Permanent is hard to compute even on a good day

Yuval Filmus

September 11, 2012

Abstract

We give an exposition of Cai, Pavan and Sivakumar's result on the hardness of permanent. They show that assuming it is hard to compute the permanent in the worst case, an algorithm cannot compute the permanent correctly even on (say) a $1/n$ fraction of the inputs.

1 Introduction

Complexity theory usually analyzes problems and algorithm from the worst-case point of view. However, in practice, one is often more interested in what happens on the average. Sometimes one can show that a specific problem, which is known to be hard in the worst-case, is also hard on average. One way of doing this is to give a reduction from an algorithm which works on average to an algorithm which works on an arbitrary input (with high probability, regardless of the input). In this talk we will do just that for the problem of computing the *permanent*.

The permanent of an $n \times n$ matrix A is

$$\operatorname{per} A = \sum_{\pi \in S_n} \prod_{i=1}^{n} a_{i\pi(i)}.$$

In other words, the permanent is defined like the determinant, only without the signs. This makes a big difference: while the determinant can be computed in polynomial time using Gaussian elimination, computing the permanent is hard for #P. Note, however, that when the underlying field has characteristic two, then the permanent is equal to the determinant, and so easy to compute. Instead, one often considers the problem of computing the permanent of a 0/1 matrix over the integers, or, equivalently, over $\mathbb{Z}_p$ for a prime p in the range $[n!, 2n!]$.

We will focus on the problem of computing the permanent of a matrix with arbitrary entries over a finite field which is "big enough". The size of the finite field, which we denote by q, will have to grow with n. This setting is somewhat unnatural, but the results can also be cast in terms of permanents over the integers.

2 Testing

Suppose you have written an algorithm that computes the permanent. How can you check your algorithm? Two ideas come to mind. The first idea uses the expansion of the permanent along the first row:

$$\operatorname{per} A = \sum_{i=1}^{n} a_{1i} A_{1i}, \tag{1}$$

where A_{ij} is the (i,j)-minor of A. This equation gives rise to the following algorithm:

- Compute $\operatorname{per} A$ and $\operatorname{per} A_{1i}$ for $i \in [n]$. Verify that (1) holds.
- Choose an arbitrary $i \in [n]$, and use the same procedure to verify the value of $\operatorname{per} A_{1i}$.

The procedure stops once the matrix gets small enough so that we can compute the permanent using a reliable algorithm.

The second idea, due to Lipton [4], uses the fact that the permanent is a sum of terms, each having n factors. This implies that for any two matrices A, B of size $n \times n$, the function $f(t) = \operatorname{per}(A + tB)$ is a polynomial of degree n. In order to test our algorithm, we evaluate $f(t)$ at many points (at least $n+1$), and check that the result corresponds to some polynomial of degree n.

3 Verification

Now suppose you're in a slightly different scenario: someone else has written an algorithm that computes the permanent, and you want to use it to compute the permanent of some matrix A. However, you don't trust the other party. Therefore you use the following interactive protocol, due to Lund, Fortnow, Karloff and Nisan [5] (they attribute this version of the protocol to Babai; see also Feige and Lund [2]).

- Obtain the value of $\operatorname{per} A$ and $\operatorname{per} A_{1i}$ for $i \in [n]$. Verify that (1) holds.
- Define the matrix
$$B_t = \sum_{i=1}^{n} \delta_i(t) A_{1i},$$
where δ_i is a polynomial of degree $n-1$ such that $\delta_i(i) = 1$ and $\delta_i(j) = 0$ for $j \in [n] - i$.
- Determine the $n(n-1)$-degree polynomial $f(t) = \operatorname{per} B_t$ by obtaining its value on $n(n-1)+1$ points.
- Generate a random t and run the procedure recursively on B_t.

As before, the procedure stops once the matrix gets so small that we can evaluate the permanent on our own. If the value of $\operatorname{per} A$ is wrong, then at least one of the values of $\operatorname{per} A_{1i}$ is wrong. Therefore the polynomial $f(t)$ we compute is wrong. The correct polynomial $g(t)$ also has degree $n(n-1)$, and so $f(t)$ and $g(t)$ can agree on at most $n(n-1)$ points. Therefore with probability $1 - n(n-1)/p$, $f(t) \neq \operatorname{per} B_t$ for the t we choose in the last step. If p is large enough (say at least n^4), then with high probability the mistake will propagate all the way down to a point where we can notice it by computing the permanent of the relevant matrix.

