Automata

Introduction

The following are examples of how one normally writes positive rational numbers:

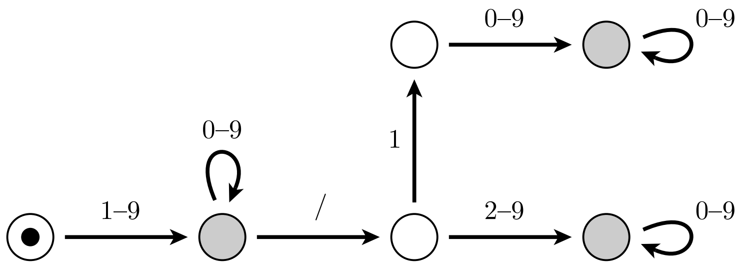
$$13, \quad 2/4, \quad 20/47, \quad 1/24 \,.$$

These are not:

$$13/1, \quad 02/4, \quad 0/0 \,.$$

Note that we are not restricting ourselves to reduced fractions. We can thus put together a recipe for expressing a positive rational number: (1) To write a **positive integer** we write a sequence of non-negative digits, starting with a positive one. (2) To write a **positive rational number**, write a positive integer, followed optionally by a slash / followed by a positive integer other than 1.

This recipe is not trivial to read, and for most people it is easier to understand if we encode it in a diagram:



The diagram needs some explaining, but even so it's probably easier to follow than the text version. The main point is that *expressions for positive rational numbers correspond to certain paths in the diagram* above. Which paths? Those continuous paths which start at the dotted circle and end at a grey circle. I call such paths **conforming**. The relationship between the path and the expression is that each arrow followed in the diagram corresponds to a character in the expression, and the character is to be chosen from the set of characters located close to the arrow. For example, an expression must begin with a non-zero digit, so the first transition arrow is labeled 1–9, but after that any sequence of digits is acceptable to make up a positive integer, so the loop on the second node is labelled 0-9. The third circle is not grey, for example, because a fractional expression cannot terminate with /. The one above that is not grey because 1 is not an acceptable denominator.

As this example suggests, automata are useful in programming reading and verification of input. We shall also see that in mathematics they can often describe patterns which are otherwise difficult to handle.

An example from mathematics

The group $\Gamma = PSL_2(\mathbb{Z}) = SL_2(\mathbb{Z})/\{\pm 1\}$ acts discretely on the upper half plane

$$\mathcal{H} = \{z = x + iy \mid y > 0\}$$

by linear fractional transformations:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} : \; z \longmapsto \frac{az+b}{cz+d} \,.$$

