SIAM J. COMPUT.
Vol. 18, No. 6, pp. 1087-1097, December 1989

002

POLYNOMIAL SPACE COUNTING PROBLEMS*

RICHARD E. LADNER†

Abstract. The classes of functions $\sharp PSPACE$ and $\natural PSPACE$ that are analogous to the class $\sharp P$ are defined. Functions in $\sharp PSPACE$ count the number of accepting computations of a nondeterministic polynomial space bounded Turing machine, and functions in $\natural PSPACE$ count the number of accepting computations of nondeterministic polynomial space bounded Turing machines that on each computation path make at most a polynomial number of nondeterministic choices. In contrast to what is known about $\sharp P$, exact characterizations of both $\sharp PSPACE$ and $\natural PSPACE$ are found. In particular, $\sharp PSPACE = FPSPACE$ (the class of functions computable in polynomial space) and $\natural PSPACE = FPSPACE(poly)$ (the class of functions computable in polynomial space with output length bounded by a polynomial). Both $\sharp PSPACE$ and $\natural PSPACE$ can be characterized by counting problems related to alternating Turing machines. Both $\sharp PSPACE$ and $\natural PSPACE$ have natural complete functions. It is an easy observation that $FP \subseteq \sharp P \subseteq \natural PSPACE$, where FP is the class of functions computable in polynomial time. Relativization to oracles is considered, as are approximation techniques for obtaining a better understanding of whether either of the above inclusions is proper.

Key words. computational complexity, polynomial space, alternating polynomial time, counting problems

AMS(MOS) subject classification. 68

1. Introduction. In this paper we consider counting problems that arise in nondeterministic polynomial space bounded ($NPSPACE$) computations and in alternating polynomial time bounded ($APTIME$) computations. For example, the class of functions $\sharp PSPACE$[1] is defined by $f: \Sigma^* \to N \in \sharp PSPACE$ if and only if there is a nondeterministic Turing machine M that runs in polynomial space with the property that $f(x)$ equals the number of accepting computation paths of M on input x. (N is the set of natural numbers, $\{0, 1, 2, \cdots\}$.) We assume that our polynomial space bounded Turing machines, both deterministic and nondeterministic always halt, avoiding the possibility that $f(x)$ is infinite. Similarly, we define the class of functions $\sharp APTIME$ by $f \in \sharp APTIME$ if and only if there is an alternating Turing machine M that runs in polynomial time with the property that $f(x)$ equals the number of accepting computation trees of M on input x. Finally, define $FPSPACE$ to be the class of functions computable in polynomial space. It is worth noting that our $FPSPACE$ functions output only binary strings that represent members of N in a natural way. The order of output, high- or low-order bits first, does not matter because if $f \in FPSPACE$ then $g \in FPSPACE$, where $g(x) = f(x)^R$ ($g(x)$ is the reversal of $f(x)$) for all x. One main result of this paper is the following theorem.

THEOREM 1. $\sharp PSPACE = \sharp APTIME = FPSPACE$.

This exact characterization of counting problems in polynomial space contrasts with our current knowledge about $\sharp P$ defined by Valiant [10], [11]. The class $\sharp P$ is defined as the class of counting problems in NP, that is, $f \in \sharp P$ if and only if there is a nondeterministic Turing machine M that runs in polynomial time with the property that $f(x)$ equals the number of accepting computation paths of M on input x. Although $\sharp P$ is defined in a simple way in terms of $NTIME$ Turing machines it does not

* Received by the editors August 24, 1987; accepted for publication August 18, 1988. This work was supported by National Science Foundation grant DCR-8402565. Part of this research was done at the Mathematical Sciences Research Institute, Berkeley, California.

† Department of Computer Science, University of Washington, Seattle, Washington 98195.

[1] $\sharp PSPACE$ is pronounced "sharp p space."

seem that, in general, functions in $\#P$ are computable using an oracle in the polynomial time hierarchy. In fact, it is has been conjectured that complete functions in $\#P$ are not computable in polynomial time using an oracle in the polynomial time hierarchy. The best upper bound on $\#P$ is that it is contained in *FPSPACE*. However, it has been shown by Stockmeyer that every function in $\#P$ can be approximated by a function in $F\Delta_3^P$, the class of functions computable in polynomial time using an oracle from Σ_2^P [8]. Currently, we do not have any good characterization of $\#P$. One purpose of this paper is to try to gain more insight into $\#P$.

Because some functions in $\#PSPACE$ have exponential length and all functions in $\#P$ must have polynomial length, it is a simple observation that

$$\#P \neq \#PSPACE.$$

This inequality is a little artificial, leading us to consider restrictions of $\#PSPACE$ that might give us a more interesting comparison between $\#P$ and counting problems in polynomial space. Define the class of functions $\natural PSPACE$[2] by $f \in \natural PSPACE$ if and only if there is a nondeterministic Turing machine M that runs in polynomial space and that makes only a polynomial number of nondeterministic moves (while making potentially exponentially many moves) with the property that $f(x)$ equals the number of accepting computation paths of M on input x. With this restriction the length of $f(x)$ is bounded by a polynomial in $|x|$. Clearly, $\#P \subseteq \natural PSPACE$ and equality is not out of the question. However, $\natural PSPACE$ has some nice characterizations, which leads us to question the possibility that $\#P = \natural PSPACE$.

