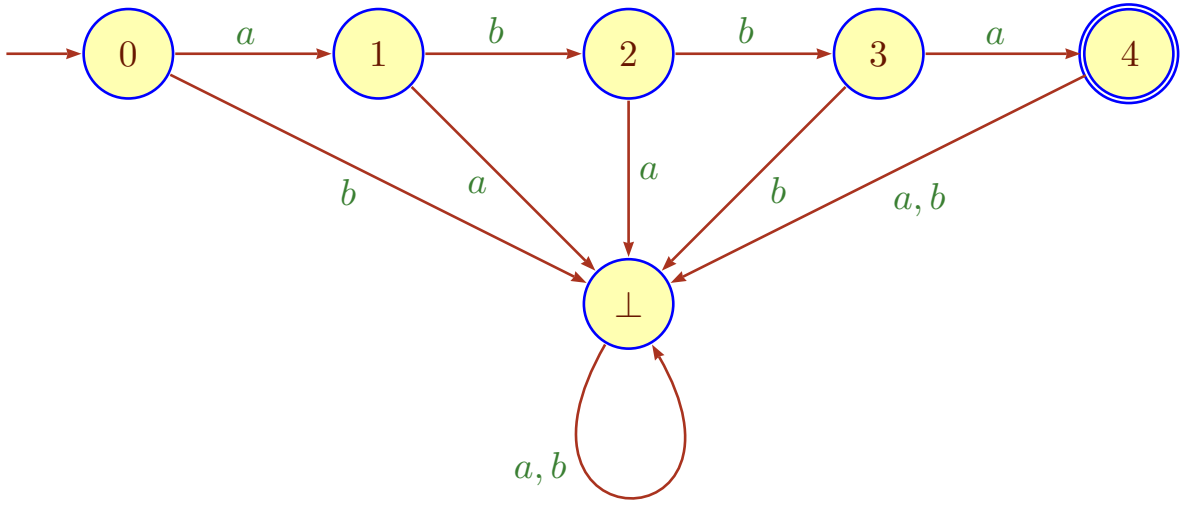
CDM

Algebra of Regular Languages

Klaus Sutner

Carnegie Mellon University

Given a word w, it is trivial to construct a DFA M_w on $|w| + 2$ states such that $\mathcal{L}(M_w) = \{w\}$. For example, for $w = abba$ we get



State 0 is initial, and 4 is final. $\perp$ is a sink state and can be eliminated if we allow partial DFAs.

Exercise

Show that $|w| + 2$ is indeed the state complexity of $\{w\}$.

It follows from our closure properties that every finite language is also regular: we can build a DFA M for any finite set of words

$$\mathcal{L}(M) = \{w_1, w_2, \ldots, w_s\}.$$

by forming the product of the M_{w_i}

Alas, this does not really work: the size of this product machine grows exponentially.

But, there are several efficient algorithms to build machines for finite sets of words. In fact, there is a whole industry of such algorithms. Bear in mind: blind application of powerful methods sometimes leads to disaster.

How can we use finite state machines to compute functions? The general situation is a bit complicated, but if we restrict ourselves to functions with finite codomain things are relatively straightforward.

Suppose we have a function $f : \Sigma^\star \to \Delta$ where Δ is a finite set (which we will think of as an alphabet).

How could a DFA compute f? If $\Delta = \mathbf{2} = \{0, 1\}$ there is no problem: we can think of the machine as returning output 1 on input $x \in \Sigma^\star$ whenever $\delta(q_0, x) \in F$, and 0 otherwise.

This may not seem too interesting, but consider the following example. There is a famous infinite binary word $T = (t_n)$ defined by

$$\begin{aligned} t_0 &= 0 \\ t_{2n} &= t_n \\ t_{2n+1} &= \overline{t_n} \end{aligned}$$

Here $\overline{x}$ denotes bitwise complement.

