The use of Schubert polynomials in SYMCHAR

A. Kohnert *

July 1991

1 Introduction

SYMCHAR is a collection of C routines, which allows to compute with symmetric groups, multivariate polynomials, representations, characters, symmetric polynomials, wreath products and many other related structures. The routines are written in standard C, so that there is no problem to use it on any machine, which has a C compiler. This was sucessfully checked for IBM-DOS, Atari-DOS and many different UNIX-machines. The code was written in Bayreuth, Paris and Aberystwyth. The program is public domain, and you can get a copy together with documentation files if you send a 3,5" HD-discette to the author.

Schubert polynomials are a generalization of Schur polynomials, and they are used inside SYMCHAR for the multiplication of Schur functions or for the decomposition of skew Schur functions into Schur functions. Schubert polynomials where introduced by Lascoux and Schützenberger in several papers, e.g. [LS1]. Schubert polynomials are labeled by permutations, in spite of partitions, which label Schur functions. We will see how we can characterize the permutations which label Schur polynomials.

2 Schubert polynomials

Schubert polynomials are multivariate polynomials, which are defined using a differential operator ∂_i. Let $A_n := \{a_1, ..., a_n\}$ be an alphabet of n commuting letters and $f \in \mathbb{Z}[A_n]$ an arbitrary polynomial. We define the operation of the elementary transpositions $\sigma_i := (i, i+1)$ on f. $f^{\sigma_i} := f(a_1, ..., a_{i-1}, a_{i+1}, a_i, a_{i+2}, ..)$. Now we are able to introduce the operator ∂_i by

$$\partial_i(f) := \frac{f - f^{\sigma_i}}{a_i - a_{i+1}}$$

The operator ∂_i is a symmetrizing operator, as after the application of ∂_i the polynomial is symmetric in a_i and a_{i+1}. Now we are in the position to define Schubert

*supported by PROCOPE and ARC

polynomials: We do it inductively according to the reduced length of the permutations π, which label the polynomials. Let $\pi \in S_n$ and $l(\pi)$ the reduced length (i.e. number of inversions). Define the **Schubert polynomial** X_π inductively:

1. For the maximal reduced length $l(\pi) = \binom{n}{2}$ we put
$$X_\pi(a_1, ..., a_n) := a_1^{n-1} a_2^{n-2} ... a_{n-1}$$
2. If $l(\pi) =: k$ is not maximal, $\pi =: \tau\sigma_i, l(\tau) = k+1$, then
$$X_\pi(a_1, ..., a_n) := \partial_i(X_\tau(a_1, ..., a_n))$$

This definition works, as we have

$$\begin{aligned} \partial_i\partial_{i+1}\partial_i &= \partial_{i+1}\partial_i\partial_{i+1} \\ \partial_i\partial_j &= \partial_j\partial_i \qquad |i-j| > 1 \end{aligned}$$

There is a second method of labeling Schubert polynomials. We define a bijection L between the permutations of S_n and certain elements of $\mathbb{N}^n$. Let $\pi \in S_n$ given in the list notation, i.e. π_i is the image of i under π. We define

$$L(\pi)_i := |\{j > i | \pi_j < \pi_i\}|$$

i.e. the number of entries to the right of π_i, which are smaller. The image $L(\pi)$ is called the **Lehmer code** of the permutation π. L is a bijection between S_n and the elements $l \in \mathbb{N}^n$ with $l_i < n - i$. The computation of the bijection is clear from an example:

