Lesson 6: The complexity classes for counting

Theme: The complexity classes for counting problems and the complexity of computing permanent.

1 Complexity classes for counting problems

1.1 The class FP

We denote by **FP** the class of functions $f : \{0,1\}^* \to \mathbb{N}$ computable by polynomial time DTM. Here the convention is that a natural number is always represented in binary form. So, when we say that a DTM $\mathcal{M}$ computes a function $f : \{0,1\}^* \to \mathbb{N}$, on input word w, the output of $\mathcal{M}$ on w is $f(w)$ in the binary representation.

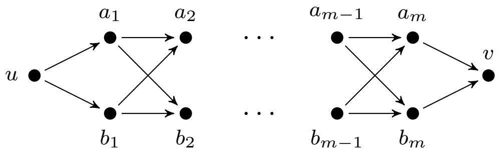
Let ♯CYCLE be the following problem.

♯CYCLE	
Input:	A directed graph G.
Task:	Output the number of cycles in G.

As before, ♯CYCLE can also be viewed as a function. Note also that the number of cycles in a graph with n vertices is at most exponential in n, thus, its binary representation only requires polynomially many bits.

Theorem 6.1 *If* ♯CYCLE *is in* **FP***, then* $\mathbf{P} = \mathbf{NP}$.

Proof. Let G be a (directed) graph with n vertices. We construct a graph G' obtained by replacing every edge (u, v) in G with the following gadget:



Note that every simple cycle in G of length ℓ becomes $(2^m)^\ell$ cycles in G'. Now, let $m \overset{\mathsf{def}}{=} n \log n$.

It is not difficult to show that G has a hamiltonian cycle (i.e., a simple cycle of length n) if and only if G' has more than $n^{(n^2)}$ cycles. So, if ♯CYCLE $\in$ **FP**, then checking hamiltonian cycle can be done is in **P**. ∎

Note that checking whether a graph has a cycle itself can be done in polynomial time. However, as Theorem 6.1 above states, it is unlikely that counting the number of cycles can be done in polynomial time.

1.2 The class ♯P

Definition 6.2 A function $f : \{0,1\}^* \to \mathbb{N}$ is in ♯**P**, if there is a polynomial $q(n)$ and a polynomial time DTM $\mathcal{M}$ such that for every word $w \in \{0,1\}^*$, the following holds.

$$f(w) \quad = \quad |\{y : \mathcal{M} \text{ accepts } (w, y) \text{ and } y \in \{0,1\}^{q(|w|)}\}|$$

Alternatively, we can say that f is in $\sharp\mathbf{P}$, if there is a polynomial time NTM $\mathcal{M}$ such that for every word $w \in \{0,1\}^*$, $f(w)$ = the number of accepting runs of $\mathcal{M}$ on w.

For a function $f : \{0,1\}^* \to \mathbb{N}$, the language associated with the function f, denoted by O_f, is defined as $O_f \overset{\mathsf{def}}{=} \{(w,i) : \text{the } i^{\text{th}} \text{ bit of } f(w) \text{ is } 1\}$. When we say that *a TM $\mathcal{M}$ has oracle access to a function f*, we mean that it has oracle access to the language O_f.

We define $\mathbf{FP}^f$ as the class of functions $g : \{0,1\}^* \to \mathbb{N}$ computable by a polynomial time DTM with oracle access to f.

Definition 6.3 Let $f : \{0,1\}^* \to \mathbb{N}$ be a function.

- f is $\sharp\mathbf{P}$-*hard*, if $\sharp\mathbf{P} \subseteq \mathbf{FP}^f$, i.e., every function in $\sharp\mathbf{P}$ is computable by a polynomial time DTM with oracle access to f.
- f is $\sharp\mathbf{P}$-*complete*, if $f \in \sharp\mathbf{P}$ and f is $\sharp\mathbf{P}$-hard.

Let ♯SAT be the following problem.

♯SAT	
Input:	A boolean formula φ.
Task:	Output the number of satisfying assignments for φ.

As before, the output numbers are to be written in binary form. We can also view ♯SAT as a function ♯SAT $: \{0,1\}^* \to \mathbb{N}$, where ♯SAT$(\varphi)$ = the number of satisfying assignment for φ.

Theorem 6.4 *♯SAT is $\sharp\mathbf{P}$-complete.*

Proof. Cook-Levin reduction (to prove the **NP**-hardness of SAT) is parsimonious. ▮

There are usually two ways to prove a certain function is $\sharp\mathbf{P}$-hard, as stated in Remark 6.5 and 6.6 below.

