Complexity of Counting

Lecture 20

#P

FP

FP

- Turing Machines computing a (not necessarily Boolean) function of the input

FP

- Turing Machines computing a (not necessarily Boolean) function of the input
  - Writes the output on an output tape

FP

- Turing Machines computing a (not necessarily Boolean) function of the input
  - Writes the output on an output tape
- FP: class of efficiently computable functions

FP

- Turing Machines computing a (not necessarily Boolean) function of the input
  - Writes the output on an output tape
- FP: class of efficiently computable functions
  - Computed by a TM running in polynomial time

Counting Problems

Counting Problems

- Counting: Functions of the form "number of witnesses"

Counting Problems

- Counting: Functions of the form "number of witnesses"
  - $\#R(x) = |\{w : R(x,w)=1\}|$

Counting Problems

- Counting: Functions of the form "number of witnesses"
  - $\#R(x) = |\{w: R(x,w)=1\}|$
    - e.g: Number of subgraphs of a given graph with some property (trees, cycles, spanning trees, cycle covers, etc.)

Counting Problems

- Counting: Functions of the form "number of witnesses"
  - $\#R(x) = |\{w: R(x,w)=1\}|$
    - e.g: Number of subgraphs of a given graph with some property (trees, cycles, spanning trees, cycle covers, etc.)
    - e.g.: Number of satisfying assignments to a boolean formula

Counting Problems

- Counting: Functions of the form "number of witnesses"
  - $\#R(x) = |\{w: R(x,w)=1\}|$
    - e.g: Number of subgraphs of a given graph with some property (trees, cycles, spanning trees, cycle covers, etc.)
    - e.g.: Number of satisfying assignments to a boolean formula
    - e.g.: Number of inputs less than x (lexicographically) that are in a language L

#P

#P

- Class of functions of the form number of witnesses for an NP language

#P

- Class of functions of the form number of witnesses for an NP language
  - #R(x) = |{w: R(x,w)=1}|, where R is a polynomial time relation

#P

- Class of functions of the form number of witnesses for an NP language
  - #R(x) = |{w: R(x,w)=1}|, where R is a polynomial time relation
    - e.g.: #SPANTREE(G) = number of spanning trees in a graph G

#P

- Class of functions of the form number of witnesses for an NP language
  - $\#R(x) = |\{w: R(x,w)=1\}|$, where R is a polynomial time relation
    - e.g.: #SPANTREE(G) = number of spanning trees in a graph G
    - e.g.: #CYCLE(G) = number of simple cycles in a directed graph G

#P

- Class of functions of the form number of witnesses for an NP language
  - $\#R(x) = |\{w: R(x,w)=1\}|$, where R is a polynomial time relation
    - e.g.: #SPANTREE(G) = number of spanning trees in a graph G
    - e.g.: #CYCLE(G) = number of simple cycles in a directed graph G
    - e.g.: #SAT(φ) = number of satisfying assignments of φ

#P

- Class of functions of the form number of witnesses for an NP language
  - $\#R(x) = |\{w: R(x,w)=1\}|$, where R is a polynomial time relation
    - e.g.: #SPANTREE(G) = number of spanning trees in a graph G
    - e.g.: #CYCLE(G) = number of simple cycles in a directed graph G
    - e.g.: $\#SAT(\varphi)$ = number of satisfying assignments of φ
- Easy to see: $FP \subseteq \#P$ [Exercise]

#P vs. NP

#P vs. NP

- $\#R(x) = |\{w: R(x,w)=1\}|$, were R is a polynomial time relation

#P vs. NP

- #R(x) = |{w: R(x,w)=1}|, were R is a polynomial time relation
  - To compute a function in #P: compute #R(x)

#P vs. NP

- #R(x) = |{w: R(x,w)=1}|, were R is a polynomial time relation
  - To compute a function in #P: compute #R(x)
  - To decide a language in NP: check if #R(x) > 0

#P vs. NP

- $\#R(x) = |\{w: R(x,w)=1\}|$, were R is a polynomial time relation
  - To compute a function in #P: compute #R(x)
  - To decide a language in NP: check if #R(x) > 0
- #P "harder" than NP

#P vs. NP

- #R(x) = |{w: R(x,w)=1}|, were R is a polynomial time relation
  - To compute a function in #P: compute #R(x)
  - To decide a language in NP: check if #R(x) > 0
- #P "harder" than NP
  - If #P = FP, then P = NP

#P vs. NP

- $\#R(x) = |\{w: R(x,w)=1\}|$, were R is a polynomial time relation
  - To compute a function in #P: compute #R(x)
  - To decide a language in NP: check if #R(x) > 0
- #P "harder" than NP
  - If #P = FP, then P = NP
  - How much harder?

How hard is it to count?

How hard is it to count?

- Not hard for some problems

How hard is it to count?

- Not hard for some problems
  - e.g.: #SPANTREE(G) = number of spanning trees in a graph G

How hard is it to count?

- Not hard for some problems
  - e.g.: #SPANTREE(G) = number of spanning trees in a graph G
    - Kirchhoff's theorem: evaluating a simple determinant gives the answer

How hard is it to count?

- Not hard for some problems
  - e.g.: #SPANTREE(G) = number of spanning trees in a graph G
    - Kirchhoff's theorem: evaluating a simple determinant gives the answer
- Hard for counting witnesses of NP-complete languages: e.g. #SAT (unless P=NP)

How hard is it to count?

- Not hard for some problems
  - e.g.: #SPANTREE(G) = number of spanning trees in a graph G
    - Kirchhoff's theorem: evaluating a simple determinant gives the answer
- Hard for counting witnesses of NP-complete languages: e.g. #SAT (unless P=NP)
- Hard for some other problems too

How hard is it to count?

