Electronic Colloquium on Computational Complexity - Reports Series 1997 - available via:

ECCC
TR97-036

FTP: ftp.eccc.uni-trier.de:/pub/eccc/
WWW: http://www.eccc.uni-trier.de/eccc/
Email: ftpmail@ftp.eccc.uni-trier.de with subject 'help eccc'

Determinant: Combinatorics, Algorithms, and Complexity*

Meena Mahajan†
Institute of Mathematical Sciences,
Chennai 600 113, INDIA.
meena@imsc.ernet.in

V Vinay
Department of Computer Science and Automation,
Indian Institute of Science,
Bangalore 560 012, INDIA.
vinay@csa.iisc.ernet.in

August 1, 1997

Abstract

We prove a new combinatorial characterization of the *determinant*. The characterization yields a simple combinatorial algorithm for computing the determinant. Hitherto, all (known) algorithms for determinant have been based on linear algebra. Our combinatorial algorithm requires no division and works over arbitrary commutative rings. It also lends itself to efficient parallel implementations.

It has been known for some time now that the complexity class GapL characterizes the complexity of computing the determinant of matrices over the integers. We present a direct proof of this characterization.

1 Introduction

The determinant has been a subject of study for over 200 years. Its history can be traced back to Leibnitz, Crammer, Vandermode, Binet, Cauchy, Jacobi, Gauss and others. Given its importance in linear algebra in particular and in geometry in general, it is not surprising that a galaxy of great mathematicians investigated the determinant from varied viewpoints.

The algorithmic history of the determinant is as old as the mathematical concept itself. After all, the determinant was invented to solve systems of linear equations. Much of the

*A preliminary version of this paper appeared in Proc. Eighth Annual ACM-SIAM Symposium on Discrete Algorithms SODA 97, 730–738.

†Part of this work was done when this author was visiting the Department of Computer Science and Automation, IISc, Bangalore.

initial effort was expended on proving the so called "Cramer's Rule", "Laplace Expansion" and the "Cauchy-Binet Theorem", and these led to a variety of interesting algebraic identities. The first definitions of determinant used *inversions* as a means of computing the sign of a permutation. Cauchy realized that the sign of a permutation may be more easily computed by considering the cycle decomposition of the permutation: if k is the number of cycles in the decomposition of a permutation over S_n, he showed that $(-1)^{n-k}$ computes the sign. In a sense, Cauchy appears to have started the combinatorial approach to determinants.

The so-called "Gaussian Elimination" is a standard procedure to calculate the determinant. It converts a given matrix into an upper triangular matrix using elementary row operations, which maintain the value of the determinant, and uses $O(n^3)$ operations. It can be shown that the sizes of numbers in the intermediate steps are small and this gives rise to a polynomial time algorithm. This algorithm, however, appears to be sequential. And, the algorithm, in its present form, would require division, that renders it useless over arbitrary rings. To use this method over a ring, one considers a field extension (eg for computing the determinant over integers, compute using rationals). This procedure, while theoretically correct, often introduces a computational problem. For instance, over integers, because of the divisions involved, this method may needlessly introduce floating point errors. Thus, in several situations, a division-free method which still has polynomial bit-complexity would be preferable to Gaussian elimination.

Numerical Analysts have looked quite closely at the problem of computing the determinant and also the associated problem of computing the characteristic polynomial of a given matrix. An authoritative book on this subject is due to Fadeev and Fadeeva [FF 63]. The book lists more than half a dozen methods for the computation of the characteristic polynomial. The most important among them seem to be (1) Krylov's Method, (2) Leverier's Method, and (3) Samuelson's Method. Csanky [Cs 76] observes that Leverier's method may be implemented in NC^2. However, Leverier's method uses division and hence is unsuitable over arbitrary fields. (The method is applicable only over fields of characteristic zero, or over fields with characteristic greater than the dimension of the matrix. So the algorithm cannot be used, in general, over finite fields.) Berkowitz [Be 84] observes that Samuelson's method [Sa 42] is division-free and may be implemented in NC^2. Valiant [Va 92] analyzes the nature of monomials that result from Samuelson's method. Independently, Chistov [Ch 85] uses arithmetic over polynomials to come up with a division-free NC^2 algorithm. Thus the Samuelson-Berkowitz algorithm as well as Chistov's algorithm can be used over any commutative ring.

Vinay[Vi 91], Damm[Da 91] and Toda[To 91] observed independently that DET (as a complexity class) has an exact characterization. They showed that over integers, DET is exactly GapL. That is, any function that is log-space reducible to computing the determinant of a matrix over integers can be computed as the difference of two #L functions. Here, #L corresponds to the number of accepting paths in an NL machine. These results establish a telling parallel between the complexity of the two major algorithmic problems: complexity of the Permanent vs the Determinant. While Valiant [Va 79] has shown that computing the Permanent is GapP complete, the Determinant is complete for GapL; both are complete for counting versions of nondeterministic classes. An interesting feature of the three independent proofs cited above is that they all rely on Samuelson's method to convert the problem of computing the determinant to iterated matrix multiplication. In this paper, we present a direct and self-contained proof of this theorem.

We give the first combinatorial algorithm for computing the determinant. We do this by extending the definition of a permutation to a `clow sequence`. A combinatorial proof establishes that all clow sequences that are not permutations cancel each other, leaving precisely the permutations. We then show how clow sequences may be realized in a simple graph-theoretic model. The model is described by a tuple $\langle G, s, t_+, t_- \rangle$, where G is a directed acyclic graph (DAG), and s, t_+, t_- are distinguished vertices in G. Let $paths(G, s, t)$ compute the number of paths from s to t in G. Then the integer function computed by $\langle G, s, t_+, t_- \rangle$ is $paths(G, s, t_+) - paths(G, s, t_-)$. The model yields a polynomial time algorithm via simple dynamic programming techniques (see Table 1). It characterizes GapL exactly and also has characterizations in terms of arithmetic skew circuits [Ve 92, To 91] and arithmetic branching programs, yielding NC^2 and GapL algorithms (see Table 2 and Table 3). The result stands out in contrast with Nisan's result [Ni 91] which shows that determinant cannot be computed by a polynomial size branching program over a non-commutative semi-ring.

The size of the DAG we construct is about $O(n^4)$, with $O(n^6)$ edges. This may be implemented on an arithmetic skew circuit with $O(n^6)$ wires. This compares rather favourably with the $O(n^{18})$ implementation of Toda. (In [To 92], Toda noted that Samuelson's method can be implemented on arithmetic skew circuits of size n^{18}.)

Our combinatorial proof is inspired by Straubing, who gave a purely combinatorial interpretation and very elegant proof of the Cayley-Hamilton theorem [St 83].

