Essays on Coxeter groups

Automata

Bill Casselman
University of British Columbia
cass@math.ubc.ca

In the world of computers, **finite automata** (also called **finite state machines**) are software tools used to analyze input at a low level, for example in reading various data types in a program. One simple example would be in reading positive fractions. The following are examples of how one normally writes them:

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

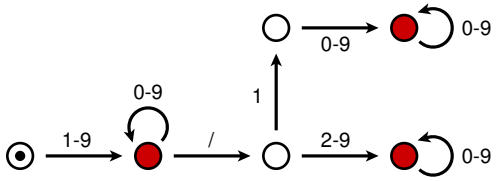
These are not:

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

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.

Note that we are not restricting ourselves to reduced fractions. What we are discussing here is *syntax*—the form of data—not *semantics*—its interpretation.

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

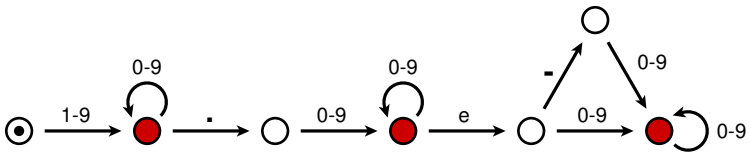


The diagram needs some explaining, but even so it's probably easier to follow than a detailed text version. The main point is that *expressions for positive rational numbers correspond to certain paths in the diagram above.* Which paths? *Those continuous paths which start at the dotted disk and end at a red disk.* I call such paths the ones that are **accepted** by the diagram. 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 indicated close to the arrow. For example, an expression for a fraction must begin with a non-zero digit, so the first transition arrow is labeled 1–9. 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 coloured because a fractional expression cannot terminate with /. The one above it is not acceptable because 1 is not an acceptable denominator.

Of course, an automaton is not really a diagram. It can be implemented in a program, with the help of tables, listing for each graph node the transitions from it, like this:

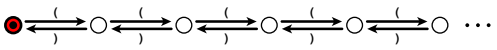
node	input	target
0	1-9	1
1	0-9	2
1	/	2
2	1	3
	2-9	4
3	0-9	3
4	0-9	5
5	0-9	5

The following diagram shows how positive real numbers (in one variant of scientific notation, for example $7.01e-3$)) can also be digested.



As these examples suggest, automata are useful in coding computer routines that parse and validate input. For example, they are used in the UNIX program LEX and its cousins (which are called lexical analyzers, as opposed to grammar parsers that analyze at a higher level). In mathematics they can be used to specify patterns that are otherwise difficult to describe. Roughly speaking, these are infinite patterns that are highly repetitive.

Computer input at the lowestlevel is generally read by automata, but there are often higher levels of organization involved. A typical example of this is the interpretation of algebraic expressions. A very simple model is offered by the strings of balanced parentheses, such as $(((())())())()$. Here the input alphabet is that of left and right parentheses (and). The strings that are accepted are those for which (1) the number of left parentheses is at least the number of right parentheses at any point; (2) at the end of input the numbers of each are the same. This set of strings can also be represented by an automaton, although infinite:



The set of balanced paraentheses is called a **context free** language. It can be described in finite terms by what are called **production rules**. An acceptable string is called an expression E, and these can be expressed recursively by the rules:

$$E = ()$$
$$E = (E)$$
$$E = EE\,.$$

In reading such an expression from left to right, one immediately replaces an expression on the right hand

side of one of these by an expression on the left. Thus in reading $(((())())()$ we record

$$
\begin{array}{l}
(\\
((\\
(((\\
((() \\
((E \\
((E) \\
(E \\
(E(\\
(E() \\
(EE \\
(E \\
(E) \\
E \\
E(\\
E() \\
EE \\
E\,.
\end{array}
$$

Incidentally, this example also illustrates that associated to the parsing of many context free languages is an automaton that recognizes **viable prefixes**, here certain strings of (and E.

Contents

1. An example from mathematics

The group $\Gamma = \mathrm{PSL}_2(\mathbb{Z}) = \mathrm{SL}_2(\mathbb{Z})/\{\pm 1\}$ 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

$$\mathcal{D} = \left\{ z = x + iy \,\middle|\, |x| \le 1/2, |z| \ge 1 \right\}$$

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 to (1) until both $-1/2 < z \leq 1/2$ and $|z| \geq 1$.

1.1. Proposition. *This procedure always halts eventually.*

Proof. I recall that

$$\text{the imaginary component of } g(x+iy) = \frac{y}{|cz+d|^2} \text{ if } g = \begin{bmatrix} a & b \\ c & d \end{bmatrix},$$

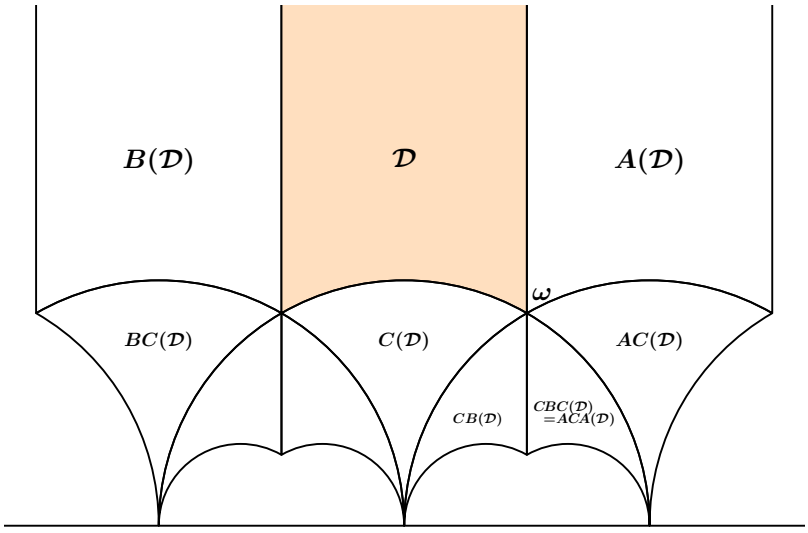
In the first step of the procedure the imaginary component of $z = x + iy$ clearly doesn't change, while in the second, according to this formula, it changes to $y/|z|^2 > y$. This means that $\text{IMAG}(z)$ strictly increases in each pass through the two steps.

