DOCUMENTATION OF MAPLE SCRIPTS FOR A COMBINATORIAL PROOF THAT SCHUBERT VS. SCHUR COEFFICIENTS ARE NONNEGATIVE

FRANK SOTTILE

Abstract. We document the maple code that Frank Sottile wrote to verify the main theorem in our paper. The basic method to generate chains was based on a suggestion of Nantel Bergeron, but otherwise it is original.

This code has four parts, which should be run in order. Each needs an internal variable `n` to be set to one of 2, 3, 4, 5, or 6, and relies on the output of the previous computation.

(1) `makeClasses.maple` generates all flattened chains of a given length, sorting them into equivalence classes as in [2], where each equivalence class consists of all chains in some interval in the Grassmannian-Bruhat order. These classes are stored in `Equivalence_Classes/Classes.n`, which is a Maple-readable file defining tables, `Equivalence_Classes[n][k]`, where `n` is the length of the words and `k` is the number of distinct indices which appear in a word.

It also starts to write files `statistics.n`, which records some information of the computation, and is updated by each of the scripts below.

(2) `makeGraphs.maple` reads `Equivalence_Classes/Classes.n` and decomposes each equivalence class into connected dual equivalence graphs (using the `n`-2 involutions at positions $2,\dots,$`n`-1 which are described below). Equivalence classes of chains decompose as not all transformations among chains in the Grassmannian-Bruhat order are edges in the dual equivalence graphs. This script creates files `Graphs.n/g.c` which contain all graphs with `c` vertices, each a chain of length `n` in the Grassmannian-Bruhat order.

(3) `Check_symmetric.maple` verifies that the quasisymmetric function of these graphs are symmetric. It needs a file `symmetric/s.n` that, for each partition λ of `n`, records the descents in the quasi-symmetric expansion of the Schur function s_λ. It reads all the files in `Graphs.n/g.c`, and for each dual equivalence graph verifying that its quasisymmetric function is symmetric and Schur-positive.

(4) `graphIsomorphism` reads the files `Graphs.n/g.c` and determines the different isomorphism classes of dual equivalence graphs among all our graphs. It creates files `Automorphism_Types.n/at.c` which contain one dual equivalence graph of each isomorphism type. It records the symmetric function of each isomorphism type.

Some statistics from the computation are presented in Tables 1 and 2. There, the columns are the different values $2, 3, 4, 5,$ and 6 of `n`.

Chains in the Grassmannian-Bruhat order. We recall the description of all chains in the Grassmannian-Bruhat order from [2]. We let $a, b, \dots$ be integers. Covers in the

Table 1. Time spent in computation

	2	3	4	5	6
Classes	.03	.04	.61	240	343123
Graphs	.02	.08	.28	12.9	2206
Symmetric	.04	.11	.75	11.6	290
Isomorphism	.04	.05	.25	13.4	2979

Table 2. Numbers

	2	3	4	5	6
Chains	6	70	1236	29400	881934
Graphs	6	46	499	5948	82294
Symmetric Functions	2	3	7	14	62
Isomorphism Classes of Graphs	2	3	7	27	178

Grassmannian-Bruhat order are transpositions $t_{a,b}$ with $a < b$ in the symmetric group. We use the symbol $\mathbf{u}_{a,b}$ to represent such a cover in what follows. This is the notation used in [2] (but t_{ab} is used in [1]). The main results of [2] are two-fold. It identifies which words in the symbols $\mathbf{u}_{a,b}$ occur in chains in some Grassmannian-Bruhat order, and it gives a set of transformations on such words which generate all the chains in a given interval. Both of these are accomplished by relations, which either declare that a subword is not allowed (equivalent to the absorbing element, 0) or that allow the substitution of one subword for another.

These relations are compactly presented in [2], and they depend only upon the relative order of the indices $a, b, c, \ldots$. It follows that we need only to study chains in which the indices form an initial segment $\{1, 2, \ldots, k\}$ of $\mathbb{N}$. Given a chain ω in the Grassmannian-Bruhat order, there is a unique order-preserving bijection between its indices and an initial segment of $\mathbb{N}$. Replacing the indices of ω by their images under this bijection gives the *flattening* $\overline{\omega}$ of ω. This flattening induces an equivalence relation on chains ω in the Grassmannian-Bruhat order which preserves the structure of intervals, so it suffices to study and classify chains up to flattening.

