The Complexity of Counting

A Thesis presented

by

Salil Vadhan

to

Mathematics and Computer Science

in partial fulfillment of the honors requirements

for the degree of

Bachelor of Arts

Harvard College

Cambridge, Massachusetts

April 3, 1995

Acknowledgements

Before beginning, I'd like to thank a few people: My advisor, Professor Leslie Valiant, for sparking my interest in computational complexity and for providing advice and support throughout my work; Dan Roth, for suggesting these problems and guiding me through the initial stages of research; and Jennifer Sun, for reading and critiquing my incomprehensible drafts.

Contents

Chapter 1

Introduction

Over the past 30 years, computational complexity theory has provided mathematicians and computer scientists with a precise and rigorous theory of computational tractability. Satisfactory techniques to solve computational problems — such as deciding whether a graph has a Hamiltonian circuit or determining if a propositional formula is satisfiable — have eluded mathematicians for years, suggesting that there are no efficient mechanical methods to solve these problems. For a long time, however, it was not even clear how to define 'efficient mechanical method,' much less to prove that one does not exist. The study of Turing machine complexity and $\mathcal{NP}$-completeness, as introduced by Cook in 1971, finally provided a theoretical framework under which it became provable that certain problems are intractable. Since then, hundreds of computational problems, including the Boolean satisfiability and Hamiltonian circuit problems mentioned above have been demonstrated to be intractable.

The primary focus of complexity theory since Cook's paper has been on decision problems — that is, problems of determining whether an object (*e.g.* a graph) has a given property (*e.g.* has a Hamiltonian circuit). However, many problems of mathematical and practical interest are not decision problems. For a large fraction of these, computationally equivalent decision problems can be found. But it is possible that studying certain classes of problems separately might provide more insight into the nature of their difficulty than disguising them as decision problems would.

Counting problems form one class whose separate study has proven to be both natural and fruitful. For years, the primary goal of enumerative combinatorics has been been to find methods to effectively count various mathematical structures. Success in this direction has been quite limited and one might wonder whether complexity theory has an explanation for the apparent difficulty of counting. In 1979, Valiant [Val79a, Val79b] provided such evidence by introducing the class $\#\mathcal{P}$ of counting problems and proving that many natural counting problems are computationally hard, in a sense to be made precise later. The techniques used to prove intractibility in his paper were quite novel, and somewhat more difficult than those commonly used in the theory of decision problems. Perhaps this is a partial explanation of why, even after Valiant's paper, counting has not received the attention afforded to decision problems. Thus, there are many interesting and important counting problems whose complexity is not known.

This is unfortunate, for counting, aside from being mathematically interesting, is closely related to important practical problems. For instance, reliability problems are often equivalent to counting problems. Computing the probability that a graph remains connected given a probability of failure on each edge is essentially equivalent to counting the number of ways that the edges could fail without sacrificing connectivity. Another application of counting arises in Artificial Intelligence research. As explained by Roth [Rot95], the Bayesian approach to reasoning requires evaluating the probability that a formula in the propositional calculus is true, and this is tantamount to counting the number of satisfying assignments. Thus, understanding the types of propositional formulæ for which counting satisfying assignments is feasible tells us the extent to which Bayesian reasoning might be useful.

In light of these examples and others, it is clear that efforts to enhance our understanding of the complexity of counting should be increased. Our efforts should not be limited to classifying new counting problems as hard or easy, but should also include determining the restrictions under which hard counting problems either remain hard or become easy. This would parallel what has been done with the theory of $\mathcal{NP}$-completeness. A quick glance and Garey and Johnson's famous catalogue of $\mathcal{NP}$-complete problems [GJ79] reveals that the restricted-case complexity of most difficult decision problems is understood in detail.

The main contribution of this thesis is to determine the restricted-case complexity of several counting problems. Most of the original work is contained in Theorems 5.1, 5.2, and 7.1. In Theorem 5.2, we show that the problems of counting vertex covers and matchings in graphs, satisfying assignments to propositional formulæ, and variants of these remain difficult even in the most restricted cases. In the proof, we introduce a technique that appears to be useful in proving complexity results for sparse graphs. Theorem 7.1 extends one of these results, as we show that even approximately counting minimum cardinality vertex covers in sparse graphs is intractable.

In this thesis, we additionally attempt to present the current state of knowledge regarding the complexity of counting, summarizing the major relevant results. We also address the natural alternative when exact counting is known to be hard — approximate counting. We discuss the close relationship between approximate counting and uniform generation problems, and the recent machinery introduced by Jerrum and Sinclair that may provide the key to solving many of these problems. Naturally, none of these areas can be discussed without the basics of complexity theory, so that is the topic of the next chapter. A reader familiar with elementary complexity theory can skim or skip the chapter without loss of continuity.

Chapter 2

Basic Complexity Theory

The model of computation that we use is the standard multi-tape Turing machine. It is well-known that reasonable variants of the Turing machine model can simulate each other with only a polynomial time (and space) loss of efficiency. Since we will only be concerned with complexity up to polynomial factors, the specifics of our Turing machine definition will not be crucial. For completeness, however, we offer an informal definition.

Definition 2.1 *A* **k-tape Turing machine (TM)** *has an input/output tape, k one-way infinite tapes, $k+1$ independent read-write heads (one for each tape), and a finite-state control. Each tape is divided into squares, each containing symbols from a finite alphabet Σ, which contains a distinguished blank symbol $\#$. At any time, the TM can be in one of finitely many states, including a halt state h. There is a transition function that, given the current TM state and the symbols under each of the read-write heads, determines the next state of the TM and whether each of the heads moves left, right, or writes a symbol. Once in the halt state, the heads remain in place, and the state never changes.*

At this point, one typically defines what it means for a Turing machine to compute a function and decide a language. However, our discussion of complexity theory will not be limited to these things, but will also include problems of approximation and uniform generation. Therefore, we proceed by defining *search problems*, which are very general and include all of the above as special cases. Our approach is a modification of that taken in [GJ79].

Definition 2.2 *Let Σ be a finite alphabet. A* **search problem** *on Σ is a map $\Pi : \Sigma^* \to F(\Sigma^*)$, where $F(\Sigma^*)$ denotes the set of all finite subsets of Σ^*, i.e. all finite languages over the alphabet Σ. We call each string x in the domain an* **instance** *of the problem, and $\Pi(x)$ the set of* **solutions** *for x.*

Let us observe how search problems encorporate some of the types of problems discussed earlier. Given any language $L \subset \Sigma^*$, the search problem Π_L corresponding to **recognizing** or **deciding** L is:

$$\Pi_L(x) = \begin{cases} \{Y\} & \text{if } x \in L \\ \emptyset & \text{otherwise} \end{cases}$$

In general, we call Π a **function problem** iff $|\Pi(x)| \leq 1$ for all x. Thus, language recognition is a kind of function problem, with the function being the 'characteristic function' of the language L. $\mathcal{D}$ denotes the class of all search problems that can be realized as the decision problem for some language, and $\mathcal{F}$ is the class of all function problems.

We now need to define what it means for a Turing machine to solve such a search problem.

Definition 2.3 *A TM M* **solves search problem Π** *iff whenever started with $\#x\#$ on the input/output tape, all other tapes blank, and all heads at the leftmost squares, M ends up in the halt state with $\#y\#$ on the input/output tape and all heads at the leftmost squares, where y is some element of $\Pi(x)$ if $\Pi(x)$ is nonempty and is the singleton string N otherwise. Here M operates on alphabet Σ' and Π is a search problem on alphabet Σ, where $\Sigma' = \Sigma \cup \{N\} \cup \{\#\}$ and $x, y \in \Sigma^*$.*

We are now prepared to define the time-complexity of a TM.

Definition 2.4 *A TM M is said to* **work in time $T : \mathbb{N} \to \mathbb{N}$** *iff for all inputs x of length n the Turing machine halts in $\leq T(n)$ steps (counting each transition as a single step) when started in the configuration described above.*

Definition 2.5 *A search problem Π is said to be* **solvable in time $T : \mathbb{N} \to \mathbb{N}$** *iff there is a TM M that solves Π and works in time T.*

We denote by $\mathcal{SP}$ the class of all search problems that can be solved in time p for some polynomial p. Some better-known classes are the class $\mathcal{P}$

of decision problems that can be solved in polynomial time, which is simply $\mathcal{SP} \cap \mathcal{D}$, and the class $\mathcal{FP}$ of function problems which can be solved in polynomial time, which is just $\mathcal{SP} \cap \mathcal{F}$. By themselves, the three classes $\mathcal{SP}$, $\mathcal{P}$ and $\mathcal{FP}$ are not very interesting from the perspective of early complexity theory. After all, one of our main goals is to prove that certain problems require exponential time to solve, and we are unlikely to accomplish this if we do not look beyond $\mathcal{P}$. One of most important contributions of Edmonds and Cook was determining which class of problems beyond $\mathcal{P}$ would be fruitful to study. They realized that the class which is now called $\mathcal{NP}$ contains most of the decision problems of computational interest[GJ79, page 118]. We shall define $\mathcal{NP}$ in terms in of the p-relations introduced in [JVV86].

Definition 2.6 *Let Σ be a finite alphabet. A relation $R \subset \Sigma^* \times \Sigma^*$ is said to be a* ***p*-relation** *iff the following conditions hold:*
1. it is **polynomially-balanced***, i.e. there exists a polynomial p such that $\langle x, y\rangle \in R \Rightarrow |y| \leq p(|x|)$*
2. it can be 'checked quickly,' i.e. the language $L = \{x\$y : \langle x, y\rangle \in R\} \in \mathcal{P}$, where \$ is a symbol not in Σ.
We also write $R(x) = \{y \in \Sigma^* : \langle x, y\rangle \in R\}$.
As with search problems, we call x an **instance** *and $R(x)$ the set of* **solutions** *for x.*

It turns out that most of the search problems we want to study can be formulated in terms of p-relations. For example, if we have encodings in which every string in Σ^* represents a boolean formula over some fixed countably-infinite set of propositional variables and every string in Σ^* also represents an assignment to a finite subset of these variables, we can define a p-relation R by saying $\langle x, y\rangle \in R$ iff y encodes a satisfying assignment to the variables in the formula encoded by x. In any reasonable encoding, we would certainly have $|y| \leq p(|x|)$ for some linear polynomial p because the number of variables appearing in x can be no greater than the length of x. And the relation can be checked quickly by recursively evaluating the formula x at the assignment y.

The search problems that we are most directly concerned with regarding a boolean formula are 1. Is it satisfiable? (decision), 2. Produce a satisfying assignment (construction), and 3. Count the number of satisfying assignments (counting). We can define these problems in terms of p-relations quite easily.

Definition 2.7 *Let R be a p-relation over alphabet Σ.*
The **construction problem** $\boldsymbol{C}_R$ *associated with R is the search problem defined by $C_R(x) = R(x)$.*
The **decision problem** $\boldsymbol{D}_R$ *associated with R is defined by*

$$D_R(x) = \begin{cases} \{Y\} & \text{if } R(x) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

Equivalently, this is the recognition problem for the language $L_R = \{x : R(x) \neq \emptyset\}$.
The **counting problem** $\#\boldsymbol{R}$ *associated with R is the function problem defined by $\#R(x) = \{$a binary encoding of $|R(x)|\}$*

The class $\mathcal{NP}$ consists of all decision problems associated with p-relations and $\#\mathcal{P}$ consists of all counting problems associated with p-relations.[1] $\mathcal{NP}$ and $\#\mathcal{P}$ respectively contain hundreds of decision and counting problems of interest, including many of the ones for which subexponential algorithms have eluded researchers. With this in mind, the natural questions to ask are whether $\mathcal{P} = \mathcal{NP}$ and whether $\#\mathcal{P} \subset \mathcal{FP}$. However, we first ought to verify that $\mathcal{P} \subset \mathcal{NP}$. This relationship is clear, for given any language (decision problem) L in $\mathcal{P}$, we can define $R_L = \{\langle x, e\rangle : x \in L\}$, where e is the empty string. Then R_L is a p-relation, as we can decide the language $\{x\$e : x \in L\}$ in polynomial time by erasing the e and the \$, and running the polynomial time algorithm that decides L.

To a reader unfamiliar with complexity theory, our informal use of the word 'algorithm' in place of 'Turing machine program' might be a bit disturbing. Our foundation for this is the 'Church-Turing thesis.' This states that Turing machines are exactly equivalent to our informal notion of algorithms — that is, that Turing machines can simulate any reasonable algorithms we can think of and that every Turing machine program can be thought of as an algorithm. By 'reasonable,' we mean that the amount of resources required at every step is finite and bounded. However, if the Church-Turing thesis is to justify the use of Turing machines for the study of computational complexity, it needs to say something more. In particular, we need to believe that Turing machines can simulate any reasonable algorithm with only a moderate (*i.e.* polynomial) loss of efficiency. This idea,

[1] Cook defined $\mathcal{NP}$ in terms of 'nondeterministic Turing machines'. It is not difficult to show that our formulation is equivalent. $\#\mathcal{P}$ was originally defined by Valiant in terms of 'counting Turing machines', and, again, the equivalence with our definition is straightforward.

known as the 'strong version of the Church-Turing thesis,' is widely accepted. None of the physical computation devices currently in existence, including the most powerful of supercomputers, violate this assumption of complexity theory. Therefore, we are justified in informally describing Turing machine programs as algorithms, with the implicit assumption that time-complexity is preserved, modulo polynomial factors.

As mentioned earlier, it is widely believed that many $\mathcal{NP}$ problems, such as Boolean satisfiability and Hamiltonian circuit detection have no polynomial-time algorithms, and hence $\mathcal{P} \neq \mathcal{NP}$. *A priori,* a proof that $\mathcal{P} \neq \mathcal{NP}$ would not necessarily identify any of these difficult problems as lying outside of $\mathcal{P}$. What we need is a way of identifying the *hardest* problems in $\mathcal{NP}$, *i.e.* problems that could not be solved in polynomial time unless everything in $\mathcal{NP}$ could. Turning this statement around, we can identify a problem as one of the 'hardest' in $\mathcal{NP}$ if a polynomial-time algorithm that solves it can be used to solve *any* $\mathcal{NP}$ problem in polynomial time. More generally: to prove that problem A is at least has hard as problem B, we show that an imaginary algorithm to solve A could be used to solve B with only a polynomial loss of efficiency. Thus, if a lower bound on the time complexity of problem B can be found, then the same lower bound will hold for A, modulo polynomial factors. We formalize this idea by defining oracle Turing machines, using a modification of the definition in [GJ79].

Definition 2.8 *An* **oracle Turing machine (OTM)** *with oracle g is an ordinary Turing machine over alphabet Σ with the following modifications:*
1. There is an oracle function $g : \Sigma^* \rightarrow \Sigma^*$.
2. M has a special oracle tape.
3. M has two additional states: q, the oracle-query state, and r, the oracle-response state.
4. When M enters state q, the action depends on the contents of the oracle tape and on the oracle function g. Let $y \in \Sigma^*$ be the string appearing in squares 1 through $|y|$ of the oracle tape, where square $|y| + 1$ is the first square to the right of square 0 that contains a blank, and let $z \in \Sigma^*$ be the value of $g(y)$. Then, in one step, the oracle tape is changed to contain the string $|z|$ in squares 1 through $|z|$ (with blanks everywhere else), the oracle head is set to scan square 1, and the finite state control is changed from state q to state r. Such a step leaves both the contents of the other tapes and the position of the other tape heads unchanged.[2]

[2] Part 4 of this definition is taken almost verbatim from [GJ79]

We can now define the time that an OTM works in and what it means for an OTM to solve a search problem. These definitions are analogous to the ones for standard TM's, with each oracle call counting as one step.