On the other hand, any product of these two steps changes z into something of the form $(az + b)/(cz + d)$, with a etc. integers. In the lattice of points $cz + d$ (c, d integral) spanned by z and 1 there are only a finite number of vectors with a given upper bound on length. Therefore among the transforms of z by matrices in $\mathrm{SL}_2(\mathbb{Z})$ only a finite number of heights are possible. ∎

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

$$A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, \quad B = A^{-1} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}, \quad C = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

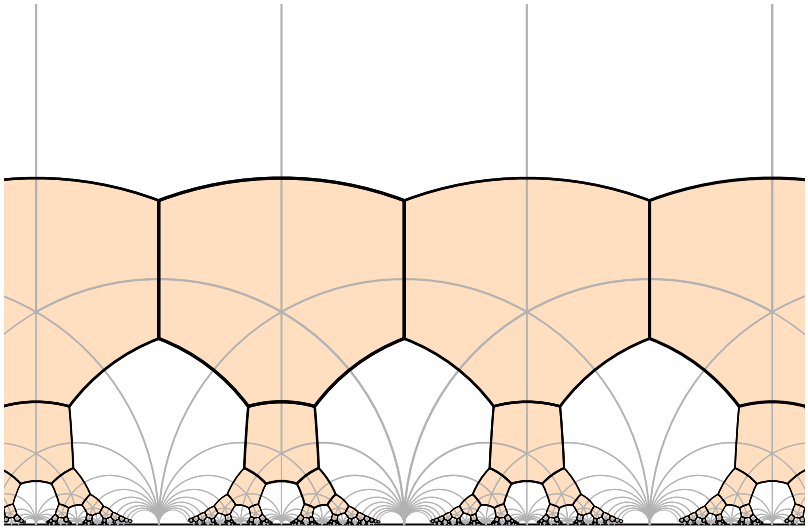
to $\mathcal{D}$.



A result stated most thoroughly in [Macbeath:1964] then guarantees that Γ is generated by A, $B = A^{-1}$, and C. The domains adjacent to the vertex $\omega = 1/2 + i\sqrt{3}/2$ of $\mathcal{D}$ are those obtained from $\mathcal{D}$ by applying A, AC, $(AC)A$, $(AC)^2$, $(AC)^2A$ and $(AC)^3 = I$. Something similar is true for the vertex $-\overline{\omega}$. Macbeath's result then implies that Γ is generated by A, B, and C with relations $C^2 = 1$, $AB = 1$, $(CA)^3 = 1$.

The rough idea of the proof is that any word $x_1 \ldots x_n$ representing the identity in the group corresponds to a sequence of neighbouring copies of the fundamental domain, which gives rise in turn to a path in a copy of the group's Cayley graph embedded in the upper half plane. This can be reduced to the trivial path by a sequence of homotopy operations guaranteed by the defining relations. In other words, any representation

of the identity I may be expressed by a path on the 1-skeleton of the cell complex in the following figure, which may then be collapsed through the hexagons corresponding to the relation $(AC)^3 = I$.



The classical algorithm for reduction of z provides a proof of generation also, since its two operations amount to replacing z by $A^{-n}(z)$ and $C(z)$ 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.

Of course elements of Γ may be represented in many different ways as a product of A, B, and C. One obvious restriction is to allow only expressions of minimal length. But even then expressions are not necessarily unique. For example, $CBC = ACA$. I adopt the following scheme for writing every element of Γ uniquely as a string $s(\gamma)$ in the symbols A, B, C.

- *The string $s(\gamma)$ has the shortest possible length among all strings representing γ;*
- *the string $s(\gamma)$ is least in dictionary order among all strings of minimal length, if we read strings backwards.*

Thus the element CBC can also be expressed as ACA, but we represent it by the second expression instead of the first, since in our dictionary it comes earlier.

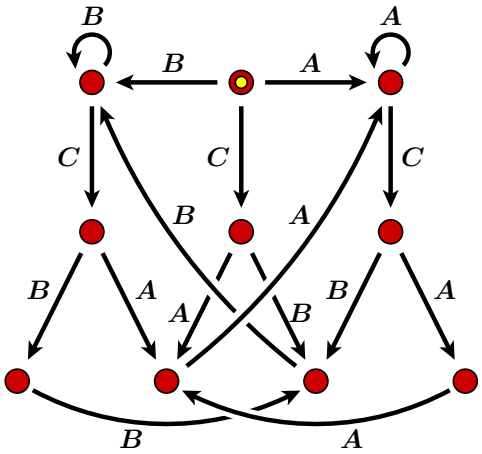
Here, for example, are all the strings representing elements of Γ of length at most 3, listed in dictionary order according to the reading-backward rule:

$\emptyset$
A
AA
AAA
AAC
AC
ACA
ACB
B
BB
BBB
BBC
BC

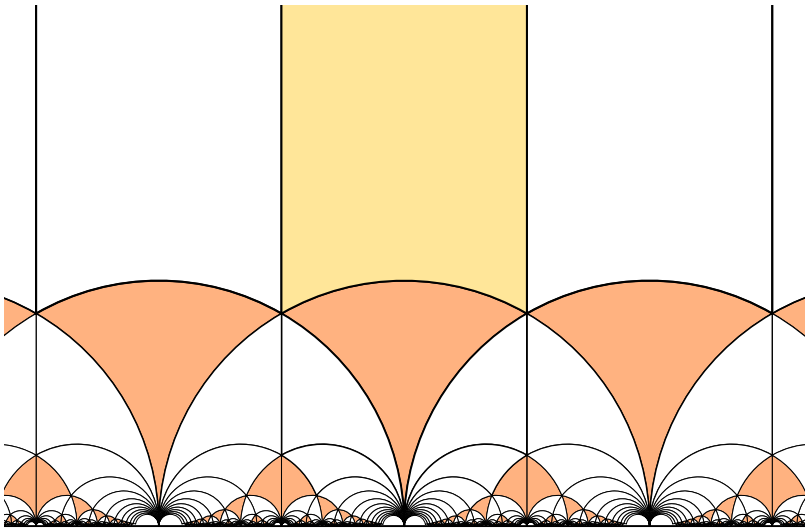
BCA
BCB
C
CA
CAA
CB
CBB

We now have the remarkable fact:

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



This regular structure underlies a number of results about the geometry of the upper half-plane. In fact, a similar theorem is true for all arithmetic subgroups of $\mathrm{SL}_2(\mathbb{R})$, as is explained in the book [Epstein *et al.*:1992]. Such a result is extraordinarily useful for producing diagrams such as this one:



The point is that it is easy to generate paths in an automaton, say up to a certain length. In the course of traveling from one state to another with transition symbol s you can perform some task that depends on both the source and the symbol s, and store the results on a stack that grows and shrinks with the length of

the path. I'll come back later on to this topic. I'll also say something later on about how you can literally see from a diagram why it is generated by a finite automaton.

