Coxeter version 3.0

Fokko du Cloux

May 1, 2024

Institut Girard Desargues
UMR 5028 CNRS
Université Lyon-I
69622 Villeurbanne Cedex FRANCE
ducloux@igd.univ-lyon1.fr

`Coxeter` is a program for the exploration of combinatorial issues related to Coxeter groups and Hecke algebras, with a particular emphasis on the computation of Kazhdan–Lusztig polynomials and related questions. It is not a symbolic algebra system; rather, it is an interface for accessing a direct C++ implementation of the concept of a Coxeter group. Although I have not been able to fully reach this goal in the current version, the idea is to make the class (actually, the class hierarchy) of Coxeter groups available in the form of a C++ library, which could then be used efficiently by other programmers.

The program aims for maximum performance, both in terms of speed and in terms of memory usage; it does not aim for maximal user-friendliness. If your needs are served by higher-level programs like `GAP/Chevie` or `Maple`, by all means use those; the aim of `Coxeter` is to pick up where these programs stop. Particularly `Chevie` includes a nice set of Kazhdan–Lusztig routines, including some which are not implemented in `Coxeter`.

Extending the program is certainly possible (see below), but only for users who are on speaking terms with C++ and are willing to walk a little through the `.h` files. Extension takes place by inserting your own additional code, typically in the `special.cpp` file, and recompiling (unless you are adding whole new files, recompiling should just be typing `make`.)

1 What is wrong with this program

What is mostly wrong with this program is that is neither C nor C++. In fact, this program has been my learning ground for C++ (where I used to be a C programmer). I made the shift with some reluctance, as I tend to love the simple-minded no-nonsense

approach of C, and particularly the blazing efficiency that you can get out of it. When I finally decided to make the switch, a sizable amount of C code had already been written; moreover I couldn't afford to take the time to learn everything I should have about C++ for such a project. Therefore, although largely C++ in spirit, the program suffers from a number of severe defects and shortcomings from the C++ standpoint :

- i/o is not C++ at all; it is plain C.
- I didn't make use of the STL, even though the program makes heavy use of things that are provided by the STL, and which I had reinvented before even realizing that the STL existed : my class `List` is essentially STL's `vector`, where I use binary trees I could have used STL's `set` class, of course I should have used STL's `string`, etc.
- I'm not using exceptions at all; instead I've implemented my own error handling mechanism, probably not rigorously enough. Badly handled error conditions have been the main source of program crashes in my testing.

Memory allocation is another thorny issue. Whether this was due to clumsy programming on my part, or to the overhead of the default memory allocator, when I tried to use the builtin `new`, in presence of heavy resizing (which happens often in large computations) performance all of a sudden became terrible (and I mean terrible : slowing down by a factor of at least ten, maybe more.) Therefore I decided to write my own primitive allocator, getting only fairly large (never smaller than 2^{16} bytes by default) blocks from the system, and never returning them during the lifetime of the program. My allocations always are a power of 2 bytes, and therefore as much as 50% of the memory might get wasted (although the ratio in practice is much better), but at least the speed is satisfactory. At that time I hadn't heard yet about the possibility of specifying a memory allocator for the STL classes; in any case I will make the switch to STL classes only if this issue is resolved.

2 What the program does

One of the main improvements with respect to previous versions of `Coxeter` is that the program is now able to handle essentially arbitrary Coxeter groups (provided of course that the computation you require does not overflow your system). For convenience, and also because it seems to cover all cases where significant computations are possible, in this version I require that twice the rank of the group not exceed the number of bits in a `long` on your system (so the rank is limited to 16 on a 32-bit machine, and to 32 on a 64-bit machine.) One could argue that 16 is a bit restrictive, but certainly 32 will cover all cases of interest.

The functionalities provided by the program may be classified in a number of categories :

elementary operations : reduced form computations; products; descent sets; elementary Bruhat order comparison; coatoms.

Kazhdan–Lusztig polynomials : individual Kazhdan–Lusztig polynomials and mu-coefficients for the ordinary, unequal-parameter and inverse cases; layout of a polynomial computation in these cases; Kazhdan–Lusztig basis elements in the Hecke algebra; singular stratification, rational singular locus, ordinary and IH Betti numbers of a Schubert variety.

Kazhdan–Lusztig cells : left, right and two-sided cells for the ordinary and unequal-parameter cases; ordering on the cells of a finite Coxeter group in both these cases.

One of the serious issues that have to be faced in computational mathematics is the sheer size of the output for the computations that modern-day computers are able to handle. Dumping the output on a screen is definitely not good enough in many cases. I have no real solution for this problem; in any case it is clear that more and more output will be handled electronically, and will have to be analyzed by other programs before being usable by humans.

