COUNTING COMPLEXITY
AND
COMPUTATIONAL GROUP THEORY

by

VINODCHANDRAN N. V.

A THESIS IN COMPUTER SCIENCE

Submitted to the University of Madras in partial fulfillment of the requirement for the degree of Doctor of Philosophy

MAY 1998

The Institute of Mathematical Sciences
C.I.T. Campus, Tharamani
Chennai (Madras), Tamilnadu - 600 113, INDIA

CERTIFICATE

This is to certify that the Ph.D. thesis submitted by VINODCHANDRAN N. V. to the University of Madras, entitled **Counting Complexity and Computational Group theory**, is a record of bonafide research work done during 1993-1998 under my supervision. The research work presented in this thesis has not formed the basis for the award to the candidate of any Degree, Diploma, Associateship, Fellowship or other similar titles.

It is further certified that the thesis represents independent work by the candidate and collaboration when existed was necessitated by the nature and scope of problems dealt with.

V. Arvind

V. Arvind

Thesis Supervisor

May 1998

[illegible]
MADRAS-600 113.

The Institute of Mathematical Sciences
C.I.T. Campus, Tharamani
Chennai (Madras), Tamilnadu - 600 113

Abstract

The study of counting complexity classes has been a very fruitful and promising area in complexity theory. This study has given important insights into the inherent complexity of many natural computational problems. Problems arising from group theory have been studied by many researchers. These problems are interesting from the complexity-theoretic viewpoint since the complexity status of many of these problems is not settled.

In this dissertation, we study some problems from group theory in the context of counting complexity. More specifically, we place some basic computational group-theoretic problems in counting classes of low complexity. These results help in giving further insights into the intriguing nature of the complexity of these problems.

This thesis consists of two parts. In Chapter 4, which comprises the first part, we study the complexity of three basic computational group-theoretic problems over black-box groups. The problems are Membership Testing, Order Verification and Isomorphism Testing. These are computational problems for which no polynomial-time algorithms exist. It was shown that over general black-box groups, Membership Testing is in NP $\cap$ co-AM, Order Verification is in AM $\cap$ co-AM, and Isomorphism Testing is in AM [BS84, Bab92]. We show that these problems, over *solvable* black-box groups, are in the counting class SPP. The proof of this result is built on a constructive version of the fundamental theorem of finite abelian groups. The class SPP is known to be *low* for the counting classes PP, $\mathrm{C_{=}P}$ and $\mathrm{Mod}_k\mathrm{P}$ for $k \geq 2$ [FFK94]. Since it is unlikely that the class NP is contained in SPP, these upper bounds give evidence that these problems are unlikely to be hard for NP.

In the second part of the thesis we study the problem of computing a generator set of an *unknown* group, given a membership testing oracle for the group. Because of the close relation of this problem with concept learning, we study this problem in the framework of learning theory. In Chapter 5, for analyzing the complexity of learning, we introduce a new model of exact learning called the *teaching assistant* model. This model can be seen as an enhancement of Angluin's [Ang88] exact learning model. The main ingredient of this model is the notion of a teaching assistant which acts as an intermediate agent between the learner and the teacher. The power of this model for studying the complexity of various concept classes, comes from the fact that it is possible to define *classes* of teaching assistants. These classes are analogous to the known complexity classes. The teaching assistant classes of main interest to us are the ones analogous to the counting complexity classes SPP and LWPP. In Chapter 5, after giving detailed definitions of all the notions involved in this model, we study the complexity of learning three representation classes in this model. These are the classes SYM of permutation groups, $\mathrm{LS}(p)$ of linear spaces over the finite field of size p and the class 3-CNF of boolean functions represented in conjunctive normal form where each clause has at most three literals. We show that the class SYM is learnable with an LWPP-assistant and $\mathrm{LS}(p)$ is learnable with an SPP-assistant. On the other hand, we also show that 3-CNF is not learnable with an SPP-assistant (LWPP-assistant) unless $\mathrm{NP} \subseteq \mathrm{SPP}$ (respectively, $\mathrm{NP} \subseteq \mathrm{LWPP}$). These containments are unlikely. Motivated by these results, in Chapter 6 we define more assistant classes and prove absolute separations among these assistant classes. For separating various assistant classes we use some natural subclasses of the representation class SYM.

All the results leading to this dissertation have been published. The results that we show in Chapter 4 have appeared in [AV97b, Vin97]. The results proved in Chapters 5 and 6 have appeared in [AV96] and [AV97a] respectively.

Acknowledgments

Foremost, I would like to express my deep gratitude to my advisor V. Arvind. He has been a wonderful guide throughout my research career. If anything that I can claim to know about research; reading and writing technical papers, solving problems and more importantly posing new problems, I learnt only from him. I am also grateful to him for giving me confidence and comfort when I required them most.

I greatly thank Meena Mahajan for teaching me circuit complexity and being accessible virtually at anytime for almost anything. Thanks are due to all the members of the TCS group at IMSc; Kamal Lodaya, Venkatesh Raman R. Ramanujam, Anil Seth, V. Kamakoti and the graduate students P. Madhusudan, Swarup Mohalik, S. V. Nagaraj and S. Srinivasa Rao, for providing an excellent research environment. Special thanks to S. S. Rao for his valuable companionship. The company of A. Srinivasan and Jyothishman Chatterjee had been wonderful during the first two years of my graduate days.

I am indebted to the TCS group at Spic Mathematical Institute for their kind encouragement. Thanks to Madhavan Mukund, K. V. Subrahmanyam, P. S. Thiagarajan and Deepak D'Souza. I am especially thankful to Deepak for carefully proofreading an earlier version of this thesis.

Thanks to all the members of the IMSc hostel, specially to those in IMSc footer team. Without the footer experience at IMSc, I strongly doubt whether my stay at IMSc would have been enjoyable.

Finally, no words can express my gratitude towards my family for all the troubles they had gone through (and still going through!) just for me. My interest in mathematics would never have been there, had Amma not taken special care at an early stage of my schooling.

Contents

Chapter 1

Introduction

A central aim of computational complexity theory is to classify computational problems according to the resources (usually time and space) required for solving them on a given model of computation. Most computational problems arising in practice fall naturally into different *complexity classes*, depending on the resource bounds of interest. Using the fundamental concept of resource-bounded reductions, it is possible to meaningfully compare the relative difficulty of two given problems within a complexity class. This gives rise to the notion of *complete problems* for each complexity class: complete problems are the hardest problems in a complexity class. The theory of NP-completeness illustrates the usefulness of these notions. Natural problems in the class NP, which are computationally intractable in practice, turn out to be NP-complete. Thus, within the complexity class NP, we have the subclass P of problems that can be feasibly solved (i.e. in polynomial time) and the subclass of NP-complete problems. A question that arises is whether there could be problems of "intermediate" difficulty. Might it not be the case that NP consists solely of P and NP-complete problems? Assuming P$\neq$NP, it was shown by Ladner [Lad75] that there are many problems in NP that are neither in P nor NP-complete.