I'll begin a more formal discussion of automata in the next section. But I hope the example of $\mathrm{SL}_2(\mathbb{Z})$ will demonstrate that finite automata are of mathematical interest. The essential point is:

Finite automata encode an infinite structure in a finite amount of data, and in a relatively simple fashion.

To decide whether or not a given mathematical structure can be described by a finite automaton is almost always an interesting, and usually a non-trivial, question. Finding a practical way to construct such an automaton can be even more difficult, and also interesting. There are many, some quite sophisticated, ways to encode an infinite structure in a finite one, but automata are among the simplest.

2. Deterministic automata

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

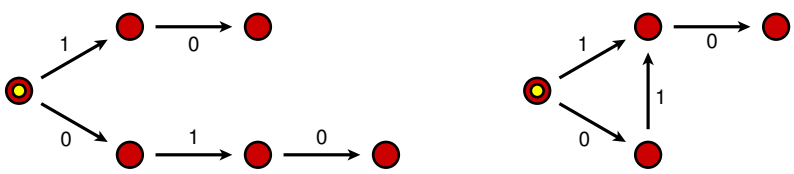
- a set of **states**;
- a finite set of **input symbols**;
- a single state designated as the **start state**;
- a subset of states designated as **accepting**;
- a set of **transitions** from one state to another, labelled by input symbols.

The transitions are subject to the condition that from each state there is no more than one transition labelled by a given symbol—i.e. the transition is *determined* by input. The transitions are therefore defined by triples (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 **formal languages** ('language' in a technical sense, which is to say a set of strings with letters chosen from an alphabet according to certain rules). A **conforming path** in an automaton is one which begins at the start state and follows edges continuously. It is **accepted** if it 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. For example, both of the following automata determine the language

$\emptyset$
0
01
010
1
10



Every language is recognized by at least one automaton, since we can make an automaton directly from the language:

(a) Every prefix of a string in the language becomes a state of the automaton;
(b) there is a transition from the prefix ω to $\omega \bullet a$ labelled by a if $\omega \bullet a$ is also prefix of a string in the language;
(c) all states corresponding to strings in the language are accepting;
(d) the start state is the empty string.

This is called the **tautologous automaton** of the language. The corresponding graph is a tree with root equal to the starting node. 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 the start state and accepting states to accepting states. Of course there is a canonical morphism from the tautologous automaton to any other accepting the same language.

In my present definition of an automaton, every input symbol defines a transition from every state. But in practice, as we have already seen, this is not how an automaton is specified. Instead, only a subset of transitions are defined. Transitions not explicitly defined are to an implicit dead state, which is not accepting and from which all transitions are to itself.

A state is said to be **accessible** if it can be reached by at least one conforming path in the automaton. A state is called a **dead end** if no accepted paths pass through it. Removing all inaccessible states from an automaton and collapsing all dead ends to a single one is harmless, in the sense that it results in an equivalent automaton. In this case, the automaton is said to be **non-degenerate**.

3. 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.

Also for an NDA is there an associated language, made up of accepted paths in the automaton following these rules:

- They may start at any of the start nodes;
- they must end at an accepting state;
- they may follow any ϵ-transition without regard to input.

Such an automaton is non-deterministic in the sense that the association of path to string is not well defined—a given string may correspond to any one of several conforming paths. Possibly non-deterministic automata are important because they arise as intermediate steps in various natural constructions. 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, as we shall see later.

I have said that NDAs occur as an intermediate in the course of very natural constructions. We shall see some later on, but there is one I can describe very simply. Suppose $\mathcal{L}$ is the language defined by a finite automaton $\mathcal{A}$. Let $\mathcal{R}$ be the language whose accepted strings are the strings of $\mathcal{L}$ written backwards, the **reverse** of $\mathcal{L}$. This, too, is the language defined by a finite automaton, but it is most naturally constructed in two steps. The first is the construction of an NDA $\mathcal{A}^{-1}$, the second the construction of a DA equivalent to $\mathcal{A}^{-1}$. As for the first step:

- the nodes of $\mathcal{A}^{-1}$ are the same as the nodes of $\mathcal{A}$;
- the edges of $\mathcal{A}^{-1}$ are the edges of $\mathcal{A}$, but oriented in the opposite direction;
- the start states of $\mathcal{A}^{-1}$ are the accepting states of $\mathcal{A}$;
- the only accepted state of $\mathcal{A}^{-1}$ is the start state of $\mathcal{A}$.

4. Constructing a DA from an NDA

Suppose $\mathcal{A}$ to be an NDA. How do we construct an equivalent DA?

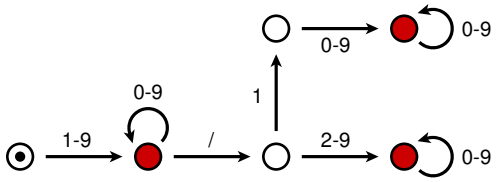
In brief, by the **subset construction**. Each state in the new automaton is a set of states in the original one. To make up one of the new states, we assemble together all the states reachable by a path conforming to one of the prefixes of the language. In the following more detailed specification, let $\mathcal{N}$ be the original NDA, $\mathcal{D}$ the new deterministic one.