Various other parallel algorithms for computing the determinant (including Chistov's method and the Samuelson-Berkowitz method) can also be interpreted combinatorially, and correctness can also be proved using purely combinatorial techniques. The objects generated by these algorithms turn out be variations of clow sequences. [MV 97] describes such interpretations for some algorithms.

Of course, the combinatorial approach cannot replace the algebraic one altogether. But it can, as we feel it does in this case, offer interesting insights into the nature of a seemingly purely algebraic problem.

2 The Combinatorics

We will start with the definition of the determinant of an n dimensional matrix, A.

$$det(A) = \sum_{\sigma \in S_n} sgn(\sigma) \prod_i a_{i\sigma(i)}$$

The summation is over all permutations on n elements. The sign of a permutation is defined in terms of the number of inversions.

$$sgn(\sigma) = (-1)^{\text{number of inversions in } \sigma}$$

To move to a combinatorial setting, we interpret the matrix A as a weighted directed graph G_A on n vertices, where the weight on the directed edge $\langle i, j \rangle$ is a_{ij}. A permutation in S_n now corresponds to a cycle cover: the cycle decomposition of the permutation, when interpreted as a graph, induces a partition on the vertex set into disjoint cycles.

This definition cannot be directly converted into an efficient algorithm for the determinant, as the number of monomials in the above definition is $n!$. Since enumeration is out of question, any algorithm should therefore implicitly count over all monomials. The bottleneck in doing so directly is that these permutation are not easily "factorizable" to allow for a simple implementation. We will get around this problem by enlarging the summation from cycle covers to **clow sequences**.

A **clow** (clow for clo-sed w-alk) is a walk $\langle w_1, \ldots, w_l \rangle$ starting from vertex w_1 and ending at the same vertex, where any $\langle w_i, w_{i+1} \rangle$ is an edge in the graph. w_1 is the least numbered vertex in the clow, and is called the **head** of the clow. We also require that the head occurs only once in the clow. This means that there is exactly one incoming edge ($\langle w_l, w_1 \rangle$) and one outgoing edge ($\langle w_1, w_2 \rangle$) at w_1 in the clow.

A **clow sequence** is a sequence of clows $\mathcal{W} = \langle C_1, \ldots, C_k \rangle$ with two properties.
The sequence is ordered: $\text{head}(C_1) < \text{head}(C_2) < \ldots < \text{head}(C_k)$
The total number of edges (counted with multiplicity) adds to exactly n.

A cycle cover is a special type of clow sequence. We will now show how to associate a sign with a clow sequence which is consistent with the definition of sign of a cycle cover. The sign of a cycle cover can be show to be $(-1)^{n+k}$, where n is the number of vertices in the graph and k is the number of components in the cycle cover. The sign of a clow sequence is defined to be $(-1)^{n+k}$ where n is the number of vertices in the graph and k is the number of clows in the sequence.

We will also associate a weight with a clow sequence which is consistent with the contribution of a cycle cover. The weight of a clow C, $w(C)$, is the product of the weights of the edges in the walk while accounting for multiplicity. For example, $w(\langle 1, 2, 3, 2, 3 \rangle) = a_{12}a_{23}^2a_{32}a_{31}$. The weight of a clow sequence $\mathcal{W} = \langle C_1, \ldots, C_k \rangle$ is $w(\mathcal{W}) = \prod_i w(C_i)$.

Theorem 1

$$det(A) = \sum_{\mathcal{W}:\ a\ clow\ sequence} sgn(\mathcal{W})w(\mathcal{W})$$

Proof:

We prove this by showing that the contribution of clow sequences that are *not* cycle covers is zero. Consequently, only the cycle covers contribute to the summation, yielding exactly the determinant.

Our proof defines an involution on a signed set. An involution φ on a set is a bijection with the property that φ^2 is the identity map on the set. The domain is the set of all clow sequences, and their signs define a natural partition of the domain into two.

We will now define an involution on this signed set. It has the property that a clow sequence that is not a cycle cover is paired with another clow sequence over the same multiset of edges but with opposing sign. The fixed points of the involution are precisely the cycle covers. This would establish the theorem.

The desired involution is the following. Let $\mathcal{W} = \langle C_1, \ldots, C_k \rangle$ be a clow sequence. Choose the smallest i such that C_{i+1} to C_k is a set of disjoint (simple) cycles. If $i = 0$, the involution maps $\mathcal{W}$ to itself. These are obviously cycle covers and the only fixed points. Otherwise, having chosen i, traverse C_i starting from the head until one of two things happen.

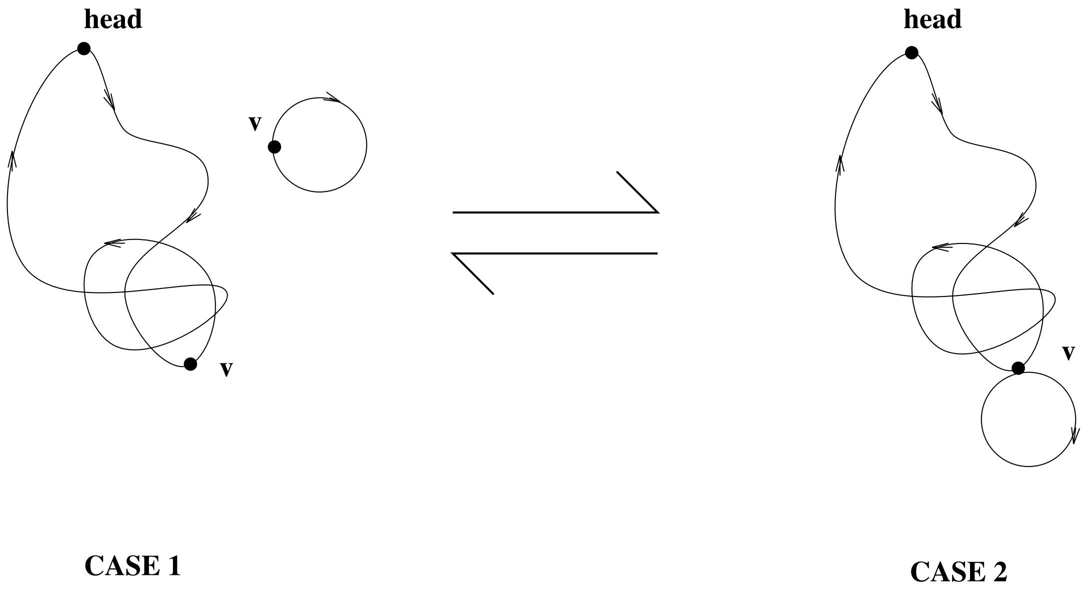

Figure 1: Pairing clow sequences of opposing signs

1. We hit a vertex that touches one of C_{i+1} to C_k.

2. We hit a vertex that completes a simple cycle within C_i.

Let us call the vertex v. Given the way we chose i, such a v must exist. Vertex v cannot satisfy both of the above conditions: If v completes a cycle *and* it touches cycle C_j, its previous occurrence (which exists, or else there can be no cycle at v) also touches C_j and the traversal would have stopped at that occurrence.

