Zero-One Permanent is $\#P$-Complete, A Simpler Proof

Amir Ben-Dor*  Shai Halevi†

Dept. of Computer Science
Technion
Haifa, Israel 32000

February 22, 1995

*This research was supported by United States-Israel Binational Science Foundation grant 88-00282
†This research was supported by the Miriam and Aharon Gutwirth memorial fellowship
Abstract

In 1979, Valiant proved that computing the permanent of a 01-matrix is \#P-Complete. In this paper we present another proof for the same result. Our proof uses “black box” methodology, which facilitates its presentation. We also prove that deciding whether the permanent is divisible by a small prime is \#P-Hard. We conclude by proving that a polynomially bounded function can not be \#P-Complete under “reasonable” complexity assumptions.
1 Introduction

The permanent has been the object of study by mathematicians since first appearing in the work of Cauchy and Binet in 1812. Despite its syntactical similarity to the determinant, no efficient procedure for computing the permanent is known. In 1979, Valiant provided a reason for this difficulty. In a landmark paper ([Val79a]) he showed that the permanent function is complete for the class #P of enumeration problems. Moreover, Valiant proved that even for 01-matrices, the problem remains #P-Complete.

Valiant’s proof has two parts. In the first part, a many-one reduction from counting the number of satisfying assignments for a CNF formula to computing the permanent of an integer matrix is presented. In the second part, Valiant proved (using the Chinese Remainder Theorem) that computing the permanent of an integer matrix can be done efficiently given access to an oracle that computes the permanent of 01-matrices. This reduction uses polynomially many queries to the 01-permanent oracle.

The proof we present here is similar in some ways to the original proof. In particular, it also consist of two parts as the original proof. There are, however, some differences between our proof and the original one. In the first part, we use a “black box” approach, which simplifies the presentation of the reduction as well as the verification of its validity. First, we build a gadget with some desirable properties. Then we construct a graph using many copies of that gadget, one per every clause in the original 3-CNF formula. We use the properties of the gadget (as a “black box”) to prove the correctness of our construction. The construction of the gadget itself (and the proof of its properties) is described separately. In retrospect one may observe that the first part of Valiant’s reduction could also be presented using “black box” approach.

In the second part, we prove that computing the permanent of an integer matrix can be done efficiently using a single query to an oracle that computes the permanent of 01-matrices. This is done in three steps.

1. First, we reduce the integer permanent problem to the problem of computing the permanent of a non-negative integer matrix.

2. Next, we reduce the non-negative permanent problem to the problem of computing the permanent of an integer matrix, with entries that are either zero or powers of two.

3. Finally, we reduce the last permanent problem to the 01-permanent problem.

The reductions we present in the second part are all many-one. A different and somewhat more complicated many-one reduction was presented in [Zan91].

In the rest of the paper we consider the problem of deciding whether Perm(A) ≡ 0 (mod p) for a given matrix A and integer p. We show that this problem is #P-Hard, even when the integer p is presented in unary. This is done by presenting an algorithm that computes Perm(A) modulo p using an oracle that decides whether Perm(A) ≡ 0 (mod p) (and using the Chinese Remainder Theorem). It was shown in [VV85] that computing Perm(A) modulo k for any fixed k that is not a power of two is NP-Hard (with respect to randomized polynomial reductions). From our algorithm it follows that for any prime $p \neq 2$, deciding whether Perm(A) ≡ 0 (mod p) is also NP-Hard with respect to the same reductions.
Finally, we prove that polynomially bounded functions cannot be \#P-Complete (under some “reasonable” complexity assumptions).

## 2 Preliminaries

We define the notions \#P and \#P-Hardness as usual [Val79b].

**Definition 1:** Let \( f \) be a function \( f : \Sigma^* \rightarrow \mathcal{N} \). We say that \( f \in \#P \) if there exists a binary relation \( T(\cdot, \cdot) \) such that
- If \( (x, y) \in T \) then the length of \( y \) is polynomial in the length of \( x \).
- It can be verified in polynomial time that a pair \((x, y)\) is in \( T \).
- for every \( x \in \Sigma^* \), \( f(x) = |\{y : (x, y) \in T\}| \)

**Definition 2:**
- Given two functions \( f, g : \Sigma^* \rightarrow \mathcal{N} \), we say there is a polynomial Turing-reduction from \( g \) to \( f \) (and denote \( g \preceq f \)) if the function \( g \) can be computed in polynomial time using an oracle to \( f \). We say that there is a many-one reduction from \( g \) to \( f \) if one call to the \( f \)-oracle suffice.
- A function \( f : \Sigma^* \rightarrow \mathcal{N} \) is \#P-Hard if for every \( g \in \#P \) there is a polynomial reduction \( g \preceq f \).
- A function \( f \) is \#P-Complete if it is both \#P-Hard and in \#P.