In spite of the fact that there are a large number of problems of intermediate complexity in NP as shown by Ladner's theorem, there are only a few *natural* candidates for problems of intermediate complexity (in contrast, the number of natural

NP-complete problems abound, see [GJ78]). In fact, to date, none of them are provably of intermediate complexity even if we assume P$\neq$NP. A well-known example is the problem of testing whether two graphs are isomorphic (in short, GI). Researchers believe that GI is not in P since there is no subexponential time algorithm known for this problem. On the other hand, it is not known whether GI is NP-complete. The theory of reductions and completeness does not give us any further insight into the computational complexity of problems such as GI.

A useful tool that provides some understanding about problems of intermediate complexity like GI is the notion of *lowness* for complexity classes introduced by Schöning [Sch83]. To make this notion precise, let $\mathcal{C}$ be any relativizable complexity class. A language A is said to be *low* for $\mathcal{C}$ if $\mathcal{C}^A = \mathcal{C}$, where $\mathcal{C}^A$ denotes the relativized version of $\mathcal{C}$ with A as oracle. Lowness of A for $\mathcal{C}$ intuitively means that A is powerless as an oracle to $\mathcal{C}$. It is easy to see that any problem in NP which is low for some level of the polynomial-time hierarchy (PH) is not NP-complete unless PH collapses. Thus, lowness of a problem for some level of PH is an evidence that the problem is unlikely to be complete for NP. It is shown in [Sch88, BHZ87] that GI is low for Σ_2^p, the second level of PH. Since the introduction of this notion the study of lowness of problems to various complexity classes has been of much interest (see [Köb95] for a survey on lowness).

1.1 Counting complexity classes and lowness

Among various complexity classes, the study of *counting classes* has received considerable attention. In general, these are the language classes related to the function class #P. Valiant [Val79] introduced the class #P as a class consisting of functions computing the number of accepting paths of polynomial-time nondeterministic Turing machines and showed that computing the permanent of a 0-1 matrix is complete for #P. The study of counting classes has been a major research area since this seminal result. Here we briefly survey some important counting classes. Some of these classes are defined later in the thesis. See the survey article [For97] and references therein for more details.

Among counting complexity classes, considerable research has gone into understanding the structure of the class PP (Probabilistic polynomial-time). This class was originally defined by J. Gill [Gil77] and independently by J. Simon [Sim75]. The class PP is very closely related to the class #P. Indeed, it is easy to show that the closure of PP and #P under polynomial-time Turing reductions coincide.

PP is computationally a hard class; the class NP is contained in PP. The hardness of PP was further established by a celebrated result due to S. Toda [Tod91]. He showed that the entire polynomial-time hierarchy is contained in P^{PP}. PP also enjoys many nice closure properties. It is closed under complementation [Gil77]. The question posed by Gill in his seminal work [Gil77], whether PP is closed under intersection (or union), was settled in the affirmative by Beigel et. al. in [BRS95]. The techniques used in [BRS95] were extended by Fortnow et. al. [FR96] to show that PP is also closed under polynomial-time truth-table reductions.

The notion of lowness, originally defined in relation to the polynomial-time hierarchy [Sch83], was first studied for the class PP by Torán in [Tor88]. He gave a sufficient condition for languages to be low for PP. More precisely, he showed that languages in CH [Wag86] (Counting hierarchy; a hierarchy of classes over PP) which are #P-rankable are low for PP. Since then, many researchers have studied the structure of low sets for PP. It was shown in [KSTT92] that the class Few (introduced in [CH90] as a generalization of FewP [All86]) is low for PP. In [KSTT92] the authors also show that the probabilistic class BPP is also low for PP. Since these lowness proofs relativize, it is easy to get more and more complex sets that are low for PP. Another interesting lowness result shown in [KSTT92] is that the sparse sets in NP are low for PP.

Other counting classes that have been studied include classes $\mathrm{C_{=}P}$ (defined in [Wag86]) and $\mathrm{Mod}_k\mathrm{P}$ for $k \geq 2$ ([PZ83, BG92, Her90, CH90]). These are also computationally hard classes; for example, from the definition of $C_=P$, it follows that co-NP $\subseteq C_=P$. Also, as an intermediate step in the proof of Toda's theorem it is shown that PH $\subseteq$ BP. $\oplus$ P (the class obtained by applying the BP operator to $\oplus$P) [Tod91]. In general, it holds that PH $\subseteq$ BP.Mod$_k$P [TO92].

The class UP introduced by Valiant [Val76] is another important complexity class. UP consists of those languages in NP accepted by nondeterministic polynomial-time machines having at most one accepting path. Valiant defined UP for studying the relative complexity of checking and evaluating. This class later found applications in the areas of one-way functions and cryptography; for example, in [GS84] it is shown that P is different from UP if and only if one-way functions exist. More recently, it is shown in [FK92] that the problem of deciding primality is in UP $\cap$ co-UP. UP is also low for many counting classes like PP, $C_=P$ and Mod$_k$P for $k \geq 2$.

The class GapP, studied by Fenner et. al. in [FFK94], is an important function complexity class. The main motivation for defining this class is from the observation that the class #P cannot take negative values. This led to the introduction of the class GapP as the closure of #P under subtraction. The class GapP satisfies many algebraic closure properties. Specifically, it is closed under exponential summation and polynomial multiplication. Using the notion of gap-definability, Fenner et. al. have given a uniform method for defining different counting complexity classes (gap-definable counting classes). These new definitions have proved to be very convenient to work with.

The notion of gap-definability also gives rise to some new counting complexity classes. Of special interest to us is the class SPP [FFK94]. (This class is also independently studied in [OH93] under the name XP and in [Gup95] under the name $\mathcal{Z}$UP). The class SPP can be seen as the *gap* analogue of the class UP. More precisely, SPP consists of languages that are accepted by polynomial-time nondeterministic Turing machines such that, for inputs in the language the difference in the number of accepting and rejecting paths of these machines is exactly one, and for inputs not in the language this difference is zero.

The class SPP is large enough to include FewP and it is contained in the classes PP, $C_=P$ and Mod_kP for $k \geq 2$. This class is interesting mainly because of its lowness properties. It is shown in [FFK94] that SPP is exactly the class of languages that are low for the function class GapP [FFK94]. From this result it easily follows that SPP is low for PP, $C_=P$ and Mod_kP for $k \geq 2$ and even SPP itself. In particular, it is closed under polynomial-time Turing reductions. Hence, intuitively, SPP is a class of "low counting complexity".

Another complexity class which is of interest is the class LWPP [FFK94]. LWPP is a generalization of SPP. From the definitions, it follows that SPP $\subseteq$ LWPP. LWPP also enjoys many of the lowness properties of SPP. For example, LWPP is low for PP and $C_=P$ and is closed under polynomial-time Turing reductions.