- The start state of $\mathcal{D}$ is made up of all the start states of $\mathcal{N}$ together with all those states of $\mathcal{N}$ reachable from one of them by an ϵ-transition, since these are the states corresponding to the empty string.
- Given the state s of $\mathcal{D}$, which is a union of $\mathcal{N}$-states and input symbol a, the transition from s on symbol a will be to the $\mathcal{D}$-state t which is the union of all states reachable from an $\mathcal{N}$-state in s by a transition $\epsilon a \epsilon$.
- One of the $\mathcal{D}$-states will be accepting if any one of the $\mathcal{N}$-states in it is accepting.

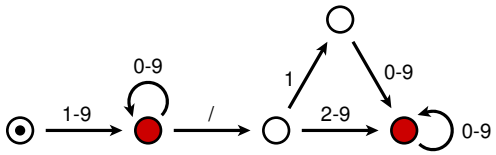
This specification can easily become an algorithm for the explicit construction of the new automaton. Because there are 2^n subsets of a set of n elements the construction may be exponential in complexity, although in practice that happens rarely.

5. 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:



As I have already mentioned, two automata are said to be equivalent if they recognize exactly the same languages, as these two do. It turns out that every automaton is equivalent to a unique minimal one. This minimal one is **strongly minimal** in the sense that every non-degenerate automaton recognizing the same language maps canonically onto it. This property suggests its construction as a kind of quotient of the original.

If σ is a live and accessible state of an automaton $\mathcal{A}$, it gives rise to an automaton $\mathcal{A}_\sigma$ of all states of $\mathcal{A}$ accessible from it and with itself as the start node, as well as the language $\mathcal{L}_\sigma$ determined by $\mathcal{A}_\sigma$. The strings of $\mathcal{L}_\sigma$ are the **continuations** of strings leading to σ, made up of all strings τ such that $x \bullet \tau$ is a prefix in $\mathcal{L}$

whenever x leads to σ. Since σ is not a dead end, $\mathcal{L}_\sigma$ is not an empty set. The empty string will lie in it if σ itself is accepted in $\mathcal{L}$.

I define the states of an automaton $\mathcal{M}$ to be the equivalence classes of states in $\mathcal{A}$. There will be an edge in $\mathcal{A}_\sigma$ from σ to $\sigma \bullet a$ if and only if the singleton string a lies in the continuation of σ. Therefore we can define an edge in automaton $\mathcal{M}$ from one equivalence class to the other. Its start state will be the equivalence class of the start state of $\mathcal{A}$, and its accepting states the equivalence classes of accepting states of $\mathcal{A}$. Every state in this quotient state is by definition attainable. Why does $\mathcal{M}$ recognize the language $\mathcal{L}$? Because every accepted path in $\mathcal{M}$ lifts to an accepted path in $\mathcal{A}$.

To each state of $\mathcal{A}$ associate the set of continuations of accepted paths in the automaton passing through that state. This gives a morphism from $\mathcal{A}$ onto $\mathcal{M}$.

As we shall see in the next section, if we are given any finite automaton, the corresponding minimal automaton $\mathcal{M}$ can be constructed in a remarkably efficient manner. This allows, among other things, a simple way to check whether tow automata are equivalent—by telling whether they give rise to the same minimal automaton.

6. The minimization algorithm

Suppose $\mathcal{A}$ to be a finite automaton. In this section I want to explain how to construct the equivalent minimal automaton. Two states σ_1 and σ_2 in $\mathcal{A}$ are 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 minimal automaton we are looking for has as its states the equivalence classes of this relation. We want to construct these equivalence classes explicitly.

This is a special case of a problem solved by a surprisingly efficient algorithm whose original appearance seems to be in [Hopcroft:1971]. This is worth explaining because the more general problem occurs elsewhere in mathematics. In this general scheme, we are given

- *a set S;*
- *a partition of S into blocks B_0, B_1, . . . , B_{n-1};*
- *a finite set of maps f_c: $S \to S$.*

In the case of an automaton the set S is that of states; there are two blocks B_0 and B_1, made up of non-accepting and accepting states; and the maps are the concatenations $\sigma \mapsto \sigma \bullet a$. But normally we shall have to modify initial data in order to fit he specification. This is because in the usual specification of an automaton concatenation is only partially defined. There is an implicit **dead state** I'll call δ allowing transitions on arbitrary input—$f_c(s) = \delta$ if f_c is not originally defined, and $f_c(\delta) = \delta$ for all c. Also, δ is not an accepting state. This is in principle not a problem. In practice, however, some miracle allows us to avoid ever working explicitly with δ, as I'll explain.

If $\gamma = (c_0, \dots, c_{n-1}$ is an array of elements in C, I define

$$f_\gamma(s) = f_{c_0} f_{c_{i+1}} \dots f_{c_{n-1}}(s)\,.$$

and then define an equivalence relation on S:

$$s \equiv t$$

means that every $f_\gamma(s)$ and $f_\gamma(t)$ lie in the same block. The maps f_c then induce maps of equivalence classes.

The problem at hand is to construct the equivalence classes explicitly. The algorithm I am about to explain is a bit subtle. Before I begin, I remark that the book [Hopcroft-Ullman:1979] (pages 68–71) explains a simpler if much less efficient algorithm, that might motivate the one I'll present here.

The rough idea of the algorithm is to keep on splitting up the original blocks into smaller ones, until no more are to be split, according to a criterion I'll explain in a moment. At that point, each of the blocks is an equivalence class. It will be immediately apparent that if a split puts two of the original states are put into

different blocks, they are not equivalent. What is subtle is that if the remain in the same block throughout the process, then they are equivalent.

There is some preliminary work to be done.

We are given at the beginning in our program: (a) a list of elements s_i in S; (b) a list of blocks B_i partitioning S; (c) a set of maps f_c from S to itself.

To be precise, each element s in S is assigned an index, an initial block assignment $B(s)$, and an array of elements $f_c(s)$ for input symbols c. In practice, as I have said, f_c is defined for only a subset of c, and there is an implicit dead element δ with $f_c(s) = \delta$ if it is not explicitly defined. Each block is a set of s, with B containing s if and only if $B(s) = B$.

The first computation is to make up lists of the sets $f_c^1(y)$ for each c and each element y, of all x with $f_c(x) = y$. The miracle is that *we do not have to do this for* $y = \delta$, as we shall see.