It has been known since the early nineteenth century that the region

$$\mathcal{D} = \{|x| \le 1/2, |z| > 1\}$$

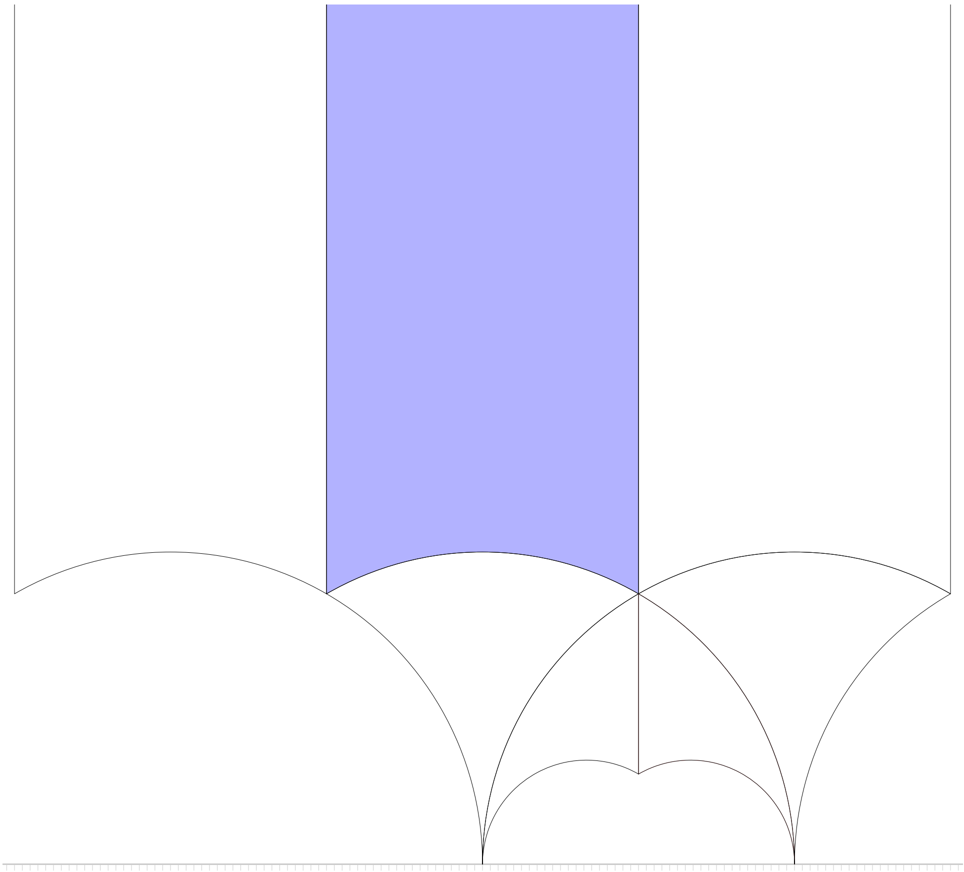
is a fundamental domain for Γ. We have in fact the following classical algorithm to find for any z in $\mathcal{H}$ a point in $\mathcal{D}$ equivalent to it.

(1) Choose the integer n so that $-1/2 < z - n \leq 1/2$. Set $z := z - n$.
(2) If $|z| < 1$ set $z := -1/z$.
(3) Loop until $-1/2 < z \leq 1/2$ and $|z| \geq 1$.

The domains neighbouring $\mathcal{D}$ (those with an edge in common) are obtained by applying

$$T = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, \quad T^{-1} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}, \quad S = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

to $\mathcal{D}$.



A result stated most completely by MacBeath (Annals of Mathematics, 1963) then guarantees that Γ is generated by S, T, and $U = T^{-1}$. The domains adjacent to a vertex of $\mathcal{D}$ are those obtained from $\mathcal{D}$ by applying S, ST, STS, $STST$, $STSTS$, and $STSTST = 1$. MacBeath's result implies then that Γ is generated by S, T, U with relations $S^2 = 1$, $TU = 1$, $STSTST = 1$.

Note that the classical algorithm for reduction of z provides a proof of generation also, since its two operations amount to replacing z by $T^{-n}z$ and Sz respectively. In fact, we can find an expression for any γ in Γ if we apply this algorithm to $\gamma(z)$ for any value of z in the interior of $\mathcal{D}$. A modification of the algorithm for the case $z = \infty$ is the classical continued fraction algorithm for a/c, and also allows us to deduce an expression using only integer arithmetic.

We adopt the following scheme for writing every element of Γ uniquely as a string $s(\gamma)$ in the symbols S, T, U.

(1) *The string $s(\gamma)$ has the shortest possible length among all strings representing γ.*

(2) *The string $s(\gamma)$ is least in dictionary order among all strings of minimal length, where we (more or less arbitrarily) adopt the convention that dictionary order of our alphabet is T, U, S—and we read the strings backwards.*