- Not hard for some problems
  - e.g.: #SPANTREE(G) = number of spanning trees in a graph G
    - Kirchhoff's theorem: evaluating a simple determinant gives the answer
- Hard for counting witnesses of NP-complete languages: e.g. #SAT (unless P=NP)
- Hard for some other problems too
  - If #CYCLE $\in$ FP, then P=NP

$\#CYCLE \in FP \Rightarrow P=NP$

#CYCLE ∈ FP ⇒ P=NP

- Reduce HAMILTONICITY to #CYCLE: Given G, to construct G′ such that #CYCLE(G′) is “large” iff G has a Hamiltonian cycle

#CYCLE ∈ FP ⇒ P=NP

- Reduce HAMILTONICITY to #CYCLE: Given G, to construct G′ such that #CYCLE(G′) is "large" iff G has a Hamiltonian cycle
  - Replace each edge in G by a gadget such that each cycle in G becomes "many" cycles in G′

#CYCLE ∈ FP ⇒ P=NP

- Reduce HAMILTONICITY to #CYCLE: Given G, to construct G′ such that #CYCLE(G′) is “large” iff G has a Hamiltonian cycle
  - Replace each edge in G by a gadget such that each cycle in G becomes “many” cycles in G′
  - Longer the cycle in G, more the cycles in G′ it results in

#CYCLE $\in$ FP $\Rightarrow$ P=NP

- Reduce HAMILTONICITY to #CYCLE: Given G, to construct G′ such that #CYCLE(G′) is "large" iff G has a Hamiltonian cycle
  - Replace each edge in G by a gadget such that each cycle in G becomes "many" cycles in G′
  - Longer the cycle in G, more the cycles in G′ it results in
  - A single n-long cycle in G will result in more cycles in G′ than produced by all shorter cycles in G put together

#CYCLE $\in$ FP $\Rightarrow$ P=NP

- Reduce HAMILTONICITY to #CYCLE: Given G, to construct G′ such that #CYCLE(G′) is "large" iff G has a Hamiltonian cycle
  - Replace each edge in G by a gadget such that each cycle in G becomes "many" cycles in G′
  - Longer the cycle in G, more the cycles in G′ it results in
  - A single n-long cycle in G will result in more cycles in G′ than produced by all shorter cycles in G put together
    - At most n^{n-1} shorter cycles in G

$\#CYCLE \in FP \Rightarrow P=NP$

- Reduce HAMILTONICITY to #CYCLE: Given G, to construct G′ such that #CYCLE(G′) is "large" iff G has a Hamiltonian cycle
  - Replace each edge in G by a gadget such that each cycle in G becomes "many" cycles in G′
  - Longer the cycle in G, more the cycles in G′ it results in
  - A single n-long cycle in G will result in more cycles in G′ than produced by all shorter cycles in G put together
    - At most n^{n-1} shorter cycles in G
    - t-long cycle in G → $(2^m)^t = n^{nt}$ cycles in G′ ($m := n \log n$)

#CYCLE $\in$ FP $\Rightarrow$ P=NP

- Reduce HAMILTONICITY to #CYCLE: Given G, to construct G′ such that #CYCLE(G′) is "large" iff G has a Hamiltonian cycle
  - Replace each edge in G by a gadget such that each cycle in G becomes "many" cycles in G′
  - Longer the cycle in G, more the cycles in G′ it results in
  - A single n-long cycle in G will result in more cycles in G′ than produced by all shorter cycles in G put together
    - At most n^{n-1} shorter cycles in G
    - t-long cycle in G $\rightarrow$ $(2^m)^t = n^{nt}$ cycles in G′ (m := n log n)
      - HAMILTONICITY(G) $\Leftrightarrow$ #CYCLES(G) $\geq n^{n^2}$

#P vs. PP

#P vs. PP

- Recall PP: x in L if for at least half the strings w (of some length) we have R(x,w)=1

#P vs. PP

- Recall PP: x in L if for at least half the strings w (of some length) we have R(x,w)=1
  - i.e., checking the most significant bits of #R

#P vs. PP

- Recall PP: x in L if for at least half the strings w (of some length) we have R(x,w)=1
  - i.e., checking the most significant bits of #R
  - Recall: We already saw NP $\subseteq$ PP

#P vs. PP

- Recall PP: x in L if for at least half the strings w (of some length) we have R(x,w)=1
  - i.e., checking the most significant bits of #R
  - Recall: We already saw NP $\subseteq$ PP
  - PP as powerful as #P (and vice versa)

#P vs. PP

- Recall PP: x in L if for at least half the strings w (of some length) we have R(x,w)=1
  - i.e., checking the most significant bits of #R
  - Recall: We already saw NP ⊆ PP
  - PP as powerful as #P (and vice versa)
    - #P ⊆ FP^{PP} [exercise] (and PP ⊆ $P^{\#P}$ [why?])

#P vs. PP

- Recall PP: x in L if for at least half the strings w (of some length) we have R(x,w)=1
  - i.e., checking the most significant bits of #R
  - Recall: We already saw NP ⊆ PP
  - PP as powerful as #P (and vice versa)
    - $\#P \subseteq FP^{PP}$ [exercise] (and $PP \subseteq P^{\#P}$ [why?])
    - So if PP = P, then #P = FP (and vice versa)

#P completeness

#P completeness

- $f \in$ #P is #P-complete if any $g \in$ #P can be Cook-reduced to f

#P completeness

Allows multiple oracle calls. Alternately, allow only one call

- $f \in \#P$ is #P-complete if any $g \in \#P$ can be Cook-reduced to f

#P completeness

Allows multiple oracle calls. Alternately, allow only one call

- f ∈ #P is #P-complete if any g ∈ #P can be Cook-reduced to f
  - From parsimonious reduction of g's NP problem to an NP-complete problem (w.r.t Karp-reductions)

#P completeness

Allows multiple oracle calls. Alternately, allow only one call

- f ∈ #P is #P-complete if any g ∈ #P can be Cook-reduced to f
  - From parsimonious reduction of g's NP problem to an NP-complete problem (w.r.t Karp-reductions)
    - #SAT is #P-complete

#P completeness

Allows multiple oracle calls. Alternately, allow only one call

- $f \in$ #P is #P-complete if any $g \in$ #P can be Cook-reduced to f
  - From parsimonious reduction of g's NP problem to an NP-complete problem (w.r.t Karp-reductions)
    - #SAT is #P-complete
  - Other #P-complete problems whose decision problems are in P

