Strongly Connected Graph Components and Computing Characteristic Polynomials of Integer Matrices

Simon Lo *, Michael Monagan *, Allan Wittkopf *
{sclo,mmonagan,wittkopf}@cecm.sfu.ca
Centre for Experimental and Constructive Mathematics,
Department of Mathematics, Simon Fraser University,
Burnaby, B.C., V5A 1S6, Canada.

Abstract

Let $\mathbf{A}$ be an $n \times n$ matrix of integers. We present details of our Maple implementation of a modular method for computing the characteristic polynomial of $\mathbf{A}$. Our implementation considers several different representations for the computation modulo primes, including the use of double precision floats.

The algorithm presently implemented in Maple releases 7–10 is the Berkowitz algorithm. We present some timings comparing the two algorithms on large matrices arising from an application in combinatorics of Jocelyn Quaintance.

The matrices are sparse and have a non-obvious block structure which can be exploited to reduce the computing time dramatically. To this end we have also implemented the linear time algorithm of Robert Tarjan for computing the strongly connected components of the directed graph corresponding to the matrix. This work is being incorporated into Maple's LinearAlgebra package.

1 Introduction

Let $\mathbf{A}$ be an $n \times n$ matrix of integers. One way to compute the characteristic polynomial $c(x) = \det(x\mathbf{I} - \mathbf{A}) \in \mathbb{Z}[x]$ is to evaluate the characteristic matrix at n points, compute n determinants of integer matrices, then interpolate to obtain the characteristic polynomial. The determinants of integer matrices can be computed using a fraction-free Gaussian elimination algorithm (see Chapter 9 of Geddes et. al [5]) in $O(n^3)$ integer multiplications and divisions. This approach will lead to an algorithm that requires $O(n^4)$ integer operations.

The algorithm presently implemented in Maple releases 7–10 is the Berkowitz algorithm [2]. It is a division free algorithm and thus can be used to compute the characteristic polynomial of a matrix over any commutative ring R. It does $O(n^4)$ multiplications in R. In [1], Abdeljaoued described a Maple implementation of a sequential version of the Berkowitz algorithm and compared it with the interpolation method and other methods.

*Supported by NSERC of Canada and the MITACS NCE of Canada.

In section 2 we present details of our Maple implementation of a modular method for computing $c(x)$, the characteristic polynomial of $\mathbf{A}$. The algorithm computes the characteristic polynomial modulo a sequence of primes and applies the Chinese remainder theorem. Our implementation considers several different representations for the computation modulo primes, including the use of double precision floats. In section 3 we present some timings comparing our modular algorithm implementations and the Berkowitz algorithm on a 364×364 sparse matrix given to us by Jocelyn Quaintance [8]. It is this matrix that motivated our work on computing characteristic polynomials of integer matrices. The matrix arises from a problem in combinatorics involving 3-dimensional lego blocks. It is one of sequence of sparse matrices of dimension 72×72, 364×364 and 1916×1916. The matrices are available from our website at `http://www.cecm.sfu.ca/CAG/products2005.shtml`. We also present some timings for large randomly generated dense matrices.

The matrices from Quaintance have a block structure, that when identified, can be exploited to reduce the time further. In section 4 we describe the command `StronglyConnectedBlocks`, a new command in Maple's LinearAlgebra package, which identifies the block structure. We present some timings for this command applied to the matrices from Quaintance. and provide additional timings for computing the characteristic polynomial of the matrices in [8]. Using both the modular algorithm and the block decomposition, we are now able to compute the characteristic polynomial of the 1916×1916 matrix in under one minute on an AMD Athlon processor.

2 A Dense Deterministic Modular Algorithm

Let $\mathbf{A} \in \mathbb{Z}^{n\times n}$ be the input matrix. We want to compute the characteristic polynomial $c(x) = \det(x\mathbf{I} - \mathbf{A}) \in \mathbb{Z}[x]$. We will utilize a modular algorithm to compute $c(x)$ by computing the characteristic polynomial of $\mathbf{A}$ modulo a sequence of primes $p_1, p_2, p_3, \ldots$ using the Hessenberg matrix algorithm, then use the Chinese remaindering algorithm to reconstruct $c(x)$. The cost of the Hessenberg approach is $O(n^3)$ arithmetic operations in $\mathbb{Z}_p$ for each prime p. Here is our modular algorithm.