Let $\natural APTIME$ be the class of functions f such that there is an *APTIME* Turing machine M with the property that $f(x)$ equals the number of equivalence classes of accepting computation trees of M on input x, where two accepting computation trees are equivalent if the initial sequence of existential moves from the initial configuration to the first universal configuration is the same for both. At first glance, the class $\natural APTIME$ appears to be a bit bizarre. However, it turns out to be the natural counting class for alternating computations that corresponds to $\natural PSPACE$. Define $FPSPACE(poly)$ to be the class of functions that is computable in polynomial space and whose outputs are bounded in length by a polynomial. We can characterize $\natural PSPACE$ in Theorem 2.

THEOREM 2. $\natural PSPACE = \natural APTIME = FPSPACE(poly)$.

We define the notion of polynomial-time reducibility between counting problems that make it possible to compare one counting problem to another even though the counting problems may be functions that are exponential in length. We say that a function f is reducible to a function g in polynomial time ($f \leqq^P g$) if there is function h that is computable in polynomial time such that for all x, $f(x) = g(h(x))$. A function k is complete for a class of functions C if (i) $k \in C$ and (ii) for all $f \in C$, $f \leqq^P k$. There are natural counting problems that are complete for each of $\#PSPACE$ and $\natural PSPACE$. For example, let $\#NFA$ be the counting problem defined as follows: if M is a nondeterministic finite automaton, then $\#NFA(M)$ equals the number of input strings not accepted by M. In case the number of input strings not accepted by M is infinite, define $\#NFA(M)$ equal to $k = s^{2^q-1}+1$, where s is the number of input symbols of M and q is the number of states in M. The reason for this choice is that if the number of strings not accepted by M is finite, then the number of such nonaccepted strings must be $<k$. Then,

$$\#NFA \text{ is complete for } \#PSPACE.$$

Another example is $\natural QBF$, which is defined as follows. Let $A = \exists x_1 \forall x_2 \exists x_3 \cdots$

[2] $\natural PSPACE$ is pronounced "natural p space."

$Qx_mB(x_1, x_2, \cdots, x_m)$ be a quantified Boolean formula. Then $\natural QBF(A)$ equals the number of x_1's such that A is true. We have

$$\natural QBF \text{ is complete for } \natural PSPACE.$$

There are more natural complete problems that we describe later in the paper. Complete problems give us some information about the difficulty of $\sharp P$. For example, it can be shown that $\natural QBF \in \sharp P$ if and only if $\sharp P = \natural PSPACE$.

Let FP be the class of functions computable in polynomial time. The inclusions

$$FP \subseteq \sharp P \subseteq \natural PSPACE$$

are clear, and these inclusions relativize to an arbitrary oracle. It is an open question whether or not one or both of the inclusions are reversible. Using the techniques of Baker, Gill, and Solovay [1] we can construct computable oracles A, B, and C such that $FP^A = \natural PSPACE^A$, $FP^B \neq \sharp P^B$, and $\sharp P^C \neq \natural PSPACE^C$. These results indicate that more information about these inclusions may be very difficult to come by.

Stockmeyer has shown that functions in $\sharp P$ can be approximated by functions computable using an oracle in the polynomial hierarchy [8]. We give a generalization of this result to a class of counting problems defined by *APTIME* Turing machines with a bounded number of alternations. On the other hand, it is a fairly easy observation that if the functions in $\natural PSPACE$ can be approximated using an oracle in the polynomial hierarchy, then *PSPACE* is included in the polynomial hierarchy. This is some evidence that $\natural PSPACE$ is harder than $\sharp P$.

In § 2 we will clarify the definitions we have made so far and make new ones that will be used later in the paper. In § 3 we give proofs of Theorems 1 and 2. In § 4 we describe some complete problems and describe how they are useful. In § 5 we examine some relativized counting problems. Finally, in § 6 we discuss approximating counting problems.

2. Definitions. Our model of computation is the usual Turing machine. All our Turing machines will either be polynomial time or space bounded. As such, we can assume that our space bounded Turing machines always halt without looping. If the Turing machine has output, then the output is printed on a one-way, write-only output tape, and whatever space bound is put on the Turing machine does not apply to the output tape. Thus, if M is a *PSPACE* Turing machine that computes a function f, then $|f(x)|$ can be as long as $2^{|x|^k}$ for some k. For uniformity we assume that the output of a *PSPACE* Turing machine is a binary representation of a natural number ($N = \{0, 1, 2, \cdots\}$). Recall that *FPSPACE* is the class of functions computable in polynomial space and *FPSPACE*(*poly*) are those $f \in FPSPACE$ with the property that there is a constant k such that $|f(x)| \leqq |x|^k$ for all x. We assume that readers are familiar with the polynomial-time hierarchy, $\Sigma_1^P, \Sigma_2^P, \cdots$ [7]. We will also want to consider functions computable in polynomial time FP, and functions computable in polynomial time using an oracle in the polynomial-time hierarchy. Define $F\Delta_k^P$ to be the class of functions computable in polynomial time using an oracle in Σ_{k-1}^P. By definition $FP = F\Delta_1^P$.