#P completeness

Allows multiple oracle calls. Alternately, allow only one call

- $f \in$ #P is #P-complete if any $g \in$ #P can be Cook-reduced to f
  - From parsimonious reduction of g's NP problem to an NP-complete problem (w.r.t Karp-reductions)
    - #SAT is #P-complete
  - Other #P-complete problems whose decision problems are in P
    - Permanent (for binary matrices) is #P-complete

Permanent

Permanent

- Permanent of a square matrix A

Permanent

- Permanent of a square matrix A
  - If A is binary (0,1 entries): perm(A) = number of perfect matchings in a bipartite graph B_A whose adjacency matrix is A

Permanent

- Permanent of a square matrix A
  - If A is binary (0,1 entries): perm(A) = number of perfect matchings in a bipartite graph B_A whose adjacency matrix is A
    - Note: finding if there exists a perfect matching is in P (using network flow)

Permanent

- Permanent of a square matrix A
  - If A is binary (0,1 entries): perm(A) = number of perfect matchings in a bipartite graph B_A whose adjacency matrix is A
    - Note: finding if there exists a perfect matching is in P (using network flow)
  - Algebraically: perm(A) = $\Sigma_\sigma \Pi_i A_{i,\sigma(i)}$ where σ are permutations

Permanent

- Permanent of a square matrix A
  - If A is binary (0,1 entries): perm(A) = number of perfect matchings in a bipartite graph B_A whose adjacency matrix is A
    - Note: finding if there exists a perfect matching is in P (using network flow)
  - Algebraically: perm(A) = $\Sigma_\sigma \Pi_i A_{i,\sigma(i)}$ where σ are permutations
    - Note: Similar to determinant (which is in FP)

Permanent

- Permanent of a square matrix A
  - If A is binary (0,1 entries): perm(A) = number of perfect matchings in a bipartite graph B_A whose adjacency matrix is A
    - Note: finding if there exists a perfect matching is in P (using network flow)
  - Algebraically: perm(A) = $\Sigma_\sigma \Pi_i A_{i,\sigma(i)}$ where σ are permutations
    - Note: Similar to determinant (which is in FP)
    - Permutations are cycle covers of complete directed graph

Permanent

- Permanent of a square matrix A
  - If A is binary (0,1 entries): perm(A) = number of perfect matchings in a bipartite graph B_A whose adjacency matrix is A
    - Note: finding if there exists a perfect matching is in P (using network flow)
  - Algebraically: $\text{perm}(A) = \Sigma_\sigma \Pi_i A_{i,\sigma(i)}$ where σ are permutations
    - Note: Similar to determinant (which is in FP)
    - Permutations are cycle covers of complete directed graph
      - Weight of a cycle cover σ, $W(\sigma) = \Pi_i A_{i,\sigma(i)}$

Permanent

- Permanent of a square matrix A
  - If A is binary (0,1 entries): perm(A) = number of perfect matchings in a bipartite graph B_A whose adjacency matrix is A
    - Note: finding if there exists a perfect matching is in P (using network flow)
  - Algebraically: perm(A) = $\Sigma_\sigma \Pi_i A_{i,\sigma(i)}$ where σ are permutations
    - Note: Similar to determinant (which is in FP)
    - Permutations are cycle covers of complete directed graph
      - Weight of a cycle cover σ, $W(\sigma) = \Pi_i A_{i,\sigma(i)}$
      - Perm(A) = $\Sigma_\sigma W(\sigma)$ over all cycle covers σ of directed graph G_A (with edge-weights from A)

Permanent is #P-complete

Permanent is #P-complete

- First will reduce #SAT to permanent of an integer (not binary) matrix

Permanent is #P-complete

- First will reduce #SAT to permanent of an integer (not binary) matrix
  - Plan: Given a SAT instance φ with m clauses, build an integer-weighted directed graph A_φ such that $perm(A_\varphi) = 4^{3m} . \#\varphi$

Permanent is #P-complete

- First will reduce #SAT to permanent of an integer (not binary) matrix
  - Plan: Given a SAT instance φ with m clauses, build an integer-weighted directed graph A_φ such that $perm(A_\varphi) = 4^{3m} . \#\varphi$
  - Almost Karp-reduction (need to rescale)