$$L([5, 6, 3, 1, 2, 8, 4, 7]) = 4, 4, 1, 0, 0, 2, 0, 0$$

$$L^{-1}(2, 3, 0, 1, 2, 0, 0) = [3, 5, 1, 4, 7, 5, 6]$$

If we use this labeling instead of the permutations we write

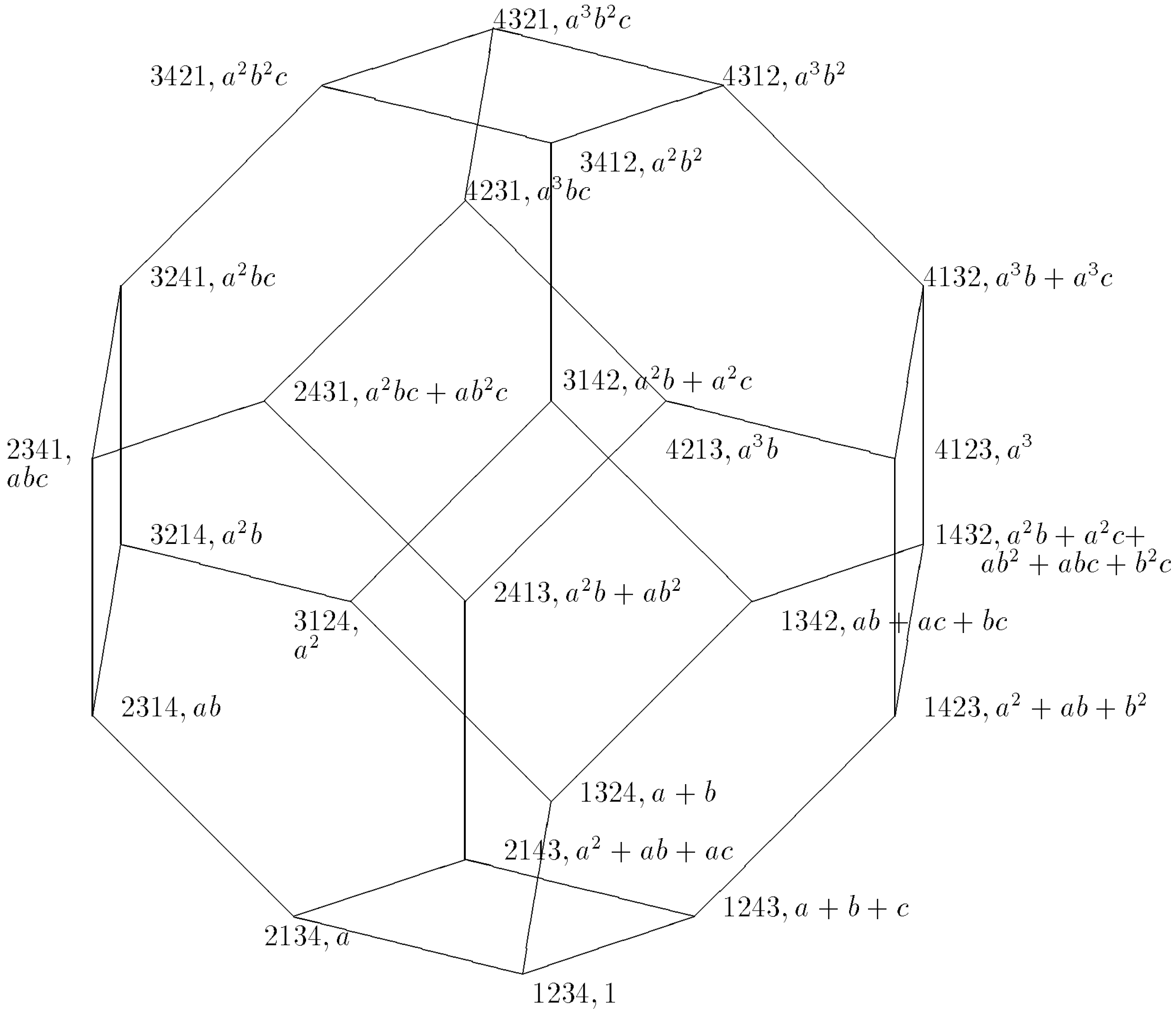
$$Y_I := X_{L^{-1}(I)},$$

where I is a Lehmer code. Now we have the following property: Let $I = 0 \leq I_1 \leq I_2 ... \leq I_k, 0, ..., 0 \in \mathbb{N}^n$ be a Lehmer code. Then

$$Y_I = S_{I_1,...,I_k}(A_k)$$

where $S_{I_1,...,I_k}(A_k)$ is the Schur polynomial labeled by the partition $I_1, ..., I_k$ in the alphabet of k letters. If we look at the following picture we will see, that for example $X_{2413} = Y_{1200}$ is the Schur polynomial $S_{12}(A_2)$.

All Schubert polynomials, labeled by permutations of the S_4.



3 Monks Rule

This is the rule for the multiplication of Schubert polynomials by a single variable. The general case is unknown. This means, there is no general rule known for the multiplication of two arbitrary Schubert polynomials (like the Littlewood Richardson rule for Schur functions).