Though SPP and LWPP are structurally defined classes, it is shown that some natural, computationally hard problems fall in these classes. In [KST92], it is shown that GI is in LWPP. They also show that Graph Automorphism problem (GA; the problem of deciding whether a graph has a nontrivial automorphism or not) is in SPP.

Membership of a problem in the class SPP or LWPP can be seen as an evidence that the problem is unlikely to be hard for the class NP. Firstly, as mentioned before, intuitively we can say that problems that are in SPP or LWPP are of low counting complexity and hence it is unlikely that these problems are NP-hard. Secondly, the classes SPP and LWPP are defined by imposing strong *restrictions* on the computation tree of nondeterministic Turing machines accepting languages in them. It will be surprising that all the problems in NP can be accepted by Turing machines with such restrictions. In view of these explanations, membership of GI in LWPP shown in [KST92] gives additional evidence that GI is unlikely to be complete for NP.

1.2 Contribution of the Dissertation

The main contribution of this dissertation is in proving upper bounds on the counting complexity of some computational problems that arise from group theory. Broadly, it consists of two parts. The first part concentrates on studying the complexity of three basic, computationally hard group-theoretic problems over black-box groups. These problems are Membership Testing, Order Verification and Isomorphism Testing. We investigate their counting complexity over a class of groups called *solvable* groups. Solvable groups form a large subclass of finite groups. Indeed, a celebrated theorem in group theory due to Fiet and Thompson says that all finite groups of odd order are solvable. It is shown that all these problems over solvable black-box groups are in the complexity class SPP. This result provides additional evidence that these problems are unlikely to be hard for NP. Another aspect of this result is that it adds to the list of natural problems that are in SPP but not known to be in P. Graph Automorphism and permutation Group Intersection are among the few members that were already known to be in SPP but not known to be in P [KST92].

The main focus of the second part of the thesis is to investigate the complexity of a computational problem over finite groups which can be thought of as an "inverse problem" of Membership Testing. The problem can be informally stated as follows. Given an *unknown* permutation group G and an oracle to test membership in G; compute a generator set for the group. We observe that this problem is very closely related to the problem of learning representation classes in computational learning theory.

Motivated by this, in the second part of the thesis, we focus on studying the complexity of various representation classes with respect to their learnability. Of particular interest to us is the learnability of some group theoretic and linear algebraic concept classes. We compare the complexity of learning these algebraic concept classes to that of learning boolean functions represented in conjunctive normal form. One of the learning models that we are interested in is the exact learning model proposed by Angluin [Ang88]. It turns out that Angluin's model of exact learning is inadequate for distinguishing the complexity of learning various classes of interest to us. In order to do a finer classification of the complexity of exact learning, we propose a refinement of Angluin's model called the *teaching assistant* model of exact learning. The main ingredient of this model is a new agent namely the teaching assistant. A teaching assistant acts as an interface between the teacher and the learner. In order to classify the complexity of learning, we define the notion of teaching assistant classes. These classes are defined in exact analogy with known complexity classes. We show that the complexity of exactly learning algebraic concepts like permutation groups or linear spaces is different from the complexity of learning bounded CNF formulas in the teaching assistant model. More specifically, as one of our main results, we show that while permutation groups (linear spaces over fixed finite fields) can be learned using a teaching assistant in the assistant class analogous to LWPP (respectively, SPP), it is unlikely that the class 3-CNF can be learned using an LWPP or SPP assistant (unless NP $\subseteq$ LWPP or NP $\subseteq$ SPP). We also investigate the power of various assistant classes and exhibit representation classes which separate these assistant classes.

Although the two parts of the thesis are addressing apparently two different issues, there are some underlying connections between the two parts. Firstly, in both the parts, we are addressing the issue of analyzing the complexity of some computational problems over finite groups, although the exact nature of the problems differ. Secondly and importantly, while in the first part the complexity class of interest to us is SPP, in the second part, our interest is in the teaching assistant classes related to low complexity counting classes SPP and LWPP. Hence, the learning algorithms we design with LWPP and SPP teaching assistants are built on the complexity-theoretic ideas used in showing the upper bound results in the first part of the thesis.

In the next two sections, we briefly explain the technical contents of the thesis more formally. Most of the group-theoretic and complexity-theoretic notations that we use are given in the next chapter.

1.3 Solvable Black-box Group Problems

In this section, we explain the contents of the first part of the thesis in more detail. This part is devoted to the investigation of the counting complexity of three basic computational problems over solvable black-box groups. Before giving the exact definitions of problems of interest to us, we explain the framework of black-box groups. Intuitively, in this framework we have an infinite family of abstract groups. The elements of each group in the family are uniquely encoded as strings of uniform length. The group operations (product, inverse etc) are assumed to be easily computable. Black-box groups are subgroups of groups from such a family and they are presented by generator sets.

Remark. We would like to note here that the black-box groups we define above are a slightly restricted version of black-box groups introduced in [BS84]. The definition in [BS84] is technically more general so as to incorporate factor groups.

Definition 1.3.1 A *group family* is a countable sequence $\mathcal{B} = \{B_m\}_{m\geq 1}$ of finite groups B_m, such that there are polynomials p and q satisfying the following conditions. For each $m \geq 1$, elements of B_m are uniquely encoded as strings in $\Sigma^{p(m)}$. The group operations (inverse, product and testing for identity) of B_m can be performed in time bounded by $q(m)$, for every $m \geq 1$. The order of B_m is computable in time bounded by $q(m)$, for each m. We refer to the groups B_m of a group family and their subgroups (presented by generator sets) as *black-box* groups. A class $\mathcal{C}$ of black-box groups is said to be a *subclass* of $\mathcal{B}$ if every $G \in \mathcal{C}$ is a subgroup of some $B_m \in \mathcal{B}$.

For example, let S_n denote the permutation group on n elements. Then, $\{S_n\}_{n\geq 1}$ is a group family of all permutation groups S_n. As another example let $GL_n(q)$

denote the group of all $n \times n$ invertible matrices over the finite field F_q of size q. The collection $\{GL_n(q)\}_{(n,q)}$ is a group family. For any group family $\mathcal{B}$, the class of all abelian (solvable) subgroups $\{G \mid G < B_m$ for some m and G is abelian (respectively solvable)$\}$ is a subclass of $\mathcal{B}$.

Let $\mathcal{B} = \{B_m\}_{m>0}$ be a group family. We consider the following three basic computational problems over black-box groups.

Membership Testing $\triangleq \{(m, S, g) \mid \langle S \rangle < B_m \text{ and } g \in \langle S \rangle\}$.
Order Verification $\triangleq \{(m, S, n) \mid \langle S \rangle < B_m \text{ and } |\langle S \rangle| = n\}$.
Group Isomorphism $\triangleq \{(m, S_1, S_2) \mid \langle S_1 \rangle \text{ and } \langle S_2 \rangle \text{ are isomorphic subgroups of } B_m\}$.

In Chapter 4, we prove the following theorem.

