The Permanent and #P

Adam Granicz
Caltech, Computer Science Dept.

31st May 2001

Abstract

This paper is an attempt to help fellow students in understanding the complexity of computing the permanent, as introduced in [Val-79]. We will introduce topics that are closely related and essential to our discussion. To read this paper, you should only need a basic but working background in computer science.

1 Preliminaries

The arithmetic definition for the permanent of an $n \times n$ matrix A is defined similarly as its determinant, except that we do not account for the signature terms, e.g. there are no negative terms in the summation. Furthermore, the summation is over all $n!$ permutations ϱ:

$$perm(A) = \sum_{\varrho \in S(n)} \prod_{i=1}^{n} A_{\varrho(i),i}$$

Despite this obvious similarity, it turns out that the computational complexities of computing the permanent and the determinant are quite different. Apparently, no one has been able to provide a polynomial-time solution for the permanent problem, but such is known to exist for quite some time for the latter.

One of the pioneer aspects of [Val-79] is that it proved this "apparent" intractability of computing the permanent, and showed that it is *complete* as far as polynomial-time bounded counting problems go.

The rest of this section contains several relevant details and is meant to refresh your memory rather than introducing these ideas. Recall the following definition of a deterministic Turing machine: $M = (S, \Sigma, \rho, s_0)$, where S is a set of states, Σ is the alphabet, s_0 is the start state, and ρ is defined as:

$$\rho\colon S \times \Sigma \longrightarrow S \cup \{Yes, No\} \times \Sigma \times \{\leftarrow, \rightarrow, \perp\}$$

In other words, ρ is mapping the current state and symbol on the tape to a new state, possibly the "Yes" or "No" state, a new symbol to be written to the current position on the tape, and the direction of moving the read-write head of the TM (left, right, no move, respectively).

Non-deterministic Turing machines (NDTM), on the other hand, allow mapping the current state and tape symbol to potentially many new states, thus resulting in a tree-shaped computation that exponentially exceeds the computational power of deterministic Turing machines, which operate on a single line of computation. The increased computational power of a NDTM is traded for exponential increase in computation-time when simulating such machines with deterministic ones. Loosely speaking, those decision problems that can be solved with polynomial-time deterministic (non-deterministic) Turing machines constitute the complexity class P (NP). Clearly, $P \subseteq NP$, but whether $P \overset{?}{=} NP\}$ is still one of the most fundamental open questions of computer science.

Cook's theorem states that every NP problem can be reduced to the boolean formula satisfiability (SAT) problem, which itself is in NP, thus SAT is NP-complete. A boolean formula can contain boolean connectives (implication, iff, AND, OR, NOT) and variables, but can converted into conjunctive normal form (CNF), consisting of a set of clauses connected by conjunctions (AND), in each of which there are literals (variables and their negated selves) connected by disjunctions (OR). A formula is said to be in $kCNF$ if every clause contains exactly k distinct literals.

We give the following definitions as they will be essential in our proofs.

Definition The *cycle cover* of a graph $G = (V, E)$ is a collection of vertex-disjoint cycles that covers all vertices in V.

The following is taken from [Koz-92].

Definition (Chinese Remainder Theorem) Let $m_1, m_2, ..., m_k$ be pairwise relatively prime positive integers, and let $m = \prod_{i=1}^{k} m_i$. Let Z_n denote the ring of integers modulo n. The ring Z_m and the direct product of rings $Z_{m_1} \times Z_{m_2} \times \cdots \times Z_{m_k}$ are isomorphic under the function $f : Z_m \to Z_{m_1} \times Z_{m_2} \times \cdots \times Z_{m_k}$ given by $f(x) = (x \bmod m_1, x \bmod m_2, \ldots, x \bmod m_k)$.

To compute $f^{-1}(x_1, \ldots, x_k)$, we first compute, for each $1 \leq i \leq k$, integers s and t such that $sm_i + t \prod_{1 \leq j \leq k, j \neq i} m_j = 1$, and take $u_i = t \cdot \prod_{1 \leq j \leq k, j \neq i} m_j$. The numbers s and t are available as a byproduct of the Euclidean algorithm. Finally, take $f^{-1}(x_1, \ldots, x_k) = x_1 u_1 + \cdots + x_k u_k \bmod m$.

