Computing Kazhdan-Lusztig Polynomials for $\tilde{A}_8$

Tim Sprowl
University of Virginia

Abstract

We examine ways to efficiently compute Kazhdan-Lusztig polynomials for the affine Weyl group $\tilde{A}_8$. After concluding that the existing algorithm requires too much memory, we have created an algorithm that computes high degree Kazhdan-Lustzig polynomials using less than two gigabytes of memory as opposed to the terabytes required by the existing algorithm. This algorithm can be applied to the computation of KL polynomials for other groups as well.

1 Introduction

In theoretical mathematics, representation theory provides a key way to reason about abstract structures from other areas of mathematics, and it also provides tools to reason about the physical world, especially in physics and communication theory. An important component in studying many representations is the theory and computation of Kazhdan-Lusztig (KL) polynomials. These polynomials provide insight into the structure of Hecke algebras, which in turn are used to parameterize information about a vast array of basic building blocks of representations that occur in nature.

One example where KL polynomials have been used prominently is in the disproof of Wall's conjecture (Wall, 1961; Liebeck & Shalev, 2007), which states that the number of maximal subgroups of any finite group is at most the number of elements in the group. In terms of Kazhdan-Lusztig theory, this implies that $\mu(\nu, \gamma) \leq n^2 - 1$ for $\tilde{A}_{n-1}$, where $\mu(\nu, \gamma)$ corresponds to a certain coefficient in a KL polynomial. While this is now known to not be the case, there still are questions about the growth rate of this particular coefficient such as if it is bounded by any other polynomial (Guralnick et al., 2012). Efforts to answer questions like this can be greatly aided by computing more KL polynomials.

Unfortunately, for large cases, computing KL polynomials is difficult. The standard approach has been to compute KL polynomials essentially "in order." That is, in order to obtain more interesting, higher degree polynomials, one must compute essentially all lower degree ones. We created an efficient serial implementation of this algorithm in order to try to compute the KL polynomials for $\tilde{A}_8$. Ultimately, we determined that scaling this algorithm for $\tilde{A}_8$ requires too much data to be stored in main memory (on the order of terabytes) even when considering sophisticated implementations.

Consequently, we created a new algorithm to compute KL polynomials that avoids calculating all polynomials and instead only calculates specific, large polynomials of interest. It

works by keeping intermediate forms of a small amount of KL polynomials in main memory, and then performing operations on these modified KL polynomials by subtracting out other modified KL polynomials so that they become the real KL polynomials. This approach uses less than two gigabytes of memory, and thus is a substantial improvement over the existing algorithm for calculating high degree polynomials.

2 Existing Algorithm

Before examining the algorithm itself, it is necessary to introduce some terminology. KL polynomials are indexed by two elements in a Coxeter group.[1] A Coxeter group is a group with given generators, subject to certain relations. For a given element ω of the group, the smallest product of the group's generators equal to this element is called its length $\ell(\omega)$. There is also an order relation $\leq$ on some pairs of elements, defined in terms of their expressions of smallest length. We need not explicitly discuss this order relation, other than noting it is compatible with the length function. That is, $\nu \leq \gamma$ implies $\ell(\nu) \leq \ell(\gamma)$. In our case, we are specifically considering the affine Weyl group $\tilde{A}_{n-1} = \langle s_0, s_1, s_2, ..., s_{n-1} \rangle$ for $n = 9$ (that is $\tilde{A}_8$) with the relations that:

$$\begin{aligned}
(s_i s_j)^2 &= 1 && \text{if } i \neq j+1 \mod n \text{ and } i \neq j-1 \mod n \\
(s_i s_j)^3 &= 1 && \text{if } i = j+1 \mod n \text{ or } i = j-1 \mod n \\
s_i s_i &= 1 && \text{for all } i
\end{aligned}$$

The elements of $\tilde{A}_8$ are called Weyl strings. One example is $w_0 w = s_1 s_2 s_1 s_3 s_2 s_1 s_4 s_1$, where $w_0 = s_1 s_2 s_1 s_3 s_2 s_1$ and $w = s_4 s_1$, the weyl string for $(2, 1, 0)$ for $n = 4$. The length of the weyl string is the number of s_i in the string, assuming that the string is "reduced" (of smallest length). For the previous example, the string is known to be reduced and so the length is 7, and is denoted $\ell(w_0 w)$. A Weyl string is reduced if, using the relations defined above, one cannot make a shorter Weyl string. Thus, the Weyl string $w_0 w = w_0 s_4 s_1 s_1$ is not reduced since $s_1 s_1 = 1$. In this case, the reduced Weyl string is actually $w_0 s_4$ with length 6. In general, when considering the Weyl string formed from appending another s_i term, the length can increase by one or decrease by one.[2]

A Weyl string also corresponds to a weight. We write $\gamma = w_0 w \cdot -2\rho$ to formally state the relationship between the Weyl string $w_0 w$ and weight γ. A weight is a $(n-1)$-tuple composed of nonnegative integers. For example $(0, 0, 0)$, $(1, 0, 1)$, and $(2, 1, 0)$ are all examples of weights for $n = 4$. Each weight has an associated length. For the previous examples, their lengths are 6, 7, and 8 respectively. There is not an obvious relationship between a weight and its length, and in order to compute the length for a given weight, we must consider the associated Weyl string. For the remainder of this article, we will use the weight notation over the Weyl string notation and use ν, γ, and ξ to denote weights.