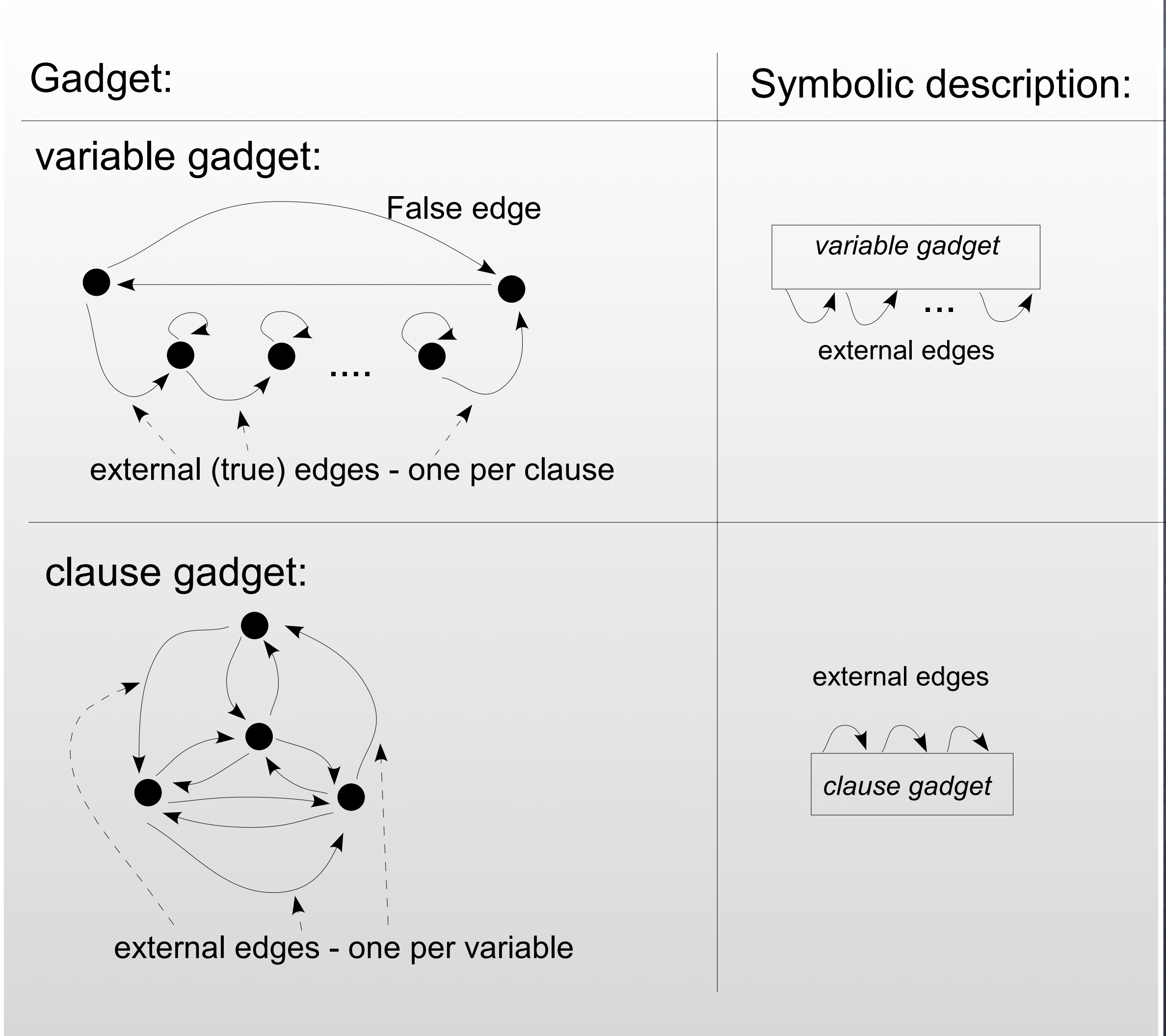
Permanent is #P-complete

Permanent is #P-complete

- For each variable add a "variable gadget" and for each clause a "clause gadget"

Permanent is #P-complete

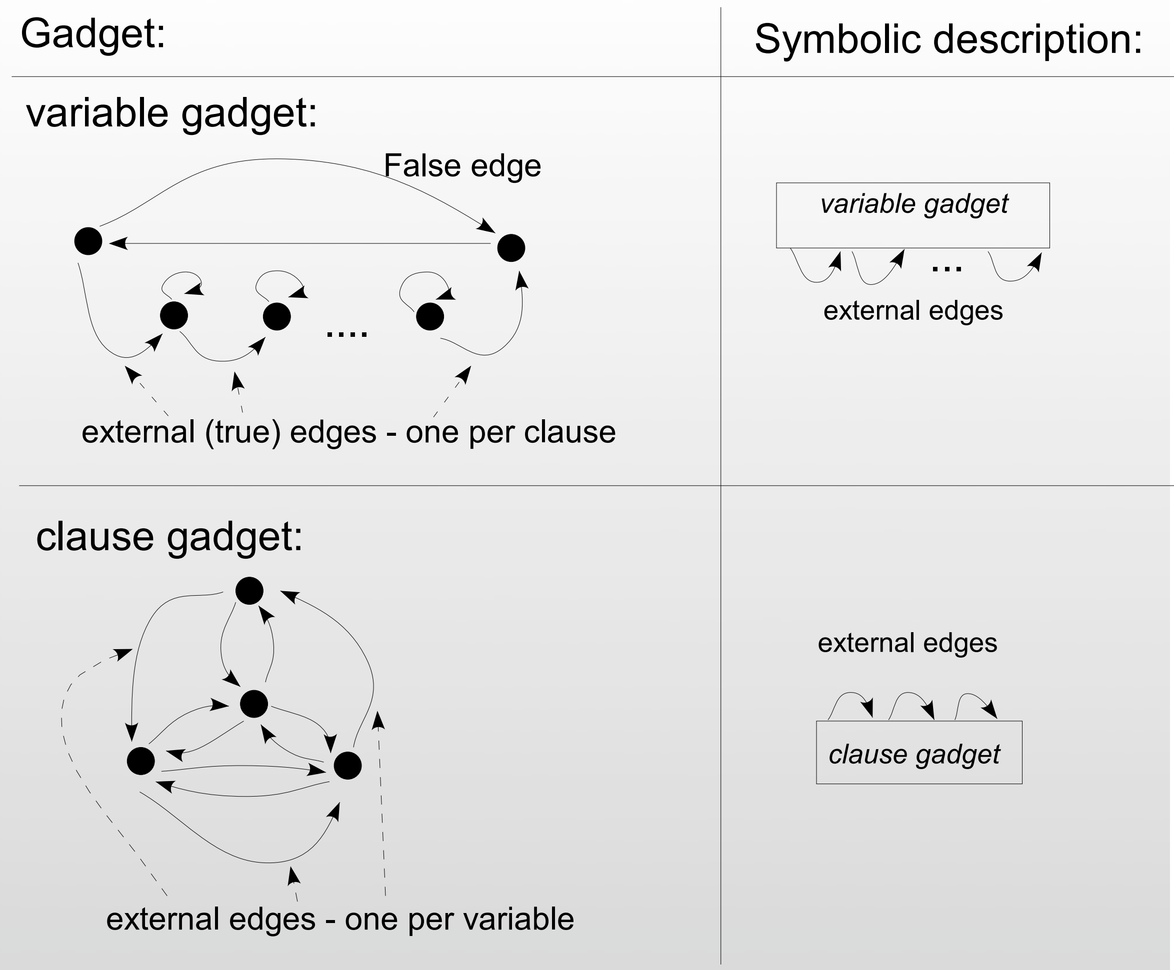
- For each variable add a "variable gadget" and for each clause a "clause gadget"