The set of all flattened chains forms a tree as follows. Given a chain $\mathbf{u}_{a,b}\mathbf{u}_{c,d}\cdots\mathbf{u}_{e,f} = \mathbf{u}_{a,b}\cdot\omega$ of length $n+1$ in the Grassmannian-Bruhat order whose indices $\{a, b, c, d, \ldots, e, f\}$ form an initial segment in $\mathbb{N}$, we remove the first term $\mathbf{u}_{a,b}$ to get a shorter chain ω in the Grassmannian-Bruhat order. The indices of ω may not necessarily form an initial segment in $\mathbb{N}$, so we flatten it to obtain such a chain, $\overline{\omega}$. In this case, we say that $\mathbf{u}_{a,b}\cdot\omega$ is a child of $\overline{\omega}$, and this give the set of all flattened chains the structure of a tree.

Generating chains. We follow this tree to generate all flattened chains up to length 6 (the limit of feasibility with this maple code). This is done by `makeClasses.maple`, which needs its internal variable `n` set to one of 2, 3, 4, 5, or 6. We must initialize the file `Equivalence_Classes/Classes.1` which has the unique flattened chain $\mathbf{u}_{1,2}$ of length one, stored in the list `Equivalence_Classes[1][2]`, where `1` is the value of `n`-1 and `2` records that the flattening uses the numbers $\{1,2\}$. This file also contains a list `numberLetters` consisting of 2.

`makeClasses.maple` first reads `Equivalence_Classes/Classes.(n-1)`, which contains lists of equivalence classes of chains of length `n`, sorted by the size, `k`, of their flattening, and a separate list containing the different sizes of flattenings. It seeks to prolong these chains in all possible ways (with a savings when the size of the flattening is `2n`.)

Given a flattened chain ω of length `n`-1, it first constructs all flattened words $\mathbf{u}_{a,b}w$ with $\overline{w}=\omega$, and then tests to see if $\mathbf{u}_{a,b}w$ is a chain in the Grassmannian-Bruhat order. If the indices in ω form the initial segment $\{1,\ldots,k\}$ in $\mathbb{N}$, then there are $2k+1$ different possibilities for a:

$$a<1 \text{ or } a=1 \text{ or } 1<a<2 \text{ or } a=2 \text{ or } \cdots \text{ or } a=k \text{ or } k<a\,.$$

For the k choices of $a=i$, we set $w:=\omega$ alone, but when $a\neq i$ for any i, which requires that a is inserted either between two indices of ω or else before or after all indices of ω, we alter the indices of ω (to obtain w) so that they together with a form the initial segment $\{1,\ldots,k{+}1\}$ of $\mathbb{N}$. There are similarly $2(l-a)+1$ choices of b, as well as further potential alterations of the indices of w, where l is either k or $k{+}1$ depending upon the choice of a.

Given a putative chain $\mathbf{u}_{a,b}w$ created in this way whose indices are $\{1,\ldots,\texttt{l}\}$, the code first checks to see if it has already found this chain. If not, then it calls the procedure `testEquivZero`, which checks if $\mathbf{u}_{a,b}w$ is a legitimate chain in the Grassmannian-Bruhat order, and if so returns its equivalence class. When $\mathbf{u}_{a,b}w$ is a legitimate chain, it is new, as are all chains in its equivalence class. These are added to the list of lists `New_Equivalence_Classes[l]`.

There are three additional speed-ups. First, `New_Equivalence_Classes[l]` is a table of lists containing all flattened chains w whose indices are $\{1,2,\ldots,\texttt{l}\}$. This helps to reduce the time spent searching to see if a chain $\mathbf{u}_{a,b}w$ has already been encountered. Additionally, whenever it is discovered in `testEquivZero` that $\mathbf{u}_{a,b}w\sim 0$, the procedure immediately exits with a value of `false`. Lastly, we can directly generate all flattened chains of length `n` with `2n` letters. There are $C_n\cdot n!$ of them (C_n is the nth Catalan number) so when $n=6$, we discard chains whose indices are $\{1,2,\ldots,12\}$, for these are either equivalent to 0 or are one of the $132\cdot 6! = 95040$ that we already know. When $\texttt{n}=5$, these speed-ups reduce the total computation time by 70%.

