

# Resource Bounds and Subproblem Independence

*R.E. Stearns*      *H.B. Hunt III*  
Department of Computer Science  
University at Albany – SUNY  
Albany, NY 12222  
E-mail: {res,hunt}@cs.albany.edu

September 12, 2003

## Abstract

There are several reasons why some NP-hard problem can be solved deterministically in  $2^{O(n^a)}$  time for some  $a$ ,  $0 < a < 1$ . One reason is that the problem can be solved with a nondeterministic procedure which uses only  $O(n^a)$  space. A second reason is that each instance of the problem exhibits a high degree of “subproblem independence” (specifically  $O(n^a)$  line channelwidth or treewidth). A third is that each instance of the problem can be solved using nondeterministic straight-line programs where the number of variables is only  $O(n^a)$ .

In this paper we show that, after imposing  $q$ -linear bounds on the computation time and obliviousness constraints on the nondeterminism, these three reasons are essentially equivalent. ( $q$ -linear functions are similar to functions of the form  $n(\log n)^k$ .) Specifically, the problem classes associated with these reasons are interchangeable using reductions which are simultaneously polynomial time and  $q$ -linear size-bounded. We call these PQ-reductions.

The classes of problems solvable by these methods we call  $\text{NQ}n^a$ . Because these classes are defined in a very general way, they include a rich variety of problems. We take this as evidence that these classes have “power index”  $a$  (ie. that  $2^{O(n^a)}$  is also a lower time bound). On the other hand, the easiness of all these problems can be derived from “subproblem independence” considerations, and thus the techniques associated with subproblem independence are seen to be more widely applicable than one might expect.

# 1 Introduction

The theory of NP-completeness suggests that many important combinatorial problems are computationally very hard. Yet even though some of these (such as **3SAT**) seem to require  $2^{\Theta(n)}$  time, we know how to solve others in only  $2^{\Theta(\sqrt{n})}$  time (eg. **CLIQUE**, **PARTITION**, **PLANAR-3SAT**) and (given any  $\epsilon > 0$ ) still others in  $2^{\Theta(n^\epsilon)}$  time. Algorithms for these easier NP-hard problems might (depending on constants) be considered practical because, for  $n=64,000$ ,  $2^{n^{1/3}}$  operations can be performed in 19 minutes on a one giga-flop machine.

One way easier problems arise is through restricted problem domains. Interest in restricted domains is evident from the number of times phrases such as “remains NP-complete even if . . .” appear in [10]. Restricted domains are also involved implicitly when one problem is reduced to another since the domain of the first problem is almost always mapped into a sub-domain of the second problem.

Here we consider three approaches to restricting domains. One approach is to restrict the domains to instances where the problems can be solved by certain nondeterministic Turing machines with restricted time and space resources. The second approach is to restrict domains to problems which exhibit a certain amount of “subproblem independence” as discussed in [28]. The third involves solving problems using nondeterministic straight-line programs with restrictions on the number of variables. Our results show that these three methods produce classes of problems which are equally hard. “Equally hard” in this case means “have the same power index” as defined below. These results establish strong links between two seemingly different concepts, namely bounded resources and subproblem independence. These links support the following conclusions:

1. Subproblem independence occurs much more often than one might suspect and is useful for solving a variety of seemingly unrelated problems.
2. The obvious algorithms for exploiting subproblem independence seem to be the best possible (in the same sense that brute force seems to be best for solving **SAT**.)
3. When attempting to find an improved algorithm for a hard problem, one thing to think about

is minimizing the amount of memory needed by a nondeterministic algorithm for that problem

First, following ideas from [15], we present a formalism for describing restricted domain problems:

**Definition 1.1** *Let  $L$  and  $D$  be subsets of  $\Sigma^*$ . The **language problem**  $\mathcal{L} = (L, D)$ , also called  **$L$  restricted to domain  $D$** , is the problem of determining if a string  $w$  in  $D$  is a member of language  $L$ . A Turing machine  $\mathcal{T}$  (with a read-only input tape) **solves  $\mathcal{L}$**  in time  $T(n)$  and space  $S(n)$  if and only if, for  $w$  in  $D$ ,  $\mathcal{T}$  determines if  $w$  is in  $L$  using at most  $T(|w|)$  moves and  $S(|w|)$  scratch tape squares. A **reduction** from language problem  $(L_1, D_1)$  to  $(L_2, D_2)$  is a mapping  $R$  from  $D_1$  to  $D_2$  such that  $w \in L_1$  if and only if  $R(w) \in L_2$ .*

Notice that the Turing machine  $\mathcal{T}$  which solves  $\mathcal{L}$  is not required to decide if an input is in the domain  $D$ . There is no restriction on what  $\mathcal{T}$  would do if presented with an input not in  $D$ . When presented with such an input  $w$ ,  $\mathcal{T}$  might use more than  $S(|w|)$  tape, use more than  $T(|w|)$  time, or even run forever without stopping. Also notice there is no restriction on  $D$ . Set  $D$  could be very hard to compute or even be unrecognizable. Although our interest is primarily in nice domains, none of our results make assumptions about  $D$ . Restricted domain problems are called “promise problems” in [9, 16, 25] because  $\mathcal{T}$  works well if we promise only to supply inputs from  $D$ .

Every language problem  $\mathcal{L}$  has a “power index” [27] defined as follows:

**Definition 1.2** *If  $\mathcal{L}$  is a language problem, the **power index** of  $\mathcal{L}$  is the greatest lower bound on the set  $\{a \mid \mathcal{L} \text{ can be solved in time } O(2^{n^a})\}$  if this set is nonempty and is  $\infty$  otherwise. We denote the power index of  $\mathcal{L}$  by  $\text{POWER}(\mathcal{L})$ . When  $\mathcal{L} = (L, \Sigma^*)$  where  $\Sigma$  is the input set of  $\mathcal{L}$ , the power index of language problem  $\mathcal{L}$  is also called the **power index** of language  $L$ .*

Notice that having power index  $a$  does not imply that  $\mathcal{L}$  can be solved in  $O(2^{n^a})$  time or even  $O(n^{n^a})$  time. It only says that any  $O(2^{n^{\epsilon}})$  ( $\epsilon > 0$ ) time algorithm for  $L$  can be improved upon. We define power index this way so that every language problem has a power index.

When  $2^{n^a}$  is time constructible, complexity theory [13] tells us via a diagonal argument that there is a language  $L$  which has power index  $a$ . The function  $2^{n^a}$  is time constructible for all rational  $a$ , so power indices impose a nontrivial hierarchy on DEXPTIME. Using unrecognizable domains  $D$ , a language problem  $\mathcal{L} = (L, D)$  having power index  $a$  can be constructed for any real  $a \geq 0$ . (Instead of having a diagonalizing machine try to stop itself in  $2^{n^a}$  time, simply define  $D$  to be the set of inputs  $w$  such that the machine stops in  $2^{|w|^a}$  or fewer steps.)

Since all problems in P have power index zero, a proof that some NP-complete problem has a non-zero power index would imply  $P \neq NP$  and that all NP-hard problems have non-zero power indices. Since proving some NP-complete has power index  $a > 0$  seems out of reach, we instead offer evidence that certain problems do have power index  $a$ . We do this by offering several classes of problems where the inter-relationships among the classes imply that their hardest members have equal power index and where that power index seems to be  $a$ .

The power index concept extends to sets of language problems as follows:

**Definition 1.3** *If  $C$  is a set of language problems, we define the **power index** of  $C$  to be the least upper bound on the set  $\{\text{POWER}(\mathcal{L}) \mid \mathcal{L} \in C\}$  or to be  $\infty$  if the set is unbounded.*

Many of our theorems concern sets of problems rather than individual problems. To help the reader to distinguish between problem sets and single problems, we use sans serif font for problem sets and **bold face** for individual problems.

In [27], we considered some consequences of the assumption that the power index of **3SAT** is one; a stronger hypothesis than  $P \neq NP$ . (Intuitively, the power index of **3SAT** is one if we cannot do better in the worst case than testing all assignments.) We see the results here as supporting a generalization of this hypothesis, namely that the power index of class  $LCn^a\text{-SAT}$  (defined below in Section 3) is  $a$  (where  $a$  is a parameter satisfying  $0 \leq a \leq 1$ ).

In general, polynomial time reductions are too crude for power index studies since they can map problems of power index  $a$  into problems of power index  $b$  where  $b/a$  is arbitrarily small. To make inferences about power index, we must also consider the **size** of the reductions:

**Definition 1.4** A function  $f(n)$  is called **q-linear** if  $f(n)$  is  $O(n^a)$  for all  $a > 1$ . A reduction  $R$  from language problem  $\mathcal{L}_1$  to language problem  $\mathcal{L}_2$  is called a **PQ-reduction** if it is polynomial time and there is a q-linear function  $f$  such that, for all strings  $w$ ,  $|R(w)| \leq f(|w|)$  (i.e. the output of the reduction has **q-linear size**).

The term “q-linear” is a generalization of the term “quasi-linear” which is frequently used in the literature to mean  $O(n(\log n)^i)$  for some  $i$ . Quasi-linear functions have often been used to severely limit (usually as a time bound) the resources of a reduction and still permit certain standard reductions to qualify. This is true for the reductions of Schnorr [24] which we relate to our work in Section 8. However, for most complexity theory applications, the important property of quasi-linear time bounded reductions is that they are both polynomial time and quasi-linear size-bounded. Grädel [11] uses q-linear time bounds in a study of linear time. He calls them “functions in  $\text{TIME}(n^{1+})$ ”. He shows that q-linear time bounds are superior to quasi-linear time bounds in that q-linearity is more robust under change of computer model.

Our PQ-reductions can take a lot more time than quasi-linear time reductions (namely polynomial time instead of quasi-linear time). They have (slightly) more size because the function  $n(\log n)^i$  is  $O(n^a)$  for all  $a > 1$  and so quasi-linear size implies q-linear size. Thus all quasi-linear time reductions are PQ-reductions. The use of q-linear size rather than quasi-linear size makes our definitions slightly more general and (more importantly) dovetails naturally with the power-index concept, but all of the PQ-reductions from our proofs are actually quasi-linear size. The key property of PQ-reductions is that they preserve power indices in the following sense:

**Proposition 1.5** *If  $\mathcal{L}_1$  is PQ-reducible to  $\mathcal{L}_2$ , then  $\text{POWER}(\mathcal{L}_1) \leq \text{POWER}(\mathcal{L}_2)$ .*

**Proof.** Let  $R$  be the PQ-reduction. If  $\mathcal{L}_2$  can be solved in  $O(2^{n^a})$  time for some  $a < \text{POWER}(\mathcal{L}_1)$ , then  $\mathcal{L}_1$  can also be solved in  $O(2^{n^b})$  time for  $b > a$  by first reducing input  $w$  to  $R(w)$  (which takes polynomial time) and then testing  $R(w)$  for membership in  $\mathcal{L}_2$  (which takes  $O(2^{|R(w)|^a})$  time). This contradicts the assumption that  $a < \text{POWER}(\mathcal{L}_1)$ . ■

Like polynomial time reductions, PQ-reductions also compose:

**Proposition 1.6** *If  $R_1$  is a PQ-reduction from  $\mathcal{L}_1$  to  $\mathcal{L}_2$  and  $R_2$  is a PQ reduction from  $\mathcal{L}_2$  to  $\mathcal{L}_3$ , then the function composition  $R = R_2(R_1)$  is a PQ-reduction from  $\mathcal{L}_1$  to  $\mathcal{L}_3$ .*

**Proof.** Clearly  $R$  is a reduction. It can be performed in polynomial time by first performing  $R_1$  and then performing  $R_2$ . For input  $w$ , the output  $R_2(R_1(w))$  has size  $q_2(q_1(|w|))$  where  $q_1$  and  $q_2$  are the q-linear functions bounding the sizes of  $R_1$  and  $R_2$ . Since the q-linear functions are closed under function composition, this size bound is also q-linear. ■

Our three methods of restricting problem domains each involve a single parameter  $a$  where  $0 \leq a \leq 1$ . In each case, the parameter is tied to functions of “polynomial index”  $a$  defined as follows:

**Definition 1.7** *The **polynomial index** of a function  $p$  from integers to integers is the greatest lower bound on the set  $\{a \mid p \text{ in } O(n^a)\}$  if this set is nonempty and  $\infty$  otherwise.*

The function  $p(n) = n^a$  has polynomial index  $a$  and, for purposes of informal discussion, we say that the polynomial index  $a$  functions are “approximately  $n^a$ ”. Definition 1.7 is written in such a way (using greatest lower bounds) that every function has a polynomial index. The q-linear functions are those functions having polynomial index one or less. Functions with polynomial index less than one are sub-linear. Our main interest in polynomial index is the following:

**Proposition 1.8** *If function  $p(n)$  has polynomial index  $a$  and  $\mathcal{L} = (L, D)$  is a language problem that can be solved in  $O(2^{p(n)})$  time, then  $\mathcal{L}$  has a power index of  $a$  or less.*

**Proof.** Suppose  $\mathcal{L}$  has power index  $b$  where  $b > a$  and let  $c$  satisfy  $b > c > a$ . By Definition 1.3,  $\mathcal{L}$  can not be recognized in time  $2^{n^c}$ . But  $2^{n^c}$  is greater (almost everywhere) than any function in  $O(2^{p(n)})$ , so the assumption that  $\mathcal{L}$  can be solved in  $O(n^{p(n)})$  is contradicted. ■

Functions in  $O(n^a)$  have polynomial index  $a$  or less and so problems solved in time  $2^{O(n^a)}$  (ie. time  $O(2^{f(n)})$  for some  $f$  in  $O(n^a)$ ) all have power index  $a$  or less. Note that  $2^{O(n^a)}$  includes functions of the form  $n^k c^{n^a}$  for constants  $k$  and  $c$  but not the function  $n^{n^a}$  even though  $n^{n^a}$  which is  $2^{n^a \log_2 n}$  and  $n^a \log_2 n$  also has polynomial index  $a$ .

