On P vs NP

Density[a]

The **density** of language $L \subseteq \Sigma^*$ is defined as

$$\text{dens}_L(n) = |\{x \in L : |\, x \,| \leq n\}|.$$

- If $L = \{0,1\}^*$, then $\text{dens}_L(n) = 2^{n+1} - 1$.
- So the density function grows at most exponentially.
- For a unary language $L \subseteq \{0\}^*$,

$$\text{dens}_L(n) \leq n + 1.$$

  – Because $L \subseteq \{\epsilon, 0, 00, \ldots, \overbrace{00 \cdots 0}^{n}, \ldots\}$.

[a]Berman and Hartmanis (1977).

Sparsity

- **Sparse languages** are languages with polynomially bounded density functions.
- **Dense languages** are languages with superpolynomial density functions.

Self-Reducibility for SAT

- An algorithm exploits **self-reducibility** if it reduces the problem to the same problem with a smaller size.
- Let ϕ be a boolean expression in n variables $x_1, x_2, \ldots, x_n$.
- $t \in \{0,1\}^j$ is a **partial** truth assignment for $x_1, x_2, \ldots, x_j$.
- $\phi[\,t\,]$ denotes the expression after substituting the truth values of t for $x_1, x_2, \ldots, x_{|\,t\,|}$ in ϕ.

An Algorithm for SAT with Self-Reduction

We call the algorithm below with empty t.

```
1: if |t| = n then
2:     return φ[t];
3: else
4:     return φ[t0] ∨ φ[t1];
5: end if
```

The above algorithm runs in exponential time, by visiting all the partial assignments (or nodes on a depth-n binary tree).

NP-Completeness and Density[a]

Theorem 78 *If a unary language* $U \subseteq \{0\}^*$ *is NP-complete, then* $P = NP$.

- Suppose there is a reduction R from SAT to U.
- We shall use R to guide us in finding the truth assignment that satisfies a given boolean expression ϕ with n variables if it is satisfiable.
- Specifically, we use R to prune the exponential-time exhaustive search on p. 611.
- The trick is to keep the already discovered results $\phi[t]$ in a table H.

[a]Berman (1978).

```
if |t| = n then
    return φ[t];
else
    if (R(φ[t]), v) is in table H then
        return v;
    else
        if φ[t0] = "satisfiable" or φ[t1] = "satisfiable" then
            Insert (R(φ[t]), 1) into H;
            return "satisfiable";
        else
            Insert (R(φ[t]), 0) into H;
            return "unsatisfiable";
        end if
    end if
end if
```

The Proof (continued)

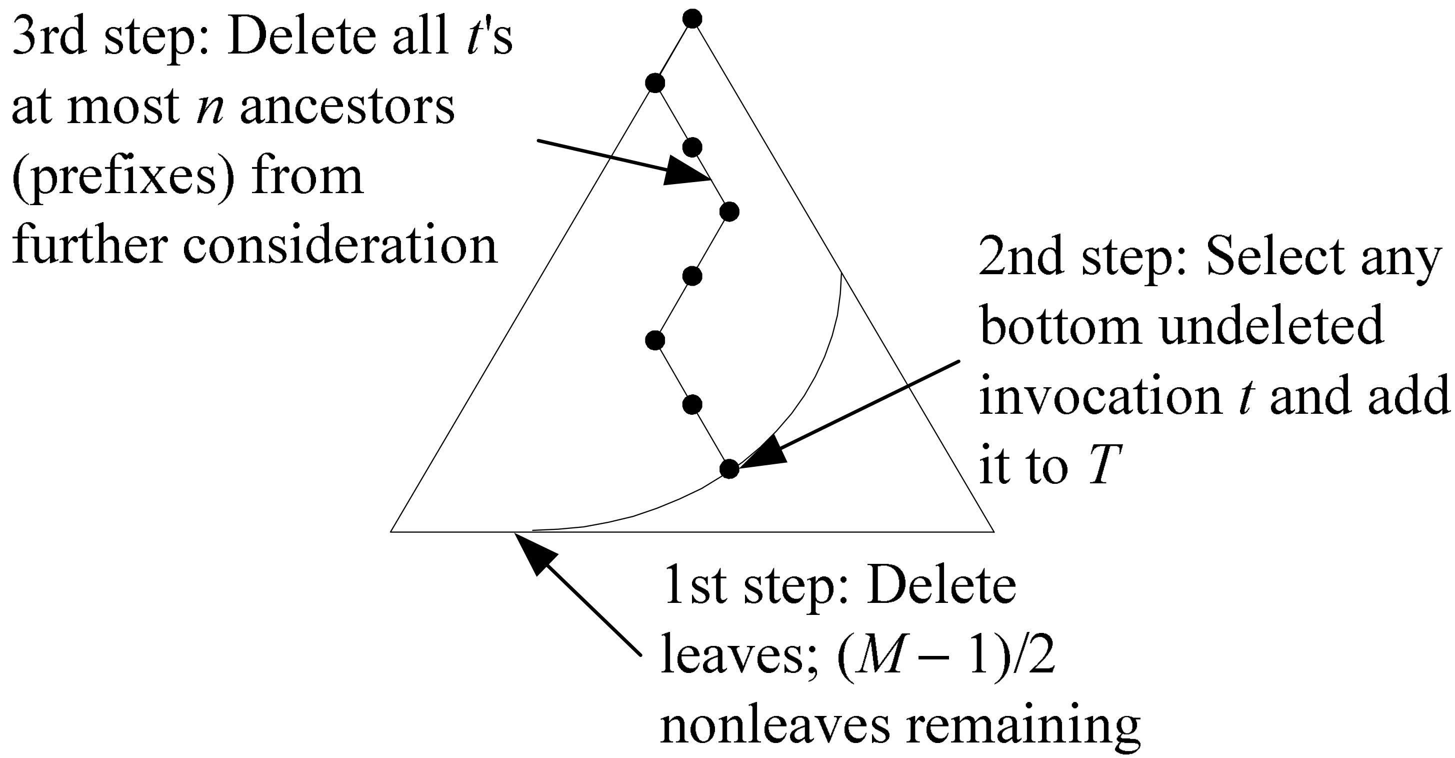
- Since R is a reduction, $R(\phi[\,t\,]) = R(\phi[\,t'\,])$ implies that $\phi[\,t\,]$ and $\phi[\,t'\,]$ must be both satisfiable or unsatisfiable.
- $R(\phi[\,t\,])$ has polynomial length $\leq p(n)$ because R runs in log space.
- As R maps to unary numbers, there are only polynomially many $p(n)$ values of $R(\phi[\,t\,])$.
- How many nodes of the complete binary tree (of invocations/truth assignments) need to be visited?
- If that number is a polynomial, the overall algorithm runs in polynomial time and we are done.

The Proof (continued)

- A search of the table takes time $O(p(n))$ in the random access memory model.
- The running time is $O(Mp(n))$, where M is the total number of invocations of the algorithm.
- The invocations of the algorithm form a binary tree of depth at most n.