Case 1: Suppose v touches C_j. We map $\mathcal{W}$ to a clow sequence

$$\mathcal{W}' = \langle C_1, \ldots, C_{i-1}, C_i', C_{i+1}, \ldots, C_{j-1}, C_{j+1}, \ldots C_k \rangle$$

The modified clow, C_i' is obtained by merging C_i and C_j as follows: insert the cycle C_j into C_i at the first occurence (from the head) of v. For example, let $C_i = \langle 8, 11, 10, 14 \rangle$ and $C_j = \langle 9, 10, 12 \rangle$. Then the new clow is $\langle 8, 11, 10, 12, 9, 10, 14 \rangle$. Figure 1 illustrates the mapping.

The head of C_i' is clearly the head of C_i. The new sequence has the same multi-set of edges and hence the same weight as the original sequence. It also has one component less than the original sequence.

In the new sequence, vertex v in cycle C_i' would have been chosen by our traversal and it satisfies case 2.

Case 2: Suppose v completes a simple cycle C in C_i. By our earlier argument, cycle C cannot touch any of the later cycles. We now modify the sequence $\mathcal{W}$ by deleting C from C_i and introducing C as a new clow in an appropriate position, depending on the minimum labeled vertex in C, which we make the head of C. For example, let $C_i = \langle 8, 11, 10, 12, 9, 10, 14 \rangle$.

Then C_i changes to $\langle 8, 11, 10, 14\rangle$ and the new cycle $C = \langle 9, 10, 12\rangle$ is inserted in the clow sequence.

To show that the modified sequence continues to be a clow sequence, note that the head of C is greater than the head of C_i and hence C occurs *after* C_i. Also, the head of C is distinct from the heads of C_j ($i < j \leq k$). In fact, C is disjoint from all cycles C_j ($i < j \leq k$). Further, the new sequence has the same multi-set of edges and hence the same weight as the original sequence. It also has one component more than the original sequence.

Figure 1 illustrates the mapping. In the new sequence, vertex v in cycle C_i' would have been chosen by our traversal and it satisfies case 1.

In both of the above cases, the new sequence constructed maps back to the original sequence. Therefore the mapping is a weight-preserving involution. Furthermore, the number of clows in the two sequences differ by one, and hence the signs are opposing. This completes the proof. ■

Corollary 1.1

$$det(A) = \sum_{\mathcal{W}:\ \text{a clow sequence with head of first clow 1}} sgn(\mathcal{W})w(\mathcal{W})$$

Proof: In the involution defined above, the head of the first clow in the clow sequence remains unchanged. And the head of the first cycle in any cycle cover must be the vertex 1. ■

3 The Sequential Algorithm

Given an $n \times n$ matrix A, we define a layered directed acyclic graph H_A with three special vertices s, t_+ and t_-, having the following property:

$$det(A) = \sum_{\rho:\ s \rightsquigarrow t_+ \text{ path}} w(\rho) - \sum_{\eta:\ s \rightsquigarrow t_- \text{ path}} w(\eta)$$

Here the weight of a path is simply the product of the weights of the edges appearing in it. The idea is that $s \rightsquigarrow t_+$ ($s \rightsquigarrow t_-$) paths will be in one-to-one correspondence with clow sequences of positive (negative) sign.

The vertex set of H_A is $\{s, t_+, t_-\} \cup \{[p, h, u, i] \mid p \in \{0, 1\}, h, u \in \{1, \ldots, n\}, i \in \{0, \ldots, n-1\}\}$. If a path from s reaches a vertex of the form $[p, h, u, i]$, this indicates that in the clow sequence being constructed along this path, p is the parity of n + the number of components already constructed, h is the head of the clow currently being constructed, u is the vertex which the current clow has reached, and i edges have been traversed so far (in this and preceding clows). Finally, an $s \rightsquigarrow t_+$ ($s \rightsquigarrow t_-$) path will correspond to a clow sequence where n + the number of components in the sequence is even (odd).

The edge set of H_A consists of the following types of edges:

1. $\langle s, [b, h, h, 0]\rangle$ for $h \in \{1, \ldots, n\}$, $b = n \bmod 2$; this edge has weight 1.

2. $\langle [p, h, u, i], [p, h, v, i+1]\rangle$ if $v > h$ and $i + 1 < n$; this edge has weight a_{uv},

3. $\langle [p,h,u,i], [\overline{p},h',h',i+1]\rangle$ if $h' > h$ and $i+1 < n$; this edge has weight a_{uh},

4. $\langle [p,h,u,n-1], t_+\rangle$ if $p=1$; this edge has weight a_{uh},

5. $\langle [p,h,u,n-1], t_-\rangle$ if $p=0$; this edge has weight a_{uh}.

Theorem 2 *For an n dimensional matrix A, let H_A be the graph described above. Then*

$$det(A) = \sum_{\rho:\ s \rightsquigarrow t_+\ path} w(\rho) - \sum_{\eta:\ s \rightsquigarrow t_-\ path} w(\eta)$$

Proof: We will establish a one-to-one correspondence between $s \rightsquigarrow t_+$ ($s \rightsquigarrow t_-$) paths and clow sequences of positive (negative) sign, preserving weights. The result then follows from Theorem 1.

Let $\mathcal{W} = \langle C_1, \ldots, C_k\rangle$ be a clow sequence of positive sign (i.e., $n+k$ is even). We will demonstrate a path from s to t_+ in H_A. Let h_i be the head of clow C_i, and let n_i be the number of edges in clows $C_1, \ldots, C_{i-1}$. The path we construct will go through the vertices $[p, h_i, h_i, n_i]$, where $p=0$ if $n+i$ is odd and $p=1$ otherwise. From s, clearly we can go to the first such vertex $[n \bmod 2, h_1, h_1, 0]$. Assume that the path has reached $[p, h_i, h_i, n_i]$. Let the clow C_i be the sequence $\langle h_i, v_1, \ldots, v_{l-1}\rangle$, a closed walk of length l. Starting from $[p, h_i, h_i, n_i]$, H_A has a path through vertices $[p, h_i, v_1, n_i+1]$, $[p, h_i, v_2, n_i+2]$, …, $[p, h_i, v_{l-1}, n_i+(l-1)]$, and finally $[\overline{p}, h_{i+1}, h_{i+1}, n_i+l]$ which is the vertex $[\overline{p}, h_{i+1}, h_{i+1}, n_{i+1}]$. At the last clow, starting from $[1, h_k, h_k, n_k]$, H_A will have a path tracing out the vertices of clow C_k and finally making a transition to t_+. Clearly, the weight of the path is identical to the weight of the clow sequence. See Figure 2.