Input: Matrix $\mathbf{A} \in \mathbb{Z}^{n\times n}$
Output: Characteristic polynomial $c(x) = \det(x\mathbf{I} - \mathbf{A}) \in \mathbb{Z}[x]$

1. Compute a bound S (see below) larger than the largest coefficient of $c(x)$.
2. Choose t machine primes $p_1, p_2, \ldots, p_t$ such that $m = \prod_{i=1}^{t} p_i > 2S$.
3. **for** $i = 1$ **to** t **do**
   (a) $\mathbf{A}_i \leftarrow \mathbf{A} \bmod p_i$.
   (b) Compute $c_i(x)$ — the characteristic polynomial of $\mathbf{A}_i$ over $\mathbb{Z}_{p_i}$ via the Hessenberg algorithm.
4. Apply the Chinese remainder theorem:
   Solve $c(x) \equiv c_i(x) \pmod{p_i}$ for $c(x) \in \mathbb{Z}_m[x]$.

5. Output $c(x)$ in the symmetric range for $\mathbb{Z}_m$.

We can compute a bound S for the largest coefficient of $c(x)$ by modifying a bound for the determinant of $\mathbf{A}$. We have

$$\det(\mathbf{A}) \leq n! \prod_{i=1}^{n} \max_{j=1}^{n} |a_{i,j}|$$

so a bound for S is

$$S \leq \det(\mathbf{A}') \quad \text{where} \quad a'_{i,j} = \begin{cases} 1 + |a_{i,i}| & i = j \\ a_{i,j} & \text{otherwise.} \end{cases}$$

2.1 Hessenberg Algorithm

Recall that a square matrix $\mathbf{M} = (m_{i,j})$ is in upper Hessenberg form if $m_{i,j} = 0$ for all $i \geq j + 2$, in other words, the entries below the first subdiagonal are zero.

$$\begin{pmatrix} m_{1,1} & m_{1,2} & m_{1,3} & \cdots & m_{1,n-2} & m_{1,n-1} & m_{1,n} \\ m_{2,1} & m_{2,2} & m_{2,3} & \cdots & m_{2,n-2} & m_{2,n-1} & m_{2,n} \\ 0 & m_{3,2} & m_{3,3} & \cdots & m_{3,n-2} & m_{3,n-1} & m_{3,n} \\ 0 & 0 & m_{4,3} & \cdots & m_{4,n-2} & m_{4,n-1} & m_{4,n} \\ \vdots & \vdots & \ddots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \ddots & m_{n-1,n-2} & m_{n-1,n-1} & m_{n-1,n} \\ 0 & 0 & 0 & \cdots & 0 & m_{n,n-1} & m_{n,n} \end{pmatrix}$$

The Hessenberg algorithm (see [3]) consists of the following two main steps:

Step 1: Reduce the matrix $\mathbf{M} \in \mathbb{Z}_p^{n \times n}$ into the upper Hessenberg form using a series of elementary row and column operations while preserving the characteristic polynomial. In the algorithm below, $\mathbf{R}_i$ denotes the i'th row of $\mathbf{M}$ and $\mathbf{C}_j$ the j'th column of $\mathbf{M}$. In total, $O(n^3)$ operations in $\mathbb{Z}_p$ are performed.

Input: Matrix $\mathbf{M} \in \mathbb{Z}_p^{n \times n}$
Output: Matrix $\mathbf{M}$ in upper Hessenberg form with the same eigenvalues

for $j = 1$ **to** $n - 2$ **do**
  search for a nonzero entry $m_{i,j}$ where $j + 2 \leq i \leq n$
  if found such entry **then**
    do $\mathbf{R}_i \leftrightarrow \mathbf{R}_{j+1}$ and $\mathbf{C}_i \leftrightarrow \mathbf{C}_{j+1}$ **if** $m_{j+1,j} = 0$
    for $k = j + 2$ **to** n **do**
      comment reduce using $m_{j+1,j}$ as pivot
      $u \leftarrow m_{k,j}\, {m_{j+1,j}}^{-1}$
      $\mathbf{R}_k \leftarrow \mathbf{R}_k - u\mathbf{R}_{j+1}$
      $\mathbf{C}_{j+1} \leftarrow \mathbf{C}_{j+1} + u\mathbf{C}_k$
    end for
  end if

comment now the first j columns of $\mathbf{M}$ is in upper Hessenberg form
end for

It is clear from the algorithm that at each step of the outer for loop, we are performing elementary row and column operations, and that at the termination of the outer for loop, the entire matrix is reduced into upper Hessenberg form. Let $\mathbf{H}$ be the matrix $\mathbf{M}$ reduced into upper Hessenberg form, let the elementary matrix $\mathbf{E}_j$ represent the j'th elementary row operation and let $\mathbf{E} = \mathbf{E}_1\mathbf{E}_2 \cdots \mathbf{E}_{n-2}$. We can write $\mathbf{H} = \mathbf{EME}^{-1}$ since the elementary column operations that we perform in the algorithm are inverses of elementary row operations. Thus, this is a similarity transformation.