[1] Actually, we are interested in parabolic KL polynomials that are indexed by two elements in the coset space formed by a Coxeter group and one of its subgroups (Deodhar, 1987) but we will ignore this distinction to simplify this exposition.

[2] In our parabolic situation, we have to effectively consider a third case in which the appended string represents the same coset as the original string. We informally think of this by saying the length stays the same.

$\ell(\gamma)$	γ	ν							
		(0,0,0)	(1,0,1)	(2,1,0)	(0,1,2)	(4,0,0)	(1,2,1)	(0,0,4)	(2,2,2)
6	(0,0,0)	1	0	0	0	0	0	0	0
7	(1,0,1)	1	1	0	0	0	0	0	0
8	(2,1,0)	1	1	1	0	0	0	0	0
8	(0,1,2)	1	1	0	1	0	0	0	0
9	(4,0,0)	1	1	1	0	1	0	0	0
9	(1,2,1)	t^2+1	1	1	1	0	1	0	0
9	(0,0,4)	1	1	0	1	0	0	1	0
10	(2,2,2)	$2t^2+1$	t^2+1	1	1	1	1	0	1

KL polynomials are computed by manipulating other KL polynomials recursively based off Hecke algebra relations, starting with the knowledge that a KL polynomial P, indexed by two elements $\nu, \gamma \in \tilde{A}_8$, has the property that $P_{\nu,\gamma} = 1$ if $\nu = \gamma$ and $P_{\nu,\gamma} = 0$ if $\ell(\nu) \geq \ell(\gamma)$. An example of KL polynomials computed for the case $n = 4$ is below, where each row of the table represents all KL polynomials for a certain γ and each column of the table represents all KL polynomials for a certain ν. The following algorithm is the recursive definition of a KL polynomial in terms of other KL polynomials (Kazhdan & Lusztig, 1979; Deodhar, 1987).

Fix ν and γ as two elements in $\tilde{A}_8$. Let $s = s_i$ for $0 \leq i < 9$ and let $\ell(\gamma s) = \ell(\gamma) + 1$:

$$\begin{aligned} P_{\nu,\gamma s}(t^2) &= P_{\nu,\gamma}(t^2) + t^2 P_{\nu s,\gamma} - \Sigma && \text{(if } \ell(\nu s) > \ell(\nu)\text{)} \\ P_{\nu,\gamma s}(t^2) &= t^2 P_{\nu,\gamma}(t^2) + P_{\nu s,\gamma} - \Sigma && \text{(if } \ell(\nu s) < \ell(\nu)\text{)} \\ P_{\nu,\gamma s}(t^2) &= (1+t^2) P_{\nu,\gamma}(t^2) - \Sigma && \text{(if } \ell(\nu s) = \ell(\nu)\text{)} \end{aligned}$$

$$\Sigma = \sum_{\substack{\ell(\nu) \leq \ell(\xi) < \ell(\gamma) \\ \ell(\xi s) \leq \ell(\xi)}} \mu(\xi, \gamma) t^{\ell(\gamma) - \ell(\xi) + 1} P_{\nu,\xi}(t^2)$$

$\mu(\xi, \gamma)$ denotes the coefficient of $t^{\ell(\gamma)-\ell(\xi)-1}$ in $P_{\xi,\gamma}$

The Σ term is the most complicated part of the formula and dominates the memory requirements of computing the polynomials since it requires that essentially all polynomials be kept in memory. This can be observed by noticing that for all other parts of the formula we only need polynomials for $P_{\nu s,\gamma}$ and $P_{\nu,\gamma}$, and so only polynomials of a certain γ need to be kept in main memory for that part of the computation. In terms of the table of KL polynomials, this means that a small subset of the rows must be kept in memory. This is opposed to $P_{\nu,\xi}$, where ξ can be any element in $\tilde{A}_8$, and hence essentially every row of the table of KL polynomials must be stored in memory.

In general, we are most interested in polynomials where $\ell(\gamma)$ is large, as this provides the best characterizations of the currently unknown properties of KL polynomials for this affine Weyl group. However, using the above algorithm, we must calculate every polynomial between our initial starting values and these larger polynomials of interest, which is computationally intensive.