4 Average-case to worst-case

It's time to go back to our original goal, that of showing that the permanent is hard even on average. When does an algorithm solve the permanent on average? In order to quantify the quality of an algorithm, we consider its performance on random matrices (this is the reason we work over a finite field rather than over the integers). Arguably, we should say that an algorithm works on average if there is only a small fraction of inputs which confuse it.

Suppose, for example, that we have an algorithm $\mathcal{P}$ that computes the permanent of an $n \times n$ matrix correctly with probability $1 - 1/3n$. The following algorithm by Lipton [4] computes the permanent of an arbitrary $n \times n$ matrix A correctly with probability at least $2/3$. (To get a fix on the correct value, run the algorithm many times and take a plurality vote.)

- Let B be a random $n \times n$ matrix.
- Compute the degree-n polynomial $f(t) = \operatorname{per}(A + tB)$ by sampling $n + 1$ random points.
- Return $f(0)$.

For each $t \neq 0$, the matrix $A + tB$ is random, and so the value of $f(t)$ is correct with probability $1 - 1/3n$. Therefore $f(t)$ is computed correctly with probability roughly $2/3$.

5 Usual-case to worst-case

Lipton's algorithm stops working once the success probability of $\mathcal{P}$ is bounded away from 1. However, a very similar procedure works even if $\mathcal{P}$ succeeds only with probability $\alpha = 1/2 + \epsilon$. We need to make two adjustments to Lipton's algorithm. First, we need to make the matrices whose permanent is sampled more independent. Second, we need a procedure that allows us to compute the polynomial $f(t)$ in the presence of errors. Here is the resulting algorithm, due to Gemmell and Sudan [3].

- Let B, C be random $n \times n$ matrices.

- Evaluate the degree-$2n$ polynomial $f(t) = \mathrm{per}(A + tB + t^2C)$ by sampling N arbitrary non-zero points and using the Berlekamp-Welch decoding algorithm.
- Return $f(0)$.

Previously, we only needed to use the fact that the matrices $A + tB$ (for $t \neq 0$) are individually random. If $\mathcal{P}$ succeeds with high probability, then the union bound allows us to deduce that it succeeds on many of the matrices $A+tB$ at once. Here, however, the situation is different: we don't expect the algorithm to succeed on all of the instances. Rather, the best we can hope for is getting the correct answers on αN of the input points. In order to deduce such a *concentration of measure* property, we need to assume something stronger. The matrices $A + tB + t^2C$ are pairwise independent, and so Chebyshev's inequality allows us to deduce that with high probability, the number of correct answers is indeed close to αN.

Previously, we computed the polynomial $f(t)$ given $n + 1$ points $(t, f(t))$. We didn't mention how this is done, but one way is solving linear equations. In the present case, only αN of the pairs $(t, f(t))$ are correct, and so this method breaks down. We are in the setting of error-correcting codes. Given a set T of N inputs, each polynomial $P(t)$ of degree $d = 2n$ gives rise to the codeword $P(t)|_{t\in T}$. Two codewords can agree on at most d points, hence the code has minimal distance $N-d$. Coding theory tells us that we can hope to correct up to $(N-d-1)/2$ errors. In other words, we need to get at least $(1/2+(d+1)/2N)N$ of the points right. For every $\alpha > 1/2$, we can choose N large enough so that $\alpha > 1/2 + (d + 1)/2N$. Then we can at least hope to be able to determine the correct polynomial $P(t)$.

We note in passing that it is possible to sample N random points only if the underlying field has at least N elements. This is why we need the underlying field to be large enough.