The Proof (continued)

- There is a set $T = \{t_1, t_2, \ldots\}$ of invocations (partial truth assignments, i.e.) such that:
  - $|T| \geq (M-1)/(2n)$.
  - All invocations in T are recursive (nonleaves).
  - None of the elements of T is a prefix of another.


3rd step: Delete all t's at most n ancestors (prefixes) from further consideration
2nd step: Select any bottom undeleted invocation t and add it to T
1st step: Delete leaves; $(M-1)/2$ nonleaves remaining

The Proof (continued)

- All invocations $t \in T$ have different $R(\phi[\,t\,])$ values.
  - None of $s, t \in T$ is a prefix of another.
  - The invocation of one started after the invocation of the other had terminated.
  - If they had the same value, the one that was invoked second would have looked it up, and therefore would not be recursive, a contradiction.
- The existence of T implies that there are at least $(M-1)/(2n)$ different $R(\phi[\,t\,])$ values in the table.

The Proof (concluded)

- We already know that there are at most $p(n)$ such values.
- Hence $(M-1)/(2n) \leq p(n)$.
- Thus $M \leq 2np(n)+1$.
- The running time is therefore $O(Mp(n)) = O(np^2(n))$.
- We comment that this theorem holds for any sparse language, not just unary ones.[a]

[a]Mahaney (1980).

coNP-Completeness and Density

Theorem 79 (Fortung (1979)) *If a unary language $U \subseteq \{0\}^*$ is coNP-complete, then P = NP.*

- Suppose there is a reduction R from SAT COMPLEMENT to U.
- The rest of the proof is basically identical except that, now, we want to make sure a formula is unsatisfiable.

Oracles[a]

- We will be considering TMs with access to a "subroutine" or black box.
- This black box solves a language problem L (such as SAT) *in one step*.
- By presenting an input x to the black box, in one step the black box returns "yes" or "no" depending on whether $x \in L$.
- This black box is called aptly an **oracle**.

[a] Turing (1936).

Oracle Turing Machines

- A **Turing machine** $M^?$ **with oracle** is a multistring deterministic TM.
- It has a special string called the **query string**.
- It also has three special states:
  - $q_?$ (the **query state**).
  - q_{yes} and q_{no} (the **answer states**).

Oracle Turing Machines (concluded)

- Let $A \subseteq \Sigma^*$ be a language.
- From $q?$, $M^?$ moves to either q_{yes} or q_{no} depending on whether the current query string is in A or not.
  - This piece of information can be used by $M^?$.
  - Think of A as a black box or a vendor-supplied subroutine.
- $M^?$ is otherwise like an ordinary TM.
- $M^A(x)$ denotes the computation of $M^?$ with oracle A on input x.

Complexity Measures of Oracle TMs

- The time complexity for oracle TMs is like that for ordinary TMs.
- Nondeterministic oracle TMs are defined in the same way.
- Let $\mathcal{C}$ be a deterministic or nondeterministic time complexity class.
- Define $\mathcal{C}^A$ to be the class of all languages decided (or accepted) by machines in $\mathcal{C}$ with access to oracle A.

An Example

- SAT COMPLEMENT $\in \mathrm{P}^{\mathrm{SAT}}$.
  - Reverse the answer of SAT oracle A as our answer.

    if $\phi \in A$ **then**
        return "no"; $\{\phi$ is satisfiable.$\}$
    else
        return "yes"; $\{\phi$ is not satisfiable.$\}$
    end if

- As SAT COMPLEMENT is coNP-complete (p. 344),

$$\mathrm{coNP} \subseteq \mathrm{P}^{\mathrm{SAT}}.$$

The Turing Reduction

- Recall L_1 is reducible to L_2 if there is a logspace function R such that $x \in L_1 \Leftrightarrow R(x) \in L_2$ (p. 195).
  - It is called **logspace reduction**, Karp reduction (p. 197), or **many-one reduction**.
- But the reduction in proving $L \in \mathcal{C}^A$ is more general.
  - An algorithm B for $\mathcal{C}$ with access to A exists.
  - B can call A many times within the resource bound.
  - We say L is **Turing-reducible** to A.

Two Types of Reductions

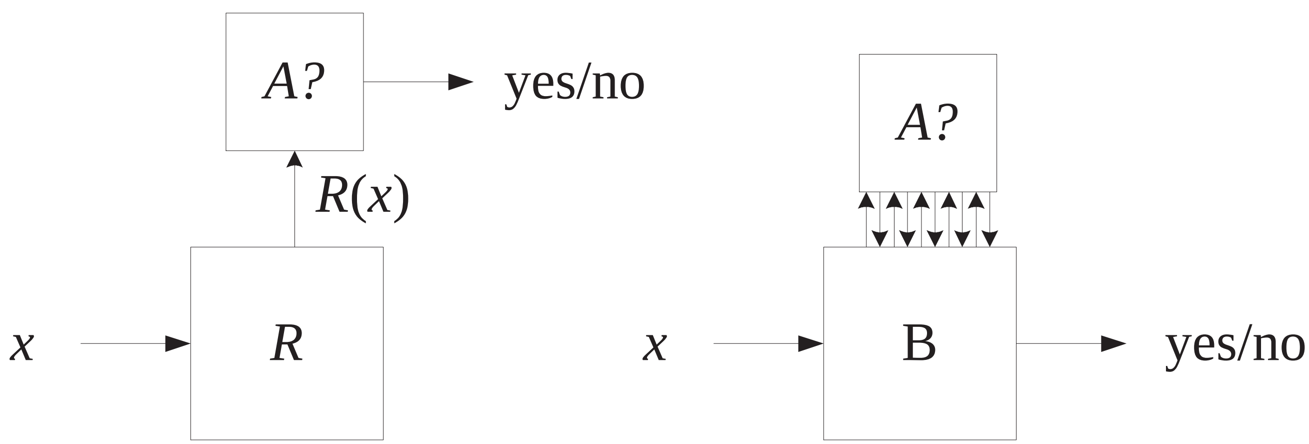
Lemma 80 *If L_1 is (logspace-) reducible to L_2, then L_1 is Turing-reducible to L_2.*

- Logspace reduction is more restrictive than Turing reduction.
- It is Turing reduction with only one query to L_2.
- Note also that a language in L also belongs in P.

Corollary 81 *If L is complete under logspace-reductions, then L is complete under Turing reductions.*

Two Types of Reductions (continued)

- Turing reduction is more general than (p. 627)—and equally valid as—logspace reduction.



- This is true even if B runs in logarithmic space and oracle A is queried only once.

Two Types of Reductions (continued)

- Turing reduction is more powerful than logspace reduction.
- For example, there are languages A and B such that A is Turing-reducible to B but not logspace-reducible to B.[a]
- However, for the class NP, no such separation has been proved.[b]

[a]Ladner, Lynch, and Selman (1975).

[b]If we assume NP does not have p-measure 0, then separation exists (Lutz and Mayordomo (1996)).

Two Types of Reductions (concluded)

- The Turing reduction is adaptive.
  - Later queries may depend on prior queries.
- If we restrict the Turing reduction to ask all queries before receiving any answers, the reduction is called the **truth-table reduction**.
- Separation results exist for the Turing and truth-table reductions given some conjectures.[a]

[a] Hitchcock and Pavan (2006).