To see that $\mathbf{H}$ has the same characteristic polynomial as the matrix $\mathbf{M}$, note that $\mathbf{H} = \mathbf{EME}^{-1}$ implies

$$\begin{aligned}
\det(x\mathbf{I} - \mathbf{H}) &= \det(x\mathbf{I} - \mathbf{EME}^{-1}) \\
&= \det(\mathbf{E}(x\mathbf{I})\mathbf{E}^{-1} - \mathbf{EME}^{-1}) \\
&= \det(\mathbf{E}(x\mathbf{I} - \mathbf{M})\mathbf{E}^{-1}) \\
&= \det(\mathbf{E}) \det(x\mathbf{I} - \mathbf{M}) \det(\mathbf{E}^{-1}) \\
&= \det(\mathbf{E}) \det(x\mathbf{I} - \mathbf{M}) \det(\mathbf{E})^{-1} \\
&= \det(x\mathbf{I} - \mathbf{M}).
\end{aligned}$$

Step 2: The characteristic polynomial $c(x) = p_{n+1}(x) \in \mathbb{Z}_p[x]$ of the upper Hessenberg form can be efficiently computed bottom up using $O(n^2)$ operations in $\mathbb{Z}_p[x]$ ($O(n^3)$ operations in $\mathbb{Z}_p$) from the following recurrence for $p_k(x)$.

$$p_{k+1}(x) = \begin{cases} 1 & k = 0 \\ (x - m_{k,k})\, p_k(x) - \sum\limits_{i=1}^{k-1} (\prod\limits_{j=i}^{k-1} m_{j+1,j})\, m_{i,k}\, p_i(x) & 1 \le k \le n+1 \end{cases}$$

The algorithm below computes the above recurrence bottom up, and clearly shows that $O(n^2)$ operations in $\mathbb{Z}_p[x]$ are required.

Input: Matrix $\mathbf{M} \in \mathbb{Z}_p^{n\times n}$ in upper Hessenberg form
Output: Characteristic polynomial $c(x) \in \mathbb{Z}_p[x]$ of $\mathbf{M}$

$p_1(x) \leftarrow 1$
for $k = 1$ **to** n **do**
  $p_{k+1}(x) \leftarrow (x - m_{k,k})\, p_k(x)$
  $t \leftarrow 1$
  for $i = 1$ **to** $k-1$ **do**
    $t \leftarrow t\, m_{k-i+1,k-i}$
    $p_{k+1}(x) \leftarrow p_{k+1} - t\, m_{k-i,k}\, p_{k-i}(x)$
  end for
end for
output $p_{n+1}(x)$

2.2 Asymptotic Comparison of the Methods

Let $\mathbf{A}$ be an $n \times n$ matrix of integers. To compare the running times of the Berkowitz algorithm and the modular algorithm, we suppose that the entries of $\mathbf{A}$ are bounded by B^m in magnitude, that is, they are m base B digits in length. For both algorithms, we need a bound S on the size of the coefficients of the characteristic polynomial $c(x)$. A generic bound on the size of the determinant of $\mathbf{A}$ is sufficient since this is the largest coefficient of $c(x)$. The magnitude of the determinant of $\mathbf{A}$ is bounded by $S = n!B^{mn}$ and its length is bounded by $n \log_B n + mn$ base B digits. If $B > 2^{15}$ then we may assume $\log_B n < 2$ in practice and hence the length of the determinant is $O(mn)$ base B digits.

In Berkowitz's algorithm, the $O(n^4)$ integer multiplications are on integers of average size $O(mn)$ digits in length, hence the complexity (assuming classical integer arithmetic is used) is $O(n^4(mn)^2)$. Since Maple 9 uses the FFT for integer multiplication and division, the complexity is reduced to $\tilde{O}(n^5 m)$.

In the modular algorithm, we will need $O(mn)$ machine primes. The cost of reducing the n^2 integers in $\mathbf{A}$ modulo one prime is $O(mn^2)$. The cost of computing the characteristic polynomial modulo each prime p is $O(n^3)$. The cost of the Chinese remaindering assuming a classical method for the Chinese remainder algorithm (which is what Maple uses) is $O(n(mn)^2)$. Thus the total complexity is $O(mnmn^2 + mnn^3 + n(mn)^2) = O(m^2n^3 + mn^4)$.