[Figures from the textbook]

Permanent is #P-complete

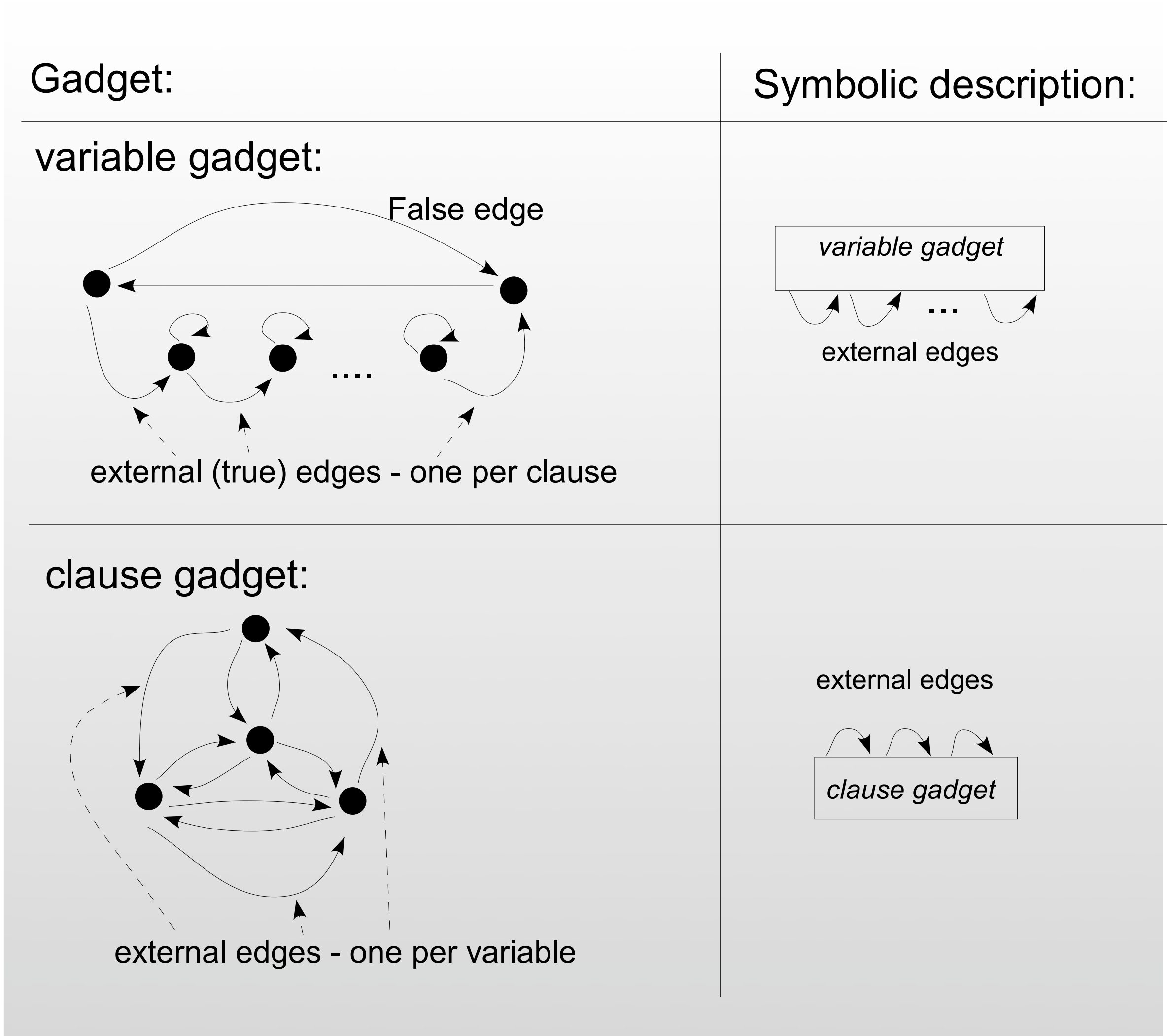
- For each variable add a "variable gadget" and for each clause a "clause gadget"
  - Variable: two possible cycle covers of weight 1 -- uses either all the true-edges or the false-edge



[Figures from the textbook]

Permanent is #P-complete

- For each variable add a "variable gadget" and for each clause a "clause gadget"
  - Variable: two possible cycle covers of weight 1 -- uses either all the true-edges or the false-edge
  - Clause: any cycle cover has to leave at least one variable-edge free



[Figures from the textbook]

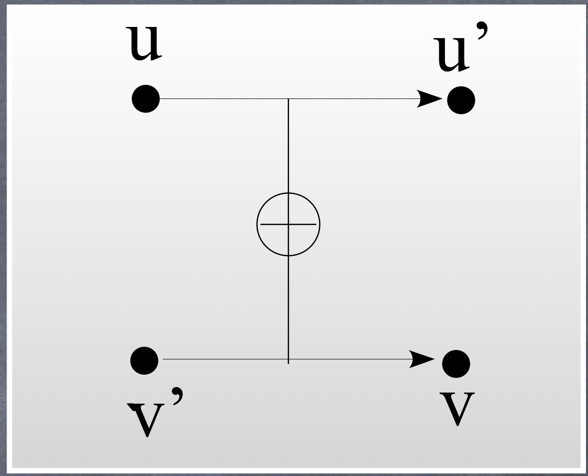
Permanent is #P-complete

Permanent is #P-complete

- XOR gadget (with negative edge weights):

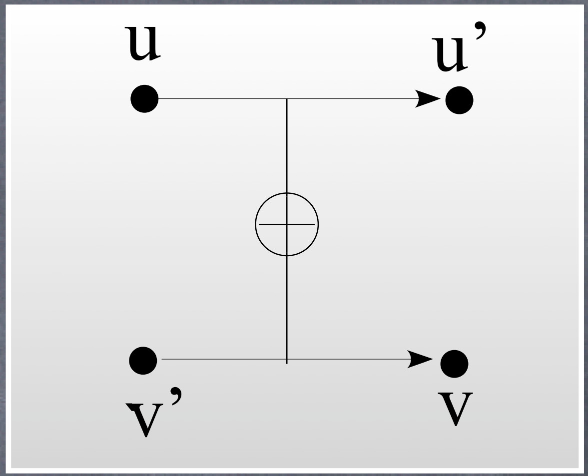
Permanent is #P-complete

- XOR gadget (with negative edge weights):