The Power of Turing Reduction

- SAT COMPLEMENT is not likely to be reducible to SAT.
  - Otherwise, coNP = NP as SAT COMPLEMENT is coNP-complete (p. 344).
- But SAT COMPLEMENT is polynomial-time Turing-reducible to SAT.
  - SAT COMPLEMENT $\in \mathrm{P}^{\mathrm{SAT}}$ (p. 625).
  - True even though the oracle SAT is called only once!
  - The algorithm on p. 625 is *not* a logspace reduction.

Computation That Counts

Counting Problems

- Counting problems are concerned with the number of solutions.
  - #SAT: the number of satisfying truth assignments to a boolean formula.
  - #HAMILTONIAN PATH: the number of Hamiltonian paths in a graph.
- They cannot be easier than their decision versions.
  - The decision problem has a solution if and only if the solution count is larger than 0.
- But they can be harder than their decision versions.

Decision and Counting Problems

- FP is the set of polynomial-time computable functions $f : \{0, 1\}^* \to \mathbb{Z}$.
  - GCD, LCM, matrix-matrix multiplication, etc.
- If $\#\text{SAT} \in \text{FP}$, then $\text{P} = \text{NP}$.
  - Given boolean formula ϕ, calculate its number of satisfying truth assignments, k, in polynomial time.
  - Declare "$\phi \in \text{SAT}$" if and only if $k \geq 1$.
- The validity of the reverse direction is open.

A Counting Problem Harder than Its Decision Version

- Some counting problems are harder than their decision versions.
- CYCLE asks if a directed graph contains a cycle.
- #CYCLE counts the number of cycles in a directed graph.
- CYCLE is in P by a simple greedy algorithm.
- But #CYCLE is hard unless P = NP.

Counting Class #P

A function f is in #P (or $f \in$ #P) if

- There exists a polynomial-time NTM M.
- $M(x)$ has $f(x)$ accepting paths for all inputs x.
- $f(x) =$ number of accepting paths of $M(x)$.

Some #P Problems

- $f(\phi)$ = number of satisfying truth assignments to ϕ.
  - The desired NTM guesses a truth assignment T and accepts ϕ if and only if $T \models \phi$.
  - Hence $f \in$ #P.
  - f is also called #SAT.
- #HAMILTONIAN PATH.
- #3-COLORING.

#P Completeness

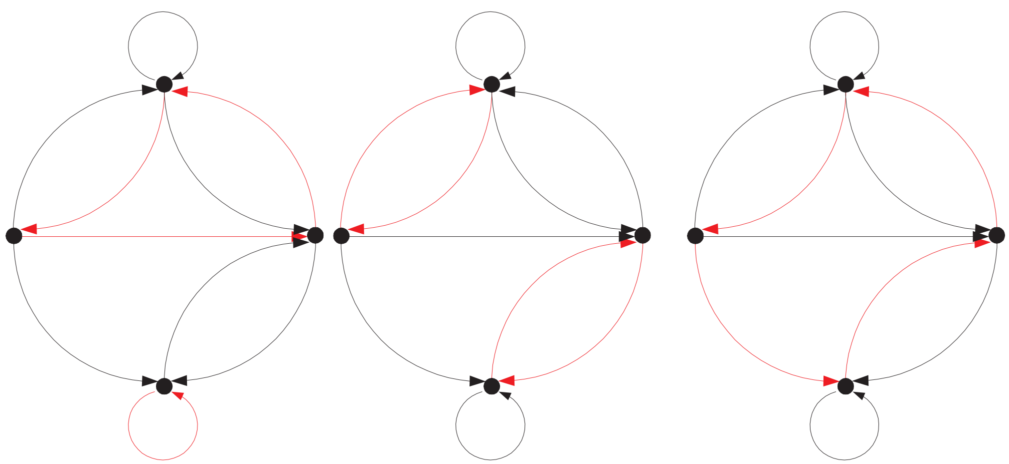
- Function f is #P-complete if
  - $f \in$ #P.
  - #P $\subseteq \mathrm{FP}^f$.
    - Every function in #P can be computed in polynomial time with access to a black box or **oracle** for f.
  - Of course, oracle f will be accessed only a polynomial number of times.
  - #P is said to be **polynomial-time Turing-reducible to** f.

#SAT Is #P-Complete

- First, it is in #P (p. 637).
- Let $f \in \#\text{P}$ compute the number of accepting paths of M.
- Cook's theorem uses a *parsimonious* reduction from M on input x to an instance ϕ of SAT (p. 247).
  - Hence the number of accepting paths of $M(x)$ equals the number of satisfying truth assignments to ϕ.
- Call the oracle #SAT with ϕ to obtain the desired answer regarding $f(x)$.

CYCLE COVER

- A set of node-disjoint cycles that cover all nodes in a directed graph is called a **cycle cover**.



- There are 3 cycle covers (in red) above.

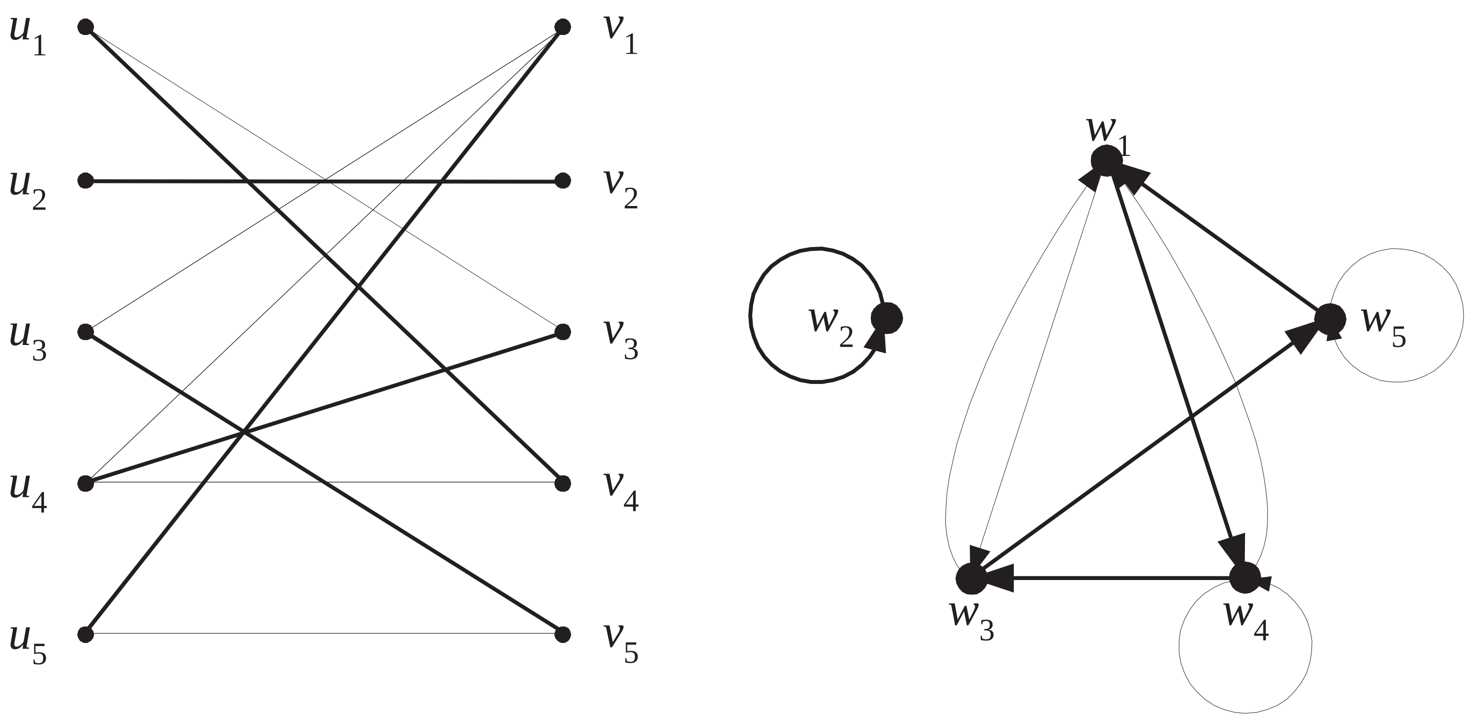
CYCLE COVER and BIPARTITE PERFECT MATCHING