Theorem *Over any group family $\mathcal{B}$, the problems* Membership Testing, Order Verification, Group Isomorphism *over the subclass of solvable groups are in* SPP *and hence low for the classes* PP, C$_=$P, *and* Mod$_k$P *for $k \geq 2$.*

A few remarks about the above theorem are in order. Using the fact that testing for primality is in UP [FK92], it can be shown that the above-mentioned problems when defined over *cyclic* groups are in the class UP. So the above result can be seen as a nontrivial extension of the upper bound for cyclic groups to solvable groups. Note that the class SPP is a generalization of the class UP.

We very briefly explain the main ideas that go into the proof of the above theorem. Since solvable groups can be viewed as a series of abelian factor groups, we first consider the above problems over abelian groups. Using a constructive version of a fundamental theorem on the structure of abelian groups, we first show the same upper bound for abelian factor groups. Then, using a procedure for computing normal closure to get generator sets for the commutator group of a finite group, we extend the upper bound for abelian groups to solvable groups.

1.4 Complexity of Exact Learning

In this section, we explain the contents of the second part of the thesis. As mentioned before, in this part we are interested in analyzing the complexity of learning some group-theoretic representation classes. Before we state our main results, we give a very brief introduction to learning theory.

Learning theory is concerned with providing mathematical models for machine learning and analyzing the learnability/non-learnability of various classes of concepts in these models. Among various models that have been proposed, Valiant's PAC learning model [Val84] and Angluin's exact learning model [Ang88] have received considerable attention. In both these models the concept classes are sets of subsets of finite strings over a finite alphabet along with a short representation for each of the sets (concepts) in the class. A concept class along with a representation for each concept is called a representation class. By learning a representation class we mean to find a representation for the concept of interest with limited access to it. Among the representation classes which have received much attention are DFAs, DNFs and CIRCUITS. See the article by Angluin [Ang92] for a survey on learning theory.

A major area of research in computational learning theory is the classification of different representation classes with respect to the difficulty of learning them in any reasonable learning model. In this direction many interesting results are known. In Angluin's model (this is the model of interest to us), one way to quantify the complexity of a representation class is to consider the type and number of queries that a learner has to ask the teacher in order to learn any concept in the representation class. The two types of queries that Angluin had introduced in her model are membership and equivalence queries. There have been successful attempts to capture the complexity of representation classes with respect to the type of queries needed to learn them in this model, by combinatorial properties like approximate fingerprints, polynomial certificates etc. [Ang90, BCG+96, HPRV96, Heg95]. Also, Watanabe [Wat90] used complexity-theoretic ideas to analyze the complexity of query learning. He defined the notion of *machine types* to capture various types of queries in Angluin's model (see also [WG94]).

Our focus is to further investigate the complexity of exact learning (henceforth, by learning we mean exact learning). Our approach towards classifying the complexity of learning representation classes is built on ideas from complexity theory. To motivate our study, consider the scenario where there are two representation classes both of which can be learned using polynomially many equivalence queries but cannot be learned using polynomially many membership queries. What further can we say about their learnability? It could be the case that one of the classes is easier to learn than the other because the full power of equivalence query may not be necessary to learn the former although it is required to learn the latter. To frame the above question more formally, we first give a brief introduction to Angluin's exact learning model.

Let $\mathcal{P}$ be a representation class (for example, the class CNF: in this case the boolean functions are the concepts of interest and the representation of a function is by conjunctive normal form formula). A learner, usually a deterministic Turing machine, has to output the representation of the concept fixed by the teacher. During the computation, the learner can ask two types of queries to the teacher about the concept of interest; membership queries and equivalence queries. To a membership query x asked, teacher gives YES/NO answer depending on whether x is in the concept or not. An equivalence query is a string y which is a representation for some concept in the class and the teacher answers YES if y represents the concept of interest or produces a string x in the symmetric difference of the concept of interest and the concept represented by y, as a counter example. An efficient learner is one which outputs a representation of the concept, in time polynomial in the length of the minimal representation of the concept.

Now consider the three representation classes; SYM of permutation groups (a subgroup G of S_n is represented by a generator set for G), LS(p) of linear spaces over finite fields (represented by a basis) and 3-CNFs (the exact definitions of these classes are given in Chapter 5). We have the following theorem on the complexity of learning these classes in Angluin's model.

Theorem *The classes* 3-CNF, SYM *and* LS(p) *are polynomial-time learnable with equivalence queries but not polynomial-time learnable with only membership queries.*

An immediate question that arises is whether we can say more about the learnability of these classes. Intuitively, we can expect that the algebraic classes SYM and LS(p) may be *easier* to learn than 3-CNF due to their inherent algebraic structure. Our goal is to investigate this possibility. We develop a new exact learning model, called the *teaching assistant* model of exact learning. This can be seen as a refinement of Angluin's model. The new ingredient in our model is the concept of a *teaching assistant*. This model allows us to make a finer classification of the complexity of exact learning than what is possible in Angluin's model.

The motivation for the definition of this model is the following. In Angluin's model of exact learning the learner communicates with the teacher through equivalence and membership queries in order to learn a concept. It is easy to show that an equivalence query can be replaced by a series of queries to an NP oracle where the machine accepting this oracle is a non-deterministic oracle Turing machine with access to the concept of interest (a similar result can be seen in [WG94]).[1] So, intuitively we can think of the NP oracle as an intermediate agent (a teaching assistant) between the learner and the teacher. To this teaching assistant the learner can make queries and the Turing machine accepting this assistant is allowed to make membership queries to the teacher. Hence informally, we can say that a representation class which is polynomial-time learnable (hereafter, we use the notation FP-learnable) using equivalence queries is also FP-learnable with an assistant in NP. Now, this idea can be extended to other well-studied complexity classes also; for example the

[1]The above statement is informal. In Chapter 5, after defining teaching assistants, we will give a more formal proof of this.

classes NP $\cap$ co-NP, UP, SPP, LWPP etc. This immediately gives a framework for comparing the complexity of exactly learning various representation classes.

The second part of the thesis spreads over into two chapters, Chapters 5 and 6. In Chapter 5, one of our main contributions is the precise definition of the new teaching assistant model and that of FP-learnability using various assistant classes. The teaching assistant classes that we define here are the ones analogous to the classes P, NP Σ_2^p, SPP and LWPP. After giving definitions of all the notions involved in our model, we formulate the notion of learning any representation class efficiently with assistants from a teaching assistant class $\mathcal{C}$ (FP-learnability with a $\mathcal{C}$-assistant). The main result we show here is stated as the following theorem.

Theorem *The representation class* SYM *is* FP*-learnable with an* LWPP*-assistant. The representation class* LS(p) *over any fixed prime p is* FP*-learnable with an* SPP*-assistant. The class 3-*CNF *is not* FP*-learnable with an* LWPP*-assistant* (SPP*-assistant*) *unless* NP $\subseteq$ LWPP (*respectively* NP $\subseteq$ SPP).