Conversely, let ρ be an $s \rightsquigarrow t_+$ path in H_A. In the sequence of vertices visited along this path, the second component of the vertex labels is monotonically non-decreasing and takes, say, k distinct values $h_1, \ldots, h_k$. Also, the first component changes exactly when the second component does, and is $n \bmod 2$ at h_1 and 1 at h_k (to allow an edge to t_+), so $n+k$ must be even. Consider the maximal segment of the path with second component h_i. The third components along this segment constitute a clow with leader h_i in G_A. When this clow is completely traversed, a new clow with a larger head must be started, and the parity of number of components must change. But this is precisely modelled by the edges of H_A. Therefore, ρ corresponds to a clow sequence in G_A of positive sign.

A similar argument shows the correspondence between paths from s to t_- and clow sequences with negative sign, preserving weights.

■

Now, to evaluate $det(A)$, we merely need to evaluate the weighted sums of paths. But this can easily be done by simple dynamic programming techniques; we give a polynomial time algorithm which evaluates this expression and hence computes $det(A)$.

We say that a vertex $[p,h,u,i]$ is at layer i in H_A. t_+ and t_- are at layer n. The algorithm proceeds by computing, in stages, the sum of weighted paths from s to any vertex at layer i in H_A. After n stages, it has the values at t_+ and t_-, and hence $det(A)$. See Table 1.

This algorithm processes each edge in H_A exactly once, and for each edge, it performs one addition and one multiplication. The total number of vertices in H_A is $2n^3+3$. However H_A is quite a sparse graph; the total number of edges is at most $4n^4$. The overall running

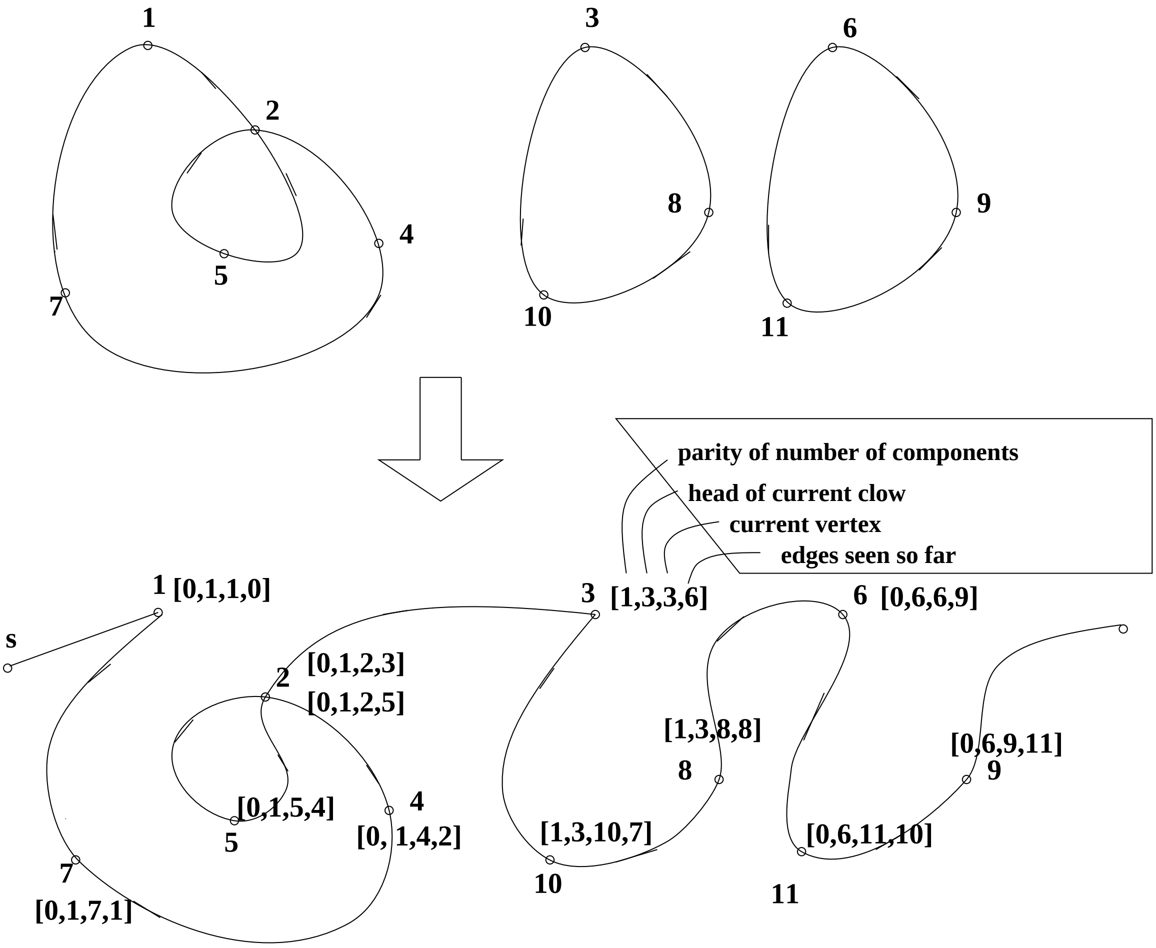

Figure 2: From a clow sequence to a path

```
# Initialise values to 0
For u, v, i ∈ [n], p ∈ {0,1} do V([p, u, v, i − 1]) = 0
V(t_+) = 0
V(t_−) = 0
# Set selected values at layer 0 to 1
b = n mod 2
For u ∈ [n], do V([b, u, u, 0]) = 1
# Process outgoing edges from each layer
For i = 0 to n − 2 do
    For u, v ∈ [n] such that u ≤ v, and p ∈ {0,1}, do
        For w ∈ {u + 1, ..., n} do
            V([p, u, w, i + 1]) = V([p, u, w, i + 1]) + V([p, u, v, i]) · a_vw
            V([p̄, w, w, i + 1]) = V([p̄, w, w, i + 1]) + V([p, u, v, i]) · a_vu
For u, v ∈ [n] such that u ≤ v, and p ∈ {0,1}, do
    V(t_+) = V(t_+) + V([1, u, v, n − 1]) · a_vu
    V(t_−) = V(t_−) + V([0, u, v, n − 1]) · a_vu
# Compute the determinant
Return V(t_+) − V(t_−)
```

Table 1: A sequential algorithm for the determinant

time is therefore $O(n^4)$. In fact, if G_A has m edges, then H_A has only $O(mn^2)$ edges, so for sparse matrices, the algorithm is faster.