Proposition 82 CYCLE COVER *and* BIPARTITE PERFECT MATCHING *(p. 390) are parsimoniously reducible to each other.*

- A polynomial-time algorithm creates a bipartite graph G' from any directed graph G.
- Moreover, the number cycle covers for G equals the number of bipartite perfect matchings for G'.
- And vice versa.

Corollary 83 CYCLE COVER $\in P$.

Illustration of the Proof



Permanent

- The **permanent** of an $n \times n$ integer matrix A is

$$\text{perm}(A) = \sum_{\pi} \prod_{i=1}^{n} A_{i,\pi(i)}.$$

  - π ranges over all permutations of n elements.
- 0/1 PERMANENT computes the permanent of a 0/1 (binary) matrix.
  - The permanent of a binary matrix is at most $n!$.
- Simpler than determinant (5) on p. 392: no signs.
- But, surprisingly, much harder to compute than determinant!

Permanent and Counting Perfect Matchings

- BIPARTITE PERFECT MATCHING is related to determinant (p. 393).
- #BIPARTITE PERFECT MATCHING is related to permanent.

Proposition 84 0/1 PERMANENT *and* BIPARTITE PERFECT MATCHING *are parsimoniously reducible to each other.*

The Proof

- Given a bipartite graph G, construct an $n \times n$ binary matrix A.
  - The (i, j)th entry A_{ij} is 1 if $(i, j) \in E$ and 0 otherwise.
- Then $\text{perm}(A) =$ number of perfect matchings in G.

Illustration of the Proof Based on p. 642 (Left)

$$A = \begin{bmatrix} 0 & 0 & 1 & \boxed{1} & 0 \\ 0 & \boxed{1} & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & \boxed{1} \\ 1 & 0 & \boxed{1} & 1 & 0 \\ \boxed{1} & 0 & 0 & 0 & 1 \end{bmatrix}.$$

- $\text{perm}(A) = 4$.
- The permutation corresponding to the perfect matching on p. 642 is marked.

Permanent and Counting Cycle Covers

Proposition 85 *0/1* PERMANENT *and* CYCLE COVER *are parsimoniously reducible to each other.*

- Let A be the adjacency matrix of the graph on p. 642 (right).
- Then $\text{perm}(A) =$ number of cycle covers.

Three Parsimoniously Equivalent Problems

From Propositions 82 (p. 641) and 84 (p. 644), we summarize:

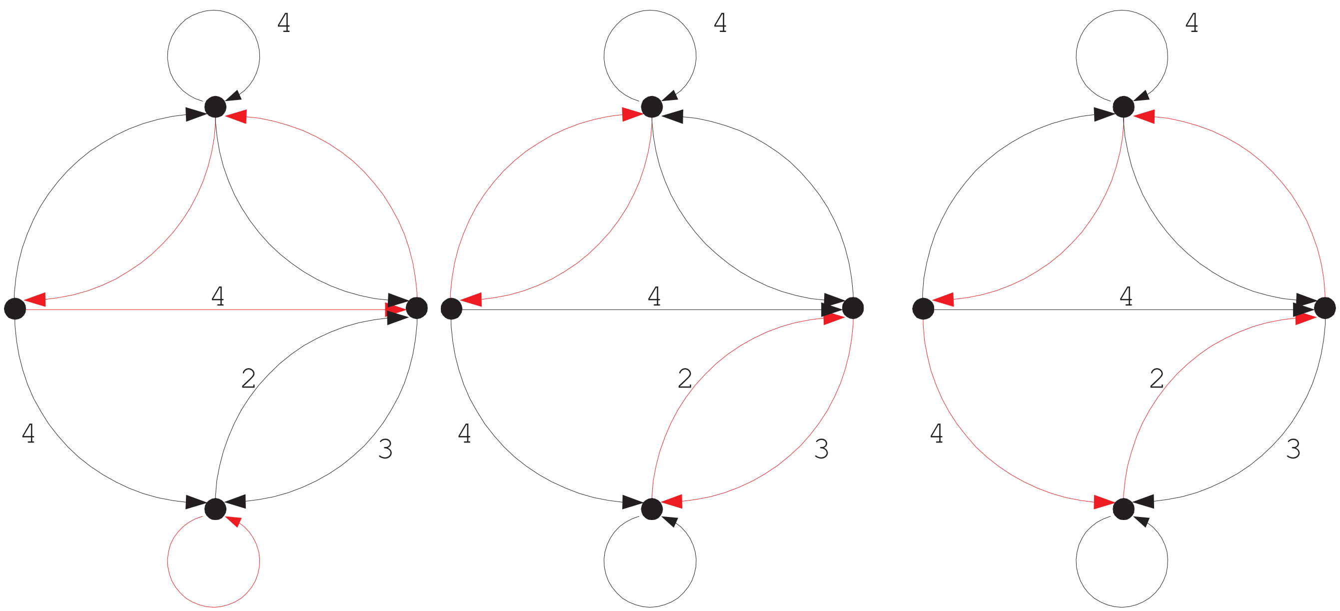
Lemma 86 0/1 PERMANENT, BIPARTITE PERFECT MATCHING, *and* CYCLE COVER *are* **parsimoniously equivalent**.

We will show that the counting versions of all three problems are in fact #P-complete.

WEIGHTED CYCLE COVER

- Consider a directed graph G with integer weights on the edges.
- The weight of a cycle cover is the product of its edge weights.
- The **cycle count** of G is sum of the weights of all cycle covers.
  - Let A be G's adjacency matrix but $A_{ij} = w_i$ if the edge (i, j) has weight w_i.
  - Then $\text{perm}(A) = G$'s cycle count (same proof as Proposition 85 on p. 647).
- #CYCLE COVER is a special case: All weights are 1.