Sections 2, 3, and 4 formalize the three approaches to restricting domains. In Section 2, for  $0 \leq a \leq 1$ , we define a language problem class  $\text{NQ}n^a$  based on PQ-reductions to problems solvable on a multi-tape nondeterministic Turing machine in simultaneous  $q$ -linear time and  $p_a(n)$ -space where  $p_a(n)$  is a function with polynomial index  $a$ . The nondeterministic machines must satisfy an “obliviousness” condition which restricts the motions of the tapes. Intuitively,  $\text{NQ}n^a$  is the set of language problems that can be recognized by a certain two step process. The first step is a preprocessing step carried out in deterministic polynomial time. The second step is an analysis performed nondeterministically on a choice-oblivious Turing machine in  $q$ -linear time and  $p_a(n)$ -space. (Choice-obliviousness is a weak form of obliviousness.) To solve such a problem deterministically, one must take the input  $w$  and somehow figure out if any of the configurations the nondeterministic machine can reach in  $p_a(|w|)$  steps (or less) is an accepting configuration. Since there can be  $2^{\Theta(p_a(|w|))}$  such configurations, it is plausible that some problem will require brute force and  $2^{\Theta(p_a(|w|))}$  steps will be required. This makes it plausible that some of these problems cannot be solved significantly faster. On this basis, we conjecture that the class  $\text{NQ}n^a$  has power index  $a$ .

Bounds on nondeterminism have been studied by many authors. A good list of references can be found in [6]. These studies usually limit the amount of nondeterminism by bounding the the number of nondeterministic moves as a function of input size. Bounded nondeterministic problems are modeled as “guess-then-check” problems in [6] where all the nondeterministic guesses are made first and the guesses then checked by some kind of automaton. Our results show that, for purposes of making nondeterministic methods deterministic, there is more to “the amount of nondeterminism” than just counting the number of nondeterministic moves. We study situations where, without changing the number of nondeterministic moves, deterministic time requirements vary with the amount of nondeterministic space used.

To illustrate the weakness of just counting guesses, suppose some nondeterministic procedure makes nondeterministic assignments to variables  $x$  and  $y$ . Normally the effect of these choices on brute force computation is multiplicative; the number of choices for  $x$  must be multiplied by the number of choices for  $y$  to get the number of cases that must be deterministically explored.

Suppose, however, that the nondeterministic algorithm uses variable  $x$  to solve a subproblem  $A_x$ , uses  $y$  to solve a subproblem  $A_y$ , and then combines the results of  $A_x$  and  $A_y$  to get a result for the whole problem. Suppose further that subproblems  $A_x$  and  $A_y$  are independent in that no information need be passed from one sub-calculation to the other. Then the effects of  $x$  and  $y$  are just additive. There is a corresponding reduction in the amount of space needed as the space used to store variable  $x$  while solving  $A_x$  can be reused to store variable  $y$  while solving  $A_y$ . In this way, the following three concepts are tied together: nondeterministic space, deterministic time, and subproblem independence.

We have a special interest in problem sets that are complete for  $\text{NQN}^a$  in the following way:

**Definition 1.9** *A set of problems  $C$  is  $\text{NQN}^a$ -complete if and only if,*

1. *for each language  $\mathcal{L}_1$  in  $\text{NQN}^a$ , there is a language  $\mathcal{L}_2$  in  $C$  such that there is a PQ-reduction from  $\mathcal{L}_1$  to  $\mathcal{L}_2$ ,*
2. *each  $\mathcal{L}_2$  in  $C$  is in  $\text{NQN}^a$ .*

Note that  $\text{NQN}^a$ -completeness is with respect to a set of languages rather than a single language as in the case of NP-completeness. However, within each  $\text{NQN}^a$ -complete class we study, the languages are very similar. Each class involves a single language under a variety of domain restrictions. Symbolically, each class has a language  $L$  and a set  $\mathcal{D}$  of possible domain restrictions on  $L$  such that the class is described by the set  $\{(L, D) \mid D \in \mathcal{D}\}$ . In each case, the unrestricted problem  $(L, \Sigma^*)$  is not in the class. Using classes is in keeping with our interests in how restricted domains arise and how they translate into each other. It is usually possible to map the class members into a single representative, but in an uninteresting and artificial way. We illustrate this technique with one of our classes (see Theorem 3.6). We note that Grädel in [11] sometimes uses a set of problems (sequences in his case) for his complexity results.

In Section 3 we define, for all  $0 \leq a \leq 1$ , a set of language problems  $\text{LCN}^a\text{-SAT}$ . “LC” stands for “line channelwidth” which specializes the concept of “channelwidth” introduced in [28].<sup>1</sup> Each

---

<sup>1</sup>Channelwidth is based on trees rather than linear orderings, but the line channelwidth is never more than  $c \log_2 n$  where  $c$  is the channelwidth and  $n$  the problem size.

problem in  $LCn^a$ -SAT is obtained from **SAT** by restricting the domain to formulas of line channelwidth bounded above by a function of polynomial index  $a$ . Channelwidth and line channelwidth are closely related to several other concepts in the literature including treewidth and pathwidth for graphs [2, 4, 20], bounded bandwidth [18], and the parameters of certain separator theorems [17]. The literature uses limitations on treewidth, pathwidth, channelwidth, bandwidth, and separator size to define easier versions of many NP-complete problems. Although much of this literature concerns constant bounds (and associated polynomial algorithms) the concepts apply equally well to bounds of small polynomial index (implying easier exponential algorithms). The set of problems to which this literature applies is very large and includes a majority of the most famous NP-complete problems. The results in this paper are a way of applying the algorithmic methods from this literature to a much larger set of problems.

We are, in effect, using a parameter  $a$  to slice up SAT into more tractable sub-pieces. A similar thing happens in “parameterized complexity” (as in [8] and [5]) where a parameter  $k$  is used to slice up NP-complete problems into more tractable sub-pieces. However the approaches are otherwise considerably different. Our slices can be solved in  $O(2^{p(n)})$  time (where  $p$  has polynomial index  $a$ ),  $O(2^{n^a})$  time for example, whereas the parameterized classes are solved in  $f(k)n^{g(k)}$  time (polynomial for fixed  $k$ ) where  $f$  and  $g$  are (possibly fast growing) functions. Parameter  $a$  varies between zero and one whereas  $k$  is an arbitrary integer. Our slices are still NP-complete whereas the parameter  $k$  slices are in P.

For any problem  $\mathcal{L}$  in  $LCn^a$ -SAT,  $\mathcal{L}$  can be solved in simultaneous deterministic  $O(2^{p(n)})$  time and  $q$ -linear space where  $p(n)$  is some function having polynomial index  $a$ . This is achieved using generic algorithms from [28] which exploit subproblem independence, or by using familiar algorithms which embody the same principles such as those in [2, 4, 17, 18, 20] and many others. Our hypothesis that  $LCn^a$ -SAT has power index  $a$  says in effect that no algorithms do can better. In Section 5, we find out that  $LCn^a$ -SAT is  $NQn^a$ -complete.

In Section 4 we introduce the concept of a nondeterministic straight-line program (NSLP). Like a Turing machine, a NSLP involves a sequence of computational steps. But like a formula,

variables (or memory locations) can be referenced in any order (instead of requiring access only to adjacent tape locations). This concept enables a more understandable presentation of the main results by giving separate consideration to going between Turing machines and programs (which entails memory management issues) and to going between programs and formulas (which entails going between the dynamic and the static). By restricting the number of variables allowed in a NSLP, we get classes analogous to  $NQn^a$  and  $LCn^a$ -SAT called  $NSLPn^a$ -EXE( $S$ ). We believe our concept of a NSLP will be useful in other contexts since NSLPs offer some of the programming conveniences of random access machines and yet have many of the nice properties of acyclic circuits.

In Section 5, we prove the main results that, for a given  $a$ , the classes defined in Sections 2, 3, and 4 are all inter-reducible by PQ-reductions and consequently all have the same power index. One implication of these results is that the class  $NQn^a$  could have been equivalently defined using PQ-reductions to  $LCn^a$ -SAT thus avoiding any reference to Turing machines. Another implication is that the power index of the more inclusive class  $NQn^a$  is inherited by the other two. Thus, if  $NQn^a$  has power index  $a$  (which seems plausible), then our conjecture that  $LCn^a$ -SAT has power index  $a$  must also be true. The  $NQn^a$ -completeness of  $LCn^a$ -SAT is implied.

To strengthen the evidence that  $NQn^a$  has power index  $a$ , it would be nice to show that this concept could also have been based on memory bounded nondeterministic RAMs or on programming languages with arrays or pointer variables. In Section 6, we give a somewhat informal indication that such a basis is possible. We describe how a “nondeterministic database” can be implemented obliviously using NSLP program segments and argue that this allows one to imitate RAMs and pointer variables.

In Section 7, we consider **CLIQUE** and **PARTITION**. These problems are easy enough to be in  $NQn^{1/2}$  because they have known  $2^{O(\sqrt{n})}$  algorithms [27]. However these algorithms do not imply that these problems are in  $NQn^{1/2}$  because the algorithms are not based on limited nondeterministic use of memory or on subproblem independence. We provide alternative nondeterministic algorithms which do show that these problems are in  $NQn^{1/2}$ . There is also a brief discussion of **PLANAR-SAT** where known algorithms already imply the problem is in  $NQn^{1/2}$ .

Finally, in Section 8, we give some overall perspective about our work that is best explained after the results have been presented. In particular, it is observed that our constructions imply that Turing machine computations can be reduced to **SAT** by a reduction which is as efficient in size as the quasi-linear reductions of [24, 21, 7] and which has the complexity implications of the reductions in [18].

## 2 The Class $\text{NQ}n^a$

The Turing machine model we use is the standard “off-line” multi-tape model: the machine has a two-way read-only input tape and the space complexity is defined in terms of the number of squares visited on the “work” or “scratch” tapes. Our class  $\text{NQ}n^a$  is based on “choice-oblivious” Turing machines defined as follows:

**Definition 2.1** *A nondeterministic Turing machine  $\mathcal{T}$  is called **choice-oblivious** if and only if, for any input  $w$ , the position of the tape heads during any computation depends only on the number of moves taken to reach the configuration.  $\mathcal{T}$  is called **weakly choice-oblivious** if the above condition is satisfied just for the input tape head.*

For choice-oblivious machines, the head motions depend on the input but are the same regardless of which branch of the nondeterministic computation is followed. For weakly choice-oblivious machines, it is just the motions on the input tape that is independent of the nondeterministic choices. A deterministic Turing machine is always choice-oblivious since there is only one computation sequence for a given input. A choice-oblivious Turing machine is always weakly choice-oblivious. Later we discuss how a choice-oblivious Turing machine can efficiently simulate a weakly choice-oblivious machine.

**Definition 2.2** *For all  $a$ ,  $0 \leq a \leq 1$ ,  $\text{NQ}n^a$  is the set of all language problems  $\mathcal{L}$  such that, for some function  $p$  of polynomial index  $a$ ,  $\mathcal{L}$  is PQ-reducible to a language problem  $\mathcal{L}'$  recognized by a nondeterministic choice-oblivious Turing machine in simultaneous  $q$ -linear time and  $p(n)$  space.*

The PQ-reduction in Definition 2.2 can be thought of as a deterministic preprocessing step which puts the input problem in a form suitable for space efficient choice-oblivious nondeterministic computation. The possibility of preprocessing means that all PQ-inter-reducible representations of a given problem have equal status in the sense that all will be in  $\text{NQN}^a$  or none will be. It also allows  $\text{NQN}^a$  to behave like a “hardness” class under PQ-reductions:

**Proposition 2.3** *For all  $a$ ,  $0 \leq a \leq 1$ , if language problem  $\mathcal{L}_1$  is in  $\text{NQN}^a$  and language problem  $\mathcal{L}_2$  is PQ-reducible to  $\mathcal{L}_1$ , then  $\mathcal{L}_2$  is in  $\text{NQN}^a$ .*

**Proof.** By Proposition 1.5, the PQ reduction from  $\mathcal{L}_2$  to  $\mathcal{L}_1$  followed by the PQ reduction mapping  $\mathcal{L}_1$  to the language problem  $\mathcal{L}'$  of Definition 2.2 is also a PQ reduction from  $\mathcal{L}_2$  to  $\mathcal{L}'$ . Thus  $\mathcal{L}_2$  is also in  $\text{NQN}^a$  by Definition 2.2. ■

Pippinger and Fischer [19] showed that  $m$  moves of an on-line Turing machine can be simulated by an “oblivious” on-line Turing machine in  $m \log(m)$  moves. We close this section with some discussion of how their techniques can be applied to weaken the obliviousness condition used in Definition 2.2. In particular, we discuss how weakly choice-oblivious Turing machines can be simulated by choice-oblivious Turing machines. The reader may skip to the next section since this discussion requires some understanding of Pippinger-Fisher and is not needed later.