2 The complexity class #P

The typical question to ask when dealing with P and NP problems is "Is there a solution to problem L?" A more powerful, but natural question to ask is "How many solutions are there to problem L?" Clearly, if the number of solutions is nonzero, we can answer the first question as "Yes, there is *a* solution to L", otherwise we output "No." The essence of $\#P$ (pronounced as "number-P" or "sharp-P") is found in the second question, namely that we are interested in the *number* of solutions to a problem.

The complexity classes P and NP are concerned with *decision problems.* They are represented with a binary NP-relation Q that maps instances of problems to their solutions. Such witness solutions can be certified in polynomial time, by definition. On the other hand, $\#P$ is the class of *counting functions* that count the number of solutions with respect to NP-relations.

Definition Given a NP-relation $Q \in \Sigma^* \times \Sigma^*$, the *counting function* $f_Q : \Sigma^* \to N$ is defined as $f_Q(x) = |\ \{y : (x, y) \in Q\ \}\ |$, in other words the number of witnesses to x with respect to Q.

Counting functions can be computed by *counting Turing machines*: standard non-deterministic Turing machines, with the additional capability of outputting the number of accepting computations induced by their input. The *time complexity* $f(n)$ of a counting Turing machine is the number of steps that the longest accepting computation takes on the set of all inputs of size n. It is worth mentioning, that with respect to deterministic Turing machines, counting Turing machines are even more unrealistic models of computation than the regular non-deterministic Turing machines (NDTM).

As we said, $\#P$ is the class of counting functions. For instance, if f is a function which maps a boolean formula into its satisfying assignments, it is a member of $\#P$. Counting versions of NP problems are often denoted with the same name preceded by $\#$. The Permanent is an instance in $\#P$. Its counting function f is associated with the NP-relation Q which relates

(0, 1)-matrices (corresponding to directed graphs) to their cycle covers. In this sense, $f_Q(X)$ is the number of cycle covers in a graph with adjacency matrix X. If X corresponds to a bipartite graph, $f_Q(X)$ yields to the number of perfect matchings. For other $\#P$ problems, consult [Wel-93].

3 Proofs

When proving that some problem P is *complete* with respect to a given complexity class Ψ, we typically show that a known complete problem P^* in Ψ simplifies to P. This *reduction* step is rather crucial, choosing the wrong complete problem P^* could mean that our proof becomes overly complicated and thus hard to follow. When dealing with NP-complete problems, we often use SAT as P^* in the above scenario, as boolean formula satisfiability can be reduced to a number of problem domains such as graphs and sets, thus providing a well-balanced formulation to many NP-completeness proofs. It is not a surprise that careful selection of *parsimonious* reductions between NP-complete problems can also lead us to proving $\#P$-completeness of many counting problems based on $\#SAT$. In this section, we will briefly examine $\#SAT$ and prove that it is $\#P$-complete, and then reduce it to *Permanent*.

Definition The permanent is #P-complete.

Proof We will proceed as follows:

1. SAT is NP-complete
2. $\#SAT$ is $\#P$-complete
3. $\#SAT \leq^p_m$ *Permanent* .

From the preliminaries, we know that (1) is true by Cook's theorem. Thus, given a NDTM M (along with its NP-relation Q) and an input x, we can construct a boolean formula $F(x)$, such that an input y to the formula will result in *true* $\iff$ M accepts (x, y), e.g. $(x, y) \in Q$.

Furthermore, we can observe that the reduction used in Cook's theorem is *parsimonious*, exactly because of the above bijection. And since this way we are expressing the counting problem associated with any NDTM M in terms of the counting version of the SAT problem, we conclude that $\#SAT$ is $\#P$-complete.

Our main result will be shown by proving that $\#SAT \leq^p_m$ *Permanent*. Recall, that $perm(A)$ of a matrix A is the number of *cycle covers* in a directed non-weighted graph $G = (V, E)$ with adjacency matrix A. From the arithmetic definition of the permanent, it is easy to see that for a directed *weighted* graph $G = (V, E, w)$, where the adjacency matrix A contains $w_{i,j}$, the $perm(A)$ gives the sum of the weights of each cycle cover. In such a graph, each cycle-cover has *weight* equal to the product of the weights of the edges along all the vertex-disjoint cycles in the cover. We will exhibit a reduction from boolean formulas F (assumed to be in $3CNF$, see theorem below) to weighted directed graphs in which the weight of distinct cycle covers can be related to the number of satisfying variable assignments to F. Finally, we will show a polynomial reduction from weighted directed graphs to non-weighted directed graphs (each edge assume to be of weight 1), such that the permanent of their adjacency matrices is preserved, thus proving that computing the permanent of even (0,1)-matrices is $\#P$-complete.