If M is a *NPSPACE* Turing machine and if x is an input, then there may be many different computations of M that can lead to the acceptance of x. Define $\sharp(M, x)$ to be the number of distinct accepting computation paths of M on input x. Naturally, if M does not accept x, then $\sharp(M, x) = 0$ and *vice versa.* Define $\sharp PSPACE$ to be the class of functions f such that there is a *NPSPACE* Turing machine M where $f(x) = \sharp(M, x)$. Note that functions in $\sharp PSPACE$ can be exponential in length. A natural restriction that limits the length of function in $\sharp PSPACE$ is to allow only a polynomial number of nondeterministic moves on any accepting path. Hence, we have the following definition. $\natural PSPACE$ is the class of functions f such that there is a $NPSPACE$ Turing machine M that makes only a polynomial number of nondeterministic moves on any computation path and $f(x)=\sharp(M, x)$. The machine M may still make exponentially many moves on input x, but only a polynomial number of them are nondeterministic, when a choice can be made between alternatives.

We will assume that the reader is familiar with alternating Turing machines [4]. If M is an alternating Turing machine and x is an input, then an accepting computation tree is a tree labeled with configurations satisfying the following conditions: (i) the root is labeled with the initial configuration; (ii) each nonleaf labeled with a universal configuration has a child labeled for each successive configuration; (iii) each nonleaf labeled with an existential configuration has exactly one child, which is labeled with a successive configuration; and (iv) all the leaves are labeled with accepting configuration. Define $\sharp(M, x)$ to be the number of accepting computation trees of M on input x. Further define $\natural(M, x)$ to be the number of equivalence classes of accepting computation trees, where two accepting computation trees are equivalent if the paths from the initial configuration to the first universal configuration are the same in both. Note that $\sharp(M, x)$ can have length exponential in $|x|$ while $\natural(M, x)$ can only have length polynomial in $|x|$.

Define $\sharp APTIME$ to be the class of functions f such that there is an $APTIME$ Turing machine M such that $f(x)=\sharp(M, x)$. Further define $\natural APTIME$ to be the class of functions f such that there is an $APTIME$ Turing machine M with $f(x)=\natural(M, x)$. By limiting the number of alternations we can, in a natural way, also define $\sharp\Sigma_k^P$ and $\natural\Sigma_k^P$. Note that $\natural\Sigma_1^P$ is simply another way of expressing $\sharp P$. Note further that $\sharp\Sigma_1^P=\natural\Sigma_1^P$ and $\sharp\Sigma_2^P=\natural\Sigma_2^P$, but for all $k>2$, $\sharp\Sigma_k^P\neq\natural\Sigma_k^P$. This latter fact follows simply from that fact that some functions in $\sharp\Sigma_k^P$ are exponential in length for each $k>2$.

3. Equivalence of function classes. In this section we sketch the proof of Theorem 1:

$$\sharp PSPACE = \sharp APTIME = FPSPACE.$$

We begin by showing $\sharp PSPACE \subseteq \sharp APTIME$. A variation of the standard simulation of a nondeterministic Turing machine that runs in space $s(n)$ by an alternating Turing machine that runs in time $s^2(n)$ [4] has the property that the number of accepting computation paths of the nondeterministic machine equals the number of accepting computation trees of the alternating machine. Let M be a $NPSPACE$ Turing machine and let x be an input to M of length n. Without changing the number of accepting paths of M on x, we can pad all computations so they are all the same length $2^{p(n)}$ for some polynomial $p(n)$; we can also assume there is a unique accepting configuration. Consider the following alternating algorithm, $reach(C, D, k)$, which accepts if and only if configuration D is reachable from configuration C in *exactly* 2^k steps.

definition $reach(C, D, k)$:
begin
  if $k=0$ then if D is reachable from C in one step then *accept* else *reject* else
    $\bigvee E\,[reach(C, E, k-1) \wedge reach(E, D, k-1)]$
end.

The notation $\bigvee E$ means existentially choose a configuration E. The notation $\wedge$ is a binary operator meaning universally choose one of its operands. It can be shown by induction on k that the number of computation paths from C to D of length exactly 2^k equals the number of accepting computation trees of $reach(C, D, k)$. The alternating algorithm that simulates M on input x is simply the call *reach*(*init*, *acc*, $p(n)$), where *init* is the initial configuration of M on input x and *acc* is the unique accepting configuration. Hence, the number of accepting computations of M on input x equals the number of accepting computation trees of *reach*(*init*, *acc*, $p(n)$).