Then we put pairs (B, c) on a process list. We do this for each block B, with one exception, and each input symbol c. Being able to except one block B is part of the miracle. If there is a dead state δ, the block we except is the one containing it—for example, in the case of an automaton, the block of non-accepting states. So in the case of an automaton we put on the process list on the pairs (B_1, c) where B_1 is the block of accepting states.

It is important keep track of which pairs (B, c) are currently on the process list. In many programming languages, this can be done by means of a hash table or dictionary, with say the pair (index of B, c) as look-up key.

The process list is just any list. What order pairs are removed from it is not important.

The most interesting question is, *why can we get away with excepting one block?* A related question: *why should this exception be the one containing the dead state?* I'll answer the second question first—in the course of the algorithm, as we shall see, we scan through all the inverse images $f_c^{-1}(y)$ for y in on eof the blocks in the process list. In practice, most transitions are to the dead state, so its inverse image will generally be relatively large, even huge. We want to avoid even thinking about scanning through it, and this we can do by keeping the block it lies in off the process list.

The first question is more interesting, and in fact justifies (and motivates) the algorithm. As with every comprehensible algorithm, this one possesses a certain invariance property as it proceeds. The basic loop in the program starts out by removing a pair (B, c) from the process list, and in the loop various blocks may be split into smaller ones. In order to understand the program, we have to understand, *what does it mean for a pair to be on the process list?* I'll answer this, essentially, by answering the contrary question.

At the moment we remove a pair, every possible pair (B, c) is on the process list unless there exist zero or more pairs (B_i, c) on the list with the property that the union of $f_c^{-1}(B)$ and the $f_c^{-1}(B_i)$ is a union of complete blocks.

We must verify that (a) this is true when the loop is first entered; and (b) whatever happens inside the loop does not invalidate this claim. As for (a) it is OK because we have excepted precisely one of the original blocks. The invariance condition (b) we shall see in time. The significant thing is that at the end, when there are no more pairs to be processed, it must happen that every $f_c^{-1}(B)$ is a union of current blocks. This is what insures that the final blocks are in fact the equivalence classes we are looking for.

At any rate, after the pairs are put on the process list:

(1) As long as the process stack isn't empty, pop a pair (B, c) from it. 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 the list of $f_c^{-1}(y)$. As we go through these x, maintain a list of x and blocks B_x encountered. It is important to add a block met to our list only if it has not already been put there, or we get an inefficient process. This is why we have to maintain the hash table of stuff on the list.

(2) At the end of the search determined by (B, c) we have a list of blocks met and the elements in it that we have met. Each block we meet is accompanied by a list of the items we met that belong to it. We check to see whether or not we have encountered the entire block. If so, we do nothing. Otherwise, we split it into two smaller blocks. The split is into the new one B_* containing the x's we have met, the other its complement B_{**}, which used to be one of the old blocks. We now run through all pairs (B_{**}, d), and for each one do one of three things, depending on conditions: (a) The old block B_{**} is the old dead block. We put (B_*, d) on the process list. (b) The old pair (B_{**}, d) is already on the process list. We put (B_*, d) on it, too. (c) The pair $B_{**}, d)$ is not on the process list. We put the smaller of (B_*, d) and (B_{**}, d) on the list. Loop back to (1).

In leave it as an exercise to verify our 'invariance' property is preserved in this loop.

The process stops when there are no pairs (B, c) still to be processed. The output of the construction is essentially just the list of the (new) blocks assigned to the elements, and a list of elements in each block.

Why does the program eventually stop? Because the refinement process is a filter. If no splits are made, no pairs are put on the process list, which must eventually be emptied. *When it does stop, why are the blocks the same as the equivalence classes?* It should be immediately clear from the criterion for splitting that if σ and τ are not in the same block then they are not equivalent. As for the other, at the end no (B, C) is not on the process list. But because the loop-invariant condition the inverse image of every $f_c(B)$ is a union of blocks. This means that for all of the final blocks B, $f_c(B)$ is contained in another of the current blocks. Induction tells us all elements of B are equivalent.

7. 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, which are actually defined as operators on regular languages. In the following examples, let `[0-9]` be a short expression for a single digit. Then the operators are:

(a) **concatenation** $R \bullet S$. The language made up the words of R followed by the words of S. For example, `[0-9]`$\bullet$`[0-9]` is the language of all two digit expressions. In practice the concatenation symbol is left out.
(b) **alternation** $R \mid S$. The words that are in either R or S. For example, `[0-9]` is short for the alternation of digits `0|1|2|3|4|5|6|7|8|9`. And `[0-9]|[a-z]|[A-Z]` is the set of single symbols which is either a digit or a letter.
(c) **repetition** $R*$. The sequences $r_1 \ldots r_n$ with each r_i in R. The length n may be 0. Thus `[0-9]*` is the set of all strings of digits, including the empty string.
(d) **negation** ^. The words in the alphabet of R that are not in R. If the alphabet is that of all digits, the negation `([^0][0-9]*` is the set of all strings of one or more digits not starting with `0`.

We have the following recursive characterization of regular expressions:

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

The parentheses are often forgotten if there is no resulting ambiguity. Thus the regular expression for a positive integer is $[1-9][0-9]*$, and that for a positive fraction is

```
([1-9][0-9]*)|([1-9][0-9]*/1[1-9][1-9]*)|([1-9][0-9]*/[2-9][1-9]*)
```

Regular expressions do not occur commonly in mathematics, but are ubiquitous in the background of computer tasks.

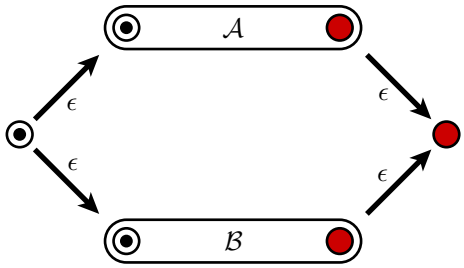
The main theorem here is that regular expressions are precisely the ones recognized by finite automata. I'll not explain the correspondence in detail. The natural construction is to construct first an NDA from a regular expression, then make the equivalent DA.

