Exact Counting is as Easy as Approximate Counting

Jin-yi Cai†
Lane A. Hemachandra‡

TR 86-761
June 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853

† Research supported by a Sage Fellowship and NSF grant DCR-8301766.
‡ Research supported by a Fannie and John Hertz Foundation Fellowship and NSF grant DCR-8301766.

Exact Counting is as Easy as Approximate Counting

Jin-yi Cai†
Lane A. Hemachandra‡

TR 86-761
June 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853

† Research supported by a Sage Fellowship and NSF grant DCR-8301766.
‡ Research supported by a Fannie and John Hertz Foundation Fellowship and NSF grant DCR-8301766.

Exact Counting is as Easy as Approximate Counting

Jin-yi Cai[*] *Lane A. Hemachandra*[†]

Department of Computer Science
Cornell University
Ithaca, NY 14853

Abstract

We show that exact counting and approximate counting are polynomially equivalent. That is,

$$\mathrm{P}^{\#\mathrm{P}} = \mathrm{P}^{\mathrm{Approx}\#\mathrm{P}},$$

where #P is a function that computes the number of solutions to a given Boolean formula f (denoted by $\|f\|$), and Approx#P computes a short ***list*** that contains $\|f\|$.

It follows that if there is a good polynomial time approximator for #P (i.e., one where the list has at most $O(|f|^{1-\epsilon})$ elements), then $\mathrm{P} = \mathrm{NP} = \mathrm{P}^{\#\mathrm{P}}$ and probabilistic polynomial time equals polynomial time. Thus we have strong evidence that #P can not be easily approximated.

1 Prelude

Suppose a demon comes into your office and says, "I have here a polynomial time machine. Give it a Boolean formula and it will print the number of satisfying assignments the formula has." Enraged, you reply, "Your claim implies P=NP, false and foul fiend. Return to your infernal home." Her deception discovered, the embarrassed demon (who ***knows*** that $\mathrm{P} \neq \mathrm{NP}$) leaves you in a cloud of smoke.

However, she returns the next day and says, "I got the last bug out of that polynomial time machine. Give it a formula and it will now print out a short list of numbers, one of

[*]Research supported by a Sage Fellowship and NSF grant DCR-8301766.
[†]Research supported by a Fannie and John Hertz Foundation Fellowship and NSF grant DCR-8301766.

which is the number of satisfying assignments the formula has." You are intrigued. Just what is this demon offering? You suddenly remember reading this paper. Without the slightest hesitation you reply, "Your claim *still* implies P=NP, dishonest demon. Return to the abyss, and trouble me no more." Your soul secure, you return to the search for a proof that P $\neq$ NP.

2 Introduction

#P is the class of functions that count the accepting paths of some NP machine [Val79b]. These functions can count the number of cliques of a given size, compute the permanent of a matrix and solve many other interesting counting versions of NP problems [Val79b,Val79a]. $\mathrm{P}^{\#\mathrm{P}}$ is the class of languages computable by polynomial time machines endowed with the power of counting.

#P questions seem hard. Though they can be answered by brute force in PSPACE, it is not known if they are in the polynomial hierarchy. Currently known structural relations are that $\mathrm{P}^{\#\mathrm{P}} \supseteq \Delta_2^p$, and, with an oracle, $\mathrm{P}^{\#\mathrm{P}} \supset \Sigma_2^p \cup \Pi_2^p$ [Ang80]. The apparent difficulty of #P problems motivated our research on the approximation of #P.

#P is usually approximated by considering #SAT. Cook's reduction [Coo71,HU79] from NP machines to formulas is easily shown parsimonious [Sim77,Val79b]. Thus #SAT, the function mapping from Boolean formulas to their numbers of solutions (e.g., #SAT($x_1 \vee x_2) = 3$), is a canonical hardest #P function. We speak interchangeably of approximating #P and #SAT, as #SAT can be approximated iff #P can.