For example, the first 64 bits of T are

$$0110100110010110100101100110100110010110011010010110100110010110$$

T has many interesting properties, perhaps the most important one being the fact that it is cube-free: it is impossible to write

$$T = \ldots x\, x\, x \ldots$$

for any non-empty finite word x. In fact, one cannot even get

$$T = \ldots x\, x\, x_1 \ldots$$

Yet, there is a DFA that computes T in the sense that on input n, written in binary, the DFA outputs t_n.

As already mentioned, one of the killer apps for automata is their use in solving decision problems in logic, e.g. in program verification.

To this end we have to construct machines for various languages, quite often over large alphabets of the form

$$\Gamma = \Sigma \times \Sigma \times \ldots \times \Sigma$$

This turns out to be the right environment for checking validity of logical formulae over certain structures.

Example: Third Symbol from the End

Consider the input $x = baaba$. Here are the possible traces of M from above with this input (for emphasis we write the transitions with arrows). The last one leads from the initial state to the final state, so the machine accepts x.

$$
\begin{array}{ccccccccccc}
3 & \xrightarrow{b} & 3 & \xrightarrow{a} & 3 & \xrightarrow{a} & 3 & \xrightarrow{b} & 3 & \xrightarrow{a} & 3 \\
3 & \xrightarrow{b} & 3 & \xrightarrow{a} & 3 & \xrightarrow{a} & 3 & \xrightarrow{b} & 3 & \xrightarrow{a} & 2 \\
3 & \xrightarrow{b} & 3 & \xrightarrow{a} & 3 & \xrightarrow{a} & 2 & \xrightarrow{b} & 1 & \xrightarrow{a} & 0
\end{array}
$$

But $x = babaa$ is not accepted, none of runs has the right source and target.

$$
\begin{array}{ccccccccccc}
3 & \xrightarrow{b} & 3 & \xrightarrow{a} & 3 & \xrightarrow{b} & 3 & \xrightarrow{a} & 3 & \xrightarrow{a} & 3 \\
3 & \xrightarrow{b} & 3 & \xrightarrow{a} & 3 & \xrightarrow{b} & 3 & \xrightarrow{a} & 3 & \xrightarrow{a} & 2 \\
3 & \xrightarrow{b} & 3 & \xrightarrow{a} & 3 & \xrightarrow{b} & 3 & \xrightarrow{a} & 2 & \xrightarrow{a} & 1
\end{array}
$$

Challenge: can we construct this DFA by hand, without any automata conversions?

Note that we need to remember the last 3 symbols of the input: if we've reached the end we have enough information to decide whether we should accept.

So we use $Q = \{a, b\}^3$ and transitions

$$xyz \xrightarrow{s} yzs$$

Final states are $\{aaa, aab, aba, abb\}$.

Initial state is bbb (a clever hack, otherwise we would have to add some more states ε, a, b, aa, ab, ...) and cone them on top of the strongly connected part.

From now on, we will focus on one particular Kleene algebra, the language semiring

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

As defined, this is an uncountable structure, but we will be mostly interested in the case where the carrier set is just the regular languages over Σ.

In order to emphasize the algebraic angle, we will often write $+$ instead of $\cup$, 1 instead of $\{\varepsilon\}$, and so on.

Note that, strictly speaking, a word w is not an element of $\mathcal{L}(\Sigma)$. But the singleton $\{w\}$ is, and so it makes sense to be sloppy with notation and identify the two.

If that sends shivers up and down your type-theoretic spine note that we can filter out singletons using only algebra.

In any Kleene algebra, define x to be an atom if $x \neq 0$ but $y \leq x$ implies $y = 0$ or $y = x$.

For example, 1 is an atom.

In the language semiring, atoms are exactly the singletons.

How about the missing operations, subtraction and division?

For subtraction we would need an additive cancellation monoid: $x + y = x + z$ implies $y = z$. This is hopelessly false in our setting: $x + x = x = x + 0$.

So how about some operation resembling division? Since our multiplication is not commutative, let's focus on left division for the time being. Here is a plausible approach.