Remark 6.5 Let f_1 and f_2 be functions from $\{0,1\}^*$ to $\mathbb{N}$. Suppose L_1 and L_2 be languages in **NP** such that f_1 and f_2 are the functions for the number of certificates for L_1 and L_2, respectively. That is, for every word $w \in \{0,1\}^*$,

$$f_i(w) \quad = \quad \text{the number of certificates of } w \text{ in } L_i, \qquad\qquad \text{for } i = 1,2.$$

If f_1 is $\sharp\mathbf{P}$-hard and there is a parsimonious (polynomial time) reduction from L_1 to L_2, then f_2 is $\sharp\mathbf{P}$-hard.

Remark 6.6 Let f and g be two functions from $\{0,1\}^*$ to $\mathbb{N}$. If f is $\sharp\mathbf{P}$-hard and $f \in \mathbf{FP}^g$, then g is $\sharp\mathbf{P}$-hard.

Since there is a parsimonious reduction from SAT to 3-SAT, by Theorem 6.4 and Remark 6.5, we have the following corollary.

Corollary 6.7 *♯3-SAT is $\sharp\mathbf{P}$-complete.*

Corollary 6.7 can also be proved by showing ♯SAT $\in \mathbf{FP}^{\sharp\textsf{3-SAT}}$.

2 The complexity of computing the permanent

2.1 Definition of permanent

For an integer $n \geqslant 1$, let $[n] = \{1, \ldots, n\}$. The *permanent* of an $n \times n$ matrix A over integers is defined as:

$$\text{per}(A) \stackrel{\text{def}}{=} \sum_{\sigma} \prod_{i=1}^{n} A_{i,\sigma(i)}$$

where σ ranges over all permutation on $[n]$. Here $A_{i,j}$ denotes the entry in row i and column j in matrix A.

Consider the following problem.

PERM	
Input:	A square matrix A over integers.
Task:	Output the permanent of A.

We denote it by 0|1-PERM, when the entries in the input matrix A are restricted to 0 or 1.

Theorem 6.8 (Valiant 1979) 0|1-PERM *is* $\sharp\mathbf{P}$*-complete.*

To show that 0|1-PERM is in $\sharp\mathbf{P}$, consider the following algorithm.

```
Input: A 0-1 matrix $A$.
Guess a permutation $\sigma$ on $[n]$, i.e., for each $i \in [n]$, guess a value $v_i \in [n]$.
If the guessed $\sigma$ is not a permutation, REJECT.
Compute the value $\prod_{i=1}^{n} A_{i,\sigma(i)}$.
ACCEPT if and only if the value is 1.
```

It is obvious that on input A, the number of accepting runs is the same as per(A).

2.2 Combinatorial view of permanent

Let $G = (V, E, w)$ be a complete directed graph, i.e., $E = V \times V$, and each edge (u, v) has a weight $w(u, v) \in \mathbb{Z}$. We write a (simple) cycle as a sequence $p = (u_1, \ldots, u_\ell)$, and its weight is defined as:

$$w(p) \stackrel{\text{def}}{=} w(u_1, u_2) \cdot w(u_2, u_3) \cdot \ldots \cdot w(u_{\ell-1}, u_\ell) \cdot w(u_\ell, u_1)$$

A loop (u, u) is considered a cycle.

A *cycle cover* of G is a set $R = \{p_1, \ldots, p_k\}$ of pairwise disjoint cycles such that for every vertex $u \in V$, there is a cycle $p_j \in R$ such that u appears in p_j. The weight R is defined as:

$$w(R) \stackrel{\text{def}}{=} \prod_{p_j \in R} w(C_j)$$

Note that a cycle or a cycle cover can also be viewed as a set of edges.

Assuming that the vertices in G are $\{1, \ldots, n\}$, let A be the adjacency matrix of G, i.e., A is an $(n \times n)$ matrix over $\mathbb{Z}$ such that $A_{i,j} = w(i, j)$.

A permutation $\sigma = (d_{1,1}, \ldots, d_{1,k_1}), \ldots, (d_{l,1}, \cdots, d_{l,k_l})$ on $[n]$ can be viewed as a cycle cover whose weight is exactly the value $\prod_{i \in [n]} A_{i,\sigma(i)}$. Thus, we have the equation:

$$\text{per}(A) = \sum_{R \text{ is a cycle cover of } G} w(R)$$

3 Reduction from 3-SAT to cycle cover

In this section we will show how to encode 3-SAT as the cycle cover problem.

3.1 Overview of the main idea

Let Ψ be a formula in 3-CNF. Let $x_1, \ldots, x_n$ be the variables and $C_1, \ldots, C_m$ be the clauses. We will construct a complete directed graph $G = (V, E, w)$, where the weight of each edge can be arbitrary integer and every boolean assignment $\phi : \{x_1, \ldots, x_n\} \to \{0, 1\}$ is associated with a set F_ϕ of cycle covers of G such that the following holds.