Definition 2.9 *Let* Π *be a search problem. A search problem* Γ *is said to be* **solvable in time T by OTM's with Π-oracles** *iff there is an OTM program* M *that solves* Γ *in time* T *when equipped with any oracle function* g *that 'solves'* Π, *i.e.* $g(x) \in \Pi(x)$ *for all* x.

Note that in the above definition, we don't consider g to be part of the OTM 'program.' Additionally, if Π is not a function problem, there may be several possible oracle functions that solve Π. We require that the *same* program work with the *same* time complexity for all possible choices of the oracle function. Thus oracle machines allow us to pretend that we have an efficient (1 step!) solution to Π and see what problems can be solved quickly in this relativized world. Those that can are considered to be at least as easy as Π.[3]

For each of the classes of search problems defined earlier, we define analogous relativized classes: $\mathcal{SP}^{\Pi}$, $\mathcal{P}^{\Pi}$, $\mathcal{FP}^{\Pi}$, $\mathcal{NP}^{\Pi}$, and $\#\mathcal{P}^{\Pi}$. The definitions are identical to those for the non-relativized classes, except that "solvable in time T" should be replaced with "solvable in time T by OTM's with Π-oracles". This modification also applies, of course, to the statement that a p-relation be polynomially-checkable. Finally, we are able to formalize what we mean when we say that one problem is more difficult than another.

Definition 2.10 *Search problem* Π *is said to be* **polynomial-time Turing-reducible** *to search problem* Γ *iff* $\Pi \in \mathcal{SP}^{\Gamma}$. *That is, there is an OTM program that solves* Π *in polynomial time when equipped with oracles that solve* Γ. *Such a program is called a* **polynomial-time Turing reduction** *from* Π *to* Γ. *In this case, we write* $\Pi \propto \Gamma$.

Given our motivation that $\Pi \propto \Gamma$ should mean that Γ is at least as hard as Π, we ought to verify that $\propto$ is in fact transitive.

Proposition 2.11 $\propto$ *is reflexive and transitive when restricted to the search problems* Π *for which there is a polynomial* Q_{Π} *such that* $y \in \Pi(x) \Rightarrow |y| \leq Q_{\Pi}(|x|)$.[4]

[3] Please pardon the pun.

[4] This condition on Π is identical to the polynomially-balanced condition we placed on p-relations. In this case, it's crucial because a problem which requires exponentially

Proof Reflexivity is straightforward: given an oracle for Π, we can solve Π in time $O(n + Q_{\Pi}(n))$ by copying the input to the oracle tape (time $O(n)$), calling the oracle, and copying the oracle's output to the input/output tape (time $O(Q_{\Pi}(n))$).
Transitivity is also easy. Suppose $\Pi_1 \propto \Pi_2 \propto \Pi_3$. Then, given an oracle for Π_3, we could solve Π_1 by using the same algorithm used to solve Π_1 with Π_2-oracles, except that every time we need to call the Π_2-oracle, we instead simulate the polynomial time Π_3-oracle algorithm that solves Π_2. Observe that if $p(n)$ is the polynomial-time bound on the Π_2-oracle algorithm for Π_1, then the lengths of the inputs to the Π_2-oracle and the number of times the Π_2-oracle is called are all bounded by $p(n)$, where n is the length of the input (because writing a string of length k takes at least k steps and each oracle call takes at least 1 step). So, if $q(n)$ is the polynomial time bound on the Π_3-algorithm for Π_2, the simulation described above to solve Π_1 with a Π_3-oracle takes time $O(p(n) + p(n)q(p(n)))$. $\Box$

Recall that we hoped to classify certain problems as being the hardest in $\mathcal{NP}$. So far, we've only defined the difficulty of problems relative to other problems. Defining the difficulty of a problem relative to a class of problems is only one step away.

Definition 2.12 *If $\mathcal{C}$ is any class of search problems, then a search problem Π is said to be **$\mathcal{C}$-hard** iff for every problem $\Gamma \in \mathcal{C}$, $\Gamma \propto \Pi$. If, in addition, Π is also in $\mathcal{C}$, Π is said to be **$\mathcal{C}$-complete**.*

Definition 2.13 *Conversely, we say that Π is **$\mathcal{C}$-easy** iff there exists $\Gamma \in \mathcal{C}$ such that $\Pi \propto \Gamma$. If Π is both $\mathcal{C}$-easy and $\mathcal{C}$-hard, then Π is said to be **$\mathcal{C}$-equivalent**.*
(N.B. if $\Pi \in \mathcal{C}$ then Π is $\mathcal{C}$-easy. In particular, if Π is $\mathcal{C}$-complete then Π is $\mathcal{C}$-equivalent.)

Complete problems are at the heart of complexity theory. A complete problem defines the complexity of the entire class it is complete for. That is, the problem is exactly as easy or as hard as the class for which it is complete. We need an additional definition before we can formalize this. Following [Pap94, page 166]:

long outputs cannot be solved in polynomial time in any relatived world, because it takes exponential time just to write down the answer! All the search problems that we shall consider will trivially satisfy this condition, so we shall not mention it again.

Definition 2.14 *A class $\mathcal{C}'$ of complexity problems is said to be* **closed under (polynomial-time Turing) reductions** *iff $L \propto L'$ and $L' \in \mathcal{C}'$ implies that $L \in \mathcal{C}'$.*

Informally, this says that $\mathcal{C}'$ is robust with respect to polynomial time reductions. A crucial feature of the type of reduction that we use and the classes we consider is that this closure property holds.

Proposition 2.15 *$\mathcal{SP}$ is closed under reductions. $\mathcal{FP}$ and $\mathcal{P}$ are closed under reductions if we restrict the definition to function problems or decision problems, respectively.*

Proof We prove it for $\mathcal{SP}$ and it follows immediately for $\mathcal{FP}$ and $\mathcal{P}$. Say $\Pi \propto \Gamma$ and $\Gamma \in \mathcal{SP}$. Suppose Γ were solvable in polynomial-time $q(n)$ and the Γ-oracle algorithm for Π takes time $p(n)$. Then the oracle algorithm makes at most $p(n)$ calls to the oracle, and each time, the input to the oracle is of length at most $p(n)$. Hence, we can give a polynomial-time algorithm for Π by simulating the Γ-oracle algorithm, except each time an oracle call is needed, we simulate the polynomial time algorithm for Γ. This entire simulation takes time $O(p(n) + p(n)q(p(n)))$. $\Box$

The following proposition demonstrates the importance of complete problems.

Proposition 2.16 *Let Π be a search problem, $\mathcal{C}$ be any class of search problems, and $\mathcal{C}'$ be a class of search problems closed under reductions.*
1. If Π is $\mathcal{C}$-easy and $\mathcal{C} \subset \mathcal{C}'$, then $\Pi \in \mathcal{C}'$.
2. If Π is $\mathcal{C}$-hard and $\Pi \in \mathcal{C}'$, then $\mathcal{C} \subset \mathcal{C}'$.
3. If Π is $\mathcal{C}$-equivalent (or $\mathcal{C}$-complete) then $\Pi \in \mathcal{C}' \Leftrightarrow \mathcal{C} \subset \mathcal{C}'$.
4. If both Π and Γ are either complete or equivalent for $\mathcal{C}$ then $\Pi \in \mathcal{C}' \Leftrightarrow \Gamma \in \mathcal{C}'$.

Proof All follow immediately from the definitions. $\Box$

As a consequence of these two propositions, problems complete or equivalent for a class like $\mathcal{NP}$ or $\#\mathcal{P}$ can be solved in polynomial time if and only if every problem in $\mathcal{NP}$ or $\#\mathcal{P}$ can. As no polynomial-time algorithms have been found for any of the many $\mathcal{NP}$-complete problems, the conjecture that $\mathcal{P} \neq \mathcal{NP}$ is quite plausible. If we take $\mathcal{P} \neq \mathcal{NP}$ as an assumption, then identifying a problem as $\mathcal{NP}$-hard is proof that the problem is intractable.

Thus, we would like an easy technique for proving that a problem Π is $\mathcal{C}$-hard for a class $\mathcal{C}$. Surely we do not expect to give an explicit reduction from every problem in $\mathcal{NP}$ to Π every time we want to prove that a problem Π is $\mathcal{NP}$-hard. No — the following fact proves to be most useful.

Proposition 2.17 *If Π is $\mathcal{C}$-hard, $\Pi \propto \Gamma$, then Γ is $\mathcal{C}$-hard.*

Proof This is an immediate consequence of the transitivity of $\propto$. $\Box$

So, to start, we need to give only one direct proof of hardness. Then, we can identify other hard problems via reduction from known hard problems. Cook's Theorem provided this starting point for $\mathcal{NP}$.

Theorem 2.18 (Cook) SAT, *the problem of deciding whether a formula in the propositional calculus is satisfiable, is $\mathcal{NP}$-complete.*

Proof(sketch) Observe that SAT is simply the decision problem associated with the p-relation for propositional formulæ described earlier. So SAT$\in \mathcal{NP}$. We sketch the proof that SAT is $\mathcal{NP}$-complete, as it is quite long if written out in detail and it can be found in all elementary texts on the theory of computation. If L_R is an arbitrary language in $\mathcal{NP}$ (with p-relation R), we have a TM M that decides $\{x\$y : \langle x, y\rangle \in R\}$ in polynomial time $p(n)$. Say M has r states, has t tapes and is over an alphabet of size v. To reduce L_R to SAT, given input x, we construct a propositional formula on variables $Q_{i,k}$, $H_{i,j,p}$, $S_{i,j,m,p}$ where $0 \leq i, j \leq p(n)$; $1 \leq k \leq r$; $1 \leq m \leq v$; and $0 \leq t \leq p$. $Q_{i,k}$ is intended to mean that M is in state k at time i, $H_{i,j,p}$ is intended to mean that the head for tape p is over square j at time i, and $S_{i,j,m,p}$ is intended to mean that square j on tape p contains symbol m at time i. We can write a formula which enforces the rules for Turing machine transitions; guarantees that the input/output tape looks like $\#x\$y\#$ for some string y at time 0; and requires that the output tape contains $\#Y\#$ and that M is in the halt state at time $p(n)$. This formula will be satisfiable iff there is some way to choose the string y such that $\langle x, y\rangle \in R$, *i.e.* iff $x \in L_R$. The construction requires only polynomial time. (We can bound tape-square parameters by $p(n)$ because in time $p(n)$, a TM read/write head cannot move more than $p(n)$ squares from its starting point.) $\Box$

Cook also showed that 3SAT, the more restricted problem of determining whether a formula in 3-CNF (defined below) is satisifiable is also $\mathcal{NP}$-complete.

Definition 2.19 *A propositional formula F is said to be in* **conjunctive normal form (CNF)** *iff it is written as the conjunction ($\wedge$) of a number of clauses, each of which contains the disjunction ($\vee$) of literals (which are variables or their negations). F is said to be in* **k-CNF** *iff each clause contains exactly k literals.* **Disjunctive normal form (DNF)** *and* **k-DNF** *are defined analogously, with the conjunctions and disjunctions switched.*

The $\mathcal{NP}$-completeness of 3SAT is proved via reduction from SAT . For any propositional formula F, one can produce, by introducing new variables, a formula F' in 3-CNF such that F' is satisfiable iff F is. This, too, is a standard theorem from elementary complexity theory and the specifics are omitted. For detailed proofs of both this result and Cook's theorem, see [GJ79] or [LP81].

With 3SAT as a starting point, a wealth of problems have been shown to be $\mathcal{NP}$-complete. The situation with counting is similar, but not quite as good. A parallel approach can be taken to find a 'first' $\#\mathcal{P}$-complete problem, but, after that, progress has been a bit more difficult. Unlike the hundreds of problems that are known to be $\mathcal{NP}$-complete, the list of $\#\mathcal{P}$-complete problems is only in the dozens, and for most of these, the restricted cases are not well-understood. Counting, of course, is our main topic, and so we shall proceed to it without further delay.

Chapter 3

The Complexity of Counting

For a few years after Cook's theorem, the main focus of complexity was on decision problems, and it appeared that the study of decision problems would tell us all we want to know about computational complexity. After all, the construction and counting problems for any p-relation are at least as hard as the corresponding decision problem. Formally,

Proposition 3.1 *If R is any p-relation, $D_R \propto C_R$ and $D_R \propto \#R$.*

Proof This is clear: given x, we would answer N to the decision problem if a construction oracle answered N and answer Y if the construction oracle produced a solution in $R(x)$. Similarly, we would answer N to the decision problem iff a counting oracle gave an answer of 0 solutions. $\square$

This fact tells us that the construction and counting analogues of $\mathcal{NP}$-complete problems are $\mathcal{NP}$-hard, so it is not clear that we need to study counting problems separately from their decision counterparts. However, there are many p-relations whose decision problems are solvable in polynomial time, but whose counting problems appear to be quite hard. In addition, the type of reduction commonly used among decision problems is not the general Turing reduction described in Chapter 2. Instead, **Karp reductions** are used, and these don't generalize nicely to counting problems.

Definition 3.2 *Language L_1 is* **Karp-reducible** *to L_2 iff there is a polynomial time computable function f such that $x \in L_1$ iff $f(x) \in L_2$*

This is a stronger form of reduction than polynomial-time Turing reductions, for if L_1 is Karp-reducible to L_2, then it is Turing-reducible to

L_2, but the converse is unknown. The most natural generalizations of Karp reductions to counting problems are parsimonious reductions.

Definition 3.3 *A polynomial-time computable function f is said to be a* **parsimonious reduction** *from p-relation R_1 to p-relation R_2 iff $|R_2(f(x))| = |R_1(x)|$ for all x.*

It is clear that a parsimonious reduction is also a Karp reduction from the decision problem for R_1 to the decision problem for R_2. Moreover, such a parsimonious reduction also yields a Turing reduction from the counting problem for R_1 to the counting problem for R_2 as the number of solutions is preserved. Given the initial focus on decision problems, parsimonious reductions might seem like the natural way to reduce counting problems to each other. However, it is easy to construct counting problems which are inter-reducible via polynomial-time Turing reductions but are not parsimoniously reducible to each other. In contrast, no one has been able to demonstrate that there are decision problems which are $\mathcal{NP}$-hard via Turing reductions, but not via Karp reductions, and for most $\mathcal{NP}$-complete problems, Karp reductions are known. This suggests that counting problems might be of a different nature than decision problems and ought to be studied separately.

Having come to the conclusion that counting problems deserve independent study, we are faced with the same questions that we had when we first studied the complexity of decision problems. Namely, we need to determine which class of counting problems is worthy of study and identify an initial complete problem for that class. In this case, the answers are parallel those we had for decision problems. Again, the class of counting problems that interest us are those associated with p-relations, *i.e.* the class $\#\mathcal{P}$. And, our first $\#\mathcal{P}$-complete problem, as with $\mathcal{NP}$-completeness, comes from Boolean satisfiability.