Specifically, for $n = 9$, there are approximately $1,700,000$ elements in $\tilde{A}_8$ that have

a length less than or equal to $\gamma = (6, 7, 7, 7, 7, 7, 7, 6)$, which is the predetermined maximal weight that we consider for our calculations. There is a KL polynomial for every combination of two of these elements, which means we are interested in potentially $(1.7 \cdot 10^6)^2 = 2.89 \cdot 10^{12}$ polynomials. Since we are only really concerned with polynomials where $\nu < \gamma$ (otherwise they are trivially computed), this number is effectively halved so we really only have to consider $1.45 \cdot 10^{12}$ polynomials. Still, if we assume that each polynomial takes eight bytes of memory (or 64 bits) then we are dealing with approximately 10.55 terabytes of data, which is larger than most hard drives, let alone main memory.

One major serial optimization is to store a pointer to a KL polynomial rather than the data for an actual KL polynomial. This saves a substantial amount of memory because there is a lot of repetition in KL polynomials. For example, in the $n = 4$ case above, the polynomials 0, 1, and $t^2 + 1$ are all used multiple times. Hence, the previous estimate of using eight bytes to store a KL polynomial is not unreasonable as this is the size of a pointer on a 64-bit machine. However, this necessitates the use of an auxiliary data structure that keeps track of what polynomials have been generated, which slows down the computation and impedes parallelization.

Since it is virtually impossible for a naive implementation to work, we considered trying to parallelize this computation. However, this is difficult because of the complex data sharing relationships required to calculate all KL polynomials for a particular γ. The rows of the KL table would need to be stored on separate nodes, with other nodes needing to be able to retrieve the rows when necessary for the computation. Our calculations for $\tilde{A}_7$, which is a smaller case that only required 100 gigabytes of memory, took on the order of days, and it was feared that once we introduced more overhead in retrieving information from other nodes, the calculation would be far too slow and would outweigh any gains in execution time from parallelization.

Another approach is to store only a fraction of the KL polynomials and recompute the others as necessary. This obviously trades off execution time for memory, and it is difficult to quantify just how much slower the computation would be if only a fraction of the rows of the KL table are stored. However, it is conceivable that recomputation, coupled with parallelization, could allow $n = 9$ to be computed using the existing algorithm, albeit slowly. Furthermore, regardless of the optimizations used, the existing algorithm still requires computers that have as much memory as possible to minimize the amount of time spent fetching or recomputing KL polynomials. Fortunately, our algorithm uses much less memory and so does not have these limitations.

3 Our Algorithm

Once again, we need to introduce some terminology before examining the algorithm. Weights and Weyl strings are used to index T's. We write T_γ to denote that T has index γ. Different T's are indexed by different weights. The coefficient of a T is a polynomial in terms of t. There are special rules for multiplying T's based off their respective weights. These are:

$$
\begin{aligned}
T_\nu T_{s_i} &= T_{\nu s_i} && (\ell(\nu s_i) > \ell(\nu)) \\
T_\nu T_{s_i} &= (t^2-1)T_\nu + t^2 T_{\nu s_i} && (\ell(\nu s_i) > \ell(\nu)) \\
T_\nu T_{s_i} &= t^2 T_\nu && (\ell(\nu s_i) = \ell(\nu))
\end{aligned}
$$

Similarly, C's are a linear combination of T's. For example, $C_{s_i} = (1/t)(1+T_{s_i})$. Since C_{s_i} is just a linear combination of T's. We can modify the above rules for multiplication.

$$
\begin{aligned}
T_\nu C_{s_i} &= (1/t)(T_{\nu s_i} + T_\nu) && (\ell(\nu s_i) > \ell(\nu)) \\
T_\nu C_{s_i} &= t(T_{\nu s_i} + T_\nu) && (\ell(\nu s_i) > \ell(\nu)) \\
T_\nu C_{s_i} &= (t+t^{-1})T_\nu && (\ell(\nu s_i) = \ell(\nu))
\end{aligned}
$$

The coefficients of the T terms in C are the KL polynomials. Specifically $P_{\nu,\gamma}$ is the coefficient of T_ν in C_γ.

The idea of the algorithm is to find C_γ. First, we start with $C_{w_0} = T_{w_0}$ and compute the product $FatC_\gamma = C_{w_0}C_{s_i}C_{s_j}...C_{s_k}$ by converting to the T notation and multiplying as one would normally. Then , we look for a coefficient of a nonnegative power of t in $FatC_\gamma$ for the term $T_\nu/t^{\ell(\nu)-\ell(w_0)}$ where $\ell(\nu) < \ell(\gamma)$. If none are found, then the algorithm is finished and $C_\gamma = FatC_\gamma$. If one is found, we focus on largest $\nu < \gamma$ for which it is found. Let $f(t)$ be the coefficient of $T_\nu/t^{\ell(\nu)-\ell(w_0)}$.