The data of this calculation are stored in files `Equivalence_Classes/Classes.n`.

Testing if equivalent to zero. The paper [2] studies chains in the Grassmannian-Bruhat order, which are certain words $\omega=\mathbf{u}_{a,b}\cdots\mathbf{u}_{c,d}$ in the free monoid on the symbols $\mathbf{u}_{a,b}$ with $a<b$. A word is null (or 0) if it does not come from a chain in the Grassmannian-Bruhat order, otherwise all the words of chains in an interval in the Grassmannian-Bruhat order are equivalent. The content of [2] is that this equivalence relation is given by five simple rules defined on subwords of lengths 2 and 3. Three explain how to transform one word into another and two describe when a subchain is equivalent to zero (is null).

The procedure `testEquivZero` is a wrapper for `makeAllChains`. Given a word w in these symbols, `makeAllChains` first tests to see if a subword of w is null. If so, it terminates with a boolean `false`, otherwise it recursively generates all words equivalent to w, testing if each newword has a null subword. The recursively called procedure is `makeChainStep`, which has two arguments `Recent` and `All`. The first, `Recent` contains words equivalent to w that have not had relations applied to them and `All` contains all words equivalent to w that have been generated. Initially, `Recent`=`All`=$[w]$. The procedure `makeChainStep` first creates a list `New` which will contain new words generated in this recursive call. It then loops over the words in `Recent`, and for each, it tries to apply the two- and three-term relations from [2]. Whenever it can apply such a relation, the new word w' is tested to see if it has a null subword, and if it does not, then it tests if w' is a member of `All`. If it is not, then w' is new, and it is added to `New`, and to `All`.

If after processing all words in `Recent` there have been no new words found, and no words were encountered that were null, then all equivalent words have been found and the original words was not equivalent to zero, so `makeChainStep` terminates, passing this information (including all the chains equivalent to the original word) back to `makeAllChains` and then to `testEquivZero`. If not, then `makeChainStep` calls itself, replacing `Recent` with `New`.

Implementation of the relations.
Subchains equivalent to zero. These involve either two adjacent symbols or three.

$$\begin{array}{rrcll} \textsf{Two terms} & \mathbf{u}_{a,b}\mathbf{u}_{c,d} & \equiv & 0 & \text{if } a \leq c < b \leq d \text{ or } c \leq a < d \leq b \\ \textsf{Three terms} & \mathbf{u}_{a,b}\mathbf{u}_{c,d}\mathbf{u}_{e,f} & \equiv & 0 & \text{if } |\{a,b,c,d,e,f\}| = 3 \end{array}$$

To compare this formulation to that in [2], observe that the first line is simply relation (4) on page 2 of [2]. For the second line, suppose that $\{a,b,c,d,e,f\} = \{1,2,3\}$, and consider words of length three composed of $\mathbf{u}_{1,2}$, $\mathbf{u}_{1,3}$, and $\mathbf{u}_{2,3}$. If the initial or final subwords of length 2 are not already equivalent to zero, there are only two possibilities,

$$\mathbf{u}_{1,2}\mathbf{u}_{2,3}\mathbf{u}_{1,2} \qquad\qquad \mathbf{u}_{2,3}\mathbf{u}_{1,2}\mathbf{u}_{2,3}\,,$$

which are both equivalent to zero, by (5) on page 2 of [2].

These two relations are coded in the procedures `testZeroII` and `testZeroIII` in the code. The routine `testEasyZero` performs these tests on all consecutive subwords of lengths 2 and 3 of a given word. Whenever a word is generated, either from prolonging an existing word, or by applying a valid commutation relations, the routine `testEasyZero` is called to see if the word is equivalent to zero.