Stockmeyer [Sto85] has analyzed the complexity of $r(\cdot)$-bracketing #SAT in the sense of bracketing the value within a multiplicative factor: finding a function g so that:

$$\frac{\|f\|}{r(|f|)} \leq g(|f|) \leq \|f\|\, r(|f|).$$

He shows that in Δ_3^p we can $(1+\epsilon|f|^{-d})$-bracket #SAT. Note that for formulas with *many* solutions, this bound is weak — the size of the set of possible values it predicts may be exponentially large. [Sto85] also shows that there is a relativized world where for no constant k can #SAT be k-bracketed even with a Δ_2^p function.

The lower bounds of [Sto85] do not prove that #SAT is hard to approximate. They only show that computer scientists lamentably lack the mathematical tools needed to determine whether #SAT is easy to approximate. In this paper, we present a more aggressive approximation that reduces the number of possible values of $\|f\|$ to a polynomial sized set, and show that the existence of such approximators implies $\mathrm{P} = \mathrm{NP} = \mathrm{PP} = \mathrm{P}^{\#\mathrm{P}}$. Thus we may believe that #SAT *is* hard to approximate with at least the certainty with which we believe $\mathrm{P} \neq \mathrm{NP}$ — a belief we fervently hold from intuition and evidence wholly independent of the arcane theory of relativization [CH86,HH86b].

A function A is said to *$s(\cdot)$-approximate* #SAT if $A(f)$ is a polynomial sized list of at most $s(|f|)$ integers in which $\|f\|$ appears. If A can be computed in polynomial time, we call it an *$s(\cdot)$-P-approximator* for #SAT. For example, a 4-P-approximator given $x_1 \vee x_2 \vee x_3$ might reply $\{0, 1, 4, 7\}$.

Section 3 introduces our proof techniques in a simple setting. If #SAT has a k-P-approximator then P = NP. Section 4 extends this result to show that if Approx#P is a function n^α-approximating #SAT, $\alpha < 1$, then $\mathrm{P}^{\#\mathrm{P}} = \mathrm{P}^{\mathrm{Approx}\#\mathrm{P}}$. Thus approximate counting and exact counting are polynomially equivalent.

These results demonstrate that efficiently approximating #P brings on the apocalypse — P = NP. Thus we have strong structural evidence that #P can not be approximated.

3 A Simple Proof that P=NP

The proofs of this section and Section 4 have the same architecture. Using a technique for combining formulas so that their individual numbers of solutions can be deciphered from that of the combined formula, we repeatedly expand and prune a formula tree. Here we keep the tree constantly thin. Section 4 allows trees that are polynomially bushy.

This section shows that if #SAT can be k-approximated then P = NP. First, we state a lemma about combining formulas. Lemma 3.1 says we can easily combine many small formulas into a single larger formula. This big formula has the property that, given its number of solutions, we can quickly determine the number of solutions of each of the small formulas.

Lemma 3.1 *There are polynomial time functions combiner and decoder such that for any Boolean formulas f and g, combiner(f, g) is a Boolean formula and decoder($\|$combiner(f,g)$\|$ prints $\|f\|$, $\|g\|$.*

Proof Sketch Let $f = f(x_1, \ldots, x_n)$ and $g = g(y_1, \ldots, y_m)$, where $x_1, \ldots, x_n$, $y_1, \ldots, y_m$ are distinct. Let z and z' be two new Boolean variables. Then

$$h = (f \wedge z) \vee (\bar{z} \wedge x_1 \wedge \cdots \wedge x_n \wedge g \wedge z')$$

is the desired combination, since $\|h\| = \|f\| 2^{m+1} + \|g\|$ and $\|g\| \leq 2^m$. **QED**

We can easily extend this technique to combine more than two formulas. For example, to combine three formulas $f(x_1, \ldots, x_l)$, $g(y_1, \ldots, y_m)$, $h(z_1, \ldots, z_n)$, our combining formula would be,

$$h = (f \wedge z) \vee \left\{\bar{z} \wedge x_1 \wedge \cdots \wedge x_l \wedge \left[(g \wedge z') \vee \left(\bar{z'} \wedge y_1 \wedge \cdots \wedge y_m \wedge h \wedge z''\right)\right]\right\}.$$

Now we show that if #SAT has a k–P–approximator then P = NP.

Theorem 3.2 *If* #SAT *can be* k*–*P*–approximated then* P = NP.