An Example[a]



There are 3 cycle covers, and the cycle count is

$$(4 \cdot 1 \cdot 1) \cdot (1) + (1 \cdot 1) \cdot (2 \cdot 3) + (4 \cdot 2 \cdot 1 \cdot 1) = 18.$$

[a] Each edge has weight 1 unless stated otherwise.

Three #P-Complete Counting Problems

Theorem 87 (Valiant (1979)) 0/1 PERMANENT, #BIPARTITE PERFECT MATCHING, *and* #CYCLE COVER *are #P-complete.*

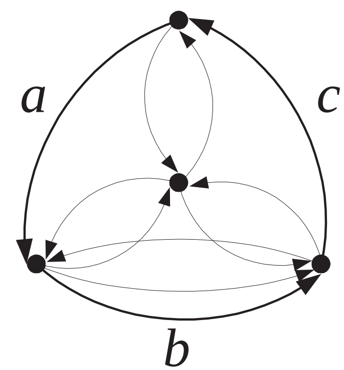
- By Lemma 86 (p. 648), it suffices to prove that #CYCLE COVER is #P-complete.
- #SAT is #P-complete (p. 639).
- #3SAT is #P-complete because it and #SAT are parsimoniously equivalent (p. 256).
- We shall prove that #3SAT is polynomial-time Turing-reducible to #CYCLE COVER.

The Proof (continued)

- Let ϕ be the given 3SAT formula.
  - It contains n variables and m clauses (hence $3m$ literals).
  - It has $\#\phi$ satisfying truth assignments.
- First we construct a *weighted* directed graph H with cycle count
$$\#H = 4^{3m} \times \#\phi.$$
- Then we construct an unweighted directed graph G.
- We make sure $\#H$ (hence $\#\phi$) is polynomial-time Turing-reducible to G's number of cycle covers (denoted $\#G$).

The Proof: the Clause Gadget (continued)

- Each clause is associated with a **clause gadget**.



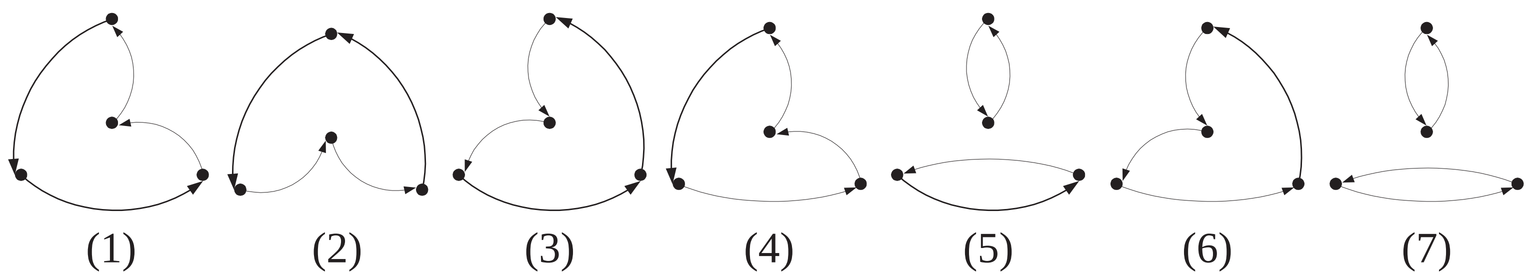
- Each edge has weight 1 unless stated otherwise.
- Each bold edge corresponds to one literal in the clause.
- There are not *parallel* lines as bold edges are schematic only (preview p. 666).

The Proof: the Clause Gadget (continued)

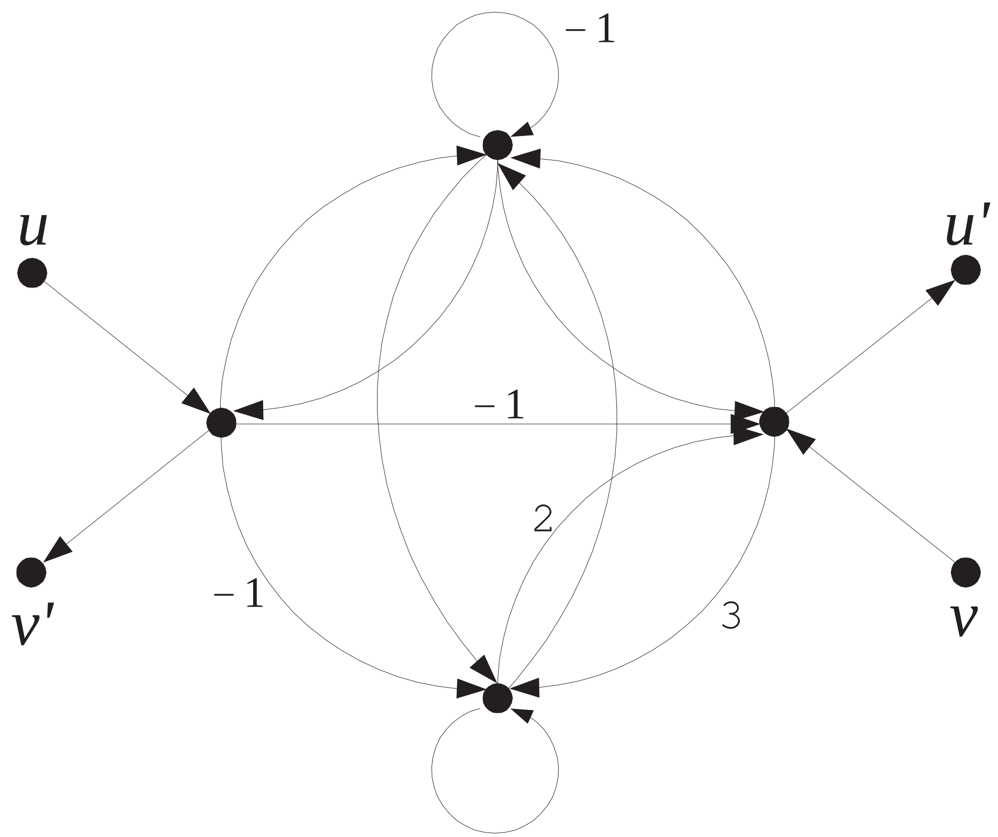
- Following a bold edge means making the literal false (0).
- A cycle cover cannot select *all* 3 bold edges.
  - The interior node would be missing.
- Every proper nonempty subset of bold edges corresponds to a unique cycle cover of weight 1 (see next page).

The Proof: the Clause Gadget (continued)

7 possible cycle covers, one for each satisfying assignment: (1) $a = 0, b = 0, c = 1$, (2) $a = 0, b = 1, c = 0$, etc.