I offer a number of output formats for output to files : a human-readable one, a "terse" one which is designed to be easily computer-readable, for analysis or further processing, and `GAP`-format. The latter is a first step towards connecting (some appropriate version of) `Coxeter` with `GAP`.

3 What is missing from this version

A number of things that I would have liked to put in the program are missing. Foremost are parabolic Kazhdan–Lusztig polynomials. They are important in their own right, but also because the parabolic setting comes much closer to capture the real difficulty of a Kazhdan–Lusztig computation. Some computations, say of parabolic Kazhdan–Lusztig polynomials in type E_8, will not go through in this version even though they are actually rather small, only because as a preliminary the program tries to construct an enormous Bruhat interval. If we handled things in a parabolic setting, the parabolic interval would be constructed instead, which would not be a problem. This would certainly be my number one project (apart from improving the code structure) for a future version.

Also, there is nowhere as much stuff on Bruhat intervals as I would like to have. A *lot* more can be said, asked, and done about them. But perhaps this really would require a different program. It may not be a good idea trying to do everything in one place.

There is no attempt to handle special groups in any special way. The only way in which finite groups are special in this program is through the fact that some computations, such as Kazhdan–Lusztig cell computations for instance, are defined only for

them. It is likely that Bruhat order computations could be much improved for finite groups; still, my measurements seem to show that even in the current state, Bruhat order computations are not all that dominant, so that at most one could expect to gain a factor of two (and usually much less). In view of the speed already attained, this doesn't seem worth the effort. Users who are especially interested in type A should check out Gregory Warrington's programs (see `http://www.math.upenn.edu/~gwar/research/research.html`)

4 How to extend the program

The usual mechanism used by computer algebra programs to extend the capabilities of the program beyond the predefined commands is to provide an interpreted programming language that allows the user to write his own routines.

No such thing is provided here. However, there is a command called `special`, defined in `special.cpp`, which executes the `special_f` function defined in that file. Currently this function just prints out a short warning message, but it can be redefined by inserting any code you wish (and then recompiling the program of course.) It is also easy to insert new command names executing user-defined functions; it should be easy to guess what needs to be done by looking at the way the `special` command is defined in `special.cpp`. Of course this is only useful for users which are knowledgeable with C++. Also, to do anything useful, you will want to use the functionalities already provided by the `CoxGroup` class; this will require reading a few of the `.h` files to see what is available. In fact, I have tried to make the most useful commands available already at the level of the `CoxGroup` class (or sometimes in the derived class `FiniteCoxGroup` if the command doesn't make sense for a general Coxeter group.) So in principle it would be enough to look there; however, most of the functions from `CoxGroup` are simply forwarded from the various components of the class, so it will be necessary to look at a number of other files to see the definitions and comments for all these functions.

5 How things are done: word reduction

The following sections describe in some more detail the algorithms that are used in the program to perform the computations. We begin with the most fundamental one, which is word reduction.

What I have done is provide a complete, purely combinatorial implementation of the minimal root algorithm discovered by Brigitte Brink and Bob Howlett [3]; the usage of minimal roots for word reduction and normalization, and the practical aspects of the construction of the set of minimal roots, are nicely discussed in [4].

For any Coxeter system (W, S), Brink and Howlett define a canonical finite subset E of the set of positive roots of W, called the set of minimal (or elementary) roots, which contains the simple roots. Define an action of S on the set $X = E \cup \{+, -\}$, where $+$ and $-$ are two special symbols, by letting $+$ and $-$ be fixed points, and setting $s.\alpha = +$ if α is a positive root not in E, $s.\alpha = -$ if $\alpha = \alpha_s$ is the simple root corresponding to s, and $s.\alpha = \beta \in E$ otherwise. So we can now view X as a finite state automaton with alphabet S (except that we do not choose an initial state or a set of accept states at this point.) Then it is shown in [3] that if $s_1 \dots s_p$ is a *reduced* word in W, and if we read this word through the automaton, starting from the state α_s, the word $s_1 \dots s_p s$ is reduced, unless we reach the state $-$; moreover, if the generator s_j takes us from state α_t to state $-$, we have $s_j = t$ and $s_{j+1} \dots s_p s = t s_{j+1} \dots s_p$; hence the reduction. It is clear now how the above automaton can be used to reduce an arbitrary word in the generators, in at most quadratic time. This algorithm is implemented in the function `MinTable::prod` in `minroots.cpp`.