First the good news. Although the simulation from [19] was intended for deterministic Turing machines, it clearly applies also to nondeterministic Turing machines. Our weakly choice-oblivious machines are less oblivious than [19] (since our tape movements can depend on the input) and this makes it easier for our machines to simulate other machines.

The bad news is that the oblivious simulator from [19] uses space  $m \log m$  to simulate  $m$  moves of the original machine, regardless of how much space the original machine actually uses. Since we are interested in sub-linear space, this is too much. If we impose some kind of “constructibility” constraints on the space bound, the simulator can mark off the space needed in advance and then run a modified simulation which doesn’t look far beyond the marked off space. The marking of space will not be oblivious of the input size, but this is permitted under our definitions.

A more serious technical problem is the difficulty of accommodating the two-way read-only input tape of the machine to be simulated. The construction in [19] requires the simulating machine to rearrange the information on each of the simulated tapes, an activity not allowed on read-only tapes. Ordinarily, we could get around this problem by copying the information from the read-only input tape to a read-write work tape. However, because we are seeking to use sub-linear amounts of work tape, we don't have enough work space to hold a copy of the input. We do not know of any technique for getting around this problem, and so we settle for simulations of weakly choice-oblivious Turing machines. For these machines, the movement of the input tape is already oblivious and the need to modify the input tape movement disappears.

In summary, Definition 2.2 could be changed to “weakly-choice oblivious” without changing  $NQn^a$ . We work with the definition as given to avoid formalizing the above paragraphs. The proof of Lemma 5.4 indicates a third possibility, namely changing the definition to say “recognized by an on-line real-time Turing machine”.

### 3 CNF Formulas with Channelwidth Bounds

As usual, CNF formulas will be constructed from clauses, each of which is the disjunction of literals (negated and unnegated variables) and constants (TRUE or FALSE). For our purposes, the ordering of the clauses is important.

**Definition 3.1** *A CNF formula  $F$  is a list of clauses  $c_1, \dots, c_l$  for some integer  $l$ . For each  $i$ ,  $1 \leq i \leq l$ , let  $V_i$  be the set of variables  $x$  in clause  $c_i$  and let  $X_i$  be the set of variables  $x$  such that  $x$  in  $V_j$  for some  $j \leq i$  and  $x$  in  $V_k$  for some  $k \geq i$ . The variables in  $X_i$  are called the **channel variables** at  $c_i$ . The value  $\max_i |X_i|$  is called the **line channelwidth** of  $F$ . The length of  $F$  written by  $|F|$  is defined by  $|F| = \sum_i |c_i|$  where  $|c_i|$  is the number of literals and constants in  $c_i$ . The set of all satisfiable formulas is called **SAT**.*

Intuitively, when evaluating clauses in left-to-right order,  $X_i$  is the set of variables whose value must be remembered while evaluating clause  $c_i$ . This set includes variables appearing in  $c_i$  and variables appearing in earlier clauses (clauses on the left) which also appear in later clauses (clauses

on the right). Thus line channelwidth is the maximum number of variables that must be remembered at any one time when evaluating from left-to-right and is thus a bound on the amount of space needed. By a similar argument, it is also the maximum for right-to-left evaluation.

The order of the clauses is important here because rearranging the clauses can change the line channelwidth. At the end of this section, we discuss the complexity of taking an unordered clause set and finding an order with favorable line channelwidth, but this discussion is not germane to our formalism.

The next definition formalizes the idea of limited line channelwidth:

**Definition 3.2** *For any function  $p : N \rightarrow N$ , let  $\mathbf{LC}(p)$  denote the language problem  $(\mathbf{SAT}, D)$  where  $D$  is the set of CNF formulas  $F$  having line channelwidth less than or equal to  $p(|F|)$ . For all  $a$ ,  $0 \leq a \leq 1$ , let  $\mathbf{LC}n^a\text{-SAT}$  denote the set of language problems  $\{\mathbf{LC}(p) \mid p \text{ has polynomial-index } a\}$ . Let  $\mathbf{LC}n^a\text{-3SAT}$  be the set of language problems from  $\mathbf{LC}n^a\text{-SAT}$  restricted to formulas with clauses having at most 3 literals.*

Intuitively, a language in  $\mathbf{LC}n^a\text{-SAT}$  is  $\mathbf{SAT}$  restricted to formulas with line channelwidth approximately  $n^a$ . Note that, under Definition 1.1, an algorithm to solve some  $\mathbf{LC}(p)$  is not required to verify that the line channelwidth of the input falls within the bound  $p$ . Thus  $\mathbf{LC}(p)$  is defined even if  $p$  is difficult or impossible to compute. However, when  $p$  can be computed in polynomial time,  $\mathbf{LC}(p)$  has the same power index as the unrestricted problem “set of all satisfiable formulas  $F$  having line channelwidth  $p(|F|)$  or less.” This follows immediately from the fact that finding the line channelwidth of a list of terms is easy.

There are two seemingly different reasons why language problems in  $\mathbf{LC}n^a\text{-SAT}$  can be solved in  $O(2^{p(n)})$  time for some  $p(n)$  of polynomial index  $a$ . One reason which is made apparent by the proof of the main results (and Lemma 5.3 in particular) is that the problems can be solved in  $q$ -linear time using nondeterministic straight-line programs that have only  $p(n)$  variables. The second reason is that, the smaller the line channelwidth, the more quickly the problems can be solved by a certain deterministic procedure which recursively breaks the problem in to smaller and smaller “independent subproblems”. To understand the nature of this “subproblem independence”,

we give a brief explanation of this recursive procedure. The discussion which follows is a special case of material covered in [28].

If  $V$  is a set of variables, we let  $\Gamma(V)$  denote the set of assignments to  $V$ . If  $c$  is a clause and  $\gamma$  an assignment in  $\Gamma(V)$ , then  $c[\gamma]$  denotes the clause obtained by replacing each occurrence of a variable  $x$  from  $V$  in  $c$  by the constant assigned to  $x$ . If  $F = c_1, \dots, c_l$  is a formula, we let  $F[\gamma]$  denote  $c_1[\gamma], \dots, c_l[\gamma]$ . For any  $i$ , let  $F_L^i$  denote the formula  $c_1, \dots, c_{i-1}$  and  $F_R^i$  denote the formula  $c_{i+1}, \dots, c_l$ . For any assignment  $\gamma$  in  $\Gamma(X_i)$ ,  $F_L^i[\gamma]$  and  $F_R^i[\gamma]$  have no variables in common. Thus  $F$  is satisfied by an extension of  $\gamma$  if and only if  $\gamma$  satisfies clause  $c_i$  and both  $F_L^i[\gamma]$  and  $F_R^i[\gamma]$  are satisfiable.

The above suggests the following divide and conquer algorithm for testing  $F$ :

1. Find a value of  $i$  such that  $|F_L^i| \leq |F|/2$  and  $|F_R^i| \leq |F|/2$ .
2. Consider all assignments  $\gamma$  in  $\Gamma(X_i)$  and test to see if clause  $c_i$  is satisfied and (using recursion) to see if each of the formulas  $F_L^i[\gamma]$  and  $F_R^i[\gamma]$  are satisfiable.
3. If one of the assignments in Step 2 passes the tests, then  $F$  is satisfiable. Otherwise  $F$  is not satisfiable.

The correctness of the algorithm depends critically (in Step 2) on the fact that subproblems  $F_L^i[\gamma]$  and  $F_R^i[\gamma]$  are “independent” in the sense that they have no variables in common. If these two subformulas had a variable in common, then the satisfiability of the formulas individually would not imply the satisfiability of their conjunction.

To speed up the recursion, it is helpful to compute the various  $i$  from Step 1 in advance. That is, find the  $i$  which breaks the formula in half, then the  $i$ 's which break  $F_L^i[\gamma]$  and  $F_R^i[\gamma]$  in half, and so fourth. These can be stored in a tree so that each procedure call can be given its  $i$  by the calling procedure. It is also helpful to pre-compute the list of variables that each call must assign. If the call is associated with  $c_i$ , these variables are the channel variables at  $c_i$  minus the channel variables already assigned by earlier calls. For a reason given later, it is also useful to give each variable its own memory location even though subproblem independence tells us that some variables can share

space. The space needed to do all this is  $O(|F|)$  words of memory (or  $O(|F| \log |F|$  bits) whether or not variable space is shared.

The depth of the recursion is at most  $\log_2 |F| + 1$ . Each call involves assignments to  $k$  or fewer variables so each clause is tested for at most  $2^{k(\log_2 |F| + 1)}$  assignments. If  $k = p(|F|)$  where  $p$  is a function of polynomial index  $a$ , then the total time is at most  $2^{p'(|F|)}$  where  $p'(n) = p(n)(\log_2(n) + 1) + \log_2(n)$  also has polynomial index  $a$ .

The procedure also finds a satisfying assignment (if any) in addition to answering satisfiability. If the program terminates with a positive answer to satisfiability, the variables corresponding to the formula variables contain a satisfying assignment. This assignment would not be readily available if variable locations were shared.

**Proposition 3.3** *For all  $a$ ,  $0 \leq a \leq 1$ , the power index of language problems in  $\text{LCn}^a\text{-SAT}$  is at most  $a$ .*

**Proof.** From previous discussion, the algorithm solves  $\text{LC}(p)$  in  $O(2^{p'(|F|)})$  time where  $p'$  has polynomial index  $a$  and the result follows from Proposition 1.8. ■

The situation with regard to  $\text{LCn}^a\text{-3SAT}$  is as follows:

**Proposition 3.4** *There is a PQ-reduction  $R$  from  $\text{SAT}$  to  $\text{3SAT}$  such that, for all  $a$ ,  $0 \leq a \leq 1$ ,  $R$  reduces every problem in  $\text{LCn}^a\text{-SAT}$  to a problem in  $\text{LCn}^a\text{-3SAT}$ .*

**Proof.** The standard reduction from  $\text{SAT}$  to  $\text{3SAT}$  breaks large clauses into shorter clauses by introducing new variables. This can be done for ordered formulas in a straightforward way so that each new variable appears only in adjacent clauses and only two new variables are used in any of the new clauses. For example, clause  $v \wedge w \wedge x \wedge y \wedge z$  is replaced by the sequence of clauses  $v \wedge w \wedge a$ ,  $\bar{a} \wedge x \wedge b$ , and  $\bar{b} \wedge y \wedge z$ . Consequently the line channelwidth is increased only by 2 and the result follows. ■

The previous theorem has features which re-occur frequently in later theorems. The theorem says that each problem in one set can be reduced to a problem in another set, and that *all the reductions are accomplished by the same mapping*. Restrict this mapping to the domain of a problem

in the first set and it performs one reduction, restrict it to the domain of another problem and you get a second reduction. The mapping itself (without domain restriction) is also a reduction, in this case from **SAT** to **3SAT**.

**Corollary 3.5** *For all  $a$ ,  $0 \leq a \leq 1$ ,  $\text{LC}n^a\text{-SAT}$  and  $\text{LC}n^a\text{-3SAT}$  have the same power index.*

We now show how members of  $\text{LC}n^a\text{-SAT}$  can be reduced to a single problem:

**Theorem 3.6** *If  $p(n)$  has polynomial index  $a$ ,  $0 \leq a \leq 1$ , and  $\lceil p(n)^{1/a} \rceil$  is polynomial time constructible, then there is a PQ-reduction from  $\text{LC}n^a\text{-SAT}(p(n))$  to  $\text{LC}n^a\text{-SAT}(n^a)$ .*

**Proof.** The reduction is as follows: Find the input size  $n$  and compute  $\lceil [p(n)^{1/a}]/n \rceil$  which is constructible in polynomial time. Take the input formula and output the same formula with each clause repeated  $\lceil [p(n)^{1/a}]/n \rceil$  times. The size of the new formula is  $\lceil p(n)^{1/a} \rceil$  which is q-linear because  $p(n)$  has polynomial index  $a$ . The line channelwidth remains  $p(n)$ . Thus the new formula is in  $\text{LC}n^a\text{-SAT}(n^a)$ . ■

The reduction in the proof is achieved by padding and does not make the problem easier to solve even though, technically, the problem has been moved to a class with a more constrained time bound (assuming  $p(n) > n$ ). The reduction serves as an example of a padding idea that can be applied to the other classes in this paper, but will not be presenting other theorems of this type.