It was shown in [Val79b] that the problem of counting the number of satisfying assignments for a 3-CNF formula is \#P-Complete, with respect to many-one reductions. Let us denote this problem by \#3-SAT.

**Definition 3:** Given a \( n \times n \) matrix \( A \), the **permanent** of \( A \) is defined as

\[
Perm(A) \overset{\text{def}}{=} \sum_{\sigma} \prod_{i=1}^{n} a_{i,\sigma(i)}
\]

where the summation is over the \( n! \) permutations of \( \{1, 2, \ldots, n\} \). We denote the problem of computing the permanent of a 01-matrix by \( 01 Perm \).

Throughout the paper we refer interchangeably to matrices and their corresponding weighted directed graphs.

**Definition 4:** We say that a \( n \)-node weighted directed graph \( G \) and a \( n \times n \) matrix \( A \) correspond to one another if for every \( i, j \in \{1, \ldots, n\} \), \( A_{i,j} \) is the weight of the edge \( i \rightarrow j \).

**Definition 5:** A **cycle-cover** of a weighted directed graph \( G = (V, E) \) is a subset \( R \subseteq E \) that forms a collection of node-disjoint directed cycles that cover all the nodes of \( G \). The **weight** of \( R \), denoted by \( W(R) \), is the product of the weights of the edges in \( R \).

From the above definitions it follows that if \( G \) is a weighted directed graph that correspond to a matrix \( A \), then the permanent of \( A \) equals to the sum of weights of all the cycle-covers of \( G \). We use \( Perm(G) \) to denote this sum.
3 Main Result

We present another proof for the following result (that was proved in [Val79a] combined with [Zan91]).

Theorem 1: 01-Perm is \#P-Complete (with respect to many-one reductions).

Proof sketch: Let $\phi$ be a 3-CNF formula with $n$ variables and $m$ clauses. We denote by $S(\phi)$ the number of satisfying assignments of $\phi$. The reduction goes as follows:

- First we construct a weighted directed graph $G_\phi$ with weights from the set $\{-1, 0, 1, 2, 3\}$ and size linear in $m$ and $n$ such that
  $$Perm(G_\phi) = 12^m \cdot S(\phi)$$

  The graph $G_\phi$ can be constructed from $\phi$ in polynomial time. The construction is described in Section 4. This construction proves that computing the permanent of an integer matrix is \#P-Hard (with respect to many-one reductions).

- In order to prove that 01-Perm is \#P-Complete (with respect to many-one reductions), we describe in Section 5 a chain of three transformations from an integer matrix to a 01-matrix that maintain the permanent of the matrix.

Therefore, there is a many-one reduction from \#3-SAT to 01-Perm.

4 Constructing $G_\phi$ for a formula $\phi$

Given a 3-CNF formula $\phi$ with $m$ clauses and $n$ variables, we construct a weighted directed graph $G_\phi$ such that there is a mapping between assignments for $\phi$ and cycle-covers in $G_\phi$. This mapping satisfies the following conditions:

- The sum of weights of all the cycle-covers that correspond to each satisfying assignment of $\phi$ equals $12^m$.

- The sum of weights of all the other cycle-covers equals 0.

Clearly, this graph satisfies $Perm(G_\phi) = 12^m \cdot S(\phi)$.

In the construction of $G_\phi$ we use a special clause component as a "black box". The graph $G_\phi$ consist of $m$ clause components (one for each clause in $\phi$), and $n$ additional nodes (one for each variable in $\phi$). Assuming the clause component has some desirable properties, we prove that $Perm(G_\phi) = 12^m \cdot S(\phi)$. In Appendix A we describe the structure of the clause component (which is independent of $\phi$), and prove its properties. We remark that Valiant’s construction yields a different constant ($4^5$) rather than 12.
4.1 Construction of $G_\phi$ Using Clause Components

The graph $G_\phi$ is constructed as follows:

- For each variable $x_i$ in $\phi$ there is a node in $G_\phi$. We refer to these nodes as variable nodes.

- For each clause $c_j = (\alpha_1 \lor \alpha_2 \lor \alpha_3)$ in $\phi$, there is a clause component in $G_\phi$, denoted by $H_j$. The clause component has three input edges labeled $I_1, I_2, I_3$, and three output edges labeled $O_1, O_2, O_3$. These edges connect the component to other components or to variable nodes. Intuitively, the edges $I_k$ and $O_k$ ($1 \leq k \leq 3$) of $H_j$ correspond to the literal $\alpha_k$ in $c_j$.