- For two different assignments ϕ_1, ϕ_2, the sets F_{ϕ_1} and F_{ϕ_2} are disjoint.
- If ϕ is a satisfying assignment for Ψ, the total weight of cycle covers in F_ϕ is 4^{3m}, i.e.,

$$\sum_{R \in F_\phi} w(R) \quad = \quad 4^{3m}$$

- If ϕ is not a satisfying assignment for Ψ, the total weight of cycle covers in F_ϕ is 0, i.e.,

$$\sum_{R \in F_\phi} w(R) \quad = \quad 0$$

- The total weight of cycle covers not in any F_ϕ is 0, i.e.,

$$\sum_{R \notin F_\phi \text{ for any } \phi} w(R) \quad = \quad 0$$

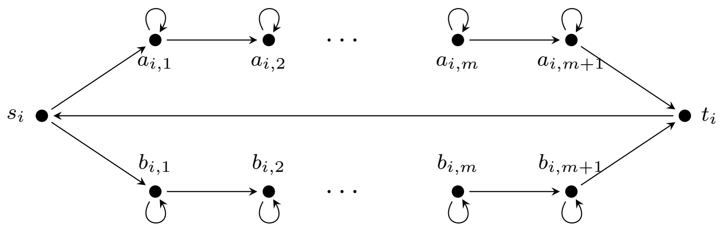
If A is the adjacency matrix of G, it is clear that:

$$\text{per}(A) \quad = \quad 4^{3m} \times (\text{the number of satisfying assignment for } \Psi)$$

3.2 The construction of the graph G

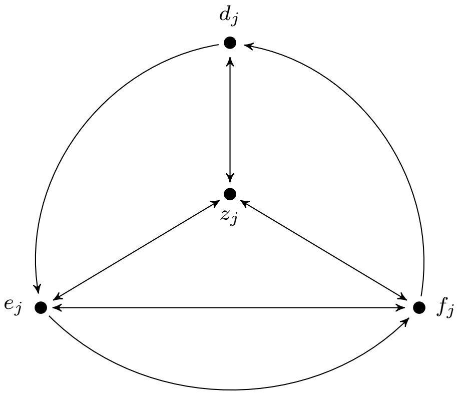
In the following we will draw an edge with a label indicating its weight. If the label is missing, it means the weight is 1. When an edge is not drawn, it means the weight is 0.

Variable gadget. For each variable x_i, we have the following "variable gadget":



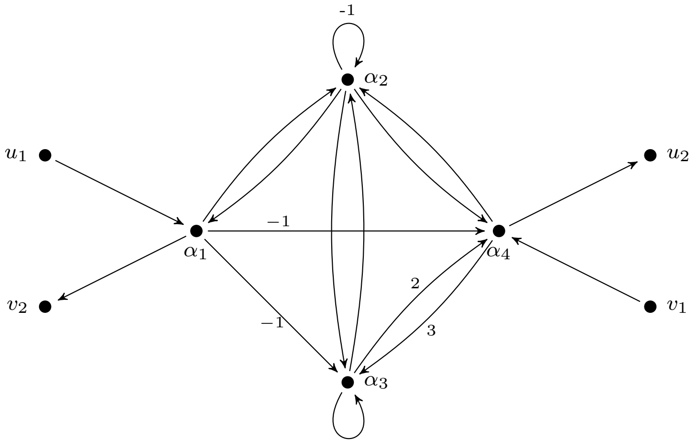
The upper edges, i.e., $(a_{i,1}, a_{i,2}), \ldots, (a_{i,m}, a_{i,m+1})$, are called the *external "true" edges* of x_i, and the lower edges, i.e., $(b_{i,1}, b_{i,2}), \ldots, (b_{i,m}, b_{i,m+1})$, the *external "false" edges* of x_i.

Clause gadget. For each clause C_j, we have the following "clause gadget":



The "outer" edges $(d_j, e_j), (e_j, f_j), (f_j, d_j)$ are intended to represent the literals in C_j. If ℓ_1, ℓ_2, ℓ_3 are the literals in C_j, then their associated edges are $(d_j, e_j), (e_j, f_j), (f_j, d_j)$, respectively. To avoid clutter, we will call those edges ℓ_1-edge, ℓ_2-edge and ℓ_3-edge, respectively.

The XOR operator. We also have the "XOR operator" between two edges (u_1, u_2) and (v_1, v_2):



Definition 6.9 Let H be a graph, and let (u_1, u_2) and (v_1, v_2) are two non-adjacent edges in H.

- For a cycle cover R of H, we say that R *respects the property* $(u_1, u_2) \oplus (v_1, v_2)$, if R contains exactly one of (u_1, u_2) or (v_1, v_2).
- Let H' denotes the graph obtained from H by replacing the edges $(u_1, u_2), (v_1, v_2)$ with the edges in the XOR operator above.