To show that $FPSPACE \subseteq \#PSPACE$, let $f \in FPSPACE$; let M be a $PSPACE$ Turing machine that computes f. We define a $NPSPACE$ machine M' with the property that $\#(M', x) = f(x)$ for all inputs x. Let x be fixed. We define a $PSPACE$ subroutine $bit(i)$ that returns the ith bit of $f(x)$, where $bit(0)$ is the lowest-order bit. We can assume without loss of generality that there are $m = 2^{|x|^k}$ bits in $f(x)$. If $f(x)$ is actually a small number, then there will be a large number of 0's as trailing bits. We define a nondeterministic recursive procedure, $check(i)$, which has the property that $\#check(i) = bit(i) + 2 \times \#check(i+1)$, where $\#check(i)$ refers to the number of accepting paths of $check(i)$. Hence, $\#check(0) = f(x)$. The machine M' simply runs $check(0)$.

```
definition check(i):
begin
   if i > m then reject else
      if bit(i) = 0 then check(i+1) ∨ check(i+1)
      else (bit(i) = 1) accept ∨ check(i+1) ∨ check(i+1)
end.
```

To complete the proof, we show $\#APTIME \subseteq FPSPACE$. Let M be an alternating Turing machine that runs in polynomial time and let x be an input. Consider the full computation of tree T of M on input x. Unlike an accepting computation tree, each existential node in T may have more than one child, one child for each choice that can be made. $\#(M, x)$ can be computed, not in polynomial space, in the following way. Construct T and label each node in T with a number starting with the leaves of T. An accepting leaf is labeled with a one and a rejecting leaf is labeled with a zero. If a universal node has k children labeled $a_1, \cdots a_k$, respectively, then label the universal node with $\prod_{i=1}^{k} a_i$. If an existential node has k children labeled $a_1, \cdots, a_k$, respectively, then label the existential node with $\sum_{i=1}^{k} a_i$. The number labeled at the root is $\#(M, x)$.

There are two difficulties in trying to do this calculation of $\#(M, x)$ within polynomial space. The first is that the tree T requires exponential space to store it all, and the second is that the numbers labeled on the nodes of the tree can be exponentially long. Both these difficulties can be overcome by realizing that the whole tree and the number labels do not all have to be written down at once and that arithmetic on exponentially long numbers can be done in polynomial space. To be more specific, if C is a configuration, define $number(C)$ to be the number of accepting subtrees of M on input x when M is started in configuration C. We can define a recursive procedure $bit(i, C)$ that is the value of the ith bit of $number(C)$. Roughly, *bit* is defined recursively by:

```
definition bit(i, C):
begin
   if C is accepting then if i = 0 then return 1 else return 0 else
   if C is rejecting then return 0 else
   if C is universal with children D_1, ···, D_k then
      return the ith bit of product(number(D_1), ···, number(D_k)) else
   if C is existential with children D_1, ···, D_k then
      return the ith bit of sum(number(D_1), ···, number(D_k))
end.
```

It is well known that there are procedures for *product* and *sum* that on inputs of length m run in $\log m$ space. This can be seen by taking standard logarithmic depth circuits for *product* and *sum* (see [6]) and converting them to logarithmic space Turing machines using a result of Borodin [3]. To compute the ith bit of $product(number(D_1), \cdots, number(D_k))$, run the logarithmic space Turing machine for *product* without actually writing any of the output. Keep a count of the number of output symbols that have been produced so far. When the count reaches i, the ith bit is produced. While *product* is running it may request a bit from its input. At that point a recursive call to *bit* is made. The full arguments of *product* are never written down all at once. A similar computation for *sum* can be made. The depth of recursion is bounded by the height of the computation tree, which is polynomial in $|x|$. The amount of space needed at each level of recursion is logarithmic in the length of $number(C)$, which again is polynomial in $|x|$. Hence, the computation of $bit(i, C)$ can be done in polynomial space. Clearly, the number of accepting subtrees of M on input x can be output by successive calls to $bit(i, init)$, where *init* is the initial configuration.

The proof of Theorem 2,

$$\natural PSPACE = \natural APTIME = FPSPACE(poly),$$

is a bit simpler than the result we have just proven.

We begin by showing that $\natural PSPACE \subseteq \natural APTIME$. Let M be a *PSPACE* Turing machine that makes at most a polynomial number of nondeterministic moves in a computation. There is a *PSPACE* Turing machine M' that takes two inputs x and y with the property that the number of accepting computations of M on input y is exactly the number of distinct x's such that M' accepts the pair (x, y). Furthermore, the length of x is bounded by a polynomial. Essentially, M' on input (x, y) simulates M on input y except when M reaches a nondeterministic move. The ith nondeterministic move is then determined by the ith character of x. By the standard alternating time theorem [4] M' can be simulated by an *APTIME* Turing machine M'' that begins in a universal configuration. Now, consider the alternating machine M''' that on input y first existentially guesses an x, then runs M'' on the input (x, y). We have $\#(M, y) = \natural(M''', y)$.

