Deterministic Polynomial Time with O(log n) Queries

Jim Kadin

TR 86-771
August 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853

Deterministic Polynomial Time with O($\log n$) Queries

Jim Kadin
Cornell University

August 21, 1986

Abstract

$P^{NP[\log n]}$ is the class of languages recognizable by deterministic polynomial time machines that make $O(\log n)$ queries to an oracle for NP. Many of the languages related to optimal solution sizes of NP optimization problems are members of this class. We relate $P^{NP[\log n]}$ to the study of sparse oracles for NP by showing that if NP has a sparse $\leq_T^P$-complete set, then the polynomial time hierarchy collapses to $P^{NP[\log n]}$. We also discuss complete problems and show that UOCSAT, the set of CNF formulas with the property that every assignment that satisfies the maximum number of clauses satisfies the same set of clauses, is $\leq_m^P$-complete for $P^{NP[\log n]}$.

1 Introduction

Binary search is an important tool in computer science. The exponential speedup gained by using binary search over linear search makes many algorithms tractable. It is not suprising that this exponential speedup finds its way into complexity theory and the study of polynomial time computations and nondeterminism. For instance, the polynomial self-reducibility of satisfiable Boolean formulas depends upon a binary search based algorithm for finding a satisfying assignment from the exponentially many possible assignments. Hence we know that given an oracle for satisfiable Boolean formulas (SAT), there is a polynomial time function for finding satisfying assignments of formulas.

Similarly, given an oracle for SAT, there are binary search algorithms for finding the optimal solution sizes to NP-complete optimization problems. For example, the cost of the optimal TSP (traveling salesperson problem) tour of a graph with weighted edges can be computed with binary search (relative to SAT). Again since the cost of the TSP tours can be exponential in the size of the representation of the weighted graph, the search for the optimal cost can take polynomially many queries.

For many other NP-complete optimization problems such as CLIQUE and K-SAT (a pair $\langle F, k \rangle$ is a member of K-SAT if F is a conjunctive normal form (CNF) formula and there is an assignment to the variables of F that satisfies k or more of the clauses of F) the size of whatever is being optimized is bounded by a polynomial in the size of the problem representation. The maximal clique size of a graph is bounded by the number of nodes of the graph, and the maximal number of simultaneously satisfiable clauses of a formula is bounded by the number of clauses. Thus for these problems, there are deterministic polynomial time functions for computing optimal solution sizes that require only $O(\log n)$ queries to an oracle for NP (SAT). Intuitively, algorithms that require only $O(\log n)$ queries do not seem to use the full power of polynomial time oracle machines.

If we look at language recognition problems that are related to optimal solution sizes, we find the same type of split: languages that can be recognized with $O(\log n)$ queries and languages that seem to require polynomially many queries. Recognizing UOTSP, the set of weighted graphs that have a unique optimal TSP tour, seems to require polynomially many queries – a binary search to compute the optimal tour cost and one more query to determine uniqueness.

UOCLIQUE, on the other hand, can be recognized with $O(\log n)$ queries. Similarly the "unique optimal" versions of many other NP-complete optimization problems such as K-SAT, vertex cover, graph colorability, etc. can also be recognized with $O(\log n)$ queries. Other languages such as the set of graphs whose maximal clique size is even also have this property. These languages are members of a class that is probably distinct from P^{SAT}.

Definition 1 $\mathrm{P}^{\mathrm{NP}[\log n]}$ *is the class of all languages accepted by polynomial time oracle machines that make* $\mathrm{O}(\log n)$ *queries to an* NP *oracle. That is,* $L \in \mathrm{P}^{\mathrm{NP}[\log n]}$ *if and only if there exists a deterministic, polynomial time oracle machine* M*, a constant* k_M*, and an* NP*-complete set* C *such that* $L = \mathrm{L}(M^C)$ *and for all inputs* x*,* M *makes at most* $k_M(\log |x|) + k_M$ *queries.*

Similarly, $\mathrm{P}^{\mathrm{SAT}[\log n]}$ *is the class of all languages accepted by polynomial time oracle machines that make* $\mathrm{O}(\log n)$ *queries to* SAT *(where* SAT *is the set of CNF Boolean formulas that have satisfying assignments).*

Since SAT is $\leq_{\mathrm{m}}^{\mathrm{P}}$-complete for NP, any machine that asks queries to an NP-complete oracle set C can easily be transformed into a machine that asks queries of SAT. The transformed machine will ask the same number of queries and will take only polynomially more steps than the original machine. Hence $\mathrm{P}^{\mathrm{NP}[\log n]} = \mathrm{P}^{\mathrm{SAT}[\log n]}$. We will use these terms as synonyms. Similarly P^{NP} and $\mathrm{P}^{\mathrm{SAT}}$ will be used synonymously.

This paper presents results concerning $\mathrm{P}^{\mathrm{NP}[\log n]}$. As many of the results illustrate, the distinction between $\mathrm{P}^{\mathrm{NP}[\log n]}$ and P^{NP} is often seen as the distinction between binary search over polynomially many elements and binary search over exponentially many elements.

It is important to keep in mind that $\mathrm{P}^{\mathrm{NP}[\log n]}$ and P^{NP} are classes of languages and not classes of functions. Krentel has shown a pretty result for the corresponding function classes $\mathrm{FP}^{\mathrm{NP}[\log]}$ and $\mathrm{FP}^{\mathrm{NP}}$. He has proved that if $\mathrm{P} \neq \mathrm{NP}$, then there are polynomial time functions that make polynomially many queries to a SAT oracle that cannot be computed with $\mathrm{O}(\log n)$ queries [Kr]. Thus, for functions, he has indeed proved that making $\mathrm{O}(\log n)$ queries does not use the full power of polynomial time oracle machines (under the assumption that $\mathrm{P} \neq \mathrm{NP}$).

As we will see in section 4, Krentel's result does not hold for relativized versions of $\mathrm{P}^{\mathrm{NP}[\log n]}$ and P^{NP}. Thus these language classes and function classes are not necessarily as closely related as one might think.

Our discussion of $\mathrm{P}^{\mathrm{NP}[\log n]}$ begins with section 2 where we introduce some preliminary notions and define some of the languages in $\mathrm{P}^{\mathrm{NP}[\log n]}$.

Section 3 shows that in addition to containing many natural languages, $\mathrm{P}^{\mathrm{NP}[\log n]}$ has natural complete languages. We show that UOCSAT, a version of "unique optimal" K-SAT, and a variant of UOCLIQUE are $\leq_{\mathrm{m}}^{\mathrm{P}}$-complete.

These results are reminiscent of a result by Papadimitriou. He has shown that UOTSP is $\leq^{\mathrm{P}}_{\mathrm{m}}$-complete for P^{NP} [Pa]. This is strong evidence that recognizing UOTSP really does require polynomially many queries and thus is probably not in $\mathrm{P}^{\mathrm{NP}[\log n]}$.

In section 4 we prove a result that ties $\mathrm{P}^{\mathrm{SAT}[\log n]}$ to the study of sparse oracles for NP.

Recall that Karp and Lipton showed that if there exists a sparse oracle for NP, then the polynomial time hierarchy (PH) collapses down to Σ^{P}_2 ($\mathrm{NP}^{\mathrm{SAT}}$) [KL].

Mahaney took things a step further by proving that if the sparse oracle is itself a set in NP, then the PH collapses down to Δ^{P}_2 ($\mathrm{P}^{\mathrm{SAT}}$). His proof involves showing that a $\mathrm{P}^{\mathrm{SAT}}$ machine can actually enumerate (write down) the strings in the oracle set [Ma,Ma2].

