1 Introduction

The human genome, the internet, particle accelerators - the modern world offers many sources that flood us with information we wish to analyze and understand. However, calculations that are feasible for relatively small input sizes may become infeasible for inputs of huge size. In the latter case we may even wish to devise an algorithm that runs in time that is smaller than the input size itself. Such an algorithm will obviously be able to access only a limited part of its input.

An additional setting which is similar in some ways is when access to any part of the input is prohibitively "expensive". In such cases we would like to devise an algorithm that would only access a small number of bits, even if it takes a long time to compute. Access to any part of the input may, for example, require setting up a sample for an electron microscope, or checking for radiation levels on a specific square meter of Mars. More common cases include polling the population of a country.

1.1 Our Work

In this thesis we study complexity classes for sets decidable in sub-linear time or using a sub-linear number of queries. We use a random access model, as an algorithm running in sub-linear time cannot access each of the bits of its input in the standard Turing-machine model (it wouldn't have time to reach the last bit). A random access model allows an algorithm that runs in time shorter than its input length to access any bit of the input, even if it cannot access them all.

As the thesis's title implies, we focus our study of sublinear algorithms on the study complexity classes of algorithms running in time that is bounded by a polylogarithmic function of the length of their input, or where the number of queries the algorithm may perform to the input is restricted in a similar manner. Bounds on the running time that are at least logarithmic in the size of the input are required because they allow a random access machine to access any bit of its input. Polylogarithmic bounds are technically appealing because polylogarithms, like polynomials, are closed under addition, multiplication and composition. In particular, this allows us to use polylogarithmic-time building blocks in the construction of polylogarithmic algorithms. Essentially, one can think of these algorithms as those that take time polynomial in the binary representation of the length of their input. However, our results can generally be extended to other choices of "small" complexity bounds.

The sub-linear time and query algorithms generally discussed in computer science literature are mostly approximation algorithms [15, 8]. Such algorithms either give an approximation of some numerical value with high probability, or decide whether a combinatorial object is "close" to having a property in some sense (i.e., relate to promise problems [7, 9]).[1] In our work we begin by discussing computations that require an exact result and do not restrict the input to any computation (i.e., refer to standard decision problems rather than to promise problems), and only later do we turn to discuss promise problems.

In Section 2 we describe our model and give simple separations between deterministic, nondeterministic and probabilistic computations, based on "needle in a haystack" style arguments. These arguments hold equally well for polylogarithmic-time and for polylogarithmic-query computations, as they are essentially information theoretic in nature. Our arguments are similar in flavor to the one used by Baker *et. al.* [2] to show that there exists an oracle under which $\mathcal{P}$ does not equal $\mathcal{NP}$. However, while Baker *et. al.* use this argument in a meta-computational context, in our work the arguments relate to concrete and natural computational problems in a concrete and natural computational model.

In Section 3, which holds the more complex technical contribution of this thesis, we consider the intersection of nondeterministic and co-nondeterministic polylogarithmic computations. Unlike the results in Section 2, here the analysis of time-bounded computations is not the same as that used for query-bounded computations. Our main result shows that for query-bounded computations, the intersection of nondeterministic (denoted $\mathcal{NPLQ}$) and co-nondeterministic (denoted co$\mathcal{NPLQ}$) computations can be emulated by a deterministic machine. Our main result in Section 3 is reminiscent of a similar result in communication complexity, and we shall discuss this similarity.

In section 4 we turn our attention to complexity classes of promise problems. We argue that promise problems are an appealing framework for the study polylogarithmic computations, and discuss classes of promise problems that can be solved using polylogarithmic time by deterministic, non-deterministic and randomized machines. We give separations and complete problems for the various complexity classes.

Finally, in section 5 we return to the decision problem setting, and consider two possible definitions of the Polylogarithmic-time Hierarchy. We show that each definition is a natural extension of polylogarithmic non-deterministic complexity classes in the sense that both hierarchies have deterministic polylogarithmic time computations as their 0'th level, and non-deterministic polylogarithmic time computations as their first level. Despite this, we show that according to one definition the polylogarithmic hierarchy doesn't even include

[1]Chazelle has compiled an extensive bibliography of articles discussing such topics [5].

`Parity`, while the other definition yields a hierarchy that is the equivalent of $\mathcal{PH}$.

1.2 Similar Results in the Literature

After the submission of this thesis Ran Raz noted that many of our results have actually been reached in the past. These results come generally from two lines of research. The first is research in relativized complexity, and the second is research in decision-tree complexity.

Relativized complexity research focuses on the computational power of machines that are given access to an "oracle" that decides membership in a certain set. These machines are then given an input and while performing a computation on that input may query the oracle. There are considerable conceptual differences between the work done in relativized complexity and ours. In relativized complexity the motivation for the research is often to better understand what proof techniques may assist in separating various complexity classes (which refer to the input, not to the oracle), whereas our research relates to a realistic computational model. In our model the oracle access is to the input that (naturally) changes from one computation to the next, whereas in relativized complexity the oracle is fixed. Non-the-less, many results similar to ours were reached before, and the technical similarity of the settings allows a transformation of results and proofs from one setting to the other. In particular, Blum and Impagliazzo [3] have shown results equivalent to Theorem 3.2 (that shows the intersection of $\mathcal{NPLQ}$ and co$\mathcal{NPLQ}$ to be deterministically decidable) in their study of generic oracles, and Tardos [18] gives an algorithm almost identical to ours for the same result.

Research in decision-tree complexity is conceptually closer to ours, and similar results are known there too. The concept of *certificate complexity* is closely related to our concepts of positive and negative t-restricted views (defined in Section 3), and Beals *et. al.* [1] show a result regarding certificate complexity that corresponds directly to Theorem 3.2. Lov*á*sz *et. al.* [14] study search problems in the decision tree model, and give a separation result that is similar to Theorem 2.6 (which separates non-deterministic and randomized computations). Buhrman and de Wolf [4] survey many results in this field. The work that comes closest to ours is by Impagliazzo and Naor [12]. In their work they cover much of the work done in Sections 2 and 3 of this thesis. In particular, they define classes of deterministic, nondeterministic and randomized computations that take polylogarithmic time, and discuss the relationship between these classes.

To the best of our knowledge the results in Sections 4 and 5, unlike those in sections 2 and 3, have not been reached before.

2 Deterministic, Nondeterministic and Probabilistic Polylogarithmic Time and Queries

This work examines computations that are performed in time that's significantly shorter than the length of the input. In order to allow this we consider machines that have direct access to individual bits of their input. Formally, we consider a Turing Machine with oracle access to a string x of length n. When the machine writes a binary number i, ranging from 1 to n, on the oracle access tape and invokes the oracle, the oracle returns the i'th bit of x. If anything else is written on the query tape (such as a number greater than n), the oracle returns a special symbol $\bar{b}$. Note that the machine needn't be given the length of x explicitly - it can determine it using queries of exponentially increasing length, and performing a binary search to determine $|x|$, in $O(log(|x|))$ queries. Therefore, when describing an algorithm we may assume, without loss of generality, that it has access to the length of its input. In general, when we refer in this work to "a machine" we are discussing a deterministic (or nondeterministic, depending on context) Turing machine with this type of oracle access to its input. We note this model is consistent with the behavior of real world computers.

Polylogarithmic time computations have technical difficulties that are not generally encountered in other types of computation. In particular, when we state that a machine M is given as input a $3-tuple$ of the form (x, y, z), for instance, we cannot automatically assume that M can tell where the one input ends and the next begins. We can solve this problem in several ways. The approach taken in this text is generally to have y and z have lengths that are an easily computable function of the length of x. Other solutions are possible. We will encounter multiple inputs to a polylogarithmic time machine in Sections 4 and 5.

2.1 Time Complexity

Definition 2.1 *(PolyLogarithmic Time): We denote by $\mathcal{PLT}$ the class of sets decidable by a deterministic machine running in time polylogarithmic in the length of its input.*

An obvious subclass of $\mathcal{PLT}$ consists of sets obtained by exponential padding of sets in $\mathcal{P}$. That is, for every set $S \in \mathcal{P}$ we can define a set $S' \in \mathcal{PLT}$ as follows:

$$S' = \{x \circ y : x \in S, y \in \{0,1\}^{2^{|x|}-|x|}\}$$

Definition 2.2 *(Nondeterministic PolyLogarithmic Time): We denote by $\mathcal{NPL}$ the class of sets decidable by a nondeterministic machine running in time polylogarithmic in the length of its input.*

Like in the case of $\mathcal{PLT}$, an obvious subclass of $\mathcal{NPL}$ consists of sets obtained by exponential padding of sets in $\mathcal{NP}$.

Unlike the case of $\mathcal{P}$ versus $\mathcal{NP}$, one can easily show that $\mathcal{PLT}$ is a strict subset of $\mathcal{NPL}$. We use a "needle in a haystack" argument to show this - we consider the decision problem of whether an input string contains at least a single bit of value 1. In the deterministic case a machine querying a polylogarithmic number of bits may fail to query any bit of value 1 in a string containing such bits, and must therefore return an incorrect result, while in the nondeterministic case the machine needs only guess the location of a bit with value 1. Thus:

Theorem 2.3 $\mathcal{PLT} \neq \mathcal{NPL}$.