Can we correct $(N - d - 1)/2$ errors? Berlekamp and Welch designed (and patented) an efficient algorithm that does just that, in the context of Reed-Solomon codes (these are codes in which the set T consists of the entire field). Here is how their algorithm works. Suppose $f(t)$ is the actual polynomial, and $(t, P(t))|_{t\in T}$ are our samples, of which those in $W \subset T$ are incorrect. Then for any $t \in T$,

$$E(t)P(t) = E(t)f(t), \quad \text{where } E(t) = \prod_{w\in W} (t - w).$$

We can convert this into linear equations in terms of the coefficients of $E(t)$ and of $Q(t) = E(t)f(t)$. In total, the number of unknowns is

$$2 \deg E + \deg f + 2 = N - d - 1 + d + 2 = N + 1.$$

Since the total number of equations is N, there is at least one non-trivial solution $(E(t), Q(t))$, from which we can compute $f(t) = Q(t)/E(t)$. Are there extraneous solutions? If there are two non-trivial solutions $(E_1(t), Q_1(t)), (E_2(t), Q_2(t))$,

then

$$E_1(t)Q_2(t) = E_1(t)E_2(t)P(t) = E_2(t)Q_1(t).$$

Since $\deg E_i + \deg Q_j = N - 1$ and the two polynomials $E_1(t)Q_2(t), E_2(t)Q_1(t)$ agree on N points, it follows that $E_1(t)Q_2(t) = E_2(t)Q_1(t)$ and so $Q_1(t)/E_1(t) = Q_2(t)/E_2(t)$.

6 Lucky-case to worst-case

Gemmell and Sudan's algorithm breaks when the success probability of $\mathcal{P}$ drops below $1/2$. The problem is that it is not possible to decode an error-correcting code unless at least half the symbols are correct. This is true for any code: suppose that C_1 and C_2 are two different codewords. Form a codeword C by taking the first half of C_1 and the second half of C_2. We can think of C both as coming from C_1 with only half the symbols correct, or as coming from C_2 with only half the symbols correct.

How prevalent is this situation, though? The code we considered above has q^{d+1} codewords, each lying inside a space of size q^N. The size of each Hamming ball of radius $N/2$ is roughly $2^N(q-1)^{N/2}$. Hence on average, a point in space is $N/2$-close to this many codewords:

$$\frac{q^{d+1}2^N(q-1)^{N/2}}{q^N}.$$

For reasonable values of q and N, this number is very small. Hence if we take a codeword and change half of its coordinates, then most of the time it will be $N/2$-close only to its original codeword. We get similar results even with a smaller fraction of correct symbols. This suggests the following heuristic algorithm, which can handle a smaller success probability of $\mathcal{P}$: run the same algorithm as Gemmell and Sudan's, using a different procedure to obtain $f(t)$. We expect that most of the time, we will be able to determine $f(t)$ uniquely.

The problem with this heuristic algorithm is that the pattern of errors can be complicated, since all we know about the values $A + tB + t^2C$ are that they are pairwise independent. Also, our reasoning assumed the results are random, whereas we know that they are somewhat close to a codeword. So we have to relax the demand that $f(t)$ be obtained uniquely. This leads us to the concept of *list decoding*. Using an algorithm of Sudan (which we describe below), for appropriate values of α and N it is possible to get a small list of possible values for $f(t)$. The list will often contain only one polynomial, but might potentially contain more. Sudan's algorithm guarantees that the size of the list will be reasonable (polynomial in n).

Given a list of possible values of $f(t)$, how do we know which one is correct? The trick is to use the LFKN protocol, described above, as part of the procedure. Here is the algorithm of Cai, Pavan and Sivakumar [1].

- Let B, C be random $n \times n$ matrices.

- Evaluate the degree-$n(n+1)$ polynomial

$$f(t) = \operatorname{per} M_t, \quad M_t = \sum_{i=1}^{n} \delta_i(t) A_{1i} + (B + tC) \prod_{i=1}^{n} (t - i)$$

  at N arbitrary points $t > n$.

- Using Sudan's (or Guruswami's) list decoding algorithm, obtain a small list F of possible polynomials.

- Find a point t^* on which all the polynomial in F disagree.

- Recursively compute the permanent of M_{t^*}.