If we assume $m = O(n)$, that is, the size of the integer grows proportionally with the size of the matrix, the complexity of the Berkowitz algorithm and the modular algorithm is $\tilde{O}(n^6)$ and $O(n^5)$ respectively.

If we have $|A_{i,j}| < B$ for all n (in the second set of timings $|A_{i,j}| < 10^3$), the complexity of the Berkowitz algorithm and the modular algorithm is $\tilde{O}(n^5)$ and $O(n^4)$ respectively. We mention here that algorithms which are asymptotically faster than $O(n^4)$ are known. Two recent reference are Kaltofen and Villard in [6] and Dumas, Pernet and Wan in [4]. We have not implemented any of these algorithms.

3 Timings and Implementations

For a machine prime p, in order to improve the running time of our algorithm, we've implemented the Hessenberg algorithm over $\mathbb{Z}_p$ in the C programming language and the rest of the algorithm in Maple. We used the Maple external function interface to call the C code (see [7]). We've implemented both the 32-bit integer version and 64-bit integer versions, and also several versions using 64-bit double precision floating point values for comparison.

The following table consists of some timings (in seconds) of our modular Hessenberg algorithm for a sparse 364×364 input matrix arising from an application in combinatorics (see [8]). Rows 1-9 below are for the modular algorithm using different implementations of arithmetic for $\mathbb{Z}_p$. The accelerated floating point version **fprem** using 25-bit primes generally give the best times.

Versions	Xeon 2.8 GHz	Opteron 2.0 GHz	AXP2800 2.08 GHz	Pentium M 2.00 GHz	Pentium 4 2.80 GHz
64int	100.7	107.4			
32int	66.3	73.0	76.8	35.6	57.4
new 32int	49.7	54.7	56.3	25.5	39.6
fmod	29.5	32.1	33.0	35.8	81.1
trunc	67.8	73.7	69.6	88.5	110.6
modtr	56.3	62.5	59.5	81.0	82.6
new fmod	11.0	11.6	14.5	15.2	28.8
fprem	10.4	10.9	13.7	13.9	26.8
fLA	17.6	19.9	21.9	26.2	27.3
Berkowitz	2053.6	2262.6			

Implementations

64int The 64-bit integer version is implemented using the *long long int* datatype in C, or equivalently the *integer[8]* datatype in Maple. We can use 32-bit primes. All modular arithmetic first executes the corresponding 64-bit integer machine instruction, then reduces the result mod p because we work in $\mathbb{Z}_p$. We allow both positive and negative integers of magnitude less than p.

32int The 32-bit integer version is similar, but implemented using the *long int* datatype in C, or equivalently the *integer[4]* datatype in Maple. 16-bit primes are used here.

new 32int This is an improved **32int**, with various hand/compiler optimizations.

fmod This 64-bit float version is implemented using the *double* datatype in C, or equivalently the *float[8]* datatype in Maple. 64-bit float operations are used to simulate integer operations. Operations such as additions, subtractions, multiplications are followed by a call to the C library function *fmod()* to reduce the results mod p, since we are working in $\mathbb{Z}_p$. We allow both positive and negative floating point representations of integers with magnitude less than p.

trunc This 64-bit float version is similar to above, but uses the C library function *trunc()* instead of *fmod()*. To compute $b \leftarrow a \bmod p$, we first compute $c \leftarrow a - p \times trunc(a/p)$, then $b \leftarrow c$ if $c \neq \pm p$, $b \leftarrow 0$ otherwise. The *trunc()* function rounds towards zero to the nearest integer.

modtr A modified **trunc** version, where we do not do the extra check for equality to $\pm p$ at the end. So to compute $b \leftarrow a \bmod p$, we actually compute $b \leftarrow a - p \times trunc(a/p)$, which results in $-p \leq b \leq p$.

new fmod An improved **fmod** version, where we have reduced the number of times *fmod()* is called. In other words, we reduce the results mod p only when the number of accumulated arithmetic operations on an entry exceeds a certain threshold. In order to allow this, we are restricted to use 25-bit primes. We call this delayed mod acceleration. See the next subsection.

fprem Equivalent to **new fmod** version, but via direct assembly programming using *fprem* instruction, removing the function call overhead and making some efficiency improvements.

fLA An improved **trunc** version using delayed mod acceleration. It is the default used in Maple's `LA:-Modular` routines.

3.1 Efficiency Considerations

There are a few considerations for use of floating point for mod p computations. Keeping these in mind, one can implement faster external code for the algorithms than is possible with the integer codes, and still have the advantage of using larger primes on 32-bit machines.