Of course it remains to construct the set of minimal roots, together with the action of the generators above. In [3] an algorithm is described in terms of the standard geometrical realization of the group. This algorithm has the drawback that it requires finding the sign of potentially complicated algebraic real numbers. Using the detailed analysis of minimal roots in [2], it is possible to make the algorithm entirely combinatorial, using only a few explicit irrationalities (and even then, in a formal fashion.) This is explained in more detail at the beginning of `minroots.cpp`. The upshot is that we are able to construct the "minimal root machine" for essentially any Coxeter group. The only possible source of trouble is memory overflow; even though for the most frequently used Coxeter groups (such as finite or affine groups) the number of minimal roots is fairly small (typically a few hundred), it can grow larger for more exotic cases. For ranks ≤ 16, if the entries of the Coxeter matrix are not too large, there should not be more than a few tens of thousands of roots; but for ranks ≤ 32, millions of minimal roots should be expected in bad cases. The program will quit if it is unable to construct the minimal root table; nothing useful can be done without word reduction.

Another problem that is neatly solved with the minimal root machine is the *word problem* for Coxeter groups : recognizing when two words in the generators represent the same element. If $s_1 \dots s_p$ is a reduced word in the generators, and s is a generator such that $s_1 \dots s_p s$ is reduced, it turns out that the procedure described above also finds all possible ways to insert a generator t in $s_1 \dots s_p$, say after the j-th letter, so that $s_1 \dots s_j t s_{j+1} \dots s_p = s_1 \dots s_p s$. Define the normal form of an element $w \in W$ to be the lexicographically smallest reduced expression of w (with respect to a chosen linear ordering on the set S). Then it is known that if $s_1 \dots s_p$ is a normal form, and $s_1 \dots s_p s$ is reduced (not reduced), the normal form of $s_1 \dots s_p s$ is obtained through a suitable insertion (deletion) in $s_1 \dots s_p$. Now it is easy to show that the minimal root machine in fact will find all possible insertion/deletion places in a given word, for the multiplication

by an additional generator s; so it is also possible to construct the normal form of an element using the finite state automaton described above.

Of course, once we have word reduction it is an easy matter to determine (though not very efficiently) the descent set of any given element of the group (recall that the left (right) descent set of $w \in W$ is the set of $s \in S$ such that $l(sw) < l(w)$ ($l(ws) < l(w)$)), where l is the usual length function on W.

Also, there is an elementary algorithm to decide, given two elements x and y in W, if $x \leq y$ for the Bruhat ordering on W. It goes as follows : *(a)* if $y = e$ (the identity element in W), then $x \leq y$ if and only if $x = e$; *(b)* otherwise, choose $s \in S$ such that $l(ys) < l(y)$ (for instance, the last element in a given reduced expression of y); *(c)* if $l(xs) < l(x)$, then $x \leq y$ if and only if $xs \leq ys$; otherwise, $x \leq y$ if and only if $x \leq ys$.

Because of the availability of efficient word reduction, the general representation of group elements in the program is through *reduced* expressions; this is the `CoxWord` class defined in `coxtypes.h`. Of course, a word in the generators cannot know that it is reduced; it is the programmer's responsibility to ensure (using the available reduction tools if necessary) that words remain reduced at all times. This makes it possible to implement the `length` function, for instance, simply by returning the length of the word as a string. On the other hand, I do not insist that words always be normal forms; this would impose too heavy an overhead for no great benefit, particularly since we are allowing the user to redefine the chosen ordering of the generators (this changes all the normal forms). I treat normal forms as an input/output issue.

An annoying little twist in the program is the following : because in C the zero-character is used as a string-terminator, it could not be used in the string-representation of group elements. On the other hand, it is really a bad idea not to start numbering the generators from zero. So we end up with the awkward situation that generator $\#s$ is represented by character $s + 1$; this requires some shifting when reading and writing strings. Hence the distinction between the types `Generator` (numbered from 0) and `CoxLetter` (numbered from 1) in `coxtypes.h`. In C++ strings are represented with an explicit length, so no special terminator character is required, and we could use zero directly. Since I'm using C++-style strings now, I could dispense with `CoxLetter`'s entirely, but haven't had the courage to take this on yet.

6 How things are done: Bruhat ordering

The elementary operations described above are actually used very little in the program, because it seems to me that they become prohibitively expensive as soon as one attempts serious Kazhdan–Lusztig computations.

In my setup, every Coxeter group W has an *enumerated part* P, which is always required to be a decreasing set (also called a closed set, or an order ideal) for the Bruhat

ordering : if $y \in W$ belongs to P, then all $x \leq y$ also belong to P. This just means that we have set up once and for all a $(1,1)$ correspondence between the integers in $[0, N[$, where N is the cardinality of P, and the elements of P; the only requirement for this correspondence is that it be increasing with respect to the Bruhat ordering, *i.e.* if $x \leq y$ for the Bruhat ordering, then x has a smaller label than y. In particular, the label of the identity element e is necessarily 0. Initially the enumerated part is the one-element set $\{e\}$. An element of the enumerated part of the group may now be represented by a single number; this is the type `CoxNbr`, defined in `coxtypes.h`.