Proof: Clearly, $\mathcal{PLT} \subseteq \mathcal{NPL}$. It is left to show that $\mathcal{NPL} \nsubseteq \mathcal{PLT}$. We'll consider a rather simple set - `Or`, defined as the set of all binary strings that include at least one non-zero bit. By showing `Or` $\in \mathcal{NPL}$ and `Or` $\notin \mathcal{PLT}$ we show $\mathcal{PLT} \neq \mathcal{NPL}$.
To see that `Or` $\in \mathcal{NPL}$, define the following simple $\mathcal{NPL}$ machine M. Machine M performs a nondeterministic choice of a query location $1 \leq i \leq |x|$, and returns the bit x_i.
To see that `Or` $\notin \mathcal{PLT}$, we consider any $\mathcal{PLT}$ machine M and its operation on input $x = 0^n$. Being deterministic, M performs some computations and a series $q_1, \ldots, q_m$ of queries (all returning 0, of course). For any M that decides `Or`, it must reject after these queries when obtaining 0 on every answer. Now consider any input string x' that has 0 in the locations $q_1, \ldots, q_m$, and 1's in some of the other locations. M must reject x', and therefore fails to compute `Or` correctly. ■

A similar argument can convince us that the class of complements of sets in $\mathcal{NPL}$, which we denote co$\mathcal{NPL}$, is not equal to $\mathcal{NPL}$.

Theorem 2.4 $\mathcal{NPL} \neq$ co$\mathcal{NPL}$

Proof: It will suffice to show that `Or` $\notin$ co$\mathcal{NPL}$, as we've already proved `Or` $\in \mathcal{NPL}$. Consider any co$\mathcal{NPL}$ machine M and its operation on input $x = 0^n$. Along each nondeterministic path w, M performs some computations and a series $q_1^w, \ldots, q_m^w$ of queries (all returning 0, of course). For any M that decides `Or`, it must reject after these queries when obtaining 0 on every answer along at least one such nondeterministic path w'. Now consider any input string x' that has 0 in the locations $q_1^{w'}, \ldots, q_m^{w'}$, and 1's in some of the other locations. M must reject x' along w', and therefore fails to compute `Or` correctly. $\blacksquare$

Up to this point we discussed deterministic and nondeterministic computations running in time polylogarithmic in the length of their inputs. It is also natural, however, to design probabilistic algorithms when we are limited in our access to the input. We therefore define a natural complexity class for probabilistic computations with running time bound by a polylogarithmic function.

Definition 2.5 *(Bounded-Error Probabilistic PolyLogarithmic-Time): We denote by $\mathcal{BPPL}$ the class of sets decidable by a probabilistic machine running in time polylogarithmic in the length of its input, such that*

1. *If the answer is 'yes' then with probability at least $\frac{2}{3}$ the machine accepts.*
2. *If the answer is 'no' then with probability at most $\frac{1}{3}$ the machine accepts.*

Theorem 2.6 $\mathcal{NPL} \not\subseteq \mathcal{BPPL}$

Proof: It suffices to show that `Or` $\notin \mathcal{BPPL}$, as we've already proved `Or` $\in \mathcal{NPL}$. Consider any $\mathcal{BPPL}$ machine M and its operation on input $x = 0^n$. Any such machine deciding `Or` must return 0 with probability at least $\frac{2}{3}$. Let t denote the polylogarithmic function limiting the running time of M. By a simple counting argument there is at least one location in x queried with probability of at most $\frac{t}{n}$. We denote this bit location l. Now consider the operation of M on the input $x_l = 0^{l-1}10^{n-l}$. In events where M does not query l it must return the same result as in the case of $M(0^n)$, so $M(x_l)$ returns 1 with probability of at most $\frac{1}{3} + \frac{t}{n}$. For any sufficiently large n it follows that M returns 1 with probability less than $\frac{2}{3}$ and thus does not decide `Or`. $\blacksquare$

While the class $\mathcal{BPP}$ is not known to add any power to $\mathcal{P}$, a different class that uses randomization in the polynomial settings, $\mathcal{IP}$, is known to add a great deal of power and is, in fact, equal to $\mathcal{PSPACE}$ [17]. One could thus hope that the polylogarithmic version of interactive proofs would prove to be powerful. To see that this is not the case we first define the relevant complexity class.

Definition 2.7 *(PolyLogarithmic Interactive Proof): We denote by $\mathcal{PL} - \mathcal{IP}$ the class of decision problems for which a "yes" answer can be verified by an* interactive proof *taking polylogarithmic time. Here a $\mathcal{BPPL}$ verifier sends messages back and forth with an all-powerful prover. They can have a polylogarithmic number of rounds of interaction. Given the verifier's algorithm, at the end:*

1. *If the answer is 'yes' the prover must be able to behave in such a way that the verifier accepts with probability at least $\frac{2}{3}$ (over the choice of the verifier's random bits).*

2. *If the answer is 'no' then however the prover behaves the verifier must reject with probability at least $\frac{2}{3}$.*

To show that $\mathcal{PL}-\mathcal{IP}$ is not as powerful as one would hope, we prove that `And` - the set of all strings of the form 1^*, is not in $\mathcal{PL}-\mathcal{IP}$. This result is not surprising given that co$\mathcal{NP}$ is not a subset of $\mathcal{IP}$ relative to a random oracle [6]. We thus see that co$\mathcal{NPL}$ is not a subset of $\mathcal{PL}-\mathcal{IP}$ (Given that `And` is equivalent to Not `Or`, up to the sign of bits, and `Or` is in $\mathcal{NPL}$).

Claim 2.8 co$\mathcal{NPL} \not\subset \mathcal{PL}-\mathcal{IP}$

Proof: Assume that `And` is in $\mathcal{PL}-\mathcal{IP}$. Then there exists a $\mathcal{BPPL}$ verifier V that, interacting with a computationally unbounded prover P accepts (for any n) the string 1^n with probability at least $\frac{2}{3}$, and rejects any other string of length n with probability at least $\frac{2}{3}$ for any prover. Let t denote the polylogarithmic function limiting the running time of V[2]. In the interaction between P and V, where P is proving that 1^n is in `And`, there is at least one location in 1^n queried by V with probability of at most $\frac{t}{n}$ (by a simple counting argument). We denote this bit location l. Assume V is given oracle access to the input $x_l = 1^{l-1}01^{n-l}$, and that a prover P' attempts to prove this string is in `And`. P' emulates the behavior of P proving 1^n is in `And`, and as V only queries the bit l with probability $\frac{t}{n}$. V must accept with probability $\frac{2}{3} - fractn$ which for a sufficiently large n is greater than $\frac{1}{3}$. Thus, `And` isn't in $\mathcal{PL}-\mathcal{IP}$. $\blacksquare$

2.2 Query Complexity

As in the previous section, we will concern ourselves with decision problems where the deciding machine accesses a polylogarithmic number of bits of its input. However, while the polylogarithmic limit to the number of bits accessed in $\mathcal{PLT}$ is implicit in the runtime, we must now make it explicit.

Definition 2.9 *(PolyLogarithmic Queries): We denote by $\mathcal{PLQ}$ the class of sets decidable by a deterministic machine performing a polylogarithmic number of queries to its input.*

[2] t limits the total running time of V in the entire interaction

In a similar manner to the "padding" shown for $\mathcal{PLT}$, an obvious subclass of $\mathcal{PLQ}$ consists of sets obtained by exponential padding of *computable sets*. Using such a padding we can convince ourselves that $\mathcal{PLQ}$ is not a subset of $\mathcal{NPL}$ - simply let the padded set be a computable function not in $\mathcal{NP}$. The proof for such a claim is similar in flavor to the proof of Theorem 3.1.

Definition 2.10 *(Nondeterministic PolyLogarithmic Queries): We denote by $\mathcal{NPLQ}$ the class of sets decidable by a nondeterministic machine performing a polylogarithmic number of queries to its input. All non-deterministic paths must terminate after a number of steps bounded by a computable function of the length of the input.*

Using the same proof as that used for Theorem 2.3 we get the following two theorems (they are not exactly corollaries, as they don't follow from the claim but simply use the same proof):

Theorem 2.11 $\mathcal{NPLQ} \neq \mathcal{PLQ}$

Theorem 2.12 *$\mathcal{NPL}$ is not a subset of $\mathcal{PLQ}$*

In addition, using the same proof as that used for Theorem 2.4, we get the following theorem (where co$\mathcal{NPLQ}$ is the class of complements of sets in $\mathcal{NPLQ}$):

Theorem 2.13 $\mathcal{NPLQ} \neq \mathrm{co}\mathcal{NPLQ}$

3 The Intersection of Nondeterministic and Co-nondeterministic Classes

3.1 Does the Intersection of $\mathcal{NPL}$ and co$\mathcal{NPL}$ equal $\mathcal{PLT}$?

As the reader may have noted, the relation between the polylogarithmic time complexity classes is similar to the *suspected* relation between the polynomial-time complexity classes: We suspect that $\mathcal{NP} \neq \mathcal{P}$, and have that $\mathcal{NPL} \neq \mathcal{PLT}$. We suspect that $\mathcal{NP} \neq \mathrm{co}\mathcal{NP}$, and have that $\mathcal{NPL} \neq \mathrm{co}\mathcal{NPL}$. A similar structure holds for the polylogarithmic query complexity classes inspected so far. While the proofs of the separation of the aforementioned complexity classes are simple and rely on no specific computational assumptions, we have failed to find a similar proof for the claim that $\mathcal{NPL} \cap \mathrm{co}\mathcal{NPL} \neq \mathcal{PLT}$, which is analogous to the conjecture that $\mathcal{NP} \cap \mathrm{co}\mathcal{NP} \neq \mathcal{P}$. We can, however, prove the following claim:

Theorem 3.1 *If $\mathcal{NP} \cap \text{co}\mathcal{NP} \neq \mathcal{P}$ then $\mathcal{NPL} \cap \text{co}\mathcal{NPL} \neq \mathcal{PLT}$*

Proof: If $\mathcal{NP} \cap \text{co}\mathcal{NP} \neq \mathcal{P}$ there exists a set $S \in \mathcal{NP} \cap \text{co}\mathcal{NP}$ such that $S \notin \mathcal{P}$. Consider the set $S' = \{x \circ y : x \in S, y \in \{0,1\}^{2^{|x|}-|x|}\}$ which is a simple padding of S. Clearly, S' is in $\mathcal{NPL}$ and in co$\mathcal{NPL}$. Assume towards contradiction that $S' \in \mathcal{PLT}$, and denote the $\mathcal{PLT}$ machine deciding it by M'. We define a polynomial-time machine M that decides S, thus reaching a contradiction. When M is given the input x it emulates $M'(x \circ 0^{2^{|x|}-|x|})$ in the straightforward way (returning bits of x for queries to bits of locations $1 \ldots |x|$ and 0 otherwise), running in time polylogarithmic in $2^{|x|}$ and thus polynomial in $|x|$. As M is a deterministic polynomial time machine deciding S (which is not in $\mathcal{P}$), a machine M' cannot exist and S' is not in $\mathcal{PLT}$. ■

If The intersection of $\mathcal{NP}$ and co$\mathcal{NP}$ does equal $\mathcal{P}$, does this mean that the intersection of $\mathcal{NPL}$ and co$\mathcal{NPL}$ equals $\mathcal{PLT}$? This remains an open question. It would be nice to see a $\mathcal{PLT}$ algorithm for $\mathcal{NPL} \cap \text{co}\mathcal{NPL}$ using an $\mathcal{NP} \cap \text{co}\mathcal{NP}$ oracle.

3.2 The Intersection of $\mathcal{NPLQ}$ and co$\mathcal{NPLQ}$ Equals $\mathcal{PLQ}$

As the title of this subsection suggests, the relation between $\mathcal{NPLQ} \cap \text{co}\mathcal{NPLQ}$ and $\mathcal{PLQ}$ is different than the one suspected between $\mathcal{NP} \cap \text{co}\mathcal{NP}$ and $\mathcal{P}$. The rest of this section is dedicated to proving the following result:

Theorem 3.2 $\mathcal{NPLQ} \cap \text{co}\mathcal{NPLQ} = \mathcal{PLQ}$

In order to prove this theorem we must first define and develop several concepts.

Definition 3.3 *(n-View): An n-view is a partial function from $[n]$ to $\{0,1\}$. We denote the subset of $[n]$ on which a view v is defined as dom(v).*

When an algorithm performs queries to some locations $l_1 \ldots l_m$ on an input string x of length n, we can describe the information that the algorithm has about x using an n-view, defined only on the values $l_1 \ldots l_m$, so that $v(l_1) = x_{l_1} \ldots v(l_m) = x_{l_m}$. If $l_1 \ldots l_m$ are not all the values in $[n]$, there are several different strings that have the same values in $l_1 \ldots l_m$. When a string and a view have the property that the view can be obtained by corresponding queries to the string, we say the view and the string are consistent. When two views may have been defined by queries to the same string we say the two views are consistent. The following definitions formalize these notions.

Definition 3.4 *(View and string consistency): Denoting the i'th bit of a string x as x_i, we say a string x is consistent with a view v if and only if for every $i \in dom(v)$, it holds that $v(i) = x_i$.*

Definition 3.5 *(View consistency): The views v_1 and v_2 are said to be consistent if and only if there exists a string x such that x is consistent with v_1 and x is consistent with v_2.*

It is clear that one need not search all the strings x of length n to tell if two n-views are consistent. The following two facts capture two simple ways to determine consistency.

Fact 3.6 *(View consistency): Let v_1 and v_2 be n-views, then v_1 and v_2 are consistent if and only if for all $i \in dom(v_1) \cap dom(v_2)$ it holds that $v_1(i) = v_2(i)$.*

To see this fact is true it suffices to consider any string of length n where $x_i = v_1(i)$ where $i \in dom(v_1)$, and $x_i = v_2(i)$ where $i \in dom(v_2)$. Such a string exists if and only if for all $i \in dom(v_1) \cap dom(v_2)$ it holds that $v_1(i) = v_2(i)$.

Fact 3.7 *(View and sub-view consistency): Let v_1 be a view, v_2 be a view such that v_1 extends v_2(i.e., $dom(v_2) \subseteq dom(v_1)$, and for all $i \in dom(v_2)$ it holds that $v_2(i) = v_1(i)$). Then if x is a string consistent with v_1, x is consistent with v_2, and it holds that v_1 and v_2 are consistent.*

Obviously any set $S \in \{0,1\}^n$ can be defined by a set of n-views S', where a string is in S if and only if it is consistent with some view in S'. This is trivially true as we can define the set S' to be the views defined on all n bits, so each view is consistent with exactly one member of S. Sets based on polylogarithmic query computations, however, can be defined using views in a more compact manner. We will use this fact, but we will first define the notion of a computable t-restricted collection of views, and we will prove the connection between such views and $\mathcal{NPLQ}$ sets.

Definition 3.8 *(Computable t-restricted views): A collection of views $V = \{V_n\}$, where each $v \in V_n$ is an n-view, is called computable if there exists a Turing machine that on input n outputs the list of all views in V_n. Such a collection is called t-restricted (for some function t) if each $v \in V_n$ is defined on at most $t(n)$ points, i.e., for all $v \in V_n$ it holds that $|dom(v)| < t(n)$.*

Lemma 3.9 *If a set S is in $\mathcal{NPLQ}$ then there exists a polylogarithmic function t and a computable t-restricted collection of views, $V = \{V_n\}$, such that $x \in S$ if and only if x is consistent with some view in $V_{|x|}$.*

Proof: Let M be an $\mathcal{NPLQ}$ machine deciding S. We describe a simple deterministic algorithm that, given n, outputs a t-restricted collection of views V_n such that $x \in S$ if and only if x is consistent with some view in $V_{|x|}$. On input n the algorithm works as follows:

Define an (initially) empty collection of views V. For every string x of length n, and for every nondeterministic computation path w taken by $M(x)$, perform the following. If w terminates unsuccessfully, disregard it. If w terminates successfully after querying the bits at locations $l_1, \dots, l_m$ define the view v to be the partial function from $[n]$ to $\{0, 1\}$ defined at locations $l_1 ... l_m$ and taking the values $v(l_i) = x_{li}$. If $v \notin V$ add v to V. Once all x's and all w's were examined, output the final V.

The algorithm terminates, because by Definition 2.10 every $\mathcal{NPLQ}$ machine, and in particular M, terminates on all nondeterministic paths. Setting t to be the polylogarithmic function bounding the number of queries performed by M, the outputted views are all t-restricted.

It remains to show that $x \in S$ if and only if there exists a view $v \in V$ such that x is consistent with be v.

Suppose, first, that $x \in S$. Therefore, there exists a nondeterministic computation path w taken by $M(x)$ that terminates successfully. The view v is thus added to V. Clearly, x is consistent with v (as v is defined by queries on x).

Suppose, on the other hand, that x is consistent with some view $v \in V$. It follows that M queried along some nondeterministic path w all locations $l \in dom(v)$ of some $x' \in S$, and terminated successfully. As both x and x' are consistent with v, we have that $x_l = v(l) = x'_l$ for all $l \in dom(v)$. Clearly, $M(x)$ will only query, along path w, locations in $dom(v)$ and will thus accept x along this path. As M decides S, it follows that $x \in S$. ■

As an easy corollary, we can see that co$\mathcal{NPLQ}$ sets can also be characterized by a polylogarithmic-restricted collection of views, where strings consistent with these views are *not* in the set.

Corollary 3.10 *If a set S is in co$\mathcal{NPLQ}$ there exists a polylogarithmic function t and a computable t-restricted collection of views, $V = \{V_n\}$, such that $x \notin S$ if and only if V_n contains a view consistent with x.*

We saw above that consistency with a certain view v can ensure that some input x is a member of some set S. Likewise, consistency with some view v' can ensure that some input string x' is not a member of some set S'. We call such views positive views and negative views.

Definition 3.11 *(Positive view): An n-view v is said to be a positive view for a set S if any string $x \in \{0,1\}^n$ that is consistent with v is in S.*

Definition 3.12 *(Negative view): An n-view v is said to be a negative view for a set S if any string $x \in \{0,1\}^n$ that is consistent with v is not in S.*

Note that we can now see Lemma 3.9 as stating the fact that for every set $S \in \mathcal{NPLQ}$ there exists a machine M that on input n outputs a set V of positive views for S such that every $x \in S$ is consistent with some view $v \in V$. In a similar manner, Corollary 3.10 can be seen as stating the fact that for every set $S \in \text{co}\mathcal{NPLQ}$ there exists a machine M that on input n outputs a set V of negative views for S such that every $x \notin S$ is consistent with some view $v \in V$. We now turn to a claim that helps us understand the relation between positive and negative views for a set.