We close this section by discussing the alternative of defining the domain of  $\text{LC}(p)$  to be the set of CNF formulas for which there exists some rearrangement of the clauses having line channelwidth no larger than  $p(|F|)$ . The reader who is not interested in such discussion can skip to Section 4.

First we observe that the problem of determining if a set of clauses has an ordering of line channelwidth at most  $k$  is NP-complete, even if each variable only appears in two clauses. The problem **PATHWIDTH** (is the pathwidth of a graph  $k$  or less) is easily reduced to this line channelwidth problem. Map nodes to clauses, edges to variables, and let the variables in a clause be the variables corresponding to edges with end points at the corresponding node. **PATHWIDTH** is known to be NP-complete [12]. However, the time required to find an ordering (if any) of line channelwidth  $k$  is at worst exponential only in  $k$ . This bound is achieved by a recursive algorithm

which begins “consider all variable subsets of size  $k$  and see if the assignments to this set break the problem into independent subproblems of approximately equal size.”

The above implies that an ordering of line channelwidth  $p(n)$  (if one exists) can be found in time which has power index  $a$ , the same power index used to test satisfiability after the ordering is given. However,  $\mathbf{LC}(p)$  as defined in Definition 3.2 belongs to  $\mathbf{NQ}n^a$  (see Corollary 5.6) whereas this may not be true under the alternative definition (we lack an appropriate sub-linear space algorithm for finding an ordering.)

## 4 Nondeterministic Straight-line Programs

The concept of a straight-line program has been used in the literature in a number of contexts. See for example [1]. A straight-line program (SLP) is a sequence of statements or operations which take the values from certain specified memory variables and assign new values to certain specified memory variables. The semantics of a deterministic operation are thus described as a function from input values to output values. For any initial assignment to the memory variables, the operations cause a sequence of changes resulting in a final assignment.

A nondeterministic SLP is similar except that an operation may produce any one of several alternative assignments. The semantics of a nondeterministic operation can be described by a relation on input and output values. This relation can be characterized by a Boolean-valued function on the input and output values which is TRUE if and only if the combination of input and output values are a tuple in the relation described by the function. For any initial assignment to memory variables, the nondeterministic operations may permit many sequences of changes thereby producing many alternative final assignments. For some assignments, there may be no legal sequence of changes and hence the set of alternative final assignments might be empty. Deciding if a legal sequence exists will be called the “executability problem.”

We now formally define the nondeterministic case.

**Definition 4.1** *A nondeterministic operator is a pair  $(f, \tau)$  where, for some  $k$ ,*

1.  $f$  is a Boolean valued  $k$ -ary function

$$f : D_1 \times \dots \times D_k \rightarrow \{\text{TRUE}, \text{FALSE}\}$$

where each domain  $D_i$  is finite;

2.  $\tau$  is a mapping of  $\{1, \dots, k\}$  into  $\{\text{INPUT}, \text{OUTPUT}\}$  which partitions the parameters into **input parameters and output parameters**.

A **program operation**  $\Theta$  based on operator  $(f, \tau)$  is the application of  $f$  to an argument list  $a_1 \dots a_k$  where

1. the type of each argument  $a_i$  is the corresponding domain  $D_i$ ,
2. each argument matched to an input parameter is a variable or constant,
3. each argument matched to an output parameter is a variable, and
4. no two output parameters are matched to the same variable.

A **nondeterministic straight-line program** (or *NSLP*) is a finite sequence of program operations. The **length** of the program  $P$ , denoted by  $|P|$ , is the sum of the number of parameters in each operation.  $\text{VAR}(P)$  is the set of variables in  $P$ .

Given a program operation  $\Theta$  and two assignments  $\gamma_1$  and  $\gamma_2$  on a set of variables  $V$  containing  $\text{VAR}(\Theta)$ , we say  $\Theta$  **changes**  $\gamma_1$  to  $\gamma_2$  and write  $\gamma_1 \Theta \gamma_2$  if and only if

1.  $\gamma_1(x) = \gamma_2(x)$  for  $x$  which are not matched to output arguments of  $\Theta_i$ , and
2. the operator function for  $\Theta$  gives the value **TRUE** when the input arguments matched to variables have the values given by  $\gamma_1$  and the output arguments have values given by  $\gamma_2$ .

Given a *NSLP*  $P = \Theta_1 \dots \Theta_k$ , a set  $V$  of variables containing  $\text{VAR}(P)$ , and two assignments  $\gamma_0$  and  $\gamma_k$ , we say that  $P$  **changes**  $\gamma_0$  to  $\gamma_k$  and write  $\gamma_0 P \gamma_k$  if and only if there exists a sequence of assignments  $\gamma_0 \dots \gamma_k$  to  $V$  such that, for all  $i$ ,  $1 \leq i \leq k$ ,  $\gamma_{i-1} \Theta_i \gamma_i$ . We say that this sequence of assignments **satisfies**  $P$ . A *NSLP*  $P$  is called **executable** if and only if there is a sequence of assignments which satisfies  $P$ .

The decision problem associated with an NSLP is deciding if there is an initial assignment to the variables for which the program executable.

**Definition 4.2** *Let  $S$  be a set of nondeterministic operators. Then  $\mathbf{NSLP-EXE}(S)$  denotes the set of executable NSLPs constructed from operators in  $S$ .*

We have a special interest in  $\mathbf{NSLP-EXE}(S)$  for the case where  $S$  is a certain set of “basic Boolean program operations”, a case we call “ $\mathbf{BNSLP-EXE}$ ”.

**Definition 4.3** *The nondeterministic operators defined below, namely AND, OR, NOT, COPY, ON, OFF, and SET will be called the **basic Boolean program operators**. Letting  $x$  and  $y$  be Boolean input parameters and  $z$  a Boolean output parameter, define  $\mathbf{AND}(x, y, z) \leftrightarrow z = x \wedge y$ ,  $\mathbf{OR}(x, y, z) \leftrightarrow z = x \vee y$ ,  $\mathbf{NOT}(x, z) \leftrightarrow z = \bar{x}$ ,  $\mathbf{COPY}(x, z) \leftrightarrow z = x$ ,  $\mathbf{ON}(x) \leftrightarrow x$ ,  $\mathbf{OFF}(x) \leftrightarrow \bar{x}$ , and  $\mathbf{SET}(z) \leftrightarrow \mathbf{TRUE}$ . A NSLP constructed from these operations will be called a *BNSLP*. Let  $\mathbf{BNSLP-EXE}$  denote  $\mathbf{NSLP-EXE}(B)$  where  $B$  is the set of basic boolean program operators.*

The program operators AND, OR, NOT, and COPY represent the familiar programming language assignment statements  $z \leftarrow x \wedge y$ ,  $z \leftarrow x \vee y$ ,  $z \leftarrow \neg x$  and  $z \leftarrow x$ . These operators are deterministic in that, for any  $\Theta$  based on these operators and any assignment  $\gamma_1$ , there is exactly one  $\gamma_2$  such that  $\gamma_1 \Theta \gamma_2$ , namely the assignment obtained by performing the corresponding assignment statement.

The operators ON and OFF have the property that, for any  $\Theta$  based on these operators and for any  $\gamma_1$ , there is at most one  $\gamma_2$  (namely  $\gamma_2 = \gamma_1$ ) such that  $\gamma_1 \Theta \gamma_2$ . Specifically,  $\gamma \mathbf{ON}(x) \gamma$  if and only if  $\gamma$  assigns  $x$  the value  $\mathbf{TRUE}$  and  $\gamma \mathbf{OFF}(x) \gamma$  if and only if  $\gamma$  assigns  $x$  the value  $\mathbf{FALSE}$ . In effect, ON and OFF perform tests which, if they fail, block continued execution of the program. They thereby endow the program with some decision making capability without introducing branching. (Think of  $\mathbf{ON}(x)$  as “if  $x$  then continue else halt.”)

The operator SET nondeterministically assigns a new value to a variable. It is the only nondeterministic operator. Without it, a program will either change an initial assignment uniquely into some final assignment or will get blocked by some ON or OFF operator.

Intuitively, the next lemma says that BNSLPs can test the satisfiability of CNF formulas. Specifically, the lemma implies there is an efficient linear sized reduction from **SAT** to **BNSLP-EXE**.

**Lemma 4.4** *If  $F$  is a CNF formula,  $V$  is a set of variables containing  $\text{VAR}(F)$ , and  $t$  is a (temporary) variable not in  $V$ , then there is a program  $P$  in BNSLP with variable set  $V' = V \cup \{t\}$  such that, for any assignment  $\gamma \in \Gamma(V')$ ,*

1. *if  $\gamma$  satisfies  $F$ , then there is exactly one  $\gamma'$  such that  $\gamma P \gamma'$ . Furthermore,  $\gamma$  and  $\gamma'$  are identical for all variables in  $V$ .*
2. *if  $\gamma$  does not satisfy  $F$ , then there is no assignment  $\gamma'$  such that  $\gamma P \gamma'$ .*

**Proof.** It is sufficient to show the lemma is true for a single clause  $c$  because then a program for  $F$  can be constructed by concatenating the programs for each of its clauses. If the first literal in  $c$  is an unnegated literal  $x$ , begin the program with  $\text{COPY}(x, t)$ . If instead the first literal is  $\bar{x}$ , begin the program with  $\text{NOT}(x, t)$ . Then for each subsequent unnegated literal  $y$ , append  $\text{OR}(y, t, t)$ . For each subsequent  $\bar{y}$ , append  $\text{NOT}(y, y)\text{OR}(y, t, t)\text{NOT}(y, y)$ . Finally append the operation  $\text{ON}(t)$ .

We claim that the program does not permanently change the value of any variable in  $V$ . To see this, observe that, with one exception, none of the operations change a variable of  $V$ . In the one exception, variable  $y$  gets negated by a  $\text{NOT}(y, y)$ , but the change is undone two operations later by another  $\text{NOT}(y, y)$ . Thus the net result of all the operations is no change.

The effect of the operations prior to the final  $\text{ON}$  operation is to evaluate the clause  $c$  and put the result in variable  $t$ . If  $t$  has been set **TRUE**, the  $\text{ON}$  operation leaves the assignment unchanged as required by Condition 1. If  $t$  has been set **FALSE**, the  $\text{ON}$  operation prevents any valid next assignment and the requirement of Condition 2 is met. ■

To illustrate the proof construction, the program for clause  $(\bar{x} \vee y \vee \bar{z})$  is

$$\text{NOT}(x, t)\text{OR}(y, t, t)\text{NOT}(z, z)\text{OR}(z, t, t)\text{NOT}(z, z)\text{ON}(t)$$

**Lemma 4.5** *Let  $S$  be any finite set of nondeterministic operators. There is a PQ reduction  $R$  from **NSLP-EXE**( $S$ ) to **BNSLP-EXE** and a constant  $c$  such that  $|\text{VAR}(R(P))| \leq c|\text{VAR}(P)|$  for all programs  $P$ .*

**Proof.** Because  $S$  is finite, there is only a finite set of domains available as variable types. Therefore there is a constant  $b$  such that domain values can be mapped one-to-one into vectors of  $b$  Boolean values. Under this mapping, each of the operators  $(f, \tau)$  in  $S$  can be interpreted as an operator  $(f', \tau')$  on Boolean domains where each original domain is replaced by  $b$  Boolean domains. Specifically,  $f'$  is the function which has value `TRUE` if and only if each group of  $b$  arguments represents a value in the original domains and  $f$  maps these corresponding values to `TRUE`. Function  $\tau'$  says the variables in vectors representing input variables are input variables and the variables in vectors representing output variables are output variables. By replacing each program variable by a vector of Boolean formulas, a program constructed from operators in  $S$  can be converted to a program constructed from the corresponding operators with Boolean domains. The corresponding programs are either both executable or both unexecutable and the new program is  $b$  times the size of the original. (The number of program operations is the same but the number of arguments is multiplied by  $b$ .) It is therefore sufficient to prove the result for operators with Boolean domains.

Assuming now that all variables and domains are Boolean, we must show that a program  $P$  based on some operator set  $S$  can be changed to an equivalent program using the basic Boolean operations of Definition 4.3. The constructed program will have one variable for each variable in the original program, a temporary variable  $t$  to be used as in the proof of Lemma 4.4, and a set of temporary variables associated with output variables of operations in  $P$ .