- For each variable $x_i$ in $\phi$ we form two “cycles” in the graph $G_\phi$.
  
  - Let $c_{j_1}, \ldots, c_{j_k}$ be the clauses that contain the literal $x_i$ in the order they appear in $\phi$. The $T$-cycle of $x_i$ starts at the variable node $x_i$, visits the clause components $H_{j_1}, \ldots, H_{j_k}$ and goes back to $x_i$. If $x_i$ is the $k$’th literal in a clause $c_j$, then the $T$-cycle of $x_i$ enters $H_j$ through the input edge $I_k$ and exits it through the output edge $O_k$.

  - Let $c_{j_1}, \ldots, c_{j_k}$ be the clauses that contain the literal $\neg x_i$ in the order they appear in $\phi$. The $F$-cycle of $x_i$ starts at the variable node $x_i$, visits the clause components $H_{j_1}, \ldots, H_{j_k}$ in a similar way and goes back to $x_i$.

Formally, there is an edge from the output edge $O_{k_1}$ of $H_{j_1}$ to the input edge $I_{k_2}$ of $H_{j_2}$ if the next occurrence of the $k_1$’th literal of the clause $c_{j_1}$ is as the $k_2$’th literal of the clause $c_{j_2}$.

If the literal $x_i$ does not appear in $\phi$, then the $T$-cycle of $x_i$ is a self loop, and the same goes for $\neg x_i$. The weights of all the edges in the $T$-cycles and $F$-cycles of every literal are one.

We call the edges inside the clause components internal edges and the edges between the clause components (or between clause components and variable nodes) external edges. Notice that every external edge belong to either the $T$-cycle or the $F$-cycle of some variable. An example of a graph $G_\phi$ for some formula is presented in Figure 1.

4.2 Correspondence between assignments and cycle-covers

The construction of $G_\phi$ yields a natural correspondence between cycle-covers in $G_\phi$ and assignments for $\phi$.

**Definition 6:** We say that a cycle-cover $R$ of $G_\phi$ induces an assignment $v$ if for each variable $x$ in $\phi$ holds:

- If $v(x) = TRUE$ then $R$ contains all the external edges in the $T$-cycle of $x$ and none of the external edges in the $F$-cycle of $x$. 

Figure 1: The graph $G_\phi$ for the formula $\phi = (x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor x_2 \lor x_3)$. T-cycles are solid and F-cycles are dashed.

- If $v(x) = \text{FALSE}$ then $R$ contains all the external edges in the \textit{F-cycle} of $x$ and none of the external edges in the \textit{T-cycle} of $x$.

Observation 1:

- There are cycle-covers of $G_\phi$ that do not induce any assignment.
- Every cycle-cover of $G_\phi$ can induce at most one assignment, as two different assignments must have at least one variable on which they disagree.
- All the cycle-covers that induce the same assignment agree on their external edges.

Intuitively, the structure of $G_\phi$ ensures that the clause component $H_j$ contributes a multiplicative factor of 12 to the weight of the cycle-covers that induce an assignment which satisfies the clause $c_j$, and a multiplicative factor of zero to the weight of the cycle-covers that do not induce such assignment. To formalize this intuition, we need the following definition.

Definition 7: We say that a cycle-cover is proper with respect to a clause component $H$ if

1. At least one of $H$'s input edges is in $R$.
2. For every $1 \leq k \leq 3$, the edge $I_k$ is in $R$ if and only if the edge $O_k$ is in $R$.

The clause component is constructed such that it contributes a multiplicative factor of 12 to the weight of cycle-covers that are proper with respect to it, and a multiplicative factor of zero to the weight of the other cycle-covers.
Lemma 1: A cycle-cover $R$ induces a satisfying assignment if and only if $R$ is proper with respect to every clause component in $G_\phi$.

Proof: We first show that a cycle-cover that induces a satisfying assignment is proper with respect to every clause component in $G_\phi$. Let $R$ be a cycle-cover that induces a satisfying assignment $v$. Let $c$ be a clause in $\phi$ and $H$ be the corresponding clause component. Consider the literal $\alpha_k$ in $c$, assume w.l.o.g. that $\alpha_k = x_i$.

If $v(x_i) = TRUE$, then $R$ contains all the external edges in the $T$-cycle of $x_i$. By definition of the $T$-cycle of $x_i$, it contains both the external edges $I_k$ and $O_k$ of $H$. Hence these edges are in $R$.

Otherwise, $v(x_i) = FALSE$, so $R$ does not contain any of the external edges in the $T$-cycle of $x_i$. Therefore, neither $I_k$ nor $O_k$ of $H$ are in $R$.