8. Automaton algebra

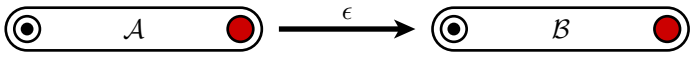
There are explicit algorithms leading back and forth between a regular expression and an automaton recognizing its language, and also an explicit algebra constructing: (1) from a single automaton A with language $\mathcal{L}$ to one recognizing $\mathcal{L}*$; (2) from automata A and B recognizing languages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

I have already described how to construct the reverse as an NDA. The other constructions are similar, and I'll describe them briefly.

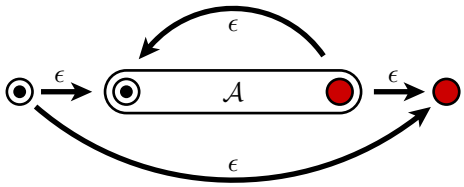
• *Alternation.*



• *Concatenation.*



• *Iteration.*



The intersection of two regular languages is also regular defined by the product of two automata.

9. Traversal

It is often useful to display all strings in a regular language up to, say, a given length. Or to perform some computation that proceeds by induction on the length of a path. This is done by means of a **traverse tool**. This uses a stack or a queue to follow paths in an automaton. Which you use depends on the precise nature of the induction you are applying. The stack is slightly quicker. I don't know how to explain how this business works except to show you a Python class that I use. Here, `fsm` is a class that represents the automaton, with a single start state and for each state an array `move`.

```
class Traverse:
    def __init__(self, fsm, *args):
        self.fsm = fsm
        if len(args) > 0:
            self.ht = args[0]
        else:
            self.ht = -1
        self.stack = None

    def start(self):
        self.stack = []
        for s in self.fsm.start_list():
            self.stack.append([ 0, -1, s ])
        return self.next()

    def next(self):
        stack = self.stack
        if len(stack) > 0:
            [h, s, x] = stack.pop()
            if self.ht < 0 or h < self.ht:
                for i in range(len(x.move)):
                    for y in x.move[i]:
                        stack.append([h+1, i, y])
            return [h, s, x]
        else:
            return None

    def count(self):
        count = 0
        n = self.start()
        while n:
            count += 1
            n = (self.next())
        self.reset(self.ht)
        return count

    def wordlist(self):
        stack = [None]*(self.ht+1):
        stack[0] = []
        n = self.start()
        while n:
            [h, s, S] = n
            if h > 0:
                stack[h] = stack[h-1] + [s]
            print stack[h]
            n = self.next()
```

The function `count` is an example of how to use the tool. It returns the number of paths through the automaton up to a certain height. The procedure `wordlist` prints all the paths encountered up to that height. At the moment the triple `[h, s, S]` is encountered, the stack up to height $h - 1$ holds the prefix of the current path.

10. The hierarchy of languages

There are formal languages of interest in mathematics more complicated than regular ones. My guess is that they occur far more often than is generally recognized. They vary a lot in nature, primarily in how complicated they are. I believe it was the linguist Noam Chomsky who first proposed to classify the formal languages that one has any hope of dealing with according to the complexity required to deal with them. In rough terms, this hierarchy is:

- finite;
- regular;
- context free;
- context-sensitive;
- recursively enumerable.

These categories are listed in order of the complexity of interpretation and parsing. We have already seen that regular languages those which one can parse by a finite state machine. The context free languages are those for which one might need a stack—a kind of 'machine' with an arbitrary number of states—to parse, and context-sensitive languages might require a Turing machine. The book [Brookshear:1989] explains this subject very clearly in much more detail.

I'll not discuss these notions in detail, but I would like to mention some examples in which something more complicated than regular languages occurs naturally in mathematics. Let me first emphasize: a finite state machine recognizes the acceptability of a string by incorporating only a fixed amount of data from previous input, which is compressed into the identity of the current state, one of a finite set of states.

But languages we encounter in real life do not possess his limited dependence on history. The first example I'll look at is the language of legitimate algebraic expressions, say involving normal arithmetic expressions and two variables x and y. This is a context free language, because in an algebraic expression any variable may be replaced by an algebraic algebraic expression, which may itself be very complicated. The definition of algebraic expressions is therefore by recursion. It is not a language recognized by a finite state machine. To explain why not, I consider a simple model, the language of balanced parentheses, in which every left parenthesis `(` is matched to its right by a `)`, and what is in between is also balanced. You can tell whether a string of parentheses is balanced by scanning from left to right, keeping track of balance by calculating a 'balance' count n, adding 1 to it when we met a `(` and subtracting 1 when we meet a `)`. The state of the scan is encapsulated in this single number: we must have $n \geq 0$ at all times, and at the end it must be 0. But since there are an infinite number of non-negative numbers, the number of states of the scan is also infinite.

The state can also be recorded by means of what programmer call a **stack**—in this case an array that is shortened when n is decremented, extended when n is incremented. Context-free languages are always, in principle, readable by a suitable use of a stack.

Languages like that of algebraic expressions are familiar to mathematicians, but in fact natural languages are very close to context-free as well, and the human brain seems to have built into a capacity for context-free recursion to match this.

Most modern programming languages, such as `C` or `Java` are based on context-free specifications. (The few that are not, for example PERL, are notoriously tricky to work with.) Teaching computers how to parse expressions in one of these offers some interesting problems. I don't want to say anything about these here, except to point out that the basic practical method used in parsing languages of this sort is one of Donald Knuth's principal contributions. For more about this, you can look at almost any book on compilers, for example [Aho et al.:2007], and also the collection [Knuth:2003], particularly paper # 15.

I'll look at another collection of examples in the next section. These will provide more evidence that concepts from the theory of formal languages can be helpful in understanding certain mathematical phenomena.

11. Regular languages in Coxeter groups

A **Coxeter group** is one defined by generators from a finite set S and relations of the form

$$(st)^{m_{s,t}} = I$$

where $m_{s,t} = m_{t,s}$ is a positive integer and $m_{s,s} = 1$. The $m_{s,t}$ make up a **Coxeter matrix**. Since $m_{s_i,s_i} = 1$, each s_i has order two. Every Coxeter group possesses a distinguished family of geometric realizations in which the s_i are mapped to reflections.