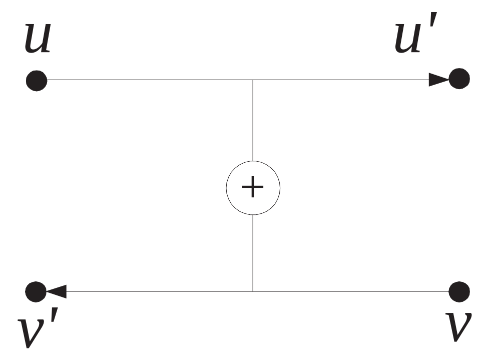


The Proof: the XOR Gadget (continued)



The Proof: Properties of the XOR Gadget (continued)

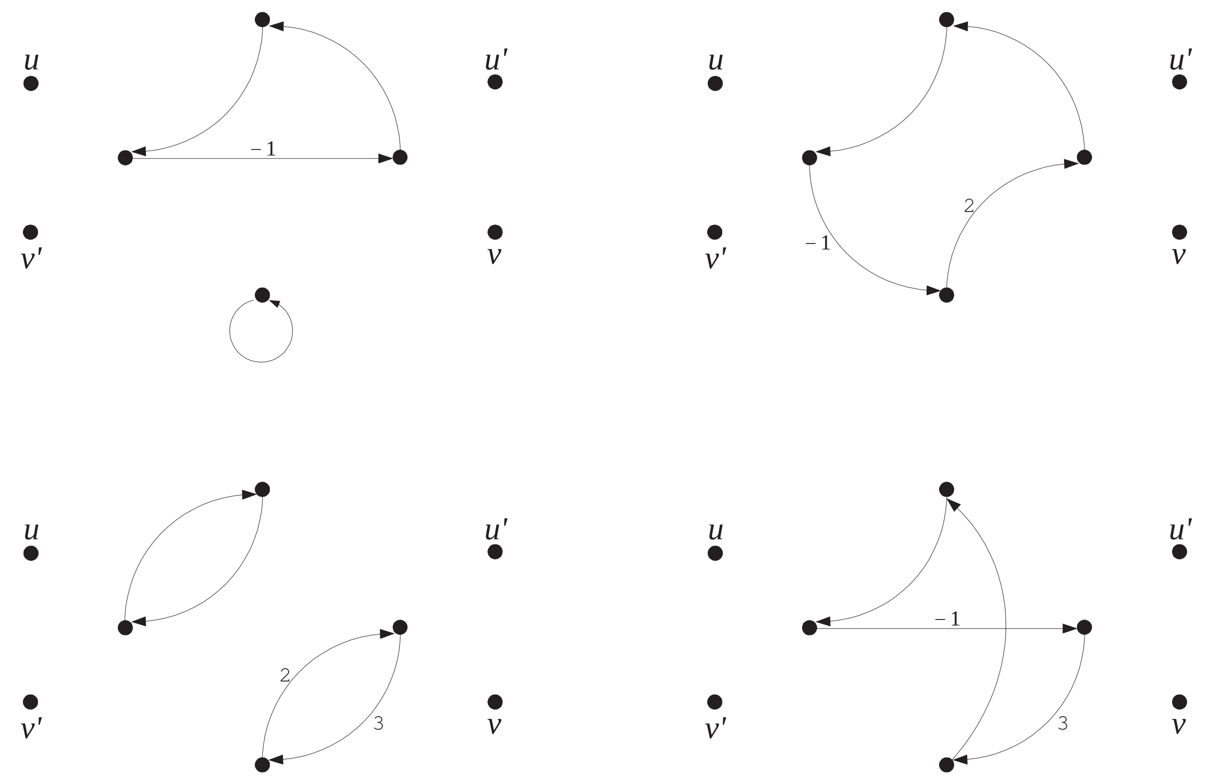
- The XOR gadget schema:



- *At most one* of the 2 schematic edges will be included in a cycle cover.
- There will be $3m$ XOR gadgets, one for each literal.

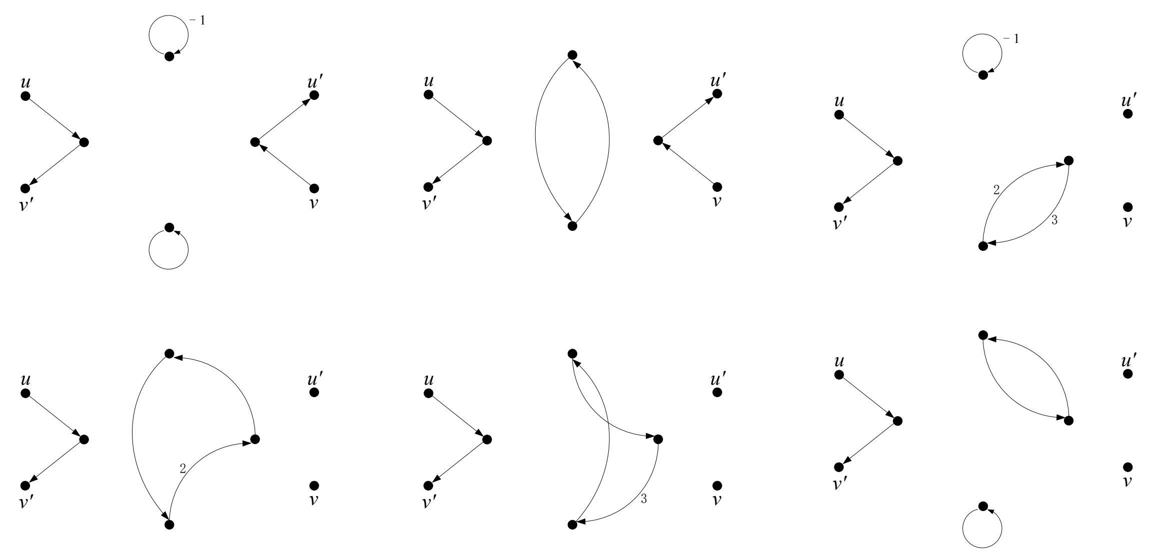
The Proof: Properties of the XOR Gadget (continued)

Total weight of $-1 - 2 + 6 - 3 = 0$ for cycle covers not entering or leaving it.



The Proof: Properties of the XOR Gadget (continued)

- Total weight of $-1 + 1 - 6 + 2 + 3 + 1 = 0$ for cycle covers entering at u and leaving at v'.[a]