Claim 3.13 *Let S be a set, v_1 be a positive view for S, and v_2 be a negative view for S, then v_1 and v_2 are not consistent.*

Proof: Assume towards contradiction that v_1 and v_2 are consistent. This means there exists a string x they are both consistent with (by Definition 3.5). But then, by Definitions 3.11 and 3.12, we get $x \in S$ and $x \notin S$ respectively, which is a blunt contradiction. ■

Fact 3.14 *Let u be an n-view, and let v_1 and v_2 be n-views that are each consistent with u. If v_1 is not consistent with v_2, then there exists a location i in $dom(v_1) \cap dom(v_2)$ and not in $dom(u)$.*

Proof: Assume towards contradiction that $dom(v_1) \cap dom(v_2) \subseteq dom(u)$. As both v_1 and v_2 are consistent with u, for each i in $dom(v_1) \cap dom(v_2)$, the equalities $v_1(i) = u(i)$ and $v_2(i) = u(i)$ hold, and thus for every i in $dom(v_1) \cap dom(v_2)$ it holds that $v_1(i) = v_2(i)$. But then v_1 and v_2 are consistent (according to Fact 3.6), leading to a contradiction. ■

By combining Claim 3.13 and Fact 3.14, we get a key tool for the proof.

Proposition 3.15 *Let S be a set, v_1 be a positive n-view for S, and v_0 be a negative n-view for S. If v_1 is consistent with some n-view u and v_0 is also consistent with u, then there exists a location i in $dom(v_1) \cap dom(v_0)$ and not in $dom(u)$.*

We now turn to showing a $\mathcal{PLQ}$ algorithm that decides sets in $\mathcal{NPLQ} \cap \text{co}\mathcal{NPLQ}$. Such an algorithm will establish Theorem 3.2.

Let the set S be in $\mathcal{NPLQ} \cap \text{co}\mathcal{NPLQ}$. By Lemma 3.9 there exists a machine M_1 that on input n outputs a set of positive views for S, which we denote V_1, such that every x of length n in S is consistent with some view in V_1. Likewise, by Corollary 3.10 there exists a machine M_0 that on input n outputs a set of negative views for S, which we denote V_0, such that every x of length n that is not in S is consistent with some view in V_0. Note that the views outputted by M_0 and the views outputted by M_1 are restricted by polylogarithmic functions (which we denote t_0 and t_1 respectively). We construct a $\mathcal{PLQ}$ algorithm for deciding $x \in S$ that works as follows:
The algorithm begins by determining the length of x, using binary search, and initializes two sets of views: V_1 that holds the positive $|x|$-views for S computed using M_1, and V_0 that holds the negative $|x|$-views for S computed using M_0. The algorithm also initializes an (initially) empty view (that is, a view that is defined nowhere), denoting it v_x. Next, the algorithm repeats the following steps until the membership of x in S is decided:

1. If $|V_1| = 0$ then return 0 (indicating $x \notin S$).
2. If v_x extends a view $v \in V_0$ (i.e., v and v_x are consistent and $dom(v) \subseteq dom(v_x)$) then return 0.[3]
3. If $|V_0| = 0$ return 1 (indicating $x \in S$).
4. If v_x extends a view $v \in V_1$ then return 1.
5. Select an arbitrary view $v \in V_0$ and query all locations $i \in dom(v)$. For each such i update v_x setting $v_x(i) = x_i$.[4]
6. For all views $v \in V_0$, if v is not consistent with v_x then remove v from V_0.
7. For all views $v \in V_1$, if v is not consistent with v_x then remove v from V_1.

[3]Steps 2 and 4 are not really required, but help keep the analysis simple

[4]An alternative version of this algorithm would only query $i \notin dom(v_x)$, and would alternate picking $v \in V_0$ and $v \in V_1$. Such an algorithm would improve our query complexity by a factor of 2.

Let us first establish the algorithm's correctness. If the algorithm returns 0 in Step 1 then there are no positive views consistent with x in V_1 (whereas only non consistent views were dropped in Step 7), and thus by Lemma 3.9 $x \notin S$. If the algorithm returns 0 in Step 2 then there is a view $v \in V_0$ such that x is consistent with v. This is true because for all $l \in dom(v)$ it holds that $v(l) = x_l$ (see Definition 3.4). Thus, as v is a negative view, $x \notin S$. A similar analysis holds for the two possible cases where the algorithm returns 1.
We now calculate the algorithm's query complexity, showing it to be in $\mathcal{PLQ}$. Let the positive views for S be restricted by a polylogarithmic function t_1, and let the negative views for S be restricted by a polylogarithmic function t_0 (such bounds exist by Lemma 3.9 and Corollary 3.10 respectively). We denote by V_0^i the views in V_0 at the end of the i'th iteration, by V_1^i the views in V_1 at the end of the i'th iteration, and by v_x^i the view v_x at the end of the i'th iteration. We will show below that for every positive view $v \in V_1^i$ it holds that $|dom(v) - dom(v_x^i)| \leq (t_1(|x|) - i)$. When $|dom(v) - dom(v_x^i)| = 0$ for all $v \in V_1^i$, either $|V_1^i| = 0$ and we return 0 in the next round, or for all $v \in V_1^i$, it holds that v_x^i is an extension of v and we return 1. In the initialization phase we perform at most $O(log(|x|)$ queries, and in each iteration we perform at most t_0 queries, so we get a total of $t_0 t_1 + O(log(|x|)$ queries. As t_0 and t_1 are polylogarithmic functions we have a polylogarithmic query complexity.
It remains to show that for every view $v \in V_1^i$ it holds that $|dom(v) - dom(v_x^i)| \leq (t_1(|x|) - i)$. We will do this by induction.
Base case: $i = 0$. As the views are t_1-restricted, the claim holds.
Induction: At the end of the i'th iteration, for every view $v_1 \in V_1^i$ it holds that $|dom(v_1) - dom(v_x^i)| \leq (t_1(|x|) - i)$. During the $i+$'th iteration, at Step 5 of the algorithm we select a negative view v_0 that is consistent with v_x^i. By Proposition 3.15 v_0 has at least one location l in it's domain that intersects with the domain of any $v_1 \in V_1^i$ and isn't in $dom(v_x^i)$. Location l is added to $dom(v_x^i)$ and thus at the end of the $i + 1$'th iteration, for every $v_1 \in V_1^{i+1}$ it holds that $|dom(v_1) - dom(v_x^{i+1})| \leq (t - i - 1)$. ■

Note that our result does not depend on the polylogarithmic nature of t_1 and t_0. The proof holds just as well for any values of of t_1 and t_0, giving us a bound of of $log(|x|) + t_1 t_0$. In addition, a non-uniform version of $\mathcal{PLQ}$ could decide sets in the intersection of the non-uniform versions of $\mathcal{NPLQ}$ and co$\mathcal{NPLQ}$.

Theorem 3.2 is similar to a result in communication complexity. The setting for two-party communication complexity has two communicating parties, Alice and Bob, who are both computationally unbounded [13]. Alice is given access to an input $x \in X$ and Bob is given access to an input $y \in Y$. Together they must compute $f(x, y)$ by sending bits to each other according to a well defined protocol. The deterministic communication complexity of a function f is the number of bits that Alice and Bob transfer to each other in the process of computing f. An easy way to think of nondeterminism in communication complexity is in terms of verifiable proofs. If for every $(x, y) \in S$ there is a proof of length N_1 that can be verified by both Alice and Bob for the membership of (x, y) in S, we consider N_1 to be the nondeterministic computational complexity of deciding S. Likewise, if there is a proof of non-membership of length N_0, we consider N_0 to be the co-nondeterministic computational complexity of deciding S. In such a case there is a deterministic protocol for deciding membership in S that requires the communication of $O(N_1 N_0)$ bits. This result, as well as the algorithm for performing such a protocol, relies heavily on the computationally unbounded nature of the communicating parties, and one cannot but be reminded of it when seeing the result $\mathcal{NPLQ} \cap \mathrm{co}\mathcal{NPLQ} = \mathcal{PLQ}$, and the quantitative version of it presented in the proof.

4 Promise Problems and Complete Problems

Up to this point our discussion of polylogarithmic time computations has focused on decision problems. However, we claim that a more natural setting for sublinear time algorithms is that of promise problems (as defined below). In fact, the field of *property testing* [8, 15], that deals exclusively with sublinear algorithms, generally does so in a promise-problem setting. Furthermore, there are many algorithms that are actually sublinear algorithms that solve promise problems, but are not generally conceptualized as such. Every time we use a predefined data structure and perform a query running in sublinear time on that structure, we're essentially given a promise - that our data structure is well-shaped, and we perform a sublinear time computation. A simple example of this is performing binary search on a sorted list of tokens, to decide whether a token appears in the list. Given the promise that we're dealing with a sorted list, deciding whether a token is in it can be done by a $\mathcal{PLT}$ machine. In contrast, when considering the analogous search problem without this promise, the problem is in $\mathcal{NPL}$ and not in $\mathcal{PLT}$ (see the special case of `Or` discussed in claim 2.3).

Definition 4.1 *(Promise problems ([7], see also [9])): A promise problem* Π *is a pair of non-intersecting sets, denoted* (Π_{YES}, Π_{NO})*; that is,* $\Pi_{YES} \subseteq \{0,1\}^*$ *and* $\Pi_{YES} \cap \Pi_{NO} = \phi$*. The set* $\Pi_{YES} \cup \Pi_{NO}$ *is called the* promise*.*

To summarize, promise problems provide a suitable framework for discussing polylogarithmic complexity classes (because they are the framework in which such computations are most often used). When we investigate promise classes rather than standard decision classes the full power of the different polylogarithmic-time computational models comes to fore. While in Section 2 we were incapable of separating some of the complexity classes we defined, their promise problem extensions are easily separable.