I'll look here at just two examples, the group A_2 with

$$m = \begin{bmatrix} 1 & 3 \\ 3 & 1 \end{bmatrix}$$

and the group $\widetilde{A}_2$ with

$$m = \begin{bmatrix} 1 & 3 & 3 \\ 3 & 1 & 3 \\ 3 & 3 & 1 \end{bmatrix} .$$

The generators s_0, s_1 of the first may be considered as a subset of the generators of the second, thus making the first group a subgroup of the second.

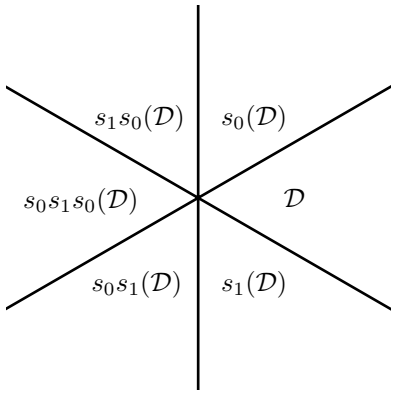
In any Coxeter group, since $s_i^2 = s_j^2$ and $(s_i s_j)^{m_{i,j}} = 1$, we also have the braid relation

$$s_i s_j s_i \ldots = s_j s_i s_j \ldots \quad (m_{i,j} \text{ terms on each side}) .$$

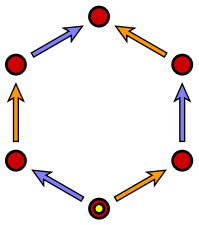
The group A_2 is finite, made up of the six elements

$$I,\ s_0,\ s_1,\ s_0 s_1,\ s1 s_0,\ s_0 s_1 s_0 = s_1 s_0 s_1 .$$

It is in fact isomorphic to $\mathfrak{S}_3$. Its geometric realization is by permutation of coordinates in $\mathbb{R}^3$. The element s_i acts by reflection in the plane $x_i - x_{i+1} = 0$. The group takes the plane $x_0 + x_1 + x_2 = 0$ to itself, and in this plane the region $x_0 - x_1 \geq 0$, $x_1 - x_2 \geq 0$ is a fundamental domain. One may picture this plane and the action of A_2 on it like this:

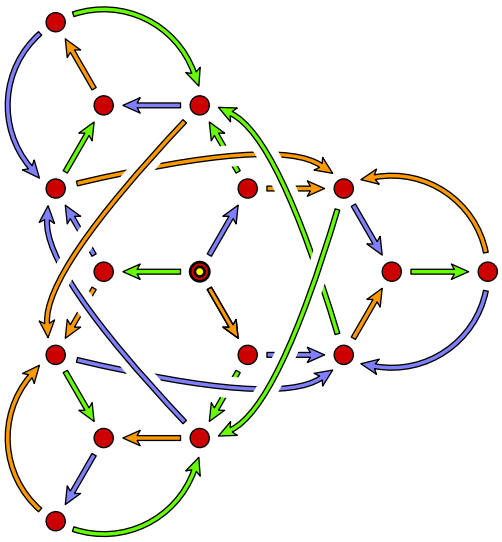


Any element of any Coxeter group can be expressed as a product of the generators in S. A **reduced** expressions for w is one of minimal length $\ell(w)$. The states of the tautologous automaton of all reduced expressions is called the **weak Bruhat graph** of the group. There is an oriented edge from w to ws when $\ell(ws) > \ell(w)$. For A_2 this is the finite automaton

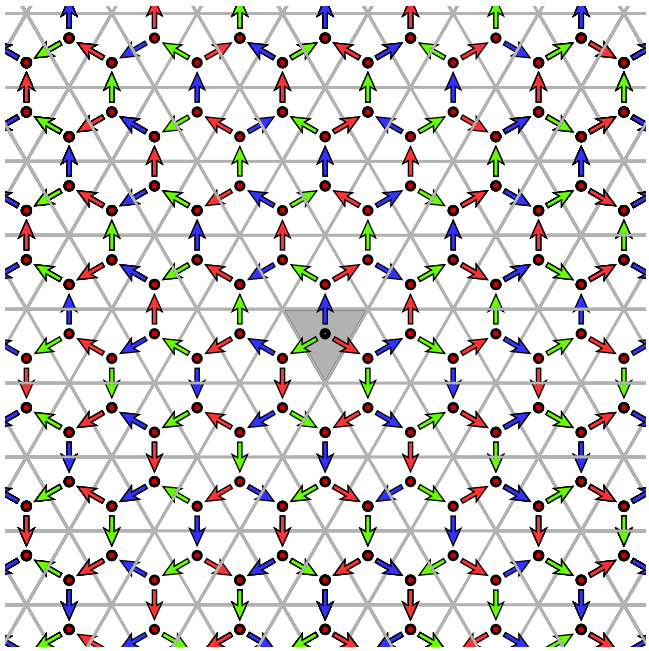


The principal theorem of [Brink-Howlett:1993] is that for any Coxeter group the language of reduced expressions is determined by a finite automaton. For any finite Coxeter group the minimal automaton recognizing reduced expressions is may be identified with the tautologous automaton.

For $\widetilde{A}_2$ the tautologous automaton is infinite, but that recognizing reduced expressions is this:

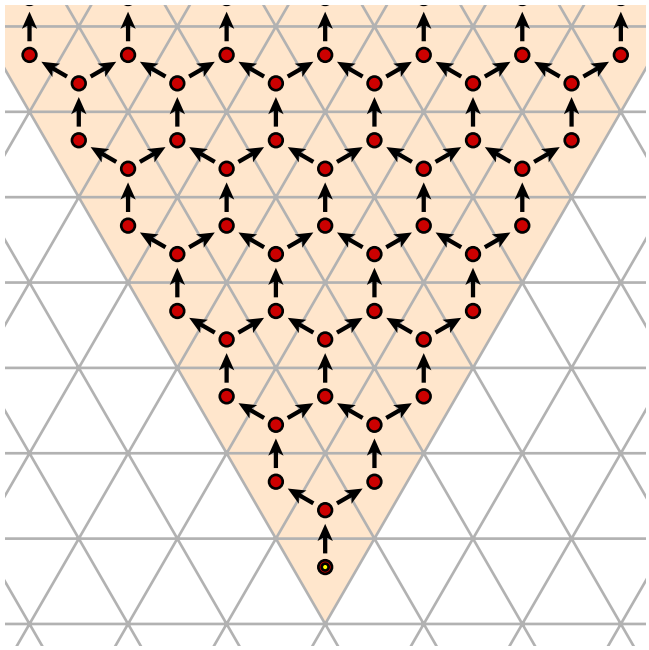


In both figures I have color-coded transitions instead of putting in labels for input symbols. The geometric realization of $\widetilde{A}_2$ takes each s_i to a reflection in one of the sides of an equilateral triangle. The transforms of this triangle tile the plane. Here is the tautologous automaton (the graph of the weak Bruhat order):