The following tables are maintained for the enumerated part : *(a)* a table containing the lengths; *(b)* a table containing the left and right descent sets; actually both are packed into a single long integer, where the n rightmost bits flag the right descents, and the n next bits flag the left descents, if n is the rank of the group; *(c)* a table recording the result of right or left multiplication by a generator (in other words, a table with N rows, each having $2n$ entries); here a special value `undef_coxnbr` is used when the multiplication takes the element outside P; *(d)* a table which gives for each $x \in P$ the list of *coatoms* of x; these are the elements $z < x$ such that $l(z) = l(x) - 1$ — in principle, this table entirely describes the Bruhat ordering on P. The possibility of packing all descents in one `long` is the main reason for the requirement that $2n$ should not exceed the number of bits in a `long`.

Note that the data contained in these tables are (more than) sufficient to define completely the correspondence between $[0, N[$ and the elements of P. Indeed, if an element of P is given, say as a reduced word $s_1 \dots s_p$, it is enough to start from 0, and using the multiplication table p times we find the corresponding number. Conversely, if a number $a \in [0, N[$ is given, we look at the descent set to find a left descent for a, then apply this to a, and continue until we reach 0. The corresponding sequence of generators, read left-to-right, will then constitute a reduced expression of the group element labelled a. If we choose the smallest left descent (for a given ordering of the generators) at each step, we even find the normal form of the element corresponding to a.

The `SchubertContext` class, defined in `schubert.h`, maintains the enumerated part of the group. The main issue is the problem of *extension* : given an element w in the group (in the form of a reduced word, as always), which does not belong to the current enumerated part P, enlarge P so that P contains w (in fact, what we do is replace P by $P \cup [e, w]$, which is the smallest possible decreasing subset containing both P and w.) An algorithm is described in [5] which does exactly that, in a purely "internal" fashion : using only the data contained in the various tables in P, and the knowledge of the Coxeter matrix of W, it is able to find which elements of $[e, w]$ do not currently belong to P, and enlarge P and all its tables accordingly, putting the new elements on top. This enlargement has the very nice property that the only modifications that are made to the already existing enumerated part is that some previously right or left

multiplications may become defined; otherwise all existing references to elements in P remain valid. The function which performs this extension is the `extendContext` member function of the `CoxGroup` class.

The same algorithm which is used to build up the enumerated part may be used to extract a given interval from the identity $[e, w] \subset P$; things are much simpler here because nothing needs to be constructed, it is only a matter of lookup. This is done by the member function `extractClosure` from the `CoxGroup` class; this (or rather the member function from the `SchubertContext` class that it refers to) is probably the most heavily-used function in the program.

7 How things are done: Kazhdan–Lusztig polynomials

I believe that it is clear to anyone who has ever attempted to compute Kazhdan–Lusztig polynomials of any substance, that it is essential to remember polynomials already computed. On the other hand, we do not want too remember too many, since the size of the table of computed polynomials is usually the limiting factor for Kazhdan–Lusztig computations. It is well-known, already from the original Kazhdan–Lusztig paper [6], that the computation of $P_{x,y}$, for $x \leq y$ in W, readily reduces to the case where the two-sided descent set $\mathrm{LR}(y)$ is contained in $\mathrm{LR}(x)$; I call such pairs *extremal*.

Kazhdan–Lusztig polynomial computations are attempted only for elements of W which belong to the enumerated part P (cf. section 6) (in other words, the first stage of the computation is extending the enumerated part if necessary.) Again a number of tables are maintained. The first one is the table of extremal pairs : for each y in P, this table contains a pointer (initially zero), which when non-zero points to a row of numbers representing the $x \leq y$ which are extremal w.r.t. y. The second one contains for each y in P a pointer to a row of polynomials (more precisely, of pointers to polynomials), one for each extremal x; finally, for each y in P a list is maintained which, when initialized, is guaranteed to contain all $x \leq y$ for which the mu-coefficient $\mu(x, y)$ (the coefficient of highest possible degree in $P_{x,y}$) is non-zero. In the latter list, ideally we would like to have an entry exactly when $\mu(x, y)$ is non-zero; however to do this consistently would require the computation of the full row as soon as the row is created, which appears at first sight to be rather expensive (but I think that it would be worthwile to explore this issue further.)