4.1 Complexity Classes for Promise Problems

It is natural to extend complexity classes to include promise problems that can be solved using specific computational resources. Thus, if $\mathcal{PLT}$ is the class of sets decidable by machines that run in time bounded by a polylogarithmic function in the length of their input, *promise-*$\mathcal{PLT}$ is the class of promise problems decidable using the same resources. Obviously, $\mathcal{PLT}$ is a subset of *promise-*$\mathcal{PLT}$, where the sets in $\mathcal{PLT}$ are those sets that have a trivial promise of $\{0,1\}^*$. Likewise, we can define *promise-*$\mathcal{NPL}$ as follows.

Definition 4.2 *(promise-$\mathcal{NPL}$): promise-$\mathcal{NPL}$ is the class of promise problems* $\Pi = \{\Pi_{YES}, \Pi_{NO}\}$ *that are decidable by a nondeterministic Turing Machine M running in polylogarithmic time, so that:*

1. *If $x \in Pi_{YES}$ then $M(x)$ returns 1 along at least one nondeterministic path.*
2. *If $x \in Pi_{NO}$ then $M(x)$ returns 0 along all nondeterministic paths.*

In the reminder of this text we will not repeat the definitions of complexity classes for each promise problem version, but will simply use the prefix "*promise-*" to denote the promise versions of our complexity classes.

In previous sections we showed that $\mathcal{NPL}$ isn't a subset of $\mathcal{BPPL}$ but failed to separate $\mathcal{BPPL}$ from $\mathcal{NPL}$, and more importantly, failed to separate $\mathcal{BPPL}$ from $\mathcal{PLT}$. The separation of the promise versions of these classes, however, is easy, and we can in fact show that *promise-*$\mathcal{BPPL} \nsubseteq$ *promise-*$\mathcal{NPL}$ (and thus clearly *promise-*$\mathcal{BPPL} \nsubseteq$ *promise-*$\mathcal{PLT}$).

Theorem 4.3 *promise-*$\mathcal{BPPL} \not\subset$ *promise-*$\mathcal{NPL}$

While the following proof tells us that *promise-*$\mathcal{BPPL}$ isn't a subset of *promise-*$\mathcal{NPL}$, note that a similar proof holds for the computationally unbounded case.

Proof: We will consider the following promise problem version of majority, which we call *strong majority.* Π_{Yes} is the set of strings that have at least a $\frac{2}{3}$ of their bits with the value 1, and Π_{No} is the set of strings that have at most a $\frac{1}{3}$ of their bits with the value 1. This promise problem is decidable by a $\mathcal{BPPL}$ machine that chooses a location in its input at random and returns the value of the bit in that location. However, it is not decidable by any $\mathcal{NPL}$ machine. Assume towards a contradiction that such an $\mathcal{NPL}$ machine M exists, and consider M's operation on the string $x = 1^n$. M must obviously accept x along at least one nondeterministic computational path w. Along this path M would inspect at most a polylogarithmic number of bits in some locations $l_1, \ldots, l_m$, all having the value 1. Now consider the string with 1's in locations $l_1, ..., l_m$, and 0 in all other locations. M must clearly accept such a string, yet it is in Π_{No}. ■

If two decision problem complexity classes are separate, their promise versions are separate too. The fact that two decision problem complexity classes are equal, however, does not mean that their promise versions are equal. A concrete example follows. Recall that we proved that $\mathcal{PLQ}$ equals $\mathcal{NPLQ} \cap \text{co}\mathcal{NPLQ}$ (see theorem 3.2). We now prove that *promise*-$\mathcal{PLQ}$ does not equal the intersection of *promise*-$\mathcal{NPLQ}$ and *promise*-co$\mathcal{NPLQ}$.

To do this we introduce what we call "the missing cat problem". Assume you lost your cat, and wish to decide whether it is in the southern hemisphere or the northern hemisphere of the earth. The claim that the cat is in the northern hemisphere can easily be validated if someone shows you the cat in Quebec, and can likewise be easily refuted if someone shows you the cat in New-Zealand. However, no reasonable amount of careful deliberation and searching will lead you to the location of the cat (assuming that the cat is, naturally, as likely to be in Northern America as it is to be in Oceania). In a similar manner, assume you are given a string x that includes only a single bit of value 1. Deciding whether this bit is in the first half of the string is not possible using $\mathcal{PLQ}$ resources. However, the claim that the bit is in the first half (respectively, the second half) of the string can easily be validated by being given a pointer to the bit.

Theorem 4.4 *promise*-$\mathcal{PLQ} \neq$ *promise*-$\mathcal{NPLQ} \cap$ *promise*-co$\mathcal{NPLQ}$

Proof: We will show the inequality using "the missing cat problem". Our promise set is 0^*10^* and members of Π_{Yes} are strings where the 1 (a.k.a. "the missing cat") is in the first $\lfloor \frac{|x|}{2} \rfloor$ bits. The problem is in *promise*-$\mathcal{NPLQ}$ (choose $i \in [1, \lfloor \frac{|x|}{2} \rfloor]$ nondeterministically, and return the i'th bit), and is in *promise*-co$\mathcal{NPLQ}$ (choose $i \in [\lfloor \frac{|x|}{2} \rfloor + 1, |x|]$ nondeterministically, and return the complement of the i'th bit). However, the problem is not in *promise*-$\mathcal{PLQ}$ for the same needle-in-a-haystack argument that shows `Or` is not in $\mathcal{PLQ}$: Assume towards contradiction that a certain $\mathcal{PLQ}$ machine M decides the problem. Given a string with the 1 value in the first half of the input, M must query certain bits in locations $l_1, ...l_m$. All these bits may be 0's, and M must return 1. Now given a different string, where the bits in locations $l_1, ...l_m$ are 0's but the 1 bit is in the second half of the string, M must again return 1, thus failing to decide Π. ■

Note that the argument above does not use the computationally unbounded nature of the machines. Thus, the same promise problem allows us to state the following theorem:

Theorem 4.5 *promise*-$\mathcal{PLT} \neq$ *promise*-$\mathcal{NPL} \cap$ *promise*-co$\mathcal{NPL}$

While problems involving missing cats do not generally motivate any research in computational complexity[5], "the missing cat problem" was not introduced merely to amuse the reader. In many cases actions taken in the macroscopic world are aimed at deciding a question, e.g., "Is Mr. X alive or dead"? Without the assumption that there is only one copy of Mr. X, this question could not be answered unambiguously. Indeed, by (analogy to) Theorem 3.2, for any question that a court of law could decide using evidence presented to it, it would not require any evidence from the concrete world that it could not itself collect, after careful deliberation. In general, we wish to stress that computational problems involving the macroscopic world are often sublinear problems with a promise.

4.2 $\mathcal{PLT}$ Reductions and Complete Problems

A complete problem may offer a way to better understand a complexity class. By seeing a canonical type of problem for a class, we gain insight into what computational resources are required to solve problems in that class. Of course, the type of reduction under which a problem is complete is essential, and should conceptually cover basic computational resources, where the meaning of "basic" may vary according to the settings. When we say `SAT` is $\mathcal{NP}$-complete under $\mathcal{P}$ reductions, we're considering polynomial time computations to be a basic resource, and stating that the "added ability" to solve `SAT` allows us to solve any problem in $\mathcal{NP}$. In the context of polylogarithmic-time complexity classes, the basic resources available are $\mathcal{PLT}$ computations.

$\mathcal{PLT}$ reductions have a technical difficulty that is not shared by the standard types of reductions. A $\mathcal{PLT}$ algorithm doesn't have sufficient time to read all of its input and

[5]To the best of our knowledge. The case is different with physics [16].

transform it, or even to copy it, and thus cannot present a reduced instance in an explicit manner. Thus, we use in our definition of $\mathcal{PLT}$ reductions an implicit representation of the reduced instance given the original input: The reduction is a $\mathcal{PLT}$ machine that, given oracle access to the original input and an index of a bit in the reduced instance, returns the value of that bit.

Following the notational convention used in this work of denoting the i'th bit of the string x as x_i, we denote the i'th bit of the reduced instance of a string x created by the use of a $\mathcal{PLT}$ machine M as M_i^x. Thus, the reduced version of a string x will be of the form $M_1^x M_2^x ... M_T^x$. In an attempt to retain a standard mode of operation for our $\mathcal{PLT}$ machines, we assume the machine that is used for the reduction is given oracle access to the number of the required bit, concatenated to the original input, and returns the value of that bit. Therefore M_i^x is the output of $M^{i \circ x}$[6].

Definition 4.6 *($\mathcal{PLT}$ reductions): A promise problem Π is said to be $\mathcal{PLT}$-reducible to a promise problem Π' if there exist a $\mathcal{PLT}$ machine M and a polylogarithmic function t, such that the following hold:*

1. *If x is a member of Π_{Yes} then $M_1^x ... M_{2^{t(|x|)}}^x$ is a member of Π'_{Yes}*

2. *If x is a member of Π_{No} then $M_1^x ... M_{2^{t(|x|)}}^x$ is a member of Π'_{No}*

We claim that $\mathcal{PLT}$ reductions as defined here capture the correct notion of a reduction in the polylogarithmic settings.

Proposition 4.7 *Any promise problem S that is $\mathcal{PLT}$-reducible to a promise-$\mathcal{PLT}$ promise problem T is in promise-$\mathcal{PLT}$.*

Proof: Let M_T be the $\mathcal{PLT}$ machine deciding T, let M be the $\mathcal{PLT}$ machine used in the reduction, and let r be the polylogarithmic function from the reduction. We denote by m and t the polylogarithmic running times of M and M_T respectfully. We construct the $\mathcal{PLT}$ machine M_S that decides S as follows: M_S emulates M_T on the string $M_1^x ... M_{2^{r(|x|)}}^x$ in the straight-forward way (using M to compute the results of any query to bits of the string). That is, if M_T queries the i'th bit of the reduced instance, it is given M_i^x.