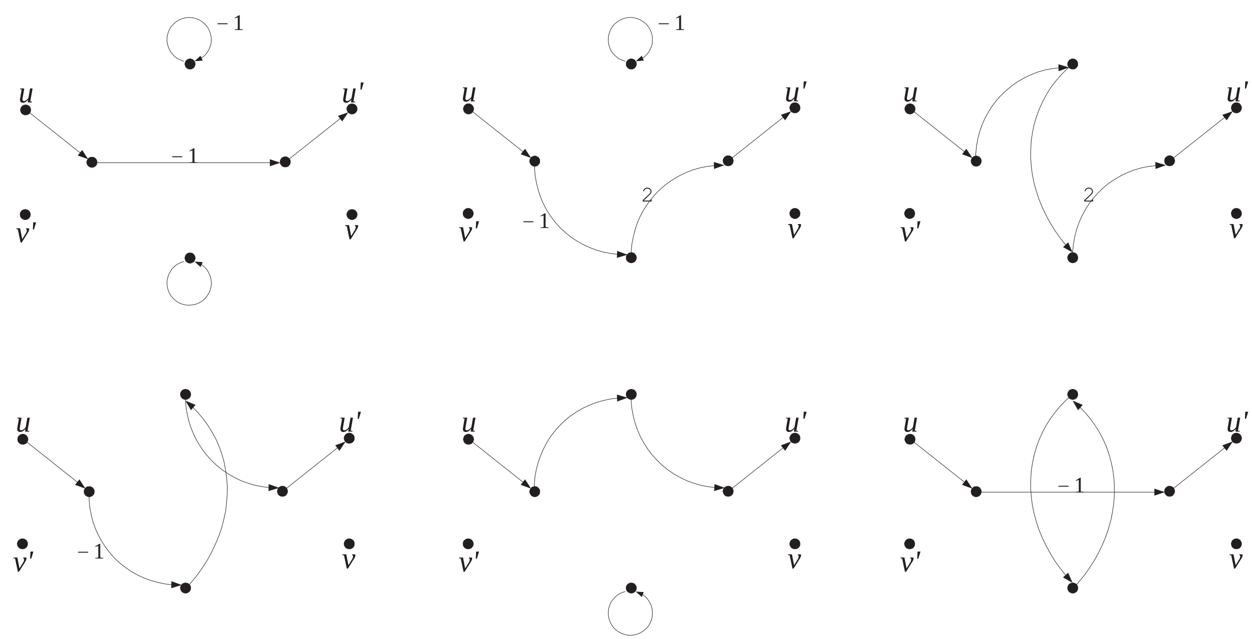


- Same for cycle covers entering at v and leaving at u'.

[a]Corrected by Mr. Yu-Tshung Dai (`B91201046`) and Mr. Che-Wei Chang (`R95922093`) on December 27, 2006.

The Proof: Properties of the XOR Gadget (continued)

- Total weight of $1 + 2 + 2 - 1 + 1 - 1 = 4$ for cycle covers entering at u and leaving at u'.



- Same for cycle covers entering at v and leaving at v'.

The Proof: Summary (continued)

- Cycle covers not entering *all* of the XOR gadgets contribute 0 to the cycle count.
  - Let x denote an XOR gadget not entered for a cycle cover c.
  - Now, the said cycle covers' total contribution is

$$
\begin{aligned}
&= \sum_{\text{cycle cover } c \text{ for } H} \text{weight}(c) \\
&= \sum_{\text{cycle cover } c \text{ for } H-x} \text{weight}(c) \sum_{\text{cycle cover } c \text{ for } x} \text{weight}(x) \\
&= \sum_{\text{cycle cover } c \text{ for } H-x} \text{weight}(c) \cdot 0 \\
&= 0.
\end{aligned}
$$

The Proof: Summary (continued)

- Cycle covers entering *any* of the XOR gadgets and leaving illegally contribute 0 to the cycle count.
- For every XOR gadget entered and exited legally, the total weight of a cycle cover is multiplied by 4.
  - With an XOR gadget x entered and exited legally fixed,

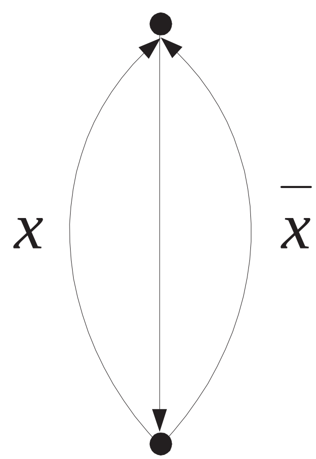
$$
\begin{aligned}
& \text{contributions of such cycle covers to the cycle count} \\
& \sum_{\text{cycle cover } c \text{ for } H} \text{weight}(c) \\
= & \sum_{\text{cycle cover } c \text{ for } H - x} \text{weight}(c) \sum_{\text{cycle cover } c \text{ for } x} \text{weight}(x) \\
= & \sum_{\text{cycle cover } c \text{ for } H - x} \text{weight}(c) \cdot 4.
\end{aligned}
$$

The Proof: Summary (continued)

- Hereafter we consider only cycle covers which enter every XOR gadget and leaves it legally.
  - Only these cycle covers contribute nonzero weights to the cycle count.
- They are said to **respect** the XOR gadgets.

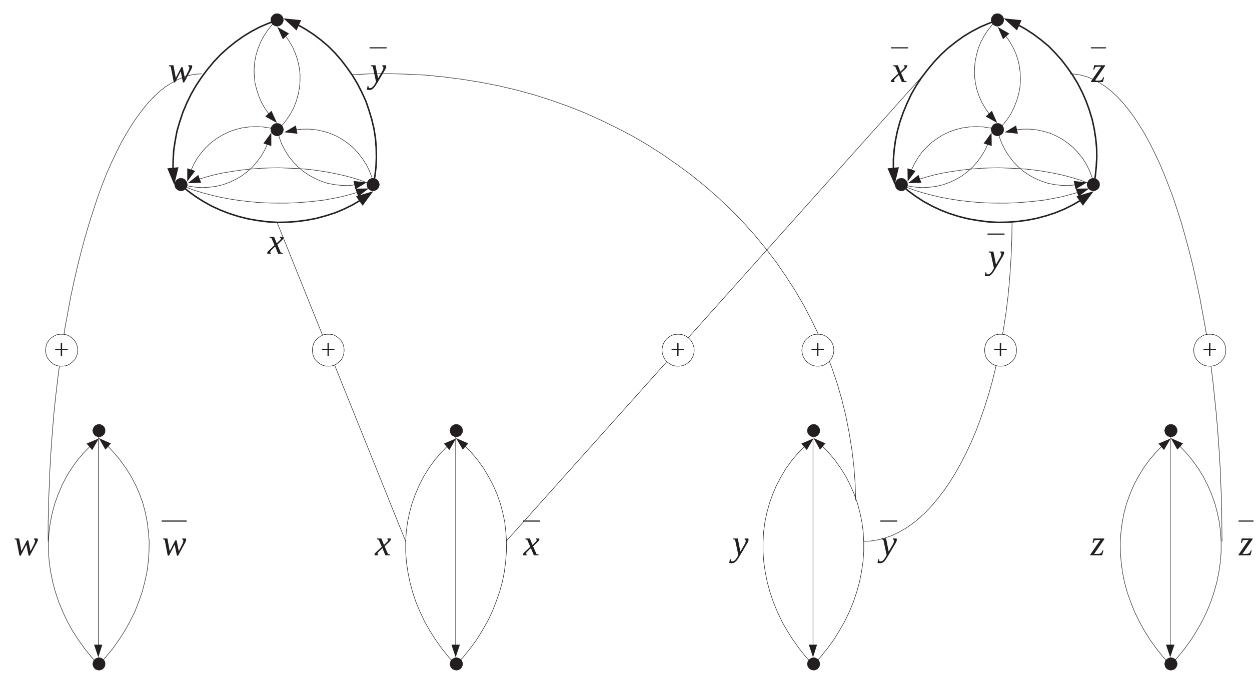
The Proof: the Choice Gadget (continued)

- One choice gadget (a schema) for each variable.