The number of operations, addition or multiplication, is $O(n^4)$. The largest partial product at any stage is $m^n|a_{\max}|^n$, where $a_{\max}$ is the largest entry in A, m is the number of edges in G_A, and m^n is an upper bound on the number of clow sequences. This can be represented with $N = n\log m + n\log|a_{\max}|$ bits, so each operation needs at most $M(N)$ time, where $M(t)$ is the time required to multiply two t bit numbers. Clearly, even in terms of bit complexity, the algorithm needs only polynomial time.

The space used in this implementation is also polynomial; however, it is only $O(n^2)$, since at any stage, the values at only two adjacent layers need to be stored. (Again, there are $O(n^2)$ values to be stored; each may require upto N bits.)

4 Computing the Characteristic Polynomial

Our technique can be used as easily to compute *all* coefficients of the characteristic polynomial $\Phi_A(\lambda) = det(\lambda I_n - A) = c_n\lambda^n + c_{n-1}\lambda^{n-1} + \ldots + c_1\lambda + c_0$ of the matrix A. In fact, we will show that the graph defined in the previous section already does so. Rewriting $det(\lambda I_n - A)$ in terms of cycle covers and regrouping terms, we see that the coefficient of λ^r in $\Phi_A(\lambda)$, c_r, can be computed by summing, over all permutations σ with at least r fixed points, the weight of the permutation outside these fixed points.

$$c_r = \sum_{S\subseteq[1,\ldots,n]:|S|=r} \quad \sum_{\sigma\in S_n: j\in S\Rightarrow\sigma(j)=j} sgn(\sigma)\prod_{i\notin S}(-a_{i\sigma(i)})$$

If σ has only fixed points in S, let $sgn(\sigma|S)$ denote the parity of the number of components of σ, not counting the fixed point cycles of S ($+1$ if the parity is even, and -1 otherwise). Then

$$c_r = \sum_{S\subseteq[1,\ldots,n]:|S|=r} \quad \sum_{\sigma\in S_n: j\in S\Rightarrow\sigma(j)=j} sgn(\sigma|S)\prod_{i\notin S}(a_{i\sigma(i)})$$

But each term here is the weight of a partial cycle cover, covering exactly $n-r$ vertices. To compute this sum, not surprisingly, we look at partial clow sequences! An l-clow sequence is a sequence of clows (ordered by strictly increasing head) with total number of edges exactly l, accounting for multiplicity. Its sign is $(-1)^k$, where k is the number of clows in the sequence. The involution on the set of l-clow sequences is defined in the same fashion as in Section 2, and shows that the net contribution of sequences which are not partial cycle covers is zero. So now instead of H_A, we construct $H_A(r)$ with $n-r$ layers, with paths corresponding to $(n-r)$-clow sequences. We then compute the weights of $s \rightsquigarrow t_+$ and $s \rightsquigarrow t_-$ paths in this reduced graph, and report the difference as c_r.

Actually all coefficients can be computed using the single graph H_A, by introducing n copies of t_+ and t_-, one for each coefficient. Further, in this graph, if the ith layer reports a non-zero value, we can immediately conclude that the matrix has rank at least i. However, we do not know how to infer small rank from this construction.

Note that our graphs H_A, or $H_A(r)$ for computing the coefficient c_r, have a very regular structure: the edge connectivity across layers is identical. By dropping layer information from the vertices, we can construct a graph (finite-state automaton) with $O(n^2)$ vertices. Then to compute c_r, we find the contribution of $s \rightsquigarrow t_+$ or t_- paths in this graph of length exactly $n-r$.

5 Improving the algorithm

The algorithm of section 3 can be made more efficient if the number of vertices and edges in the graph H_A can be pruned. One simple saving is obtained by noting that we do not really need two copies of each vertex $[p,u,v,i]$ for the two values of p. Instead, we can keep one copy, and where in the original H_A this component was to be changed via an edge $\langle[p,u,v,i],[\overline{p},x,y,i+1]\rangle$ of weight w, we now have an edge from $[u,v,i]$ to $[x,y,i+1]$ with weight $-w$. This reduces the number of vertices by a factor of 2. More crucially, it allows the dynamic programming algorithm to do subtractions and cancellations at earlier layers. So the sizes of partial products, and hence the bit complexity, reduces.

Another pruning which also results in a saving by a constant factor (but with a larger constant than above) follows from this simple observation: paths going through vertices of the form $[p,h,u,i]$ with $h > i+1$ cannot correspond to cycle covers. This is because in a cycle cover, all vertices are covered exactly once, so at layer i, with $n-i$ vertices still to be covered, the head (minimum element) of the current cycle cannot be greater than i. Alternatively, once h becomes the head, at least $h-1$ edges should have been seen in preceding cycles. We can require our clow sequences also to satisfy this property. We formalize the prefix property: a clow sequence $\mathcal{W} = \langle C_1, \ldots, C_k \rangle$ has the prefix property if for $1 \leq j \leq k$, the total lengths of the clows $C_1, \ldots, C_{j-1}$ is at least $\text{head}(C_j) - 1$.

It is easy to verify that the involution defined in Section 2 also works on such restricted clow sequences: $\varphi(\mathcal{W})$ has the prefix property if and only if $\mathcal{W}$ does. So we may instead construct a pruned version of H_A which generates only such clow sequences. (Consider the induced subgraph of H_A obtained by deleting all vertices $[p, h, u, i]$ where $h > i + 1$.) This will lead to an algorithm with essentially the same complexity, but smaller constants. But pruning is not without its drawbacks: we can no longer directly extract the coefficients of the characteristic polynomial.

Interestingly, clow sequences with the prefix property are precisely the terms computed by Samuelson's method for computing the determinant. As observed by Valiant[1] in [Va 92], the correctness of Samuelson's algorithm gives a proof, based on linear algebra, that such sequences which are *not* cycle covers "cancel" out. Our involution gives a combinatorial proof of this fact (for details, see [MV 97]).

Of course, if the algorithm is to be used to compute the coefficient c_r of the characteristic polynomial, then we cannot use this kind of prefix property. The right prefix property for computing c_r would be that in the clow sequence $\mathcal{W} = \langle C_1, \ldots, C_k \rangle$, for $1 \leq j \leq k$, the total lengths of the clows $C_1, \ldots, C_{j-1}$ is at least $\text{head}(j) - 1 - r$. The graph $H_A(r)$ can be pruned consistent with this property. However, if a single graph is to be used to compute all coefficients, then we must work with the unpruned version.

6 Parallel Algorithms: GapL and NC implementations

In this section, we describe two different approaches towards obtaining parallel algorithms which exploit the combinatorial Theorem 1. The first approach is to apply the standard divide-and-conquer technique to compute the contributions of all clow sequences, and so directly obtain an NC or PRAM algorithm. The second approach is indirect; we show how our algorithm places integer determinant in the class of functions GapL, and then appeal to standard parallelizations of GapL functions. This approach is particularly interesting from the complexity-theory point of view, since the GapL implementation gives a very good instance of how to effectively use nondeterminism in a space-bounded computation.