Proof Say we are given a formula $F(x_1, \ldots, x_n)$ and we would like to know if $F \in$ SAT. We substitute variables one at a time so that we always have a set S of at most k partial assignments satisfying:

($\star$) $F \in$ SAT $\iff$ there is a satisfying assignment consistent with a partial assignment in S.

Each stage assigns a new variable and has three steps. Initially, S consists of the empty assignment.

Stage i:

1. EXPAND TREE: For each partial assignment in S, assign the variable x_i both true and false (Figure 1A). Applying these assignments to F, we have at most $2k$ formulas of which, if $F \in$ SAT, at least one is satisfiable.

2. COMBINE FORMULAS and RUN APPROXIMATOR: Combine the $2k$ formulas into a single super-formula as described in Lemma 3.1. Run our k–P–approximator on that super-formula. The approximator prints k guesses for the number of solutions of the super-formula. For example, in Figure 1B (where $k = 3$) the first guess says that the four little formulas in our super-formula have, respectively, $7, 0, 3,$ and 3 solutions.

3. PRUNE THE TREE: If these k guesses are all zero vectors then the formula is unsatisfiable. Otherwise, we can choose a set T of at most k columns so that each nonzero row of our guess matrix (Figure 1B) has a nonzero in a column in T. For example, we let $T = \{p \mid$ a row of the guess matrix has its first nonzero entry in column $p\}$.

   Now we prune by setting S to the partial assignments corresponding to the columns in T (there are at most k). Suppose $F \in$ SAT. Then by the inductive hypothesis one of the predecessors is satisfiable. Thus by our choice of T the true guess has a nonzero in some column of T. We have assigned another variable and maintained invariant $(\star)$.

End of Stage i.

At the final stage all variables are assigned and we just have to look at our set of k complete assignments to F and see if any of them satisfies F. We know by $(\star)$ that F is satisfiable if and only if one of these assignments satisfies F. **QED**

4 Main Result

This section demonstrates that approximate and exact counting are polynomially equivalent. We dramatically strengthen the result of the previous section. From a weaker assumption we draw a stronger conclusion.

Theorem 4.1 *If* Approx#P *is an n^α-approximator for* #SAT, $\alpha < 1$, *then*

$$\mathrm{P}^{\#\mathrm{P}} = \mathrm{P}^{\mathrm{Approx}\#\mathrm{P}}.$$

Corollary 4.2 *If* #SAT *can be n^α-*P*-approximated,* $\alpha < 1$, *then* $\mathrm{P} = \mathrm{P}^{\#\mathrm{P}}$.

Since $\mathrm{P}^{\#\mathrm{P}} = \mathrm{P}^{\mathrm{PP}}$ [Gil77,Sim75], where PP is probabilistic polynomial time, the existence of good approximators for #SAT also implies that probabilistic and deterministic polynomial time are equivalent.

Corollary 4.3 *If* #SAT *can be n^α-*P*-approximated,* $\alpha < 1$, *then* P = PP.

Theorem 4.1 differs from Theorem 3.2 in two important ways. One is that we are satisfied with an n^α-approximator, $\alpha < 1$. The more interesting point is that we conclude $\mathrm{P} = \mathrm{P}^{\#\mathrm{P}}$. We now discuss each of these improvements.

4.1 How to Count Solutions

The first major change is that we find out not only if a formula is satisfiable, but also how many satisfying assignments it has. We do this with a more rigorous analysis of the guess matrix and a refined pruning scheme.

Lemma 4.4 *If* #SAT *can be k-*P*-approximated then* $\mathrm{P} = \mathrm{P}^{\#\mathrm{P}}$.

Proof Consider the tree pruning procedure in Theorem 3.2. Here we want to keep a set S of p $(1 \le p \le k)$ leaves in the partially grown tree, such that $\langle \|f_1\|, \|f_2\|, \ldots, \|f_p\| \rangle$ uniquely determine $\|f\|$, where f_j is the formula obtained from f by the partial assignment associated with the jth leave in S. (We will speak interchangeably of the jth leave in S and f_j.)

Again we substitute variables one at a time. Inductively, for the formulas $f_1, \ldots, f_p$ in S, we wish to maintain at most k vectors u_i $(i = 1, \ldots, q,\ q \le k)$ of dimension p, and integers $s_1, \ldots, s_q$, such that the following conditions hold.