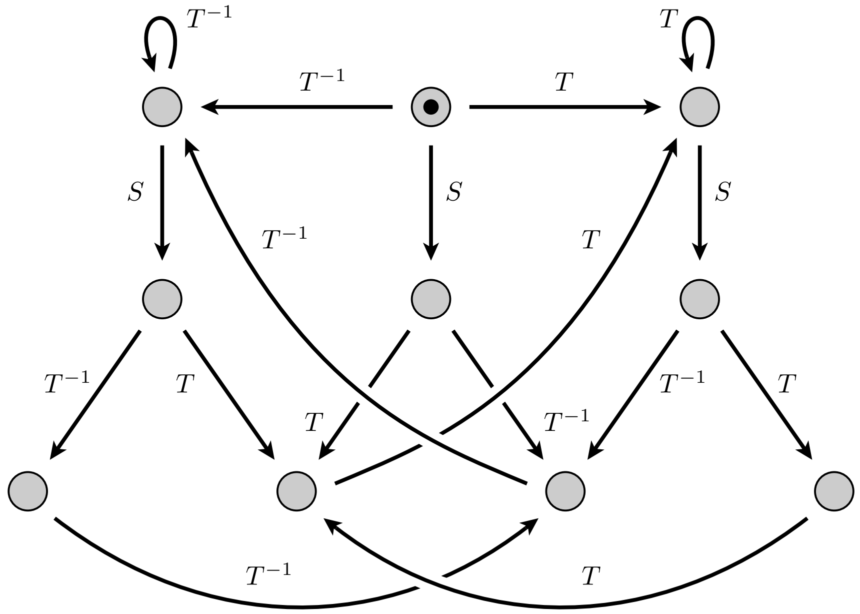
Thus we write the element $ST^{-1}S$ can also be expressed as TST, but we represent it by the second expression instead of the first since in our dictionary it comes earlier.

Here, for example, are all the strings representing elements of Γ of length at most 4, listed in 'alphabetic' order:

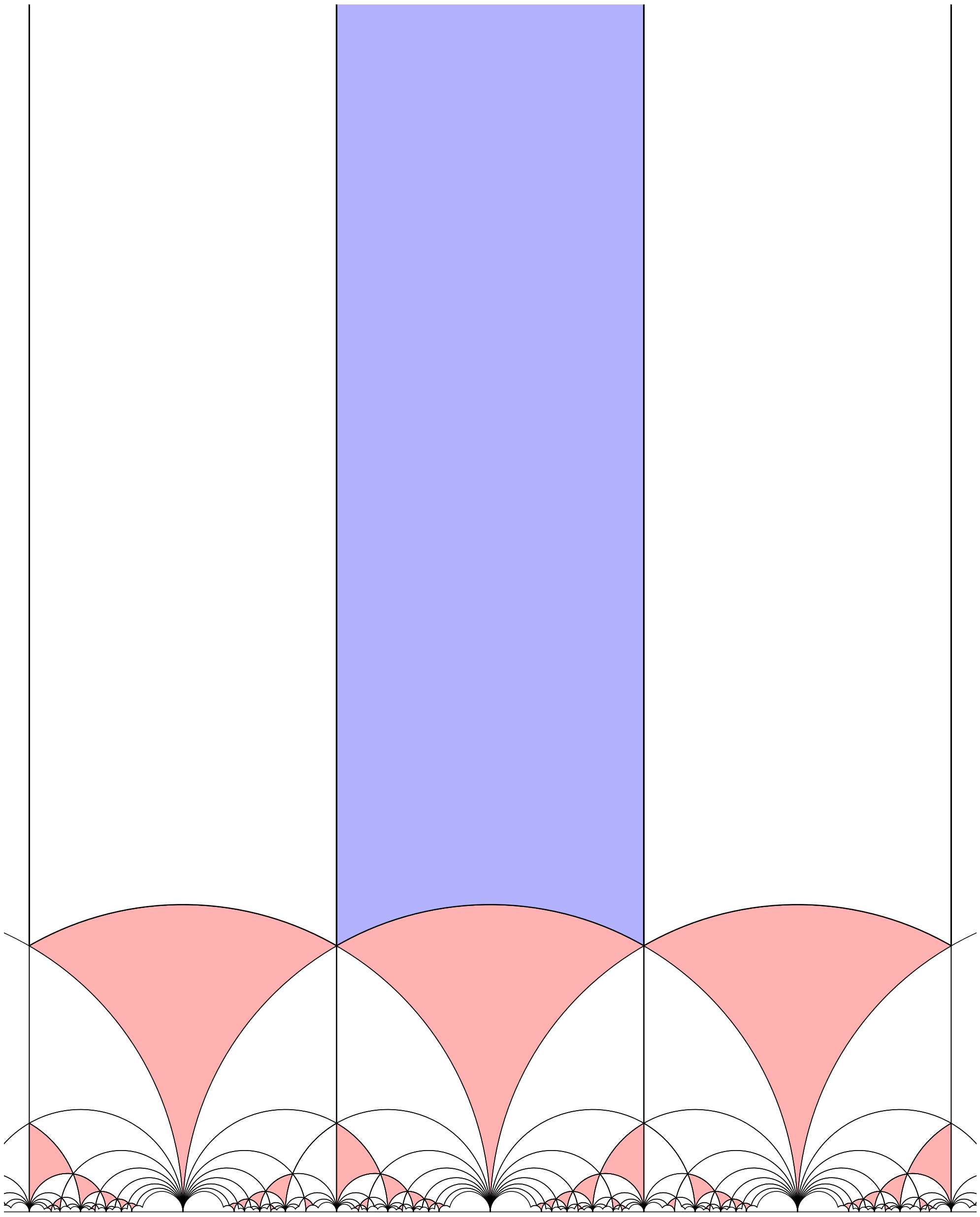
T
TT
TTT
$TTTT$
$TTTS$
TTS
$TTST$
$TTST^{-1}$
TS
TST
$TSTT$
TST^{-1}
$TST^{-1}T^{-1}$
T^{-1}
$T^{-1}T^{-1}$
$T^{-1}T^{-1}T^{-1}$
$T^{-1}T^{-1}T^{-1}T^{-1}$
$T^{-1}T^{-1}T^{-1}S$
$T^{-1}T^{-1}S$
$T^{-1}T^{-1}ST$
$T^{-1}T^{-1}ST^{-1}$
$T^{-1}S$
$T^{-1}ST$
$T^{-1}STT$
$T^{-1}ST^{-1}$
$T^{-1}ST^{-1}T^{-1}$
S
ST
STT
$STTT$
$STTS$
ST^{-1}
$ST^{-1}T^{-1}$
$ST^{-1}T^{-1}T^{-1}$
$ST^{-1}T^{-1}S$

We now have the remarkable fact:

Theorem. *The set of all strings generated according to those rules are the same as the ones corresponding to conforming paths in the following diagram.*



This regular structure underlies a number of results about the geometry of the upper half-plane, and of course it is also useful for producing pictures like this one:



What I want to do in the rest of this section is show how the structure of the automaton is mirrored in the patterns one can see, if one looks closely, in this figure.

The first pattern to notice is connected with the right cosets of Γ with respect to the group $N_{\mathbb{Z}}$ of unipotent elements in Γ generated by T and T^{-1}. One such coset is made up of all the elements 1, T, T^{-1}, TT, $T^{-1}T^{-1}$, $\ldots$ (i.e. the subgroup $N_{\mathbb{Z}}$ itself) and another is made up of S, ST, ST^{-1}, STT, $ST^{-1}T^{-1}$, $\ldots$ (which is $SN_{\mathbb{Z}}$).

Theorem. *Every right coset of $N_{\mathbb{Z}}$ possesses a unique element w whose expression $s(w)$ terminates with S. The other elements in the same right coset are those with expressions $s(w)TT\ldots$ and $s(w)T^{-1}T^{-1}\ldots$.*

The proof of this claim is direct from the definition of the strings in our language.