Commutation relations. If a word is not null, and $\mathbf{u}_{a,b}\mathbf{u}_{c,d}$ is a subword, then either $\{a,b,c,d\} = 3$ and there is not a relation to apply (it has the form $\mathbf{u}_{1,2}\mathbf{u}_{2,3}$ or else $\mathbf{u}_{2,3}\mathbf{u}_{1,2}$), or else $\{a,b,c,d\} = 4$, and $\mathbf{u}_{a,b}\mathbf{u}_{c,d} \equiv \mathbf{u}_{c,d}\mathbf{u}_{a,b}$. Since the subword is not null, this has one of the two forms

$$\mathbf{u}_{1,2}\mathbf{u}_{3,4} \;\equiv\; \mathbf{u}_{3,4}\mathbf{u}_{1,2} \qquad \text{or} \qquad \mathbf{u}_{1,4}\mathbf{u}_{2,3} \;\equiv\; \mathbf{u}_{2,3}\mathbf{u}_{1,4}\,.$$

This is hard-coded into a segment of `makeChainStep`.

Transformations. The remaining two types of relations transform a subword of length three into a different one, and these are hard-coded into a segment of `makeChainStep`, as the subroutine `threeTerm`. For these, we have a subword of length three, $\mathbf{u}_{a,b}\mathbf{u}_{c,d}\mathbf{u}_{e,f}$, where $|\{a,b,c,d,e,f\}| = 4$. Let $\alpha < \beta < \gamma < \delta$ be the four indices. There are four cases to consider.

$$\mathbf{u}_{\beta,\gamma}\mathbf{u}_{\gamma,\delta}\mathbf{u}_{\alpha,\gamma} \iff \mathbf{u}_{\beta,\delta}\mathbf{u}_{\alpha,\beta}\mathbf{u}_{\beta,\gamma} \qquad \mathbf{u}_{\alpha,\gamma}\mathbf{u}_{\gamma,\delta}\mathbf{u}_{\beta,\gamma} \iff \mathbf{u}_{\beta,\gamma}\mathbf{u}_{\alpha,\beta}\mathbf{u}_{\beta,\delta}$$

Other points. `makeGraphs.maple` decomposes each equivalence class into connected dual equivalence graphs. These have edges that are coloured by an integer from 2 to $\mathtt{n}-1$, which come from certain involutions $\varphi_2, \ldots, \varphi_{\mathtt{n}-1}$ defined on chains in the Grassmannian-Bruhat order. To a chain $\mathbf{u}_{a,b}\mathbf{u}_{c,d}\cdots\mathbf{u}_{e,f}$, we associate the sequence of its second indices, $(b,d,\ldots,f) = (b_1,\ldots,b_{\mathtt{n}})$. This has a descent at position i if $b_i > b_{i+1}$. These involutions satisfy many conditions, including

(a) The fixed points of φ_i are exactly those words ω with either (1) descents at both $i{-}1$ *and* i or (2) no descent at either position.
(b) If $\varphi_i(\omega) \neq \omega$, then $\varphi_i(\omega)$ has the same descents as ω, in positions before $i{-}1$ and after $i{+}1$, and exactly one of ω and $\varphi_i(\omega)$ has a descent at each of the positions i and $i{+}1$.

Considering these conditions, the relations in [2], and our desire to define the φ_i which fixes a word ω outside of the positions $i{-}1, i, i{+}1$, there is only one way to define φ_i (see our paper [1]) and this is encoded into `makeGraphs.maple`.

The scripts `makeGraphs.maple` and `Check_symmetric.maple` are both quite straightforward, and well documented. The last, `graphIsomorphism.maple`, needed to use an algorithm exploiting the structure of these graphs (colored edges and rather sparse) to check if two graphs were isomorphic, because of the absurd complexity of simple brute force algorithms for graph isomorphism. First, each graph was taken to be pointed with the distinguished vertex `v` one of those whose descent set was lexocigraphically least. Then the graph is represented by shells, where each shell consisted of all vertices a distance r from `v`, and the edges between adjacent shells and within a given shell were also recorded.

This greatly simplified the determination of when two pointed graphs were isomorphic, for there were typically very few choices, once the information of the color of the edges, the shell, and the descent sets of each vertex were taken into account.

References

1. Sami H. Assaf, Nantel Bergeron, and Frank Sottile, *Combinatorial construction of Schubert vs. Schur coefficients*, in progress.
2. Nantel Bergeron and Frank Sottile, *A monoid for the Grassmannian Bruhat order*, European J. Combin. **20** (1999), no. 3, 197–211.