1. Although floating point integer computations can represent 53-bit numbers accurately, we restrict the modulus to $p < 2^{25}$, which allows for more efficient mod operations, and multiple mod operations (up to 8) to occur before having to reduce modulo p. We call this the delayed mod acceleration.

2. Leveraging the smaller primes allows up to 8 computations (using a maximal size prime) to occur before we must perform a mod. This can be efficiently utilized in row-based algorithms, as a counter associated with each row can count the number of operations performed, and the algorithms can be made to only perform the mod once the maximal number of computations is reached.

3. Floating point computations have a number of ways in which a mod can be performed, including but not limited to subtracting the floor of the inverse of the modulus times the number from the number, the floating point mod operation *fmod* or *fprem*, using *trunc*, etc.

In our experiments we found the following: Use of the smaller primes, and delayed mod, mentioned in items 1 and 2 above increased performance by a factor of 2-3.

With these modifications, use of floating point modular arithmetic generally demonstrated better performance than integer modular arithmetic.

The use of the C-library *fmod* function or direct assembly programming using the *fprem* instruction (essentially equivalent modulo function call overhead and some efficiency improvements made available for our specific use of *fmod*) showed better performance than the other floating point schemes, except on the Pentium 4, on which it was approximately equal. Note also that on Pentium M the *fprem* performance was nearly a factor of 2 times better.

3.2 Timings for Dense Matrices

The following table consists of some timings (in seconds) of our modular Hessenberg algorithm using float (**fprem**) and integer (**new 32int**) implementations on dense $n \times n$ matrices, with uniformly random integer entries between -999 and 999. We also compare with Maple's Berkowitz algorithm. The timings were done on a dual Opteron 2.2 GHz processor running Unix.

n	float	integer	Berkowitz
50	<0.1	0.13	7.85
100	0.61	1.85	128.6
200	8.82	30.8	2248.1
300	45.4	153.2	
400	173.5	493.4	
600	1195.2	2973.1	
800	4968.8		

Note that the time for Berkowitz algorithm on a dense 200×200 integer matrix is even slower than a sparse 364×364 integer matrix, resulting from the cost of large integer arithmetic. Maple's Berkowitz algorithm is much much slower than the others, as the timings suggest.

From the above data, we can see that the float version is always faster than the integer version (about 3 times faster). Therefore in practice, we would always use the float version.

3.3 Making the algorithm output sensitive

One way to improve the running time further would be to use the early termination technique described below. Consider the following matrix

$$\mathbf{A} = \begin{pmatrix} 1 & u & v & w \\ 0 & 2 & x & y \\ 0 & 0 & 3 & z \\ 0 & 0 & 0 & 4 \end{pmatrix}.$$

The characteristic polynomial of $\mathbf{A}$ is $c(x) = (x-1)(x-2)(x-3)(x-4) = x^4 - 10x^3 + 35x^2 - 50x + 24$. Notice that the largest coefficient of $c(x)$ does not depend on any of the entries u, v, w, x, y, z. So if u, v, w, x, y, z are large, then the bound S for the largest coefficient of $c(x)$ would be arbitrarily far off.[1] Our modular algorithm would use too many primes and would do too many unnecessary modular computations. This observation suggests that we use a output sensitive version of the algorithm and not use a bound at all. We will incrementally apply the Chinese remainder theorem to reconstruct $c(x)$ and stop the Chinese remaindering once ζ consecutive modular images "remain the same".

Let $p_1, p_2, \ldots, p_s, p_{s+1}, \ldots, p_{s+\zeta}$ be machine primes, $c_i(x) \equiv c(x) \pmod{p_i}$, $C_i(x) \equiv c(x) \pmod{p_1 \cdots p_i}$. Application of the Chinese remainder theorem allows us to construct $C_i(x)$ from $c_i(x)$ and $C_{i-1}(x)$. This is an incremental version of Garner's algorithm. Now suppose that $C_s(x) = C_{s+1}(x) = \cdots = C_{s+\zeta}$, then there is a high probability that $c(x) = C_s(x)$. Choosing ζ carefully will ensure that the probability of premature termination of the Chinese remaindering is low.

This output sensitive probabilistic version of the modular algorithm is much faster than the deterministic version when the largest coefficient of the characteristic polynomial is much smaller than the bound. On the sparse 364×364 example in the previous section, the timing improves by about 30%.