Theorem 3.4 #SAT *and* #3SAT*, the problems of counting the number of satisfying assignments of arbitrary propositional formulæ and of those in 3-CNF, respectively, are $\#\mathcal{P}$-complete*

Proof To see that both of these problems are in $\#\mathcal{P}$, we simply note that they are the counting problems associated with the same p-relations that we mentioned for the SAT and 3SAT. In the future, when the corresponding p-relation is obvious, we shall omit proofs that problems are in $\#\mathcal{P}$. To prove that they are $\#\mathcal{P}$-hard, one need only note that the Karp reductions used to prove $\mathcal{NP}$-completeness for each of these are in fact parsimonious. $\Box$

These results are trivial, given what we already knew from our theory of decision problems. Other proofs of $\#\mathcal{P}$-completeness for counting analogues of $\mathcal{NP}$-complete problems can be found by modifying the original Karp reductions to be parsimonious. This process, however, only tells us that the counting analogues of hard decision problems are also hard, which we already knew by Proposition 3.1. We want to prove that counting is hard even in cases where decision is easy. One simple example is counting the number of satisfying assignments to a formula in disjunctive normal form.

Proposition 3.5 #DNF, *the problem of counting the number of satisfying assignments to a formula in disjunctive normal form, is* $\#\mathcal{P}$*-complete, whereas* DNF*, the problem of deciding if a DNF formula is satisfiable, is in* $\mathcal{P}$.

Proof We (Turing) reduce from #3SAT. Given F in 3-CNF, we can put $\neg F$ into DNF in polynomial time using DeMorgan's Laws (simply switch all $\vee$'s and $\wedge$'s and negate all literals). Then $|F| = 2^n - |\neg F|$, where $|G|$ denotes the number of satisfying assignments to a formula G and n is the number of variables in F. So, with a single call to the #DNF oracle, we can solve #3SAT. To see that DNF is in $\mathcal{P}$, simply note that a DNF formula is satisfiable iff there is some clause in which no variable has both negated and nonnegated occurrences. This can be checked in polynomial time. $\Box$

One can construct analogous examples by similar trivial modifications. The first nontrivial proof of $\#\mathcal{P}$-completeness was given by Valiant when he showed that computing the permanent is hard.

Definition 3.6 *If* $A = (a_{i,j})$ *is an* $n \times n$ *matrix, the* **permanent of *A*** *is*

$$perm(A) = \sum_{\sigma \in S_n} a_{1,\sigma(1)} a_{2,\sigma(2)} \cdots a_{n,\sigma(n)}$$

(S_n *is the group of permutations of* $\{1, \ldots, n\}$*.)*

Superficially, the permanent looks quite like the determinant, as its formula is identical except for factors of ± 1 in the products. But, whereas the determinant can be computed quickly using Gaussian elimination, no one has found an efficient algorithm to compute the permanent. The permanent lends itself well to complexity analysis as computing it corresponds to a counting problem associated with a p-relation.

Definition 3.7 *If $G = (V, E)$ is an undirected graph, a* **matching** *in G is a set of edges $M \subset E$ such that no two edges in M share an endpoint.*

Definition 3.8 *If $G = (X \cup Y, E)$ is a bipartite graph with $|X| = |Y| = n$, a* **perfect matching** *in G is a matching M containing n edges.*

Proposition 3.9 *Let $G = (X \cup Y, E)$ be a bipartite graph with $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_n\}$, and let $A = (\, a_{i,j} \,)$ be the adjacency matrix of G, i.e. $a_{i,j} = 1$ if $(x_i, y_j) \in E$ and $a_{i,j} = 0$ if $(x_i, y_j) \notin E$. Then perm(A) = the number of perfect matchings in G.*

Corollary 3.10 *The problem of computing the permanent for 0-1 $n \times n$ matrices is in $\#\mathcal{P}$.*

Proof Each term in the sum of the permanent is either 0 or 1. The value of the permanent is the number of terms that are 1. Each term that is 1 corresponds to a unique set of edges $(x_1, y_{\sigma(1)}), \ldots, (x_n, y_{\sigma(n)}) \in E$ which form a perfect matching because σ is a permutation. And every perfect matching in G corresponds to a term that is 1 (namely, given perfect matching M, let $\sigma(i) =$ the unique j such that $(x_i, y_j) \in M$.)

The corollary follows because every $0-1$ matrix is the adjacency matrix for a bipartite graph and it is easily checked whether a given set of edges does in fact form a perfect matching. □

The following result explains why the search for an algorithm to compute the permanent had not been fruitful.

Theorem 3.11 (Valiant, 1979) *Computing the permanent, i.e $\#$*Perfect Matchings *is $\#\mathcal{P}$-complete.*

Proof Valiant's reduction is quite involved. For a proof, the reader is referred to [Val79a] or [Pap94, Ch. 18]. □

Valiant's hardness result was significant in that Permanent is not closely related to any $\mathcal{NP}$-complete problem. In fact, determining whether a graph has a perfect matching and finding perfect matchings can both be done in polynomial time. These algorithms represent some of the major successes of computer science, and one can find them in any good textbook on algorithms. Using Valiant's result to prove that other counting problems are hard was not, however, a trivial task. In the same year, he introduced several novel reduction techniques, while proving a number of other problems to be $\#\mathcal{P}$-complete. Continuing this work, Provan & Ball [PB83], Dyer & Frieze [DF89], Roth [Rot95], Goldberg [Gol93], and others have added to the list of $\#\mathcal{P}$-hard problems. Some of the problems known to be $\#\mathcal{P}$-hard are computing the probability that a probabilistic graph remains connected, counting matchings in a bipartite graph, computing the volume of a polyhedron, evaluating definite integrals of multivariate polynomials, and evaluating the cycle-index polynomial of Polya theory.

The reductions used in the mentioned works often use what can be called a 'blow-up' procedure. These techniques tend to lose special properties of the problem instance, making it hard to use these proofs to demonstrate hardness in restricted cases. For example, even though we know that #PERFECT MATCHINGS is $\#\mathcal{P}$-complete for graphs of degree 3, the standard reduction from #PERFECT MATCHINGS to #MATCHINGS does not show that the latter remains $\#\mathcal{P}$-complete for low-degree graphs, because that reduction blows up the degree of the graph. Hence, if we wish to understand the complexity of counting in more restricted cases, more work needs to be done. We shall demonstrate that it is possible to do such analysis by proving hardness for restricted cases of both #MATCHINGS and another important problem, #VERTEX COVER.

Chapter 4

The Vertex Cover Problem

Before discussing the vertex cover problem, we ought to define it.

Definition 4.1 *Let $G = (V, E)$ be an undirected graph. $V' \subset V$ is said to be a* **vertex cover** *on G iff every edge in E has at least one endpoint in V'.*

In the remaining chapters, one of the things we will attempt to do is classify the types of graphs in which the vertex cover counting problem and its variants are tractable. One important reason for doing this is that the vertex cover problem has potential to serve as an easy problem to reduce from in showing other counting problems to be $\#\mathcal{P}$-complete. A decision version of the vertex cover problem — given a graph G and a number k, determining whether G has a vertex cover of size less than k — has been dubbed one of the "six basic $\mathcal{NP}$-complete problems" by Garey and Johnson [GJ79, Ch. 3]. This is because a large fraction of other $\mathcal{NP}$-completeness results have come via reduction from the vertex cover problem. This suggests that the vertex cover counting problem might also serve as a good starting point for $\#\mathcal{P}$-completeness proofs. Hence, proving that counting vertex covers is hard even in very restricted cases may help in proving that other counting problems are also hard in restricted cases.

Another related reason for studying the vertex cover problem is its intimate relationship with three other important counting problems: #Clique, #Independent Set, and #Monotone 2Sat.

Definition 4.2 *Let $G = (V, E)$ be an undirected graph. $V' \subset V$ is said to be an* **independent set** *iff no two vertices in V' are connected by an edge. $V' \subset V$ is said to be a* **clique** *iff every pair of vertices in V' is connected by an edge.*

Definition 4.3 *A propositional formula F is said to be* **monotone** *iff it contains no negations, i.e. it consists entirely of nonnegated variables, $\wedge$'s, and $\vee$'s.*

Proposition 4.4 *The following counting problems are equivalent, i.e. parsimoniously reducible to each other (although the relationship is even stronger than that):*
1. #VERTEX COVER, *counting vertex covers in an undirected graph.*
2. #INDEPENDENT SET, *counting independent sets in an undirected graph.*
3. #CLIQUE, *counting cliques in an undirected graph.*
4. #MONOTONE 2SAT, *counting satisfying assignments to a monotone 2-CNF propositional formula.*

Proof 1≈2. Let $G = (V, E)$ be an undirected graph. Note that V' is a vertex cover iff $V - V'$ is an independent set. So G has the same number of vertex covers and independent sets.
2≈3. Let $G^C = (V, V \times V - E)$ be the complement of the graph, *i.e.* flip edges and non-edges. Note that V' is an independent set in G iff V' is a clique in G^C, so G has the same number of indendent sets as G^C has cliques.
1≈4. Given $G = (V, E)$, we can write a monotone 2-CNF formula F using V as the set of variables: $F = \bigwedge_{(u,v)\in E}(u \vee v)$. Note that every satisfying assignment α of F corresponds to the vertex cover $\{v \in V : \alpha(v) = 1\}$ and every vertex cover V' yields a satisfying assignment β, where $\beta(v) = 1$ if $v \in V'$ and $\beta(v) = 0$ if $v \notin V'$. Thus F has exactly as many satisfying assignments as G has vertex covers. We can reverse this reduction, too, to produce a graph from any monotone 2-CNF formula such that the graph has exactly as many vertex covers as the formula has satisfying assignments. □

In light of these relationships, we see that any result about counting vertex covers in restricted cases will lead to analogous results for the other three problems. One of these, counting the number of satisfying assignments in a propositional formula, is a very important problem. For example, Roth [Rot95] shows that this problem is inter-reducible with the problem of computing what is called the 'degree of belief' in a Bayesian Belief Network, a significant Artificial Intelligence problem. His reduction shows that if it is known that counting satisfying assignments is hard for restricted classes of formulæ, then we can conclude that computing the degree of belief is hard even for restricted network topologies. He proceeds to show that for some restricted classes of propositional formulæ, counting the number of satisfying assignments is hard. The results we show in the next chapter about the difficulty of counting vertex covers will have as a corollary (in light of the above discussion) that counting satisfying assignments for very restricted classes of monotone 2-CNF formulæ is hard. These results are stronger than any others known for counting satisfying assignments.

One of the variants of the vertex cover problem that we will consider is counting minimal vertex covers.

Definition 4.5 *A vertex cover V' in a graph G is said to be* **minimal (with respect to inclusion)** *iff no proper subset of V' is a vertex cover, i.e. the removal of any vertex from V' uncovers an edge.*

By the relationship between vertex covers and satisfying assignments to monotone 2-CNF formulæ, it is easy to see that minimal vertex covers correspond to minterms of a monotone 2-CNF formula — that is, satisfying assignments for which changing any variable from true to false would no longer satisfy the formula. It is clear that the smallest DNF form for a monotone formula F is simply the disjunction of all minterms, writing an individual minterm M as the conjunction of the variables in M. Conversion of a monotone formula from CNF to DNF is an important problem, but it cannot be solved efficiently in general, because the smallest DNF form for a formula such as $(x_1 \vee y_1) \wedge (x_2 \vee y_2) \wedge \cdots \wedge (x_n \vee y_n)$ has 2^n clauses. The best we can hope for is to solve it time polynomial in the sizes of both the CNF and DNF forms. One way of doing this would be to generate minterms for the formula with only 'polynomial delay.' As we shall see later, generation and counting problems are closely related, so results about the difficulty of counting minterms suggest that CNF to DNF conversion would be intractable even in very restricted cases.[1] Hence, the tractability of counting and generating minimal vertex covers is also an important question.

Finally, the vertex cover problem can be related to a surveillance problem of practical importance known as the "Art Gallery Problem." Say we have an art gallery consisting of various connected hallways. We want to place guards at the junctures of hallways in such a way that every hallway has a guard at at least one end. If we construct a graph G in which edges are hallways, and the hallways meet at vertices, we see that this problem is

[1] The results we have, however, don't *prove* anything about the conversion problem. We only demonstrate that exact counting of minterms is hard, whereas approximate counting is the problem related to generation. Additionally, generation with polynomial delay is not equivalent to uniform generation, which is the problem related to counting.

equivalent to finding a vertex cover for G. Suppose now that each guard has a certain probability p of falling asleep. The question is: "Given the art gallery layout, the initial positioning of guards, and p, what is the probability that all hallways remain under guard?" If we had an algorithm to solve this problem, then we could count the number of vertex covers in a graph as follows: Given $G = (V, E)$, position a guard at every vertex, and set $p = \frac{1}{2}$. Then, for every $V' \subset V$, the probability that the awake guards correspond exactly to V' is $(2^{-|V'|})(2^{-(|V|-|V'|)}) = 2^{-|V|}$. So the probability of having the alert guards correspond to V' is the same for all subsets V' of V. Therefore, the probability that the Art Gallery remains under complete surveillance is exactly (# of vertex covers)$/2^{|V|}$. Multiplying this by $2^{|V|}$, we obtain the number of vertex covers of G, as desired. Hence, determining the reliability of such a surveillance system is at least as hard as counting vertex covers. Thus, when we demonstrate counting vertex covers to be hard in restricted cases, we can conclude that determining the reliability of an Art Gallery surveillance system is hard in analogous restricted cases.

For its simplicity, its relationship to other problems, and its practical significance, the vertex cover problem is an important one to consider. So motivated, we proceed to classify the cases in which counting vertex covers is and is not feasible.

Chapter 5

The Complexity of Counting Vertex Covers

In this chapter, we demonstrate that the problems of counting vertex covers and matchings and variants of these remain hard even in very restricted cases. In particular, we show that counting matchings, maximal matchings, perfect matchings, vertex covers, minimal vertex covers, and minimum cardinality vertex covers all remain $\#\mathcal{P}$-complete even when restricted to bipartite graphs whose degree is bounded by a small constant. Previously, it was only known that these problems are $\#\mathcal{P}$-complete for unrestricted bipartite graphs.[1][Val79b, PB83]

We begin with a statement of results, postponing the proofs until the end of the chapter. First come our positive results, in which we show that all these problems can be solved in polynomial time for graphs of degree 2. This illustrates how things can become much easier in low-degree graphs, making our negative results a bit more surprising. Additionally, some of the relationships we derive while obtaining the positive results are useful in proving the negative results.

Theorem 5.1 (Positive Results) *The following problems are in $\mathcal{FP}$:*

1. $\#2\mu$-Vertex Cover

Input: An undirected graph G of degree ≤ 2.

Output: The number of vertex covers in G.

2. $\#2\mu$-Minimal Vertex Cover

[1] Except for counting perfect matchings, which was shown to be $\#\mathcal{P}$-complete for graphs of degree 3 in [DL92].

Input: An undirected graph G of degree ≤ 2.