This theorem illustrates the possibility of a finer classification of the complexity of exact learning than what is possible in Angluin's model. For proving the upper bounds, we make use the algebraic structure of these representation classes. In particular, in the case of representation class of permutation groups, we make use of the properties of special generator sets called *strong generator sets* for any permutation group.

In Chapter 6, we further investigate the fine inclusion structure that is possible among various teaching assistant classes. In particular, we consider the FP-learnability of some subclasses of SYM, with teaching assistants from assistant classes UP$\cap$co-UP, UP and NP$\cap$co-NP. We show upper bounds and absolute lower bounds on the learnability of these representation classes.

1.5 Organization of the Thesis

This thesis consists of seven chapters. In Chapter 1, we have already seen the basic motivation for studying the complexity of computational problems over finite groups. Chapter 2 consists of the necessary notations and definitions which we use throughout the thesis; both from complexity theory and group theory. Most of these are standard. At the end of Chapter 2 we prove a lemma (Lemma 2.0.2) which is the basic complexity-theoretic technique (both the lemma and the proof method) we use in almost all our upper bound proofs. So a good understanding of this lemma as well as the proof will considerably aid in understanding most of the upper bound proofs.

Chapters 3 and 4 contain the first part of the thesis. In Chapter 3, we give a very brief survey of computational group theory. Chapter 4 is devoted completely for proving the upper bound of SPP for the three basic computational problems over solvable black-box groups; namely Membership Testing, Order Verification and Group Isomorphism. The proof of this upper bound is built on the fundamental theorem of finite abelian groups (Theorem 4.2.1). The proof of the fundamental theorem given in [Bur55] is recommended for easily understanding the proof of our upper bound result. A proof of Lemma 4.2.6, which is the basic ingredient of the proof of the fundamental theorem, is given in the Appendix. Though the upper bounds we prove here are for solvable groups, we devote a large part of this chapter for dealing with abelian groups. This helps in understanding the basic ideas involved in the proofs for solvable groups.

Chapters 5 and 6 contain the second part of the thesis. These can be read independently of Chapters 3 and 4. In the first two sections of Chapter 5, we develop the teaching assistant model of learning. Naturally, these sections are basic requirements for the rest of the results in Chapters 5 and 6. Among all the learning algorithms that we design in Chapters 5 and 6 , the design of the learning algorithm for the class of permutation groups, SYM, is the most involved (Theorem 5.5.1).

For proving the correctness of this algorithm, the notion of a strong generator set for a permutation group and its properties are crucial.

Finally in Chapter 7, we conclude the thesis and state some open questions that arise from this investigation.

Much of the group theory (and some linear algebra) that we use are elementary and can be derived fairly easily from basic definitions. But, for completeness sake, in the Appendix we give proofs for most of the group-theoretic and linear-algebraic results we use in this thesis.

Chapter 2

Preliminaries

In this chapter we give the necessary notations and basic definitions that we use throughout the thesis.

Complexity-theoretic Notations and Definitions

We fix the finite alphabet $\Sigma = \{0,1\}$. Σ^* denotes the set of strings over Σ. A subset of Σ^* is called a language. Σ^n ($\Sigma^{\leq n}$) denotes the set of strings over Σ of length n (respectively, $\leq n$). λ denotes the empty string. For an $x \in \Sigma^*$, $|x|$ denotes the length of x. For $y_1, y_2 \in \Sigma^*$ $y_1 \leq y_2$ denotes that y_1 is lexicographically smaller than or equal to y_2. For any finite set X, $|X|$ denotes the cardinality of X. Note that we are using the same notation for the cardinality of a set and length of a binary string. The meaning of the notation will be clear from the context. For $L \subseteq \Sigma^*$, $0L$ (1L) denotes the language obtained by prefixing 0 to all the strings in L (respectively 1). For sets A and B, symmetric difference of A and B is denoted by $A \triangle B$. The base-2 logarithm function is denoted as log. We use some standard pairing functions, denoted by $\langle \cdot, \cdot \rangle$ which can be computed and inverted in polynomial time. $\mathbf{Z}$ and $\mathbf{N}$ denote the set of integers and natural numbers respectively. Let $X = \{x_1, \ldots, x_n\}$ be an ordered set. Then $(l_{x_1}, \ldots, l_{x_n})$ denotes a formal tuple indexed by set X. Some times we denote such a tuple by $(l_x | x \in X)$.

Next we very briefly describe the framework of computation we will be working with. For detailed definitions, we refer the reader to standard text books like [BDG88, Pap94].

The standard Turing machine will be our basic model of computation. We will consider both deterministic and nondeterministic Turing machines. Sometimes we allow Turing machines to have access to an oracle. In this case Turing machines will have an additional write-only tape and a special state called QUERY-state, which are used in a standard way to access the oracle. We assume the familiarity with the standard complexity classes such as P, NP, PH, FP, #P. Refer [Pap94] for the definitions of these classes. For any complexity class $\mathcal{L}$, Co-$\mathcal{L}$ denotes the class of languages whose complement is in $\mathcal{L}$.

Fenner et. al., in [FFK94] defined the class of functions GapP as the closure of Valiant's class #P under subtraction. A function $f : \Sigma^* \to \mathbf{Z}$ is *gap-definable* if there is a polynomial-time nondeterministic (in short, NP) machine M such that, for each $x \in \Sigma^*$, $f(x)$ is the difference between the number of accepting paths (denoted by $acc_M(x)$) and the number of rejecting paths (denoted by $rej_M(x)$) of M on input x. For each NP machine M let gap_M denote the gap-definable function defined by it. Let GapP denote the class of gap-definable functions. It is easy to see that GapP is the closure of #P under subtraction. The class GapP enjoys nice closure properties. Most of these closure properties have been shown in [FFK94]. Many of the counting classes that have been studied in the literature have equivalent characterizations using gap-definable functions.

Now we give the definitions of three counting classes which are of main importance to us.

Definition 2.0.1 A language L is in

- UP if there is an NP machine M such that: $x \in L$ implies that $acc_M(x) = 1$, and $x \notin L$ implies that $acc_M(x) = 0$.

- SPP if there is an $f \in$ GapP such that: $x \in L$ implies that $f(x) = 1$, and $x \notin L$ implies that $f(x) = 0$.
- LWPP if there are functions $f \in$ GapP and $h \in$ FP such that: $x \in L$ implies that $f(x) = h(0^{|x|})$, and $x \notin L$ implies that $f(x) = 0$.

It follows from the definitions that UP $\subseteq$ SPP $\subseteq$ LWPP.

For the sake of completeness, we give definitions of PP, $\mathrm{C_{=}P}$ and $\mathrm{Mod}_k\mathrm{P}$. A language L is in PP if there is a GapP function f such that: $x \in L$ iff $f(x) > 0$. A language L is in $\mathrm{C_{=}P}$ if there are functions $f \in$ GapP and $h \in$ FP such that: $x \in L \Leftrightarrow f(x) = h(0^{|x|})$. A language L is in $\mathrm{Mod}_k\mathrm{P}$ if there is function $f \in$ GapP such that $x \in L \Leftrightarrow f(x) = 0 \pmod{k}$.