[1] Yes, it is true, we could improve the algorithm for computing the bound to "notice" that this matrix is diagonal. But we can always fool the bound code by, for example, permuting the rows and columns of A

4 Strongly Connected Components

Consider again the matrix

$$\mathbf{A} = \begin{pmatrix} 1 & u & v & w \\ 0 & 2 & x & y \\ 0 & 0 & 3 & z \\ 0 & 0 & 0 & 4 \end{pmatrix}.$$

The Hessenberg algorithm does not need to do any work since $\mathbf{A}$ is already in upper Hessenberg form. Now consider the matrix $\mathbf{A}^T$, i.e.,

$$\mathbf{A}^T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ u & 2 & 0 & 0 \\ v & x & 3 & 0 \\ w & y & z & 4 \end{pmatrix}$$

which has the same characteristic polynomial. The algorithm would be required to do some actual reductions. This observation lead us to ask the following question: For what matrix structures can we compute the characteristic polynomial quickly? Consider a matrix of the form

$$\mathbf{B} = \begin{pmatrix} a & b & w & x \\ c & d & y & z \\ 0 & 0 & e & f \\ 0 & 0 & g & h \end{pmatrix}.$$

Let $c_{\mathbf{B}}(x)$ be the characteristic polynomial of $\mathbf{B}$. Then $c_{\mathbf{B}}(x) = c_{\mathbf{B}_1}(x)c_{\mathbf{B}_2}(x)$ where

$$\mathbf{B}_1 = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \text{ and } \mathbf{B}_2 = \begin{pmatrix} e & f \\ g & h \end{pmatrix}.$$

If the input matrix is block upper (lower) triangular, then the computation of the characteristic polynomial reduces to the product of the characteristic polynomials of the diagonal blocks which are easier to compute. But often matrices are not block upper (lower) triangular, but are row and column permutations of block upper (lower) triangular matrices. For example,

$$\mathbf{P} = \begin{pmatrix} h & 0 & g & 0 \\ z & d & y & c \\ f & 0 & e & 0 \\ x & b & w & a \end{pmatrix}$$

is the matrix $\mathbf{B}$ with rows 1,4 and columns 1,4 interchanged, and is no longer block upper triangular. The main observation here is that simultaneous row and column interchanges do not modify the characteristic polynomial (the arguments of section 2.1 also work here), so that if the rows and columns of a matrix are permuted by the same permutation, then the characteristic polynomial remains unchanged.

In order to apply the above observation, we need to efficiently compute the permutation that would make the matrix block upper triangular. It turns out that we can compute this permutation in linear time by finding the *strongly connected components* of a directed graph.

Let $\mathbf{A}$ be a $n \times n$ matrix. We denote by $Graph(\mathbf{A})$ the weighted directed graph with n vertices such that $\mathbf{A}$ is the adjacency matrix of $Graph(\mathbf{A})$. Recall that a directed graph G is strongly connected if for each pair of vertices $u, v \in G, u \neq v$, there exists a path $u \rightsquigarrow v$. Also, every directed graph can be partitioned into maximal strongly connected components. Note that permuting the vertices corresponds to permuting the rows and columns of $\mathbf{A}$.

Denote by $\mathbf{A}_{(u_1,u_2,...,u_r),(v_1,v_2,...,v_s)}$ the $r \times s$ submatrix of $\mathbf{A}$ such that

$$(\mathbf{A}_{(u_1,u_2,...,u_r),(v_1,v_2,...,v_s)})_{i,j} = \mathbf{A}_{u_i,v_j}.$$

The method works as follows:

1. Compute (see below) the k strongly connected components of $Graph(\mathbf{A}) : V_1, V_2, ..., V_k$ where $V_i = \{v_{i1}, v_{i2}, ..., v_{in_i}\}$.
2. For $1 \leq i \leq k$, compute the characteristic polynomial of the submatrix $\mathbf{A}_{(v_{i1},v_{i2},...,v_{in_i}),(v_{i1},v_{i2},...,v_{in_i})}$. We denote these submatrices as $\mathbf{A}_{V_i,V_i}$.
3. Output (the characteristic polynomial $c(x)$ of $\mathbf{A}$) the product of the characteristic polynomials computed in step 2.