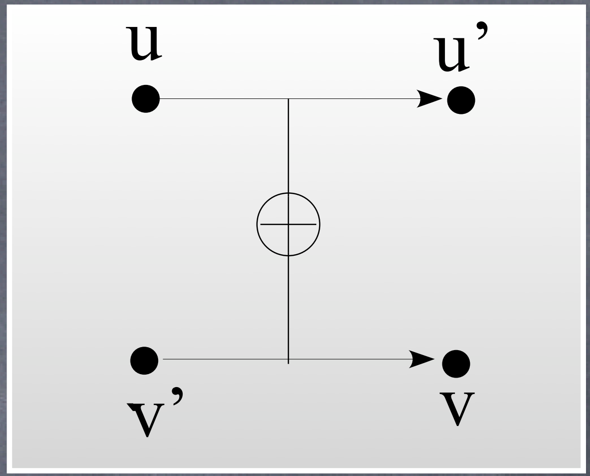
Permanent is #P-complete

- XOR gadget (with negative edge weights):
  - Replacing a pair of edges by an XOR gadget changes total weight of cycle covers using neither or both the edges to 0, and scales total weight of cycle covers using exactly one of them by 4.



Permanent is #P-complete

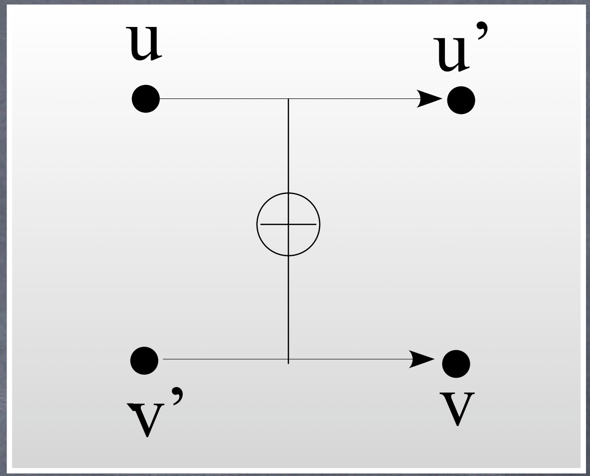
- XOR gadget (with negative edge weights):
  - Replacing a pair of edges by an XOR gadget changes total weight of cycle covers using neither or both the edges to 0, and scales total weight of cycle covers using exactly one of them by 4.



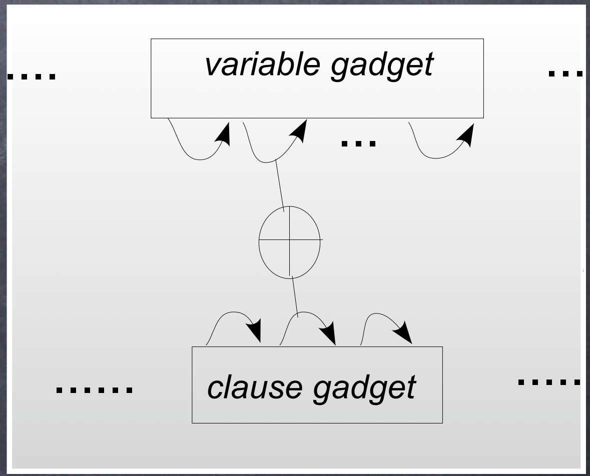
- Final graph

Permanent is #P-complete

- XOR gadget (with negative edge weights):
  - Replacing a pair of edges by an XOR gadget changes total weight of cycle covers using neither or both the edges to 0, and scales total weight of cycle covers using exactly one of them by 4.

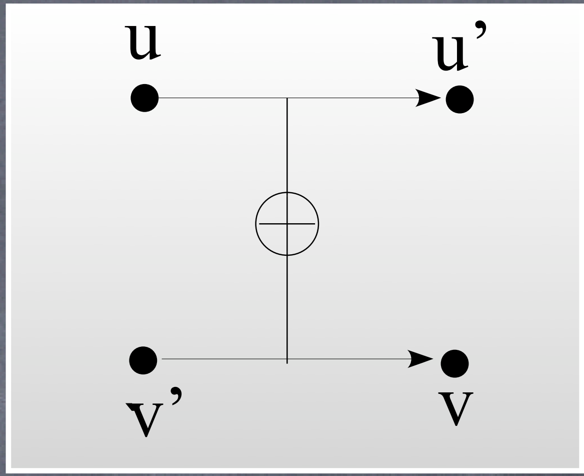


- Final graph

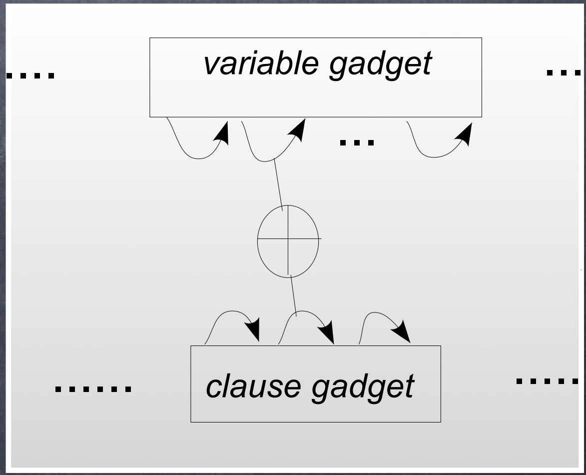


Permanent is #P-complete

- XOR gadget (with negative edge weights):
  - Replacing a pair of edges by an XOR gadget changes total weight of cycle covers using neither or both the edges to 0, and scales total weight of cycle covers using exactly one of them by 4.