It follows that for each literal $\alpha_k$ in every clause, either both $I_k$ and $O_k$ are in $R$ (If $\alpha_k$ is satisfied by $v$), or neither $I_k$ nor $O_k$ are in $R$ (If $\alpha_k$ is not satisfied by $v$). Hence $R$ satisfies the second condition in the definition of properness, with respect to every clause component. As $v$ is a satisfying assignment for $\phi$, every clause $c$ has at least one literal that is satisfied by $v$. Thus the input edge of $H$ that corresponds to that literal must be in $R$. Thus $R$ satisfies the first condition as well.

We now show that a cycle-cover that is proper with respect to every clause component must induce a satisfying assignment. Let $R$ be a cycle-cover that is proper with respect to every clause component. We define an assignment $v_R$ so that a variable $x$ in $\phi$ is assigned true if the external edge in $R$ that goes out of the variable node $x_i$ belongs to its $T$-cycle and false otherwise. Let $x$ be some variable in $\phi$.

If $v_R(x) = TRUE$ then the first edge in the $T$-cycle of $x$ is in $R$. Notice that it implies that the first edge in the $F$-cycle of $x$ is not in $R$ (since $R$ is a cycle-cover). Let $c$ be a clause that contains either $x$ or $\neg x$ and let $H$ be the corresponding clause component. Since $R$ is proper with respect to $H$, either both the input and output edges that correspond to that literal are in $R$, or neither are. It follows by easy induction that all the edges in the $T$-cycle of $x$ and none of the edges of the $F$-cycle of $x$ are in $R$. The case where $v_R(x) = FALSE$ is treated similarly. Therefore, $R$ induces the assignment $v_R$.

Finally we show that the assignment $v_R$ satisfies $\phi$. Again, let $c$ be a clause in $\phi$ and $H$ be the corresponding component clause. As $R$ is proper with respect to $H$, at least one input edge of $H$ is in $R$. Therefore, $H$ must be on some $T$-cycle or $F$-cycle of some variable. Hence, the corresponding literal in $c$ must be satisfied by $v_R$. Thus $R$ induces $v_R$ which is a satisfying assignment. \qed

4.3 The properties of the Clause Component

As the condition of Lemma 1 depends only on the external edges, we can partition the cycle-covers of $G_\phi$ according to their use of external edges. Formally we have

Definition 8: Let $F$ be a subset of external edges. We say that a set of internal edges $C$ is a $F$-completion if $F \cup C$ is a cycle-cover of $G_\phi$. We denote by $C^F$ the set of all $F$-completions, and by $R^F$ we denote the set of resulting cycle-covers of $G_\phi$. Note that $C^F$ may be empty.
Since the weight of every external edges in $G_\phi$ is one, the weight of every cycle-cover $R \in \mathcal{R}^F$ equals to the product of the weights of its internal edges. Therefore for every set $F$ of external edges we have

$$\sum_{R \in \mathcal{R}^F} W(R) = \sum_{C \in \mathcal{C}^F} W(C)$$

We can partition the edges in every $F$-completion according to their membership in the different clause components. For every $F$-completion $C$, we denote by $C_j$ the set of internal edges in $C$ from the clause component $H_j$. Also, we denote the set $\{C_j | C \in \mathcal{C}^F\}$ by $\mathcal{C}^F_j$.

**Observation 2:** For every subset $F$ of external edges, the internal edges in different clause components can be chosen independently to form a $F$-completion. This is because internal edges in different clause components never share a common node. Therefore, we can compute the sum $\sum_{C \in \mathcal{C}^F} W(C)$ by computing $\sum_{C_j \in \mathcal{C}^F_j} W(C_j)$ for each clause component $H_j$, and then multiplying these sums. That is, for every $F$

$$\sum_{C \in \mathcal{C}^F} W(C) = \prod_{1 \leq j \leq m} \sum_{C_j \in \mathcal{C}^F_j} W(C_j)$$

The properties of the clause component can be expressed as constraints on $F$-completions. Let $F$ be a set of external edges in $G_\phi$ and $H_j$ be some clause component in $G_\phi$:

- If $F$ is proper with respect to $H_j$ then $\sum_{C \in \mathcal{C}^F_j} W(C) = 12$.
- Otherwise $\sum_{C \in \mathcal{C}^F_j} W(C) = 0$.

In Appendix A we describe a 7-node component with weights from the set $\{-1, 0, 1, 2, 3\}$ that satisfy these conditions.