To see why the above method works let the k strongly connected components of $G = Graph(\mathbf{A})$ be $V_1, ..., V_k$. We can topologically sort the k strongly connected components such that $V_i \prec V_j$ implies there does not exist $u \in V_i, v \in V_j$ such that $(v, u) \in G$. Without loss of generality assume $V_1 \prec V_2 \prec ... \prec V_k$. Consider relabeling the vertices in increasing topological order (which is not unique) and consider the adjacency matrix $\mathbf{A}'$ that represents the relabeled graph. First, it is clear that $\mathbf{A}'$ has the same characteristic polynomial as $\mathbf{A}$ since $\mathbf{A}'$ is obtained by permuting rows and columns of $\mathbf{A}$. Note that the (i, j)'th blocks of $\mathbf{A}'$, denoted by $\mathbf{A}'_{i,j}$ are precisely the submatrices $\mathbf{A}_{V_i,V_j}$. Therefore, the diagonal $\mathbf{A}'_{i,i}$'s are precisely the $\mathbf{A}_{V_i,V_i}$'s. Furthermore, the relabeling guarantees $i < j$ if and only if $V_i \prec V_j$ so if $i < j$, then we have $\mathbf{A}'_{j,i} = \mathbf{0}$. Thus, $\mathbf{A}'$ is block upper triangular, so the characteristic polynomial is just the product of the characteristic polynomials of the diagonal blocks, the $\mathbf{A}_{V_i,V_i}$'s. In practice, it is not necessary to topologically sort the V_i's because in order to compute $c(x)$, we do not need to know explicitly any of the entries in $\mathbf{A}'_{i,j}$, where $i < j$. Also, the ordering of the vertices in each V_i does not matter.

Now consider the example matrix $\mathbf{P}$ above

$$\mathbf{P} = \begin{pmatrix} h & 0 & g & 0 \\ z & d & y & c \\ f & 0 & e & 0 \\ x & b & w & a \end{pmatrix}.$$

Here, the strongly connected components of $Graph(\mathbf{P})$ are $V_1 = \{1, 3\}$ and $V_2 = \{2, 4\}$. To see this, it might help the reader to draw $Graph(\mathbf{P})$. Next we compute the product of the characteristic polynomials of

$$\begin{pmatrix} h & g \\ f & e \end{pmatrix} \text{ and } \begin{pmatrix} d & c \\ b & a \end{pmatrix}.$$

Clearly we would get the characteristic polynomial of $\mathbf{P}$, which is also the characteristic polynomial of $\mathbf{B}$. Observe that $V_2 \prec V_1$. Suppose we choose the ordering 2,4,1,3, then we obtain the matrix

$$\mathbf{P}' = \begin{pmatrix} d & c & z & y \\ b & a & x & w \\ 0 & 0 & h & g \\ 0 & 0 & f & e \end{pmatrix}.$$

It doesn't quite look like $\mathbf{B}$, but the characteristic polynomial of $\mathbf{P}'$ is the same as $\mathbf{B}$ above. The difference is that the vertices in V_i's are permuted. If we choose the ordering 4,2,3,1, then we obtain $\mathbf{B}$.

Maple code for the computation of the strongly connected component blocks of a Matrix is available at `http://www.cecm.sfu.ca/CAG/papers/SCC.txt`. It computes the strongly connected components in linear time and outputs the submatrices in a list (see [9] for a description and proof of the algorithm).

The output is a list of non-zero square matrices $\mathbf{A}_1, \mathbf{A}_2, ..., \mathbf{A}_r$. Let $m = \sum_{i=1}^{r} \dim(\mathbf{A}_i)$. If the input $\mathbf{M}$ is an $n \times n$ matrix, the output satisfies

$$c_{\mathbf{M}}(x) = x^{n-m} \prod_{i=1}^{r} c_{\mathbf{A}_i}(x)$$

where $c_{\mathbf{A}}(x)$ is the characteristic polynomial of $\mathbf{A}$.

Note that one bottleneck in the implementation of the algorithm is the extraction of the nonzero entries in the input matrix. In the code available on the web, there is a line

```
A:=[seq([seq(`if`(M[i,j]=0,NULL,j),j=1..n)],i=1..n)]
```

to extract these entries. For the sparse 364×364 matrix, the code runs in a total of 0.25 seconds, 0.20 of which is the extraction process.

In the research version of Maple, this algorithm has been implemented as part of the **LinearAlgebra** package as the **StronglyConnectedBlocks** function. This implementation uses external (i.e. C) code both to extract the nonzero entries, and to perform the computation of the strongly connected components. The syntax is:

```
LinearAlgebra[StronglyConnectedBlocks](M, [returnsingular=truefalse])
```

where $\mathbf{M}$ is the input matrix, and the option **returnsingular** is provided to allow one to avoid reconstruction of the block matrices if the input matrix can be determined to be singular.