1° $(\forall i)[u_i \neq 0]$,

2° $(\forall i \neq j)[u_i \neq u_j]$, and

3° $f \in \mathrm{SAT} \Longrightarrow (\exists i)[u_i = \langle \|f_1\|, \ldots, \|f_p\| \rangle \;\&\; s_i = \|f\|]$.

Notice that when the tree has fully grown, for the formulas $f_1, \ldots, f_p$ in S it can be easily checked whether some $u_i = \langle \|f_1\|, \ldots, \|f_p\| \rangle$. If such u_i exists, it must be unique, by condition 2°. If $f \in \mathrm{SAT}$ we are guaranteed such a u_i, and we can output $\|f\| = s_i$, by 3°. Otherwise f is unsatisfiable and then $\langle \|f_1, \ldots, \|f_p\| \rangle = 0$, and must be unequal to any u_i, by 1°, in which case we output $\|f\| = 0$.

At heart, the proof is a double induction. Each stage has the same general structure as before in the proof of Theorem 3.2.

Initially S consists of f and we apply our approximation on f. If the approximator guessed all zeros, then output $\|f\| = 0$. Otherwise, let $u_1, \ldots, u_q$ be all the distinct nonzero guesses $(1 \leq q \leq k)$.

We inductively maintain 1°, 2° and 3° as we go along, and at each stage we use a second induction for the tree pruning process.

Stage l:

1. EXPAND TREE: For each partial assignment in S, assign the variable x_l both true and false. We have r new leaves, where $2 \leq r = 2|S| \leq 2k$.

2. COMBINE FORMULAS and RUN APPROXIMATOR: Combine f with the formulas $f_1, \ldots, f_r$ associated with these new leaves. Let G be the resulting "superformula." Run our k-approximator of G. We obtain k guesses for the number of solutions of G. Using our decoder (see Section 4.2) we get up to k distinct vectors, say $\widehat{v_1}, \ldots, \widehat{v_{q'}}$, $1 \leq q' \leq k$, where $\widehat{v_i} = \langle v_{i0}, v_{i1}, \ldots, v_{ir} \rangle$ is a guess for $\langle \|f\|, \|f_1\|, \ldots, \|f_r\| \rangle$.

3. PRUNE THE TREE: Let $v_i = \langle v_{i1}, \ldots, v_{ir} \rangle$.

   If some $v_i = 0$ we may discard $\widehat{v_i}$. In fact, if $f \in \mathrm{SAT}$ then one of the formulas in S is satisfiable, by inductive hypotheses 3° and 1°. Thus one of the new leaves is satisfiable. Since $v_i = 0$ can't be a true guess if $f \in \mathrm{SAT}$, deleting it causes no harm.

Secondly, for any pair $\widehat{v_i}$, $\widehat{v_j}$, if $v_i = v_j$, we can effectively delete at least one of them. This is because $\widehat{v_i}, \neq \widehat{v_j}$ implies $v_{i0} \neq v_{j0}$. Now $\langle v_{i1} + v_{i2}, \ldots, v_{i\,r-1} + v_{ir} \rangle$ must equal one of the u's (call it u_t) associated with the formulas in S (otherwise $\widehat{v_i}$ as well as $\widehat{v_j}$ is clearly false by 3°). This u_t must be unique by 2°; furthermore, either $v_{i0} \neq s_t$ or $v_{j0} \neq s_t$.

If $v_{i0} \neq s_t$, clearly $\widehat{v_i}$ is false; we may delete $\widehat{v_i}$. The same argument applies to $\widehat{v_j}$. Without loss of generality, we are left with $\langle \widehat{v_1}, \ldots, \widehat{v_q} \rangle$, $q \leq k$. If $q = 0$ then output $\|f\| = 0$. In fact, if $f \in$ SAT then some $\widehat{v_i}$ with $v_i \neq 0$ must represent the truth and must have been kept.