6.1 PRAM and NC algorithms

The signed weighted sum of all clow sequences can be evaluated in parallel using the standard divide-and-conquer technique, giving an NC^2 algorithm for the determinant. We describe the algorithm below. We first show how to construct an arithmetic SAC^1 circuit for computing the determinant (an arithmetic polynomially sized circuit with $O(\log n)$ depth where the $+$

[1] There is a minor technical error in Valiant's formulation. He claims that Samuelson's algorithm generates all clow sequences, referred there as loop covers. However his preceding discussion makes it clear that clow sequences without prefix property are not generated.

gates have unbounded fanin but each $\times$ gate has constant fanin). We then show how to implement this circuit as an EREW PRAM algorithm requiring $O(\log^2 n)$ parallel time. We also analyze the bit complexity of the algorithm, and show an implementation in Boolean NC^2.

The goal is to sum up the contribution of all clow sequences at the output gate of the circuit. The output gate is a sum, over all $1 \leq k \leq n$, of C_k, where C_k is the sum of the contributions of all clow sequences with exactly k clows. To compute C_k, we use a divide-and-conquer approach on the number of clows: any clow sequence contributing to C_k can be suitably split into two partial clow sequences, with the left sequence having $2^{\lceil \log k \rceil - 1}$ clows. The heads of all clows in the left part must be less than the head of the first clow in the rightmost part. And the lengths of the left and the right partial clow sequences must add up to n. We can carry this information in the gate label. Let gate $g[p, l, u, v]$ sum up the weights of all partial clow sequences with p clows, l edges, head of first clow u, and heads of all clows at most v. (We need not consider gates where $l < p$ or $u > v$.) Then $C_k = g[k, n, 1, n]$, $D_k = (-1)^{n+k} C_k$, and the output is $\sum_{k=1}^{n} D_k$. Further,

$$g[p, l, u, v] = \begin{cases} \displaystyle\sum_{\substack{2^q \leq r \leq 2^q + (l-p) \\ u < w \leq v}} g[2^q, r, u, w-1] \cdot g[p - 2^q, l - r, w, v] & \text{if } p > 1 \\ g[l, u] & \text{if } p = 1 \end{cases}$$

where $q = \lceil \log p \rceil - 1$, i.e. $2^q < p \leq 2^{q+1}$. The gate $g[l, u]$ sums up the weights of all clows of length l with head u. This gate is also evaluated in a divide-and-conquer fashion. A clow with head u is either a self-loop if $l = 1$, or it must first visit some vertex $v > u$, find a path of length $l - 2$ to some vertex $w > u$ through vertices all greater than u, and then return to u. So

$$g[l, u] = \begin{cases} a_{uu} & \text{if } l = 1 \\ \sum_{v > u} a_{uv} \cdot a_{vu} & \text{if } l = 2 \\ \sum_{v, w > u} a_{uv} \cdot c[l-2, u, v, w] \cdot a_{wu} & \text{otherwise} \end{cases}$$

The gate $c[l, u, v, w]$ sums the weights of all length l paths from v to w going through vertices greater than u. The required values can be computed in $O(\log n)$ layers as follows:

$$\begin{aligned} c[1, u, v, w] &= a_{vw} \\ c[2^s + i, u, v, w] &= \textstyle\sum_{x > u} c[2^s, u, v, x] \cdot c[i, u, x, w] \qquad \text{for } s = 0 \text{ to } \lceil \log n \rceil - 1,\ 1 \leq i \leq 2^s \end{aligned}$$

The final circuit description is summarized in a bottom-up fashion in Table 2.

Assigning a gate to each variable $c[l, u, v, w]$ or $g[l, u]$ or $g[p, l, u, v]$ or D_k in this algorithm, we obtain an arithmetic circuit with $O(n^4)$ gates and depth $O(\log n)$. Each $+$ gate has fanin at most $O(n^2)$, and each $\times$ gate has fanin at most 3. Therefore this is an arithmetic SAC^1 circuit.

To obtain an EREW PRAM implementation, we must first eliminate the large fanin and large fanout gates. Suppose we replace each $+$ gate having fanin f by a binary tree of $+$ gates (i.e. a bounded-fanin subcircuit). The number of edges only doubles, and the depth increases by $\log f$. To handle fanout, we reverse the process: we introduce an inverted binary tree of "copy" gates above each gate (a bounded-fanout subcircuit). Again, the number of edges

```
# Initialise values for paths of length one
For u, v, w ∈ [n] : v, w > u do in parallel
    c[1, u, v, w] = a_vw
# Evaluate values of paths of lengths upto 2^s
For s = 0 to ⌈log n⌉ − 1
    For 1 ≤ i ≤ 2^s and for u, v, w ∈ [n] : v, w > u do in parallel
        c[2^s + i, u, v, w] = Σ_{x>u} c[2^s, u, v, x] · c[i, u, x, w]
# Evaluate values of single clows
For l, u ∈ [n] do in parallel
    If l = 1 then g[l, u] = a_uu
    If l = 2 then g[l, u] = Σ_{v>u} a_uv · a_vu
    If l > 2 then g[l, u] = Σ_{v,w>u} a_uv · c[l − 2, u, v, w] · a_wu
# Initialise values of partial clow sequences with one clow
For l, u, v ∈ [n] : u ≤ v do in parallel
    g[1, l, u, v] = g[l, u]
# Evaluate values of partial clow sequences with upto 2^s clows
For s = 0 to ⌈log n⌉ − 1
    For 1 ≤ i ≤ 2^s and for l, u, v ∈ [n] : (u ≤ v) ∧ (l ≥ 2^s + i) do in parallel
        g[2^s + i, l, u, v] = Σ_{2^s ≤ r ≤ l−i; u < w ≤ v} g[2^s, r, u, w − 1] · g[i, l − r, w, v]
# Evaluate the sum of clow sequences
For k ∈ [n] do in parallel
    D_k = (−1)^{n+k} g[k, n, 1, n]
# Evaluate the determinant
Return det(A) = Σ_{k=1}^{n} D_k
```

Table 2: An arithmetic SAC^1 algorithm for the determinant

only doubles. Note that in the SAC^1 circuit described above, the maximum fanin of any $+$ gate is $O(n^2)$. So the total number of edges in the circuit is $O(n^6)$. So this procedure applied to the circuit gives a bounded-fanin bounded-fanout circuit of depth $O(\log^2 n)$ with $O(n^6)$ edges. Now the standard technique of placing a processor on each edge gives an EREW PRAM algorithm requiring $O(n^6)$ processors and running in $O(\log^2 n)$ parallel time. And the equally standard technique of reusing processors across layers will give an EREW PRAM algorithm performing $O(n^6)$ work and running in $O(\log^2 n)$ parallel time.