The rule:

$$a_k X_\pi = \sum_{\rho=\pi(j,k),l(\rho)=l(\pi)+1} sign(j-k) X_\rho$$

An example:

$$a_5 X_{13627458} = -X_{13672458} - X_{13726458} + X_{13628457}$$

To generate all the permutations on the right side of the equation, fix the number π_k (in the example 7) and now look to left of π_k which numbers π_j are smaller with no π_l between π_j and π_k. (in the example these are the numbers 2 and 6) These pairs π_j, π_k are exchanged giving a permutation on the right side with negative sign. Then look to the right of π_k searching for π_j bigger with no π_l in between. The exchanges of these pairs give the permutations with positve sign. (in the example the number 8) Proofs may be found in [M1] where it was proofen in the context of algebraic geometry, or in [K1] where it was proofen in the context of Schubert polynomials.

4 Transition algorithm

In general a Schubert polynomial X_π is not symmetric, but in the case $\pi_i < \pi_{i+1}$ the polynomial is symmetric in a_i and a_{i+1}. This is clear from the definition. Now let X_π be symmetric in the first l variables, the transition algorithm is an algorithm which decomposes the symmetric part of the Schubert polynomial X_π (the polynomial X_π, where $a_{l+1}, a_{l+2}, ...$ are set to zero) into Schur polynomials.

one step of the algorithm:
0. if only one decrease in π stop, X_π is Schur polynomial

X_π, k is the index of the last decrease in π, $\pi_k > \pi_{k+1} < ....$

1. exchange π_k and the biggest π_l with $l > k, \pi_l < \pi_k$
2. call the result π'
3. exchange π'_k with all π'_l with $l < k$
   such that the reduced length is increased by one.
4. call the generated permutations $\tau^1, \tau^2, ...$

one step is shortly described: input π, output $\tau^1, ...$

Example:
$13628457 \rightarrow (k=5)13627458$ exchange 7 with 2 and 6,
giving 13726458 and 13672458.
So the input was one permutation, the output are two permutations.

To see that, we used the formula, which is a special case of the Monk rule:

$$a_k X_{\pi'} = X_\pi - \sum_i X_{\tau^i}$$

k choosen as above. We look look on the symmetric part, where a_k becomes zero so we have the following decomposition

$$X_\pi = \sum_i X_{\tau^i}$$

and if we have only Schur polynomials on the right side, we have a decomposition of the symmetric part into a sum of Schur polynomials.

The single step is applied to all newly generated permutations until the algorithm stops. So we generate a tree, on its root we have the input permutation, on the leaves there are permutations, which labels Schubert polynomials, which are Schur polynomials. So you may substitute the permutations on the leaves by the partitions. A further useful property is the invariance of the generated tree under the embedding $S_n \rightarrow S_{n+1}, \pi \mapsto [1, \pi]$. So if we increase the alphabet of the Schubert polynomial, the decomposition of the now bigger symmetric part into Schur polynomials is still valid.

This algorithm was introduced in [LS1]. In the original paper it was shown, that you can stop with permutations, which are socalled vexillary, but for computational reasons, it is easier to generate new permutations until you reach the Schur polynomials.

The whole algorithm is shortly described: input permutation π, output partitions $\lambda^1, \ldots$ The algorithm will become clear, when we look on the examples in the two following special cases of the transition algorithm.

5 Useful applications of the transition algorithm

We consider two special cases of the transition algorithm, this means we look on special permutations as input of the algorithm.

5.1 Multiplication of Schur functions

We have two partitions: $I = (0 \leq I_1 \leq .. \leq I_k)$ and $J = (0 \leq J_1 \leq ... \leq J_l)$ labeling two Schur functions S_I and S_J. To compute the expansion of the product $S_I \times S_J$ we build the Schubert polynomial

$$\tilde{Y} := Y_{0, I, \underbrace{0, ..., 0}_{I_k \text{ times}}, J, 0, ...}$$

Now the transition algorithm starting from $\tilde{Y}$ gives $\sum c_K S_K$, and we have

$$S_I \times S_J \cap A_k = \tilde{Y} \cap A_k = \sum c_K S_K \cap A_k$$

As the algorithm is invariant under the embedding of S_n into S_{n+1} the decomposition is independent from the size of the alphabet A_k, so we have really the decomposition of the product of two Schur functions. This algorithm was first introduced in a paper of Lascoux and Schützenberger [LS1]. This method has been implemented and is used in SYMCHAR for the multiplication of two Schur functions. The algorithm becomes clear, when we look at the following example:

decomposition of Schur function product $S_{12} \times S_3$

$135249678 \quad Code: 012003000$
135248679
$\downarrow$

$\swarrow \qquad \searrow$

$\downarrow \qquad \downarrow$
$138245679 \qquad 135284679$
$= S_{15} \qquad 135274689$
$\downarrow$