Output: The number of minimal vertex covers in G.

3. $\#2\mu$-Minimum Cardinality Vertex Cover

Input: An undirected graph G of degree ≤ 2.

Output: The number of minimum cardinality vertex covers in G.

4. $\#2\mu$-Matchings

Input: An undirected graph G of degree ≤ 2.

Output: The number of matchings in G.

5. $\#2\mu$-Maximal Matchings

Input: An undirected graph G of degree ≤ 2.

Output: The number of maximal matchings in G.

6. $\#2\mu$-Maximum Cardinality Matchings[2]

Input: An undirected graph G of degree ≤ 2.

Output: The number of maximum cardinality vertex covers in G.

Theorem 5.2 (Negative Results) *The following are $\#\mathcal{P}$-complete:*

1. $\#3\mu$-Perfect Matchings

Input: A bipartite graph G of degree ≤ 3.

Output: The number of perfect matchings in G.

2. $\#4\mu$-Matchings

Input: A bipartite graph G of degree ≤ 4.

Output: The number of matchings in G.

3. $\#5\mu$-Maximal Matchings

Input: A bipartite graph G of degree ≤ 5.

Output: The number of maximal matchings in G, i.e. matchings that are not properly contained in any other matching.

4. $\#6\mu$-Vertex Cover

Input: A undirected graph G of degree ≤ 6.

Output: The number of vertex covers in G.

5. $\#$Bipartite Vertex Cover

Input: A bipartite graph G.

[2] This includes $\#2\mu$-Perfect Matchings as a special case, for if a bipartite graph G has a perfect matching then the maximum cardinality matchings are exactly the perfect matchings. Moreover, we can test if G has a perfect matching in polynomial time using one of the standard algorithms [Pap94, Sec 1.2].

Output: *The number of vertex covers in* G*.*

6. #BIPARTITE MONOTONE 2SAT

   Input: *A monotone formula* $F = (c_{1,1} \vee c_{1,2}) \wedge \cdots \wedge (c_{n,1} \vee c_{r,2})$ *on variables* $X \cup Y$*, where* $X \cap Y = \emptyset$ *and* $c_{i,1} \in X$*,* $c_{i,2} \in Y$ *for each* i*.*

   Output: *The number of satisfying assignments to* F*.*

7. #BIPARTITE MONOTONE WEIGHTED 2SAT

   Input: *As in 6.*

   Output: $|\{(\alpha, t) : t \in \{1,2\}^r, \alpha$ *an assignment for* F *such that* α *makes* c_{i,t_i} *true for each* $i\}|$

8. #2ν-BIPARTITE GUARANTEED WEIGHTED 2SAT

   Input: *A pair* (F, β)*, where* $F = (c_{1,1} \vee c_{1,2}) \wedge \cdots \wedge (c_{n,1} \vee c_{r,2})$*, on variables* X*, where* $c_{i,1} \in X$ *and* $\neg c_{i,2} \in X$ *for each* i*; each literal in* F *occurs at most twice; and* β *is a satisfying assignment for* F*.*

   Output: *As in 7.*

9. #3μ-BIPARTITE MINIMUM CARDINALITY VERTEX COVER[3]

   Input: *A bipartite graph* G *of degree* ≤ 3*.*

   Output: *The number of minimum cardinality vertex covers in* G*.*

10. #4μ-BIPARTITE VERTEX COVER

    Input: *A bipartite graph* G *of degree* ≤ 4*.*

    Output: *The number of vertex covers in* G*.*

11. #5μ-BIPARTITE MINIMAL VERTEX COVER

    Input: *A bipartite graph* G *of degree* ≤ 5*.*

    Output: *The number of minimal vertex covers in* G*.*

Corollary 5.3 *The following are* $\#\mathcal{P}$*-complete:*

1. #3μ-BIPARTITE MAXIMUM CARDINALITY INDEPENDENT SET

   Input: *A bipartite graph* G *of degree* ≤ 3*.*

   Output: *The number of maximum cardinality independent sets in* G*.*

2. #4μ-BIPARTITE INDEPENDENT SET

   Input: *A bipartite graph* G *of degree* ≤ 4*.*

   Output: *The number of independent sets in* G*.*

[3] In contrast to the other problems, it is not obvious that this problem is in $\#\mathcal{P}$, as the general problem of checking whether a vertex cover is of minimum cardinality is $\mathcal{NP}$-hard. However, using Hall's theorem, it can be shown that the size of the minimum cardinality vertex cover equals the size of the maximum cardinality matching in *bipartite* graphs. The latter quantity can be found using any of the standard maximum cardinality matching algorithms.

3. #5μ-Bipartite Maximal Independent Set
   Input: A bipartite graph G of degree ≤ 5.

   Output: The number of maximal independent sets in G.

4. #4μ-Bipartite Monotone 2Sat
   Input: A monotone formula $F = (c_{1,1} \vee c_{1,2}) \wedge \cdots \wedge (c_{n,1} \vee c_{r,2})$ on variables $X \cup Y$, where $X \cap Y = \emptyset$; $c_{i,1} \in X$, $c_{i,2} \in Y$ for each i; and each variable appears at most 4 times.

   Output: The number of satisfying assignments to F.

5. #5μ-Bipartite Minterms
   Input: As in 4, except each variable appears at most 5 times.

   Output: The number of satisfying assignments for F such that making any additional variable false would fail to satisfy F, i.e. The number of clauses in the smallest DNF formula equivalent to F.

We now prove the positive results, following a simple lemma relating vertex covers and matchings.

Definition 5.4 *let $G = (V, E)$ be an undirected graph. The* **line graph of G** *is an undirected graph $L(G) = (E, H)$, where $(e_1, e_2) \in H$ iff e_1 and e_2 share an endpoint in G.*

Lemma 5.5 *Let G be an undirected graph. There is a bijective correspondence between matchings in G and vertex covers in $L(G)$, such that maximal matchings correspond to minimal vertex covers and maximum cardinality matchings correspond to minimum cardinality vertex covers.*

Proof (of lemma) Notice that $M \subset E$ is a matching in G iff M is an independent set in $L(G)$. So maximal matchings correspond to maximal independent sets and maximum cardinality matchings correspond to maximum cardinality independent sets. The relationship with vertex covers follows from the discussion in Proposition 4.4. □

Proof (of positive results) For all six problems, observe that it suffices to calculate the result for connected graphs, because, for instance, the number of vertex covers in a graph is equal to the product of the numbers of vertex covers in its connected components. Also observe that we can separate a graph into its connected components in polynomial time. The same is true for the other structures in question. With these facts in mind, note that the only connected graphs of degree ≤ 2 are chains and cycles, so we need only describe how to count these structures on chains and cycles.

1. Let V_n^C be the number of vertex covers in a chain of n vertices. Given a chain $G = v_1 - v_2 - \cdots - v_n$ with $n \geq 2$, the number of vertex covers in G is the number of vertex covers of G that contain v_1 plus the number of vertex covers of G that don't contain v_1. Every vertex cover containing v_1 must be a vertex cover for the shorter chain $v_2 - \ldots - v_n$ and any choice of vertex cover for this shorter chain can be expanded to a vertex cover of G containing v_1 by adding v_1. So there are V_{n-1}^C vertex covers containing v_1. Every vertex cover of G that doesn't contain v_1 must contain v_2 and, in addition, can contain an arbitrary vertex cover of the chain $v_3 - \cdots - v_n$. Thus, $V_n^C = V_{n-1}^C + V_{n-2}^C$. Finally, note that $V_0^C = 1$ and $V_1^C = 2$, so the V_n^C's are simply the Fibonacci numbers. We will use this fact later on. For now, just observe that we can compute V_n^C in time polynomial in n by using this recurrence.[4]

Let V_n^L denote the number of vertex covers in a loop of n vertices. Given a loop $G{=}v_1 - v_2 - \cdots - v_n - v_1$ with $n \geq 3$, observe that every vertex cover of G containing v_1 can be uniquely formed by taking an arbitrary vertex cover of the chain $v_2 - v_3 - \cdots - v_n$ and adding v_1 to it. And every vertex cover of G not containing v_1 must contain v_2 and v_n, but can restrict to an arbitrary vertex cover of the chain $v_3 - \cdots - v_{n-1}$. Hence, $V_n^L = V_{n-1}^C + V_{n-3}^C$, and we can compute V_n^L in polynomial time by first computing V_{n-1}^C and V_{n-3}^C.

2. Let MV_n^C and MV_n^L denote the number of minimal vertex covers in a chain (loop, respectively) of n nodes. As above, let $G = v_1 - \cdots - v_n$ be a chain of n nodes. No minimal vertex cover of G that contains v_1 can contain v_2, for then v_1 would be removable. Not containing v_2, it must contain v_3. Then it can restrict to an arbitrary minimal vertex cover of the chain $v_4 - \ldots - v_n$. Any minimal vertex cover of G that doesn't contain v_1 must contain v_2, and can then restrict to an arbitrary minimal vertex cover on the chain $v_3 - \ldots - v_n$. Hence, $MV_n^C = MV_{n-3}^C + MV_{n-2}^C$, and we can compute MV_n^C in polynomial time using this relation.

Now let $G = v_1 - v_2 - \cdots - v_n - v_1$ be a loop of n nodes. Then every minimal vertex cover of G that doesn't contain v_1 must contain v_2 and v_n, and

[4] Generating functions can be used to derive closed forms from this and the other recurrences given in this section. Such closed forms can be used to compute quantities like V_n^C in time *polylogarithmic* in n, if we're given n in binary notation.

can restrict to an arbitrary minimal vertex cover on the chain $v_3 - \cdots - v_{n-1}$. A minimal vertex cover of G containing v_1 cannot contain both v_2 and v_n, for then v_1 would be superfluous. Those that contain neither v_2 nor v_n must contain v_3 and v_{n-1} and can be an arbitrary minimal vertex cover on the chain $v_4 - \cdots - v_{n-2}$. Those that contain v_2 but not v_n must contain v_{n-1}, but cannot contain v_3. Thus they must contain v_4, but can restrict to an arbitrary minimal vertex cover on the chain $v_5 - \cdots - v_{n-2}$. The case of minimal covers that contain v_n but not v_2 is identical. Hence, $MV_n^L = MV_{n-3}^C + MV_{n-5}^C + 2MV_{n-6}^C = 2MV_{n-3}^C + MV_{n-6}^C$, so MV_n^L can be computed in polynomial time.

3. Let CV_n^C and CV_n^L be the number of minimum cardinality vertex covers in chain of n nodes and a loop of n nodes, respectively. Then,

$$CV_n^C = \begin{cases} n-1 & n \text{ even} \\ 1 & n \text{ odd} \end{cases} \quad CV_n^L = \begin{cases} 2 & n \text{ even} \\ n & n \text{ odd} \end{cases}$$

Trivially, we can compute these in polynomial time. Let us justify these equations. Every vertex in a chain of n nodes can cover at most 2 edges and such a chain has $n-1$ edges, so the minimum cardinality of vertex covers must be $\lceil \frac{n-1}{2} \rceil$. When n is odd, the only way of covering the chain with $\frac{n-1}{2}$ vertices is to have each node cover exactly two edges, which means using every other vertex starting with the second one. There is only one such cover. When n is even, the minimum cardinality vertex cover is of size $\frac{n}{2}$. Since no pair of consecutive vertices in a cover can be separated by more than 2 edges, no vertex cover of size $\frac{n}{2}$ can contain both endpoints. The only ones containing exactly one endpoint are those that contain every other vertex beginning with one endpoint, so there are 2 of those. Those that contain neither endpoint must contain both vertices next to the endpoints. The remaining $\frac{n}{2} - 2$ vertices in the cover cut the center $n-3$ edges into $\frac{n}{2} - 1$ pieces, all of length 1 or 2. Say there are k pieces of length 1, then there are $\frac{n}{2} - 1 - k$ of length 2, and $k + 2\left(\frac{n}{2} - 1 - k\right) = n - 3$, *i.e.* $k = 1$. So there is one piece of length 1, and we can place our vertices to make any of the center $n-3$ edges this length 1 piece. Thus there are $n-3$ minimum cardinality vertex covers not containing either endpoint, for a total of $n-1$.

For a cycle of n nodes, where n is even, it is clear that a minimum cardinality vertex cover is of size $\frac{n}{2}$ (*e.g.* one that uses every other vertex). There are two such vertex covers. If n is odd, the minimum cardinality vertex cover is of size $\lceil \frac{n}{2} \rceil = \frac{n+1}{2}$. Every cover of this size cuts the cycle up into pieces of length 1 and 2. If k is the number of pieces of length 1, then the number of pieces of length 2 is $\frac{n+1}{2} - k$, so we have $k + 2\left(\frac{n+1}{2} - k\right) = n$, *i.e.* $k = 1$, and we can choose any edge to be this piece of length 1. Thus there are n minimum cardinality vertex covers.

4, 5, 6. These all follow from Lemma 5.5 and the results for vertex covers, as the line graph of a chain of n nodes is a chain of $n-1$ nodes and the line graph of a cycle of n nodes is a cycle of n nodes. In particular, note that if M_n^C is the number of matchings in a chain of n nodes, then the M_n^C's are simply the Fibonacci numbers. $\square$

Before proving the negative results, we need a few facts. For any rational q, let $\|q\|$ denote the number of bits needed to represent q, essentially $\lceil \log_2 a \rceil + \lceil \log_2 b \rceil$, where $q = a/b$ in lowest terms.