We can in fact see this grouping in the picture—I have outlined the cosets in a separate colour. Here is a picture of the cosets themselves:

Deterministic automata

A diagram such as that above provides an example of an **automaton**. There are several possible definitions of an automaton, more or less equivalent. I shall start with the simplest.

An **deterministic automaton** or DA is defined by

(1) a set of **states**
(2) a finite set of **input symbols**
(3) a single state designated as the **start state**
(4) a subset of states designated as **accepting**
(5) a set of **transitions** from one state to another, labelled by input symbols.

The transitions are subject to the condition that from each state there is no more than one transition labelled by a given symbol. The transitions are therefore determined by the triple (s, σ, t) where s is the **source** of the transition, σ its symbol, t its target. The transitions therefore make up edges in a labelled graph whose nodes are the states of the automaton.

Automata are important because of their relation with languages or sets of words. A **conforming path** in an automaton, usually called just a path, is one which starts at the start state, follows edges continuously, and ends at an accepting state. Each path thus gives rise to a string of input symbols, and the set of all such strings corresponding to accepted paths defines a language of strings in the alphabet of input symbols, the language **recognized** by the automaton.

The language does not determine the automaton. Two automata are defined to be be equivalent when they recognize the same language.

Every language is recognized by at least one automaton, since we can make an automaton directly from the language, making every prefix of a string in the language a state in the automaton. There is an edge from ω to $\omega \bullet a$ labelled by a if $\omega \bullet a$ is a prefix of a string in the language. All states corresponding to strings in the language are accepting. The start state is the empty string. This is called the **tautologous automaton** of the language. It will have an infinite number of states if the language has an infinite number of words. However, an infinite language may well be recognized by an automaton with a finite number of states—that is to say a **deterministic finite automaton** or DFA—in which case it is said to be a **regular** language.

A **morphism** from one deterministic automaton to another is a map from states and input symbols of the first to states and input symbols of the second, compatible with the transitions in each, and satisfying in addition the condition that the start state is mapped onto a the start state and accepting to states to accepting states.

There is a canonical morphism from the tautologous automaton to any other accepting the same language.

A state is said to be **attainable** if every state can be reached by at least one path in the automaton. Throwing away the unattainable states leaves behind an equivalent automaton. I will call an automaton with no unattainable states **non-degenerate**.

One minor variant of this definition requires that every input symbol gives rise to some transition from every state. We can trivially convert a deterministic automaton in the sense defined here to one satisfying this extra condition by adding a single **dead** state and adding transitions to the dead states for every source and symbol not already defining a transition.

Regular expressions

The languages recognized by finite automata are called **regular**. They are also the ones corresponding to **regular expressions** in the input symbols, which are certain expressions in the input symbols in the same way that ordinary algebraic expressions are expressions in ordinary numbers. The operators are **catenation** $\bullet$, **alternation** $|$, **repetition** $*$. We have the following recursive characterization of regular expressions:

(1) If σ is an input symbol, then σ is also a regular expression.
(2) If r and s are regular expressions then so are $(r \bullet s)$, $(r|s)$, and $(r)*$.

The parentheses are often forgotten if there is no resulting ambiguity.

For example, let [0-9] be a short expression for 0—1—2—3—4—5—6—7—8—9, etc. Then the regular expression for a positive integer is [1-9][0-9]*, and that for a positive fraction is

$$([1\text{–}9][0\text{–}9]^*)|([1\text{–}9][0\text{–}9]^*/1[1\text{–}9][1\text{–}9]^*)|([1\text{–}9][0\text{–}9]^*/[2\text{–}9][1\text{–}9]^*)$$

The association between languages and regular expressions is that

> The symbol σ corresponds to the singleton string σ;
> $r \bullet s$ corresponds to the concatenation of strings from r and s;
> $r|s$ corresponds to the strings which come from r or s;
> the strings corresponding to $r*$ are those obtained by any finite concatenation of 0 or more strings from the language corresponding to r.