Theorem 3.1 *A formula F in CNF can be transformed into a formula in $3CNF$ in polynomial time.*

Proof Let F be a a formula in CNF consisting of clauses C_i. We will rewrite F into another formula F^* in $3CNF$ by replacing each clause C_i in F with C_i^* as follows:

If $C_i = (a)$, then $C_i^*= (a \lor \beta \lor \gamma) \land (a \lor \overline{\beta} \lor \gamma) \land (a \lor \beta \lor \overline{\gamma}) \land (a \lor \overline{\beta} \lor \overline{\gamma})$, where β and γ stand for new variables not used in any other clause.

If $C_i = (a \lor b)$, then $C_i^*=(a \lor b \lor \gamma) \land (a \lor b \lor \overline{\gamma})$, where γ stands for a new variable not used in any other clause.

If $C_i = (a \lor b \lor c)$, then $C_i^*= C_i$.

If $C_i = (a_1 \lor a_2 \lor a_3 \lor ... \lor a_k)$, then $C_i^*= (a_1 \lor a_2 \lor b_1) \land (\overline{b_1} \lor a_3 \lor b_2) \land (\overline{b_2} \lor a_4 \lor b_3) \land ... \land (\overline{b_{k-3}} \lor a_{k-1} \lor a_k)$, where $b_1...b_{k-3}$ are new variables not used in any other clause.

Note that each transformation step takes polynomial time. In the above transformations, we have replaced each clause C_i with a set of new clauses C_i^* by introducing new variables where needed. But, we can observe that no matter what these extra variables are assigned, C_i^* is satisfiable if and only if C_i is. Thus, a formula F in CNF is satisfiable $\iff$ its corresponding F^* in $3CNF$ is satisfiable. $\diamond$

A reduction from $3SAT$ to *Permanent* must map the features of a $3CNF$ formula F into the new problem domain; cycle covers in a directed graphs. The most basic feature of F is the assignment of each variable to either *True* or *False*. Since variables can be present in more than one clause in F, it must be guaranteed that each copy of a variable corresponds to the same value throughout the entire formula. This *consistency* is the second fundamental feature of boolean formulas. The third is that all clauses have to be satisfied at the same time in order for the entire formula to be satisfied, hence forcing a *constraint* on F.

Now that we have determined the basic building blocks in a $3CNF$ (or CNF for that matter) formula, it is time to map them into graphs. We will use a similar technique as described in [Pap-94], but will mention others as well at the conclusion of this paper.

The assignment of a variable can easily be encoded as the following *choice gadget*:

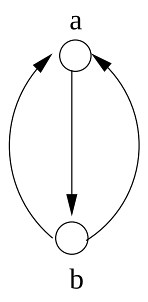

Figure 3-1. The choice gadget

Each variable in our $3SAT$ formula F is represented with the above choice gadget using distinct a and b nodes for every variable. Any cycle cover that traverses each pair (corresponding to different variables) a and b must do so by choosing one of the external edges but not both. We will see later that these external edges actually represent connections to different other gadgets instead of being parallel edges. For now, it suffices to say that choice gadgets guarantee that any variable will take on one value only throughout the entire formula.

Another gadget we need is the exclusive-or (XOR) gadget. The XOR gadget is the most important part of our proof, and not accidentally the most complex one as well. The idea is the following: we need a gadget which has two entry-exit points, say a and d. In any cycle cover if a is entered, d must be the leaving point, and symmetrically if d is entered a must be the leaving point. With this behavior, if we have our XOR gadget and two vertex-disjoint sub-graphs G_1 and G_2 with at least two distinct vertices in each, say $\{u_1, v_1\}$ and $\{u_2, v_2\}$, respectively, then we can add edges (u_1, a), (d, v_1), (u_2, d), (a, v_2) and obtain the following behavior: in any cycle cover of the entire graph, if the edges (u_1, a), (d, v_1) are covered, then the edges (u_2, d), (a, v_2) will not be traversed, and vice versa. In other words, if we regard the entire XOR gadget and $\{u_1, v_1, u_2, v_2\}$ as (u_1, v_1) connected with (u_2, v_2) via the XOR gadget, only one of these two edges will be traversed in any cycle cover. Now, let's look at how the XOR gadget can be implemented.