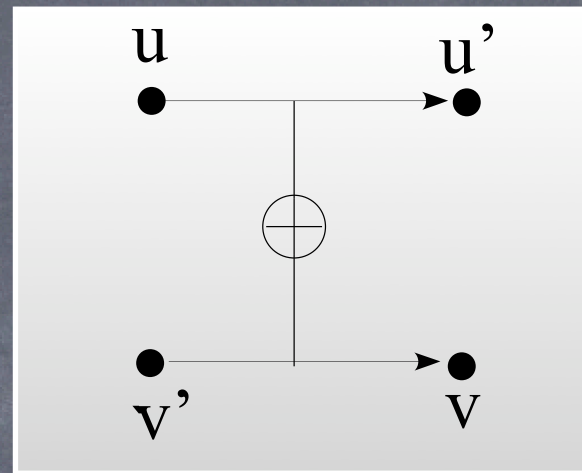


- Final graph
  - "XOR" each clause-gadget's "variable-edge" with the corresponding edge in a variable-gadget: 3m XOR gadgets

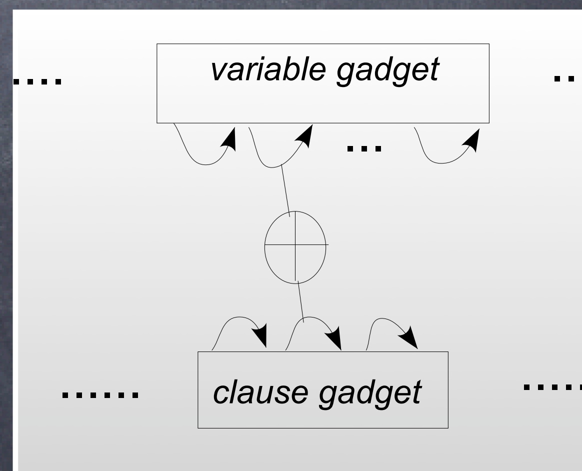


Permanent is #P-complete

- XOR gadget (with negative edge weights):
  - Replacing a pair of edges by an XOR gadget changes total weight of cycle covers using neither or both the edges to 0, and scales total weight of cycle covers using exactly one of them by 4.



- Final graph
  - "XOR" each clause-gadget's "variable-edge" with the corresponding edge in a variable-gadget: 3m XOR gadgets
  - Each satisfying assignment gives a cycle cover of weight 4^{3m}



Permanent is #P-complete

Permanent is #P-complete

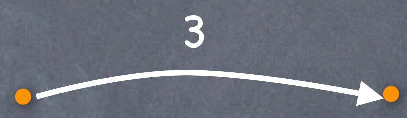
- Can use binary matrix instead of integer matrix

Permanent is #P-complete

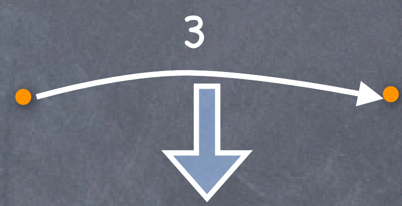
- Can use binary matrix instead of integer matrix
  - First change to +1/-1 weights (adding vertices)

Permanent is #P-complete

- Can use binary matrix instead of integer matrix
  - First change to +1/-1 weights (adding vertices)



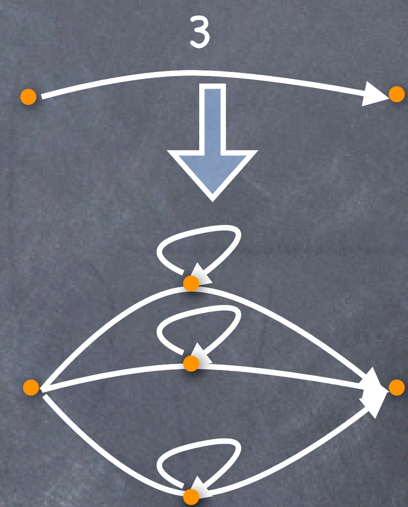
Permanent is #P-complete



- Can use binary matrix instead of integer matrix
  - First change to +1/-1 weights (adding vertices)

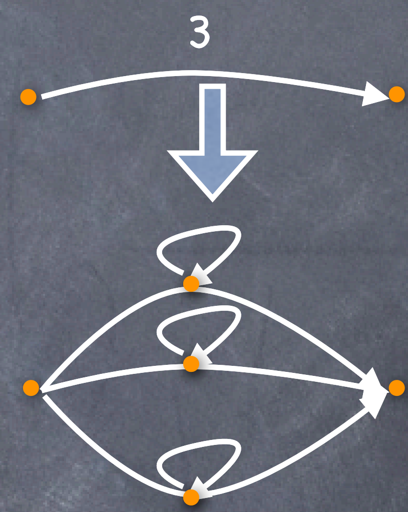
Permanent is #P-complete

- Can use binary matrix instead of integer matrix
  - First change to +1/-1 weights (adding vertices)

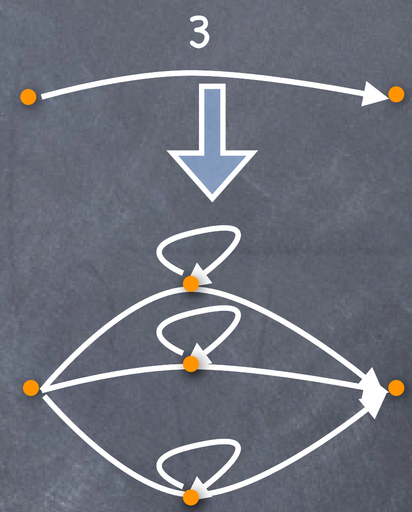


Permanent is #P-complete

- Can use binary matrix instead of integer matrix
  - First change to +1/-1 weights (adding vertices)
  - To replace -1: working modulo M+1 (for say $M=2^{n \log n} > n!$) does not change positive values. $M = 2^k$.