Now let us consider the bit complexity of the above algorithm. (In the PRAM model, each addition and each multiplication was considered to be a unit cost operation.) All operations in the SAC^1 circuit are on N-bit numbers; recall that $N = n \log n + n \log |a_{\max}|$, where $a_{\max}$ is the largest entry in A. Each operation in the above algorithm involves either adding n^2 numbers or multiplying 3 numbers, and can be performed in NC^1. Plugging in these Boolean circuits at each gate gives a Boolean NC^2 circuit computing the determinant.

6.2 Integer Determinant and GapL

In this section we demonstrate how #L functions (first studied in [AJ 93]) can be used to compute the determinant. In particular, we show that the determinant can be expressed as the difference of two #L functions. This, coupled with the fact that functions in #L can be computed in Boolean NC^2 (and subtraction is in NC^1), gives us another approach towards building small-depth circuits for the determinant.

To place things in perspective, we first sketch the history of research efforts directed towards the connections between the determinant and the function class #L. We denote by `Det` the function which, given a matrix with integer entries, evaluates to the determinant of the matrix.

Cook [Co 85] introduced NC^1-reductions; he used NC^1-Turing reductions to formally define and study many parallel complexity classes. The decision to use reductions with oracle gates as opposed to many-one NC^1-reductions is deliberate; in Cook's view oracle reductions are more suitable for the study of functions than many-one reductions. Among the parallel complexity classes introduced, Cook defined DET^* to be the class of functions NC^1-reducible to `Det`. Cook also listed many complete problems for DET^*; the most important of them was Iterated Matrix Product (over integers). He observed that the Samuleson's method in fact establishes an NC^1-reduction from `Det` to `IterMatProd`. In the other direction, `IterMatProd`$\leq$ `MatrixPowering` $\leq$ `MatrixInversion` $\leq$ `Det`. (All reductions are NC^1 reductions.)

Unfortunately, the relation $NC^1 \subseteq$ DLOG does not relativize [Wi 87]. This caused some confusion earlier on in many papers dealing with the determinant as a complexity class.

In particular, Damm's claim that the logspace many-one reduction closure of `Det` is equivalent to $L^{\#L}$, and Vinay's claim that DET^* is equivalent to $L^{\#L}$, are both in error. Also, Immerman and Landau erroneously claimed in the conference version of their paper [IL 95] that the quantifier-free-projection closure of `Det`, qfp(`Det`), equals DET^*.

Buntrock et al [BDHM 92] show $L^{\#L}$ is contained in DET^*. Vinay and Damm use the following chain to establish their results: `IterMatProd` $\leq$ DiffL $\subseteq$ $L^{\#L}$ $\subseteq$ DET^*. Wilson's results [Wi 87] imply that "`IterMatProd` is complete for DET^*" is inadequate to show "DET^* equals $L^{\#L}$". And the proof technique of Buntrock et. al. [BDHM 92] fails when DET^* is replaced with the weaker logspace many-one reduction closure of determinant.

In fact, the correct statement should be: [Vi 91a, To 91]

Theorem 3 $DET^* = NC^1(\#L)$.

Immerman and Landau [IL 95] and Toda [To 91] observe that the Samuelson-Berkowitz algorithm is in fact a (first-order) projection from `Det` to `IterMatProd`. Consequently, by defining (a possibly new class) DET as the logspace many-one closure of the determinant, and by defining GapL as the difference of two #L functions, one can hope for the following theorem:

Theorem 4 GapL = DET.

This result was essentially discovered independently by Vinay, Damm and Toda (see [Vi 91, Vi 91a, Da 91, To 91, To 92]) around the same time. The proofs are somewhat different, though. Damm, inspired by Babai and Fortnow's [BF 91] characterization of #P, used

```
Nondeterministically choose head ∈ {1,...,n}.
current = head
count = 0
If n is odd, then parity = 1 else parity = 0
# [parity, head, current, count] is the vertex traced out in a clow sequence
#     of G_A or an s ~> t_+ or s ~> t_- path in H_A.
While count < n - 1 do
        Nondeterministically choose next ∈ {head,...,n}.
        count= count + 1
        Branch l-ways, where l = A[current,next].
        If next > head then current = next
                        else  Nondeterministically choose newhead ∈ {head + 1,...,n}.
                              parity = (parity + 1) mod2
                              head = newhead
                              current = head
# At this point, count = n - 1.
Branch l-ways, where l = A[current,head].
parity = (parity + 1) mod2
If parity = 0 then accept, otherwise reject.
```

Table 3: A GapL algorithm for the determinant over non-negative integers

(positive) arithmetic straight line program with restricted multiplications ((P)ARM model) to characterize #L. These programs are equivalent to skew arithmetic circuits; in fact, in the #P setting, this equivalence was already known (see [BF 91]). Toda and Vinay show how Det is equivalent to the difference between the number of (s,t) paths in two directed acyclic graphs. All the proofs rely on the Samuelson-Berkowitz algorithm. We will now present a complete proof of this theorem.

Proof: In one direction, the result can be seen in a particularly direct way from our sequential implementation of clow sequences in section 3. Observe that given A, the construction of H_A is logspace-uniform, and tracing out $s \rightsquigarrow t_+$ or $s \rightsquigarrow t_-$ paths can be done by an NL machine. (We must be careful here: the partial products along a path can be too large to be stored in logspace. The NL machine, to traverse an edge of weight $|w|$, should simply branch into w paths and remember only the sign. This way, a path of weight w in H_A will generate $|w|$ accepting/rejecting paths of the NL machine.) Essentially, H_A gives us a uniform polynomial size polynomial width branching program, corresponding precisely to GapL.

Table 3 gives the code for an NL machine computing, through its gap function, the determinant of a matrix A with non-negative integral entries. (Negative integers can be accomodated by appropriately updating the parity variable.) Most steps are easily seen to be possible on an NL machine. The l-way branching is the only step which has to be done carefully. Since l could be n bits long, it cannot be stored on the worktape. Instead, the NL machine has to step through the bit description of l on the input tape and branch accordingly.

Input: k bits $a_{k-1}, \ldots, a_1, a_0$ specifying the number $l = \sum_{i=0}^{k-1} 2^i a_i$

Goal: To produce exactly l paths ending in some prespecified configuration C; any extra paths produced should be rejecting.

NL algorithm

$j = 0$

While $j < k$ do

Branch 3-ways

On Branch 1, if $a_j = 0$ then reject and exit loop

otherwise enter configuration C and exit loop

On Branches 2 and 3, increment j

Endwhile

Table 4: NL code to produce l accepting paths, given l in binary.

The details are shown in Table 4.