Figure 3-2(a) shows the actual XOR gadget with the rest of the graph (including the vertices $\{u_1, v_1, u_2, v_2\}$ as understood in the above paragraph) represented by the node G. It is clear that if the edge (G, d) is traversed in any cycle cover, so should (a, G) be, but neither (G, a) nor (d, G) must be traversed, and vice versa. Assume that we enter the gadget through the edge (G, d). Since the gadget itself has connections to G only through a and d, there are two possibilities to leave the gadget, either through the edge (a, G) or (d, G). But we would like to rule out leaving through (d, G) because that way vertex a has to be picked up by another cycle. And since the only such cycle would be $(G, a) - (a, G)$, we would not satisfy our primary objective: not to traverse both $(G, d), (a, G)$ and $(G, a), (d, G)$ in the same cycle cover. Therefore we must leave through (a, G). This is depicted in Figure 3-2(b).

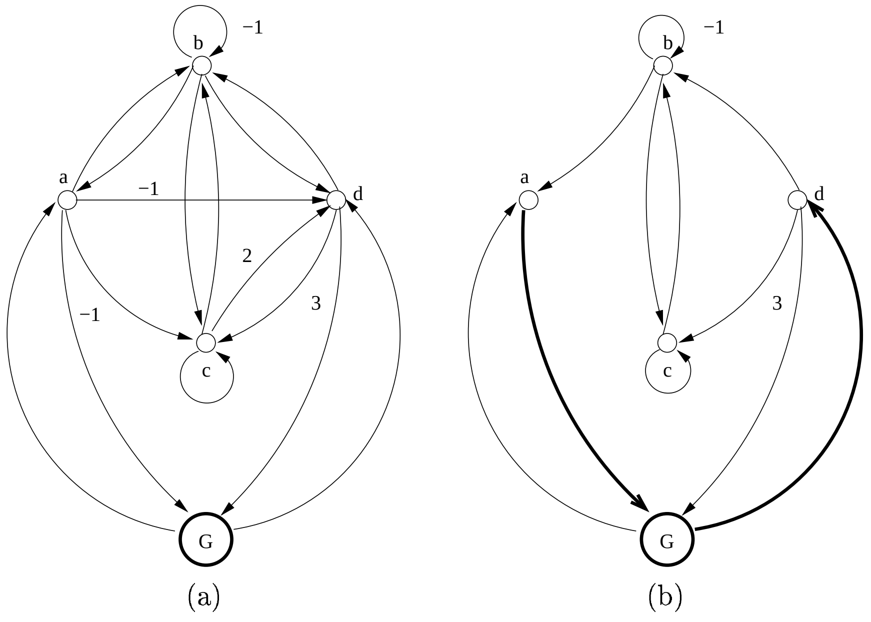

Figure 3-2(a). The XOR gadget, **(b).** $(G, d), (a, G)$ traversed

Following a similar line of argument, if a cycle cover traverses (G, a) it must leave through (d, G). All other cycle covers should have weight 0. On the other hand, legal cycle covers (those covering edges $(G, d), (a, G)$ or $(G, a), (d, G)$) should have a non-zero weight, so we can relate them to satisfying assignments of boolean formulas, as we will explain later.

As we just showed, assume that (G, d) and (a, G) are traversed. Since vertex a and d had been seen already, we can just as well delete the column that corresponds to d (because no edge can return to d) and the row that corresponds to a (because no edge can originate from a, as we already have edge (a, G)) from the matrix that corresponds to our XOR gadget. With this new matrix, we must have a fixed, non-zero contribution towards the weight of cycle covers. This should similarly be equal to the contribution from the case when (G, a) and (d, G) are traversed, e.g. from the permanent of the matrix resulting when deleting the column that corresponds to a and the row that corresponds to d. With a similar argument, bad cycle covers where either $(G, a), (a, G)$ or $(G, d), (d, G)$ are traversed correspond to the permanent of the matrix resulting when deleting the row and the column corresponding to a, and the row and the column corresponding to d, respectively.