Fact 1 *Let $A = (a_{ij})$ be an $n \times n$ matrix with rational entries. Then the determinant and inverse of A (if A is invertible) can be computed in time polynomial in n and $\max_{i,j} \|a_{ij}\|$.*

Fact 2 *Suppose we have linear equations $\sum_{j=1}^{n} a_{ij} x_j = c_j$ for with unknowns $x_1, \ldots, x_n$ for $j = 1, \ldots, n$. Then, if the matrix $A = (a_{ij})$ is invertible, we can recover $x_1, \ldots, x_n$ in time polynomial in n and $\max_{i,j} \|a_{ij}\|$. In particular, if the matrix A satisfies either $a_{ij} = z_i^{j-1}$ or $a_{ij} = z_j^{i-1}$ for distinct $z_1, \ldots, z_n$ (i.e. A is Vandermonde), we can recover $x_1, \ldots, x_n$ in polynomial time.*

Fact 3 *Let $f(x) = \sum_{i=0}^{d} a_i x^i$ be a polynomial with rational coefficients. If $(\alpha_0, \beta_0), \ldots, (\alpha_d, \beta_d)$ such that $f(\alpha_i) = \beta_i$ are known for distinct rational α_i's, then the coefficients of f can be recovered in time polynomial in the $\max_i \|\alpha_i\|$,$\max_i \|\beta_i\|$, and d.*

Fact 4 *Let $f(x) = \sum_{i=0}^{d} a_i x^i$ be a polynomial with nonnegative integer coefficients. If (α, β) is known where $f(\alpha) = \beta$ and α is an integer satisfying $\alpha > a_i$ for each i, then the coefficients of f can be recovered in time polynomial in $\|\alpha\|$, $\|\beta\|$, and d.*

Fact 5 *Let f_n denote the nth Fibonacci number. That is, $f_0 = 1$, $f_1 = 1$, and $f_{n+2} = f_n + f_{n+1}$ for all $n \geq 0$. Let $r_n = f_{n+1}/f_n$. Then $r_i \neq r_j$ for any $i \neq j$.*

Proof (of Fact 1) This is well-known. We generalize the method described in [Val79b]. To compute $\det A$ for an integer matrix, perform Gaussian elimination with arithmetic modulo 2^k for k chosen so that 2^k is larger than $2(\max_{i,j}(|a_{i,j}|))^n(n!)$ to obtain an upper-triangular matrix. (Notice that k can be chosen polynomially large.) At each stage, choose as pivot the element of the row with the fewest factors of 2. This ensures that the multipliers will never be divisible by 2, and will therefore be invertible modulo 2^k. The determinant of A modulo 2^k is the determinant of the upper-triangular matrix obtained, divided by the multipliers. Notice that k was chosen sufficiently large so that we could obtain $\det A$ (with sign) from $\det A \pmod{2^k}$. To compute $\det A$ for a rational matrix, first compute p, the product of all the denominators of the entries. Then, $\det A = (\det pA)/p^n$, and pA is an integer matrix. To invert A quickly, simply use the determinantal rule for the inverse in conjunction with the above. □

Proof (of Fact 2) If $A = (a_{ij})$ is the matrix of coefficients, $c = (c_j)$ is the vector constants, and $x = (x_i)$ is the vector of unknowns, we have $Ax = c$. So, if A is invertible, we can use Fact 1 to obtain $x = A^{-1}c$. For the part about Vandermonde matrices, assume that A is not invertible and that $a_{ij} = z_i^{j-1}$ for all i, j. Then there is a vector a vector $v = (v_i)$ such that $Av = 0$. But this tells us that the polynomial $\sum_{k=0}^{n-1} v_{k+1}x^k$ has n distinct roots $z_1, \ldots, z_n$, which is a contradiction. For the case that $a_{ij} = z_j^{i-1}$, we have just shown that the transpose of A is invertible, so A must be, too. □

Proof (of Fact 3) This is just a special case of Fact 2, with a Vandermonde matrix. □

Proof (of Fact 4) All we need to do is write β as a number in base α and the digits are our coefficients. In more detail, note that $\sum_{i=0}^{d-1} a_i\alpha^i \leq \sum_{i=0}^{d-1}(\alpha^i - 1)\alpha^i = \alpha^d - 1 < \alpha^d$, so $a_d = \lfloor \beta/\alpha^d \rfloor$, which we can obtain quickly just by integer multiplication and division. Now, consider $f_1(x) = f(x) - a_d x^d$. We can repeat this process for f_1, obtaining the coefficients in sequence. □

Proof (of Fact 5) It is a well-known fact that $r_n \to \delta$ as $n \to \infty$, where $\delta = (1+\sqrt{5})/2$. From the Fibonacci recurrence, we see that $r_{n+1} = 1 + \frac{1}{r_n}$. Hence, if $r_i = r_j$ for some $i < j$, the sequence $\{r_n\}$ would have to be eventually periodic with period $j - i$. As a nonconstant periodic sequence never converges, it would have to be eventually constant at δ. By noting that $r_n = (r_{n+1} - 1)^{-1}$ and that δ is a fixed point of this recurrence, we can

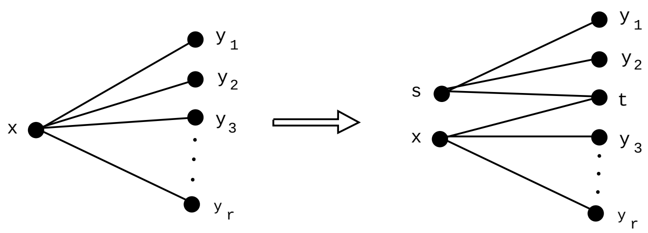

Figure 5.1: The construction in reduction 1

conclude that the sequence would have to be constant at δ starting with r_0. But $r_0 = 1$. □

Proof (of negative results)
1. #PERFECT MATCHINGS $\propto$ #3μ-PERFECT MATCHINGS
Given a bipartite undirected graph $G = (X \cup Y, E)$, let $x \in X$ be any node of degree > 3. Let $y_1, y_2, \ldots, y_r \in Y$ be the neighbors of x. Consider $G' = (X' \cup Y', E')$, where

$$\begin{aligned} X' &= X \cup \{s\}, \\ Y' &= Y \cup \{t\}, \text{ and} \\ E' &= (E - \{(x, y_1), (x, y_2)\}) \cup \{(x, t), (s, t), (s, y_1), (s, y_2)\} \end{aligned}$$

Then G' has the same number of perfect matchings as G: Any perfect matching in G which matches x to y_1 or y_2 yields a perfect matching in G' matching x to t and s to y_1 or y_2. Any perfect matching in G which matches x to y_k for $k > 2$ yields one in G' matching s to t. Conversely, any perfect matching in G' matches t to either x or s. If t is matched to x, then s is matched to either y_1 or y_2, and we obtain a perfect matching in G by matching x to y_1 or y_2. If t is matched to s, then we obtain a perfect matching in G by removing (s, t) from the matching. Notice that the new nodes s and t both have degree ≤ 3 and the degree of x is reduced by 1. We can repeat this construction on all nodes in X and Y until G has degree 3. This can be done in polynomial time.

2. #3μ-PERFECT MATCHINGS $\propto$ #4μ-MATCHINGS
Let G be an undirected bipartite graph of degree ≤ 3 with vertices $v_1, \ldots, v_n$. For each $s = 1, \ldots, n+1$, construct G_s as follows: Add nodes $v_{i,j}$ to G for $1 \leq i \leq n$, $1 \leq j \leq s$. Add edges to create disjoint chains $v_i - v_{i,1} - v_{i,2} - \cdots - v_{i,s}$

of $s+1$ nodes. Note that G_s has degree at most 4, since G has degree at most 3, and that G_s remains bipartite. Now, for every matching M in G, we can obtain matchings in G_s by putting some of the new edges into M. For each vertex v_i of G which is matched by M, $(v_i, v_{i,1})$ cannot be added to the matching, but we can choose an arbitrary matching for the chain $v_{i,1} - \cdots - v_{i,s}$, and there are M_s^C ways to do this. (Recall that M_k^C is the number of matchings in a chain of k nodes.) For each vertex v_i which is not matched by M, we can add an arbitrary matching of the chain $v_i - v_{i,1} - \cdots - v_{i,s}$ to M, and there are M_{s+1}^C ways to do this. Additionally, every matching of G_s is obtained by extending a matching of G in this fashion, because the restriction of any matching in G_s must be a matching in G. Let A_i be the number of matchings in G in which exactly i vertices are matched. Then the number of matchings in G_s is exactly $\sum_{i=0}^{n} A_i (M_s^C)^i (M_{s+1}^C)^{n-i}$. We can obtain these values for $s = 1, \ldots, n+1$ with $n+1$ oracle calls. Dividing by $(M_{s+1}^C)^n$, we obtain the evaluation of $f(x) = \sum_{i=0}^{n} A_i x^i$ at the points (M_s^C / M_{s+1}^C) for $s = 1, \ldots, n+1$. Recall that the M_s^C's were the Fibonacci numbers. By Fact 5, we evaluate f at $n+1$ distinct points, and by Fact 3 we can efficiently recover the coefficients of f. A_n is the number of perfect matchings in G.[5]

3. #4μ-Matchings $\propto$ #5μ-Maximal Matchings

The reduction we use is from [PB83], though they use it to prove something different. Given a bipartite undirected graph $G = (V, E)$ of degree ≤ 4, construct $G' = (V', E')$, where $V' = V \cup \{v' : v \in V\}$ and $E' = E \cup \{(v, v') : v \in V\}$. Note that every matching W in G yields a unique maximal matching $W' = W \cup \{(v, v') : v \text{ not matched by } W\}$ in G' and every maximal matching of G' can be obtained in this fashion. (In every maximal matching, every $v \in V$ is matched to either v' or another vertex in V.) Observe that G' is bipartite and of degree ≤ 5.

4. #4μ-Matchings $\propto$ #6μ-Vertex Cover

This is immediate from Lemma 5.5, noting that the degree of a node e in $L(G)$ is the sum of the degrees of its endpoints in G minus 2.

[5] It appears that the number of oracle calls in this reduction and reduction 10 cannot be reduced to a constant. This is in constrast to the reductions done in [Val79b]. There, Valiant asserts that all the reductions can be done with a single oracle call, because the arithmetic can be simulated by operations on the graph or formula in question. Here that does not appear to work, because the graph manipulations necessary all blow up the degree.

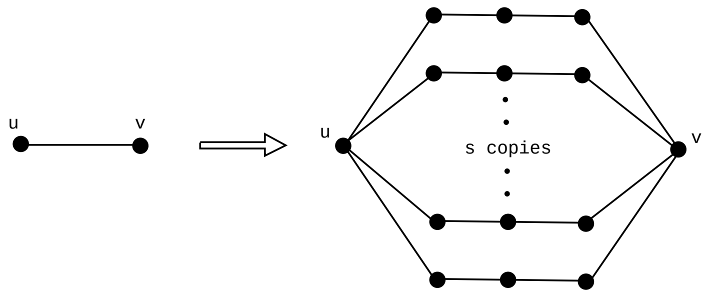

Figure 5.2: The gadget in reduction 5

5. #6μ-Vertex Cover $\propto$ #Bipartite Vertex Cover
This reduction is from [PB83]. Given $G = (V, E)$, let $N = (|E|^2 + |E|)/2$. For each $s = 0, 1, 2, \ldots, N-1$, form a graph G_s by replacing each edge (u, v) with the gadget shown in Figure 5.

This gadget has 2^s vertex covers containing neither u nor v, 3^s vertex covers containing u but not v, 3^s vertex covers containing v but not u, and 5^s vertex covers containing both u and v. For each $i, j, k \geq 0$ such that $i + j + k = |E|$, let $A_{i,j,k}$ be the number of subsets of V which contain neither endpoint of i edges (in G), exactly one endpoint of j edges, and both endpoints of k edges. Then, the number of vertex covers in G_s is

$$\sum_{i+j+k=|E|} A_{i,j,k}(2^s)^i(3^s)^j(5^s)^k = \sum_{i+j+k=|E|} A_{i,j,k}(2^i 3^j 5^k)^s$$

By making N oracle calls, we can evaluate these sums for $s = 0, 1, \ldots, N$. The matrix of coefficients in the $N \times N$ system of linear equations we obtain is Vandermonde, as $2^i 3^j 5^k = 2^{i'} 3^{j'} 5^{k'}$ if and only if $i = i'$, $j = j'$, and $k = k'$. Hence, by Fact 2, we can recover the coefficients $A_{i,j,k}$ in polynomial time. In particular, we can obtain $\sum_{j+k=|E|} A_{0,j,k}$, which is the number of vertex covers in G.

6. #Bipartite Vertex Cover $\propto$ #Bipartite Monotone 2Sat
This is immediate from the discussion in the proof of Proposition 4.4.

7. #Bipartite Monotone 2Sat $\propto$
#Bipartite Monotone Weighted 2Sat
This reduction is from [Val79b]. Given bipartite monotone 2-CNF F as

described, let A_k be the number of satisfying assignments to F that make both literals true in exactly k of the clauses (and make exactly one literal true in the remaining $n-k$ clauses). Consider the #Bipartite Monotone Weighted 2Sat-oracle applied to $F^s = F \wedge \cdots \wedge F$. For each satisfying assignment to F that makes both literals true in exactly k clauses, there are 2^{sk} choices of t for F^s. Thus, the value returned by the oracle will be $\sum_{i=0}^{r} A_i (2^s)^i$. If we choose s to be the number of variables in F plus one, then $2^s > A_i$ for each i, and then, by Fact 4, we can recover $A_1, \ldots, A_r$. The sum of these is the number of satisfying assignments to F, which is what we want. Notice that each F^s satisfies the bipartite condition.

8. #Bipartite Monotone Weighted 2Sat $\propto$
#2ν-Bipartite Guaranteed Weighted 2Sat
Given bipartite monotone F in 2-CNF on variables $X \cup Y$, first negate every variable in Y. This does not alter the number of satisfying assignments nor the number of choices of t. Now consider each variable $x \in X \cup Y$ in turn. If x occurs $k > 2$ times in F, replace the ith occurrence with a new variable x_i and conjoin the clauses $(x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3) \cdots \wedge (x_n \vee \neg x_1)$ onto F. This corresponds to requiring that $x_1 \leftarrow x_2 \leftarrow \cdots \leftarrow x_n \leftarrow x_1$, which is to say that all the x_i's must take on the same value in any satisfying assignment. Doing this for each of the original variables in F, we obtain a formula F', in which each *literal* appears at most twice. (Positive literals appear at most twice, negative literals once). Now, every satisfying assignment to F yields a satisfying assignment to F' by assigning $x_1, \ldots, x_k$ the value that x got, and conversely. We need to note that the number of choices of t does not change. This is clear, because the number of ways to choose t_i does not change if the ith clause was originally in F, and in each of the new clauses, exactly one literal is true in every satisfying assignment, so there is only one way to choose t_i. Observe, additionally, that we can use the assignment β which sets all variables from X to true and all variables from Y to false as the required assignment in the input. Finally, note that every clause is of the form $(x \vee \neg y)$, where x and y are variables.

9. #2ν-Bipartite Guaranteed Weighted 2Sat $\propto$
#3μ-Bipartite Minimum Cardinality Vertex Cover
Given F, we construct a graph with $n+r$ dumbbells, where n is the number of variables in F and r is the number of clauses. Specifically, for each variable x, we have two nodes, x and $\neg x$, which are connected by an edge. For each clause $(c_{i,1} \vee c_{i,2})$, we have a dumbbell and we connect the endpoints

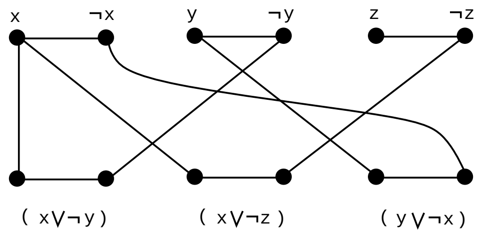

Figure 5.3: The graph produced for $F = (x \vee \neg y) \wedge (x \vee \neg z) \wedge (y \vee \neg x)$ in reduction 9

to the nodes corresponding to the literals $c_{i,1}$ and $c_{i,2}$. Formally, if F is over variables $\{x_1, \ldots, x_n\}$, let $G = (V, E)$, where

$$V = \{x_i, \neg x_i : 1 \leq i \leq n\} \cup \{(i,j) : 1 \leq i \leq r, 1 \leq j \leq 2\}$$

$$\begin{aligned} E &= \{\langle x_i, \neg x_i\rangle : q \leq i \leq n\} \\ &\cup \{\langle (i,1),(i,2)\rangle : 1 \leq i \leq r\} \\ &\cup \{\langle c_{i,j},(i,j)\rangle : 1 \leq i \leq r, 1 \leq j \leq 2\} \end{aligned}$$

Observe that every vertex cover in G is of size at least $n + r$ (to cover every dumbbell.) We will show that there is exactly one such cover for every choice of (α, t) for F. Given (α, t) satisfying the appropriate conditions, let

$$V' = \{x : \alpha(x) = 1\} \cup \{\neg x : \alpha(x) = 0\} \cup \{(i,j) : t_i \neq j\}$$

Note that V' a vertex cover of size $n + r$. It is clear that the first two types of edges are covered. To see that the edges of the last type are covered, just note that if $c_{i,j}$ is not in the cover, then $c_{i,j}$ is not true, so $t_i \neq j$, which means that (i,j) is in the cover. There is at least one cover of this size, since F is satisfiable. Additionally, every vertex cover V' of size $n + r$ must contain exactly one of each pair $x_i, \neg x_i$ and exactly one of each pair $(i,1),(i,2)$, so if we set $\alpha(x_i) = 1$ iff $x_i \in V'$ and $t_i = j$ iff $(i,j) \notin V'$, we obtain a correct pair (α, t). (Note that if $t_i = j$, then $(i,j) \notin V'$, so $c_{i,j} \in V'$, so $\alpha(c_{i,j}) = 1$, as needed.)