$\swarrow \qquad \searrow$

$\downarrow \qquad \downarrow$
$137254689 \qquad 135724689$
$137245689 \qquad = S_{123}$
$\downarrow$

$\swarrow \qquad \searrow$

$\downarrow \qquad \downarrow$
$147235689 \qquad 137425689$
$= S_{24} \qquad 137245689$
$\downarrow$

$\downarrow$

$\downarrow$
237145689
$= S_{114}$

So we get: $S_{12} \times S_3 = S_{24} + S_{114} + S_{123} + S_{15}$

5.2 Decomposition of Skew Schur functions

We have a skewpartition $I = (0 \leq I_1 \leq .. \leq I_k)/J = (0 \leq J_1 \leq ... \leq J_k)$ labeling the skew Schur function $S_{I/J}$. In order to decompose into a sum of Schur functions, we build the following Schubert polynomial:

$$\tilde{Y} := Y \quad \underbrace{0, ..., 0}_{l \text{ leading zeros}} \quad ... \quad \underbrace{I_{k-1} - J_{k-1}}_{\text{position } l+k-1+J_{k-1}} \quad , 0..0, \quad \underbrace{I_k - J_k}_{\text{position } l+k+J_k} \quad , 0, ...$$

The transition starting from $\tilde{Y}$ gives $\sum c_K S_K$, and we have

$$S_{I/J} \cap A_l = \tilde{Y} \cap A_l = \sum c_K S_K \cap A_l$$

The same argument as in the case of the product of Schur functions, shows that we have a decomposition of skew Schur functions. This algorithm was presented in [K2]. Again we look on an example:

decomposition of Skew Schur function $S_{133/12}$

$$
\begin{array}{ccccc}
 & & 12437586 & Code : 00102010 & \\
 & & 12437568 & & \\
 & & \downarrow & & \\
 & & \downarrow & & \\
 & & \downarrow & & \\
 & & 12437658 & & \\
 & & 12437568 & & \\
 & & \downarrow & & \\
 & \swarrow & & \searrow & \\
 & \downarrow & & \downarrow & \\
 & 12537468 & & 12457368 & Code : 00112000 \\
 & 12536478 & & = S_{112} & \\
 & \downarrow & & & \\
\swarrow & & \searrow & & \\
\downarrow & & \downarrow & & \\
12635478 & & 12563478 & Code : 00220000 & \\
12634578 & & = S_{22} & & \\
\downarrow & & & & \\
\downarrow & & & & \\
\downarrow & & & & \\
12643578 & & & & \\
12634578 & & & & \\
\downarrow & & & & \\
\downarrow & & & & \\
\downarrow & & & & \\
13624578 & Code : 01300000 & & & \\
= S_{13} & & & & \\
\end{array}
$$

And we have $S_{133/12} = S_{112} + S_{22} + S_{13}$.

6 Implementation

The algorithm has been implemented in the system SYMCHAR. It is written in standard C. To implement the generation of the tree we use a stack, of permutations. If the top level permutation is one which labels a Schur polynomial, it is written to the output, if not, the top level permutation is substituted by the permutations, which are generated in one step of the algorithm. The run time of this algorithm depends heavily on the structure which is used to store the result, if we use a tree structure for the result (= list of partitions with coefficent) we get nice run times. These are listed on the last table.

The run times where taken on a HP9000-425, runing UNIX Version 7.0.5, the files where compiled using the optimizer. As an example we used the decomposition of skew Schur functions.

skewpartition	degree of result	number of parts in result	run time (sec.)
123456/1234	11	283	0.1
1234567/12345	13	1833	0.5
12345678/123456	15	13561	4.7
123456789/1234567	17	112745	46.6
12345678910/12345678	19	1039929	658.8

References

[K1] Kohnert A., Die computerunterstützte Berechnung von Littlewood-Richardson Koeffizienten mit Hilfe von Schubertpolynomen, Diplomarbeit Bayreuth 1987

[K2] Kohnert A., Skew Schur functions and Schubert polynomials, preprint 1991

[LS1] Lascoux A. & Schützenberger M.P., Schubert Polnomials and the Littlewood Richardson Rule, Letters in Math. Physics 10 (1985) 111-124

[M1] Monk D., The Geometry of Flag Manifolds, Proc. London Math. Soc. (3) 9 (1959) 253-286

Adress of the author:
Axel Kohnert
Lehrstuhl Mathematik II
Universität Bayreuth
Postfach 101251
D-W8580 Bayreuth
axel@btm2x2.mat.uni-bayreuth.de