One of the novel features in `Coxeter3` is the fact that it will handle unequal-parameter and inverse Kazhdan–Lusztig polynomials as well as the ordinary ones. In each case, the corresponding tables are maintained (although the mu-tables in the unequal-parameter case are a bit different; one table needs to be maintained for each generator s, and the entries in the table are (Laurent) polynomials, not numbers.) All these computations reduce to the same set of extremal pairs, so the extremal-pair tables

are shared among the three contexts, and managed by the `KLSupport` class, defined in `klsupport.h`. Perhaps it is worth explaining how the extremal lists are constructed. First we construct the interval $[e, y]$ using `extractClosure`, as explained in section 6. The result is returned in the form of a bitmap. Now the required set of extremal elements is the subset of $[e, y]$ made up by those elements which are taken down by each left/right multiplication from the descent set $\mathrm{LR}(y)$. What we do is for each left or right multiplication by a generator, maintain a bitmap of the set of elements in P which are taken down (these bitmaps have to be enlarged at each extension of the enumerated part.) Then to compute the extremal elements, it is enough to intersect $[e, y]$ with the appropriate bitmaps, which is a very fast operation, and finally read the result into a list. One of the nice features of the enumerated-context setup, by the way, is the completely symmetric treatment of right and left multiplication; in fact in the program the parameter s takes values between 0 and $2n - 1$, where n is the rank. If $s < n$ we are dealing with a right multiplication, otherwise with a left multiplication; but very rarely is this distinction relevant.

In the Kazhdan–Lusztig polynomial-tables I bear the self-imposed burden of filling the row for y only if y is smaller than its inverse. Otherwise the lookup function knows that it has to go over to y^{-1} and get the polynomial from there, as $P_{x,y} = P_{x^{-1},y^{-1}}$. This does save a sizable amount of memory, but more importantly the discipline of systematically going over to y^{-1} when possible seems to drive the recursion down faster than it would otherwise go, for some reason which I don't fully understand yet. However, this setup introduces complications which mar the elegance of the program considerably, and I'm always on the verge of renouncing it. Since the mu-tables are in fact rather small (more precisely, the number of non-zero terms in them is small, so that they could in theory be made small), I have not imposed this additional constraint on them.

The approach taken by `Coxeter3` to the computation of Kazhdan–Lusztig polynomials is rather different from that of its predecessor `Coxeter1`, in that it computes the polynomials on demand, whereas `Coxeter1` would compute the full table of Kazhdan–Lusztig polynomials for the group before doing anything else. (This has the advantage that the computation can be organized with optimal efficiency; for instance only non-zero mu's need enter the picture, and a lot of searching can be avoided. So for these full-table computations, when they are possible, the old program will still be faster.) In the case of `Coxeter3`, only the row for the identity is filled at startup. Then if a polynomial $P_{x,y}$ is required, and we are not in a trivial case where the answer can be given without computation, first the extremal list for y is constructed (if it was not already available), and the row of polynomials for y is allocated and initialized with zero-pointers; then the recursion formula for $P_{x,y}$ is mapped out, the corresponding polynomials and mu-coefficients are computed if not already available, the result is found, and its address is looked up in the table of existing Kazhdan–Lusztig polynomials, adding the new polynomial if necessary; then a reference to that table entry is returned. In the course

of the computation, positivity of the coefficients is checked; a negative coefficient will cause an immediate exit with a (congratulatory) error message. The recursion I use is the original recursion formula from [6]; to my knowledge this is still the most efficient one, and will automatically take advantage of all the simplifications that I'm aware of in special cases.

If a whole row of polynomials is requested (this will be the case, for instance, when an element of the Kazhdan–Lusztig basis of the Hecke algebra of W is desired, or when one studies the singularities of the Schubert variety corresponding to y), then the computation can be done quite a bit more efficiently. In this case, all computations are done by full rows (even though this might lead to the computation of some more polynomials than strictly necessary), and also it turns out that all calls to individual Bruhat order comparisons, which take up a good deal of the computing time for individual polynomial computations, can be avoided. Even though this is elementary, it is perhaps worth explaining as I only realized it very recently. Recall that the recursion formula for $P_{x,y}$, for an $s \in S$ such that $ys < y$ and $xs < x$, amounts to adding $P_{xs,ys}$ and $qP_{x,ys}$, and then subtracting $\mu(z, ys)q^{\frac{1}{2}(l(y)-l(z))}P_{x,z}$ for all $x \leq z < ys$ such that $zs < z$. The first two terms can be gotten from the row for ys (and clearly, as x varies, most if not all of that row will be used, so that one might as well compute it all.) The set of $z < ys$ s.t. $zs < z$ can be gotten from one call to `extractClosure`, and intersection with the "downset" for right multiplication by s as explained above for the construction of extremal lists. Then we check $\mu(z, ys)$; this requires one row in the mu-table, which we get for free since it is deduced from the row in the Kazhdan–Lusztig table which we already have. If $\mu(z, ys)$ is zero, which will happen most of the time, we move on to the next z. The annoying condition is $x \leq z$. But this is avoided as follows : since we are computing a full row, we are considering *all* such x'es; so we extract $[e, z]$ again with `extractClosure`, intersect with the extremal list for y, and get the set of x'es for which we have to do the subtraction of $\mu(z, ys)q^{\frac{1}{2}(l(y)-l(z))}P_{x,z}$. Thanks to this trick which is much better than what I was doing in `Coxeter1`, some of the speed-differential between the two programs is removed.