For all the above-mentioned classes, their relativized versions can be defined by allowing the nondeterministic machines accepting the languages in the class to have access to an oracle. If M is an NP machine which accesses an oracle A, then the corresponding *gap* function is denoted by gap_{M^A}. For any class $\mathcal{C}$ which is relativizable, $\mathcal{C}^A$ denotes the relativized version of $\mathcal{C}$ with respect to the set A. Then a language L is said to be *low* for $\mathcal{C}$ if $\mathcal{C}^A = \mathcal{C}$.

Group Theory: Notations, Definitions and Basic Results

Here we give some notations and basic definitions from group theory. We also describe some basic results. For further results and their proofs, please refer to standard textbooks [Bur55, Hal59].

A *group* is a tuple $\langle G, *\rangle$ where G is a nonempty set and $*$ is a binary operation (we call this operation "product") on G satisfying the following properties. G is closed under $*$. The operation $*$ is associative. There is an element $e \in G$, called the *identity* of G, such that $x * e = e * x = x$, for all $x \in G$. For every $x \in G$ there exists a unique $x^{-1} \in G$, called the *inverse* of x, such that $x * x^{-1} = x^{-1} * x = e$.

When there is no ambiguity we do not explicitly specify the group operation and denote the product $x * y$ by xy. Also, by abusing notation, henceforth we denote the group $\langle G, * \rangle$ by just G.

Let G be a group. A subset H of G is called a *subgroup* of G (denoted $H < G$ or $G > H$) if H is a group under the group operation of G. For a subset S of G, the smallest subgroup of G containing S is called the group *generated by* S and is denoted by $\langle S \rangle$ (note that we are using the same notation for the pairing function also. The exact meaning of the notation will be clear from the context). This group is the same as the set of all finite products of elements from S. A subset S of G is a generator set for G if G is identical to $\langle S \rangle$. A group G is *finite* if the cardinality of the set G is finite. In this thesis, we are only interested in finite groups. Henceforth by a group we mean a finite group. The *order* of G is defined as the cardinality of the set G and is denoted by $|G|$. A group is said to be cyclic if it is generated by a single element. For an element $g \in G$, the *order* of g (denoted as $o(g)$) is the order of the cyclic subgroup generated by g. This is the same as the smallest positive integer k such that $g^k = e$, (g^k denotes the product of g, k times) where e is the identity of G.

A fundamental theorem in finite group theory, due to Lagrange, states that if $H < G$ then $|H|$ divides $|G|$. This theorem has a large number of algorithmic applications. For example, it follows from Lagrange's theorem that any finite group G is generated by a set of group elements of cardinality bounded by $\log |G|$.

Let H be a subgroup of a group G. For $g \in G$ the set $\{hg \mid h \in H\}$, denoted by Hg, is called a *right coset* of H in G. Similarly, the set $gH = \{gh \mid h \in H\}$ is called a *left coset* of H in G. H is a *normal* subgroup of G if for all $g \in G$ it holds that $Hg = gH$. A fundamental result in the theory of groups is that if H is a normal subgroup of G, then the set of right cosets of H in G forms a group (called the *factor group* or a *quotient group* induced by H and denoted by G/H) under the binary operation $\cdot$ defined as $Hx \cdot Hy = Hxy$. The identity element of this group is the coset $He = H$. For a set $X \subseteq G$, the *normal closure* of X is the smallest normal subgroup containing X.

Two groups G and H are said to be isomorphic if there is a bijection (set-theoretic) ϕ from G to H such that for any $x, y \in G$, $\phi(xy) = \phi(x)\phi(y)$. Isomorphism preserves all structural properties of groups.

Let p be a prime. A p-group is a finite group whose order is a power of p. Let G be finite group such that $|G| = p_1^{e_1} p_2^{e_2} \dots p_r^{e_r}$. The existence of subgroups in G which are p-groups, is given by Sylow's theorem. That is, for each i there is a subgroup of G of order $p_i^{e_i}$. A subgroup of G of order $p_i^{e_i}$ is referred to as a p_i-*Sylow subgroup* of G.

Let $(X, *)$ and $(Y, .)$ be two groups. The *direct product* of the groups X and Y is defined as the group $(X \times Y, \circ)$, where $X \times Y$ is the cartesian product of sets X and Y, and for $(x_1, y_1), (x_2, y_2) \in X \times Y$ their $\circ$ composition is defined as $(x_1, y_1) \circ (x_2, y_2) = (x_1 * x_2, y_1.y_2)$. Let H, K be subgroups of a group G. Suppose that H is normal in G and the set $\{xy \mid x \in H, y \in K\} = G$. Then G is isomorphic to the direct product $H \times K$.

Solvable groups

Here we give the definition of a *solvable* group and state some properties of them. Intuitively solvable groups can be thought of as a generalization of *abelian groups*. A group G is *abelian* if $\forall x, y \in G : xy = yx$; that is $xyx^{-1}y^{-1} = e$. In general, the element $xyx^{-1}y^{-1}$ is called the *commutator* of elements x and y in G. The subgroup of G generated by the set $\{xyx^{-1}y^{-1} \mid x, y \in G\}$ is called the *commutator subgroup* of G. We denote this subgroup by G'. Observe that if G is abelian, G' is the trivial group containing only the identity element. The commutator subgroup G' is actually a normal subgroup of G and the factor group G/G' is abelian. For a group G, the sequence $G = G_0 > G_1 > \dots$ is called the commutator sequence, where each group G_i is the commutator subgroup of G_{i-1}. G is solvable if the commutator sequence terminates in the trivial subgroup $\langle e \rangle$ in finitely many steps. This intuitively means that any solvable group can be decomposed into a series of abelian factor groups.

Solvable groups form a large subclass of all finite groups. In fact, a celebrated result due to Fiet and Thompson says that any finite group of odd order is solvable. Any subgroup of a solvable group is solvable. It follows from Lagrange's theorem that, if G is solvable, then length of the commutator sequence is bounded by $\log |G|$. From a computational viewpoint this fact is very useful. It also holds that two solvable groups G and H are isomorphic if and only if the factor group H_{i-1}/H_i is isomorphic to G_{i-1}/G_i for all i. Here H_i (G_i) is the i^{th} element in the commutator series of H (respectively, G).

A Complexity-theoretic Technique

The main complexity-theoretic technique that we use for showing membership in SPP is the following lemma proved in [KST92]. In fact, in [KST92] a more general version of this lemma is proved. For our purposes the following is enough. For the statement of this lemma, we first give some definitions.

Let M be an oracle NP machine and let $A \in$ NP be accepted by an NP machine N. We say that a query y made by M is *1-guarded* for N if $N(y)$ has at most 1 accepting path. We say that M makes 1-guarded queries to A, if there exists an NP machine N accepting A such that on all the inputs x to M, the queries made by M to A are 1-guarded for N.