Long generalized Mahaney's result by showing that if the oracle set is anywhere in Δ^{P}_2, then the collapse occurs down to Δ^{P}_2. Long's theorem relies on the fact that if there exists a sparse oracle in Δ^{P}_2, then all the relevant strings in the oracle up to a certain length can be enumerated by a $\mathrm{P}^{\mathrm{SAT}}$ machine [Lo].

Thus both Long's and Mahaney's results depend upon enumerating the strings of the oracle set with algorthims that make polynomially many queries.

We show that if the oracle set is in NP, then there is no need to enumerate the oracle in order to prove that the PH collapses. Instead, the census function of the sparse oracle is all that is needed, and that can be computed – by binary search – with $\mathrm{O}(\log n)$ queries. Therefore we know that if there is a sparse oracle for NP that is in NP, then the PH collapses to $\mathrm{P}^{\mathrm{SAT}[\log n]}$. This sharpens Mahaney's result and reasserts the distinction between sparse oracles inside NP and oracles outside NP that was blurred by Long's result.

2 Preliminaries

The reader is assumed to be familiar with the concepts of Turing machines, oracle machines, P (deterministic polynomial time), NP (nondeterministic polynomial time), and PH (the polynomial time hierarchy).

Consider the following sets encoded in some reasonable syntax:

UOCSAT (Unique Optimal Clause Satisfiability) - the set of CNF formulas with the property that all the assignments that satisfy the maximal number of clauses happen to satisfy the same set of clauses.

UOASAT (Unique Optimal Assignment Satisfiability) - the set of CNF formulas that have exactly one assignment that satisfies the maximal number of clauses, and all other assignments satisfy fewer clauses.

UOBTSP(k) (Unique Optimal Bounded Traveling Salesperson Problem) - the set of undirected graphs $G = (V, E)$ with edge costs no more than $|V|^k + k$ such that there is a unique optimal Hamiltonian circuit through the graph.

UOCLIQUE (Unique Optimal Clique) - the set of undirected graphs that have one clique containing more vertices than each of the other cliques of the graph.

UOCOLORING (Unique Optimal Graph Coloring).

UOVCOVER (Unique Optimal Vertex Cover).

SAT-MOD-k - the set of pairs $\langle F, \#^m \rangle$ such that F is a CNF formula, and the maximal number of simultaneously satisfiable clauses of F is equal to 0 mod m.

CLIQUE-MOD-k - the set of pairs $\langle G, \#^m \rangle$ such that G is an undirected graph whose maximal clique size is equal to 0 mod m.

All these languages and many other problems similarly defined are easily seen to be in $\mathrm{P}^{\mathrm{SAT}[\log n]}$. The optimal solution size can be determined by a binary search that uses $O(\log n)$ queries. The uniqueness of the optimal solution can be determined with one more query; equivalence mod m can be determined without further access to the oracle.

While $\mathrm{P}^{\mathrm{SAT}[\log n]}$ is the natural class that captures this binary search technique, it is also represented by a very "clean" machine model. If M is a $\mathrm{P}^{\mathrm{SAT}[\log n]}$ machine, then M is a nice, deterministic, polynomial time bounded machine that makes only $O(\log n)$ queries. Given an input x, we

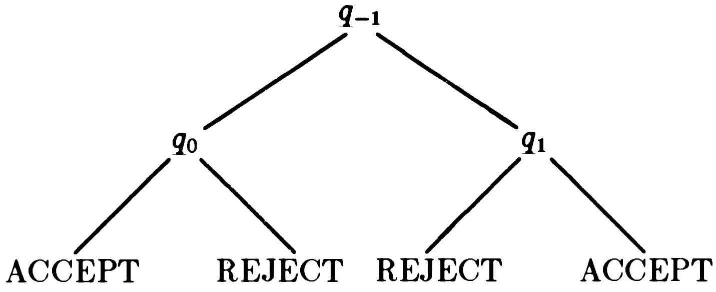

Figure 1: Query tree of height 2.

can map out M's query behavior on x in polynomial time – without access to the oracle. We can simulate M on x trying both possible answers to each query and see whether M accepts or rejects with each sequence of query answers.

The query behavior of M on x can be represented by a *query tree.* A query tree is a binary tree with query strings for internal nodes and leaves that are labeled either ACCEPT or REJECT. The left branch from a node, q, leads to a node that describes M's behavior if the oracle answer to q is "no". Thus if the next query M would ask were q', the left child of q would be labeled q'. If an answer of "no" to q would make M reject or accept without any more queries, the left child of q would be a leaf labeled accordingly. The right child of q similarly describes the behavior of M if the answer to q is "yes". Since M asks only $O(\log n)$ queries, the height of the tree is $O(\log n)$, and the number of nodes is bounded by a polynomial in the length of x.

For example, if M makes two queries on input x, the query tree for $M(x)$ might look like the tree in Figure 1. In this tree, q_{-1} is the very first query asked; q_0 is the query asked if $q_{-1} \notin \text{SAT}$; q_1 is the query asked if $q_{-1} \in \text{SAT}$. If $q_{-1} \notin \text{SAT}$ and $q_0 \notin \text{SAT}$, then M accepts x. Similarly if $q_{-1} \in \text{SAT}$ and $q_1 \in \text{SAT}$, then M also accepts x.

We say a path from the root to a leaf is an *accepting path* if the leaf is labeled ACCEPT. A path is a *rejecting path* if it ends at a leaf labeled REJECT. There may be many accepting paths and many rejecting paths in the tree for $M(x)$, but there is only one path from the root to a leaf for which all the queries are answered correctly. This path is called the *valid path*. $M^{\mathrm{SAT}}(x)$ accepts if and only if the valid path is an accepting path. That is, $M^{\mathrm{SAT}}(x)$ accepts if and only if the path that represents the true query answers leads to acceptance.

From the preceeding discussion, it should be clear that the following theorem is true.

Theorem 2 *The set*

$$T_{log} \stackrel{\text{def}}{=} \{T\#^n \mid T \textit{ is a query tree of CNF queries of height} \leq \log(n), \\ \textit{and the valid path of } T \textit{ is an accepting path}\}$$

is $\leq_{\mathrm{m}}^{\mathrm{P}}$*-complete for* $\mathrm{P}^{\mathrm{NP}[\log n]}$.

3 Complete Problems

In this section we show that UOCSAT is $\leq_{\mathrm{m}}^{\mathrm{P}}$-complete for $\mathrm{P}^{\mathrm{SAT}[\log n]}$. The proof involves reducing T_{log} to UOCSAT by encoding a query tree as a CNF formula such that the formula has a unique maximal set of simultaneously satisfiable clauses if and only if the valid path through the tree ends at a leaf labeled ACCEPT.

The encoding of a query tree T into a single CNF formula F is done in three major stages. First, each query q_i in T is reduced to two formulas $q_{i,y}$ and $q_{i,n}$ such that

$$\begin{aligned} q_i \in \mathrm{SAT} &\iff q_{i,y} \in \mathrm{UOCSAT}, \quad \text{and} \\ q_i \notin \mathrm{SAT} &\iff q_{i,n} \in \mathrm{UOCSAT}. \end{aligned}$$

Second, the $q_{i,n}$'s and $q_{i,y}$'s are used to encode each accepting path p_j in T as a single formula F_{p_j}.