**Lemma 2:** For each 3-CNF formula $\phi$ with $m$ clauses, the graph $G_\phi$ satisfies

$$Perm(G_\phi) = 12^m \cdot S(\phi)$$

**Proof:** The proof follows from these two claims:

1. For each satisfying assignment $v$, the sum of weights of all the cycle-covers that induce $v$ equals $12^m$.
2. The sum of weights of all the other cycle-covers equals zero.

**Proof of claim 1:** Let $v$ be a satisfying assignment for $\phi$. Recall that all the cycle-covers that induce $v$ agree on their external edges. Let us denote this set of external edges by $F_v$. Notice that the set of the cycle-covers which induce $v$ is exactly $\mathcal{R}^{F_v}$. From Observation 2 we get

$$\sum_{R \in \mathcal{R}^{F_v}} W(R) = \sum_{C \in \mathcal{C}^{F_v}} W(C) = \prod_{1 \leq j \leq m} \sum_{C_j \in \mathcal{C}^{F_v}_j} W(C_j)$$

Since $v$ is a satisfying assignment, $F_v$ is proper with respect to every clause component. From the properties of the clause component we have that for every $j$, $\sum_{C_j \in \mathcal{C}^{F_v}_j} W(C_j) = 12$, and the proof of the first claim follows.
Proof of claim 2: Consider now the cycle-covers that do not induce any satisfying assignment. We partition these into equivalence classes according to their use of the external edges of $G_p$. Let $F$ be a subset of external edges that is used by such equivalence class. Note that the equivalence class is exactly $RF$.

Since the cycle-covers in $RF$ do not induce any satisfying assignment, it follows that $F$ is not proper with respect to at least one clause component $H_j$. From the properties of the clause component we have for this $j$, $\sum_{C \in C_j} W(C) = 0$. Thus, we have

$$\sum_{R \in RF} W(R) = \sum_{C \in C} W(C) = \prod_{1 \leq j \leq m} \sum_{C_j \in C_j} W(C_j) = 0$$

and the proof of the second claim follows. \qed

5 01-Perm is \#P-Complete

In the previous section we have proved that computing the permanent of an integer matrix is \#P-Hard with respect to many-one reductions. Let us define the following three problems:

- \textit{IntPerm} - Given an integer matrix $A$, compute $\text{Perm}(A)$.
- \textit{NoNegPerm} - Given a nonnegative integer matrix $A$, compute $\text{Perm}(A)$.
- \textit{2PowersPerm} - The same as NoNegPerm, where $A$ entries can only be zeros or powers of 2.

To show that computing the permanent of a 01-matrix is \#P-Complete, we show the following chain of polynomial time many-one reductions

$$\text{IntPerm} \propto \text{NoNegPerm} \propto \text{2PowersPerm} \propto \text{01Perm}$$

5.1 A Reduction from IntPerm to NoNegPerm

Let $A$ be an $n \times n$ integer matrix in which no entry is larger than $\mu$ in magnitude. From the definition of the permanent it follows that $|\text{Perm}(A)| \leq n! \cdot \mu^n$. To compute $\text{Perm}(A)$ it is sufficient to compute its value modulo $Q$ for $Q > 2n! \cdot \mu^n$. Formally, given $A$ we do the following:

- compute $Q = 2n! \cdot \mu^n + 1$.
- compute $A' = A \mod Q$.
- compute $P = \text{Perm}(A') \mod Q$.
- if $P < Q/2$ then $\text{Perm}(A) = P$. Otherwise $\text{Perm}(A) = P - Q$.

Notice that the transformation from $A$ into $A'$ is polynomial in $n$ and $\log \mu$, as the number of bits that is needed to write $Q$ is polynomial in $n$ and $\log \mu$. 

9
5.2 A Reduction from NoNegPerm to 2PowersPerm

Let $G$ be a $n$-node weighted directed graph with non-negative weights, where the largest weight in $G$ is $W$. We describe a transformation from $G$ into weighted directed graph $G'$ such that the weights in $G'$ are powers of 2, and $Perm(G) = Perm(G')$. The size of $G'$ is polynomial in $n$ and $\log W$. The transformation is performed locally on each edge $e$ in $G$. The edge $e$ is replaced by a subgraph $L_e$. Each replacement maintains the permanent of the graph.

Let $e = (u, v)$ be an edge in $G$ with weight $w$, we can represent $w$ as a sum of increasing powers of 2:

$$w = 2^{x_1} + 2^{x_2} + \cdots + 2^{x_r}, \quad 0 \leq x_1 < x_2 \cdots < x_r \leq \log w$$

The subgraph $L_e$ is composed of $r$ nodes, and $3r$ edges (As in Figure 2).