Let $s_i = v_{i0}$, $i = 1, \ldots, q$, $1 \leq q \leq k$. Note that $v_1, \ldots, v_q$ and the s_i's satisfy conditions 1°, 2° and 3°. Let the guess matrix consist of $v_1, \ldots, v_q$ as row vectors. We will inductively extract at most q columns of the matrix, so that the q row vectors of the submatrix also satisfy 1°, 2° and 3°.

Since 3° is automatically satisfied with any subset of columns, we need only to maintain 1° and 2°.

To prune, initially let $j_1 = \min\{j \geq 1 \,|\, v_{1j} \neq 0\}$. Let $w_1 = (v_{1j_1}) \in Z^1$. Inductively, suppose $w_1, \ldots, w_h \in Z^{h'}$ have been constructed, $h' \leq h < q$, satisfying 1° and 2°. Each w_i is the projection of v_i onto the h' chosen columns. Take these h' columns of v_{h+1}; call this vector $\widetilde{w_{h+1}}$.

If $\widetilde{w_{h+1}} = w_i$ for some $1 \leq i \leq h$, then this i is unique, by 2°. Also, $\widetilde{w_{h+1}} \neq 0$ by 1°. Now all we need to do is to distinguish w_{h+1} from w_i. But since $v_{h+1} \neq v_i$, this is easily done by choosing one more column. (Every $w_1, \ldots, w_h$, $\widetilde{w_{h+1}}$ is extended one dimension.)

If $\widetilde{w_{h+1}} \neq w_i$, for all $1 \leq i \leq h$, then we only need to insure $\widetilde{w_{h+1}} \neq 0$. Again this is easy since $v_{h+1} \neq 0$.

Finally, $w_1, \ldots, w_q$ are constructed. Set u_i to w_i and p to dimension(w_i); S consists of those new leaves corresponding to the p selected columns. 1°, 2° and 3° are satisfied.

End of Stage l.

QED

4.2 Dealing With Polynomial Approximators

This section notes that we can combine many formulas into a super-formula efficiently and can still prune so that our tree does not blow up in size. This is really just an extension of the way, in Section 3, we went from combining two to combining three formulas.

We briefly discuss how to proceed with an $n^{1-\epsilon}$–approximator. We maintain a polynomially wide band as we prune down the tree. For any $\alpha < 1$, we can maintain a bushy tree of width N^t, where t is a constant depending on α. A careful analysis reveals that, counting the cost of making all variables distinct, applying an n^α–approximator to a super-formula made from $2N^t + 1$ formulas each of length N produces less than N^t guesses.

5 Open Questions and Conclusions

We've seen that if #SAT can be n^α–approximated, $\alpha < 1$, apocalypse results. One wonders if n^α–approximability, for arbitrary α, also has earthshaking consequences. We present a relativized answer. The following theorem is easily proved using the relativization techniques of [Sto85].

Theorem 5.1 *There is a relativized world in which no Δ_2^p function can, for any k, n^k-approximate* #SAT.

We conjecture that even n^k–P–approximability implies $\mathrm{P} = \mathrm{P}^{\#\mathrm{P}}$.

Our evidence on the structural consequences approximating #P shows that the fabric of complexity theory is tightly woven. The conjecture that #P is hard to approximate is closely tied to the ubiquitous demon of computer science — P =?NP.

Acknowledgement

We are indebted to Professor Juris Hartmanis for invaluable advice and encouragement.

References

[Ang80] Dana Angluin. On counting problems and the polynomial-time hierarchy. *Theoretical Computer Science*, 12:161–173, 1980.

[CH86] Jin-yi Cai and Lane Hemachandra. The Boolean hierarchy: hardware over NP. In *Structure in Complexity Theory*, pages 105–124, Springer-Verlag *Lecture Notes in Computer Science #223*, 1986.

[Coo71] S. A. Cook. The complexity of theorem-proving procedures. In *ACM Symposium on Theory of Computation*, pages 151–158, 1971.

[Gil77] John Gill. Computational complexity of probabilistic Turing machines. *SIAM Journal on Computing*, 6(4):675–695, December 1977.

[HH86a] Juris Hartmanis and Lane Hemachandra. Complexity classes without machines: on complete languages for UP. In *Automata, Languages, and Programming (ICALP 1986)*, Springer-Verlag *Lecture Notes in Computer Science*, 1986. To appear.