Finally, we take the formulas $F_{p_1},\ldots,F_{p_m}$ encoding the accepting paths $p_1,\ldots,p_m$ and combine them into one formula F_T such that $F_T \in$ UOCSAT if and only if one of $p_1,\ldots,p_m$ is valid. Recall that there is exactly one valid path, and it could be either accepting or rejecting. Thus at most one and possibly none of $p_1,\ldots,p_m$ are valid, and this implies that at most one and possibly none of $F_{p_1},\ldots,F_{p_m}$ are in UOCSAT. Hence our F_T is designed so that $F_T \in$ UOCSAT if and only if exactly one of $F_{p_1},\ldots,F_{p_m}$ is in UOCSAT.

At each step of the reduction, we have to count the number of clauses and the number of simultaneously satisfiable clauses of formulas rather carefully. We use the following notation:

Let F be a CNF formula.

Definition 3 NC(F) *is the number of clauses in F.*

Definition 4 MC(F) *is the maximal number of clauses of F that can be simultaneously satisfied.*

Definition 5 UC(F) *is* NC(F) − MC(F).

Definition 6 *Let y be a Boolean literal (a variable or the negation of a variable), then $F \vee y$ is the CNF formula produced by inserting the literal y into each clause of F.*

The proofs of the following lemmas are left to the reader.

Lemma 7 $F \in \mathrm{SAT} \Longrightarrow F \in \mathrm{UOCSAT}$.

Lemma 8 *Let F be a CNF formula. Let F^k be F repeated (*AND*ed together) k times. Then*

$$\begin{aligned} \mathrm{NC}(F^k) &= k\mathrm{NC}(F), \\ \mathrm{MC}(F^k) &= k\mathrm{MC}(F), \\ \mathrm{UC}(F^k) &= k\mathrm{UC}(F), \quad \textit{and} \end{aligned}$$
$$F^k \in \mathrm{UOCSAT} \Longleftrightarrow F \in \mathrm{UOCSAT}.$$

Lemma 9 *Let $F_1,\ldots,F_k$ be CNF formulas with disjoint sets of variables. Let $F = F_1 \wedge \cdots \wedge F_k$. Then*

$$\begin{aligned} \mathrm{NC}(F) &= \sum_{i=1}^{k} \mathrm{NC}(F_i), \\ \mathrm{MC}(F) &= \sum_{i=1}^{k} \mathrm{MC}(F_i), \\ \mathrm{UC}(F) &= \sum_{i=1}^{k} \mathrm{UC}(F_i), \quad \textit{and} \end{aligned}$$
$$F \in \mathrm{UOCSAT} \Longleftrightarrow \forall i\ F_i \in \mathrm{UOCSAT}.$$

Theorem 10 UOCSAT *is* $\leq_m^P$*-complete for* $P^{SAT[\log n]}$.

Proof: We already know UOCSAT $\in P^{SAT[\log n]}$. We will show UOCSAT is $\leq_m^P$-hard for $P^{SAT[\log n]}$ by reducing T_{log} to UOCSAT. Let $T\#^n$ be a candidate for membership in T_{log}. Thus T is a query tree of height $\leq \log n$.

Step 1: *Reduce* $\overline{\text{SAT}}$ *to* UOCSAT. For each query q_i in T, reduce q_i to $q_{i,n}$ such that

$$q_i \notin \text{SAT} \iff q_{i,n} \in \text{UOCSAT}.$$

We do this reduction in several steps.

First, apply the reduction from $\overline{\text{SAT}}$ to USAT, the set of formulas that have exactly one satisfying assignment [BG]. Given q_i with variables $x_1, \ldots, x_k$, let

$$q'_{i,u} \stackrel{\text{def}}{=} (x_{new} \wedge x_1 \wedge \cdots \wedge x_k) \vee (\sim x_{new} \wedge q_i)$$

where x_{new} is a new variable. It is easy to see that

$$\begin{aligned} &q'_{i,u} \in \text{SAT}, \quad \text{and} \\ &q_i \in \text{SAT} \iff \text{there are at least 2 satisfying} \\ &\qquad\qquad\qquad\quad \text{assignments for } q'_{i,u}. \end{aligned}$$

Parsimoniously put $q'_{i,u}$ into CNF [Si], and let $q_{i,u}$ be the CNF formula. Since $q'_{i,u}$ and $q_{i,u}$ have the same number of satisfying assignments,

$$\begin{aligned} &q_{i,u} \in \text{SAT}, \quad \text{and} \\ &q_i \notin \text{SAT} \iff q_{i,u} \in \text{CNF-USAT} \end{aligned}$$

where CNF-USAT is the set of CNF formulas with exactly one satisfying assignment.

In later steps, the number of variables in the $q_{i,u}$'s affect the number of clauses in certain formulas. To keep things tidy, alter the $q_{i,u}$'s so that they all have the same number of variables. Let

$$v \stackrel{\text{def}}{=} \max_i \,(\text{number of variables of } q_{i,u}).$$

For each $q_{i,u}$ with fewer than v variables, add new variables and clauses

$$(y_{i,new_1})(y_{i,new_2})\cdots$$

so all the $q_{i,u}$'s have exactly v variables. Note that adding these new clauses does not affect whether or not $q_{i,u} \in$ CNF-USAT.

Reduce $q_{i,u}$ to $q_{i,n}$ so that

$$q_{i,u} \in \text{CNF-USAT} \iff q_{i,n} \in \text{UOCSAT}.$$

If $q_{i,u} = C_1 \cdots C_k$ (i.e. the clauses of $q_{i,u}$ are $C_1 \cdots C_k$) with variables $x_1, \ldots, x_v$, then let

$$q_{i,n} \stackrel{\text{def}}{=} C_1 \cdots C_k (x_1)(\sim x_1) \cdots (x_v)(\sim x_v).$$

Since $q_{i,u} \in$ SAT, all the clauses $C_1 \cdots C_k$ are simultaneously satisfiable. Each assignment to $x_1, \ldots, x_v$ satisfies exactly v of the $2v$ clauses $(x_1) \cdots (\sim x_v)$, and any two distinct assignments satisfy distinct sets of these clauses. Hence

$$\text{MC}(q_{i,n}) = \text{NC}(q_{i,n}) - v, \quad \text{and}$$
$$q_i \notin \text{SAT} \iff q_{i,u} \in \text{CNF-USAT} \iff q_{i,n} \in \text{UOCSAT}.$$

Thus we have reduced $\overline{\text{SAT}}$ to UOCSAT, and in polynomial time we can map q_i to $q_{i,n}$.

Step 2: *Reduce* SAT *to* UOCSAT. For each q_i, reduce q_i to $q_{i,y}$ such that

$$q_i \in \text{SAT} \iff q_{i,y} \in \text{UOCSAT}.$$

This is fairly straightforward since a CNF formula that is in SAT is already in UOCSAT. The formula merely has to be altered so that if all the clauses are not simultaneously satisfiable, then the uniqueness is lost. If $q_i = C_1 \cdots C_k$, let z_{new} be a new variable, and let

$$q_{i,y} \stackrel{\text{def}}{=} (C_1 \vee z_{new}) \cdots (C_k \vee z_{new})(C_1 \vee \sim z_{new}) \cdots (C_k \vee \sim z_{new}).$$

If $q_i \in$ SAT, then $q_{i,y} \in$ SAT which implies $q_{i,y} \in$ UOCSAT. If $q_i \notin$ SAT, then $\text{MC}(q_i) < k$ and $\text{MC}(q_{i,y}) = k + \text{MC}(q_i)$. This maximal number of clauses of $q_{i,y}$ can be achieved by setting z_{new} true and satisfying all of the first k clauses or by setting z_{new} false and satisfying all of the last k clauses. Hence $q_i \notin$ SAT implies $q_{i,y} \notin$ UOCSAT. Note that

$$q_i \in \text{SAT} \implies \text{MC}(q_{i,y}) = \text{NC}(q_{i,y}), \quad \text{and}$$
$$q_i \notin \text{SAT} \implies \text{MC}(q_{i,y}) < \text{NC}(q_{i,y}).$$

Step 3: *Combine the encoded accepting paths.* Step 4 shows how to encode each accepting path p_j of T into a CNF formula F_{p_j} such that

$$p_j \text{ is the valid path of } T \iff F_{p_j} \in \text{UOCSAT}.$$

F_{p_j} is basically an ANDing together of $q_{i,y}$'s and $q_{i,n}$'s; see Step 4 for the details. In this step, we show how to combine all the path formulas into a single formula F_T so that $F_T \in \text{UOCSAT} \iff T\#^n \in T_{log}$. In doing so, it will become clear what constraints have to be placed on the F_{p_j}'s.