Our needs can be summarized with the following notation. Assume $X[a,b]$ means deleting row a and column b from matrix X, X is the adjacency matrix of the directed weighted graph corresponding to our XOR gadget, and vertices $\{a,b,c,d\}$ in the above example are numbered 1..4, respectively. Then $perm(X[1,4]) = perm(X[4,1]) = constant$, but $perm(X) = perm(X[1,1]) = perm(X[4,4]) = 0$.

$$X = \begin{vmatrix} 0 & 1 & -1 & -1 \\ 1 & -1 & 1 & 1 \\ 0 & 1 & 1 & 2 \\ 0 & 1 & 3 & 0 \end{vmatrix}$$

One XOR gadget that satisfies all of our requirements that we have set up in the previous paragraphs is depicted in Figure 3-2(a) and its adjacency matrix is found above. From our previous discussion, assuming that (G,d) and (a,G) are traversed, Figure 3-3 shows the two possible cycle covers of total weight 4, the first one with weight 3 and the second one with weight 1. Similarly, it can be verified that if (G,a) and (d,G) are traversed the total weight of all cycle covers is also 4.

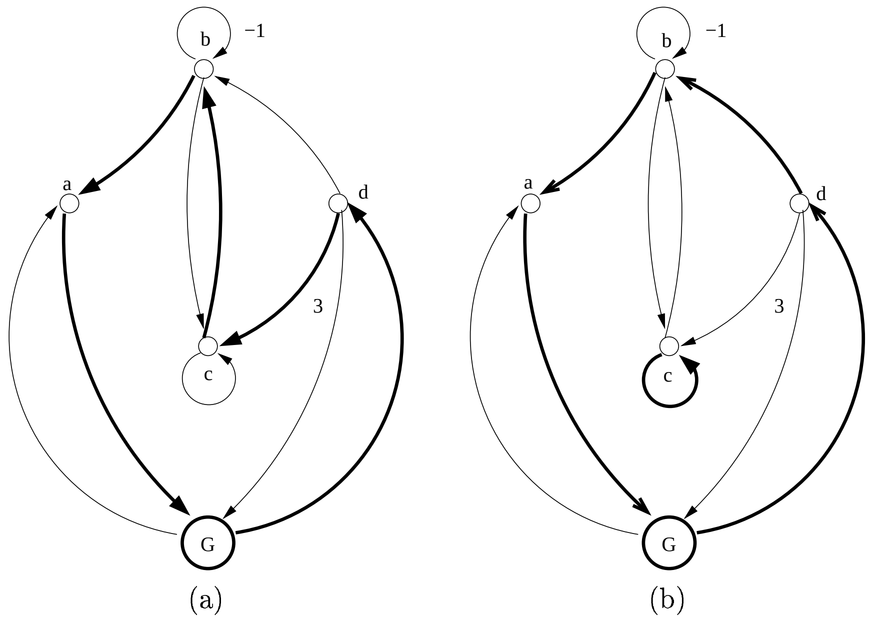

Figure 3-3. 2 possible cycle covers

Figure 3-4(a) shows the new graph that results when deleting the row corresponding to a and the column corresponding to d from our XOR gadget's adjacency matrix. As we can see, the resulting graph G^{**} is quite different than if we had simply removed all out-edges from a and all in-edges to d from the XOR gadget. But the $perm(G^{**}) = 4$, just as expected. Figures 3-4(b) and 3-4(c) show the two cycle covers of weight 3 and 1, respectively.

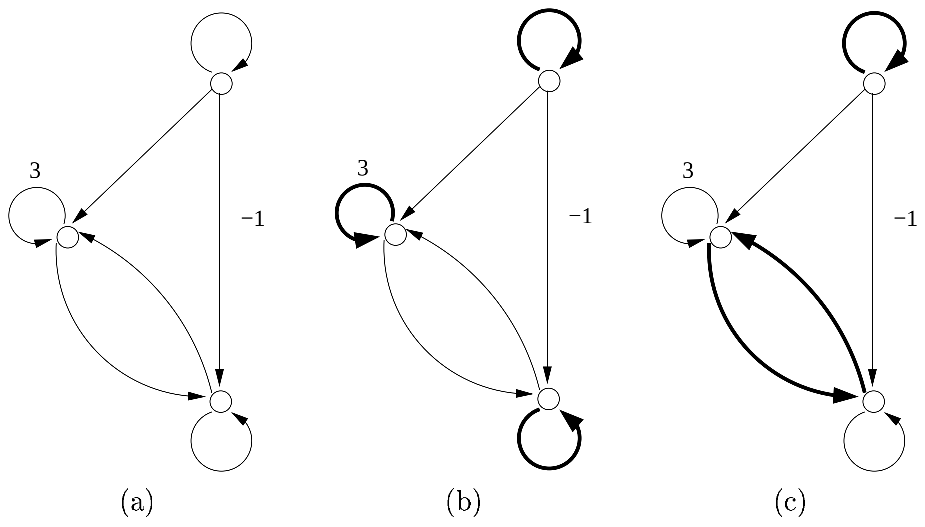

Figure 3-4. XOR[1,4]