Definition

Let $L \subseteq \Sigma^\star$ be a language and $x \in \Sigma^\star$. The left quotient of L by x is

$$x^{-1} L = \{ y \in \Sigma^\star \mid xy \in L \}.$$

So we are simply removing a prefix x from all words in the language that start with this prefix. If there is no such prefix we get an empty quotient.

It is standard to write left quotients as

$$x^{-1}\, L$$

Here is the bad news: left quotients are actually a right action of $\Sigma^\star$ on $\mathcal{L}(\Sigma)$.

As a consequence, the first law of left quotients below looks backward at first sight.

We could fix the problem by writing something like L/x but that's awkward since it seems to suggest that we are removing a suffix.

Lemma

Let $a \in \Sigma$, $x, y \in \Sigma^\star$ and $L, K \subseteq \Sigma^\star$. Then the following hold:

- $(xy)^{-1}L = y^{-1}x^{-1}L$,
- $x^{-1}(L \odot K) = x^{-1}L \odot x^{-1}K$ *where* $\odot$ *is one of* $\cup$, $\cap$ *or* $-$,
- $a^{-1}(LK) = (a^{-1}L)K + \chi_L\, a^{-1}K$,
- $a^{-1}L^\star = (a^{-1}L)\, L^\star$.

Here we have used the abbreviation χ_L to simplify notation:

$$\chi_L = \begin{cases} 1 & \text{if } \varepsilon \in L, \\ 0 & \text{otherwise.} \end{cases}$$

So χ_L is either zero or one in the language semiring and simulates an if-then-else.

Note that $(xy)^{-1}L = y^{-1}x^{-1}L$ and NOT $x^{-1}y^{-1}L$. As already mentioned, the problem is that algebraically left quotients are a right action.

Quotients coexist peacefully with Boolean operations, we can just push the quotients inside.

But for concatenation and Kleene star things are a bit more involved; the lemma makes no claims about the general case where we divide by a word rather than a single letter.

Exercise

Prove the last lemma.

Exercise

Generalize the rules for concatenation and Kleene star to words.

The ultimate reason we are interested in quotients is that they provide an elegant tool to construct the minimal automaton for a regular language. And the associated algorithms can be made very efficient.

For the time being, though, let us focus on the algebra. We write $\mathcal{Q}(L)$ for the set of all quotients of a language L.

How would we go about computing $\mathcal{Q}(L)$?

In general this will be difficult, but for languages described in terms of Kleene's operations we can use algebra (there is a little glitch, though).

Fix some language L once and for all.

Definition

A k-subfactorization (of L) is a k-tuple of languages X_i, $1 \leq i \leq k$, such that

$$X_1 X_2 \dots X_k \subseteq L$$

For emphasis, we write $X_1 : X_2 : \dots : X_k$ for a subfactorization.
A k-factorization (of L) is a k-subfactorization where every term is maximal: adding another word to X_i breaks the inclusion, for all $i \in [k]$.

Note that $\dots : X : Y : \dots$ is a subfactorization iff $\dots : XY : \dots$ is a subfactorization. Alas, the corresponding result for factorizations is wrong, in either direction.

We are mostly interested in the case $k = 2, 3$.

A subfactorization $X_1{:}X_2{:}\ldots{:}X_k$ dominates as subfactorization $Y_1{:}Y_2{:}\ldots{:}Y_k$ if $Y_i \subseteq X_i$ for all i. So a factorization is a subfactorization that is not dominated by another.

A factor is a term that appears in some place in some factorization. A left/right factor is one that appears at the left/right end of a factorization.

Question: For a regular language L, what can we say about its factors?

There is a surprisingly detailed answer, but we need to build up a few tools first.

Claim 1: Every subfactorization can be extended to a factorization. Maximal terms are preserved in the process.