Let $p_1,\ldots,p_m$ be all the accepting paths of T, and let $F_{p_1},\ldots,F_{p_m}$ be the corresponding formulas. Since the height of $T \leq \log n$, $m \leq n$. If $m = 0$, then $T\#^n \notin T_{log}$, and we can simply let F_T be some fixed formula that is not in UOCSAT. If $m = 1$, then we can let $F_T \stackrel{\text{def}}{=} F_{p_1}$. The rest of this step is devoted to the case where $m > 1$.

F_T is constructed as follows.

Rename the variables in $F_{p_1},\ldots,F_{p_m}$ so each formula has its own set of variables.

Introduce new variables $y_1,\ldots,y_m$.

Let EXACT1 be the CNF formula that is satisfied if and only if exactly one of $y_1,\ldots,y_m$ is set to true.

$$\begin{array}{llllll}
\text{EXACT1} \stackrel{\text{def}}{=} (y_1 \vee \cdots \vee y_m) & (\sim y_1 \vee \sim y_2) & \cdots & \cdots & (\sim y_1 \vee \sim y_m) \\
 & & (\sim y_2 \vee \sim y_3) & \cdots & (\sim y_2 \vee \sim y_m) \\
 & & \ddots & & \vdots \\
 & & & & (\sim y_{m-1} \vee \sim y_m).
\end{array}$$

Note that $\text{NC(EXACT1)} < m^2 \leq n^2$. Thus EXACT1 can be constructed in polynomial time.

Let

$$c = \max_{j=1,\ldots,m} \text{NC}(F_{p_j}).$$

Finally,

$$F_T \stackrel{\text{def}}{=} (F_{p_1} \vee \sim y_1)(F_{p_2} \vee \sim y_2) \cdots (F_{p_m} \vee \sim y_m)\text{EXACT1}^{c+1}.$$

Observe that F_T is set up so that if y_i is set to false, then the clauses from F_{p_i} are all satisfied. If y_i is set to true, then the maximal number of the clauses from F_{p_i} that are satisfiable is $\mathrm{MC}(F_{p_i})$. EXACT1 is designed to choose one y_i to set to true, and it is repeated enough times $(c+1)$ so that it weighs more heavily than any of the F_{p_j}'s in the determination of $\mathrm{MC}(F_T)$.

We claim that the maximum number of clauses are satisfied when exactly one y_i is set to true. To see why, let

$$N \stackrel{\text{def}}{=} \sum_{j=1}^{m} \mathrm{NC}(F_{p_j}), \quad \text{and}$$

$$E \stackrel{\text{def}}{=} \mathrm{NC}(\text{EXACT1}).$$

If all the y_j's are set to false, then all the F_{p_j} clauses are satisfied (for $j = 1,\ldots,m$) and all but the first clause of EXACT1 are satisfied. If we let n_0 be the number of satisfied clauses with none of the y_j's set to true, we have

$$n_0 = N + (c+1)E - (c+1). \tag{1}$$

If exactly one y_j is set to true, then all the clauses of EXACT1 are satisfied, and all the clauses from all but one of the F_{p_j}'s are satisfied. Since the F_{p_j} that is "chosen" contains at most c clauses, at most c clauses are not satisfied. Thus if n_1 is the number of clauses satisfied when exactly one y_j is set to true, we know

$$n_1 \geq N - c + (c+1)E \tag{2}$$

Stringing equations (1) and (2) together gives us

$$n_1 \geq N - c + (c+1)E > N + (c+1)E - (c+1) = n_0.$$

So setting one y_j to true satisfies more clauses than setting them all to false.

What happens if two y_j's are set to true? One of the clauses in EXACT1 is not satisfied, namely the clause containing the negations of the two variables set to true. Thus if n_2 is the maximum number of satisfiable clauses with two y_j's true,

$$N + (c+1)E - (c+1) \geq n_2, \quad \text{and}$$

$$n_1 \geq N - c + (c+1)E > N + (c+1)E - (c+1) \geq n_2.$$

Thus setting one variable true satisfies more clauses than setting two variables to true.

Things get even worse if more y_j's are set to true since this causes fewer of the clauses of EXACT1 to be satisfied. We conclude that the most clauses of F_T are satisfied when exactly one y_j is set to true.

Which y_j should be set to true to satisfy the maximal number of clauses of F_T? The answer is the y_j such that F_{p_j} has the least number of "unsatisfiable" clauses. That is, we want to pick j so that we maximize

$$(\sum_{i \neq j} \mathrm{NC}(F_{p_i})) + \mathrm{MC}(F_{p_j})$$

which equals

$$(\sum_{i=1}^{m} \mathrm{NC}(F_{p_i})) - (\mathrm{NC}(F_{p_j}) - \mathrm{MC}(F_{p_j})).$$

This is the same as picking j so as to minimize $\mathrm{NC}(F_{p_j}) - \mathrm{MC}(F_{p_j})$. Recall that

$$\mathrm{UC}(F_{p_j}) \stackrel{\mathrm{def}}{=} \mathrm{NC}(F_{p_j}) - \mathrm{MC}(F_{p_j}).$$

In summary, the most clauses of F_T are satisfied if we pick the j such that $\mathrm{UC}(F_{p_j})$ is minimal, set y_j to true (and all other y_i's to false), and then pick an assignment that satisfies the most clauses of F_{p_j}. This satisfies all the clauses that are not in $(F_{p_j} \vee \sim y_j)$. Hence this maximum number of satisfiable clauses is achieved by a unique set of clauses if and only if there is a unique j with $\mathrm{UC}(F_{p_j})$ minimal, and the maximal group of simultaneously satisfiable clauses of F_{p_j} is unique. That is,

$$F_T \in \mathrm{UOCSAT} \iff (\exists! j)\ \begin{array}{l}(F_{p_j} \in \mathrm{UOCSAT}, \quad \text{and} \\ (\forall i \neq j)\ \mathrm{UC}(F_{p_j}) < \mathrm{UC}(F_{p_i})).\end{array} \tag{3}$$

We now know what constraints to place on $F_{p_1}, \ldots, F_{p_m}$ to ensure that $F_T \in \mathrm{UOCSAT} \iff T\#_n \in T_{log}$. One constraint that we have mentioned already is

$$p_j \text{ is valid} \iff F_{p_j} \in \mathrm{UOCSAT}. \tag{4}$$

The other constraint is that we need to know that if p_j is valid, then the maximal number of clauses of F_T is satisfied when y_j is set to true. That is

$$p_j \text{ is valid} \implies (\forall i \neq j)\ \mathrm{UC}(F_{p_j}) < \mathrm{UC}(F_{p_i}). \tag{5}$$

If we can design $F_{p_1}, \ldots, F_{p_m}$ so that properties (4) and (5) hold, then

$$
\begin{aligned}
F_T \in \mathrm{UOCSAT} &\Longrightarrow \text{some } F_{p_j} \in \mathrm{UOCSAT} && \text{(by (3))}. \\
&\Longrightarrow p_j \text{ is valid} && \text{(by (4))}. \\
&\Longrightarrow T\#^n \in T_{log} && \text{(since } p_j \text{ is an accepting path)}.
\end{aligned}
$$

Going the other direction,

$$
\begin{aligned}
T\#^n \in T_{log} &\Longrightarrow \text{some accepting path } p_j \text{ is valid}. \\
&\Longrightarrow \forall i \neq j \ \mathrm{UC}(F_{p_j}) < \mathrm{UC}(F_{p_i}), \\
&\qquad \text{and } F_{p_j} \in \mathrm{UOCSAT} && \text{(by (4) and 5))}. \\
&\Longrightarrow F_T \in \mathrm{UOCSAT} && \text{(since there is only one valid path)}.
\end{aligned}
$$

This means the proof will be complete when we show how to encode the accepting paths into the formulas $F_{p_1}, \ldots, F_{p_m}$ so that properties (4) and (5) hold.