By assumption, $f(t) = f_{\geq 0}(t) + f_{<0}(t)$. $f_{\geq 0}(t)$ is a linear combination of nonnegative powers of t, and $f_{<0}(t)$ is a linear combination of negative powers of t. We can write $f_{\geq 0}(t) = z + f_{>0}(t)$, where z is an integer and $f_{>0}(t)$ is a polynomial with only positive powers. Then, we compute $FatC'_\gamma = FatC_\gamma - g(t)FatC_\nu$ where $g(t) = f_{>0}(t) + f_{>0}(t^{-1}) + z$. Note that $FatC_\nu$ has to be calculated by using the techniques above. Lastly, $FatC_\gamma = FatC'_\gamma$.

3.1 Example

As an example, consider calculating $FatC_\gamma$ where $\gamma = w_0 s_4 s_1$ for $n = 4$. $FatC_\gamma$ equals:

$$C_{w_0}C_{s_4}C_{s_1} = ((1/t)(T_{w_0 s_4} + T_{w_0})C_{s_1} = (1/t^2)(T_{w_0 s_4 s_1} + T_{w_0 s_4}) + (1/t)(t+t^{-1})T_{w_0}$$

Now, we need to transform the $FatC_\gamma$ into C_γ. First, we need to rewrite each term as $T_\nu/t^{\ell(\nu)-\ell(w_0)}$. Thus:

$$
\begin{aligned}
C_{w_0}C_{s_4}C_{s_1} &= (1/t^2)(T_{w_0 s_4 s_1} + T_{w_0 s_4}) + (1/t)(t+t^{-1})T_{w_0} \\
&= (1)(T_{w_0 s_4 s_1}/t^2) + (t^{-1})(T_{w_0 s_4}/t) + (t^0 + t^{-2})(T_{w_0}/t^0)
\end{aligned}
$$

We look at the largest $\nu < \gamma$ so we only need to look at the last term, which means $\nu = w_0$. which has the nonnegative power t^0, so $g(t) = t^0 = 1$. We have to now calculate $FatC_\nu$. In this case, it is easy since $\nu = w_0$ and so $FatC_\nu = T_{w_0}$. We now have everything we need to calculate $FatC'_\gamma$

$$
\begin{aligned}
FatC'_\gamma &= FatC_\gamma - g(t)FatC_\nu \\
&= (1)(T_{w_0 s_4 s_1}/t^2) + (t^{-1})(T_{w_0 s_4}/t) + (t^{-2})(T_{w_0}/t^0)
\end{aligned}
$$

Seeing there is no more terms other than T_γ that has a positive degree of t we can conclude that $FatC'_\gamma = C_\gamma$.

4 Results

For a particular large $\gamma = (6, 7, 7, 7, 7, 7, 7, 6)$ for $n = 9$, our implementation of our algorithm took 15 days on a 2.2 gigahertz processor and required around 1.2 gigabytes of memory to compute the KL polynomials $P_{\nu,\gamma}$ where $\nu < \gamma$. This shows that high degree polynomials can be computed for $\tilde{A}_8$, and all that is required is an average computer, where as the existing algorithm requires computers with a very large amount of memory so that as many KL polynomials can be fitted into memory as possible. Furthermore, our algorithm can be extended to other groups since there is nothing in our algorithm that takes special advantage of the properties of $\tilde{A}_8$. In fact, our implementation precomputes the group relations before execution, so all that would be required to compute KL polynomials for other groups is changing the input file that records the group relations.

In summary, computing KL polynomials is difficult. The current approach requires computing KL polynomials essentially "in order." Thus, in order to obtain more interesting, higher degree polynomials, one must compute essentially all lower degree ones. Consequently, we created a new algorithm to compute KL polynomials that avoids calculating all polynomials and instead only calculates specific, large polynomials of interest. This algorithm requires less than two gigabytes of memory where the existing algorithm requires terabytes of memory and so represents a dramatic improvement over the status quo.

Bibliography

Deodhar, V. (1987). *On some geometric aspects of Bruhat orderings. II. The parabolic analogue of Kazhdan-Lusztig polynomials.* Journal of Algebra, 111(2), 483-506.

Kazhdan, D., & Lusztig, G. (1979). *Representations of coxeter groups and hecke algebras.* Inventiones Mathematicae, 53(2), 165-184.

Liebeck, M. & Shalev, A. (2007). *On a conjecture of G.E. Wall.* Journal of Algebra, 317, 184-197.

Wall, G. (1961). *Some applications of the Eulerian functions of a finite group.* Journal of the Australian Math Society, 2, 35-59.

Guralnick, R., Hodge, T., Parshall B., & Scott, L. (2012). *AIM workshop counterexample to Wall's conjecture.* Retrieved from http://aimath.org/news/wallsconjecture/wall.conjecture.pdf.