Proof. For simplicity consider a 2-factorization $X{:}Y$. We can saturate, say, the left term via $X' = \bigcup\{\, Z \mid ZY \subseteq L \,\}$. Then $X'{:}Y$ is still a subfactorization, dominates $X{:}Y$, and X' is maximal. Repeat for Y. □

But note that the process does not commute: the final result depends on the saturation order. For example, for most L, we could extend $0{:}0$ to either $\Sigma^\star{:}0$ or $0{:}\Sigma^\star$. In fact, there are many other ways we can saturate the components by adding words in some fairly arbitrary manner.

Exercise

Figure out a general algorithm to extend $0{:}0$ to all possible factorizations.

Claim 2: There is a one-one correspondence between all left factors and all right factors.

Proof. Suppose X is a left factor and let $X{:}Y$ be a corresponding factorization. Since Y is maximal there can be no other right factor matching X. The same argument works in the opposite direction, done. □

We write $\rho(X)$ for the right factor corresponding to left factor X, and $\lambda(Y)$ for the inverse function. For example, for $L = a^\star \subseteq (a+b)^\star$ we have $\rho(0) = \Sigma^\star$, $\rho(a^\star) = a^\star$ and $\rho(\Sigma^\star) = 0$.

Now saturate the middle term in $X{:}0{:}Y$, X and Y left/right factors, and write $Z = Z(X, Y)$ for the result. From Claim 1 we have that all factors occur as one of these $Z(X, Y)$.

Claim 3: The language L itself is a right factor $\rho(X')$ as well as a left factor $\lambda(Y')$. Moreover, all left factors are of the form $Z(X', Y)$, and all right factors are of the form $Z(X, Y')$. Lastly, $Z(X', Y') = L$.

Proof.

$1{:}L$ is a subfactorization and uniquely saturates to $X'{:}L$, so that $X' = \lambda(L)$. By symmetry, $Y' = \rho(L)$.

By our choice of X', $X'{:}\lambda(Y){:}Y$ is a subfactorization and even a factorization (check). The claim about enumerating left factors follows; right factors are analogous. Lastly, $X'{:}L{:}Y'$ is a factorization, and we are done. □

Theorem

The number of factors of L is finite iff L is regular. Moreover, the number of left/right factors is $\Delta(\overline{L})$ in this case.

Proof. To see why, let $X{:}Y$ be a factorization of L. We have $Y = \bigcap_{w \in X} w^{-1}L$. To saturate the left term we choose X maximal so as to maintain the intersection. More precisely, by complementing we get

$$\overline{Y} = \bigcup_{w \in X} w^{-1}\overline{L} = X^{-1}\overline{L}$$

But L is regular iff the number of quotients (word or language) is finite.

□

Careful, though: in general $\Delta(L) \neq \Delta(\overline{L})$, unlike with δ. It follows that there are at most $2^{\delta(L)}$ many left/right pairs.

Suppose $\mathcal{A}$ is the minimal DFA for L. As just mentioned, we need to determine all intersections

$$Y = \bigcap_{p \in P} [\![p]\!]$$

where $P \subseteq Q$. Let's call P critical if P produces Y in this manner, and P is maximal such. Note that P must actually be maximum (just take unions).

Given P critical for right factor Y we obtain the corresponding left factor $\lambda(Y)$ as

$$X = \mathcal{L}(\mathcal{A}(q_0, P))$$

Hence we can construct a list

$$X_1{:}Y_1, X_2{:}Y_2, \ldots, X_m{:}Y_m$$

of all left/right pairs where $Y_i = \rho(X_i)$.

Suppose $U, V \subseteq Q$ are critical, and let X be the left factor for U, and Y the right factor for V. To determine $Z = Z(X, Y)$ note that

$$\begin{aligned} w \notin Z &\Leftrightarrow \exists\, x \in X, y \in Y\, (xwy \notin L) \\ &\Leftrightarrow \exists\, q \in U, y \in Y\, (q \cdot wy \notin F) \\ &\Leftrightarrow \exists\, q \in U\, (Y \not\subseteq [\![q \cdot w]\!]) \\ &\Leftrightarrow \exists\, q \in U\, (q \cdot w \notin V) \end{aligned}$$

But then $\overline{Z}$ is the language of $\mathcal{A}(U, \overline{V})$.