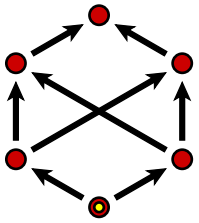
Even if you really know nothing about Coxeter group, you can understand how the paths in this diagram are generated—you start at the grey triangle around the origin, then proceed outwards, following the extremely simple rule that once you cross a line you never recross it.

But what I am most interested in at the moment is a smaller if related automaton, the weak Bruhat graph of the quotient $A_2\backslash\widetilde{A}_2$. In fact, I'll work with a simple version of it, the diagram generated by the paths in the previous diagram that are contained in one of the natural acute cones embedded in it. I ask the question, *is the following also the graph of an automaton?*

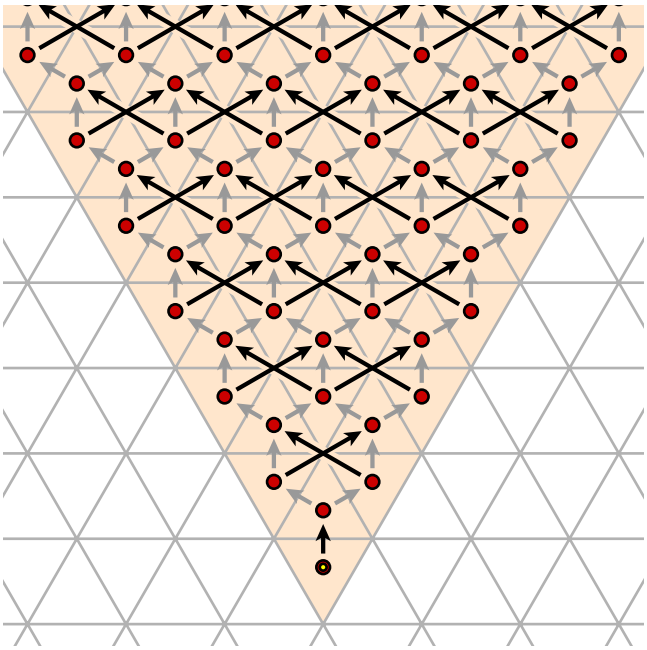


In order to understand one feature of this diagram, keep in mind that the group A_2 acts on the collection of triangles generated by $\widetilde{A}_2$, with fundamental domain an acute cone. At any rate, this diagram is generated by a finite state machine and the automaton doing that, which is non-deterministic, is very simple:

But now I introduce a very minor change, which I'll first motivate. The weak Bruhat graph of a Coxeter group has oriented edges $w \to ws$for s in S when $\ell(ws) > \ell(w)$. In the **strong Bruhat graph** there is an edge $w \to ws_\lambda$ when s_λ is any reflection in the group and $\ell(ws_\lambda) = \ell(w) + 1$. One can easily classify all the reflections in A_2, and verify that this is its strong graph:



For the left quotient $A_2\backslash\widetilde{A}_2$, there is also a strong graph, which is an assembly of copies of the one for A_2, replacing the basic pattern of the earlier figure by this one. Here it is:



This turns out to describe the closure relations of certain simple subvarieties in an infinite-dimensional flag variety. *But what is important is that even though the diagram itself was formed by a simple local substitution, the paths through it are of an entirely different character from earlier ones.* It cannot be drawn by a finite automaton, because we have to keep track of how far we are from right and left in order to know how far right or left we can traverse. We require some integer variables to keep track of these data, and this means that there are an infinite number of states involved in traversing the path. In effect, this is like the language of balanced parentheses. We have moved up one level in the hierarchy of formal languages, to one requiring a better memory for the past (i.e. stacks) in order to generate them.

There is one curious feature of this diagram that is reminiscent of computer programs—the local structure of the diagram is just that of a finite state machine, although globally it is not a finite state machine.

There seems to be one important calculation in dealing with Coxeter groups, that of the Kazhdan-Lusztig polynomials, that involves the level beyond context-free languages. These give rise in turn to the oriented graph defining the left and right Kazhdan-Lusztig cells. These also give rise to extremely interesting repetitive patterns, nonetheless of an apparent complexity beyond what we have seen so far. I hope to discuss these in a later version of this essay.

12. References

1. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, **Compilers—Principles, techniques, and tools**, Addison-Wesley, 2007.

2. Brigitte Brink and Robert Howlett, 'A finiteness property and an automatic structure for Coxeter groups', *Mathematische Annalen* **296** (1993), 179–190.

3. J. Glenn Brookshear, **Formal languages, automata, and complexity**, Benjamin-Cummings, 1989.

4. David B. A. Epstein, J. W. Cannon, S. V. F. Levy, M. S. Paterson, W. P. Thurston, **Word processing in groups**, Jones and Bartlett, 1992.

5. John E. Hopcroft, 'An $n \log n$ algorithm for minimizing the states in a finite automaton', pp. 189–196 in **The theory of machines and computation** (edited by Z. Kohavi) Academic Press, 1971.

6. —— and Jeffrey D. Ullman, **Introduction to automata theory, languages, and computation**, Addison-Wesley, 1979.

7. Donald E. Knuth, **Selected papers on computer languages**, CLSI, Palo Alto, 2003.

8. A. M. Macbeath, 'Groups of homeomorphisms of a simply connected space', *Annals of Mathematics* **79** (1964), 473–488.