Now we have means to guarantee that certain edges are not traversed in the presence of some edges in a given cycle cover, and we have means to guarantee that one variable assumes one value only. What we need now is means to connect these pieces with respect to every clause in the boolean formula F. If we had a *clause gadget* with three designated edges that are hooked up via the XOR gadget to either the $True$ or the $False$ edges in the choice gadgets corresponding to the literals found in a clause, we could conclude that if any of such connected choice gadget edges are not traversed, its corresponding literal in the clause is $False$. All we need now, is to guarantee that our *clause gadget* can never be covered in a cycle cover such that all the three designated edges are traversed.

Figure 3-4 shows such a *clause gadget*. It can be verified that there exists no cycle cover that traverses all three thick edges (which correspond to loops to XOR gadgets, similarly as the designated edges in the choice gadget). Furthermore, for any combination of the three edges there is only one corresponding cycle cover, since there is a cycle cover in the gadget if and only if the literals corresponding to traversed edges yield to a satisfying assignment of the clause, e.g. there is at least one uncovered thick edge.

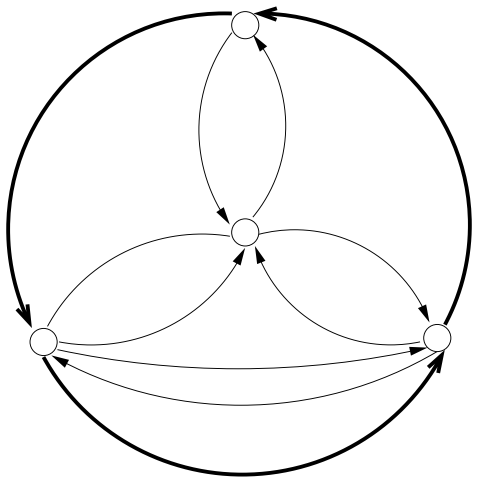

Figure 3-5. The clause gadget

With these gadgets it is straightforward to give our reduction algorithm from a $3CNF$ formula F to directed weighted graphs in which the cycle covers are related to satisfying variable assignments in F. Our construction is as follows:

1. Assume the new formula F has m variables and n clauses.
2. Create a choice gadget for each m variables in F.
3. Create a clause gadget for each clause C_i in F, where $1 \leq i \leq n$.
4. Do for all n clauses C_i: create 3 XOR gadgets and connect a distinct XOR gadget (for each of the following edge-pairs) with each of the three designated edges in C_i and their appropriate edge in the corresponding choice gadget, e.g. the $False$ edge if $literal_{i,j}$ (the jth literal in the ith clause) is negated, the $True$ edge otherwise.

In step (4), if more than one XOR gadget is connected to any choice gadget, the XOR gadgets have to be linked together. Assume that we have t XOR gadgets linking to the choice gadget corresponding to variable x_i (which is represented as a choice gadget with vertices $\{a, b\}$, say). We can link these XOR gadgets together by adding edges $(b, XOR_{1,in})$, $(XOR_{1,out}, XOR_{2,in})$ $\ldots (XOR_{t,out}, a)$-recall that each XOR gadget has 2 sets of in-out points, in our case one set of these is already taken up by the connection to the appropriate clause gadget, thus $XOR_{i,in}$ and $XOR_{i,out}$ refers to the other set of in-out points (to which edges are reversed in direction with respect to the first set). This way, a given value of a variable is "forced" on all XOR gadgets linking to it.

Notice that *any* variable assignment ensures that the XOR gadgets that are hooked up with the appropriate variable values of the assignment and all choice gadgets are traversed via cycles. But there may still be some XOR gadgets (those corresponding to values of variables that were not picked in the given assignment) and all the clause gadgets that need to be traversed. Because of the way we designed the clause gadgets, if there are three XOR gadgets still not traversed belonging to the same clause, the current variable assignment can not satisfy the formula. Otherwise, all remaining XOR gadgets can be traversed along with all the clause gadgets via a number of cycle covers. Out of these cycle covers, though, there is only one which has nonzero weight (all covers not properly traversing the remaining XOR gadgets, meaning that they cover these XOR gadgets via cycles not containing vertices from outside of the gadgets [e.g. from their corresponding clause gadgets], have zero weight - because $perm(XOR){=}0$). Therefore, our construction is correct, and the number of different cycle covers that can be established yields to the number of satisfying variable assignments. Furthermore, since each successful traversal of the clause and the choice gadgets always results in cycles of weight 1, and successfully traversing each XOR gadget results in cycles of weight 4, we conclude that the sum of the weights of all possible cycle covers is $4^m \cdot s$, where s is the number of cycle covers, and 4^m is the weight from traversing all m XOR gadgets (since each satisfying variable assignment has to do this) within any one of these cycle covers (and m is the total number of literals in the formula.)