Together with the list of left/right pairs we now have a coordinate system and can organize the collection of all factors into a $m \times m$ matrix $\mathfrak{F}$ with entries $Z_{ij} = Z(X_i, Y_j)$.

The star-free language $L = a^\star b^\star c^\star$ has 5 left/right factors:

left	$\Sigma^\star$	L	$a^\star b^\star$	$a^\star$	0
right	0	$c^\star$	$b^\star c^\star$	L	$\Sigma^\star$

Factor matrix $\mathfrak{F} = (Z_{ij})$:

	0	$c^\star$	$b^\star c^\star$	L	$\Sigma^\star$
$\Sigma^\star$	$\Sigma^\star$	0	0	0	0
L	$\Sigma^\star$	$c^\star$	0	0	0
$a^\star b^\star$	$\Sigma^\star$	$b^\star c^\star$	$b^\star$	0	0
$a^\star$	$\Sigma^\star$	L	$a^\star b^\star$	$a^\star$	0
0	$\Sigma^\star$	$\Sigma^\star$	$\Sigma^\star$	$\Sigma^\star$	$\Sigma^\star$

Theorem

Consider the $m \times m$ factor matrix $\mathfrak{F} = (Z_{ij})$. Then

- $Z_{ij}Z_{jk} \leq Z_{ik}$
- $X_1X_2 \ldots X_s \leq L$ *iff* $X_j \leq Z_{i_{j-1}i_j}$ *for some* $i_j \in [m]$, $j \in [s]$, *where* $i_0 = \lambda(L)$ *and* $i_s = \rho(L)$.

Proof. By definition, $X_iZ_{ij}Y_j \leq L$, so that $X_iZ_{ij} \leq X_j$. Hence $X_iZ_{ij}Z_{jk}Y_k \leq X_jZ_{jk}Y_k \leq L$, and our claim follows.

It suffices to prove the binary case: $XY \leq Z_{ik}$ iff there is some j such that $X \leq Z_{ij}$ and $Y \leq Z_{jk}$.

To see this, note that $(X_iX)(YY_k) \leq L$, so that $X_iX \leq X_j$ and $YY_j \leq Y_j$ for some j. But then $X_iXY_j \leq L$ and $X_JYY_k \leq L$, and the claim follows.

This may seem obvious, but we now have a proof that the factors of a factor are again factors of the original language. OK, a bit anticlimactic ...

Recall that $\mathfrak{F}$ itself lives in another Kleene algebra and thus has a star. We have $\mathfrak{F}^\star = \mathfrak{F}$.

Exercise

Extract this information from the last theorem.

Back to our original complaint: the lack of invariance under string reversal.

Theorem (Conway)

Let L be a regular language. Then $\Delta(L) = \Delta(L^{\mathrm{op}})$.

Proof.

Consider all 2-factorizations $X{:}Y$ of $\overline{L}$.

As we have just seen, there are $\Delta(L)$ choices for X.

By symmetry, there are $\Delta(L^{\mathrm{op}})$ choices for Y.

But we already know that these two numbers agree.

□

From the definition of a Turing machine, the read-only input tape can be scanned repeatedly and the tape head may move back and forth over it.

As it turns out, one can assume without loss of generality that the read head only moves from left to right only: at each step one symbol is scanned and then the head moves right and never returns.

Theorem (Rabin/Scott, Shepherdson)

Every decision problem solved by a constant space two-way machine can already be solved by a constant space one-way machine.

The original proof of this result is quite messy, see Rabin/Scott 59. Here is a sketch.