Non-deterministic automata

One important variant of the preceding definition is that of a **non-deterministic automaton** or NDA, which should more properly be called an automaton which is not necessarily deterministic. For this, several conditions in the earlier definition are relaxed: (1) There may be several starting states; (2) There may be several transitions (but a finite number) from a given source labelled by the same input symbol. (3) There may also be transitions labelled by the **null** symbol, conventionally called ϵ.

For such an automaton, a conforming path may start at any of the starting states, and may follow any ϵ transition without any condition on input symbols.

Such an automaton is non-deterministic in the sense that the association of path to string is not well defined—a given word may correspond to any one of several conforming paths. We shall see later why non-deterministic automata are important, but the main fact about them is this:

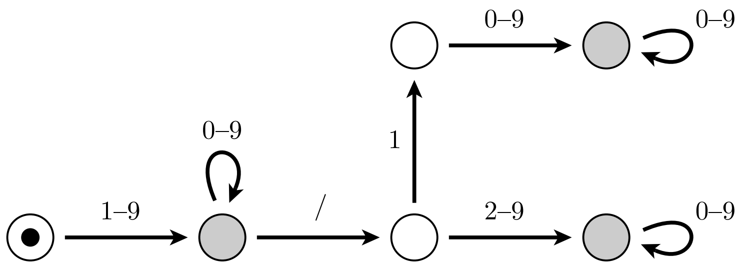
Theorem. *Every non-deterministic automaton is equivalent to a deterministic one.*

In fact, there is a canonical construction of an equivalent deterministic one. This is called the **subset construction**. Each state in the new automaton is a set of states in the original one. To make up one of the new states, we assemble together all the states reachable by a path conforming to one of the strings in the language. For example, the new start state is made up of all the original start states together with all those states reachable from one of them by an ϵ-transition, since these are the states corresponding to the empty string. There will be a transition from one of the new states s to another t on symbol σ if there is a string x reaching one of the states in s such that $x \bullet \sigma$ reaches a state in t. This recipe can be expanded to become an algorithm for the explicit construction of the new automaton, but because there are 2^n subsets of a set of n elements it may be exponential in complexity, although that happens rarely in practice.

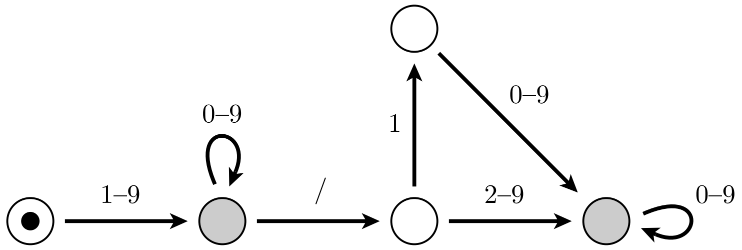
Yet another variant of a deterministic automaton is one that allows the transitions to be labelled by regular expressions rather than single characters. Each of these is also equivalent to a deterministic automaton.

Minimal automata

Let's look again at the automaton describing positive fractions:



The final states are equivalent in the sense that once a string reaches either of them, the possible subsequent choices of characters for a string are the same. So as far as recognizing fractional expressions is concerned, we may amalgamate these two states into one, obtaining this automaton:



There is in fact a canonical way to associate to every automaton a unique minimal one by identifying states in this way. It is **strongly minimal** in the sense that every non-degenerate automaton recognizing the same language maps canonically onto it. How is it constructed? If σ is a prefix of strings in a language $\mathcal{L}$, define the set $\mathcal{C}_\sigma$ of its **continuations** to be the set of all strings τ such that $\sigma \bullet \tau$ lies in $\mathcal{L}$. By definition, it is not an empty set. The empty string will lie in it if σ itself lies in $\mathcal{L}$. There will be an edge in the tautologous automaton from σ to $\sigma \bullet a$ if and only if the string a lies in the continuation of σ. Therefore we can define an automaton $\mathcal{M}$ whose states are the equivalence classes of state of the tautologous automaton of $\mathcal{L}$. Its start state will be the class made up of prefixes whose continuations are equal to all of $\mathcal{L}$, which is the equivalence class containing the equivalence class of the empty string, the start state of the tautologous automaton. Since a prefix will be in $\mathcal{L}$ precisely when its continuation set contains the empty string, the accepting states of the new automaton will be those made up of prefixes which are themselves in $\mathcal{L}$. There is certainly a canonical morphism from the tautologous automaton onto this one. Every state in it is by definition attainable. Why does $\mathcal{M}$ recognize the language $\mathcal{L}$? Because every accepting path in $\mathcal{M}$ lifts to an accepting path in the tautologous automaton.

Let $\mathcal{A}$ be any automaton recognizing $\mathcal{L}$, all of whose states are attainable. To each state of $\mathcal{A}$ associate the set of continuations of accepting paths in the automaton passing through that state. This gives a morphism from $\mathcal{A}$ onto $\mathcal{M}$.

As we shall see in a while, if we are given any finite automaton we can construct explicitly and efficiently the corresponding minimal automaton. Therefore one way we can tell whether two automata are equivalent—by telling whether they give rise to the same minimal automaton recognizing the same language.

The minimization algorithm

We start with a finite automaton $\mathcal{A}$. We define two states σ_1 and σ_2 in $\mathcal{A}$ to be equivalent if for every string τ the states $\sigma_1 \bullet \tau$ and $\sigma_2 \bullet \tau$ are either both accepted or both not. The automaton we are looking for has as its states the equivalence classes of this relation. We want to construct the equivalence classes explicitly.

We do this inductively, by introducing approximations to this equivalence. We say $\sigma_1 \equiv_0 \sigma_2$ if σ_1 and σ_2 are either both accepting or both not. We say $\sigma_1 \equiv_{n+1} \sigma_2$ if $\sigma_1 \equiv_n \sigma_2$ and $\sigma_1 \bullet a \equiv_n \sigma_2 \bullet a$ for all input symbols a. Then two states are equivalent if they are equivalent for each equivalence of order n. This suggests the following procedure:

> Divide the states into two components, the accepting states and the rest. This gives blocks of states B_0 and B_1. Let B be the smaller of the two. For each input symbol a, put (B, a) on a process list. While this list is not empty: pop the top pair (B, a) off the list. For each y in B look at all states x such that $x \bullet a = y$. Record the block B_x in which x appears. At the end of this progress through all y, look at each of the blocks B_* that we have met in this way. If the count of appearances is not all of B_*, it means that we must divide B_* into two components, the $x \in B_*$ such that $x \bullet a \in B$ and its complement in B_*. Make these splits, and in each case, let C be the smaller block. For each input symbol b put (C, b) on the process list.

The AHU algorithm

In practice this uses an idea of Aho-Hopcroft-Ullmann a bit more general. We are given

- *A set S*
- *A partition of S into blocks B_0, B_1, ..., B_{n-1}.*
- *A set of maps $f_c\colon S \to S$.*

We define an equivalence relation

$$s \equiv t$$

to mean that

$$f_{c_1} f_{c_2} \ldots f_{c_k} s \text{ and } f_{c_1} f_{c_2} \ldots f_{c_k} t$$

lie in the same block B_i for all $k = 0$, $k = 1$, ...

Often implicit is a **dead** element consisting of a single element δ with $f_c(\delta) = \delta$ for all c. There may be other (explicit) elements associated with it—for example, in the case of an automaton where the non-accepting states are put in the same block as the implicit fail state. So there must be an optional *dead* block specification.