The figure 3-6 contains a complete example for the formula $F = (x_1 \vee x_2) \wedge (x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3)$. [First it is converted into $3CNF$: $F^* = (x_1 \vee x_2 \vee b) \wedge (x_1 \vee x_2 \vee \overline{b}) \wedge (x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3)$.] We have depicted each XOR gadget as crossed cycles to simplify the figure. Furthermore, each edge that connects any gadget to another *edge* is linking that gadget to the two vertices of the edges involved via a cycle, as explained in the earlier sections. Multiple XOR gadgets connecting to the same choice gadgets are linked together (as explained in the previous paragraphs).

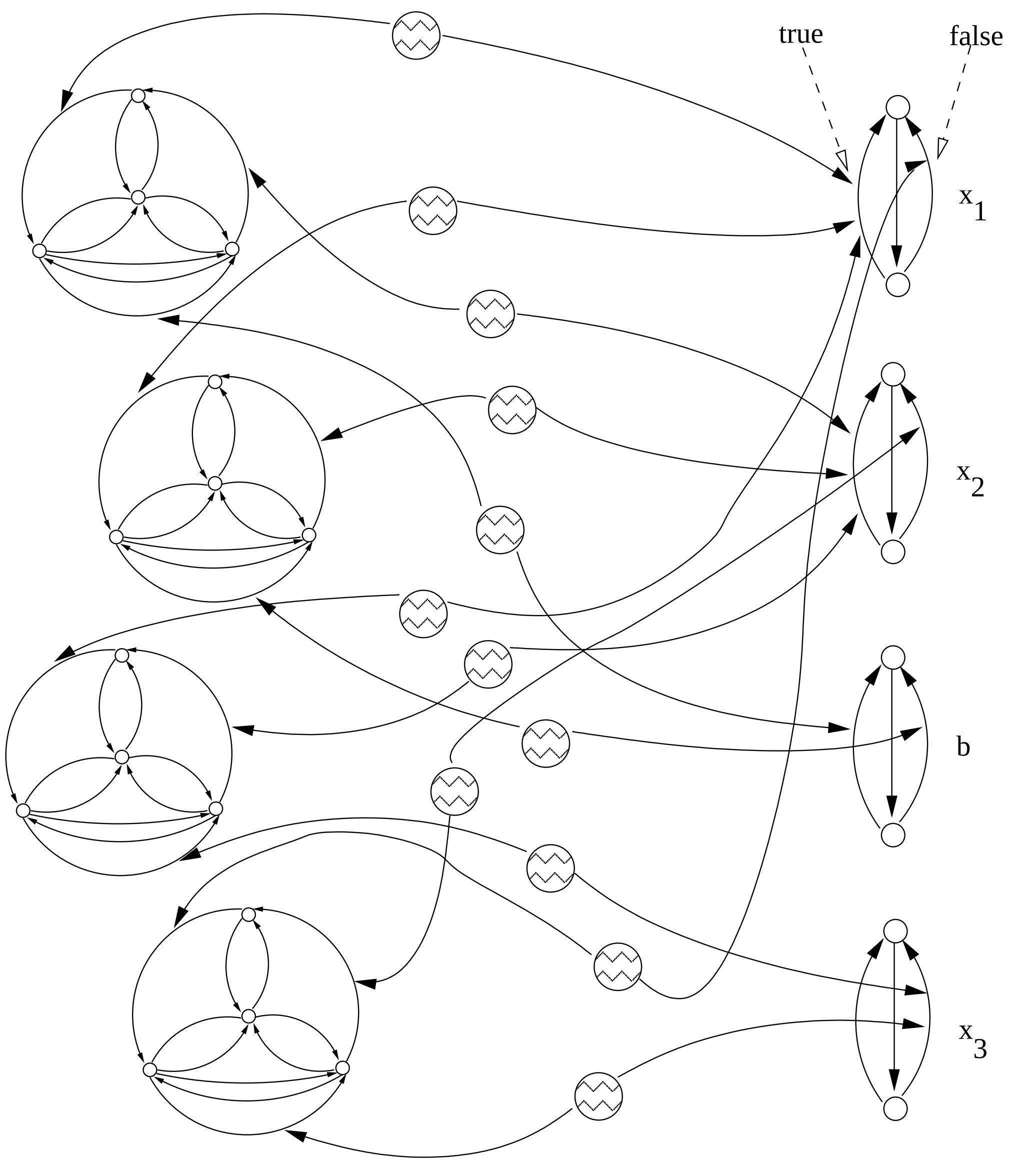

Figure 3.6. $F^* = (x_1 \vee x_2 \vee b) \wedge (x_1 \vee x_2 \vee \overline{b}) \wedge (x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3)$

This concludes the proof that computing the permanent of integer-entry matrices is $\#P$-complete. Now we show a polynomial-time reduction from general integer-entry matrices to $(0,1)$-matrices, such that the permanent is preserved.

Positive matrix entries (edges (a,b) with weight w) can be simulated by adding w new vertices with self loops and connecting each new vertex v_i via edges (a,v_i) and (v_i,b). It is easily seen that traversing such gadget contributes w to the permanent (since there are w cycle covers, each with weight 1), just like an edge with weight w would. Exponentially large weights (2^n) can be simulated by arranging n such gadgets in tandem [see Pap-94, pg. 447].

The only problem we have now is the edges with -1 weight (remember, we introduced these edges in our XOR gadget). To solve this, we can use the Chinese Remainder Theorem (see Preliminaries), and compute the permanent modulo p_i, for some set of distinct primes $\{p_1, \ldots\ p_k\}$. Assuming that the largest entry in our $n \times n$ matrix M is μ, then it must be true that $\prod_{i=1}^{k} p_i \geq 2\mu^n \cdot n!$ to accommodate the largest possible value of the permanent $(\mu^n \cdot n!)$. For each $perm(M)$ mod p_i, we can replace the -1 entry with $p_i - 1$, since they are the same modulo p_i. Finally, $perm(M)$ can be recovered as explained in the Preliminaries. ∎