If we partition the vertices into $X = \{x_i\} \cup \{(i,2)\}$ and $Y = \{\neg x_i\} \cup \{(i,1)\}$, we see that G is bipartite. Also observe that each 'clause' node has degree 2, whereas each 'literal' node has degree one greater than the number of times it appears in F. Hence, the graph has degree at most 3, and this is a parsimonious reduction to #3μ-Bipartite Minimum Cardinality Vertex Cover.

10. #3μ-Bipartite Minimum Cardinality Vertex Cover $\propto$
#4μ-Bipartite Vertex Cover
This is similar to 2. Let G be a bipartite graph of degree ≤ 3 on vertices $v_1, \ldots, v_n$. For each $s = 1, \ldots, n+1$, construct G_s as follows: Add nodes $v_{i,j}$ to G for $1 \leq i \leq n$, $1 \leq i \leq s$. Add edges to create disjoint chains $v_i - v_{i,1} - v_{i,2} - \cdots - v_{i,s}$ of $s+1$ nodes. Note that G_s is bipartite and of degree at most 4, since G is bipartite and of degree at most 3. Now, for every vertex cover V' in G, we can obtain vertex covers in G_s by putting some of the new vertices into V' until each chain is covered. For each vertex v_i of G which is in V', we must cover the chain $v_{i,1} - \cdots - v_{i,s}$, and there are V_s^C ways to do this. (Recall that V_k^C is the number of vertex covers in a chain of k nodes.) For each vertex v_i which is not in V', $v_{i,1}$ must be in a vertex cover of G_s. But we can cover the remaining chain $v_{i,2} - \cdots - v_{i,s}$ arbitrarily and there are V_{s-1}^C ways to do this. Thus, if A_k is the number of vertex covers in G of size exactly k, the number of vertex covers in G_s is exactly $\sum_{i=0}^{n} A_i (V_s^C)^i (V_{s-1}^C)^{n-i}$. We can obtain these values for $s = 1, \ldots, n+1$ with $n+1$ oracle calls. Dividing by $(V_{s-1}^C)^n$, we obtain the evaluation of $f(x) = \sum_{i=0}^{n} A_i x^i$ at the points (V_s^C / V_{s-1}^C) for $s = 1, \ldots, n+1$, which are ratios of consecutive Fibonacci numbers. By Facts 3 and 5, we can efficiently recover the coefficients of f. The first nonzero coefficient is the number of minimum cardinality vertex covers in G.

11. #4μ-Bipartite Vertex Cover $\propto$
#5μ-Bipartite Minimal Vertex Cover
This is similar to 3. Given a bipartite graph $G = (V, E)$ of degree ≤ 4, construct $G' = (V', E')$, where $V' = V \cup \{v' : v \in V\}$ and $E' = E \cup \{(v, v') : v \in V\}$. Note that every vertex cover W in G yields a unique minimal vertex cover $W' = W \cup \{v' : v \notin W\}$ in G' and every minimal vertex cover of G' can be obtained in this fashion. (Every minimal vertex cover contains exactly one of each pair v, v', for if both were contained, we could remove v'.) Observe that G' is bipartite and of degree ≤ 5. □

Proof (of Corollary 5.3) These follow immediately from Theorem 5.2 and the discussion in Proposition 4.4. □

Chapter 6

Approximate Counting and Uniform Generation

At first, complexity theory appears to be a very negative science. After all, a proof that a problem is $\mathcal{NP}$-hard or $\#\mathcal{P}$-complete tells us what we cannot do, without providing alternatives. However, hardness results from complexity theory can serve the very positive purpose of redirecting research in directions that are potentially more fruitful. The study of the complexity of counting is a striking example of this phenomenon. After discovering that important counting problems are $\#\mathcal{P}$-complete, one can either attempt to solve these counting problems in restricted cases or seek algorithms that give provably good estimates to the exact counts. As our results indicate, exact counting remains hard even in the most restricted cases, leaving approximate counting as the only viable alternative. Jerrum and Sinclair have recently introduced techniques which solve the approximate counting analogues of several $\#\mathcal{P}$-complete problems and can potentially solve many more. Their approach, based on the simulation of Markov chains, might never have been taken had $\#\mathcal{P}$-completeness results not demanded an alternative to exact counting.

Let us now make precise the notion of approximate counting.

Definition 6.1 *Given three nonnegative real numbers α, β, and r, we say that* **α approximates β within ratio r** *iff $\alpha/r \leq \beta \leq \alpha r$. (Clearly $r \geq 1$ is necessary for this to hold, and the smaller $r-1$ is, the better the estimate.)*

Definition 6.2 *Let R be a p-relation. The* **approximate counting prob-**

lem $\approx\boldsymbol{R}$ *associated with* R *is the search problem*

$$\approx R(\langle x, \epsilon\rangle) = \{y \in \mathbb{N} : y \textit{ approximates } |R(x)| \textit{ within ratio } 1+\epsilon\}$$

Notice that the desired accuracy of the approximate counter is included with the input. We can also discuss variants of the approximate counting problem, in which we ask for an approximation to $|R(x)|$ within a ratio depending on the instance x. For example, we can ask if there is an efficient algorithm which approximates the number of vertex covers in a graph within ratio n^2, where n is the number of vertices in the graph.

The complexity of approximate counting as we've defined can be discussed solely with respect to standard deterministic Turing machines, as defined in Chapter 2. However, randomized computation has proved to be a more useful model for the study of approximate counting. In order to take this approach, we need to define probabilistic Turing machines. Following [JVV86],

Definition 6.3 *A* **probabilistic Turing machine (PTM)** M *is a standard Turing machine, except that some states are permitted to be coin-tossing states. When in such a coin-tossing state, the action of the PTM (its next state, movement of the tape heads, etc.) is determined by the flip of a fair coin. Thus, we can regard the output* $M(x)$ *of* M *as a random variable depending on the outcome of the coin flips.*

Now it is necessary to define what it means for a PTM to solve a search problem. We would like our definition to say that a PTM M solves a search problem Π if, on any input x, M is *likely* to output a member of $\Pi(x)$. However, it is not clear what M should be required to do the rest of the time: should we demand that M indicate failure with an output of N, or should we allow it to give arbitrary output? Unfortunately, different answers to this question have proved useful in different contexts. We choose the latter option, as it is more relevant to our discussion of counting.

Definition 6.4 *Let* Π *be a search problem over alphabet* Σ*. A PTM* M *is said to* **solve** $\boldsymbol{\Pi}$ **with bounded probability of error** *iff*

$$\begin{aligned} Pr(M(x) \in \Pi(x)) &\geq \tfrac{3}{4} \quad \textit{if } \Pi(x) \neq \emptyset \\ Pr(M(x) = N) &\geq \tfrac{3}{4} \quad \textit{if } \Pi(x) = \emptyset \end{aligned}$$

Definition 6.5 $\mathcal{SBPP}$ *is the class of all search problems solved by polynomial-time PTM's with bounded probability of error.* $\mathcal{BPP} = \mathcal{SBPP} \cap \mathcal{D}$ *is the class of all such decision problems.*

Thus, we allow false answers to be given, but only with probability bounded above by $\frac{1}{4}$. An important feature of $\mathcal{BPP}$ algorithms is that with k independent trials, the probability that the majority of the trials have a wrong answer is at most $e^{-k/12}$.[1] For all practical purposes, an answer obtained by taking the majority from 1200 runs of a $\mathcal{BPP}$ algorithm is as good as a certain. With this in mind, the basic question of complexity theory becomes "$\mathcal{NP} \subset \mathcal{BPP}$?" rather than "$\mathcal{P} = \mathcal{NP}$?". Again, extensive research has failed to produce $\mathcal{BPP}$ algorithms for any of the $\mathcal{NP}$-complete problems, so it is widely believed that $\mathcal{NP} \not\subseteq \mathcal{BPP}$. However, it appears that randomization does reduce complexity to some degree. For example, deciding whether a multivariate polynomial over $\mathbb{Z}$ in the form $\sum_i \prod_j \sum_k t_{i,j,k}$ (where the $t_{i,j,k}$ are terms) is identically zero is in $\mathcal{BPP}$ but no polynomial-time deterministic algorithms are known. Primality testing is another problem known to be in $\mathcal{BPP}$ but not known to be in $\mathcal{P}$.[2]

As a whole, $\mathcal{SBPP}$ is not a very useful complexity class. If Π is an arbitrary search problem, in which $|\Pi(x)|$ can be exponentially large, there is no obvious way to obtain an element of $\Pi(x)$ with arbitrary high probability using an algorithm that gives an element of $\Pi(x)$ with probability only 3/4. Repeated runs may give us entirely distinct, unrelated elements of $\Pi(x)$ and there need not be any way of distinguishing those from the false outputs. With decision and approximation problems, however, $\mathcal{SBPP}$ algorithms are useful. Our only reason for defining the class $\mathcal{SBPP}$ is to allow us to relate the complexity of approximation problems to those of decision problems, as the following proposition allows us to do.

Proposition 6.6 $\mathcal{SBPP}$ *is closed under polynomial-time Turing reductions which make 1 oracle call.*

Proof The proof is the same as for $\mathcal{SP}$, as in Proposition 2.15, noting that the probability of a correct answer is maintained if we make only a single oracle call. □

[1] This can be obtained via the Chernoff Bound, as given in [Pap94, page 258].

[2] A deterministic algorithm for testing primality is known, but it is based on the Rieman hypothesis of number theory.

This proposition says that if we show a search problem Π to be $\mathcal{NP}$-hard via reductions that make only 1 oracle call, then $\Pi \in \mathcal{SBPP}$ implies that $\mathcal{NP} \subset \mathcal{BPP}$. This can be considered convincing evidence that Π has no $\mathcal{SBPP}$ algorithm. Because statements like this depend on the reduction type, it is important to always be explicit about the kind of reduction being used.

Although they fall into the class of algorithms defined above, we define randomized approximation algorithms separately to make the terminology completely clear.

Definition 6.7 *Let R be a p-relation and $\approx R$ be the associated approximate counting problem. A* **randomized approximation scheme for $\approx \boldsymbol{R}$** *is a PTM that solves $\approx R$ with bounded probability of error. A* **fully-polynomial randomized approximation scheme(fpras) for $\approx \boldsymbol{R}$** *is a randomized approximation scheme that runs in time polynomial in $|x|$ and $1/\epsilon$ (where $\langle x, \epsilon \rangle$ is the input).*

Now, as with $\mathcal{BPP}$, given a fpras which can fail up to 1/4 of the time, we can obtain another fpras whose probability of error is arbitrarily small. We do this by running the original algorithm k times in succession and taking the median of the outputs. The probability that the median is not an approximation within ratio $1+\epsilon$ is bounded above by the probability that the majority of outputs are not approximations within ratio $1+\epsilon$. As with $\mathcal{BPP}$, this happens with probability at most $e^{-k/12}$. This result will be useful later, so we state it as a separate lemma.

Lemma 6.8 ([JVV86]) *If there is a fpras for $\approx R$, then there is a PTM M, that on input $\langle x, \epsilon, \delta \rangle$ outputs an approximation to $|R(x)|$ within ratio $1+\epsilon$ with probability at least $1-\delta$. Moreover, the running time of M is bounded by a polynomial in $|x|, 1/\epsilon$, and $\log(1/\delta)$.*

Something unusual in our definition of fpras is that we only require that the running time depend polynomially on $1/\epsilon$ and not $\log(1/\epsilon)$. We typically define running time in terms of length of the input, and the number of bits to express rationals is only logarithmic in the numerator and denominator. Our definition corresponds to representing ϵ as a rational in *unary* notation (*i.e.* with only a 1-symbol alphabet) and asking for a $\mathcal{SBPP}$ algorithm. If we had followed our usual convention of using binary representations of rationals, then approximate counting would be *equivalent* to exact counting, making it a useless concept! Specifically, if R is a p-relation on alphabet Σ with

$y \in R(x) \Rightarrow |y| \leq p(|x|)$, then $|R(x)| \leq |\Sigma|^{p(|x|)}$. Thus, an approximation to $|R(x)|$ within ratio $1+\epsilon$, where $\epsilon = 1/2^{1+p(x)\log_2|\Sigma|}$, is in fact an exact count. Since $\log\epsilon$ is polynomially bounded in $|x|$, an approximation scheme that could run in time polynomial in $|x|$ and $\log\epsilon$ would yield an polynomial-time exact solution to $\#R$.

A priori, there is no reason to believe that approximate counting would be much easier than exact counting. The first result that suggested that there is a large complexity gap between these two types of problems was given by Stockmeyer [Sto83]. He showed that randomized approximate counting is $\mathcal{NP}$-easy. In other words, for any p-relation R, there exists a PTM equipped with an oracle from $\mathcal{NP}$ which performs as a fully-polynomial randomized approximation scheme for $\approx R$.[3] This is in contrast to $\#\mathcal{P}$, which is believed to be much harder than $\mathcal{NP}$ [Pap94, page 448]. Stockmeyer's result gives hope for finding fpras's for p-relations whose exact counting problems are $\#\mathcal{P}$-complete. Most of the success in this direction has come from the intimate ties between approximate counting and almost uniform generation, as first noted in [JVV86].

Definition 6.9 *Let R be a p-relation on Σ^*.*
An **almost uniform generator** *for R is a PTM M which takes inputs $\langle x, \epsilon\rangle \in \Sigma^* \times \mathbb{R}^+$, such that for some function $\varphi: \Sigma^* \rightarrow (0,1]$ and all $y \in \Sigma^*$:*

1. *$M(x,\epsilon) \in R(x) \cup \{N\}$, where $N \notin \Sigma^*$*
2. *$Pr[M(\langle x,\epsilon\rangle) = y] = 0$ if $y \notin R(x)$*
3. *$\varphi(x)/(1+\epsilon) \leq Pr[M(\langle x,\epsilon\rangle) = y] \leq \varphi(x)(1+\epsilon)$ if $y \in R(x)$*
4. *$Pr[M(\langle x,\epsilon\rangle) \in R(x)] \geq \frac{1}{2}$ if $R(x) \neq \emptyset$*

Such a generator is said to be **fully-polynomial** *if its running time is bounded by a polynomial in $|x|$ and $\log(1/\epsilon)$.*

In the above definition, the parameter ϵ indicates how much the distribution of outputs is allowed to deviate from uniform. An output of N indicates failure, and this is only allowed to occur with probability smaller than 1/2. Observe that our definition of fully-polynomial in this case requires polynomial dependence on $\log(1/\epsilon)$, in contrast with approximate counting.