8 How things are done: μ-coefficients

As explained in the previous section, when one is computing whole rows of polynomials, or the full table of Kazhdan–Lusztig polynomials, the *mu*-coefficients come for free : they can be read off from the corresponding polynomials, which are available when the μ's are needed. The situation is rather different when one aims for the computation of a single polynomial, and one wishes, as I try to do in this program, to compute only the strictly necessary ingredients. It would then be rather wasteful to compute a whole polynomial (and of course many others because of all the recursions that might

be triggered) when only the μ-part is required. This is even more so when we are only interested in the μ-table, for instance when we wish to determine the W-graph of the group, or its decomposition into Kazhdan–Lusztig cells.

It turns out that for the ordinary Kazhdan–Lusztig polynomials, the computation of the μ's affords some remarkable simplifications which makes it several orders of magnitude easier than the computation of the corresponding polynomials. I should confess, by the way, that I became fully aware of this fact only rather late in the construction of the program, and that I'm not quite sure I assessed it yet to the full. Just as in [6], Corollary 4.3, one sees that the for any given $x \leq y$ in W, the ordinary Kazhdan–Lusztig polynomial $P_{x,y}$ and the inverse one $Q_{x,y}$ share the same μ-coefficient. So the same simplifications hold for the inverse μ-coefficients as well. (In fact, the two computations could actually share the same μ-table, although this is not currently the case in the program.)

Here is how it goes. Let $x \leq y \in W$. When the length difference between x and y is even, we already know that $\mu(x,y) = 0$; when the length difference is one (*i.e.*, x is a coatom of y), $\mu(x,y) = 1$. So assume that $l(y) - l(x)$ is odd and > 1. If x is not extremal w.r.t. y, we also know that $\mu(x,y) = 0$; so we may also assume that x is ectremal w.r.t. y. These easy reductions, by the way, determine the default allocation of a row in the μ-table : when no further reductions are available, we allocate one entry for each x satisfying the above conditions. Now look at the recursion formula for $\mu(x,y)$ obtained by taking the term of degree $\frac{1}{2}(l(y) - l(x) - 1)$ in the recursion formula for $P_{x,y}$. We start with $\mu(xs, ys)$, then add the coefficient in degree $\frac{1}{2}(l(ys) - l(x) - 2)$ in $P_{x,ys}$ (note that the length difference between x and ys is even, so this is the highest-possible degree term in $P_{x,ys}$, but not a μ-coefficient), and subtract the sum of all $\mu(x,z)\mu(z,ys)$, where $x < z < ys$ is such that $zs < z$ (and of course we may assume that $l(z) - l(x)$ is odd.) Notice already that the only term which prevents this formula from being a recursion internal to the μ-table is the one coming from $P_{x,ys}$. But if it is the case that there is a generator t such that $yst < ys$, $xt > x$ (following our usual conventions we write products on the right, but when $t > n$ this is in fact multiplication on the left), in other words, if x is not also extremal w.r.t. ys, then we see that the term we need from $P_{x,ys}$ is in fact $\mu(xt, ys)$, and the recursion takes place entirely within the μ-table. But much more is true : following [6], section 4, one sees that in the subtracted sum there are at most two non-vanishing terms. Indeed, if $l(ys) - l(z) = 1$, then we will have $zt < z$ unless $z = yst$; and if $zt < z$ we have $\mu(x,z) = 0$ because x is then not extremal w.r.t. z. If $l(z) - l(x) = 1$, we see symmetrically that $zt > z$ unless $z = xt$; and if $zt > z$, z is not extremal w.r.t. ys. And if both length differences are > 1, we have $\mu(x,z) = 0$ if $zt < z$, and $\mu(z, ys) = 0$ if $zt > z$. Since μ-coefficients are 1 when the length difference is one, we see that the subtracted sum reduces at most to the two terms $\mu(x, yst) + \mu(xt, ys)$.

The additional condition $zs < s$ yields the final result :

$$\mu(x,y) = \begin{cases} \mu(xs,ys) & \text{if } xts < xt,\ ysts > yst \\ \mu(xs,ys) - \mu(x,yst) & \text{if } xts < xt,\ ysts < yst \\ \mu(xs,ys) + \mu(xt,ys) & \text{if } xts > xt,\ ysts > yst \\ \mu(xs,ys) + \mu(xt,ys) - \mu(x,yst) & \text{if } xts > xt,\ ysts < yst \end{cases}$$