We now provide data comparing the run-time and memory usage for the computation of the strongly connected blocks for the sparse 364×364 matrix, as well as sparse 72×72 and 1916×1916 matrices which also result from the combinatorics application from Quaintance [8]. Note that the 364×364 matrix decomposed into 12 blocks of sizes $5, 5, 9, 10, 10, 10, , 22, 22, 48, 54, 76, 93$ while the 72×72 matrix decomposed into 6 blocks of sizes $4, 4, 8, 13, 20, 23$, and the 1916×1916 matrix decomposed into 31 blocks of sizes $6\$6, 11\$2, 12\$9, 13\$2, 32\$2, 59, 70\$3, 103\$2, 241, 260, 306, 378$. The timings below are for an AMD Athlon(tm) 64 X2 Dual Core 4400+ (2.2GHz) with 2GB memory under 64-bit Linux.

Matrix	Maple SCC Maple Extract		Maple SCC C Extract		C SCC Maple Extract		C SCC C Extract	
72×72	0.01s	0.52M	0.02s	0.13M	0.01s	0.39M	.004s	<.001M
364×364	0.20s	5.24M	0.06s	2.75M	0.09s	5.11M	.004s	0.26M
1916×1916	5.20s	51.1M	1.06s	12.8M	3.57s	46.4M	0.11s	5.11M

From this we can see that the externally coded extraction process provides a significant reduction in the run time and memory usage of the implementation, and the combination of the two reduces the time and memory usage for the difficult example by factors of $\approx 50, 10$ respectively.

4.1 Final Timings

To complete the section on improvements we provide a timing comparison for computation of the characteristic polynomial of the three matrices of the prior section (72×72, 364×364, and 1916×1916). Maple's Berkowitz implementation is compared with the new Hessenberg implementation, computing both with and without block decomposition, but the new Hessenberg implementation always uses early termination. Timings are for an AMD Athlon(tm) 64 X2 Dual Core 4400+ (2.2GHz) with 2GB memory under 64-bit Linux.

Matrix	Berkowitz No blocks	Berkowitz With blocks	Hessenberg No blocks	Hessenberg With blocks
72×72	2.36	0.04	0.16	0.01
364×364	1086.50	9.59	28.45	0.18
1916×1916	N/A*	16594.79	28468.55	46.89

* the timing for the 1916×1916 matrix using Berkowitz and no blocks was not attempted.

It is clear from the timings that block decomposition is most relevant to the efficient computation of the characteristic polynomials for this class of problems.

For the 364×364 matrix, the factor of improvement from Maple's prior algorithm to the new algorithm using block decomposition is ≈ 6000. The computation of the characteristic polynomial of the 1916×1916 might as well be considered impossible for the unaided Berkowitz algorithm (an extremely rough estimate based on the performance of the implementation for the other cases puts this at $\approx 5.8\text{e}5$ cpu seconds, or 6.7 cpu days), while it completes in under a minute for the new algorithm.

We are currently in the process of integrating these codes into the main Maple code base to aid in more efficient computation of characteristic polynomials and determinants.

References

[1] J. Abdeljaoued. The Berkowitz Algorithm, Maple and Computing the Characteristic Polynomial in an Arbitrary Commutative Ring. *MapleTech* **5**(1), pp. 21–32, Birkhauser, 1997.

[2] S. J. Berkowitz. On Computing the Determinant in Small Parallel time using a Small Number of Processors. *Inf. Processing Letters* **18**(3) pp. 147–150, 1984.

[3] H. Cohen. *A Course in Computational Algebraic Number Theory.* Graduate texts in mathematics, **138**, Springer-Verlag, 1995.

[4] J.-G. Dumas, C. Pernet, Z. Wan. Efficient Computation of the Characteristic Polynomial. To appear in *The proceedings of ISSAC 2005,* ACM Press, 2005

[5] K. O. Geddes, S. R. Czapor, and G. Labahn. *Algorithms for Computer Algebra.* Kluwer Academic Publ., Boston, Massachusetts, USA, 1992.

[6] E. Kaltofen, G. Villard. On the Complexity of Computing Determinants. *Journal of Computational Complexity* **13** pp. 91–130, 2004.

[7] M. Monagan, K. Geddes, K. Heal, G. Labahn, S. Vorkoetter, J. McCarron, P. deMarco. *Maple 9 Advanced Programming Guide*, Ch. 6, Maplesoft, 2003.

[8] J. Quaintance. $m \times n$ Proper Arrays: Geometric Construction and the Associated Linear Cellular Automata. *Proceedings of the 2004 Maple Summer Workshop*, 2004.

[9] R. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing* **1**(2) pp. 146–160, 1972.