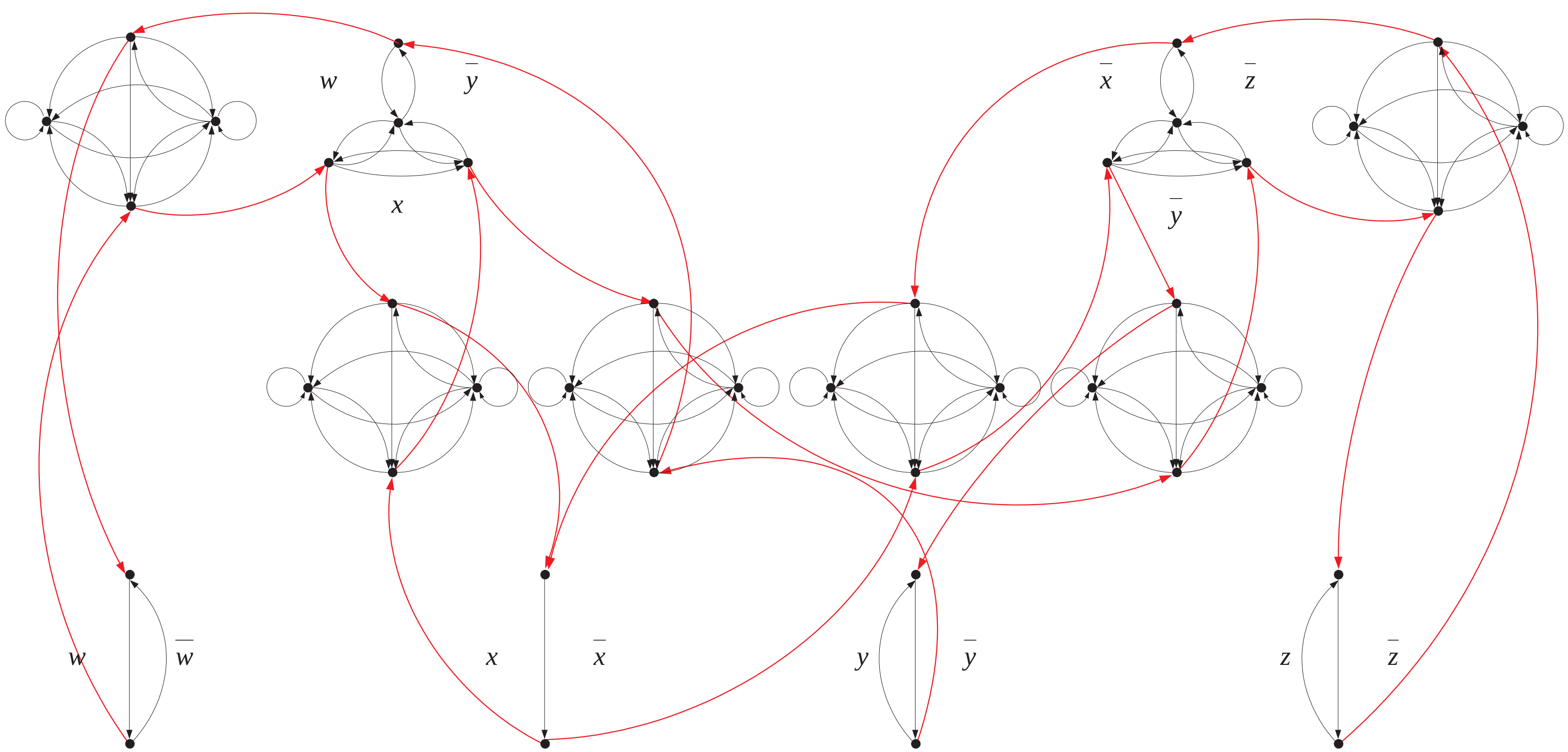


- It gives the truth assignment for the variable.
- Use it with the XOR gadget to enforce consistency.

Schema for $(w \vee x \vee \bar{y}) \wedge (\bar{x} \vee \bar{y} \vee \bar{z})$



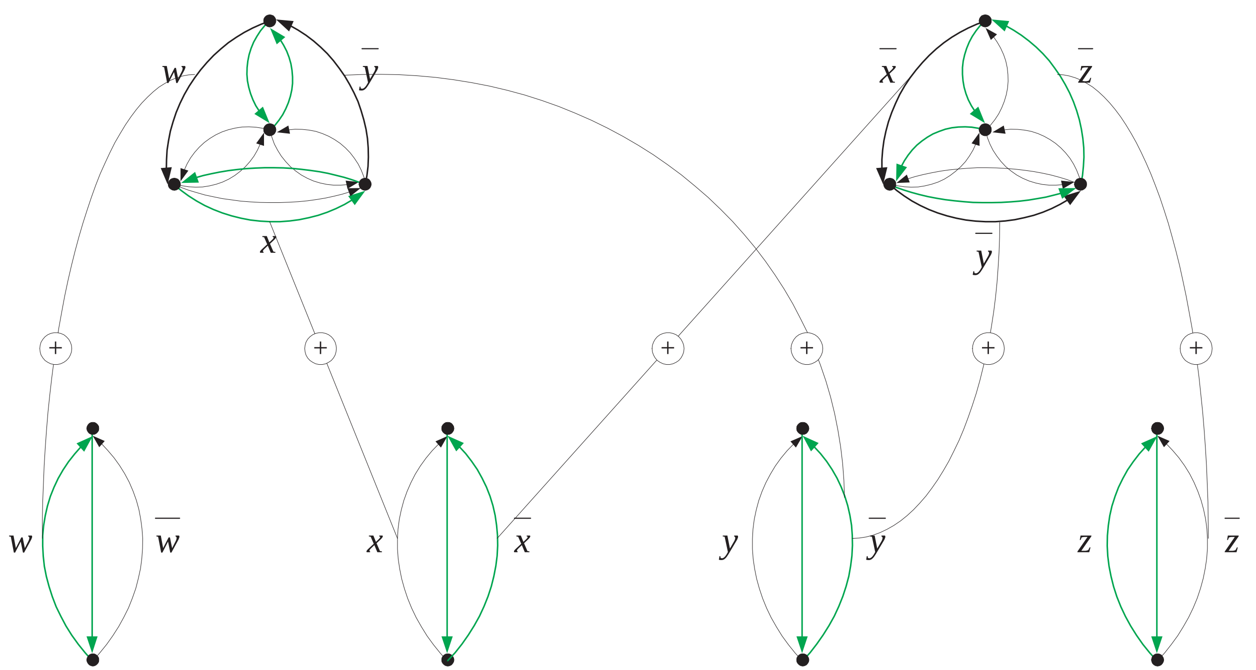
Full Graph $(w \vee x \vee \bar{y}) \wedge (\bar{x} \vee \bar{y} \vee \bar{z})$



The Proof: a Key Observation (continued)

Each satisfying truth assignment to ϕ corresponds to a schematic cycle cover that respects the XOR gadgets.

$w = 1, x = 0, y = 0, z = 1 \Leftrightarrow$ One Cycle Cover



The Proof: a Key Corollary (continued)

- Recall that there are $3m$ XOR gadgets.
- Each satisfying truth assignment to ϕ contributes 4^{3m} to the cycle count $\#H$.
- Hence

$$\#H = 4^{3m} \times \#\phi,$$

as desired.