There is a natural correspondence between cycle-covers of $G$ and cycle-covers of $G'$:

Consider some cycle-cover $R$ in $G$:

- If $e$ is not in $R$ then the only way to cover the new nodes in $L_e$ is to use all the self-loops. As the weight of all the self-loops is 1, the weight of the corresponding cycle-cover $R'$ equals the weight of $R$.

- On the other hand, if $e = (u, v)$ is in $R$ then in all the corresponding cycle-covers in $G'$ there must be a path from $u$ to $v$. There are $r$ such cycle-covers, each corresponds to a different path from $u$ to $v$. As the sum of the weights of these paths equals the weight of $e$, the sum of the weights of the corresponding cycle-covers equals the weight of $R$. 

Figure 2: Transforming an edge with weight $w$
5.3 A Reduction from 2PowersPerm to 01-Perm

Let $G$ be a $n$-node weighted directed graph where all the weights in $G$ are powers of 2, with maximal weight $2^p$. We describe a transformation from $G$ into a digraph $G'$ (with 0-1 weights). Such that $Perm(G) = Perm(G')$. The size of $G'$ is polynomial in $n$ and $p$. This transformation is also performed locally on each edge $e$ in $G$. Every edge with weight $> 1$ is replaced by a subgraph $J_e$. Each edge in $J_e$ has weight one, therefore the resulting graph $G'$ is an unweighted directed graph. Each replacement maintains the permanent of the graph.

Let $e = (u, v)$ be an edge in $G$ with weight $w = 2^r > 1$. The subgraph $J_e$ is composed of $2r$ nodes and $6r$ edges (As in Figure 3).

There is a natural correspondence between cycle-covers of $G$ and cycle-covers of $G'$. Consider some cycle-cover $R$ in $G$

- If $e$ is not in $R$ then the only way to cover the new nodes in $J_e$ is to use all the self-loops. As the weight of all the edges is 1, the weight of the corresponding cycle-cover $R'$ equals the weight of $R$.

- On the other hand, if $e = (u, v)$, with weight $w = 2^r$, is in $R$ then in the corresponding cycle-covers of $G'$ there must be a path from $u$ to $v$. There are $2^r$ such possible cycle-covers, each corresponds to a different path from $u$ to $v$. As each path has weight 1, the sum of weights of all these cycle-covers equals the weight or $R$. 

\[\text{Figure 3: Transforming an edge with weight } 2^r\]
6 Related Topics

6.1 Computing the Permanent Modulo \( p \)

We know that computing the permanent of a 01-matrix is hard, yet deciding whether this permanent equals zero is easy. We consider the following variant of the above decision problem: Given a 01-matrix \( A \) and an integer \( p \), is \( \text{Perm}(A) \equiv 0 \pmod{p} \)? We show that this problem is \#P-Hard. First we show that this problem is hard for integer matrices, and then we use the reduction from Section 5 to show that it is also hard for 01-matrices.

Clearly, given an oracle that computes \( \text{Perm}(A) \pmod{p} \) (for every \( A \) and \( p \)), one can compute \( \text{Perm}(A) \) by choosing a large enough \( p \). Notice also that using the Chinese Remainder Theorem, it is sufficient to compute \( \text{Perm}(A) \pmod{p} \) for primes that are smaller than \( n^2 \log \mu \), where \( n \) is the size of \( A \) and \( \mu \) is the magnitude of the largest entry in \( A \). The only thing left to show is how to compute \( \text{Perm}(A) \pmod{p} \) using an oracle that decides whether \( \text{Perm}(A) \equiv 0 \pmod{p} \).

Definition 9: The language \( \text{ModPerm} \) is defined as follows

\[
\text{ModPerm} \overset{\text{def}}{=} \{ < A, p > | \text{Perm}(A) \equiv 0 \pmod{p} \}
\]

where \( A \) is an integer matrix, and \( p \) is an integer. The language \( 01\text{-ModPerm} \) is defined similarly where \( A \) is a 01-matrix.

The difficulty in using an oracle to \( \text{ModPerm} \) is that it seems hard to change the permanent of a given matrix by additive factor. Notice that it is fairly simple to compute \( \text{Perm}(A) \pmod{p} \) using an oracle that decides whether \( \text{Perm}(A) \equiv 1 \pmod{p} \): One can simply multiply the first row of \( A \) by 2 over and over again, until the permanent equals 1 \( \pmod{p} \). This trick does not work with oracle to \( \text{ModPerm} \). In the following lemma we show an algorithm that computes \( \text{Perm}(A) \pmod{p} \) using an oracle to \( \text{ModPerm} \).

Lemma 3: Let \( A \) be a \( n \times n \) integer matrix, and let \( p \) be a prime. Computing \( \text{Perm}(A) \pmod{p} \) can be done in time polynomial in \( n \) and \( p \) using at most \( p n + \frac{1}{2} n^2 \) calls to a \( \text{ModPerm} \) oracle.