Or, we can define a sequence of equivalence relations $\equiv_n$ by these rules:

$$s \equiv_0 t$$

means that s and t lie in the same block.

$$s \equiv_{n+1} t$$

means that

$$s \equiv_n t, \qquad f_c(s) \equiv_n f_c(t)$$

for all c. Then $s \equiv t$ if and only if $s \equiv_n t$ for all n.

The typical (only?) example constructs the minimal automaton associated to an arbitrary one—minimal, that is to say, among those recognizing the same language. Here the c range over the input symbols, the initial block decomposition is into accepting and non-accepting states, and $f_c(s)$ is the target of the transition from s on symbol c.

The idea now is to describe the algorithm to construct the equivalence classes. It is (surprisingly) of order $n \log n$ in time.

We begin by assigning indices to the blocks and blocks to the elements, given the array of sets B. We assign indices to the blocks dynamically as we assign blocks to elements, because we don't want to deal with the empty blocks. We assign a dead index dynamically, too. We then make up lists $\ell(y, c)$ for each c and each explicit element y, of all x with $f_c(x) = y$. Then we assemble all but one of the blocks, which must be the one containing δ if it exists, into doubly-linked lists, with a pointer from each y to its node, and an assignment of block to each element. We also maintain a process stack of pairs (B, c) where B is a block and c a parameter. And for each block we keep track of its size. Start by putting all (B, c) on the process list, except possibly for the dead block. Then:

(1) As long as the process stack isn't empty, pop a pair (B, c) from the process list.

(2) Pass through all the elements y in B, and for each one through all the elements x with $f_c(x) = y$ —i.e. through all of $\ell(y, c)$. In effect we look at all x with $f_c(x)$ in B. As we go through the x, maintain a set and a list of all blocks $B_* = B_x$, building an alternate block list (doubly linked) as we go. *We add a block met to our list only if it has not already been put there* or we get an $O(n^2)$ process.

(3) At the end of this process (associated to (B, c)) we have a list of blocks B_* met. Each of these must be split into two, unless it is a complete block. The split is into the new one A_* containing the x's we have met, the other the complement B_* (the old block in fact). For each input symbol d we put (A_*, d) on the process stack if (B_*, d) is already there, and otherwise put the smaller of the two blocks on it. Go to (1).

The set S is given to us as integers in $[0, n)$ (excluding the dead element if there is one). The blocks are given as bit-mapped sets, with one implicit (containing the dead element if it exists). The maps f_c are given as a double array $m[][]$ of integers where $m[s]$ is a list of successive pairs(c, t) (symbol, target) for each s in $[0, n)$. Any maps not listed are assumed to be to the dead element. The constructor for a partition is a pair an array of sets B and an array $m[][]$. THe blocks are put in a stack, and is dynamic. The inverse image arrays are constructed. As we look at all x with $f_c(x)$ in B, we maintain a list of blocks met, as well as the list of the x themselves. Do this by maintaining in each pass over (B, c) an array of Stacks, one for each block, itself a Stack since it must grow. Each stack for a block lists all x in the block met (a pointer to the node of that x). When we have finished with (B, c) we make new blocks as we go, removing a block when we split it. WE also mainatin for each x the index of its block (in an array) and a pointer to the double link node corresponding to it (also in an array). The sets B_i must be disjoint at all times.

The actual output of the construction is essentially just the list of the (new) blocks assigned to the elements.

Automaton algebra

There are explicit algorithms leading back and forth between a regular expression and an automaton recognizing its language, and also an explicit algebra constructing: (1) from a single automaton A with language $\mathcal{L}$ to one recognizing $\mathcal{L}*$; (2) from automata A and B recognizing languages $\mathcal{L}$ and $\mathcal{M}$ to automata recognizing $\mathcal{L}|\mathcal{M}$ and $\mathcal{L} \bullet \mathcal{M}$. The natural constructions give rise to non-deterministic automata even if the original ones are deterministic.

Traversal

It is often useful to display all strings in a regular language up to, say, a given length. This is done by means of a **traverse tool**.