Since x is extremal w.r.t. y, the condition $xt > x$ implies $yt > y$; but it is easy to see that $yst < ys$ is then only possible if s and t do *not* commute; in particular they correspond to multiplications which take place on the same side. If the coefficient $m(s,t)$ of the Coxeter matrix of W is equal to three, there is a further simplification. It is then easy to see that the case $ysts < yst$ cannot occur. Moreover if $xts > xt$ we have $xs < x < xt < xts$ so that xs is the shortest element in the coset of x for the parabolic subgroup of W generated by s and t, and $xst > xs$, hence xs is not extremal w.r.t. ys, and $\mu(xs,ys) = 0$. So the formulæ simplify to :

$$\mu(x,y) = \begin{cases} \mu(xs,ys) & \text{if } xts < xt \\ \mu(xt,ys) & \text{if } xts > xt \end{cases}$$

The upshot is that the full recursion formula involving the extraction of the interval $[x,ys]$ has to be called only if $\mathrm{LR}(x)$ contains not only $\mathrm{LR}(y)$, but $\mathrm{LR}(ys)$ as well for *every* $s \in \mathrm{LR}(y)$. This will happen only in a very small number of cases. On the other hand, when it does happen, the computation of $\mu(x,y)$ will trigger a great many recursive calls, so that in the end the gain is not as big as one might expect at first.

Another aspect of things is that one might use these remarks to condense the μ-table considerably, storing only the "double-extremal" pairs, particularly in the case of simply-laced groups. This would certainly be the way to go if one were to attempt the computation of, say, the full W-graph of a group like E_7. I plan to explore these issues further using a suitably modified version of the program.

9 How things are done: Kazhdan–Lusztig cells

Another feature missing from `Coxeter1` which is provided by `Coxeter3` is the computation of (left, right and two-sided) Kazhdan–Lusztig cells. This is done only for finite groups, as I don't see as yet a rigorous way of doing it for infinite groups even when, as in the case of affine groups, it is known that the set of cells is finite.

One brute-force way of computing cells is to compute the full μ-table of the group, get from there the full W-graph, and then notice that the problem is an instance of a classic computer algebra problem, *viz.* computing equivalence classes in an oriented graph. (I wish to thank Bill Casselman for pointing this out, and for explaining to me the Tarjan algorithm which performs this computation.) In fact, if in addition to the

partition of the group in cells, one wishes to recover the poset structure on the set of cells (which for finite Weyl groups is isomorphic to the poset of primitive ideals in the enveloping algebra of the corresponding semisimple Lie algebra, with trivial infinitesimal character), then it is not very likely that one can get away with much less than that. But if only the partition is required, much less needs to be done.

Here is how one might go about it. We will say that two elements x and y in W are in the same right descent class, if their right descent sets are equal. It is known already from [6] that all elements of a given left cell are in the same right descent class. Moreover, there is a set of partially defined operations on the group, the so-called $*$-operations, which preserve left cells. One operation is defined for each pair of non-commuting elements s, t in the group. The idea is to look at the right cosets for the parabolic subgroup generated by s and t. If we set $m = m(s,t)$, then each coset has $2m$ elements, one of minimal length, one of maximal length, and two in each intermediate lengths. The domain of the operation $*_{s,t}$ is the set of elements in W which are neither of minimal nor of maximal length in their coset (equivalently, this means that $R(w) \cap \{s,t\}$ has exactly one element.) For each coset C in W with minimal length representative $x_{\min}$ we define the two $\{s,t\}$-chains of C to be the $(m-1)$-element sets $C_s = \{x_{\min}s, x_{\min}st, \ldots\}$ and $C_t = \{x_{\min}t, x_{\min}ts, \ldots\}$. Then each element w in the domain of $*_{s,t}$ is contained in exactly one $\{s,t\}$-chain $\{x_1, \ldots, x_{m-1}\}$. If $j \in \{1, \ldots, m-1\}$ is the index such that $w = x_j$, we set $w*_{s,t} = x_{m-j}$. In particular, $*_{s,t}$ permutes each $\{s,t\}$-chain. Clearly the domain of each $*_{s,t}$ is a union of right descent classes, and therefore of left cells. Then it is known [7] that $*_{s,t}$ takes each left cell in its domain to another left cell.

This allows us to refine the partition of W in right descent classes as follows. We define a sequence of partitions $(\mathcal{R}_k)_{k\geq 1}$ of W by letting $\mathcal{R}_1$ be the partition in right descent classes, and saying that x and y have same class in $\mathcal{R}_k$, $k > 1$, if and only if they belong to the same class in $\mathcal{R}_{k-1}$, and for each pair $\{s,t\}$ of non-commuting generators in the group such that $*_{s,t}$ is defined for x (and hence for y), $x*_{s,t}$ and $y*_{s,t}$ also belong to the same class in $\mathcal{R}_{k-1}$. We then define $\mathcal{R}_\infty$ by saying that x and y belong to the same class in $\mathcal{R}_\infty$ if and only if they belong to the same class in $\mathcal{R}_k$ for all $k \geq 1$. Clearly the partition of W in left cells refines the partition $\mathcal{R}_\infty$; in the case of a finite Weyl group, the partition $\mathcal{R}_\infty$ is the generalized τ-partition introduced by David Vogan [8].