Proof: We give a constructive proof, by describing a recursive algorithm that computes \( \text{Perm}(A) \pmod{p} \) using a \( \text{ModPerm} \) oracle.

For a \( 1 \times 1 \) matrix, \( \text{Perm}(A) \pmod{p} \) can be computed directly. For \( n > 1 \), we consider two cases:

1. \( \text{Perm}(A) \equiv 0 \pmod{p} \). In this case, one call to the \( \text{ModPerm} \) oracle is sufficient.

2. \( \text{Perm}(A) \not\equiv 0 \pmod{p} \). In this case, there is at least one minor of \( A \), denoted by \( A^i_j \) such that \( \text{Perm}(A^i_j) \not\equiv 0 \pmod{p} \). Using at most \( n \) calls to the oracle, find that minor. Assume w.l.o.g. that this minor is \( A^i_1 \). The algorithm continues as follows:

- Compute \( \text{Perm}(A^i_1) \pmod{p} \) recursively.
• Define a sequence of $p-1$ matrices $\{B_i\}_{i=1}^{p-1}$. For every $1 \leq i \leq p-1$, the matrix $B_i$ is identical to $A$, except that $b_{11} = a_{11} + i$. Clearly, for every $B_i$ holds

$$\text{Perm}(B_i) = \text{Perm}(A) + i \cdot \text{Perm}(A_1^1)$$

As $\text{Perm}(A_1^1) \not\equiv 0 \pmod{p}$, and $p$ is a prime, there is a unique index $i$ such that $\text{Perm}(B_i) \equiv 0 \pmod{p}$. Note that

$$\text{Perm}(A) \equiv -i \cdot \text{Perm}(A_1^1) \pmod{p}$$

Using at most $p-1$ additional calls to the oracle, find that index, (denoted by $i$) and return $(-i \cdot \text{Perm}(A_1^1)) \mod p$.

The number of calls to the oracle is given by the recurrence $T(n) \leq n + T(n-1) + p$ which yields $T(n) \leq pn + \frac{1}{2}n^2$.

**Corollary 2:** The language ModPerm is $\#P$-Hard.

Using the last two reductions from Section 5 we get,

**Corollary 3:** The language $01$-ModPerm is $\#P$-Hard.

**Proof:** We show a reduction ModPerm $\leq_1 01$-ModPerm. Given a matrix $A$ and an integer $p$ -

• Compute the matrix $A' = A \mod p$.

• Use the reductions from Section 5 on the matrix $A'$, and get a $01$-matrix $A''$ such that $\text{Perm}(A'') = \text{Perm}(A') \equiv \text{Perm}(A) \pmod{p}$.

• Use an oracle to the language $01$-ModPerm, on the input $< A'', p >$ to decide whether $\text{Perm}(A) \equiv 0 \pmod{p}$.

It was shown in [VVS85] that computing $\text{Perm}(A)$ modulo $k$ for any fixed $k$ that is not a power of two is NP-Hard (with respect to randomized polynomial reductions). Applying our algorithm, we get

**Corollary 4:** for every prime $p > 2$, the language

$$\text{ModPerm}_p \overset{\text{def}}{=} \{ A \mid \text{Perm}(A) \equiv 0 \pmod{p} \}$$

is NP-Hard with respect to randomized polynomial reductions.
6.2 Polynomially bounded functions can not be #P-Complete

All known #P-Complete functions have exponentially large range. On the other hand, it is easy to construct #P functions with small range, but these functions appear to be easy to compute. We conjecture that #P functions can not have small range unless they can be computed efficiently. We give support to this conjecture, by showing that under “reasonable” complexity assumptions, polynomially bounded functions can not be #P-complete. We remark that there exist even binary functions that are #P-Hard, (e.g., ModPerm). It follows that, under the same complexity assumptions, such functions can not be in #P.

Recall that, by definition, $P^{NP} \subseteq P^{#P}$. Moreover, Toda showed ([Tod89]) that the entire polynomial time hierarchy is contained in $P^{#P}$. Therefore, if $P^{NP} = P^{#P}$ then the polynomial time hierarchy collapses into $\Sigma^P_2$.