Lemma 2.0.2 [KST92] *Let M be a nondeterministic polynomial-time oracle machine that makes 1-guarded queries to $A \in$ NP. Then the function $gap_{M^A}(x)$ is in* GapP.

Proof. Let x be an input to M which makes 1-guarded queries to $A \in$ NP. Let N be the corresponding NP machine accepting A. For a computation path ρ of M, define $v(\rho) = 1$ if ρ is accepting and $v(\rho) = -1$ if ρ is rejecting. In this notation the gap function defined by M is $gap_{M^A}(x) = \Sigma_\rho v(\rho)$. Let q be the polynomial bounding the number of queries asked by M. Without loss of generality, we can assume that on any input x, M makes exactly $q(|x|)$ queries. Design a machine M' as follows. M' on input x guesses a computation path ρ of M and simulates M on ρ. For the i^{th} query y_i, made by M, M' guesses $a_i \in \{0, 1\}$ as an answer to the query (this way M' avoids access to the oracle) and continues simulation of M on ρ, treating a_i as the answer to the query y_i of M. At the end of the computation on ρ, M' produces the following *gap*.

$$gap \;=\; v(\rho)\Pi_{i=1}^{q(|x|)}(a_i gap_N(y_i) + (1 - a_i)(1 - gap_N(y_i))) \qquad (2.1)$$

We argue that $gap_{M'}(x) = gap_{M^A}(x)$ as follows. Call those computation paths of M', where all the guessed answers q_i to the queries y_i are correct ($a_i = 1$ if and only if $y_i \in A$), *good*. Observe that corresponding to any computation path ρ of M, there is exactly one computation path ρ' of M' which is *good*. We show that, on the *good* paths, M' produces a $gap = v(\rho)$ and on all the computation paths which are not *good*, M' produces a $gap = 0$. From this and the observation that for any computation path ρ of M, there is exactly one computation path ρ' which is *good*, it will follow that $gap_{M'}(x) = gap_{M^A}(x)$.

Consider a path ρ' of M' which is *good*. Since ρ' is *good*, all the queries constructed by M' will be same as the queries made by M on the path ρ and hence will be 1-guarded for M. Consider the i^{th} term in the product of RHS of (2.1). If $a_i = 1$, then $y_i \in A$ and $gap_N(y) = 1$ (since the query is 1-guarded for N). If $a_i = 0$ then $y_i \notin A$ and $gap_N(y) = 0$. Hence on the *good* paths, M' produces a $gap = v(\rho)1^{q(|x|)} = v(\rho)$.

Now we shall consider the gap produced by M' on paths which are not *good*. Let i' be the first value of i for which M' guesses a_i which is not correct. This means that $a_{i'} = 0$ if an only if $y_{i'} \in A$. Here we make a crucial observation that all the queries y_j for $j \leq i'$, computed by M', as queries made by M to A are 1-guarded for N (the queries y_k computed by M' for $k > i'$ may not be 1-guarded for N since the answer to query $y_{i'}$ is guessed wrong by M'). Now if $a_{i'} = 1$, $gap_N(y_{i'}) = 0$ and hence $a_i gap_N(y_i) + (1 - a_i)(1 - gap_N(y_i)) = 0$. Similarly if $a_{i'} = 0$, $gap_N(y_{i'}) = 1$ and $a_i gap_N(y_i) + (1 - a_i)(1 - gap_N(y_i)) = 0$. In any case, contribution of the i'^{th} term in the product of RHS of 2.1 is 0 and hence the total gap produced by M' on ρ is 0. This concludes the proof of the lemma. ■

The above lemma is useful in showing certain problems to be in the class SPP. For example, to show that a language B is in SPP, it is enough to show that there exists a polynomial-time deterministic machine M and a language $A \in$ NP such that M makes 1-guarded queries to A and accepts B. To see this, observe that the machine M can be very easily modified to get a machine M' such that M' will have a gap=1 if $x \in B$ and will have a gap=0 otherwise. Now we can apply the above lemma directly to get $B \in$ SPP. For all the upper bound proofs in Chapter 4, we will be explicitly using this method for proving membership in SPP. On the other hand, in Chapter 5, we will essentially follow the line of proof of the above lemma for showing upper bound on learning linear spaces using an SPP-assistant. The above lemma can be generalized for getting a sufficient condition for membership in the class LWPP. This is done essentially by generalizing the notion of 1-guarded queries to f-guarded queries, where f is a function on natural numbers. In fact, a slightly more complicated adaptation of the above proof is used for proving upper bound on the learnability of permutation groups using an LWPP-assistant in Chapter 5. By slightly modifying the above lemma we can also show that SPP is low for the classes PP, $\mathrm{C_=P}$ and $\mathrm{Mod}_k\mathrm{P}$.

Chapter 3

Computational Group Theory

In this introductory chapter we give a brief survey of some results from computational group theory. Our main focus will be on the complexity of group-theoretic problems. We are interested when the groups involved in the problem instances are specified by generator sets.

Research in computational group theory centers mainly on problems concerning *permutation groups* and *matrix groups*. Among the basic problems that have been studied are membership testing, order computation, normal closure computation, computing structural elements like composition series, center, Sylow subgroups etc[1].

Much of the research in the area has gone into developing efficient algorithms for permutation groups (these are the subgroups of the symmetric group). For this class of groups many computational problems have polynomial time solutions. Central in many of these algorithms is the notion of a *strong generator set* for a permutation group introduced by C. Sim [Sim70]. The importance of this notion is mainly because of the fact that many basic problems like membership testing, order computation and computing the normal closure are efficiently reducible to the computation of a strong generator set. Sim [Sim70] gave a method for constructing a strong generator set from an arbitrary generator set. A variant of Sim's algorithm was shown to be in

[1]We omit the formal definitions of these problems. Please see [KL90] for definitions of these problems in the context of permutation groups.

polynomial time by Furst et. al. in [FHL80]. This gives polynomial-time algorithms for membership testing, order computation and computing the normal closure.

The problem of computing the composition series and center for permutation groups was shown to be in polynomial time by E. Luks [Luk87]. The analysis of Luks' algorithm uses many deep permutation group-theoretic results along with a detailed knowledge of the classification of finite simple groups. The problem of computing a generator set of a Sylow p-subgroup for prime p, was shown to be in polynomial time by W. Kantor [Kan85]. A library of polynomial time algorithms for a long list of problems over permutation groups (even for factor groups) is given in [KL90]. In [BLS87], Babai et. al. have shown that membership testing, order computation, computing the center and the composition series can be performed in NC. These results also depend on consequences of the classification of finite simple groups for their analysis.