Right moves of the 2-DFA are easily simulated by the new DFA, so consider a left move. Say, the current configuration is xpa and $\delta(p, a) = (p, \mathsf{L})$. Then the machine enters the block x and we need to keep track of the state q it is in when it leaves the block to the right. Of course, this may never happen: the machine may have fallen off the left end of x, or may be stuck in an infinite loop. To deal with this issue, we augment Q by an additional state $\perp$. We also abuse $\perp$ to keep track of the state of the actual computation of the 2-DFA.

More formally, we define state vectors $V_x : Q_\perp \to Q_\perp$ for all non-empty words x as follows:

$$V_x(p) = \begin{cases} q & \text{if } w = ua,\ upa \vdash uaq,\ p \in Q \\ q & \text{if } w = ua,\ q_0 w \vdash wq,\ p = \perp \\ \perp & \text{otherwise.} \end{cases}$$

It is not hard to check that these vectors have the property that $V_x = V_y$ implies $V_{xa} = V_{ya}$. Hence, they can be used to define a right semigroup action, giving rise to a one-way DFA. Let $\mathcal{V}$ be the set of all state vectors and add an extra initial state $\top$. Define a transition function γ on $\mathcal{V}_\top$ by

$$\gamma(\top, a) = V_a$$
$$\gamma(V_x, a) = V_{xa}$$

and final states by

$$F' = \{\, V_x \mid V_x(\bot) \in F \,\}$$

If necessary, we can adjust to deal with ε.

Consider the finite languages

$$L_n = \{\, \#ab^{e_1}ab^{e_2}a \ldots ab^{e_n}c^k b^{e_k}\# \mid e_i, k \in [n] \,\}$$

There is a linear size 2-dfa for L_n, but every 1-dfa has exponentially many states.

Consider a regular language $L \subseteq \Sigma^+$ and define

$$\mathsf{root}(L) = \{\, x \in \Sigma^+ \mid x^+ \cap L \neq \emptyset \,\}$$

We want to show that $\mathsf{root}(L)$ is again regular. This can be done using a monoid automaton, but a very simple argument can be based on 2-DFAs.

Using endmarkers, we can build a 2-DFA that scans x and checks if it lies in L; if so, it accepts. Otherwise, it stores the current state q of the DFA for L, and rescans x, this time starting in state q. Rinse and repeat. If x is in $\mathsf{root}(L)$, this will lead to acceptance; otherwise, the 2-DFA is stuck in an infinite loop.

If you find this offensive, keep track of all states q seen so far at the end of a scan, and reject whenever a duplicate appears.

Let $\mathcal{A}$ be a DFA. We can define an equivalent DFA $\mathcal{A}^{\text{sgr}}$ whose state set is the monoid of maps $Q \to Q$. The right action, initial and final states are given by

$$\begin{aligned} f \cdot a &= f \circ \delta_a \\ q_0 &= I \\ F' &= \{\, f \mid f(q_0) \in F \,\} \end{aligned}$$

This machine is useful for conceptual purposes such as establishing a link between finite state machines and monoids, but is obviously problematic from an algorithmic perspective (even if we restrict the state set to the monoid generated by the δ_a.

As and example, consider $L \subseteq \Sigma^+$ and define

$$\mathsf{root}(L) = \{\, x \in \Sigma^+ \mid x^+ \cap L \neq \emptyset \,\}$$

Proposition

$\mathsf{root}(L)$ *is regular whenever* L *is so regular.*

Proof.

Change the final states in the monoid automaton for L to be

$$F = \{\, f : Q \to Q \mid \mathsf{orb}^+(q_0; f) \cap F_{\mathcal{A}} \neq \emptyset \,\}$$

The new automaton accepts x iff there is some $k \geq 1$ such that $x^k \in L$, as required. □

Figure out what the darn states are.

There are many ways one can associate a matrix with a finite state machine. Say, we are dealing with a DFA $\mathcal{A}$.

Probably the most obvious approach is to consider the transition function of $\mathcal{A}$ as a matrix $\Sigma^{Q \times \Sigma}$. This is useful from a data structure perspective, but not particularly interesting.