Step 5: *Encode the accepting paths* so that properties (4) and (5) hold.

First, rename variables in all the formulas $q_{i,y}$ and $q_{i,n}$ so each has its own unique set of variables.

Each F_{p_j} is simply the ANDing together of a sequence of $q_{i,n}$'s and $q_{i,y}$'s with each $q_{i,n}$ and $q_{i,y}$ repeated a certain number of times. If query q_i is on path p_j, and p_j follows the "yes" branch from q_i, then F_{p_j} will have copies of $q_{i,y}$ ANDed into it. If q_i is on path p_j and p_j follows the "no" path from q_i, then F_{p_j} will have copies of $q_{i,n}$.

We claim that p_j is valid $\Longleftrightarrow F_{p_j} \in \mathrm{UOCSAT}$. To see why, recall that $q_i \in \mathrm{SAT} \Longleftrightarrow q_{i,y} \in \mathrm{UOCSAT}$, and $q_i \notin \mathrm{SAT} \Longleftrightarrow q_{i,n} \in \mathrm{UOCSAT}$. Thus we have the following chain:

$$
\begin{aligned}
p_j \text{ is valid} &\Longleftrightarrow \text{each query answer on } p_j \text{ is correct}. \\
&\Longleftrightarrow \text{the } q_{i,n}\text{'s and } q_{i,y}\text{'s that are ANDed together} \\
&\qquad \text{to make } F_{p_j} \text{ are all in UOCSAT}. \\
&\Longleftrightarrow F_{p_j} \in \mathrm{UOCSAT} && \text{(by Lemmas 8 and 9)}.
\end{aligned}
$$

How many times do we repeat $q_{i,n}$ and $q_{i,y}$ in the construction of these formulas? The idea is to repeat a $q_{i,n}$ or $q_{i,y}$ enough times so that any F_{p_k} representing path p_k that splits off from the valid path at q_i will pick up so many "unsatisfiable" clauses with $q_{i,y}$ or $q_{i,n}$ that $\mathrm{UC}(F_{p_k})$ will be large, even if all the later queries on p_k contribute no more "unsatisfiable" clauses. This boils down to repeating the $q_{i,n}$'s and $q_{i,y}$'s that are higher in the tree more than ones that are lower in the tree.

Recall that

$$\begin{array}{rcl} q_i \in \mathrm{SAT} & \Longrightarrow & \mathrm{UC}(q_{i,y}) = 0, \\ q_i \notin \mathrm{SAT} & \Longrightarrow & \mathrm{UC}(q_{i,y}) \geq 1, \quad \text{and} \\ \mathrm{UC}(q_{i,n}) & = & v \quad \text{regardless of whether } q_i \in \mathrm{SAT}. \end{array}$$

Let q_i be a query of height h in the tree. If path p_j goes through q_i with a "yes" answer, i.e. if F_{p_j} is to contain $q_{i,y}$, then repeat $q_{i,y}$ $2^{h+1}v$ times in F_{p_j}. If path p_j goes through q_i with a "no" answer, then repeat $q_{i,n}$ in F_{p_j} 2^h times. Since the height of the tree is bounded by $\log n$, the construction of F_{p_j} with all these repetitions can be done in polynomial time.

Let p_{valid} be the valid path through the tree, and let p_j be another path. We claim that $\mathrm{UC}(F_{p_{valid}}) < \mathrm{UC}(F_{p_j})$. Let q_i of height h be the query where p_{valid} and p_j diverge. Above q_i, p_{valid} and p_j are identical. Hence the numbers of clauses of $\mathrm{UC}(F_{p_{valid}})$ and $\mathrm{UC}(F_{p_j})$ attributable to queries above q_i are equal. Call that number A;

$$A \stackrel{\text{def}}{=} \mathrm{UC}(F_{p_{valid}} \text{"above } q_i\text{"}) = \mathrm{UC}(F_{p_j} \text{"above } q_i\text{"}).$$

Below q_i, $\mathrm{UC}(F_{p_{valid}})$ is maximal if all the query strings are not in SAT. Thus

$$\mathrm{UC}(F_{p_{valid}} \text{"below } q_i\text{"}) \leq \sum_{k=1}^{h-1} 2^k v < v \sum_{k=0}^{h-1} 2^k = v(2^h - 1).$$

$\mathrm{UC}(F_{p_j}$"below q_i"$)$ is minimal if p_j is correct below q_i, and all the query strings on the path are in SAT. In this case no more "unsatisfiable" clauses are picked up below q_i, hence $\mathrm{UC}(F_{p_j}$"below q_i"$) \geq 0$.

Finally, consider the part of $\mathrm{UC}(F_{p_{valid}})$ and $\mathrm{UC}(F_{p_j})$ attributable to q_i. If $q_i \in \mathrm{SAT}$, then $q_{i,y}$ is repeated $2^{h+1}v$ times in $F_{p_{valid}}$, and $q_{i,n}$ is repeated 2^h times in F_{p_j}. Then

$$\begin{array}{rcl} \mathrm{UC}((q_{i,y})^{2^{h+1}v}) & = & 0, \quad \text{and} \\ \mathrm{UC}((q_{i,n})^{2^h}) & = & 2^h v. \end{array}$$

Pulling together these bounds for $\mathrm{UC}(F_{p_{valid}})$ and $\mathrm{UC}(F_{p_j})$ under the assumption that $q_i \in \mathrm{SAT}$, we have

$$\mathrm{UC}(F_{p_{valid}}) \;<\; A + 0 + v(2^h - 1) \;<\; A + 2^h v + 0 \;\leq\; \mathrm{UC}(F_{p_j}).$$

If $q_i \notin \mathrm{SAT}$, then

$$\begin{array}{lcl} q_{i,n} \text{ is repeated } 2^h \text{ times in } F_{p_{valid}}, & & \\ q_{i,y} \text{ is repeated } 2^{h+1}v \text{ times in } F_{p_j}, & & \\ \mathrm{UC}((q_{i,n})^{2^h}) & = & 2^h v, \quad \text{and} \\ \mathrm{UC}((q_{i,y})^{2^{h+1}v}) & \geq & 2^{h+1} v. \end{array}$$

This implies

$$\mathrm{UC}(F_{p_{valid}}) \;<\; A + 2^h v + v(2^h - 1) \;<\; A + 2^{h+1} v + 0 \;\leq\; \mathrm{UC}(F_{p_j}).$$

In either case $\mathrm{UC}(F_{p_{valid}}) < \mathrm{UC}(F_{p_j})$. Thus we have succeeded in encoding $p_1,\ldots,p_m$ into $F_{p_1},\ldots,F_{p_m}$ while satisfying the necessary constraints.