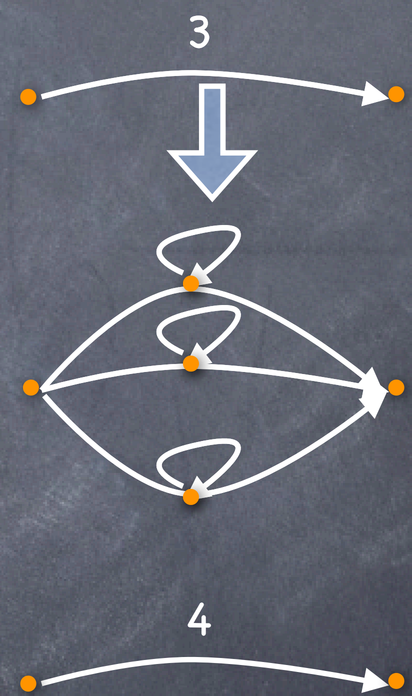
Permanent is #P-complete



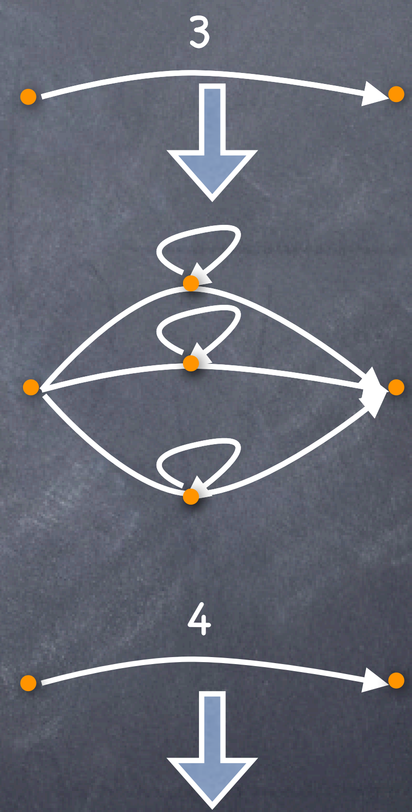
- Can use binary matrix instead of integer matrix
  - First change to +1/-1 weights (adding vertices)
  - To replace -1: working modulo M+1 (for say $M=2^{n \log n} > n!$) does not change positive values. $M = 2^k$.
    - -1 is then M. Replace M by log M edges of weight 2 in series, each further replaced by +1 weight edges

Permanent is #P-complete

- Can use binary matrix instead of integer matrix
  - First change to +1/-1 weights (adding vertices)
  - To replace -1: working modulo M+1 (for say $M=2^{n \log n} > n!$) does not change positive values. $M = 2^k$.
    - -1 is then M. Replace M by log M edges of weight 2 in series, each further replaced by +1 weight edges



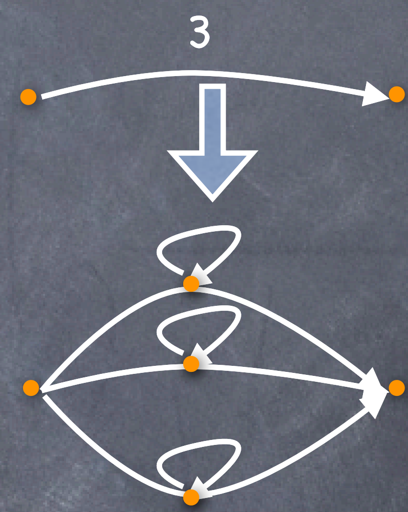
Permanent is #P-complete



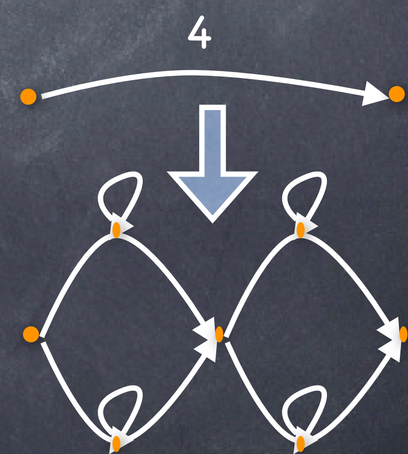
- Can use binary matrix instead of integer matrix
  - First change to +1/-1 weights (adding vertices)
  - To replace -1: working modulo M+1 (for say $M=2^{n \log n} > n!$) does not change positive values. $M = 2^k$.
    - -1 is then M. Replace M by log M edges of weight 2 in series, each further replaced by +1 weight edges

Permanent is #P-complete

- Can use binary matrix instead of integer matrix



  - First change to +1/-1 weights (adding vertices)
  - To replace -1: working modulo M+1 (for say $M=2^{n \log n} > n!$) does not change positive values. $M = 2^k$.



    - -1 is then M. Replace M by log M edges of weight 2 in series, each further replaced by +1 weight edges

Today

Today

- #P

Today

- #P
- Can be hard: even #CYCLE is not in FP (unless P = NP)

Today

- #P
- Can be hard: even #CYCLE is not in FP (unless P = NP)
- $\#P \subseteq FP^{PP}$ (and $PP \subseteq P^{\#P}$)

Today

- #P
- Can be hard: even #CYCLE is not in FP (unless P = NP)
- $\#P \subseteq FP^{PP}$ (and $PP \subseteq P^{\#P}$)
- #P complete problems

Today

- #P
- Can be hard: even #CYCLE is not in FP (unless P = NP)
- $\#P \subseteq FP^{PP}$ (and $PP \subseteq P^{\#P}$)
- #P complete problems
  - #SAT

Today

- #P
- Can be hard: even #CYCLE is not in FP (unless P = NP)
- #P $\subseteq$ FP^{PP} (and PP $\subseteq$ $P^{\#P}$)
- #P complete problems
  - #SAT
  - Permanent

Today

- #P
- Can be hard: even #CYCLE is not in FP (unless P = NP)
- $\#P \subseteq FP^{PP}$ (and $PP \subseteq P^{\#P}$)
- #P complete problems
  - #SAT
  - Permanent
- Next: Toda's Theorem: $PH \subseteq P^{\#P} = P^{PP}$