Much more useful is to consider square matrices $M^{Q \times Q}$ that live in some monoid, with matrix multiplication as the operation. In particular we can associate every input symbol to a Boolean matrix

$$\varphi : \Sigma \to \mathbf{2}^{Q \times Q}$$

giving rise to a monoid homomorphism. We have

$$x \in L \iff I \cdot \varphi(x) \cdot F = 1$$

where I and F are Boolean vectors indicating the initial and final states.

Suppose we have some set X and a collection F of endofunctions on X.

Definition

(X, F) is a transformation semigroup or composition semigroup if F is closed under composition, and a transformation monoid if, in addition, F contains a unit element.

If you prefer, you can think of the semigroup F as acting on X on the left in the natural way:

$$f \cdot x = f(x)$$

This is for standard composition; if we use diagrammatic composition (which is more natural in connection with finite state machines), we get a right action.

To see the connection to finite state machines, note that we can think of the transition function of a DFA as a Σ-indexed list of functions from states to states:

$$\delta_a : Q \to Q$$
$$\delta_a(p) = \delta(p, a)$$

This turns the DFA into a Σ-algebra

$$\mathcal{A} = \langle Q; \delta_{a_1}, \ldots, \delta_{a_k} \rangle$$

This may seem like a pointless exercise, but it naturally leads to another interesting perspective: algebra.

But Why?

First off, nothing is lost: we can "iterate" these functions according to some input word $u = u_1 u_2 \dots u_n$ (diagrammatic composition):

$$\delta_u = \delta_{u_1} \circ \delta_{u_2} \circ \dots \delta_{u_{n-1}} \circ \delta_{u_n}$$

Acceptance then translates into: $\mathcal{A}$ accepts a word u iff $\delta_u(q_0) \in F$.

Plus, we get some additional concepts more or less for free: a subautomaton of $\mathcal{A}$ is another Σ-algebra $\mathcal{B} = \langle P; \gamma_{a_1}, \dots, \gamma_{a_k} \rangle$ such that $P \subseteq Q$ and $\gamma_a(p) = \delta_a(p)$.

Similarly, a morphism $\varphi : \mathcal{A} \to \mathcal{B}$ of Σ-algebras must be a map $\varphi : Q \to P$ such that

$$\varphi(\delta_a(p)) = \gamma_a(\varphi(p))$$

One may want to augment this by conditions about initial and final states.

It is also straightforward to define products of the form

$$\mathcal{A} \times \mathcal{B}$$

And we get congruences: an equivalence relation E on Q is a congruence if

$$p \; E \; q \;\; \text{implies} \;\; \delta_a(p) \; E \; \delta_a(q)$$

To be sure, all these concepts can be developed without any appeal to algebra, given enough thought.

But the whole point here is that they pop up for free, courtesy of some general, universal ideas.

Exercise

Figure out exactly what morphism, product and congruence mean in this context.

The functions δ_a, $a \in \Sigma$, generate a transformation semigroup (monoid) T over Q, a subsemigroup of the full monoid of endofunctions $Q \to Q$.

Definition

T is called the transformation semigroup (monoid) of the DFA.

One way of writing down T is

$$T = \{ \delta_x : Q \to Q \mid x \in \Sigma^\star \} = \langle \delta_a \mid a \in \Sigma \rangle$$

where $\delta_x(p) = \delta(p, x)$.

Analyzing this semigroup can help quite a bit in getting a better understanding of a DFA. And, there are powerful algebraic tools available that help in dealing with the monoid.

Example 1: Even-Even

There is a natural 4-state DFA that accepts all strings over $\{a, b\}^\star$ that contain an even number of a's and an even number of b's.

p	1	2	3	4
$\delta_a(p)$	2	1	4	3
$\delta_b(p)$	3	4	1	2

The initial state is 1 and $F = \{1\}$.