4 Conclusion

The proof we presented in this paper is not unique. There are several variations based on the $\#P$ complete problem that is chosen to be reduced to the permanent. Several authors reduce the permanent to the #Hamiltonian-cycle, or the #Vertex-Cover in a similar proof.

Computing the permanent of a matrix M is $\#P$-complete, even for $(0,1)$-matrices, therefore a polynomial solution for it is very unlikely, unless $P = NP$. Despite the obvious similarity between the permanent and its cousin the determinant, attempts to reduce the permanent of general matrices to the determinant via matrix transformations had brought little success, typically only "shaving off" some portion of the exponent from the expected run-time $O(g(n) \cdot c^{f(n)})$ of the permanent [BF-97, BF2-97]. There had been some success to approximate the permanent via Monte Carlo algorithms, as described in [KKLLL-93], roughly in the square root of the above running time, but still in the exponential range.

References

[Val-79] L.G. Valiant. The complexity of computing the permanent. Theoretical Computer Science, 8(2):189-201, April 1979.

[Wel-93] D.J.A. Welsh. Complexity: Knots, Colourings and Counting. Cambridge University Press, 1993.

[CLR-90] T.H. Cormen, C.E. Leiserson, R.L. Rivest. Introduction to Algorithms. MIT Press, 1990.

[Koz-92] Dexter C. Kozen. The Design and Analysis of Algorithms. Springer-Verlag New York, Inc., 1992.

[Pap-94] Christos H. Papadimitriou. Computational Complexity. Addison-Wesley Publishing Company, Inc., 1994.

[KKLLL-93] N. Karmarkar, R. Karp, R. Lipton, L. Lovász, M. Luby. A Monte-Carlo Algorithm for Estimating the Permanent. SIAM J. of Computing, Vol. 22. No. 2. pp. 284-293. April 1993.

[BF-97] E. Bax, J. Franklin. A Permanent Algorithm With $\Omega(exp[\frac{\sqrt[4]{n}}{4\,log_2 n}\frac{1}{\sqrt{2\pi e}}])$ Expected Speedup for 0-1 Matrices. Caltech TR. 1997.

[BF2-97] E. Bax, J. Franklin. A Permanent Formula With Many Zero-Valued Terms. Caltech TR. 1997.