It is not surprising that approximate counting and almost uniform generation are related; anyone who has spent some time working on generation

[3] The result mentioned was not stated by Stockmeyer [Sto83], but is implicit in his proofs, as noted by [JVV86].

problems would agree that uniform generation can often be reduced to counting. What is surprising, however, is that natural versions of these two problems are *equivalent* for many p-relations. The main result of [JVV86] says that a fully-polynomial randomized approximation scheme for a p-relation R yields a fully-polynomial uniform generator for R and conversely, as long as R is *self-reducible.*

Definition 6.10 *A p-relation $R \subset \Sigma^* \times \Sigma^*$ is* **self-reducible** *iff there exist polynomial-time computable functions $g : \Sigma^* \to \mathbb{N}$, $\sigma : \Sigma^* \to \mathbb{N}$, $\psi : \Sigma^* \times \Sigma^* \to \mathbb{N}$ satisfying:*
1. $y \in R(x) \Rightarrow |y| = g(x)$
2. $\sigma(x) \in O(\log |x|)$
3. $g(x) > 0 \Rightarrow \sigma(x) > 0$ *for all* $x \in \Sigma^*$
4. $|\psi(x, w)| \leq |x|$ *for all* $x, w \in \Sigma^*$
5. $\langle x, y_1 \cdots y_n \rangle \in R \iff \langle \psi(x, y_1 \cdots y_{\sigma(x)}), y_{\sigma(x)+1} \cdots y_n \rangle \in R$ *for all* $x \in \Sigma^*, y = y_1 \cdots y_n \in \Sigma^*$

This definition captures the idea that solutions associated to an instance x of R can be expressed in terms of solutions of a smaller instance of R. Many natural relations are self-reducible in this intuitive sense, and their p-relations can be made to fit the above definition. For example, consider the p-relation associating a graph x to the set $R(x)$ of vertex covers in x. We can choose our encodings so that a potential vertex cover (a subset of the vertices) in a graph of n vertices is represented by a string of n 0's and 1's (with a 1 in the ith position meaning that the ith vertex is in the cover). The vertex covers in a graph x which contain the first vertex v_1 are in natural 1-1 correspondence with the vertex covers in the smaller graph x' formed by removing v_1 and all its incident edges. The vertex covers in x not containing v_1 are in 1-1 correspondence with the vertex covers in the smaller graph x'' formed by removing v_1, all its neighbors, and all edges incident to its neighbors. Condition 4 implicitly states that the solutions to x' and x'' must be of the same size. In order to accomodate this, we can allow our graphs to have a numbering of vertices that don't begin with 1 and require that any solution has a 1 in positions corresponding to vertices not actually in the graph. Thus, in the above definition $g(x) = n, \sigma(x) = 1, \psi(x, 1) = x'$, and $\psi(x, 0) = x''$.

By similar techniques, it can be shown that there are self-reducible p-relations for most of the problems we have considered, including Perfect Matchings, Matchings, Maximal Matchings, and Minimal Vertex

COVER. MINIMUM CARDINALITY VERTEX COVER is not likely to be self-reducible in general, as it is $\mathcal{NP}$-hard to determine the size of the minimum cardinality vertex cover in an arbitrary graph, but it is self-reducible when restricted to bipartite graphs.

We are now ready for a precise statement of the equivalence between uniform generation and approximate counting.

Theorem 6.11 *Let R be a self-reducible p-relation. There is a fully-polynomial randomized approximation scheme for R if and only if there is a fully-polynomial almost uniform generator for R.*

Proof (sketch) $\Rightarrow$ Let Ext_R be the p-relation defined by $\langle x\$w, y\rangle \in \mathrm{Ext}_R$ iff $\langle x, wy\rangle \in R$. $|\mathrm{Ext}_R(x\$w)|$ counts the number of solutions for x that can be obtained by 'completing' the partial solution w. If R is self-reducible, then we can obtain a fpras for Ext_R from a fpras for R as follows: If $|w| > \sigma(x)$, then write $w = uv$, where $|u| = \sigma(x)$. Then $|\mathrm{Ext}_R(x\$w)| = |\mathrm{Ext}_R(\psi(x,u)\$v)|$. Repeating this, we reduce the problem of approximating $|\mathrm{Ext}_R(x\$w)|$ to approximating $|\mathrm{Ext}_R(y\$z)|$ for strings y, z such that $|z| < \sigma(y)$ and $|y| \le |x|$. If we let $m = \sigma(y) - |z|$, this can be computed using the following relationship:

$$|\mathrm{Ext}_R(y\$z)| = \sum_{|u|=m} |\mathrm{Ext}_R(\psi(y,zu)\$e)| = \sum_{|u|=m} |R(\psi(y,zu))|$$

(σ and ψ are the functions from the definition of self-reducible and e is the string of length 0.)

The number of strings u of length m is polynomial in $|x|$ because $\sigma(y) \in O(\log|y|)$. Thus, with polynomially many calls to an approximate counter for R, we obtain an approximate counter for Ext_R. In order to make the probability of success at least 3/4, we need to apply Lemma 6.8 to the approximate counter for R.

Now, to go from an approximate counter for Ext_R to an almost uniform generator for R is quite simple: Given an instance x, we obtain approximations to $n_w = |\mathrm{Ext}_R(x\$w)|$ for all w of length $\sigma(x)$. We then choose the first $\sigma(x)$ symbols of our solution to be v with probability $n_v/(\sum_{|w|=\sigma(x)} n_w)$. We then recursively apply this procedure to generate a solution y for $\psi(x,v)$, and output vy. It is clear that we will be choosing each consecutive set of symbols with approximately the correct probability. Proving that this can be done with the accuracy and probability of success required by the definition involves some technicalities that we omit. The fact that PTM's can only branch on probabilities that are of the form $a2^{-b}$ for integers a and b also introduces some complications that can be dealt with quite easily.

$\Leftarrow$ Here, too, the basic idea is quite simple. By moderately many calls to an almost uniform generator for $R(x)$, we can estimate the fraction of strings in $R(x)$ that begin with each prefix of length $\sigma(x)$. If w, a string of length $\sigma(x)$, appears as a prefix m times out of the n calls to the uniform generator, then $|R(x)|$ is approximately $(n/m)|R(\psi(x,w))|$. By recursively applying this, we obtain an estimate for $|R(x)|$. To obtain maximum possible accuracy, we can use the prefix w that appears most often at each level of recursion. Again, proving that the algorithm gives a correct estimate with sufficiently high probability involves some technicalities that we omit for the sake of clarity. $\square$

Both approximate counting and almost uniform generation problems come up quite often in practice. This theorem tells us that, very often, a solution to one of these problems yields a solution for the other. Similarly, a complexity-theoretic hardness result for one will also give a hardness result for the other. In practice, most of the negative results have come first for approximate counting, whereas most of the positive results have begun with uniform generation. Some of the uniform generation techniques that have been developed were designed to deal with a specific problem. For example, a polynomial-time exactly uniform generator for satisfying assignments to DNF formulæ is given in [JVV86]. Recently, however, most of the successful uniform generators have come from much more general techniques, based on the simulation of Markov chains.

The basic idea behind the Markov chain techniques for almost uniform generation is to do a random walk on the set of solutions we seek to generate from. The walk must have several important properties. First, the set of solutions should have a steady-state probability distribution which is uniform. Second, the walk should converge to this steady-state probability distribution quickly. In particular, we want the probability distribution to be almost uniform (in the sense of Definition 6.9) after only polynomially many steps of the walk. This second condition is called rapid mixing. If these two conditions hold, we obtain a fully-polynomial almost uniform generator by simulating the Markov chain for the requisite number of steps. It is not too difficult to find Markov chains satisfying the first condition for many problems of interest. Proofs of rapid mixing, however, were elusive for all but the simplest of chains until Sinclair and Jerrum's pioneering work in 1989.

In [SJ89], Sinclair and Jerrum relate the convergence of a Markov chain to a simple property of the underlying graph. (We can think of the walk as occurring on a graph whose vertices are the states of the walk and whose edges are the transitions.) They show that a chain is rapidly mixing if and only the 'conductance' of the underlying graph is high. They also present a general technique based on 'canonical paths' for arguing that the conductance of a chain is high. Their techniques have been extremely successful for solving the almost uniform generation and approximate counting analogues of $\#\mathcal{P}$-complete problems. The successes include approximately counting perfect matchings in dense graphs, counting matchings in arbitrary graphs, computing the partition function for the monomer-dimer systems and Ising models of statistical physics, counting the number of graphs with a given degree sequence, evaluating the Tutte polynomial for dense graphs at certain points, and computing the volume of convex sets. [JS89], [Kan94]

These results illustrate that even when exact counting is hard, approximate counting is often easy. It is not true, however, that approximate counting is always easy for p-relations. An obvious example is approximately counting the number of satisfying assignments to a formula in 3-CNF. This is $\mathcal{NP}$-hard, as an approximate count would certainly decide 3SAT.[4]

In contrast with 3SAT, hardness results for approximate counting problems whose related decision problems are not $\mathcal{NP}$-complete are much more difficult to find. In [JVV86], it is shown that approximately counting the number of cycles in a graph is $\mathcal{NP}$-hard. The proof involves a blow-up procedure on the original graph G. This reduction has the property that most of the cycles in the modified graph G' correspond to large cycles in G. Thus, an approximate count to the number of cycles in G' indicates whether G had a cycle of n nodes. Such a cycle is called a Hamiltonian cycle and the detection of these is known to be $\mathcal{NP}$-complete, completing the reduction. A similar approach shows that approximating the number of independent sets in a graph is $\mathcal{NP}$-hard via reduction from the $\mathcal{NP}$-complete problem of determining whether an n-node graph has an independent set of size $n/2$. [Rot95]

[4]Note that the Turing reduction here requires only a single oracle call. Thus, the existence of a fpras for approximately counting satisfying assignments to a 3-CNF formula would imply $\mathcal{NP} \subset \mathcal{BPP}$.

However, in many cases, there is not such a closely related $\mathcal{NP}$-complete problem, explaining the lack of hardness results for approximate counting. Moreover, these types of reductions tend to lose restrictions on the problem reduced from. For example, determining whether a graph has an independent set of size $n/2$ remains hard for graphs of degree 3, but the reduction in [Rot95] produces a graph of unbounded degree and thus does not prove that approximating the number of independent sets in bounded-degree graphs is hard. In the next section, we overcome this difficulty for one of the problems considered earlier, showing that counting minimum cardinality vertex covers remains $\mathcal{NP}$-hard even for graphs of degree 3.

Chapter 7

The Complexity of Approximately Counting Vertex Covers

As noted earlier, there are relatively few results about the hardness of approximate counting, and even fewer that apply to restricted cases. Roth has shown that, for every $\epsilon > 0$, it is $\mathcal{NP}$-hard to approximate the number of vertex covers (equivalently, independent sets) in an n-node graph within a ratio of $2^{n^{1-\epsilon}}$[Rot95]. This is a very strong result, as it says there is no randomized algorithm that can approximately count vertex covers within any subexponential ratio unless $\mathcal{NP} \subset \mathcal{BPP}$. In particular, there can be no fully-polynomial randomized approximation scheme for counting vertex covers in general. Using reduction 11 from Theorem 5.2, we can obtain analogous hardness results for approximately counting minimal vertex covers and minimum cardinality vertex covers. Roth's result, however, does not tell us anything about the difficulty of these counting problems if we restrict to bipartite or bounded-degree graphs.

In this chapter, we prove that it remains $\mathcal{NP}$-hard to approximately count minimum cardinality vertex covers within a factor of $2^{n^{1-\epsilon}}$ even in graphs of degree 3. In order to obtain this restricted-case result, we go through a somewhat indirect sequence of reductions. Ideally, we would be able to use something like reduction 10 from Theorem 5.2 to obtain a corollary that approximately counting vertex covers is hard in graphs of degree 4. Unfortunately, that reduction is far from parsimonious. It involves evaluating a polynomial, one of whose coefficients is the number of minimum cardinality vertex covers, at a sequence of points that converge exponentially fast by calls to an oracle that counts vertex covers. Thus, even if we could obtain approximate evaluations to the polynomial via an approximate-counting oracle for vertex covers, we are not guaranteed to obtain good approximations to the coefficients. Regardless, the hardness result we have is the first of its kind for such a restricted case. Perhaps it can be used to show that other approximation problems are hard under similar restrictions, or maybe the techniques themselves will prove useful.

Theorem 7.1 *The following are $\mathcal{NP}$-hard (3-7 are for every $\epsilon > 0$):*

1. $\frac{n}{2}$-VERTEX COVER

   Input: An undirected graph G on n nodes.

   Output: Y if G has a vertex cover of size $\frac{n}{2}$, N otherwise.

2. $\frac{n}{2}$-MONOTONE 2SAT

   Input: A monotone 2-CNF formula F on n variables.

   Output: Y if F has a satisfying assignment with exactly $\frac{n}{2}$ variables true, N otherwise.

3. $3^{r^{1-\epsilon}}$-APPROX #GUARANTEED WEIGHTED 3SAT

   Input: A formula $F = C_1 \wedge C_2 \wedge \cdots \wedge C_r$, where $C_i = (c_{i,1} \vee c_{i,2})$ or $C_i = (c_{i,1} \vee c_{i,2} \vee c_{i,3})$, on m variables, and an assignment β which satisfies F.

   Output: An approximation to
   $|\{(\alpha, t) : t \in \{1,2,3\}^r, \alpha \text{ an assignment for } F \text{ such that } \alpha \text{ makes } c_{i,t_i} \text{ true for each } i\}|$ *within ratio $3^{r^{1-\epsilon}}$.*

4. $3^{(\frac{2m+3r}{18})^{1-\epsilon}}$-APPROX #$2\nu$-GUARANTEED WEIGHTED 3SAT

   Input: As in 3, but with each literal occurring at most twice.

   Output: An approximation to the quantity in 3 within ratio $3^{(\frac{2m+3r}{18})^{1-\epsilon}}$.

5. $3^{(\frac{n}{18})^{1-\epsilon}}$-APPROX #$3\mu$-MINIMUM CARDINALITY VERTEX COVER

   Input: An undirected graph G of degree ≤ 3 on n nodes.

   Output: An approximation to the number of minimum cardinality vertex covers in G within ratio $3^{(\frac{n}{18})^{1-\epsilon}}$.

6. $2^{n^{1-\epsilon}}$-APPROX #3μ-MINIMUM CARDINALITY VERTEX COVER

   Input: An undirected graph G of degree ≤ 3 on n nodes.

   Output: An approximation to the number of minimum cardinality vertex covers in G within ratio $2^{n^{1-\epsilon}}$.