It turns out that it is not hard to compute the partition $\mathcal{R}_\infty$, by an algorithm rather similar to the one which will construct the minimal automaton corresponding to any given finite state automaton (see for instance [1] algorithm 3.6.) On the other hand, there is also an easy lower approximation to the partition of W in left cells. Indeed, it is easy to see that each left cell C contains a whole left $\{s,t\}$-string as soon as it contains one of its elements. Denote $\mathcal{S}$ the smallest equivalence relation on W which is compatible with $\{s,t\}$-strings for all non-commutaing pairs $\{s,t\}$. Again it is not hard to compute the partition $\mathcal{S}$.

Our approach to left cells is to first compute the partitions $\mathcal{S}$ and $\mathcal{R}_\infty$; the former

is a refinement of the latter. Any class for $\mathcal{R}_\infty$ which is also a class for $\mathcal{S}$ must be a left cell; for the remaining $\mathcal{R}_\infty$ classes we compute the W-graph of the class and get the left cells from there. It is known ([6] section 5) that for type A the partitions $\mathcal{R}_\infty$ and $\mathcal{S}$ coincide; this seems also to be always the case in type B. So in these cases the cell partition can be determined without any Kazhdan–Lusztig computations whatsoever!

Right cells can of course be deduced from left cells by inversion. I have to confess that I haven't found the time to study up on two-sided cells as much as I would have liked; currently their determination is implemented really brute force-like from the full two-sided W-graph of the group. This is certainly a point which should be improved in the future. Another thing that is currently missing from the program is the computation of Lusztig's a-function. Hmm...

We now describe the member functions of the `FiniteCoxGroup` class which are available for accessing the various partitions defined in this section. Only the actual Kazhdan–Lusztig cells have a corresponding command in the default command interface, and can be printed out in various formats. The standard way of representing a partition of a set in p classes, is through a function taking values in the set $\{0, \ldots, p-1\}$; and a general function on a set with N elements is just a sequence of N numbers. The various partition functions all return references to objects of type `Partition`, defined in `bits.h`; in addition to the actual partition function, this class contains the number of classes of the partition, and a number of member functions for sorting and traversing partitioned sets conveniently. Since some of these partitions may be rather expensive to compute, we avoid computing them more than once; the finite group structure contains predefined partition objects, initially empty, which are filled when first required. So after the first call, access should be instantaneous.

- `lCell, rCell, lrCell` : the partitions in ordinary Kazhdan–Lusztig cells (these partitions may be printed out using the commands `lcells`, `rcells`, `lrcells` from the default command interface);
- `lUneqCell, rUneqCell, lrUneqCell` : the partitions in Kazhdan–Lusztig cells for unequal parameters (these partitions may be printed out using the commands `lcells`, `rcells`, `lrcells` as above, after passing to "unequal parameter mode" using the command `uneq`);
- `lDescentPartition, rDescentPartition` : the partitions in left and right descent classes;
- `lGeneralizedTau, rGeneralizedTau` : the partitions in equivalence classes for the generalized tau-invariant (the relation $\mathcal{R}_\infty$);
- `lStringPartition, rStringPartition` : the partitions for the equivalence relation $\mathcal{S}$ defined above.

References

[1] A.V. Aho, R. Seti and J.D. Ullman. *Compilers*. Addison-Wesley, Reading, Massachussets, 1986.

[2] B. Brink. The set of dominance-minimal roots. *J. Algebra*, **206**:371–412, 1998.

[3] B. Brink and D. Howlett. A finiteness property and an automatic structure for Coxeter groups. *Math. Ann.*, **296**:179–190, 1993.

[4] W. Casselman. Computation in Coxeter groups. I. Multiplication. *Electron. J. Combin.*, **9**(1):Research Paper 25, 22 pages, 2002.

[5] F. du Cloux. Computing Kazhdan-Lusztig polynomials in arbitrary Coxeter groups. *Experiment. Math.*, **11**(3):387–397, 2002.

[6] D. Kazhdan and G. Lusztig. Representations of Coxeter groups and Hecke algebras. *Invent. Math.*, **53**:165–184, 1979.

[7] G. Lusztig. Cells in Affine Weyl Groups. In *Algebraic Groups and relaed topics (Kyoto/Nagoya 1983)*, volume 6 of *Adv. Stud. Pure Math.*, pages 255–287, Amsterdam, 1985. North-Holland.

[8] D.A. Vogan Jr. A generalized τ-invariant for the primitive spectrum of a semisimple Lie algebra. *Math. Ann.*, **242**:209–224, 1979.