To show $FPSPACE(poly) \subseteq \natural PSPACE$ we can use the same nondeterministic procedure *check* used to prove $FPSPACE \subseteq \# PSPACE$. In the case that the function is polynomial length bounded, the nondeterministic procedure *check* makes only a polynomial number of nondeterministic moves on any computation path.

Finally, to show that $\natural APTIME \subseteq FPSPACE(poly)$ let M be an *APTIME* Turing machine. We define a new *PSPACE* machine M' that takes pairs (x, y) as input. On input (x, y), M' simulates the initial existential moves of M on input y with these moves determined by the characters of x, reaching a configuration C of M. That is, if M can make at most d moves from an existential configuration, then $x \in \{1, 2, \cdots, d\}^*$. Initially, M' sets a pointer to the first character of x. For each initial existential move of M, M' makes the move determined by the current character of x pointed to and moves the pointer one character to the right. If C is existential and x is exhausted, or C is universal and x is not yet exhausted, then M' rejects. Otherwise, a deterministic polynomial space simulation of the alternating machine M on input y starting in configuration C is used to discover if C is the root of an accepting computation tree of M on input y. If so, M' accepts (x, y). From M' we construct a Turing machine M'' that computes $\natural(M, y)$. The Turing machine M'' simply counts the number of x's such that M' accepts (x, y). The number of such x's is bounded in length by a polynomial in $|y|$.

4. Complete functions. We would like to be able to classify the hardest problems in $\#PSPACE$ and $\natural PSPACE$ in a way similar to the way that the NP-complete problems are classified as the hardest problems in NP. To this end we define a generalization of the Karp reducibility [5] that extends it to a relation between functions. We say that f is reducible to g in polynomial time ($f \leqq^P g$) if there is function h that is computable in polynomial time such that for all $x, f(x) = g(h(x))$. It is easy to see that $\leqq^P$ is a reflexive and transitive relation between functions. If C is any of the classes of functions we have talked about so far, $FP, \#P, \#\Sigma_k^P, \natural\Sigma_k^P, \#PSPACE$, or $\natural PSPACE$, $g \in C$ and $f \leqq^P g$ then $f \in C$ also. Define a function k to be *complete* for a class of functions C if (i) $k \in C$ and (ii) for all $f \in C, f \leqq^P k$.

Since some functions in $\#PSPACE$ are exponential in length, it is a trivial observation that $\#PSPACE$ cannot equal any of $FP, \#P, \#\Sigma_2^P, \natural\Sigma_k^P$, or $\natural PSPACE$, each of which contains only polynomial length bounded functions. But it is not inconceivable that $\#PSPACE = \#\Sigma_k^P$ for some $k > 2$. The following proposition relates the complexity of complete functions in $\#PSPACE$ to whether $\#\Sigma_k^P = \#PSPACE$ for some k.

PROPOSITION 3. *Let f be complete for $\#PSPACE$. Then, $f \in \#\Sigma_k^P$ if and only if $\#\Sigma_k^P = \#PSPACE$.*

Functions in $\natural PSPACE$ are bounded in length by a polynomial so there is some chance that $\natural PSPACE$ could equal one of $FP, \#P$, or $\natural\Sigma_k^P$ for some k. Knowing the complexity of complete functions in $\natural PSPACE$ enables us to better characterize the class $\natural PSPACE$, and perhaps even $\#P$.

PROPOSITION 4. *Let f be complete for $\natural PSPACE$.*

(1) *$f \in FP$ if and only if $FP = \natural PSPACE$,*

(2) *$f \in \#P$ if and only if $\#P = \natural PSPACE$,*

(3) *$f \in \natural\Sigma_k^P$ if and only if $\natural\Sigma_k^P = \natural PSPACE$, for $k \geqq 2$.*

It is worth noting that if f is complete for $\natural PSPACE$ and $f \in FP$, then $P = PSPACE$.

The first complete counting functions we consider are derived from the QBF (quantified Boolean formulas) problem first considered by Stockmeyer and Meyer [9]. A quantified Boolean formula is one of the form

$$A = \exists x_1 \forall x_2 \exists x_3 \cdots Qx_m B(x_1, x_2, \cdots, x_m),$$

where x_i is a vector of Boolean variables for $1 \leqq i \leqq m$, B is a Boolean formula, and $Q = \forall$ if m is even and $Q = \exists$ if m is odd. If A is true, then its truth can be verified by constructing a *verifying tree*, the root of which is labeled with a truth assignment for x_1. The root has $2^{|x_2|}$ children, one for each possible truth assignment of x_2. Each child of the root has exactly one child labeled with a truth assignment for x_3. This process is carried on until all the variables are assigned. Thus, each leaf corresponds to an assignment of all the variables. For each leaf in the verifying tree the formula B must be true for the assignment associated with the leaf. Define $\#QBF$ to be the function that is defined by $\#QBF(A) =$ the number of verifying trees for A. We can also define $\natural QBF(A) =$ the number of x_1's such that there is a verifying tree for A with root labeled x_1.