But note that

$$\delta_a \circ \delta_a = I$$
$$\delta_b \circ \delta_b = I$$
$$\delta_a \circ \delta_b = \delta_b \circ \delta_a$$

so that the transformation semigroup consists of $\{I, \delta_a, \delta_b, \delta_a \circ \delta_b\}$. Note that this is actually a monoid and even a group (Kleinsche Vierergruppe).

Moreover, from the equations it is easy to see that for any word x

$$\delta_x = I \iff \#_a x \text{ even}, \#_b x \text{ even}$$

Similarly we have

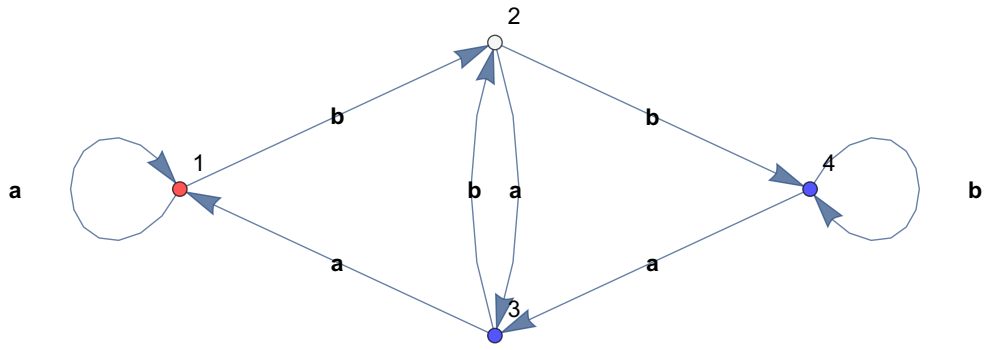
$$\delta_x = \delta_a \iff \#_a x \text{ odd}, \#_b x \text{ even}$$

and so on.

At the very least this very elegant and concise.

Example 2: $L_{a,-2}$

The usual de Bruijn automaton



yields the transformations

p	1	2	3	4
$\delta_a(p)$	1	3	1	3
$\delta_b(p)$	2	4	2	4

These generate the semigroup (no monoid here)

$$(1,3,1,3),(2,4,2,4),(1,1,1,1),(2,2,2,2),(3,3,3,3),(4,4,4,4)$$

Question: What do the constant functions mean?

The distinction between semigroups and monoids here is a bit of a technical nuisance, but there is no easy way to get rid of it.

At any rate, note that we can turn any semigroup $\mathcal{S}$ into a monoid $\mathcal{S}^1$ by simply adding a new element 1 and defining

$$x \cdot 1 = 1 \cdot x = x$$

for all x in $\mathcal{S}$.

Clearly, $\mathcal{S}$ and $\mathcal{S}^1$ are essentially the same.

Also note that a transformation semigroup may be a monoid without containing the identity function.

The reason monoids are important here is because they provide a characterization of regular languages that is free of any combinatorial aspect. Always remember: algebra is the great simplifier.

Theorem

A language $L \subseteq \Sigma^\star$ is regular iff there is a finite monoid M, $M_0 \subseteq M$ and a monoid homomorphism $f : \Sigma^\star \to M$ such that $L = f^{-1}(M_0)$.

Proof.

If L is regular, let M be the transformation monoid of a DFA that recognizes L, and define $f(x) = \delta_x$ and $M_0 = \{\, g \in M \mid g(q_0) \in F \,\}$.

The opposite direction is more interesting: we construct a DFA

$$\mathcal{A} = \langle M, \Sigma, \delta; 1_M, M_0 \rangle$$

where $\delta(p, a) = p \cdot f(a)$. Then $\delta(p, x) = p \cdot f(x)$ and $\delta(q_0, x) = f(x)$. □

Message: anything goes as a state set, as long as the set is finite. For the implementer, the state set is always $[n]$, but that's not a good way to think about it.

Algebraic automata theory is a fascinating subject with lots of elegant results, but it requires work and there is no essential algorithmic payoff. So, we won't go there.