This concludes the proof of Theorem 10. □

In pondering this result, particularly in light of Papadimitriou's result about UOTSP, the obvious question is what about UOASAT? What about the stronger uniqueness condition? Is UOASAT $\leq_{\mathrm{m}}^{\mathrm{P}}$-complete for $\mathrm{P}^{\mathrm{NP}[\log n]}$? At present, this is an open question.

The requirement of a single assignment that satisfies the maximal number of clauses is much more stringent than the requirement that all the best assignments satisfy the same clauses. The techniques developed for UOCSAT simply do not work for UOASAT.

UOASAT is very similar to USAT. Both of these languages are easily seen to be co-NP-hard, but it is not clear if they are NP-hard. Blass and Gurevich have studied USAT [BG]. They have shown that USAT is $\leq_{\mathrm{m}}^{\mathrm{P}}$-complete for D^{P} if and only if $\mathrm{SAT} \leq_{\mathrm{m}}^{\mathrm{P}} \mathrm{USAT}$ and that D^{P} can be relativized so that the relativized versions of USAT can be made complete or incomplete for D^{P}. This indicates that resolving whether or not USAT is $\leq_{\mathrm{m}}^{\mathrm{P}}$-complete for D^{P} is probably going to be very difficult. An interesting open problem is whether similar results can be achieved for UOASAT and $\mathrm{P}^{\mathrm{NP}[\log n]}$.

What about the uniqueness conditions of the other "unique optimal" problems? Are these conditions too stringent to allow us to show their completeness? At present, the answer is yes. The clique problem, however, does provide us with another example of how relaxing the uniqueness condition lets us prove completeness.

Definition 11 UOGCLIQUE *(Unique Optimal Grouped Clique) is the set of undirected graphs whose vertices are partitioned into groups, with no edges between vertices in the same group, with the property that all the maximal cliques contain vertices from the same set of groups.*

Theorem 12 UOGCLIQUE *is* $\leq_m^P$*-complete for* $\mathrm{P}^{\mathrm{NP}[\log n]}$.

Proof: A $\mathrm{P}^{\mathrm{NP}[\log n]}$ algorithm to recognize UOGCLIQUE is similar to the algorithm for recognizing UOCLIQUE.

We prove UOGCLIQUE is hard for $\mathrm{P}^{\mathrm{NP}[\log n]}$ by reducing UOCSAT to UOGCLIQUE. Actually, the standard reduction from SAT to CLIQUE [AHU] suffices to reduce UOCSAT to UOGCLIQUE.

Let F be a CNF formula with clauses $C_1 \cdots C_k$. Each clause C_i is reduced to a group of vertices, one for each literal of C_i. The graph contains edges between each pair of vertices from different groups except for the pairs of vertices derived from a variable and its negation.

Each clique in the graph consists of vertices from different groups whose corresponding literals are consistent. That is, the Boolean values that make the corresponding literals true form a valid, if partial, assignment to the variables of F. Thus the groups containing the vertices of a clique correspond to the clauses that are satisfied by the assignment. Hence there is a unique set of groups containing the maximal cliques if and only if there is a unique maximal set of satisfiable clauses. □

The introduction of the groups serves only as a mechanism for relaxing the definition of uniqueness, and structurally the groups obviously parallel the clauses very closely.

We are not aware of ways to relax the uniqueness conditions on the other similarly defined problems that permit us to prove completeness.

Krentel has found other complete problems for $\mathrm{P}^{\mathrm{NP}[\log n]}$ [Kr]. He has studied polynomial time functions that access NP-complete oracles and has shown that all such functions that make $O(\log n)$ queries can be reduced to the function that outputs the maximum number of simultaneously satisfiable clauses of a CNF formula. Thus knowing the maximum number of simultaneously satisfiable clauses is "equivalent" to $O(\log n)$ queries. A similar result holds for the maximum clique size of a graph. As corollaries of these results, he shows that SAT-MOD-k and CLIQUE-MOD-k are $\leq_{\mathrm{m}}^{\mathrm{P}}$-complete for $\mathrm{P}^{\mathrm{NP}[\log n]}$.

From a functional point of view, knowing the number of simultaneously satisfiable clauses of a formula is "complete" for polynomial time functions that make $O(\log n)$ queries. Thus, from a language point of view, a language whose recognizer would seem to require knowing the number of satisfiable clauses would be a good candidate for a complete language for $\mathrm{P}^{\mathrm{NP}[\log n]}$ (e.g. UOCSAT and UOASAT). Of course complete languages must also have a certain richness and robustness. While we have shown that UOCSAT is robust enough, UOASAT simply may not have the required richness to be complete. On the other hand, we may just require different proof techniques to the show the completeness of UOASAT.

4 Sparseness Results

Much attention has been focused on questions about sparse sets and NP. One of the major areas of concern is whether NP is reducible to a sparse set under any of the standard polynomial time reducibilities.

If NP is sparse reducible, then we could in principle spend large amounts of computational resourses to compute and store the relatively few strings in the sparse set up to a certain length. Once constructed, this table of strings could be used to solve many instances of NP problems in deterministic polynomial time.

Another more theoretical issue that has spurred on the research about sparse sets is the conjecture by Berman and Hartmanis that all NP-complete sets (under $\leq_{\mathrm{m}}^{\mathrm{P}}$) are polynomial time isomorphic [BH]. If their conjecture is true, then there can be no NP-complete sparse sets because no sparse set can be polynomial time isomorphic to the known, relatively "fat", NP-complete sets.

Mahaney pretty much sewed up the question for $\leq_{\mathrm{m}}^{\mathrm{P}}$ by showing that if there is a sparse NP-complete set, then P = NP, and if there is a sparse NP-hard set, then there is a sparse NP-complete set [Ma].

The results concerning sparse $\leq_T^P$-hard sets are a bit less dramatic. Karp and Lipton (along with the results of Meyer appearing in [BH]) proved that if NP has a sparse $\leq_T^P$-hard set (i.e. $NP \subseteq P^S$), then $PH \subseteq \Sigma_2^P$ (NP^{SAT}) [KL]. They also noted that if there exist any such sparse sets, then there exists one that is in Σ_2^P. Mahaney added that if NP has a sparse $\leq_T^P$-complete set (that is, the sparse oracle is itself in NP), then $PH \subseteq \Delta_2^P$ (P^{SAT}) [Ma,Ma2]. Long then generalized Mahaney's result by showing that if there is a sparse set S anywhere in Δ_2^P such that $NP \subseteq P^S$, then $PH \subseteq \Delta_2^P$ [Lo].

Mahaney's result can be broken into two lemmas. The first states that if N is an NP^S machine ($NP^S = NP^{SAT} = \Sigma_2^P$ if S is a sparse oracle for NP), then there is an NP machine that can simulate $N^S(x)$ if it is given a table of the strings in S up to the length of the longest query that $N^S(x)$ makes. Since S is sparse, and N is a polynomial time machine, the number of strings in the table (and the total length of the table) is bounded by a polynomial in the length of x. The second lemma states that if S is sparse and $S \in NP$, then there is a P^{SAT} machine that can enumerate all the strings in S of length $\leq n$ on input 1^n, that is, S is P^{SAT}-printable. This enumeration machine does a binary search making at least one query for each bit of each string and hence requires polynomially many queries.

Thus Mahaney proves $NP^{SAT} \subseteq P^{SAT}$ by showing that for every NP^S machine N, there is a P^{SAT} machine that generates an initial segment of S, and then accepts if the NP simulator of N^S accepts.