[6]A critical reader may object to this model, claiming that we have no way of appending bits to an oracle-access string. Even if we cannot do this the model is conceptually robust, as we can easily emulate $M^{i \circ x}$ given access to M and i, and oracle access to x

The running time of M_S is a polylogarithmic function of the length of the reduced instance. As this length is $2^{r(|x|)}$, and r is a polylogarithmic function in the length of x, the total running time is also polylogarithmic in the length of x. $\blacksquare$

While $\mathcal{PLT}$ reductions may initially seem very different from their polynomial-time counterparts, we note that many polynomial-time reductions in the literature are actually $\mathcal{PLT}$ reductions. In particular, the classic reduction of any $\mathcal{NP}$ problem to $k-$ `SAT` (for a fixed k) is a $\mathcal{PLT}$ reduction (see, for instance, the version of this reduction presented by Garey and Johnson [10]). We will use this fact later, in Section 5.

Using $\mathcal{PLT}$ reductions we can now give complete problems that capture the essence of the different polylogarithmic time complexity classes. We begin by showing that `Or` is complete for *promise*-$\mathcal{NPL}$ problems. In this reduction, if M' is a machine that nondeterministically decides a promise problem, we reduce the input x to a string that is a kind of "truth table" for the different nondeterministic choices that can be made by $M'(x)$. Obviously, if x is a YES instance of the promise problem decided by M', then such a truth table includes at least a single 1, and thus the reduced instance is in `Or`.

Theorem 4.8 `Or` *is complete for promise-*$\mathcal{NPL}$

Proof: We already know that `Or` is in $\mathcal{NPL}$ (and thus in *promise*-$\mathcal{NPL}$), so it remains to show that every promise problem Π in *promise*-$\mathcal{NPL}$ is reducible to `Or`. Let M' be the nondeterministic machine deciding Π, and let t be a polylogarithmic function bounding its running time. We construct the following $\mathcal{PLT}$ reduction, denoted M. M reduces an input x to a string of length $2^{t(x)}$, where the w'th bit of the reduced instance is the emulation of $M'(x)$ where the nondeterministic choices of M' are made according to w. As M' decides Π nondeterministically, if x is a member of Π_{Yes} then M' accepts along at least one nondeterministic path w. M^x_w will thus have the value 1, and $M^x_1 ... M^x_{2^{t(|x|)}}$ is thus a member of `Or`. The converse also holds - if x is a member of Π_{No} then M' does not accept it along any nondeterministic path, and thus $M^x_1 ... M^x_{2^{t(|x|)}} = 0^{2^{t(|x|)}}$ and is not a member of `Or`. $\blacksquare$

Note that in the previous case, while `Or` is complete for a set of promise problems, `Or` itself is a "regular" set, only having the trivial promise, that all input is in $\{0,1\}^*$. In a similar manner to Theorem 4.8, we can show that `And` (the set 1^*) is complete for *promise*-co$\mathcal{NPL}$.

Theorem 4.9 `And` *is complete for promise-*co$\mathcal{NPL}$

In the following theorem we show that *strong majority*, a promise problem with a non-trivial promise, is complete for *promise*-$\mathcal{BPPL}$. The problem (as well as the short proof) are analogous to a promise problem that is complete for $\mathcal{BPP}$. As in the case of a complete problem for *promise*-$\mathcal{NPL}$ our reduction of an input string x is also to a type of "truth table" of the deciding machine $M'(x)$. Here the truth table is not of nondeterministic choices, however, but of coin tosses.

Theorem 4.10 *promise-$\mathcal{BPPL}$ has a complete problem*

Proof: Recall that the promise problem strong majority, which was used previously in proving Theorem 4.3 is defined as follows: Π_{Yes} is the set of strings that have at least $\frac{2}{3}$ of their bits with value 1, and Π_{No} is the set of strings that have at most $\frac{1}{3}$ of their bits with value 1. We've already shown that strong majority is in *promise*-$\mathcal{BPPL}$, so it remains to show that any problem in *promise*-$\mathcal{BPPL}$ can be reduced to it. Let M' be the $\mathcal{BPPL}$ machine deciding a promise problem Π', and let t be the polylogarithmic function that limits the running time of M'. We construct a $\mathcal{PLT}$ reduction denoted M that transforms an input x to a string of length $2^{t(x)}$. The i'th bit of the reduced string is the emulation of $M'(x)$ where the coin tosses of M' are determined according to i. If x is in Π'_{Yes} then M' accepts with probability of at least $\frac{2}{3}$ over its coin tosses, and thus $M^x_1...M^x_{2^{t(|x|)}}$ has at least a $\frac{2}{3}$ fractions of its bits as 1, and $M^x_1...M^x_{2^{t(|x|)}}$ is in Π_{Yes}. Similarly, if x is in Π'_{No} then M' rejects with probability of at least $\frac{2}{3}$ over its coin tosses, and thus $M^x_1...M^x_{2^{t(|x|)}}$ has at most a $\frac{1}{3}$ of its bits set to 1 and is in Π_{No}. ■

Finally, after showing complete problems for *promise*-$\mathcal{NPL}$, *promise*-co$\mathcal{NPL}$ and *promise*-$\mathcal{BPPL}$, we show a complete promise problem for *promise*-$\mathcal{NPL} \cap$ *promise*-co$\mathcal{NPL}$. In the previous complete problems, we used a type of reduction of the input x to a "truth table". In the case of the complete problem for *promise*-$\mathcal{NPL}$ we had a specific type of machine (a nondeterministic machine running in polylogarithmic time) that was used in the definition of the class, and we showed that the "truth table" for any such machine on an accepting input always has a specific form. A similar method was used for *promise*-$\mathcal{BPPL}$, where the class is defined by probabilistic machines. However, in the case of *promise*-$\mathcal{NPL} \cap$ *promise*-co$\mathcal{NPL}$ our complexity class was not defined by a single type of machine, and thus this technique doesn't work directly. Instead, we adapt it by simply using two "truth tables" as our reduced instance. If M_1 is the nondeterministic machine deciding a promise problem, and M_0 the co-nondeterministic machine deciding it, we use as a reduced instance the concatenated "truth tables" for the nondeterministic choices made by $M_1(x)$ and those made by $M_0(x)$ to get a complete problem.

Theorem 4.11 *promise-$\mathcal{NPL} \cap$ promise-co$\mathcal{NPL}$ has a complete problem*

Proof: We define the promise problem Π as follows: Π_{Yes} is the set of strings that have at least one bit set to 1 in the first half of the bits, and no bit set to 1 in the second half, while Π_{No} is the set of strings that have at least one bit set to 1 in the second half of the bits, and no bit set to 1 in the first half[7]. Let Π' be a promise problem in *promise*-$\mathcal{NPL} \cap$ *promise*-co$\mathcal{NPL}$, let M_1 and M_0 be the nondeterministic and co-nondeterministic machines deciding Π' respectively, and let t be a polylogarithmic function bounding both the running time of M_1 and of M_0. We construct the following $\mathcal{PLT}$ reduction, which we denote M. M reduces an input x to a string of length $2^{t(x)+1}$. The w'th bit of the first half of the reduced instance is given by emulating M_1^x along the nondeterministic path w. In a similar manner, the w'th bit of the second half of the reduced instance is given by emulating M_0^x along the nondeterministic path w, and taking the complement of the result.

Obviously, if x is a member of Π'_{Yes} then $M_0(x)$ returns 1 along every computation path and thus all the bits in the second half of the reduced instance $(M_1^x ... M_{2^{t(|x|)+1}}^x)$ will be set to 0. For such an x at least one computation path of $M_1(x)$ will return 1 and thus the first half of the reduced instance will include at least a single 1. A similar analysis holds for the case where x is a member of Π'_{No}. Π is in *promise*-$\mathcal{NPL} \cap$ *promise*-co$\mathcal{NPL}$ as a *promise*-$\mathcal{NPL}$ machine can decide it by choosing nondeterministically a bit from the first half of the input and returning it, and is in *promise*-co$\mathcal{NPL}$ as the machine can choose nondeterministically a bit from the second half of the input and return its complement. ■

5 Polylogarithmic Heirarchies

In this section we present two alternative definitions for polylogarithmic-time hierarchies, that is, polylogarithmic-time analogues of $\mathcal{PH}$. Studying their properties, we shall show that both hierarchies are natural extensions of $\mathcal{NPL}$ computations. However, while one definition leads us to a class that (like $\mathcal{NPL}$) does not include even the set `Parity`, the other definition leads us to a class that is equal to $\mathcal{PH}$.

In the previous section we extended our treatment of complexity classes to include promise problems. In this section, however, we return to studying classic decision problems. The first hierarchy we define is based on expressions that have a number of quantified bits that is *polylogarithmic* in the length of the input.

Definition 5.1 *(the class Σ_k^{pl}): For a natural number k, a set $S \subseteq \{0,1\}^*$ is in Σ_k^{pl} if there*

[7]rounding issues are uninteresting

exists a polylogarithmic *function p and a $\mathcal{PLT}$ algorithm V such that $x \in S$ if and only if*

$$\exists y_1 \in \{0,1\}^{p(|x|)} \forall y_2 \in \{0,1\}^{p(|x|)} \cdots Q_k y_k \in \{0,1\}^{p(|x|)} V(x, y_1, ..., y_k) = 1$$

where Q_i is an existential quantifier if i is odd and is a universal quantifier otherwise.