It is sufficient to find an equivalent subprogram for one operation  $\Theta$  since the subprograms for individual operations in program  $P$  can be concatenated to get a program corresponding to  $P$ . Let  $(f, \tau)$  be the operation upon which operator  $\Theta$  is based. We will associate a temporary variable  $y_i$  with each output variable  $y_i$  of  $\Theta$  and a temporary variable  $t$  to be used as in Lemma 4.4. The first steps in the subprogram initialize the temporary variables  $y'_i$  nondeterministically to be either `TRUE` or `FALSE`. This assignment to the  $y'_i$  can be interpreted as proposed values for the output variables. Next the program confirms that the program variables used as input and the proposed outputs together satisfy  $f$ . Finally the  $y'_i$  are copied to the corresponding  $y_i$ . The initialization is carried

out with  $\text{SET}(y'_i)$  operations, the confirmation is carried out by a program obtained from Lemma 4.4 applied to the CNF representation of  $f$ , and the copy is achieved by  $\text{COPY}(y'_i, y_i)$  operations. It is clear the subprogram has the same effect as  $\Theta$  and hence the concatenation of subprograms is equivalent to  $P$ .

Now consider the size of the reduction. The operation  $\Theta$  is replaced by a string of Boolean operations whose size depends on the underlying operator  $(f, \tau)$  from  $S$ . However, since  $S$  is finite, the sizes of the replacement strings are no more than a constant times the sizes of the operators they replace. It follows that the reduction is PQ.

Finally, we need to prove the assertion about variable sets. The transformation to a Boolean-valued program replaces each variable by a vector of at most  $b$  variables and therefore multiplies the number of variables by  $b$  or less. The step from Boolean programs to a program in basic Boolean operators results in the addition of a temporary variable  $t$  and temporary variables to put in correspondence with operator outputs. Because the association between outputs and temporaries can be re-assigned after each operation in  $P$ , the number of such temporaries needed is just to the maximum number of of output variables belonging to some operation in  $P$ . The combined effect of the two transformations is therefore a constant multiple of the number of original variables. ■

Notice that the finiteness of  $S$  implies that a constant bound can be placed on the number of outputs of any operation of any program constructed from  $S$ . Therefore the transformation from Boolean operations to basic Boolean operations, as given in the above proof, can be achieved while changing the number of variables by an additive constant rather than by a multiplicative constant.

Also note that the construction in the above proof creates more nondeterminism than necessary. For example, in the case of deterministic operations, the basic Boolean operations can be used to simulate a circuit which computes the appropriate outputs deterministically.

We now define NSLP executability problems where the domains are obtained by restricting the number of variables (i.e. amount of memory) as a function of program length.

**Definition 4.6** *For any function  $p : N \rightarrow N$  and any set  $S$  of nondeterministic operators, let  $\text{NSLP-EXE}(S, p)$  be the language problem  $(\text{NSLP-EXE}(S), D)$  where domain  $D$  is the set of*

*NSLPs  $P$  constructed from operators in  $S$  such that the number of variables in  $P$  is no more than  $p(|P|)$ . For all  $a$ ,  $0 \leq a \leq 1$ , let  $\text{NSLP}n^a\text{-EXE}(S)$  be the set of language problems  $\text{NSLP-EXE}(S, p)$  such that  $p$  has polynomial index  $a$ . Let  $\text{BNSLP}n^a\text{-EXE}$  be  $\text{NSLP}n^a\text{-EXE}(B)$  where  $B$  is the set of basic Boolean operators.*

**Theorem 4.7** *For all finite sets  $S$  of nondeterministic operations, there exists a PQ reduction  $R$  from  $\text{NSLP-EXE}(S)$  to  $\text{BNSLP-EXE}$  such that, for all  $a$ ,  $0 \leq a \leq 1$ ,  $R$  reduces every language problem in  $\text{NSLP}n^a\text{-EXE}(S)$  to a language problem in  $\text{BNSLP}n^a\text{-EXE}$ .*

**Proof.** This follows at once from Lemma 4.5 and the fact that a linear change to a function of polynomial index  $a$  results in another function of polynomial index  $a$ . ■

## 5 The PQ Reductions

In this section we show that the classes  $\text{NQ}n^a$ ,  $\text{BNSLP}n^a\text{-EXE}$ ,  $\text{LC}n^a\text{-SAT}$ , and  $\text{LC}n^a\text{-3SAT}$  are all PQ inter-reducible and hence have the same power index. For certain finite sets  $S$  of nondeterministic operators, PQ-reductions from  $\text{NQ}n^a$  to  $\text{NSLP}n^a\text{-EXE}(S)$  will also be used. We have already given a PQ-reduction of  $\text{NSLP}n^a\text{-EXE}(S)$  to  $\text{BNSLP}n^a\text{-EXE}$  for all finite  $S$  in Theorem 4.7 and a reduction of  $\text{LC}n^a\text{-SAT}$  to  $\text{LC}n^a\text{-3SAT}$  in Proposition 3.4. A PQ-reduction from  $\text{LC}n^a\text{-3SAT}$  to  $\text{LC}n^a\text{-SAT}$  is achieved by the identity function. Since PQ reductions compose (Proposition 1.6), it is only necessary to supply a selected subset of the needed reductions and the remaining relationships are implied by the transitivity of PQ-reductions.

**Theorem 5.1** *For all  $a$ ,  $0 \leq a \leq 1$ , every language problem in  $\text{NQ}n^a$  is PQ-reducible to a language problem in  $\text{NSLP}n^a\text{-EXE}(S)$  for some  $S$  of size two.*

**Proof.** Let  $\mathcal{L}$  be a language problem in  $\text{NQ}n^a$ . By Definition 2.2,  $\mathcal{L}$  is PQ-reducible to a language problem  $\mathcal{L}'$  that can be recognized by a choice-oblivious Turing machine  $\mathcal{T}$  in  $q$ -linear time and  $p(n)$ -space for some function  $p$  with polynomial index  $a$ . It will be sufficient to find a PQ-reduction from  $\mathcal{L}'$  to some language problem in  $\text{NSLP}n^a\text{-EXE}(S)$ .

The Turing machine  $\mathcal{T}$  has some fixed number of scratch tapes, say  $k$ , plus a read only input tape. For  $1 \leq i \leq k$ , let  $x_i$  and  $y_i$  be parameters whose type is the tape alphabet of the  $i$ -th tape of  $\mathcal{T}$ . Let  $s$  and  $t$  be parameters whose type is the state set of  $\mathcal{T}$  and let  $z$  be a parameter taking any value from the input tape. Define  $s, x_1, \dots, x_k, z$  to be input parameters and  $t, y_1, \dots, y_k$  to be output parameters. Define  $\text{MOVE}(s, x_1, \dots, x_k, z, t, y_1, \dots, y_k)$  to be **TRUE** if and only if the transition relation for  $\mathcal{T}$  allows a transition which, from state  $s$  with  $x_1, \dots, x_k$  under the scratch tape heads and input  $z$ , changes the state to  $t$  and writes  $y_1, \dots, y_k$  on the corresponding scratch tapes. Define  $\text{ACCEPT}(s, x_1, \dots, x_k, z)$  to be **TRUE** if and only if  $\mathcal{T}$  stops and accepts in state  $s$  while reading  $x_1, \dots, x_k, z$ . Let  $S = \{\text{MOVE}, \text{ACCEPT}\}$ . Note that nothing in the specifications of **MOVE** or **ACCEPT** says where the tape heads are located or where (in the case of **MOVE**) they are moved to. It will be the responsibility of the programmer or algorithm creating a program to indicate these head positions through the use of appropriate variables.

We now describe a PQ-algorithm which takes a description of choice-oblivious nondeterministic Turing machine  $\mathcal{T}$  and an input string  $w$  for  $\mathcal{T}$  and produces a NSLP which is satisfiable if and only if  $\mathcal{T}$  accepts  $w$ . Because  $\mathcal{T}$  is choice-oblivious, the location of its tape heads as it processes  $w$  is a function of the number of steps taken and is independent of which nondeterministic choices are taken. This means these tape locations can be computed deterministically by simulating one non-deterministic path of  $\mathcal{T}$  using any one of the nondeterministic choices available at each step.

The NSLP is to have one variable  $s$  which stores a state and one variable for each scratch tape square visited during the simulation. Since the symbols on the read-only input tape are unchanged, the NSLP does not need variables to remember the contents of input tape squares. If the NSLP needs the input symbol from some square of the input tape, the reduction program figures out what the value is and inserts it into the NSLP as a constant. The NSLP cannot afford to have variables for input tape squares because it would then have  $O(|w|)$  variables and the number of variables would not be sub-linear.

The NSLP has one **MOVE** operation for each move taken by  $\mathcal{T}$  and then one **ACCEPT** operation. For each move, the corresponding operation is  $\text{MOVE}(s, v_1, \dots, v_k, c, s, v_1, \dots, v_k)$  where  $v_1 \dots v_k$

are the names of the variables corresponding to squares under the heads at the beginning of the move and  $c$  is a constant representing the value read from the input tape. For the first move, the first parameter is the constant representing the start state rather than the variable  $s$ . The final operation is  $\text{ACCEPT}(s, v_1, \dots, v_k, c)$  where  $v_1, \dots, v_k$  are the names of the tape squares seen at the last configuration and  $c$  is a constant representing the last symbol seen from the input tape. Because  $\mathcal{T}$  is choice-oblivious, the resulting program will be the same regardless of which run is simulated.

Because the  $\text{MOVE}$  operation is successful if and only if its input and output values describe a legal move, successful runs correspond exactly to satisfying sequences of assignments. The NSLP has one statement per configuration and since the time taken by  $\mathcal{T}$  is  $q$ -linear in the input size, the program has size  $q$ -linear in the input size. The program can be constructed in polynomial time since the reduction procedure need only simulate one nondeterministic run and map each move into a  $\text{MOVE}$  operation. Thus the construction of the NSLP is PQ and the output formula is satisfiable if and only if the input sequence is accepted by  $\mathcal{T}$ .

It remains to be shown that the number  $v$  of variables in the constructed program  $P$  always satisfies  $v \leq p(|P|)$  for some function  $p$  with polynomial index  $a$ . Technically speaking, this is false because  $\mathcal{T}$  might terminate on some input  $w$  without reading all inputs and hence the program  $P$  could be significantly shorter than  $w$ . To cover this case, we allow the transformation to repeat the accept operation as many times as necessary in order to bring the length of the program up to the length of the input. Assuming the program is as long as the input, the existence of  $p$  follows from the fact that the number of variables is only one more than the number of scratch tape squares read, and this number is bounded by a polynomial index  $a$  function of the input size. ■

Notice the techniques of the above proof. The reduction maps each problem instance to a custom-made program. The custom-made programs imitate the Turing machines, but the programs have no need for input. Instead, the reduction puts symbols from the Turing machine input directly into the program as constants. This is why the memory requirements of the program reflect only the non-input memory of the Turing machines.

An alternative to inserting constants into an NSLP is to insert variables instead, variables which are initialized prior to the execution of the nondeterministic program and which are not changed during the nondeterministic computation. In the case of the above proof, these unchanging variables would be those containing the input. Other programs might involve additional variables to be initialized with non-input values and to be held constant during the nondeterministic processing. Under this approach, programs  $P$  in  $\text{NSLP}n^a\text{-EXE}(S, p)$  are programs which *change* (rather than *have*) at most  $p(|P|)$  variables.

**Theorem 5.2** *There exists a PQ-reduction  $R$  from **BNSLP-EXE** to **SAT** such that, for all  $a$ ,  $0 \leq a \leq 1$ ,  $R$  reduces every problem in  $\text{BNSLP}n^a\text{-EXE}$  to a problem in  $\text{LC}n^a\text{-3SAT}$ .*

**Proof.** First we present the method used by  $R$  for mapping programs into formulas such that the resulting formula is satisfiable if and only if the program is executable. The method itself can be viewed as a two step process, a step which modifies the program and a step which converts the modified program to a CNF formula.

The objective of modifying the program is to obtain an equivalent program where no variable is assigned a new value after it is first used. Intuitively, this is achieved by renaming a variable each time it is used as the output of a program operation.

Suppose the original program is  $P = \Theta_1 \dots \Theta_k$ . For each occurrence of a variable  $x$  as an output parameter of an operation  $\Theta_i$ , construct a new variable  $z$  and replace the occurrence of  $x$  with  $z$ . Also replace input occurrences of  $x$  with  $z$  in all  $\Theta_j$  such that  $i < j$  and  $x$  doesn't occur as output in any  $\Theta_\ell$  for  $i < \ell < j$ . The result is a program  $P'$  in which each variable is assigned a value before that variable is ever used as input and that variable is never reassigned another value. This property of variables enables us to regard  $P'$  as an ordered formula  $F$ , the conjunction of "terms" constructed from the operator functions (without any distinction between input and output). Satisfying assignments for  $F$  and satisfying assignment sequences for  $P$  and  $P'$  can easily be obtained from one another since variables in  $F$  and  $P'$  correspond to values held by variables in  $P$  during the period where the value is unchanged.

Next we take formula  $F$  and convert it into an equivalent ordered CNF formula. To do this, replace each term by an equivalent CNF formula with the clauses ordered arbitrarily. Because the basic Boolean operators have at most three arguments, the clauses have at most three literals. The constructed 3CNF formula is satisfiable if and only if the program  $P$  is executable.

The whole construction consists of making simple variable replacements and replacing operators with their Boolean equivalents. Therefore the construction can be performed in polynomial time, has q-linear size, and the input is executable if and only if the output is satisfiable. All that remains is to establish a q-linear relation between the number of variables in the input program  $P$  and the line channelwidth of the output formula.