In the other direction, we follow Toda, who follows Valiant [Va 79]. It suffices to show that counting $s \rightsquigarrow t$ paths in an acyclic graph G_1 can be reduced to computing the determinant of a matrix. We first replace each edge in G_1 by a path of length 2, add edges $\langle t, s\rangle$ and $\{\langle u, u\rangle \mid u \notin \{s, t\}\}$. Now an $s \rightsquigarrow t$ path in G_1 corresponds exactly to a cycle cover in this new graph G_2. And all cycle covers in G_2 have positive sign. So the number of $s \rightsquigarrow t$ paths in G_1 equals $\det(\mathrm{adj}(G_2))$ equals $\mathrm{perm}(\mathrm{adj}(G_2))$ (by $\mathrm{adj}(G_2)$ we mean the adjacency matrix of the graph G_2). ■

What is striking is a complexity theoretic analog of the classical `Det` vs `Perm` problem: they are complete for GapL and GapP respectively. A corollary of the the proof above shows that for every integer matrix A, there is an integer matrix B whose dimensions are polynomially related to the dimensions of A, such that the determinant of A is the permanent of B. Of course, we do not know if `Perm` can be reduced to `Det` in logspace/polynomial time.

The future of the class DET* is not clear. Allender and Ogihara [AO 94] note that there is no reason to believe that NL is a subset of GapL(=DET). (We mean subset in the following sense: a language is in GapL if its characteristic function is in GapL.) This is because the 0-1 valued functions in DET must differ in at most one accepting path. (However, a recent result of Reinhardt and Allender [RA 97] shows that the inclusion is true in a non-uniform setting.) On the other hand, notice that NL is contained in DET*. In fact, Allender and Ogihara [AO 94] consider AC^0(`Det`) and show that AC^0(`Det`) corresponds to a certain counting hierarchy (LH) that may be defined on #L. They also claim (without proof) that if AC^0(`Det`) and DET* coincide, LH collapses.

Two nice applications of the GapL characterization in Theorem 4 have been in the drastic simplification of Jung's theorem on PL (see [AO 94]) and in characterizing the complexity of computing the rank of a matrix (see [ABO 96]).

Acknowledgments

We thank Eric Allender for setting us on the track of pursuing the determinant problem, and for supplying us with several interesting references.

References

[ABO 96] E. Allender, R. Beals and M. Ogihara. The Complexity of Matrix Rank and Feasible Systems of Linear Equations, *Proc. 28th ACM Symposium on Theory of Computing* (STOC) (1996), 161–167.

[AO 94] E. Allender and M. Ogihara. Relationships among PL, #L, and the determinant, *RAIRO Theoretical Information and Applications*, **30** (1996), 1–21. Conference version in *Proc. 9th IEEE Structure in Complexity Theory Conference* (1994), 267–278.

[AJ 93] C. Àlvarez and B. Jenner. A Very Hard Log-space Counting Class, *Theoretical Computer Science*, **107** (1993), 3–30.

[Be 84] S. J. Berkowitz. On Computing the Determinant in Small Parallel Time Using a Small Number of Processors, *Information Processing Letters*, **18** (1984), 147–150.

[BDHM 92] G. Buntrock, C. Damm, U. Hertrampf and C. Meinel. Structure and Importance of Logspace MOD-classes, *Math. Systems Theory*, **25** (1992), 223–237.

[BF 91] L. Babai and L. Fortnow. Arithmetization: A New Method in Structural Complexity Theory, *Computational Complexity*, **1**(1), (1991), 41–66.

[Ch 85] A. L. Chistov. Fast Parallel Calculation of the Rank of Matrices over a Field of Arbitrary Characteristic, *Proc Int. Conf. Foundations of Computation Theory*, LNCS **199** (1985), 63–69.

[Co 85] S. Cook. A Taxonomy of Problems with Fast Parallel Algorithms, *Information and Control*, **64** (1985) 2–22.

[Cs 76] L. Csanky. Fast Parallel Inversion Algorithm, *SIAM J of Computing*, **5** (1976), 818–823.

[Da 91] C. Damm. DET=$L^{(\#L)}$, Informatik–Preprint 8, Fachbereich Informatik der Humboldt–Universität zu Berlin, 1991.

[FF 63] D. Fadeev and V. Fadeeva. Computational Methods in Linear Algebra, Freeman, San Francisco (1963).

[IL 95] N. Immerman and S. Landau. The Complexity of Iterated Multiplication, *Information and Control*, **116**(1) (1995), 103–116. Conference version in *Proc Structure in Complexity Theory 1989*.

[MV 97] M. Mahajan and V Vinay. Determinant: Old Algorithms, New Insights. Technical Report of the Institute of Mathematical Sciences, IMSc-TR97/08/.

[Ni 91] N. Nisan. Lower Bounds for Non-Commutative Computation, *Proc. 23th Annual Symposium on Theory of Computing* (STOC), (1991), 410–418.

[RA 97] K. Reinhardt and E. Allender. Making Nondeterminism Unambiguous, to appear in *Proc. 38th IEEE Foundations of Computer Science Conference* (FOCS), 1997.

[Sa 42] P. A. Samuelson. A Method of Determining Explicitly the Coefficients of the Characteristic Polynomial, *Ann. Math. Stat.*, **13** (1942), 424–429.

[St 83] H. Straubing. A Combinatorial Proof of the Cayley-Hamilton Theorem, *Discrete Maths.*, **43** (1983), 273–279.

[To 91] S. Toda. Counting Problems Computationally Equivalent to the Determinant, manuscript.

[To 92] S. Toda. Classes of Arithmetic Circuits Capturing the Complexity of Computing the Determinant, *IEICE Trans. Inf. and Syst.* , **E75-D** (1992), 116–124.

[Va 79] L. G. Valiant. The Complexity of Computing the Permanent, *Theoretical Computer Science*, **8** (1979), 189–201.

[Va 92] L. G. Valiant. Why is Boolean Complexity Theory Difficult? , in Boolean Function Complexity, ed M. S. Paterson, London Mathematical Society Lecture Notes Series 169, Cambridge University Press, 1992.

[Ve 92] H. Venkateswaran. Circuit Definitions of Nondeterministic Complexity Classes, *SIAM J. on Computing*, **21** (1992), 655–670.

[Vi 91] V Vinay. Counting Auxiliary Pushdown Automata and Semi-Unbounded Arithmetic Circuits, *Proc. 6th Structure in Complexity Theory Conference*, (1991), 270–284.

[Vi 91a] V Vinay. *Semi-unboundedness and complexity classes*, doctoral dissertation, Indian Institute of Science, Bangalore, July 1991.

[Wi 87] C. B. Wilson. Relativized NC, *Math. Systems Theory*, **20** (1987), 13–29.