THEOREM 5. (1) *$\#QBF$ is complete for $\#PSPACE$,*

(2) *$\natural QBF$ is complete for $\natural PSPACE$.*

The key idea in the proof of part (1) of Theorem 5 is that given a $NPSPACE$ Turing machine M and an input x a quantified Boolean formula A can be constructed in polynomial time with the property that $\#(M, x) = \#QBF(A)$. This will demonstrate that the function f is defined by $f(x) = \#(M, x)$ satisfies $f \leqq^P \#QBF$. The construction is a modification of the construction that shows *QBF* is complete for *PSPACE* [9]. To begin with, let us describe the more standard construction and explain why it does not work to get our result. We then modify the construction to get it to work.

Let M be a *NPSPACE* Turing machine and let x be an input of length n. We can assume that on input x, M makes exactly $2^{p(n)}$ moves on every computation path before halting, where $p(n)$ is a polynomial. A configuration of M on input x can be represented by a bit vector u of length $q(n)$ for some polynomial $q(n)$. We define a quantified Boolean formula $reach_k(u_k, v_k)$ that has $2q(n)$ free variables u_k, v_k and has the meaning "the configuration represented by v_k is reachable from the configuration represented by u_k in *exactly* 2^k moves of M." For $k>0$, define

$$reach_k(u_k, v_k) = \exists z_k \, \forall u_{k-1}, v_{k-1}$$
$$[(u_{k-1} \equiv u_k \wedge v_{k-1} \equiv z_k) \vee (u_{k-1} \equiv z_k \wedge v_{k-1} \equiv v_k) \Rightarrow reach_{k-1}(u_{k-1}, v_{k-1})].$$

In this context $u \equiv v$ means that u and v are pointwise equivalent. For $k = 0$, $reach_0(u_0, v_0)$ is a quantifier-free formula expressing that v_0 follows from u_0 in exactly one move of M. The formula for $reach_k(u_k, v_k)$ can be easily rewritten so that all the quantifiers are leading quantifiers instead of embedded in the formula. Let c and d be specific bit vectors representing configurations of M. Unfortunately, the number of verifying trees of $reach_k(c, d)$ is in general larger than the number of computation paths of length 2^k from the configuration represented by c to the one represented by d. This is because in the definition of $reach_k(u_k, v_k)$ if $\neg[(u_{k-1} \equiv u_k \wedge v_{k-1} \equiv z_{k-1}) \vee (u_{k-1} \equiv z_k \wedge v_{k-1} \equiv v_k)]$, then the body of $reach_k(u_k, v_k)$ is true no matter what the values are of the rest of the variables. To overcome this difficulty we modify the formula to completely determine the values of the existentially quantified variables in all cases. To do this we define a formula $reach1_k$ that has free variables $u_0, \cdots, u_k, v_0, \cdots, v_k, z_1, \cdots, z_k$. If $k=0$, then $reach1_k = reach_k$. If $k>0$, then

$$reach1_k = [(u_{k-1} \equiv u_k \wedge v_{k-1} \equiv z_k) \vee (u_{k-1} \equiv z_k \wedge v_{k-1} \equiv v_k) \Rightarrow reach1_{k-1}] \wedge$$
$$[\neg[(u_{k-1} \equiv u_k \wedge v_{k-1} \equiv z_k) \vee (u_{k-1} \equiv z_k \wedge v_{k-1} \equiv v_k)] \Rightarrow \textstyle\bigwedge_{i=1}^{k-1} z_i \equiv 0].$$

In this context 0 represents the constant zero vector of length $q(n)$. Now $reach_k$ with free variables u_k, v_k is defined by

$$reach_k(u_k, v_k) = \exists z_k \, \forall u_{k-1}, v_{k-1} \, \exists z_{k-1} \, \forall u_{k-2}, v_{k-2} \cdots \exists z_1 \, \forall u_0, v_0 \; reach1_k.$$

With this definition of $reach_k$ it can be shown that if c and d are bit vectors representing configurations, then the number of verifying trees of $reach_k(c, d)$ equals the number of computation paths of length 2^k from the configuration represented by c to the one represented by d. Hence if *init* is the initial configuration and *acc* is the accepting configuration, then the formula $reach_{p(n)}(init, acc)$ has the number of verifying trees equal to the number of accepting computations of M on input x. Furthermore, the formula $reach_{p(n)}(init, acc)$ can be produced in time polynomial in n.