Let  $c$  be a clause associated with operator  $\Theta$  in  $P'$ . At  $c$ , there are possibly two channel variables associated with each variable  $x$  in  $P$ . One is the variable corresponding to  $x$  in  $P'$  that is available for input to  $\Theta$ . In the case that  $\Theta$  outputs a value to a variable corresponding to  $x$ , that variable is also a channel variable at  $c$ . Other variables corresponding to  $x$  occur only to the left of  $c$  (if the variable was changed before  $\Theta$ ) or to the right (if the variable was initialized after  $\Theta$ ). The line channelwidth of the formula is therefore at most twice the number of original variables and the construction gives the required PQ-reduction. ■

The next result is a stronger version of Lemma 4.4.

**Theorem 5.3** *There exists a PQ-reduction  $R$  from SAT to BNSLP-EXE such that, for all  $a$ ,  $0 \leq a \leq 1$ ,  $R$  reduces every problem in  $\text{LCn}^a\text{-SAT}$  to a problem in  $\text{BNSLPn}^a\text{-EXE}$ .*

**Proof.** Suppose we are given an ordered CNF formula  $F$  of line channelwidth  $k$ . The variables of  $F$  can be partitioned into  $k$  sets such that no two variables in a block appear in the same channel variable set. Such a partition can be found by scanning the clauses in order and, upon finding a variable  $x$  for the first time, add it to a block whose variables have all made their final appearances in earlier clauses. Such a block must exist for otherwise the clause would have one channel variable from each block plus channel variable  $x$ .

To change  $F$  into a program, replace each clause by the corresponding program given in the proof of Lemma 4.4 and, for each formula variable  $x$ , insert  $\text{SET}(x)$  just prior to the first operation

where variable  $x$  occurs. Then pick a representative from each partition block and replace each occurrence of a variable in the program with the representative of the block to which the variable belongs. The result is then a BNSLP  $P$ .

The program  $P$  is essentially a program which guesses an assignment for the variables in  $F$  and tests to see if the assignment satisfies  $F$ . Instead of guessing the complete assignment at the beginning of the program, it defers guessing a value until the time when the value is first needed. This allows the program to save memory by reusing a location whose value is no longer needed rather than simply allocating one location per variable. We take it to be obvious that  $P$  is satisfiable if and only if  $F$  is satisfiable, that the construction can be performed in polynomial time, and that the size is q-linear.

The program  $P$  uses just  $k + 1$  variables; one for each block of the partition and the temporary  $t$  needed for Lemma 4.4. Thus if the line channelwidth of  $F$  is bounded by a polynomial index  $a$  function of  $|F|$ , the number of variables in  $P$  is bounded by a polynomial index  $a$  function of  $|P|$ . ■

**Theorem 5.4** *For all  $a$ ,  $0 \leq a \leq 1$ , every problem in  $\text{BNSLP}n^a\text{-EXE}$  is  $PQ$ -reducible to a problem in  $\text{NQ}n^a$ .*

**Proof.** We will describe a transformation which takes a SLP constructed from basic Boolean program operators and produces a string  $w$  of input symbols for a particular nondeterministic choice-oblivious Turing machine  $\mathcal{T}$ . The Turing machine will be “real-time” in that it reads one input every move from a one-way read-only input tape. In order for the transformation to be  $PQ$ , the length of string  $w$  must be q-linear in the length of the program. We begin with a description of  $\mathcal{T}$ .

Turing machine  $\mathcal{T}$  has an input tape with a large input alphabet (to be described later) and two scratch tapes with tape alphabet  $\{0, 1\}$ . The state set of  $\mathcal{T}$  remembers the values of four binary variables  $x, y, z$ , and  $r$ . The machine accepts if and only if the value of  $r$  is 1 when the machine stops. The start state has  $r$  equal to 1 and  $r$  gets changed permanently to 0 if any one of many tests fail. The input sequence will cause one test for each operation in the corresponding NSLP.

The input symbols are in one to one correspondence with certain primitive Turing machine actions and reading an input causes the corresponding action to take place. The actions are classified into five categories: copies, shifts, tests, sets, and halts. The copy actions cause the value read from one scratch tape to be copied to the other scratch tape or to one of the state variables  $x$ ,  $y$ , or  $z$ . (Thus there are eight distinct copy actions.) The shift actions are move tape 1 left, move tape 1 right, move tape 2 left, and move tape 2 right.

There is one test action on the state variables for each basic boolean program operator listed in Definition 4.3. These actions perform the test specified in Definition 4.3 and set  $r$  to 0 if the test fails. For example, the “and”-test will set  $r$  to 0 if the test  $z = x \wedge y$  fails.

There is an action which nondeterministically sets the square under the head of tape 1 to be some value from  $\{0, 1\}$  and a similar action for tape 2. These are the only nondeterministic actions. There are also deterministic actions which set  $x$  or  $y$  to either 0 or 1.

There is one halt action. This action is placed at the end of the input to terminate the computation.

Because each input symbol of  $\mathcal{T}$  causes some primitive action, input sequences can be thought of as machine language programs. Thus given an SLP, our objective is to find a machine language program to test the satisfiability of the SLP. The overall plan is as follows: Allocate a square on tape 1 for each variable in the SLP and nondeterministically initialize these locations with a set action. Then consider each successive operation from the SLP. As described below, get the values from the tape into the  $x$  and  $y$  registers as called for by the operation. If the operation has output, give the corresponding variable a new nondeterministic value (i.e. guess the output) and put that value into the  $z$  variable. Then apply the corresponding test operation to verify that the values are allowed by operator function (and use variable  $r$  to remember a failure). When all tests are completed, halt.

The plan just given is incomplete in that we have not specified exactly what sequence of steps will be used to move a value from memory into one of the state variables. There is an obvious way to do this, namely move the head to the corresponding tape square and copy the value. This

method is unsatisfactory since it could require moving the head across all the tape squares for each access and consequently violate the requirement that the Turing machine program be  $q$ -linear in the size of the SLP. Instead, we will adopt a more sophisticated strategy.

Suppose the SLP has  $k$  variables. We can call them  $v_1 \dots v_k$  where the index indicates the square of tape 1 chosen to store the variable's value.

Let  $b_1, \dots, b_n$  be the sequence of bits we want to put into state memory to carry out the outlined plan. This sequence begins with the input values for the first operator followed by the guessed output value and so forth. By definition,  $n$  is the size of the SLP. We think of the bit sequence as partitioned into blocks of  $k$  consecutive bits. For each block in succession, we plan to get the required value instances onto tape 2, then sort the instances into the required order, and then perform the tests which apply to the subsequence.

To get the values onto tape 2, visit square 1 of tape 1 and (using the replace instruction) put as many copies of  $v_1$  onto tape 2 as there are occurrences of  $v_1$  in the given block. For those occurrences which are used as output variables, apply the nondeterministic set action to  $v_1$  prior to performing the replace. Repeat this for variable  $v_2$ , and so forth. The total number of actions is  $O(k)$ . Note that it is the reduction program constructing the input for  $\mathcal{T}$  that figures out the sequence of copy, shift and set actions needed to accomplish this task. Machine  $\mathcal{T}$  just executes this precomputed action sequence.

The next step is to get the instances sorted into the order that they occur in  $b_1, \dots, b_n$ . By employing both tapes, this sorting can be accomplished by performing  $O(k \log k)$  copy and shift actions suggested by merge-sort (or any other efficient two-tape sort method). The sequence of copy and shift operations needed is precomputed by the reduction program which can figure out the appropriate position for each value in the bit sequence and which can perform the comparisons on these positions to determine the proper sequence of copies and shifts. The machine  $\mathcal{T}$  merely performs the precomputed sequence of copy and shift actions.

Once the bits have been sorted into the required order, it is a simple  $O(k)$  matter of putting the bits into appropriate state memory locations and performing the required tests. Note that it is

the reduction program which understands which locations and tests are appropriate and puts the corresponding action symbols on the input tape of  $\mathcal{T}$ . The movement of heads on the tapes of  $\mathcal{T}$  is dictated by the input tape symbols and requires no testing. Thus  $\mathcal{T}$  makes the same motions regardless of the outcomes of the tests and hence  $\mathcal{T}$  is choice-oblivious.

The procedure is a reduction in that the output is accepted by  $\mathcal{T}$  if and only if the input SLP is executable. We need to verify all the resource conditions. Each of the  $n/k$  blocks of the bit sequence gets associated with  $O(k \log k)$  actions so the length of the reduction output is  $O(n \log k)$  which is q-linear because  $k \leq n$ . The reduction can obviously be performed in polynomial time and thus the reduction is PQ. The amount of memory used by  $\mathcal{T}$  is  $O(k)$  and so an input SLP with the number of variables limited by a polynomial index  $a$  function will be transformed into a language accepted by a nondeterministic  $\mathcal{T}$  using space bounded by a polynomial index  $a$  function. Since  $\mathcal{T}$  is choice-oblivious, the result is proven. ■

**Corollary 5.5** *For all  $a$ ,  $0 \leq a \leq 1$ , the classes  $\text{NQN}^a$ ,  $\text{BNSLP}n^a\text{-EXE}$ ,  $\text{LC}n^a\text{-SAT}$ , and  $\text{LC}n^a\text{-3SAT}$  are all PQ-reducible to one another and have the same power-index.*

**Proof.** All the necessary PQ-reductions have been provided. ■

**Corollary 5.6** *For all  $a$ ,  $0 \leq a \leq 1$ , the classes  $\text{BNSLP}n^a\text{-EXE}$ ,  $\text{LC}n^a\text{-SAT}$ ,  $\text{LC}n^a\text{-3SAT}$ , and  $\text{NSLP}n^a\text{-EXE}(S)$  for finite  $S$  are subsets of  $\text{NQN}^a$ .*

**Proof.** Immediate from Proposition 1.5, Corollary 5.5, and Theorem 5.1. ■

**Corollary 5.7** *For all  $a$ ,  $0 \leq a \leq 1$ , the classes  $\text{BNSLP}n^a\text{-EXE}$ ,  $\text{LC}n^a\text{-SAT}$ , and  $\text{LC}n^a\text{-3SAT}$  are all  $\text{NQN}^a$ -complete.*

**Proof.** Immediate from corollaries 5.5 and 5.6 ■

## 6 Oblivious Retrieval

We are interested in methods of showing that a specific language problem belongs to some specific  $\text{NQN}^a$  (eg. that **CLIQUE** belongs to  $\text{NQN}^{1/2}$ ). One such technique is to show how the language

problem in question can be solved (after preprocessing) with NSLPs. Writing straight line programs would be much easier if arrays, pointer variables, and other random access instructions were available. These useful programming constraints and others can be imitated if we can nondeterministically and obliviously maintain a data structure we call a “nondeterministic database”.

A nondeterministic database is a data structure which, for some  $k$  and  $\ell$ , contains  $2^k$  records where each record is a binary vector of length  $\ell$ . The data structure has an “access” operation which nondeterministically returns any of the  $2^k$  records and allows that record to be overwritten. The data structure must be implemented in such a way that the next record to be accessed is always found in the same vector of  $\ell$  variables, and a NSLP can thus read the data from and write data to the fixed location of the vector.

To imitate a RAM with this data structure, use the records to keep a  $k$ -bit address field and a value field. Follow each access with a program segment which verifies that the value in the address field matches the value in the instruction counter. Under the assumption that the value field is about the same size as the address field, the number of bits involved in the access is about twice the log of the memory size. This is a natural assumption because most computers have a word size about equal to its address size.

To support the nondeterministic database, we employ  $2^k$  vectors of variables and we name the vectors  $v_0$  to  $v_{2^k-1}$ . Each vector consists of  $\ell$  binary variables and is used to store one of the  $2^k$  records. The current record to be accessed is always found in vector  $v_0$ , a fixed location, thus making the record available obliviously. Between accesses, a sequence of operations is performed on the vectors so that some subset of the records is nondeterministically permuted among the vector locations they occupy. We will need to show that the set of permutations available to the nondeterminism is rich enough so that, with suitable guesses, any sequence of records can be made to appear in  $v_0$  for accessing. Sometimes the number of operations used for this permutation is  $\Theta(\ell k 2^k)$  and sometime only  $\Theta(\ell)$ , but the average number is  $\Theta(\ell k^2)$ . The amount of extra memory used to perform the permutations is just one bit.

We think of the vectors as organized into nested zones  $Z_0$  to  $Z_k$  where  $Z_i$  consists of the the

vectors  $v_0$  to  $v_{2^i-1}$ . Zone  $Z_0$  is thus the one vector  $z_0$  and  $Z_k$  is the set of all  $2^k$  vectors. Each  $Z_i$  is twice as big as  $Z_{i-1}$  and contains all the smaller zones.

For  $n \geq 0$ , define  $\rho(n)$  by

1.  $\rho(n) = i$  if  $n = a2^{i-1}$  for some odd integer  $a$  and  $i \leq k$ ,
2.  $\rho(n) = k$  if  $n = 0 \pmod{2^k}$ .