  A cycle cover R' of H' is *an associated cycle cover* of R, if it satisfies the following condiiton.

  - If R contains (u_1, u_2), then R' contains a path from u_1 to u_2.
  - If R contains (v_1, v_2), then R' contains a path from v_1 to v_2.
  - $R \setminus \{(u_1, u_2), (v_1, v_2)\} \subseteq R'$.

Lemma 6.10 *Let H, H', R and $(u_1, u_2), (v_1, v_2)$ be as in Definition 6.9. Then, the following holds.*

$$\sum_{R' \text{ is associated with } R} w(R') \;=\; \begin{cases} 4w(R), & \textit{if } R \textit{ respects } (u_1,u_2) \oplus (v_1,v_2) \\ 0, & \textit{otherwise} \end{cases}$$

Constructing the graph G. The graph G is defined as the disjoint union of all the variable and clause gadgets and the following additional edges to connect them: For every clause C_j, for every literal ℓ in C_j, if $\ell = x_i$, we "connect" the ℓ-edge in the clause gadget of C_j with the edge $(a_{i,j}, a_{i,j+1})$ via the XOR operator; and if $\ell = \neg x_i$, we "connect" it with the edge $(b_{i,j}, b_{i,j+1})$.

For an assignment $\phi : \{x_1, \ldots, x_n\} \to \{0, 1\}$, we say that *a cycle cover R is associated with ϕ*, if the following holds for every variable x_i.

- If $\phi(x_i) = 1$, the cycle $(s_i, a_{i,1}, \ldots, a_{i,m+1}, t_i)$ is in R.
- If $\phi(x_i) = 0$, the cycle $(s_i, b_{i,1}, \ldots, b_{i,m+1}, t_i)$ is in R.

Lemma 6.11 *For every assignment $\phi : \{x_1, \ldots, x_n\} \to \{0, 1\}$, the following holds.*

$$\sum_{R \text{ is associated with } \phi} w(R) \;=\; \begin{cases} 4^{3m}, & \textit{if } \phi \textit{ is satisfying assignment for } \Psi \\ 0, & \textit{if } \phi \textit{ is not} \end{cases}$$

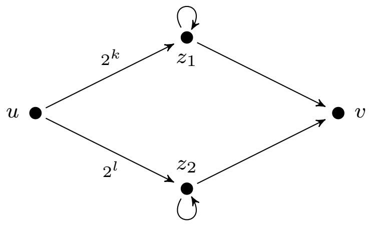
Combining Lemmas 6.10 and 6.11, it is immediate that the following holds.

$$\mathrm{per}(A) \;=\; 4^{3m} \times (\text{the number of satisfying assignments for } \Psi)$$

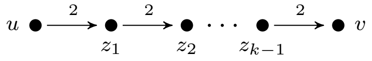
Here A is the adjacency matrix of G.

4 Reduction from matrices over $\mathbb{Z}$ to matrices over $\{0, 1\}$

Reduction to matrices over integers of the form -2^k, 0 or 2^k. For each edge (u, v) with weight $2^k + 2^l$, we can replace it with 2 "parallel" edges with weights 2^k and 2^l, respectively.



Reduction to matrices over integers of the form -1, 0 or 1. For each edge (u, v) with weight 2^k, we can replace it with k "series" edges, each with weights 2.



Each weight 2 edge can be further reduced to weight 1 edge as above.

Reduction to matrices over $\{0, 1\}$ (but on modular arithmetic). The permanent of an $n \times n$ matrix A over $\{-1, 0, 1\}$ can only in between $-n!$ and $n!$. Let $m = n^2$. Since $2^m + 1 > 2n!$, it is sufficient to compute $\mathrm{per}(A)$ in $\mathbb{Z}_{2^m+1}$. Since $-1 \equiv 2^m \pmod{2^m + 1}$, we can replace each -1 with 2^m, which can then be reduced to 1 as above.

5 ♯P-hardness of PERM – Putting all the pieces together

Putting together all the pieces from Sections 3 and 4, we design a polynomial time algorithm to compute ♯3-SAT (with oracle access to language O_{per}, i.e., the language associated with permanent). On input 3-CNF formula Ψ, do the following.

- Let n and m be the number of variables and clauses in Ψ.
- Construct a matrix A over $\{-1,0,1\}$ such that $\mathrm{per}(A)$ is 4^{3m} times the number of satisfying assignments for Ψ.
- Let m be an integer for which we can compute $\mathrm{per}(A)$ modulo 2^m+1.
- Let A' be the matrix obtained by replacing every -1 in A with 2^m.
- Compute $\mathrm{per}(A')$ by querying the oracle on each bit.
- Let Z be the remainder of $\mathrm{per}(A')$ divided by 2^m+1.
- Divide Z by 4^{3m} and output it.