Definition 5.2 *(the class Π_k^{pl}): A set is in Π_k^{pl} if it is the complement of a set in Σ_k^{pl}.*

Definition 5.3 *(Polylogarithmically Limited Polylogarithmic Hierarchy): We denote by $\mathcal{PLH}$ the union over all natural k's of the classes Σ_k^{pl}.*

A different natural polylogarithmic-time hierarchy can be defined by using expressions that have a number of quantified bits that is *polynomial* in the length of the input.

Definition 5.4 (the class *poly*Σ_k^{pl}): *For a natural number k, a set $S \subseteq \{0,1\}^*$ is in polyΣ_k^{pl} if there exists a* polynomial *function p and a $\mathcal{PLT}$ algorithm V such that $x \in S$ if and only if*

$$\exists y_1 \in \{0,1\}^{p(|x|)} \forall y_2 \in \{0,1\}^{p(|x|)} \cdots Q_k y_k \in \{0,1\}^{p(|x|)} V(x, y_1, ..., y_k) = 1$$

where Q_i is an existential quantifier if i is odd and is a universal quantifier otherwise.

Definition 5.5 (the class *poly*Π_k^{pl}): *A set is in polyΠ_k^{pl} if it is the complement of a set in polyΣ_k^{pl}.*

Definition 5.6 *(Polynomially Limited Polylogarithmic Hierarchy): We denote by* poly$\mathcal{PLH}$ *the union over all natural k's of the classes polyΣ_k^{pl}.*

Like in the case of the polynomial-time hierarchy, we note that each level of the polylogarithmic-time hierarchies can be defined recursively, based on the complement of the previous level.

Proposition 5.7 *For every $k \geq 0$, a set S is in Σ_{k+1}^{pl} (respectively, in polyΣ_{k+1}^{pl}) if and only if there exists a polylogarithmic function (respectively, a polynomial) p and a set $S' \in \Pi_k^{pl}$ (respectively, in polyΠ_k^{pl}) such that $S = \{x : \exists y \in \{0,1\}^{p(|x|)} (x, y) \in S'\}$.*

The proof of Proposition 5.7 is straightforward and very much like the similar proof for the recursive definition of $\mathcal{PH}$. The only difference is that some technical difficulties arise due to the fact that Definitions 5.1 and 5.4 have a fixed length for the strings of quantified bits, but these are easily solvable.

5.1 $poly\Sigma_1^{pl} = \Sigma_1^{pl} = \mathcal{NPL}$

Obviously, both $poly\Sigma_0^{pl}$ and Σ_0^{pl} equal $\mathcal{PLT}$. We now show that $poly\Sigma_1^{pl}$ and Σ_1^{pl} both equal $\mathcal{NPL}$. Thus both hierarchies are natural extensions of the notion of nondeterminism in the polylogarithmic setting.

Proposition 5.8 $\Sigma_1^{pl} = \mathcal{NPL}$

Proof: Let S be a set in Σ_1^{pl}, where $x \in S$ if and only if $\exists y \in \{0,1\}^{p(|x|)} V(x,y) = 1$, where p is a polylogarithmic function and V is a $\mathcal{PLT}$ machine. S is decidable by an $\mathcal{NPL}$ machine M that chooses nondeterministically a string $y' = \{0,1\}^{p(|x|)}$ and emulates $V(x, y')$. In the other direction, If S is decidable by an $\mathcal{NPL}$ machine M that runs in time bounded by a polylogarithmic function p, then $x \in S$ if and only if $\exists y \in \{0,1\}^{p(|x|)} V(x,y) = 1$, where V is a machine emulating $M(x)$, using the bits of y to decide what nondeterministic path M takes. ■

Proposition 5.9 $\Sigma_1^{pl} = poly\Sigma_1^{pl}$

Proof: The fact that $\Sigma_1^{pl} \subseteq poly\Sigma_1^{pl}$ is not, as it may seem, syntactically true. This is due to the fact that in Definition 5.4 we defined $poly\Sigma^{pl}$ to have fixed polynomial lengths for the quantified strings. To show that $\Sigma_1^{pl} \subseteq poly\Sigma_1^{pl}$ we note that the $\mathcal{PLT}$ machine in a $poly\Sigma_1^{pl}$ expression can be designed to read only a polylogarithmic number of bits from the beginning of the quantified string y. It now remains to show that $poly\Sigma_1^{pl} \subseteq \Sigma_1^{pl}$.

Let $S \in poly\Sigma_1^{pl}$ be a set such that $x \in S$ if and only if $\exists y \in \{0,1\}^{p(|x|)} V(x,y) = 1$, where p is a polynomial and V is a $\mathcal{PLT}$ machine, and let t be the polylogarithmic function *of the length of* x bounding V's running time. V cannot, of course, read all the bits of y. One can consider the queries made by $V(x,y)$ to different bits of y and the results given to it as a series of pairs of the form (l, σ), where l is the location in y queried by $V(x,y)$, and σ is the bit y_l. Such a series $(l_1, \sigma_1) \dots (l_{t(|x|)}, \sigma_{t(|x|)})$, where $l_i \in \{0,1\}^{log(p(|x|))}$ and $\sigma_i \in \{0,1\}$, that leads to $V(x,y) = 1$, exists if and only if $x \in S$.

To prove that S is in Σ_1^{pl}, we construct a $\mathcal{PLT}$ machine V' such that $x \in S$ if and only if $\exists y' \in \{0,1\}^{t(|x|)(log(p(|x|))+1)} V'(x,y') = 1$. V' will emulate V, reading y' as a series of $t(|x|)$ pairs of the form $(l_1, \sigma_1) \dots (l_{t(|x|)}, \sigma_{t(|x|)})$, where $l_i \in \{0,1\}^{log(p(|x|))}$ and $\sigma_i \in \{0,1\}$. V' emulates V as follows: When V queries the $l'th$ bit of x, V' returns x_l as the result of the query. On the i'th time that V queries y (at location l), V' checks whether $l = l_i$. If it does, V' returns σ_i to V as y_l. Otherwise, V' stops the emulation and returns 0. Finally, when the emulated V halts, V' returns the result returned by V.

As we stated previously, a series of the form $(l_1, \sigma_1) \dots (l_{t(|x|)}, \sigma_{t(|x|)})$ that leads to $V(x,y) = 1$ exists if and only if $x \in S$. In fact, we must state (without loss of generality) that V never queries the same location in its input twice. This is essential so that a series that includes the pairs $(l, 0)$ and $(l, 1)$ will not lead to an emulation of V. Thus, $x \in S$ if and only if $\exists y' \in \{0,1\}^{t(|x|)(log(p(|x|))+1)} V'(x,y') = 1$. ■

By combining Propositions 5.8 and 5.9 we get the following theorem:

Theorem 5.10 *$poly\Sigma_1^{pl} = \Sigma_1^{pl} = \mathcal{NPL}$*

5.2 `Parity` is not in $\mathcal{PLH}$

To see something of the power of $\mathcal{PLH}$ (or, in fact, to see something of its lack of power) we show that `Parity` is not in $\mathcal{PLH}$. We begin by showing that each set in $\mathcal{PLH}$ can be decided by a family of constant depth unbounded fan-in circuits of quasi-polynomial size. Once this has been established we invoke Håstad's bound on the size of circuits that decide `Parity` [11], and are done.

Proposition 5.11 *Let S be in Σ_m^{pl}. S is decidable by a family of constant depth circuits of depth $m+1$ and of size $2^{O(t(n))}$, where t is a polylogarithmic function.*

Proof: We prove the Proposition by induction. We begin by showing that any set in Σ_1^{pl} can be decided by a family of depth 2 unbounded fan-in circuits. A similar claim holds for Π_1^{pl}, but we do not repeat the argument. As the induction step we assume that any set in Π_m^{pl} can be decided by a family of circuits of depth $m+1$ and of quasi-polynomial size, and construct a family of circuits to decide any set in Σ_{m+1}^{pl} using the circuits that decide sets in Π_m^{pl}. Here, too, the induction step holds for Π^{pl}, and the argument is not repeated.

The base case statement is that every set in Σ_1^{pl} can be decided by a family of unbounded fan-in circuits of depth 2 and of quasi-polynomial size:

Recall that any set in Σ_1^{pl} can be decided by an $\mathcal{NPL}$ machine M. By Lemma 3.9, and by the fact that $\mathcal{NPL} \subset \mathcal{NPLQ}$ (by Definition 2.10), for any set S in $\mathcal{NPL}$ and for any length n, there is a set V_S of $t(n)$-restricted positive views for S, where t is a polylogarithmic function. Every $x \in \{0,1\}^n$ is a member of S if and only if it is consistent with some view in V_S.

Note that a simple `And` with fan-in $t(n)$ can decide whether the input is consistent with a particular view $v \in V_S$. This holds because the consistency of a string x with a t-restricted view v can be checked by comparing the value of $t(|x|)$ bits in x to the $t(|x|)$ predetermined values of v, which means a conjunction of $t(|x|)$ bit equalities. We denote the `And` gate corresponding to $v_i \in V_S$ as $\texttt{And}_i$. Using a simple counting argument there are at most $2^{log(n)t(n)}$ positive views in V_S. The circuit deciding instances of S of length n is of the form $\bigvee_i \texttt{And}_i$. Note that a similar construction holds for Π_1^{pl} (by replacing `And`'s with `Or`'s and vice-versa).

