and final states $\{9\}$.

1	1	1	1	2	2	2	4	4	4	4	4	5	5	5	6	7	7	7
a	a	a	a	a	a	a	b	b	b	b	b	a	a	a	a	a	a	a
2	3	4	9	3	4	9	4	5	6	8	9	4	8	9	7	4	8	9

Deterministic Machines

Determinizing the NFA yields a DFA on 7 states (initial state 1, all states other than the sink 6 are final).

	1	2	3	4	5	6	7
a	2	4	5	6	7	6	6
b	3	3	3	3	3	6	3

The corresponding minimal DFA has 4 states (initial state 1 and final states $\{1, 2, 3\}$).

	1	2	3	4
a	2	3	4	4
b	1	1	1	4

Blow-Up

The regular expression

$$(a + b)^\star a(a + b)(a + b)(a + b)(a + b)(a + b)$$

for $L(a, -6)$ produces a BEA with 10 states

1	2	2	2	3	4	5	5	6	6	7	7	8	8	9	9
ε	a	b	ε	ε	a	a	b	a	b	a	b	a	b	a	b
2	3	3	4	2	5	6	6	7	7	8	8	9	9	10	10

…but the DFA obtained from conversion has 65 states; the minimal DFA has 64 states.

- Kleene's Theorem
- Conversion To Regular Expression
- Equations*
- Conversion To Machine

5 Realistic Regular Expressions

Extended Regex

Our definition of regular expressions directly mimics Kleene's theorem. Sometimes it is convenient to enhance regular expressions a bit. There are two types of enhancements:

- More compact ways to describe regular languages.
- Describing non-regular languages.

Type 1 comes down to notational convenience, but may have radical effects on the running time of the conversion algorithm. For example, we know that regular languages are closed under intersection and complement. If we were to add corresponding regular expressions for these operations we would still describe regular languages. However, the conversion process based on BEAs now fails: the only way we can perform, say, complementation is by converting to a DFA first. This conversion may carry an exponential cost.

Type 2 requires a redesign of the acceptance testing algorithm: finite state machines are no longer sufficient, though the modifications may turn out to be fairly easy to do from an algorithmic point of view.

Intersection

As an example, consider the addition of a new operation symbol $\cap$ for intersection to regular expressions.

What has to change in the conversion algorithm? We need to add a product automata construction.

From the implementation perspective this is not too bad, but it breaks polynomial bounds on the size of the machine constructed from a regex. The following theorem shows that there is little hope for a simple remedy.

Theorem

Suppose $M_1, \ldots, M_n$ is a list of DFAs (over the same alphabet). It is PSPACE*-hard to check whether there is a string that is accepted by all the* M_i.

Complement

Similar problems arise when a complementation operation $\neg$ is added. In the Real World$^{\mathrm{TM}}$ complementation can often be kludged by piping.

```
fgrep  foo  file.txt | fgrep -v foobag
```

But to do this in general we have to construct a machine for the complement of a regular language – and that requires to build a DFA first, at a potentially exponential cost.

Also note that complement together with union automatically produces intersection, so the hardness result from above applies.

Iteration

A very handy feature in most regular expression matchers generalizes concatenation and Kleene star.

notation	number of matches
`*`	≥ 0
`+`	≥ 1
`?`	$= 0, 1$
`{n}`	$= n$
`{n,}`	$\geq n$
`{n,m}`	$\geq n, \leq m$

So one can write things like

```
egrep -e '0\.[0-9]{5}'
```

to find all decimal numbers starting with 0 with exactly 5 digits after the decimal point.

No DFAs

Grep maintains nondeterministic machines and performs the final matching phase with these machines, there is no determinization nor minimization.

For example, there is no problem in looking for an a in position -100.

```
egrep -e '^[a-z]*a[a-z]{99}$'
```

The corresponding minimal DFA has 2^{100} states.

Non-Regular Extensions

Many pattern matchers allow the user to specify repetitions of previously matched parts of a string.

The standard `egrep` for example allows

```
egrep -e '^([a-z]*)\1$'
```

which will match words of the form ww with lower-case letters. It is not hard to see that this is non-regular language.

The `\1` is a so-called *back-reference* and matches whatever string has already matched the expression in parens.

Back-References

In general, `\n` refers to the string that has already matched the mth paren pair, which has to occur before the back-reference.

Also note that one actually has to understand the matching mechanism in greater detail (greedy versus lazy).

Usually the longest match possible is chosen.

For example, the expression

```
(((a|b)*)c\2)*
```

matches all words in

$$\{\, xcx \mid x \in \{a,b\}^\star \,\}^\star$$

Primality

Here is a beautiful way to check primality, albeit in unary.

The file prime.txt contains strings of a's up to length 25, one on each line.

```
> egrep -ve '^a$|^(aa+)\1\1*$' prime.txt
aa
aaa
aaaaa
aaaaaaa
aaaaaaaaaaa
aaaaaaaaaaaaa
aaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaa
```

It is fast up to 23, then slows down miraculously. I have no idea why.

Warning

Careful, though, the fancy stuff in regular expression matchers sometimes does not work at all.

The expression (.?) means: match at most one single character and remember it. So the following is supposed to match all palindromes of even length up to 8:

```
egrep -e '(.?)(.?)(.?)(.?)\4\3\2\1'
```

It does, but it also accepts $aaaabbbb$ and crashes on odd length strings (this is GNU grep-2.5.2).