7. $2^{n^{1-\epsilon}}$-APPROX #3μ-MAXIMUM CARDINALITY INDEPENDENT SET

Input: *An undirected graph G of degree ≤ 3 on n nodes.*

Output: *An approximation to the number of maximum cardinality independent sets in G within ratio $2^{n^{1-\epsilon}}$.*

Proof 1. 3SAT $\propto$ $\frac{n}{2}$-VERTEX COVER

The standard reduction from 3SAT to VERTEX COVER takes a formula F in 3-CNF with m variables and r clauses and constructs a graph G on $2m+3r$ vertices such that G has a vertex cover of size $m+2r$ if and only if F is satisfiable. [GJ79] If we add r isolated vertices to G, then G has $n = 2m+4r$ vertices and has a vertex cover of size $n/2$ iff F is satisfiable.

2. $\frac{n}{2}$-VERTEX COVER $\propto$ $\frac{n}{2}$-MONOTONE 2SAT

This is immediate, given the discussion in the proof of Proposition 4.4.

3. $\frac{n}{2}$-MONOTONE 2SAT $\propto 3^{r^{1-\epsilon}}$-APPROX #GUARANTEED WEIGHTED 3SAT

Let F be the given monotone 2-CNF formula, with r clauses. Intuitively, we want to magnify the number of choices of t for assignments in which exactly half the variables are true. Notice that the number of pairs of variables which have different values is largest when exactly half the variables are true. This is the basis of our reduction. Say F has variables $x_1, x_2, \ldots, x_n$. For each pair i, j with $i \neq j$, introduce a new variable z_{ij}, and define F_{ij} as follows.

$$\begin{aligned} F_{ij} &= (x_i \vee x_j \vee z_{ij}) \wedge (\neg x_i \vee \neg x_j \vee z_{ij}) \\ &\quad \wedge \underbrace{(\neg z_{ij} \vee \neg z_{ij} \vee z_{ij}) \wedge (\neg z_{ij} \vee \neg z_{ij} \vee z_{ij}) \wedge \cdots \wedge (\neg z_{ij} \vee \neg z_{ij} \vee z_{ij})}_{s \text{ times}} \end{aligned}$$

Now we define G as follows:

$$G = F \wedge \left(\bigwedge_{i \neq j} F_{ij} \right)$$

Notice that the assignment β which sets all variables to true can be used as the satisfying assignment in the input. Consider any assignment α that satisfies F. Let us consider the number of ways we can extend α to a satisfying assignment of G and choose $t \in \{1, 2, 3\}^{r+n(n-1)(s+2)}$ so that the required relationship holds. For each pair of variables x_i, x_j which take on different values, z_{ij} can either be true or false. When z_{ij} is true, there are 4 ways to choose the components of t corresponding to the first two clauses of F_{ij} (in each of those, two literals are true.) But there is only one way to choose the components of t corresponding to the final s clauses in F_{ij}. When z_{ij} is chosen to be false, then there is only one way to choose the components of t for the first 2 clauses in F_{ij}, but there are 2^s ways to do so for the final s clauses. Thus, for each pair of variables taking different values under α, there are $4 + 2^s$ choices for z_{ij} and the relevant components of t.

For each pair of variables x_i, x_j which take on the same value under α, z_{ij} must be true to satisfy F_{ij}. Then there are 3 choices of the component of t corresponding one of the first two clauses of F_{ij}, but only one choice for the other clauses of F_{ij}. Thus, for each pair of variables taking on the same value under α, there are 3 choices for z_{ij} and the relevant components of t.

Let $m = \frac{n}{2}$. Suppose F has a satisfying assignment α in which exactly m variables are true. Then there are at least m^2 pairs of variables that take on different values under α. So there are at least $N_{min} = (4+2^s)^{m^2}$ choices of (β, t) to count for G.

Now suppose that no satisfying assignment of F has exactly m variables true. Then, for each assignment α that satisfies F, there are at most $m^2 - 1$ pairs of variables taking on different values under and at most n^2 pairs of variables taking on the same value under α. Additionally, there are at most 2^r choices of the components of t for the original r clauses in F, so there are at most $(2^r)(4+2^s)^{m^2-1}(3^{n^2}) = N_{max}$ choices of (β, t) to count for G.

Given $0 < \epsilon < 1$, choose

$$s > \left[(\log_2 3)\left(n^2 + 2(r + 3n(n-1))^{1-\epsilon}\right) + r\right]^{1/\epsilon}$$

Then,

$$\begin{aligned} s &> s^{1-\epsilon}\left[(\log_2 3)\left(n^2 + 2(r + 3n(n-1))^{1-\epsilon}\right) + r\right] \\ &\geq (\log_2 3)\left(n^2 + 2(r + n(n-1)(s+2))^{1-\epsilon}\right) + r \\ 4 + 2^s &> \left(3^{n^2 + 2(r+n(n-1)(s+2))^{1-\epsilon}}\right) 2^r \end{aligned}$$

$$\frac{N_{min}}{N_{max}} = \frac{4+2^s}{2^r 3^{n^2}} > \left(3^{(r+n(n-1)(s+2))^{1-\epsilon}}\right)^2 = \left(3^{z^{1-\epsilon}}\right)^2$$

where z is the number of clauses in G. Say γ is an approximation to the number N of (β, t) that work for G within ratio $\delta = 3^{z^{1-\epsilon}}$. If F has a satisfying assignment with $\frac{n}{2}$ variables true, then $N \geq N_{min} \Rightarrow \gamma \geq N_{min}/\delta > N_{max}\delta$. If F doesn't have a satisfying assignment with $\frac{n}{2}$ variables true, then $N \leq N_{max} \Rightarrow \gamma \leq N_{max}\delta$. Hence, by comparing γ with $N_{max}\delta$, we can decide if N has a satisfying assignment of size $\frac{n}{2}$.

4. $3^{r^{1-\epsilon}}$-Approx #Guaranteed Weighted 3Sat $\propto$
$3^{(\frac{2m+3r}{18})^{1-\epsilon}}$-Approx #$2\nu$-Guaranteed Weighted 3Sat
Reduction 8, from the proof of Theorem 5.2 works here (if we ignore the bipartite conditions on the formulæ), because it is a parsimonious reduction. We simply need to note that if the original formula F has r clauses, then the formula G constructed in the reduction has at most $3r$ variables and $4r$ clauses. Thus, if G has m variables and s clauses, an approximation within ratio $3^{(\frac{2m+3s}{18})^{1-\epsilon}}$ for G yields an approximation within ratio $3^{r^{1-\epsilon}}$ for F.

5. $3^{(\frac{2m+3r}{18})^{1-\epsilon}}$-Approx 2ν-#Guaranteed Weighted 3Sat $\propto$
$3^{(\frac{n}{18})^{1-\epsilon}}$-Approx 3μ-#Minimum Cardinality Vertex Cover
This is similar to reduction 9 in the proof of Theorem 5.2. Given F, we construct a graph with $n + r_1$ dumbbells and r_2 triangles, where n is the number of variables in F, r_1 is the number of clauses of size 2, and r_2 is the number of clauses of size 3. Specifically, for each variable x, we have two nodes, x and $\neg x$, which are connected by an edge. For each clause $(c_{i,1} \vee c_{i,2})$, we have a dumbbell and we connect the endpoints to the nodes corresponding to the literals $c_{i,1}$ and $c_{i,2}$. For each clause $(c_{i,1} \vee c_{i,2} \vee c_{i,3})$, we have a triangle and connect the vertices to the nodes corresponding to $c_{i,1}$, $c_{i,2}$, and $c_{i,3}$.

Formally, if F is over variables $\{x_1, \ldots, x_n\}$, let $G = (V, E)$, where

$$
\begin{aligned}
V &= \{x_i, \neg x_i : 1 \leq i \leq n\} \cup \{(i,j) : |C_i| \geq j, 1 \leq i \leq r, 1 \leq j \leq 3\} \\
E &= \{\langle x_i, \neg x_i\rangle : q \leq i \leq n\} \\
&\cup \ \{\langle (i,1), (i,2)\rangle : 1 \leq i \leq r\} \cup \{\langle (i,1), (i,3)\rangle, \langle (i,2,), (i,3)\rangle : |C_i| = 3, 1 \leq i \leq 3\} \\
&\cup \ \{\langle c_{i,j}, (i,j)\rangle : |C_i| \geq j, 1 \leq i \leq r, 1 \leq j \leq 3\}
\end{aligned}
$$

Observe that every vertex cover in G is of size at least $n + r_1 + 2r_2$ We will show that there is exactly one such cover for every choice of (α, t) for F. Given (α, t) satisfying the appropriate conditions, let

$$V' = \{x : \alpha(x) = 1\} \cup \{\neg x : \alpha(x) = 0\} \cup \{(i,j) : t_i \neq j, |C_i| \geq j\}$$

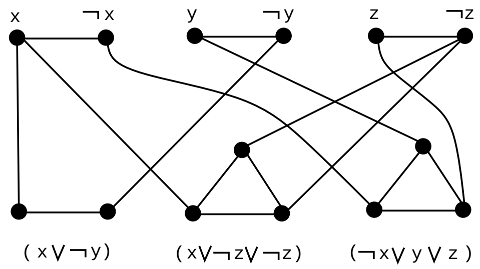

Figure 7.1: The graph produced for $F = (x \vee \neg y) \wedge (x \vee \neg z \vee \neg z) \wedge (\neg x \vee y \vee z)$ in reduction 5

V' is a vertex cover of size $n + r_1 + 2r_2$. It is clear that the first two types of edges are covered is clear. To see that the edges of the last type are covered, just note that if $c_{i,j}$ is not in the cover, then $c_{i,j}$ is not true, so $t_i \neq j$, which means that (i, j) is in the cover. There is at least one cover of this size, since F is satisfiable. Additionally, every vertex cover V' of size $n + r_1 + 2r_2$ must contain exactly one of each pair $x_i, \neg x_i$ and must be missing exactly one vertex (i, j) for each j. Thus, if we set $\alpha(x_i) = 1$ iff $x_i \in V'$ and $t_i = j$ iff $(i, j) \notin V'$, we obtain a correct pair (α, t). (Note that if $t_i = j$, then $(i, j) \notin V'$, so $c_{i,j} \in V'$, so $\alpha(c_{i,j}) = 1$, as needed.)

Finally, observe that each 'clause' node has degree ≤ 3, whereas each 'literal' node has degree one greater than the number of times it appears in F. Hence, the graph has degree at most 3, and this is a parsimonious reduction. The number of nodes n in G is at most $2m + 3r$, where m is the number of variables in F and r is the number of clauses, so an approximation within ratio $3^{(\frac{n}{18})^{1-\epsilon}}$ for G gives an approximation within ratio $3^{(\frac{2m+3r}{18})^{1-\epsilon}}$ for F.

6. $3^{(\frac{n}{18})^{1-\epsilon/2}}$-Approx #3$\mu$-Minimum Cardinality Vertex Cover $\propto$ $2^{n^{1-\epsilon}}$-Approx #3μ-Minimum Cardinality Vertex Cover

Let $\epsilon > 0$ be fixed. Then, for all n greater than some sufficiently large N, $n^{1-\epsilon} \leq \left(\frac{\log_2 3}{18^{1-\epsilon/2}}\right) n^{1-\epsilon/2}$, *i.e.* $2^{n^{1-\epsilon}} \leq 3^{(\frac{n}{18})^{1-\epsilon/2}}$. Hence, given an algorithm that approximates the number of minimum cardinality vertex covers within ratio $2^{n^{1-\epsilon}}$, we can obtain one that approximates the same quantity within ratio $3^{(\frac{n}{18})^{1-\epsilon/2}}$, by keeping a table of the number of minimum cardinality vertex covers in the (finitely many) graphs with fewer than N nodes.

7. $2^{n^{1-\epsilon}}$-Approx #3μ-Minimum Cardinality Vertex Cover $\propto$

$2^{n^{1-\epsilon}}$-Approx #3μ-Maximum Cardinality Independent Set
Immediate from Proposition 4.4. $\square$

Chapter 8

Conclusion

As we have seen, the study of counting and its computational complexity is both interesting and important. However, our understanding of the computational aspects of counting is far from complete. In particular, very little is known about counting, both exact and approximate, in restricted cases. In this thesis, we have answered a few questions about counting in restricted cases, but there are still many open problems.

A few unanswered questions related to the ones we have studied here stand out. Specifically, for vertex covers and minimal vertex covers, there are no results analogous to what we have shown about approximately counting minimum cardinality vertex covers in sparse graphs. Nothing is known about the complexity of any of these three approximation problems in bipartite graphs, either. Similarly, it is unknown whether approximation of the permanent for arbitrary graphs is tractable, although, as noted, Markov chain algorithms have been successful in certain restricted cases. It would be foolhardy to conjecture that these problems are all hard simply because the somewhat related problem of counting minimum cardinality vertex covers is. The Markov chain technique described in Chapter 6 has been successful in some surprising situations; there is no reason to believe that it cannot work here. At the same time, however, computational intractability is still a reasonable possibility, so hardness results should also be sought. Perhaps the techniques and results introduced in this thesis will help answer some of these and the many other unanswered questions about the complexity of counting.

Bibliography

[DF89] M. E. Dyer and A. M. Frieze. On the complexity of computing the volume of a polyhedron. *Society for Industrial and Applied Mathematics Journal on Computing*, 18:205–226, 1989.

[DL92] Paul Dagum and Michael Luby. Approximating the permanent of graphs with large factors. *Theoretical Computer Science*, 102:283–305, 1992.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[Gol93] Leslie Ann Goldberg. Automating polya theory: The computational complexity of the cycle index polynomial. *Information and Computation*, 105:268–288, 1993.

[JS89] Mark Jerrum and Alistair Sinclair. Approximating the permanent. *Society for Industrial and Applied Mathematics Journal on Computing*, 18(6):1149–1178, December 1989.

[JVV86] Mark R. Jerrum, Leslie G. Valiant, and Vijay V. Vazirani. Random generation of combinatorial structures from a uniform distribution. *Theoretical Computer Science*, 43:169–188, 1986.

[Kan94] Ravi Kannan. Markov chains and polynomial time algorithms. In *Annual Symposium on the Foundations of Computer Science*, pages 656–669. Institute of Electrical and Electronics Engineers, 1994.

[LP81] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*, chapter 7, pages 311–371. Prentice-Hall, Inc., 1981.

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[PB83] J. Scott Provan and Michael O. Ball. The complexity of counting cuts and of computing the probability that a graph is connected. *Society for Industrial and Applied Mathematics Journal on Computing*, 12(4):777–788, 1983.

[Rot95] Dan Roth. On the hardness of approximate reasoning. *Artificial Intelligence Journal*, 1995. To appear.

[SJ89] Alistair Sinclair and Mark Jerrum. Approximate counting, uniform generation, and rapidly mixing markov chains. *Information and Computation*, 82:93–133, 1989.

[Sto83] Larry Stockmeyer. The complexity of approximate counting. In *Proceedings of the 15th Annual ACM Symposium on the Theory of Computing*, pages 118–126. Association for Computing Machinery, Inc., 1983.

[Val79a] Leslie G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8:189–201, 1979.

[Val79b] Leslie G. Valiant. The complexity of enumeration and reliability problems. *Society for Industrial and Applied Mathematics Journal on Computing*, 8(3):410–421, August 1979.