We now prove the induction step. Let S be a set in Σ_{m+1}^{pl}. By Proposition 5.7 there exists a set S' in Π_m^{pl} and a polylogarithmic function p so that x is a member of S if and only if $\exists y \in \{0,1\}^{p(|x|)}(x,y) \in S'$. We create a circuit deciding S by constructing $2^{p(|x|)}$ circuits, each deciding S' for a different hard-wired value of $y \in \{0,1\}^{p(|x|)}$, and connecting their outputs using an `Or` gate. As each of the circuits deciding S' has (by induction) depth $m+1$ and size $2^{O(t(n))}$ (for some polylogarithmic function t), the resulting circuit for length n has depth $m+2$ and size $2^{O(t(n))}2^{p(n)}+1$. As both t and p are polylogarithmic functions, assume without loss of generality that $t(n) > p(n)$ and we have for any set in Σ_{m+1}^{pl} a family of circuits of size $2^{O(t(n))}$ and depth $m+2$ deciding it. ■

Theorem 5.12 *(Håstad [11]): Depth k circuits that compute parity require size $2^{c(n^{\frac{1}{k-1}})}$ for some positive constant c*

Now given Proposition 5.11 and Theorem 5.12, we get:

Theorem 5.13 `Parity` *is not in $\mathcal{PLH}$*

5.3 poly$\mathcal{PLH} = \mathcal{PH}$

In contrast to $\mathcal{PLH}$, which we showed not to include `Parity`, and despite the fact that Σ_1^{pl} equals $poly\Sigma_1^{pl}$, the class poly$\mathcal{PLH}$ equals $\mathcal{PH}$. We begin by proving that k-`SAT` $\in poly\Sigma_2^{pl}$. In polynomial-time settings this would be enough to convince us that $\mathcal{NP} \subseteq poly\Sigma_2^{pl}$, but in the polylogarithmic-time setting we shall need to refer to the fact that the reduction of any $\mathcal{NP}$ set to `SAT` is a $\mathcal{PLT}$ reduction.

Proposition 5.14 k-`SAT` $\in poly\Sigma_2^{pl}$

Proof: Consider the following $\mathcal{PLT}$ machine V'. V' gets as input a triplet of strings (x, z, i), where x is a k-`SAT` formula with m clauses over n variables, $z \in \{0,1\}^n$ and $i \in \{0,1\}^{log(m)}$. V' returns 1 if and only if the i'th clause of x is satisfied by the assignment z. To do this, V' reads all the bits of i, then reads the i'th clause in the formula x and finally reads the relevant bits in the assignment z, verifying that the clause is satisfied. Note that the sizes of x and z are related by a polynomial, and that i is polylogarithmic in the length of x. Still, using V' it is easy to construct a machine V that reads only the relevant bits of z and i so that $\exists z \in \{0,1\}^{p(|x|)} \forall i \in \{0,1\}^{p(|x|)} V(x,z,i) = 1$ if and only if x is a satisfiable k-`SAT` formula. ■

We now turn to prove the following:

Proposition 5.15 $\mathcal{NP} \subseteq poly\Sigma_2^{pl}$

Proof: We begin by noting that any $\mathcal{NP}$ set is $\mathcal{PLT}$-reducible to k-`SAT` for some constant $k \geq 3$, as mentioned in Section 4. Not only is such a reduction possible, but the reduced instance is of size polynomial in the length of the original instance (which is not generally required of $\mathcal{PLT}$ reductions).

For any set $S \in \mathcal{NP}$ we show that $S \in poly\Sigma_2^{pl}$ as follows. Let M be the machine reducing instances x of a set S to k-`SAT` formulas of length $r(|x|)$, where r is a polynomial. Let V' be a $\mathcal{PLT}$ machine such that if and only if x is a satisfiable k-`SAT` formula

$$\exists z \in \{0,1\}^{p(|x|)} \forall i \in \{0,1\}^{p(|x|)} V'(x,z,i) = 1$$

We construct the following machine V such that if and only if x is in S

$$\exists z \in \{0,1\}^{p(r(|x|))} \forall i \in \{0,1\}^{p(r(|x|))} V(x,z,i) = 1$$

On input (x,z,i) V emulates the operation of V' on $(M_1^x \dots M_{log(r(|x|))}^x, z, i)$. This means that V is emulating V' on the input $M_1^x \dots M_{log(r(|x|))}^x$ obtained by the reduction applied to x; that is, every query made by V' is answered by V after a polylogarithmic number of computational steps and a polylogarithmic number of queries to x. As p and r are polynomials, so is their composition, and thus S is in $S \in poly\Sigma_2^{pl}$. ■

From here, the following theorem follows easily.

Theorem 5.16 poly$\mathcal{PLH}$ *Equals* $\mathcal{PH}$

Proof: Clearly, poly$\mathcal{PLH} \subseteq \mathcal{PH}$. We show that $\mathcal{PH} \subseteq$ poly$\mathcal{PLH}$ by induction on the levels of the hierarchies, showing that $\Sigma_m \subseteq poly\Sigma_{m+1}^{pl}$. The same holds for Π_m but we do not repeat the proof (although we use this fact in the induction step).

As the base case we note that $\Sigma_1 \subseteq poly\Sigma_2^{pl}$ (by Proposition 5.15). The induction step is as follows: Let S be a set in Σ_{m+1}. This means that there exists a set S' in Π_m and a polynomial function p so that $x \in S$ if and only if $\exists y^{p(|x|)}(x,y) \in S'$. As $\Pi_m \subseteq poly\Pi^{pl}_{m+1}$ it holds that $S' \in poly\Pi^{pl}_{m+1}$. By proposition 5.7 it follows that $S \in poly\Sigma^{pl}_{m+2}$. Thus $\Sigma_{m+1} \subseteq poly\Sigma^{pl}_{m+2}$, and therefore poly$\mathcal{PLH} = \mathcal{PH}$. $\blacksquare$

Acknowledgments

I wish to thank my advisor Oded Goldreich. Oded's classes sparked my interest in theoretical computer science, and without his kind help and extreme patience this thesis would never have come to be. Our meetings were made pleasurable by Oded's good sense of humor and his tendency not to throw heavy items in my general direction.

I would also like to thank Ran Raz, for pointing out to me work done in the past that was similar to my own, and to thank my friends Kfir Barhum and Shai Machness for listening to my endless theories and wild conjectures. A special thanks is due to my wife, Avital Sasson, who tolerates my pseudo-academic lifestyle. The "missing cat" problem is in memory of our cat Lola, who was lost during the writing of this thesis.

References

[1] R. Beals, H. Buhrman, R. Cleve, M. Mosca and R. de Wolf. Quantom Lower Bounds by Polynomials In *FOCS'98*, pages 352–361, 1998.

[2] T. Baker, J. Gill and R. Solovay. Relativizatons of the P =? NP Question. In *SICOMP*, Vol. 4, No. 4, 1975, pages 431-442.

[3] M. Blum and R. Impagliazzo. Generic Oracles and Oracle Classes In *FOCS'87*, pages 118–126, 1987.

[4] H. Buhrman and R. de Wolf. Complexity Measures and Decision Tree Complexity: A Survey In *Theoretical Computer Science* 288(1), pages 21–43, 2002.

[5] B. Chazelle. Property Testing, in *Data-Powered Computing* [a bibliography] http://www.cs.princeton.edu/courses/archive/spring04/cos598B/bib.html

[6] R. Chang, B. Chor, O. Goldreich, J. Hartmanis, J. Hastad, D. Ranjan, and P. Rohatgi. The random oracle hypothesis is false. In *Journal of Computer and System Sciences*, Vol. 49(1), pages 24–39, 1994.

[7] S. Even, A.L. Selman, and Y. Yacobi. The Complexity of Promise Problems with Applications to Public-Key Cryptography. In *Inform. and Control*, Vol. 61, pages 159–173, 1984.

[8] E. Fischer. The art of uninformed decisions: A primer to property testing. In *Bulletin of the European Association for Theoretical Computer Science*, Vol. 75, pages 97–126, 2001.

[9] O. Goldreich. On Promise Problems – A Survey In *Theoretical Computer Science: Essays in Memory of Shimon Even*, Festschrift series of Springer's LNCS (as Vol 3895), pages 254–290, March 2006.

[10] M. Garey and D. Johnson. *Computers and Intractibility, A Guide to the Theory of NP-Completeness* W.H. Freeman, New York, 1979

[11] J. Håstad. *Computational Limitations for Small-Depth Circuits* MIT Press, Cambridge, Ma., 1986

[12] R. Impagliazzo and M. Naor. Decision Trees and Downward Closure In Structure in Complexity Conference, pages 29–38, 1988.

[13] E. Kushilevitz and N. Nisan. *Communication Complexity* Cambridge University Press , Cambridge, 1997.

[14] L. Lov*á*sz, M. Naor, I. Newman and A. Wigderson. Search Problems in the Decision Tree Model In *FOCS'91*, pages 576–585, 1991.

[15] D. Ron. Property testing. In *Handbook on Randomization, Volume II*, pages 597–649, 2001. (Editors: S. Rajasekaran, P.M. Pardalos, J.H. Reif and J.D.P. Rolim.)

[16] E. Schr*ö*dinger. Die gegenw*ä*rtige Situation in der Quantenmechanik In *Naturwissenschaften* Vol. 23, pages 807–812; 823-828; 844-849, 1935.

[17] A. Shamir. IP=PSPACE In *FOCS'90*, pages 11–15, 1990.

[18] G. Tardos. Query Complexity or Why is it Difficult to Seperate $\mathcal{NP}^A \cap \text{co}\mathcal{NP}^A$ from $\mathcal{P}^A$ by Random Oracles. In *Combinatorica*, 9(4), pages 385–392, 1989.