Part (2) of Theorem 5 can be shown using some of the ideas in the proof of $\natural PSPACE \subseteq \natural APTIME$ combined with the construction just given. Let M be a *PSPACE* Turing machine that makes at most a polynomial number of nondeterministic moves in a computation. There is a *PSPACE* Turing machine M' that takes two inputs x and y with the property that the number of accepting computations of M on input y is exactly the number x's such that M' accepts the pair (x, y). Furthermore, we can assume that for any y, if (x_1, y) and (x_2, y) are accepted by M', then $|x_1| = |x_2|$, $|x_1|$ is bounded by a polynomial, and x_1 is a binary string. From the machine M' we can construct $reach1_k$ just as before. In this case, for a particular input y to M there are many inputs (x, y) to M'. Let $init(x)$ be the representation of the initial configuration corresponding to (x, y). Consider the following formula:

$$\exists x\, \forall w\, reach_{p(n)}(init(x), acc), \tag{1}$$

where M' runs in time $2^{p(n)}$ and w is a dummy Boolean variable. The number of x's such that there is a verifying tree for (1) is exactly the number of accepting computation paths of M on input y.

It is worth mentioning a couple of other functions that are complete for $\sharp PSPACE$. If M is a nondeterministic finite automaton, then $\sharp NFA(M)$ is the number of strings not accepted by M if the number of strings not accepted by M is finite and $\sharp PSPACE = s^{2^q-1}+1$ (where s is the number of symbols in the input alphabet and q is the number of states of M) otherwise. The number $s^{2^q-1}+1$ is chosen to be just larger than an upper bound on the number of strings which could be in the finite complement of a set accepted by a nondeterministic finite automaton with s input symbols and q states. If E is a regular expression, then $\sharp RE(E)$ is the number of strings not in the language defined by E if the number of string not in the language defined by E is finite and $\sharp RE(E) = s^{2^l-1}+1$ (where s is the number of symbols in the alphabet of E and l is the length of E) otherwise. The number $s^{2^l-1}+1$ is chosen to be just larger than an upper bound on the number of strings which could be in the finite complement of a language defined by a regular expression with s alphabet symbols and length l. Both $\sharp NFA$ and $\sharp RE$ are complete for $\sharp PSPACE$. It would be interesting to find more natural functions that are complete for either $\sharp PSPACE$ or $\natural PSPACE$.

5. Relativization. A good characterization of the complexity of $\sharp P$ is not known. We do have some interesting characterizations of $\natural PSPACE$ but for all we know $\natural PSPACE = FP$. The following inclusions are all we know for sure:

$$FP \subseteq \sharp P = \natural\Sigma_1^P \subseteq \natural\Sigma_2^P \subseteq \cdots \subseteq \natural PSPACE.$$

This sequence of inclusions also relativizes to any oracle so we have

$$FP^A \subseteq \sharp P^A = \natural\Sigma_1^{P,A} \subseteq \natural\Sigma_2^{P,A} \subseteq \cdots \subseteq \natural PSPACE^A.$$

In the definition of $\natural PSPACE^A$ we only allow strings of polynomial length to be written on the oracle tape. This makes the comparisons between the different classes of functions more fair. The following theorem gives the possible relationships between FP, $\sharp P$, and $\natural PSPACE$.

THEOREM 6. *There are computable oracles A, B, and C such that*:

(1) $FPA^A = \natural PSPACE^A$,

(2) $FP^B \neq \sharp P^B$,

(3) $\sharp P^C \neq \natural PSPACE^C$.

This result indicates it may be very difficult to separate FP from $\sharp P$ and $\sharp P$ from $\natural PSPACE$.

To show part (1) of Theorem 6, let A be any $PSPACE$-complete set. Suppose $f \in \natural PSPACE^A$. Since A is a member of $PSPACE$, it can be easily seen that $f \in FPSPACE(poly)$. The language $L = \{\langle x, i, b\rangle \mid i\text{th bit of } f(x) \text{ is } b\}$ is clearly a member of $PSPACE$. Since A is complete for $PSPACE$, L is polynomial time reducible to A. Hence, $f \in FP^A$ because, given x, the bits of $f(x)$ are encoded in the set L which can be computed in polynomial time using the set A as an oracle.

Part (2) of Theorem 6 holds for any B such that $P^B \neq NP^B$ [1]. Suppose $FP^B = \sharp P^B$. Let M be a $NPTIME$ oracle Turing machine that accepts L using oracle B. Let f be the function in $\sharp P^B$ defined by M. Thus, f is also a member of FP^B by assumption.

To check if $x \in L$ in polynomial time, compute $f(x)$ in polynomial time using the oracle B. We have $x \in L$ if and only if $f(x) > 0$. Hence, $P^B = NP^B$.