- Find the unique polynomial $f \in F$ such that $f(t^*) = \operatorname{per} M_{t^*}$, and return

$$\sum_{i=1}^{n} a_{1i} f(i).$$

For $t > n$, the matrices M_t are pairwise independent, and so all the preceding reasoning still holds. Sudan's list decoding algorithm will leave us with a reasonably small list of possible polynomials of degree $d = n(n+1)$. Any two of them can agree on at most $n(n+1)$ points, so as long as $|F|^2 \leq 2q/n(n+1)$, we will be able to find a point t^* on which all polynomials in F disagree. By taking N large enough, the error probability in each step of the algorithm will be so small that altogether, the algorithm will succeed with high probability.

(An aside: Chebyshev's lemma gives quite mild guarantees on the fraction of errors among the N sample points, and so in order to guarantee a small overall success probability, we need to stop the process at some large constant n. Alternatively, we could use a random polynomial of degree n instead of $B + tC$. The resulting matrices M_t are n-wise independent, and so we get Chernoff-like concentration, see Schmidt, Siegel and Srinivasan [6].)

It remains to determine for what values of N it is possible to obtain the small list F alluded to above. Here is one algorithm for the problem, Sudan's algorithm. Suppose we could find a bivariate polynomial $G(x, y)$ of small degree such that $G(t, P(t)) = 0$ for any sample point $(t, P(t))|_{t \in T}$. If $f(t)$ is a polynomial of degree d such that P agrees with d on more than r points, then $G(t, f(t)) = 0$ has more than r zeroes. Hence, if we could guarantee that $\deg G(t, f(t)) \leq r$, we can deduce that $G(t, f(t))$ is the zero polynomial. It follows (using some algebra[1]) that $y - f(x)$ divides $G(x, y)$ as a polynomial. Thus we can obtain the list F by factorizing $G(x, y)$ (which is possible to accomplish efficiently using sophisticated randomized algorithms), and the size of the list is bounded by the degree of y in $G(x, y)$.

[1] We can think of $G(x, y)$ as a polynomial in y over the field $R(x)$, where R is the underlying field we are working inside. Since $G(x, f(x)) = 0$, the polynomial has the root $y = f(x)$, and so $y - f(x) | G(x, y)$ as a polynomial in y over $R(x)$. However, $y - f(x)$ is monic and so primitive, hence Gauss's lemma implies that $y - f(x) | G(x, y)$ as a polynomial in y over $R[x]$, in other words over $R[x, y]$.

How do we find such a polynomial $G(x, y)$? We know that all the monomials of $G(x, y)$ must be of the form $x^i y^j$, where $i + dj \leq r$. The number of such monomials is

$$\sum_{j=0}^{r/d} (r - dj) \approx \frac{r^2}{2d}.$$

If this number is larger than N, then we can find such a polynomial by solving linear equations, since there are more variables than equations. This leads to the bound $r^2 \geq 2dN$. The size of the list is at most $r/d \approx \sqrt{2N/d}$. (Guruswami developed an algorithm which can handle $\sqrt{dN}$ instead of $\sqrt{2dN}$ correct values, and this matches a general bound known as the Johnson bound. It is not known whether more errors can be corrected.)

In our case, $r = \alpha N$, and so this method requires $N \geq 2d/\alpha^2 = 2n(n + 1)/\alpha^2$. This means that as long as α is inversely polynomial, our algorithm is polynomial-time.

References

[1] Jin-Yi Cai, A. Pavan, and D. Sivakumar. On the hardness of permanent. In *Proceedings of the 16th annual conference on Theoretical aspects of computer science*, STACS'99, pages 90–99, Berlin, Heidelberg, 1999. Springer-Verlag.

[2] Uriel Feige and Carsten Lund. On the hardness of computing the permanent of random matrices. *Comp. Comp.*, 6:643–654, 1992.

[3] Peter Gemmell and Madhu Sudan. Highly resilient correctors for polynomials. *Inf. Process. Lett.*, 43(4):169–174, Sep 1992.

[4] Richard Lipton. New directions in testing. In *Distributed computing and cryptography*, DIMACS, 1991.

[5] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, SFCS '90, pages 2–10 vol.1, Washington, DC, USA, 1990. IEEE Computer Society.

[6] Jeanette P. Schmidt, Alan Siegel, and Aravind Srinivasan. Chernoff-hoeffding bounds for applications with limited independence. *SIAM J. Discret. Math.*, 8(2):223–250, May 1995.