The **plan for maintaining the data structure** is the following:

After  $n$  accesses, in preparation for access  $n + 1$ , the records in zone  $Z_{\rho(n)}$  are non-deterministically permuted. For all  $j$ ,  $0 \leq j < \rho(n)$ , the permutation puts the next  $2^j$  records to be accessed into zone  $Z_j$ .

There are many permutations on zone  $Z_{\rho(n)}$  which can be used to satisfy the plan. One of them is the permutation which puts the records in the order that they are to be accessed. The construction in our proof allows this and any other permutation on  $Z_{\rho(n)}$  to be used. As discussed later in this section, limiting the construction to certain smaller permutation sets is also satisfactory.

To prove that the plan works, it must be shown that, for any access sequence  $r_1 \dots r_m$ , there is a sequence of permutations permitted by the plan which causes each  $r_i$  to be in vector  $z_0$  when the  $i$ -th access is performed.

**Definition 6.1** *A sequence of permutations  $\pi_0 \dots \pi_{n-1}$  is **suitable** for accessing a sequence  $r_1 \dots r_n$  of records if and only if, for  $0 \leq i < n$  and  $0 \leq j < \rho(i)$ , permutation  $\pi_i$  puts the information for the next  $2^j$  accesses into zone  $Z_j$  (or for all remaining accesses if  $i + 2^j - 1 > n$ ).*

Notice that zone  $Z_j$ , being of size  $2^j$ , is sufficiently large to contain the records for the next  $2^j$  accesses. Since some of the accesses may involve the same record, the zone  $Z_j$  could contain other records as well. Also notice that case  $j = 0$  implies a suitable sequence of permutations always places the next record to be accessed into vector  $v_0$  (zone  $Z_0$ ).

**Lemma 6.2** *For any sequence of accesses  $r_1 \dots r_n$ , there is a suitable sequence of permutations available under the data structure maintenance plan given above.*

**Proof.** Let  $i$  be as in Definition 6.1. Since the plan allows  $\pi_i$  to be an arbitrary permutation on zone  $Z_{\rho(i)}$ ,  $\pi_i$  can be picked to properly fill in zones  $Z_j$  for  $0 \leq j < \rho(i)$  **provided zone  $Z_{\rho(i)}$  contains all the records for the next  $2^{\rho(i)-1}$  accesses.** We use induction on  $i$  to prove  $Z_{\rho(i)}$  does contain these records.

The basis of the induction is that  $\pi_0$  finds the records for the next  $2^{\rho(0)-1}$  access in zone  $Z_{\rho(0)}$ . This is trivially true since  $\rho(0) = k$  and zone  $Z_k$  contains all records.

Now suppose that permutations  $\pi_\ell$  for  $0 \leq \ell \leq i - 1$  have each found the records for the next  $2^{\rho(\ell)-1}$  accesses in zone  $Z_{\rho(\ell)}$  and that each  $\pi_\ell$  has, according to the plan, placed the records for the next  $2^j$  accesses into zone  $Z_j$  for  $0 \leq j < \rho(\ell)$ . If  $\rho(i) = k$ , then  $Z_{\rho(i)}$  contains all records as in the base case so again  $Z_{\rho(i)}$  contains the records for the next  $2^{\rho(i)-1}$  accesses.

Now suppose  $\rho(i) < k$ . By definition of  $\rho$ , this means that  $i = a2^{\rho(i)-1}$  for some odd integer  $a$ . Let  $i' = (a - 1)2^{\rho(i)-1}$ . Since  $a - 1$  is a multiple of two, this means  $\rho(i') > \rho(i)$  which, by the induction hypothesis, means  $\pi_{i'}$  put the records for the next  $2^{\rho(i)}$  accesses into  $Z_{\rho(i)}$ . Since there are no  $\ell'$  of the form  $b2^{\rho(i)-1}$  between  $i'$  and  $i$  (between  $(a - 1)2^{\rho(i)-1}$  and  $a2^{\rho(i)-1}$ ),  $\rho(\ell') < \rho(i)$  for  $i' < \ell' < i$ . This means that the  $\pi_{\ell'}$  between  $\pi_{i'}$  and  $\pi_i$  only permute subsets of  $Z_{\rho(i)}$  and so  $Z_{\rho(i)}$  retains the records placed there by  $\pi_{i'}$ . Since at most  $2^{\rho(i)-1}$  records have been accessed between  $i'$  and  $i$ ,  $Z_{\rho(i)}$  still has the records for the next  $2^{\rho(i)-1}$  accesses that were put there by  $\pi_{i'}$ . Thus  $\pi_i$  satisfies the induction hypothesis and therefore, all sequences  $r_1 \dots r_m$  can be produced. ■

The memory management plan just described follows the same principles used in [14] and [19]. The plan details are completely different for several reasons. One reason is that the next record read cannot be assumed to be adjacent to the previous record as happens when reading from a tape. Another is that we perform our permutations using exchanges rather than copying information from tape to tape.

Next we need subprograms to nondeterministically and obviously perform the initialization and the rearrangements as called for in the plan. We view initialization as a two step process. First the program initializes the data structure with zeros or “null records” using COPY operations. Then the program fetches records and, if null, replaces them with the desired initial values.

The primitive operation we use for rearranging data is the “bit exchange” operator EX defined by

$$\text{EX}(c, a_1, b_1, a_2, b_2) \Leftrightarrow \text{if } c \text{ then } (a_2 = b_1 \wedge b_2 = a_1) \text{ else } (a_2 = a_1 \wedge b_2 = b_1)$$

where  $c$ ,  $a_1$ , and  $b_1$  are input parameters and  $a_2$  and  $b_2$  are output parameters. This operation is deterministic and could be expressed as a sequence of COPY, NOT, OR, and AND operations. (Here we interpret the basic Boolean program operators as zero-one valued rather than Boolean valued.) In this paper, the EX operation is only used in the form  $\text{EX}(c, x, y, x, y)$  where parameters  $a_1$  and  $a_2$  are both matched to the same variable and parameters  $b_1$  and  $b_2$  are both matched to a second variable. The effect of operation  $\text{EX}(c, x, y, x, y)$  is to swap the values of  $x$  and  $y$  when  $c$  is 1 and to leave them unchanged when  $c$  is 0. The exchange concept extends to vectors  $x = (x_1 \dots x_\ell)$  and  $y = (y_1 \dots y_\ell)$  by the following sequence of  $\ell + 1$  operations:

$$\text{SET}(c)\text{EX}(c, x_1, y_1, x_1, y_1) \dots \text{EX}(c, x_\ell, y_\ell, x_\ell, y_\ell)$$

The choice of  $c$  determines if all bits will be swapped or no bits will be swapped. This sequence of instructions will be referred to as a “vector exchange”. The same temporary variable  $c$  can be reset and reused for all vector exchanges, so just one extra variable is needed to implement any number of exchanges.

Next we want to use vector exchanges to nondeterministically permute records. For all  $i$  and  $j$  such that  $1 \leq i \leq j \leq k$ , let  $E_i^j$  be the set of  $2^{j-1}$  exchanges of records in  $Z_j$  whereby the record in  $v_a$  is exchanged with the record in  $v_b$  where the binary representations of  $a$  and  $b$  differ only in the  $i$ -th bit (with bit 1 being least significant.) The sequence of  $(2j - 1) \cdot 2^{j-1}$  vector exchanges  $E_j^j \dots E_1^j \dots E_j^j$  has the same effect as the well-known Beneš network [3, 30] which is capable of performing any permutation.

For our construction, we don’t need the ability to perform arbitrary permutations. We just need an adequate set for each  $Z_j$  such that permutations satisfying Definition 6.1 can be constructed. In fact, the sequence of  $j \cdot 2^{j-1}$  vector exchanges  $E_1^j \dots E_j^j$  can perform an adequate set of permutations. We omit a proof of this since it is only a constant factor improvement. This sequence has the same effect as the well-known Omega network.

We now count the number of bit exchanges in the first  $2^{k-1}$  permutations and divide by  $2^{k-1}$  to get the average number of exchanges per permutation. Each subsequent  $2^{k-1}$  accesses repeats the same sequence of exchanges so the same average applies to longer sequences. For  $1 \leq j \leq k$ , the permutation  $\pi_j$  is performed on zone  $Z_j$  in  $2^{k-j-1}$  cases and each permutation on  $Z_j$  involves  $2^{j-1}(2j-1)$  vector exchanges. Zone  $Z_k$  is permuted one time and uses  $2^{k-1}(2k-1)$  vector exchanges. The average number of bit exchanges is thus bounded above by

$$\ell \cdot (k2^{k-1} + \sum_{j=1}^{k-1} j2^{j-1}2^{k-j-1})/2^{k-1} = \ell(k + \sum_{j=1}^{k-1} j/2)$$

which is  $O(\ell k^2)$ .

The data structure should only be used in a program which does at least  $2^{k-1}$  accesses, for otherwise a smaller value of  $k$  could be used. Assuming the program looks at all  $\ell$  bits of each accessed record, the length  $n$  of the program is at least  $\ell 2^{k-1}$ . To get the overhead for maintaining the data structure, we multiply the number of accesses by the average cost per access and get  $\ell k^2 2^{k+1}$  which is  $O(n(\log n)^2)$ , a q-linear function. In fact it is only  $O(n(\log r)^2)$  where  $r$  is the number of records, a number which can be much smaller than  $n$ .

Robson has given an efficient reduction of RAMs to formulas in [23]. He does this without assumptions about sizes of address fields or data fields. The channelwidths of the formulas he produces are too high for our purposes but the formulas have fewer clauses because his simulation does only one sort. A one-sort technique was also used in [29] where a size  $O(n(\log n)^2)$  reduction from Turing machines to formulas is given. We believe that the ideas in [23] can be blended with the ideas here so that general RAM simulations are reduced to NSLPs having the same sub-linear memory constraints as the RAM.

## 7 CLIQUE and PARTITION

In this section we show that the problems **CLIQUE** and **PARTITION** are in the class  $\text{NQ}n^{1/2}$ . Both these problems are known to have power index at most  $1/2$  [27] but for reasons other than performing nondeterministic computations in sub-linear space. Here we give nondeterministic algorithms for these problems which are q-linear in time and have space bounds which have polynomial

index  $1/2$  thereby establishing that these languages are in  $\text{NQ}n^{1/2}$ . These algorithms are essentially adaptations of previous algorithms to space-efficient versions.

The problem **CLIQUE** is this: given a graph with  $n$  edges and an integer  $k$ , does the graph have a clique of size  $k$ ?

The fact that **CLIQUE** can be solved in  $2^{O(\sqrt{n})}$  time is a simple consequence of the fact that a clique of size  $k$  has  $k(k-1)/2$  edges and hence we know at once that a graph with  $n$  edges cannot have a clique of size  $k$  unless  $k(k-1)/2 \leq n$ . The time required to try all cliques of size  $k$  in this case is only  $2^{O(\sqrt{n})}$ .

To show **CLIQUE** is in  $\text{NQ}n^{1/2}$ , we outline a suitable reduction from **CLIQUE** to a set of NSLPs. “Suitable” in this case means:

1. the reduction takes polynomial time,
2. the size of the output program is q-linear in the size of the problem instance given as input,
3. the number of variables in the NSLP is bounded by some function  $p$  of program size where  $p$  has polynomial index  $1/2$ .

The program constructed for a particular input graph will try to verify that the graph has a clique of size  $k$  by guessing a set of  $k$  nodes and attempting to verify that those nodes form a clique. This is, of course, a well-known nondeterministic plan. The tricky part is to construct a program to carry out this plan using only  $O(\sqrt{n})$  space.

The first thing the reduction does is check to see if  $n \geq k(k-1)/2$ . If the inequality does not hold, there is no clique of size  $k$ , and the reduction outputs a trivial unexecutable program. If the inequality holds, the reduction prepares a new description of the graph as a list of nodes followed by a list of edges where the node names use at most  $\log(n)$  bits, no graph edge occurs more than once on the edge list, and each edge has distinct end nodes. The size of the new description is  $O(n \log(n))$  and thus is q-linearly related to the size of the original representation.

Based on this new description, the reduction constructs a NSLP which uses the following plan:

1. Guess a node subset  $S$  of size  $k$  (using the node list as input).

2. Count the number of edges for which it successfully verifies that the end nodes are in  $S$  (using the edge list as input).
3. Accept if the number of such edges is  $k(k - 1)/2$ .

The only memory required by the program is the  $k \log n = O(\sqrt{n} \log n)$  bits necessary to remember the set  $S$ . (The  $O(n \log n)$  bits describing the nodes and edges are treated in the program as constants which one can view as variables that are not changed by the nondeterminism.)

To implement the plan, we use a “nondeterministic database” as described in Section 6 to store the nodes in set  $S$ . Each node that it decides to put in  $S$  is put in a blank record (provided a blank record is fetched). The number of nodes entered is counted and verified to be  $k$ . For each edge, the program fetches two nodes and tests them for a match with the given edge. A counter is incremented whenever both edge nodes are in  $S$ .