**Lemma 4:** Assuming $P^{NP} \not\subseteq P^{#P}$, a polynomially bounded function can not be #P-Complete.

**Proof:** Assume, towards a contradiction, that there is a #P-Complete function $f: \{0,1\}^* \rightarrow \mathcal{N}$ such that for every $x \in \{0,1\}^*$, $f(x) \leq Q(|x|)$ where $Q(\cdot)$ is some polynomial. We define the language $L_f$ as follows:

$$L_f \equiv \{ <x, k> \mid f(x) \geq k \}$$

Clearly, one can compute $f$ efficiently given an oracle to $L_f$. As $f \in \#P$, there is a binary relation $T$ such that $f(x) = | \{ y : (x, y) \in T \} |$. Therefore $L_f$ can be represented as follows:

$$L_f = \left\{ <x, k> \mid \begin{array}{l}
\text{there exist } k \text{ distinct } y's \text{ such that } (x, y) \in T \\
The \end{array} \right\}$$

Since $k$ can be bounded by $Q(|x|)$, it follows that $L_f \in NP$. This is because a nondeterministic polynomial TM can guess $k$ distinct $y$’s and verify that for each of them holds $(x, y) \in T$. As $f$ can be computed efficiently using a $L_f$-oracle and $f$ is #P-Hard, it follows that $P^{#P} = P^f \subseteq P^{L_f} \subseteq P^{NP}$.

This contradicts the assumption $P^{NP} \not\subseteq P^{#P}$. \qed

**Acknowledgment:** Thanks to Amos Beimel, Benny Chor, Oded Goldreich, Lee-Bath Nelson, and Erez Petrank for their useful comments.

**References**


A Constructing the Clause Component

The clause component is a 7-node weighted directed graph with weights from the set \(-1, 0, 1, 2, 3\). The input edges \(I_1, I_2, I_3\) enter nodes 1, 2 and 3 respectively. The output edges \(O_1, O_2, O_3\) exit from nodes 5, 4, and 3 respectively.

Denoting the corresponding 7 \( \times \) 7 matrix by \(A\), the properties of this component can be expressed as a set of constraints on the permanent of \(A\) itself and some of its sub-matrices. Denote by \(A^I_0\) the matrix \(A\) without the columns \(I\) and the rows \(O\), we can write the constraints on the clause component as follows:

- Whenever all the input and output edges of the component are used, the sum over all completions in it must equal 12. This can be represented by the equation
  \[
  \text{Perm}(A^{1,2,3}_{5,4,3}) = 12
  \]

- Whenever two couples of matching input and output edges of the component are used, the sum over all completions in it must equal 12. This can be represented by the equations
  \[
  \text{Perm}(A^{2,3}_{4,3}) = 12, \quad \text{Perm}(A^{1,3}_{5,3}) = 12, \\
  \text{Perm}(A^{1,2}_{5,4}) = 12
  \]

- Whenever a single couple of matching input and output edges of the component is used, the sum over all completions in it must equal 12. This can be represented by the equations
  \[
  \text{Perm}(A^1_5) = 12, \quad \text{Perm}(A^2_4) = 12, \\
  \text{Perm}(A^3_3) = 12
  \]

- Whenever two input edges and two non-matching output edges of the component are used, the sum over all completions in it must equal 0. This can be represented by the equations
  \[
  \text{Perm}(A^{2,3}_{5,3}) = 0, \quad \text{Perm}(A^{2,3}_{5,4}) = 0, \\
  \text{Perm}(A^{1,3}_{4,3}) = 0, \quad \text{Perm}(A^{1,3}_{5,3}) = 0, \\
  \text{Perm}(A^{1,2}_{4,3}) = 0, \quad \text{Perm}(A^{1,2}_{5,3}) = 0
  \]

- Whenever an input edge and a non-matching output edge of the component are used, the sum over all completions in it must equal 0. This can be represented by the equations
  \[
  \text{Perm}(A^1_{4,3}) = 0, \quad \text{Perm}(A^1_{5,2}) = 0, \\
  \text{Perm}(A^2_{3,3}) = 0, \quad \text{Perm}(A^2_{5,2}) = 0, \\
  \text{Perm}(A^3_{3,3}) = 0, \quad \text{Perm}(A^3_{5,2}) = 0
  \]
Whenever no input edge and no output edge of the component are used, the sum over all completions in it must equal 0. This can be represented by the equation

\[ Perm(A) = 0 \]

It can be verified that the following $7 \times 7$ matrix satisfies all these conditions. As the matrix $A$ satisfies the above constraints, the clause component indeed has the properties we use in the proof of lemma 2.

\[
A = \begin{pmatrix}
1 & 0 & 0 & 0 & 2 & 0 & 0 \\
0 & 1 & 0 & 3 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & -1 & 1 & 1 \\
0 & 0 & -1 & 2 & -1 & 1 & 1 \\
0 & 0 & -1 & -1 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 1 & 2 & -1 \\
0 & 0 & 1 & 1 & 1 & 0 & 1 \\
\end{pmatrix}
\]