[HH86b] Juris Hartmanis and Lane Hemachandra. On sparse oracles separating feasible complexity classes. In *STACS 1986: 3rd Annual Symnposium on Theoretical Aspects of Computer Science*, Springer-Verlag *Lecture Notes in Computer Science #210*, 1986.

[HU79] John Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[Sim75] Janos Simon. *On Some Central Problems in Computational Complexity.* PhD thesis, Cornell University, Ithaca, N.Y., January 1975.

[Sim77] Janos Simon. On the difference between one and many. In *Automata, Languages, and Programming (ICALP 1977)*, pages 480–491, Springer-Verlag *Lecture Notes in Computer Science #52*, 1977.

[Sto85] Larry Stockmeyer. On approximation algorithms for #P. *SIAM Journal on Computing*, 14(4):849–861, November 1985.

[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. *SIAM Journal on Computing*, 8(3):410–421, 1979.

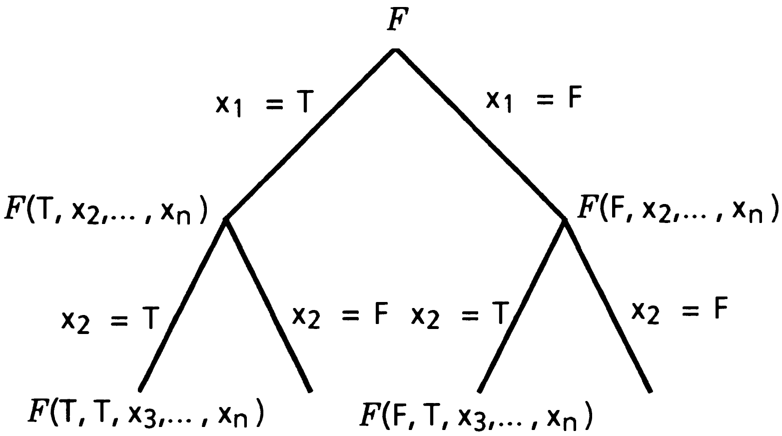

Figure 1A: The Tree

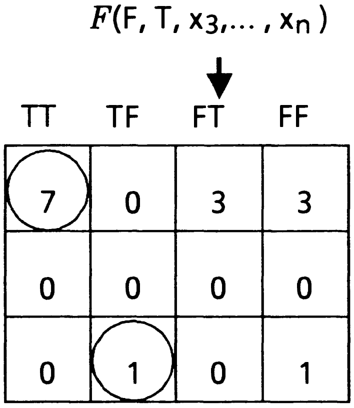

Figure 1B: The Guess Matrix

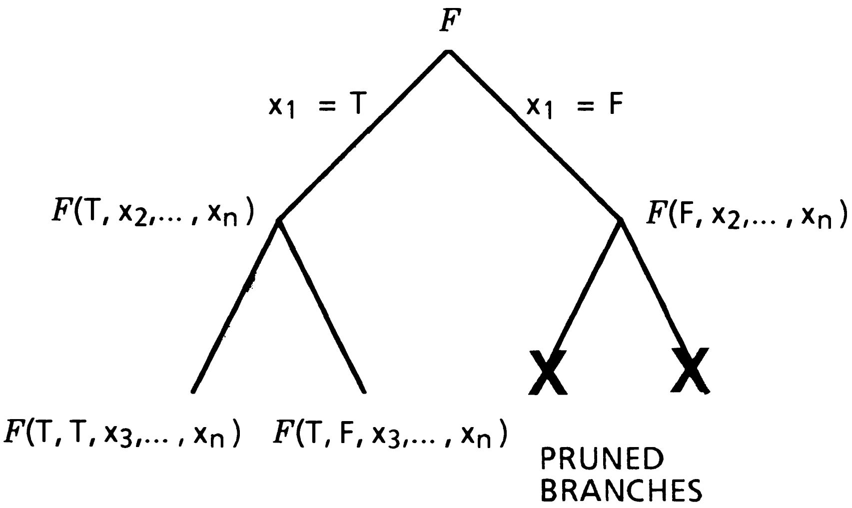

Figure 1C: The Pruned Tree