Part (3) of Theorem 6 holds for any C such that $NP^C \neq PSPACE^C$. Such sets C exist from the results of Baker and Selman [2] and Yao [12]. Suppose $\sharp P^C = \natural PSPACE^C$. Let M be a $PSPACE$ oracle Turing machine which accepts L using oracle C. Let f be the function in $\natural PSPACE^C$ defined by M thinking of M as a nondeterministic machine. By assumption, $f \in \sharp P^C$. Let M' be the $NPTIME$ oracle Turing machine which on oracle C defines f. The machine M' also accepts the set L. Hence, $NP^C = PSPACE^C$.

The proof technique of parts (2) and (3) of Theorem 6 combined with the result of Yao that the polynomial hierarchy can be separated using an oracle [12] can be used to show Theorem 7.

THEOREM 7. *There is an oracle A such that*

$$FP^A \subset \natural\Sigma_1^{P,A} \subset \natural\Sigma_2^{P,A} \subset \cdots \subset \natural PSPACE^A.$$

The notation $X \subset Y$ means that X is a proper subset of Y.

6. Approximation. Stockmeyer has shown that functions in $\sharp P$ can be approximated by functions in the polynomial hierarchy, namely by functions by $F\Delta_3^P$ [8]. This result can be generalized to functions in $\natural\Sigma_k^P$. We say that f is approximated by g if there is a constant $c \geqq 1$ such that for all $x, f(x)/c \leqq g(x) \leqq cf(x)$.

THEOREM 8. *For $k \geqq 2$, every function in $\natural\Sigma_k^P$ can be approximated by a function in $F\Delta_{k+1}^P$.*

A complete proof of Theorem 8 will not be given here since it follows immediately from an observation about a proof of Stockmeyer [8, Thm. 3.1, p. 858]. In Stockmeyer's proof a predicate of the form $z \in Acc_M(x)$, meaning "z is an accepting computation of M on input x," must be evaluated. In this context M is a $NPTIME$ Turing machine so that this predicate can be checked in polynomial time. In our context this predicate would mean that "z is an initial sequence of configurations, the last of which is a universal configuration and all but the last of which are existential configurations, in an accepting computation tree of M on input x." In our context M is an $APTIME$ Turing machine that starts in an existential configuration and makes at most $k-1$ alternations before halting. That is, M is a machine that defines a member of the class $\natural\Sigma_k^P$. Hence, in our context the predicate $z \in Acc_M(x)$ is computable in Π_{k-1}^P. Examining Stockmeyer's proof, we find this observation guarantees that a function in $F\Delta_{k+1}^P$ can approximate a function in $\natural\Sigma_k^P$.

Finally it is easy to show that functions in $\natural PSPACE$ are approximated by functions in the polynomial hierarchy if and only if $PSPACE$ is contained in the polynomial hierarchy. To see this note that if $\{0, 1\}$-valued function f is approximated by g, then g essentially exactly computes f. This is evidence that the functions in $\natural PSPACE$ are harder to compute than those in $\sharp P$ or in $\natural\Sigma_k^P$ for any k.

Acknowledgments. The author thanks Sam Buss for the stimulating conversations that inspired some of these results. Thanks also go to Larry Stockmeyer for pointing out an easy way of finding oracles satisfying parts (2) and (3) of Theorem 6 and to an anonymous referee for pointing out an easy way to satisfy part (1) of Theorem 6. Also, the author thanks both referees for many useful suggestions.

REFERENCES

[1] T. BAKER, J. GILL, AND R. SOLOVAY, *Relativizations of the* P = ?NP *question*, SIAM J. Comput., 4 (1975), pp. 431–442.

[2] T. BAKER AND A. SELMAN, *A second step toward the polynomial hierarchy*, Theoretical Computer Science, 8 (1979), pp. 177-187.

[3] A. BORODIN, *On relating time and space to size and depth*, SIAM J. Comput., 6 (1977), pp. 733-744.

[4] A. K. CHANDRA, D. C. KOZEN, AND L. J. STOCKMEYER, *Alternation*, J. Assoc. Comput. Mach., 28 (1981), pp. 114-133.

[5] R. M. KARP, *Reducibility among combinatorial problems*, in Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85-104.

[6] J. SAVAGE, *The Complexity of Computing*, John Wiley, New York, 1976.

[7] L. J. STOCKMEYER, *The polynomial-time hierarchy*, Theoret. Comput. Sci., 3 (1977), pp. 1-22.

[8] ———, *On approximation algorithms for #P*, SIAM J. Comput., 14 (1985), pp. 849-861.

[9] L. J. STOCKMEYER AND A. R. MEYER, *Word problems requiring exponential time: Preliminary report*, in Proc. 5th Annual ACM Symposium on Theory of Computing, 1973, pp. 1-9.

[10] L. G. VALIANT, *The complexity of computing the permanent*, Theoret. Comput. Sci., 8 (1979), pp. 189-201.

[11] ———, *The complexity of enumeration and reliability problems*, SIAM J. Comput., 8 (1979), pp. 410-421.

[12] A. C. YAO, *Separating the polynomial-time hierarchy by oracles*, in 26th Annual IEEE Symposium on Foundations of Computer Science, 1985, pp. 1-5.