Long's proof follows this same basic pattern; enumerate enough of the sparse oracle so that an NP oracle machine can be simulated by an NP machine that accesses the enumerated strings (as part of its input) instead of an oracle.

Long's simulation works as follows. Let N be an NP oracle machine. N', an NP machine without an oracle, can simulate $N^{SAT}(x)$ by using a P^S acceptor for SAT to answer the queries if N' is given enough of the strings in S. The length of the queries N makes is bounded by a polynomial in the length of x. So N' need only be able to handle queries up to that length. Similarly, the length of the queries made by any P^S acceptor for SAT is bounded by a polynomial. So the number of strings in S that N' needs is bounded by a polynomial in the length of x.

How can a $\mathrm{P}^{\mathrm{SAT}}$ machine enumerate the strings N' needs? Recall that by the self-reducibility of SAT, if there is a P^S machine that accepts SAT, then there is a P^S machine that outputs satisfying assignments. That is, there exists a deterministic oracle machine M such that

$$M^S(F) = \begin{cases} \text{satisfying assignment} & \text{if } F \in \mathrm{SAT} \\ \text{“no”} & \text{if } F \notin \mathrm{SAT} \end{cases}$$

If we run $M(F)$ using a subset of S as the oracle, and $M(F)$ produces an output that is incorrect, then we know that one of the queries it made was incorrectly answered as "no". Hence one of its queries is a string that is in S and not in the subset of S. Thus we could run $M(F)$ using the subset as an oracle to enumerate its queries, and then run the $\mathrm{P}^{\mathrm{SAT}}$ machine that accepts S on the queries to find a new string to add to our subset of S. In this way a $\mathrm{P}^{\mathrm{SAT}}$ machine, M_{enum}, can build up bigger and bigger subsets of the set of strings in S up to a given length.

Using its current table of strings in S (call this table A), M_{enum} can determine whether or not it has all the strings it needs by asking its SAT-oracle if there exists a satisfiable formula F such that the output of $M^A(F)$ does not satisfy F. This question can be encoded as a SAT-question since there is an NP machine N_M that on input $\langle A, \#^n \rangle$ guesses a string F of length $\leq n$, verifies that $F \in \mathrm{SAT}$, and then accepts if $M^A(F)$ does not produce a satisfying assignment (n is the maximum length of queries N would ask on input x). That is, N_M guesses a witness to the fact that M^A does not work correctly on all strings up to length n. If $N_M\langle A, \#^n \rangle$ does not accept, then M^A does work correctly implying N' can use M^A (given A as part of its input) to answer the queries during its simulation of $N^{\mathrm{SAT}}(x)$. If $N_M\langle A, \#^n \rangle$ does accept, M_{enum} can do a binary search to get hold of a witness, F, that causes N_M to accept. M_{enum} can then use F as above to get hold of at least one new string in S to add to A.

This clever enumeration gives us all the strings necessary to simulate an $\mathrm{NP}^{\mathrm{SAT}}$ machine with an NP machine, and the enumeration can be done by a $\mathrm{P}^{\mathrm{SAT}}$ machine. Hence we know that if there is a sparse oracle in $\mathrm{P}^{\mathrm{SAT}}$, then $\mathrm{PH} \subseteq \mathrm{P}^{\mathrm{SAT}}$.

The enumerations in Mahaney's and Long's proofs take polynomially many queries (at least one per enumerated string). It is hard to imagine any way of enumerating the sparse oracle without making at least one query per string. Hence showing any further collapse with a proof that requires enumerating the oracle seems very difficult.

We show that if the sparse oracle is in NP, then we can simulate any $\mathrm{P}^{\mathrm{SAT}}$machine that may make many queries with only one query, and this can be done without enumerating the sparse oracle. Rather than finding the strings of the oracle, we need only compute the census function of the oracle. Computing the value of the census function is just another application of binary search over polynomially many elements. Thus the existence of such an oracle implies that $\mathrm{P}^{\mathrm{SAT}} \subseteq \mathrm{P}^{\mathrm{SAT}[\log n]}$.

Definition 13 *For any set of strings* S, $\mathrm{CENSUS}_{\mathrm{S}}(n)$ *is the number of strings in* S *whose length is less than or equal to* n.

If S is sparse, then $\mathrm{CENSUS}_{\mathrm{S}}(\cdot)$ is bounded by a polynomial.

Theorem 14 *If* S *is sparse, and* S *is* $\leq^{\mathrm{P}}_{\mathrm{T}}$*-complete for* NP, *then* $\mathrm{PH} \subseteq \mathrm{P}^{\mathrm{SAT}[\log n]}$.

Proof: Assume S is sparse, S $\in$ NP, and SAT $\in \mathrm{P}^{\mathrm{S}}$. Let $h(\cdot)$ be a polynomial bound on $\mathrm{CENSUS}_{\mathrm{S}}(\cdot)$. From Karp, Lipton, and Mahaney we know then that $\mathrm{PH} \subseteq \mathrm{P}^{\mathrm{SAT}}$. We will show that $\mathrm{P}^{\mathrm{SAT}} \subseteq \mathrm{P}^{\mathrm{SAT}[\log n]}$.

Let L be in $\mathrm{P}^{\mathrm{SAT}}$, $L = \mathrm{L}(M_o^{\mathrm{SAT}})$. Since SAT $\in \mathrm{P}^{\mathrm{S}}$, there is a polynomial time machine M_1 such that $L = \mathrm{L}(M_1^{\mathrm{S}})$. Since M_1 is a polynomial time machine, there is a polynomial bound on the size of the longest query string M_1 asks on an input of length n. Let $g(n)$ be the bound.

We will describe a $\mathrm{P}^{\mathrm{SAT}[\log n]}$ machine M that will accept L.

Given x, a candidate for membership in L with $|x| = n$, M first computes $g(n)$, an upper bound on the length of the longest query M_1 would ask on input x. Then M computes $h(g(n))$ which is a bound on the number of strings in S that have length less than or equal to $g(n)$. Thus

$$0 \leq \mathrm{CENSUS}_{\mathrm{S}}(g(n)) \leq h(g(n)).$$

M then does a binary search on the numbers from 0 to $h(g(n))$ to determine $\mathrm{CENSUS}_{\mathrm{S}}(g(n))$. There is an NP machine N that on input 0^i1^j guesses i strings whose length is at most j and accepts if all the strings are distinct and in S. Thus $0^i1^j \in \mathrm{L}(N)$ if and only if $\mathrm{CENSUS_S}(j) \geq i$. Whether or not $0^i1^j \in \mathrm{L}(N)$ can be determined by one query to SAT. Hence the question of whether or not $i = \mathrm{CENSUS_S}(g(n))$ can be answered by two queries (is $\mathrm{CENSUS_S}(g(n)) \geq i$, and is $\mathrm{CENSUS_S}(g(n)) \not\geq i+1$?). Then the binary search of $0,\ldots,h(g(n))$ to determine $\mathrm{CENSUS_S}(g(n))$ takes at most $2\log(h(g(n)))$ queries - which is $O(\log n)$. The queries can be determined in time bounded by a polynomial in n since there is a polynomial time reduction from the set of strings accepted by N to SAT.

Once M has computed $\mathrm{CENSUS_S}(g(n))$, it needs only one more query to determine whether or not $x \in L$. The query is another "SAT encoded" question of the form "does a particular NP machine accept a particular string?". The particular NP machine that M will query about, N_1, behaves as follows. On input $\langle 0^i, 1^j, z\rangle$, N_1 deterministically simulates $M_1^S(z)$. When M_1 would query the oracle about a string q, N_1 does some nondeterministic computations.