To count using straight line code, we invoke program segments which imitate an adder circuit that adds the value of a binary variable to the counter memory. If the variable has value zero, the effect is to leave the count alone. If the variable has value one, the effect is to increment the counter. To test the value of a counter, we imitate a comparison circuit.

The problem **PARTITION** is this: given a set of numbers, can the numbers be partitioned into two sets having the same sum? We let  $n$ , the problem size, be the total number of bits needed to represent all the numbers.

To solve **PARTITION** deterministically, we try to determine if some subset of the numbers add up to  $t$  where  $t$  is one-half the sum of the numbers. To this end, the numbers are divided into two sets, those numbers which are  $\sqrt{n}$  bits or longer and those which are not. A Boolean array is created indexed from 0 to  $s$  where  $s$  is the sum of the small numbers. The array has at most  $n2^{\sqrt{n}} + 1$  entries which is  $2^{O(\sqrt{n})}$ . Using dynamic programming as in [10], the  $i$ -th array element is set to TRUE if and only if some subset of the small numbers sum to  $i$ . The time of the dynamic programming is  $2^{O(\sqrt{n})}$ .

Next, the algorithm considers all subsets of the large numbers. Because there are at most  $\sqrt{n}$  large numbers, the number of subsets is  $O(2^{\sqrt{n}})$ . The sum  $\ell$  of the numbers in the selected set of

large numbers is computed and, if  $0 \leq t - \ell \leq s$ , the  $(t - \ell)$ -th array element is checked to see if  $t - \ell$  can be obtained from a subset of small elements. If the element is `TRUE`, the answer to **PARTITION** is “yes” and computation ceases. If not, the next subset of large numbers is tried. Note that keeping the large array in memory is a key time-saving device, because computing sums of the individual small number subsets one at a time could take  $2^{O(n)}$  time.

To solve **PARTITION** nondeterministically in  $O(\sqrt{n})$  space, we again want to see if some subset of the numbers add up to  $t$ . Because the program constructed by the reduction is customized to each **PARTITION** instance, the bits of  $t$  and of the input numbers appear in the program as constants. As was done in the deterministic case, the numbers are partitioned into large numbers (those requiring  $\sqrt{n}$  or more bits) and small numbers (those which do not). The first thing the program does is to compute the sum of some guessed subset of the small numbers. This sum requires at most  $\sqrt{n} + \log_2(n)$  bits and will be kept in the program memory. To compute this sum, the program considers the small numbers one at a time and nondeterministically decides if that number should be added to the sum in memory. The nondeterministic choice is indicated by setting a certain “choice bit”. The addition is done by an addition circuit which, depending how the choice bit is set, either adds the small number to the memory or adds zero to the memory. Because the one choice bit is reused for each small number, the program has no memory about which small numbers appear in the sum. Because there might be  $O(n)$  small numbers, remembering the selected set would be a violation of the memory constraint.

The memory has one bit for each large number and these are set nondeterministically to indicate if the corresponding large number is to be added to the sum. The number of such bits is  $O(\sqrt{n})$  so this number is within the memory constraints. There is a potential difficulty in that the memory constraint makes it impossible to store sums of the large numbers. We get around this difficulty by computing successive bits of the sum starting with the least significant and working up. As the bits are computed, they are compared with the corresponding bit of  $t$  to see if the bits computed so far are the same as the bits of  $t$ . After computing and checking each bit, the memory is reused so that the sum of the numbers in the subset never resides in memory. To compute the bits, the

corresponding bit of the sum of the small numbers is added to the bits of the selected large numbers (as indicated by the bit which remembers whether or not a given large number has been selected). The memory requirements for performing the addition are  $O(\log(n))$  so that carry bits can be remembered.

The two programs for **PARTITION** are similar in that they both attempt to find a subset of numbers which sums to  $t$  and both programs treat the small numbers and the large numbers separately. The deterministic program cannot afford to consider each small number sum separately since that would take too much time. This is overcome with the dynamic programming and the corresponding  $O(2^{\sqrt{n}})$  memory, exponentially more than used by the nondeterministic program. Our constructions imply another deterministic method which uses only  $O(n)$  memory and also shows the power index is at most  $1/2$ . However “equal power index” does not imply “equal complexity”. The time of the implied algorithm ( $n^{\Theta(\sqrt{n})}$ ) is much worse than the time of the given algorithm ( $2^{\Theta(\sqrt{n})}$ ).

The introduction mentions **PLANAR-SAT** as a third problem that can be solved in  $2^{O(\sqrt{n})}$  time. In this case, well-known methods can be interpreted as putting the problem in  $\text{NQ}n^{1/2}$ . The planar separator theorem [17] can be used recursively to find an ordering of the clauses having line channelwidth  $O(\sqrt{n})$  (a PQ-reduction) and then nondeterminism can be applied to the resulting ordered formula.

## 8 Final Remarks

The composition of the reductions from choice-oblivious Turing machines to programs, programs to binary programs, and then binary programs to **SAT** is very size efficient. It is linear if we count the size of the formula to be the number of variable occurrences;  $n \log n$  if we add in the cost of the strings needed to name the variables using a finite alphabet. To apply the reduction to arbitrary nondeterministic Turing machines (not just choice-oblivious machines) one can first use the  $n \log n$  construction of [19] to simulate the machine obliviously. This is essentially the plan used by Schnorr [24] and by Cook [7]. A different  $n \log n$  plan was described by Robson in [21] and a linear plan for one-tape Turing machines is given in [22]. None of these references were concerned

with channelwidth issues.

Monien and Sudborough [18] have a reduction which is not  $q$ -linear but which bounds the bandwidth by the amount of tape memory used. Their main result, however, can be viewed as a consequence of the fact that bounded bandwidth implies bounded line channelwidth. Thus our reductions can be viewed as simultaneously having the best features of both the efficient size reductions and the Monien-Sudborough reductions.

The composition which takes formulas to choice-oblivious Turing machines is also quite efficient. The size is  $n \log n$  when the formula size is measured by variable occurrences and linear when variable names of size  $\log n$  are used.

As pointed out in [26, 27], the existence of an  $n^k$  size reduction from a problem  $\mathbf{L}$  to the problem  $\mathbf{SAT}$  suggests that the best algorithm for  $\mathbf{L}$  takes at least  $2^{O(n^{1/k})}$  deterministic time since any improvement in time would imply an improvement in the brute force algorithm for  $\mathbf{SAT}$ . For some NP-complete problems, the best known algorithm and the best known size efficient reduction are in agreement (ie.  $a = 1/k$ ). This includes **CLIQUE**, **PARTITION**, **PLANAR-SAT** and many others. On the other hand, if the best known algorithm for  $\mathbf{L}$  takes  $2^{O(n^a)}$  time for  $a > 1/k$ , then an improvement in reduction size or in solution time is possible without contradicting the assumption that the best algorithm for  $\mathbf{SAT}$  is brute force. There are many examples of NP-complete problems where there is such a discrepancy between  $1/k$  and  $a$  [27] and thus there is realistic hope that a faster algorithm or a better reduction can be found for these problems. In such cases, the above results suggest that minimizing nondeterministic space may be a good first step toward finding a better algorithm.

Finally, we work out some implications of our constructions if a problem  $P$  is solved by a choice-oblivious Turing machine  $\mathcal{T}$  in time  $n^t$  and space  $n^a$  where  $t$  and  $a$  may be constants greater than one. Of course,  $a \leq t$  since a Turing machine cannot take more space than time and so  $a/t \leq 1$ . Our constructions tell us how to convert inputs  $w$  for  $\mathcal{T}$  into formulas  $F_w$  of size  $|w|^t$  having line channelwidth  $|w|^a$ . As a function of formula size, the line channelwidth is  $|F_w|^{a/t}$  and the deterministic algorithm of Section 3 solves  $F_w$  in time  $|F_w|^{(|F_w|^{a/t})}$  or  $|w|^{t|w|^a}$ . Thus the power

index of solving problem  $P$  is at most  $a$ . But even if  $a \leq 1$ , the constructions no longer imply problem  $P$  is in  $NQn^a$ .

## References

- [1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading MA, 1974.
- [2] A. Arnborg, D.G. Corneil, and A. Proskurowski, "Complexity of finding embeddings in a  $k$ -tree," *SIAM J. Alg. and Discr. Methods* 8, 1987, pp. 277-284.
- [3] V. Beneš, *Mathematical Theory of Connecting Networks*, Academic Press, New York, 1965.
- [4] H.L. Bodlaender, "Dynamic programming on graphs with bounded tree width," *Proceedings of ICALP*, LNCS 317 (1988), 105-118.
- [5] H.L. Bodlaender, M.R. Fellows, and M.T. Hallett, "Beyond NP-Completeness for Problems of Bounded Width: Hardness for the W Hierarchy", *Proceedings of the 26th Annual ACM Symposium on the Theory of Computing*, 1994, pp. 449-458.
- [6] L. Cai and J. Chen, "On the Amount of Nondeterminism and the Power of Verifying", *SIAM Journal on Computing*, 26, 3, June 1997, pp. 733-750.
- [7] S.A. Cook, "Short Propositional Formulas Represent Nondeterministic Computations", *Information Processing Letters*, 26, 1988, pp. 269-270.
- [8] R.G. Downey and M.R. Fellows, *Parameterized Complexity*, Springer, New York, 1997.
- [9] S. Even and Y. Yacobi, "Cryptography and NP-Completeness", *Proc. 7th ICALP*, LNCS vol. 85, 1980, pp. 195-207.
- [10] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, CA, 1979.

- [11] E. Grädel, “On the Notion of Linear Time Computability”, *International Journal of Foundations of Computer Science*, 1, 3, 1990, pp. 295-307.
- [12] J. Gustedt, “Pathwidth for Chordal Graphs is NP-complete”, *Discr. Appl. Math.*, 45, 1993, pp. 223-248.
- [13] J. Hartmanis and R.E. Stearns, “On the Computational Complexity of Algorithms”, *Trans. of the Amer. Math. Society*, 117, 5, May 1965, pp. 285-306.
- [14] F.C. Hennie and R.E. Stearns, “Two-Tape Simulation of Multi-Tape Turing Machines”, *Journal of the ACM*, 13, 4, Oct. 1966, pp. 533-546.
- [15] H.B. Hunt III, “Observations on the Complexity of Regular Expression Problems”, *Journal of Computer and System Science*, 19, 3, Dec. 1970, pp. 222-236.
- [16] L. Longpré and A. Selman, “Hard Promise Problems and Nonuniform Complexity”, *Theoretical Computer Science*, 115, 3, 1993, 227-290.
- [17] R.L. Lipton and R.E. Tarjan, “Applications of a Planar Separator Theorem”, *SIAM Journal on Computing* 9, 1980, pp. 615-629.
- [18] B. Monien and I.H. Sudborough, “Bounding the Bandwidth of NP-complete Problems”, *Lecture Notes in Computer Science*, vol. 100, Springer-Verlag, Berlin, 1981, pp. 279-292.
- [19] N. Pippinger and M.J. Fischer, “Relations Among Complexity Measures”, *Journal of the ACM*, 26, 2, April 1979, pp. 361-381.
- [20] N. Robertson and P.D. Seymour, “Graph Minors III, Algorithmic Aspects of Treewidth”, *Journal of Algorithms*, 7, 1986, pp. 309-322.
- [21] J.M. Robson, “A New Proof of the NP-completeness of Satisfiability”, *Proc. 2nd Australian Computer Science Conf.*, 1979, pp. 62-69.
- [22] J.M. Robson, “Linear Size Formulas for Non-deterministic Single Tape Computations”, *Proc. 11th Australian Computer Science Conference*, 1988.

- [23] J.M. Robson, “An  $O(T \log T)$  Reduction from RAM computations to Satisfiability”, *Theoretical Computer Science*, 82, 1991, pp. 141-149.
- [24] C.P. Schnorr, “Satisfiability is Quasi-linear Complete in NQL”, *Journal of the ACM*, 25, 1, 1978 pp. 136-145.
- [25] A. Selman, “Promise Problems Complete for Complexity”, *Information and Computation* 78, 1988, 87-98.
- [26] R.E. Stearns, “It’s Time to Reconsider Time”, *Communications of the ACM* 37, 11, 1994, pp. 95-99.
- [27] R.E. Stearns and H.B. Hunt III, “Power Indices and Easier Hard Problems”, *Math. Systems Theory* 23, 1990, pp. 209-225.
- [28] R.E. Stearns and H.B. Hunt III, “An Algebraic Model for Combinatorial Problems”, *SIAM Journal on Computing* 25, 1996, pp.448-476.
- [29] R.E. Stearns and H.B. Hunt III, “On the complexity of the satisfiability problem and the structure of NP”, *Dept. of Computer Science, University at Albany*, Technical Report 86-21, 1986.
- [30] Waksman, A., “A Permutation Network”, *Journal of the ACM* 15, 1, 1968, pp. 159-163.