First it runs $N_{\mathrm{yes}}(q)$ where N_{yes} is an NP machine that accepts S. If $N_{\mathrm{yes}}(q)$ accepts, $q \in \mathrm{S}$, and the answer M_1 would get from its S-oracle would be "yes". So if $N_{\mathrm{yes}}(q)$ accepts, N_1 continues the simulation from M_1's "yes" state. If $N_{\mathrm{yes}}(q)$ does not accept, N_1 still does not know that $q \notin \mathrm{S}$ since N_{yes} may simply have guessed incorrectly. So N_1 runs another NP machine N_{no} on input $\langle 0^i, 1^j, q\rangle$. N_{no} guesses i strings of length at most j and accepts if all the strings are distinct, all are different from q, all are in S, and $|q| \leq j$. Note that if $|q| \leq j$ and $i = \mathrm{CENSUS_S}(j)$, then N_{no} can guess i distinct strings in S that differ from q if and only if $q \notin \mathrm{S}$. Hence if $|q| \leq j$ and $i = \mathrm{CENSUS_S}(j)$, then $\langle 0^i, 1^j, q\rangle \in \mathrm{L}(N_{\mathrm{no}}) \iff q \notin \mathrm{S}$. N_{no} accepts a sort of pseudo-complement of S. If N_{no} is given the correct i and j, it can be used to accept the complement of S. If $N_{\mathrm{no}}(\langle 0^i, 1^j, q\rangle)$ accepts, N_1 continues the simulation of M_1 from its "no" state. If N_{no} does not accept, N_1 does not know how to deal with q and just halts without accepting.

If N_1 gets through all the queries M_1 would make with acceptance from one of N_{yes} or N_{no}, it accepts or rejects as M_1 would.

Consider what N_1 would do on input $\langle 0^{\mathrm{CENSUS_S}(g(n))}, 1^{g(n)}, x\rangle$. Since M_1 on input x only asks queries that are no longer than $g(n)$, for any query

q in M_1's computation on x

$$\langle 0^{\mathrm{CENSUS}_{\mathrm{S}}(g(n))}, 1^{g(n)}, q\rangle \in \mathrm{L}(N_{\mathrm{no}}) \iff q \notin \mathrm{S}.$$

Thus in N_1's simulation of $M_1(x)$, if N_{yes} or N_{no} accept their input for a particular query q, N_1 is going to continue its simulation with the correct query answer. Since each query string q is either in S or not in S, there is some sequence of guesses that will cause exactly one of $N_{\mathrm{yes}}(q)$ or $N_{\mathrm{no}}(\langle 0^{\mathrm{CENSUS}_{\mathrm{S}}(g(n))}, 1^{g(n)}, q\rangle)$ to accept. Hence

$$\langle 0^{\mathrm{CENSUS}_{\mathrm{S}}(g(n))}, 1^{g(n)}, x\rangle \in \mathrm{L}(N_1) \iff x \in \mathrm{L}(M_1^{\mathrm{S}}) \iff x \in L.$$

M uses its values of $\mathrm{CENSUS}_{\mathrm{S}}(g(n))$ and $g(n)$ to construct one more query, "is $\langle 0^{\mathrm{CENSUS}_{\mathrm{S}}(g(n))}, 1^{g(n)}, x\rangle \in \mathrm{L}(N_1)$?", and accepts x if and only if the answer is "yes". □

Hartmanis has recently shown that there exist relativized worlds in which NP has a sparse $\leq_{\mathrm{T}}^{\mathrm{P}}$-complete set (implying the PH collapses to $\mathrm{P}^{\mathrm{NP}[\log n]}$), but the PH collapses no further than $\mathrm{P}^{\mathrm{NP}[\log n]}$ [Ha]. Thus we know that Theorem 14 is in some sense optimal.

In any relativization where the PH collapes only to $\mathrm{P}^{\mathrm{NP}[\log n]}$, P is not NP. Hence Hartmanis' result implies that it is possible to collapse the PH (including P^{NP}) down to $\mathrm{P}^{\mathrm{NP}[\log n]}$ and still keep the relativized versions of P and NP distinct. Thus Krentel's theorem that $\mathrm{FP}^{\mathrm{NP}} \subseteq \mathrm{FP}^{\mathrm{NP}[\log n]} \Longrightarrow \mathrm{P} = \mathrm{NP}$ does not hold for the languages classes $\mathrm{P}^{\mathrm{NP}[\log n]}$ and P^{NP}.

Corollary 15 *There exist relativized worlds in which* $\mathrm{P}^{\mathrm{NP}} \subseteq \mathrm{P}^{\mathrm{NP}[\log n]}$ *and* $\mathrm{P} \neq \mathrm{NP}$.

5 Conclusion

The results of the preceeding sections along with the results of [Kr] and [Pa] bring up several open questions.

First, are UOASAT and the other "unique optimal" languages $\leq_{\mathrm{m}}^{\mathrm{P}}$-complete for $\mathrm{P}^{\mathrm{SAT}[\log n]}$?

Second, the collapse of P^{SAT} down to $P^{SAT[\log n]}$ would seem to say a lot about the "accessability" of SAT to deterministic polynomial time machines. That is, if polynomially many queries could be crammed into logarithmically many queries, then it would seem that deterministic polynomial time machines must be able to tell something about the satisfiability of formulas. An interesting tangent would be to explore the consequences of $P^{SAT} \subseteq P^{SAT[\log n]}$ and $PH \subseteq P^{SAT[\log n]}$ (note that Hartmanis' relativization indicates that this is probably a very difficult direction to pursue).

Another direction would be to explore the relationships among FP^{NP}, $FP^{NP[\log n]}$, P^{NP}, and $P^{NP[\log n]}$. These relationships seem rather muddy in light of Krentel's work and the results of section 4.

6 Acknowledgements

I am indebted to Juris Hartmanis for his encouragement, patience, and many helpful discussions while this work took form.

References

[AHU] A. Aho, J. Hopcroft, J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[BG] A. Blass and Y. Gurevich, "On the Unique Satisfiability Problem", *Information and Control* 55 (1982), 80-88.

[BH] L. Berman and J. Hartmanis, "On Isomorphisms and Density of NP and Other Complete Sets", *SIAM J. Comput.* 6 (1977), 305-327.

[Ha] J. Hartmanis, private communication.

[KL] R.M. Karp and R.J. Lipton, "Some Connections Between Nonuniform and Uniform Complexity Classes", *Proceedings of the 12th ACM Symposium on the Theory of Computation* (1980), 302-309.

[Kr] M. Krentel, "The Complexity of Optimization Problems", *Proceedings of the 18th ACM Symposium on the Theory of Computation* (1986), 69-76.

[Lo] T. J. Long, "A Note on Sparse Oracles for NP", *JCSS* 24 (1981), 224-232.

[Ma] S. Mahaney, "Sparse Complete Sets for NP: Solution of a Conjecture of Berman and Hartmanis", *Proceedings of the 21st IEEE Foundations of Computer Science Symposium* (1980), 54-60.

[Ma2] S. Mahaney, "Sparse Complete Sets for NP: Solution of a Conjecture of Berman and Hartmanis", *JCSS* 25 (1982), 130-143.

[Pa] C.H. Papadimitriou, "On the Complexity of Unique Solutions", *J. ACM* 31 (1984), 392-400.

[Si] J. Simon, "On Some Central Problems in Computational Complexity", Ph. D. Disseration, Cornell University, Ithaca, New York, 1975, pp. 80-82.