Though there are efficient algorithms for a large number of problems over permutation groups, the scenario is quite different in the case of matrix groups over finite fields (these are subgroups of the group of invertible matrices over a finite field). The problem of testing membership for 1×1 matrix groups over finite fields is the decision version of the discrete logarithm problem, for which no polynomial time algorithm is known. The techniques developed for fast management of permutation groups will not work for matrix groups. The main reason for this is that, while permutation groups act on a set of "small" size (the subgroups of S_n act on a set of size n), the action of $n \times n$ invertible matrices over a finite field is on a vector space of size *exponential* in n. The polynomial-time algorithms for testing solvability and nilpotence of matrix groups over finite fields, due to Luks [Luk92], is an important result in this area.

From the point of view of complexity theory, these problems over matrix groups are interesting since the exact complexity of many of these problems are not characterized. In order to study the complexity of these hard problems in a general setting

Babai and Szemerédi [BS84] introduced the notion of black-box groups. The main motivation was to avoid the the actual representation of group elements while investigating the complexity of group-theoretic problems. Intuitively in this framework, the group elements are uniformly encoded as binary strings and the group operations are performed by a group oracle. Hence an upper bound for a computational problem in this framework gives the same upper bound for the problem over groups for which the operations can be done efficiently.

Let us consider the complexity of testing membership in black-box groups. Notice that it is not obvious whether the problem is in NP. Naive idea of guessing a product over generators does not work. But in [BS84] it is shown that for any finite group G, an element of G can be "reached" using a short straight line code over an arbitrary generator set of G. From this reachability result it easily follows that membership testing over general black-box groups is in NP.

Verification of the order (decision version of computing the order) is a central problem when we are interested in the complexity of group theoretic problems; many problems reduce to order verification [Bab92]. For example, it is easy to see that both membership and non-membership testing reduces to verification of order. For matrix groups over finite fields, it is believed that order verification is in NP. In [BS84] it is shown that, over solvable matrix groups, order verification in NP $\cap$ co-NP. This puts membership testing in NP $\cap$ co-NP for this class of groups. This is a strong indication that membership testing over solvable matrix groups cannot be complete for NP (unless NP = co-NP).

In [Bab92], using sophisticated combinatorial and group-theoretic techniques, it is shown that order verification over general black-box groups is in the class AM $\cap$ co-AM [Bab85]. This places membership testing over general black-box groups in NP$\cap$co-AM. Since AM$\cap$co-AM is known to be low for the second level of PH [Sch88], it is unlikely that the problems membership testing and order verification are hard for NP. In [Bab92], using this above-mentioned upper bound result, similar upper bounds were shown for a large number of problems over black-box groups. For example, it is shown that isomorphism testing over black-box groups is in AM.

Randomization has also played an important role in computational group theory. The notion of *random normal subproduct* introduced in [BCF$^+$95] has given rise to an efficient randomized algorithm for computing normal closures over black-box groups. This, along with random subproducts [BCF$^+$95] gives a co-RP algorithm for testing solvability (see also [CF93]).

There are computationally difficult problems concerning permutation groups also. Group intersection, group factorization and coset intersection are some among them (see [KST92] for exact definitions of these problems). There are no polynomial time algorithms known for these problems. On the other hand these problems are in NP $\cap$ co-AM [Bab92] and hence are not complete for NP unless the polynomial-hierarchy collapses. The counting complexity of these problems were first investigated in [KST92]. It is shown in [KST92] that group intersection is in SPP while group factorization and coset intersection (along with Graph Isomorphism), belongs to LWPP. Hence these problems are low for the counting classes PP and $C_=P$. These results shows the similarity of the complexity of these problems to that of Graph Isomorphism. There are also some problems over permutation groups that are NP-complete. One interesting example is the problem of deciding whether a group contains an element which is fixed-point free [Lub81].

In the next chapter we investigate the counting complexity of three basic problems; membership testing, order verification and isomorphism testing, over black-box groups. We show that over solvable black-box groups these problems are in SPP. This result has appeared in [AV97b, Vin97]. In [AV97b, Vin97], it is also shown that over solvable black-box groups, the problems group intersection, group factorization and coset intersection are in LWPP.

Chapter 4

Solvable Black-box Group Problems

4.1 Introduction

This chapter is devoted to showing the membership of some basic problems from computational group theory in the class SPP. For the sake of completeness, we once again give the exact definitions of problems of interest to us.

Let $\mathcal{B} = \{B_m\}_{m>0}$ be a group family. The problems of interest to us are the following:

Membership Testing $\triangleq \{\langle m, S, g\rangle \mid \langle S\rangle < B_m \text{ and } g \in \langle S\rangle\}$.
Order Verification $\triangleq \{\langle m, S, n\rangle \mid \langle S\rangle < B_m \text{ and } |\langle S\rangle| = n\}$.
Group Isomorphism $\triangleq \{\langle m, S_1, S_2\rangle \mid \langle S_1\rangle \text{ and } \langle S_2\rangle \text{ are isomorphic subgroups of } B_m\}$.

We show that these problems over any group family are in the class SPP when restricted to solvable groups. For showing this upper bound, we design appropriate deterministic oracle algorithms accepting the above languages which make only 1-guarded queries to a language in NP. Then by Lemma 2.0.2 proved in Chapter 2 we get the desired upper bound.

Remark. Note that we are considering these problems when the groups involved in the definitions of the problems are solvable. Although there is no deterministic polynomial time algorithm known for Solvability Testing over arbitrary black-box groups, later in this chapter we show that Solvability Testing is in SPP. Hence, without loss of generality we can assume that the groups encoded in the problem instances are solvable.

Let $\mathcal{B} = \{B_m\}_{m>0}$ be a group family. Recall the definition of a solvable group. A group G is said to be solvable if the commutator sequence $G = G_0 > G_1 > \ldots$ terminates in the trivial subgroup $\langle e \rangle$ in finitely many steps. Let k be the length of this series. From Lagrange's theorem, it follows that k is bounded by a polynomial in m. Now, since G_{i-1}/G_i is abelian for all $i \leq k$, we can informally say that any solvable subgroup of B_m can be viewed as a short series of abelian factor groups. So intuitively it is clear that abelian groups are the basic building blocks of solvable groups and it is natural that tight upper bounds on abelian group problems may lead to upper bounds for solvable groups. In view of this observation, in the next section we concentrate on oracle algorithms for abelian groups. As an illustration, at the end of the next section we give an upper bound of SPP for the problems over abelian groups. Then, we extend these upper bounds to more general class of solvable groups.

4.2 Managing Abelian Groups

For dealing with abelian groups, we use a constructive version of the fundamental theorem about the structure of finite abelian groups. This theorem completely characterizes any finite abelian group up to isomorphism. We next state this theorem. Proof of this theorem can be seen in any standard book in group theory (see for example [Hal59, Bur55]).

Theorem 4.2.1 [Bur55] *Let G be a finite abelian group such that $|G| = p_1^{e_1}p_2^{e_2}\ldots p_r^{e_r}$, where the p_i's are distinct primes.*

1. *The group G can be expressed as the direct product of its Sylow subgroups $S(p_1), S(p_2), \ldots, S(p_r)$ where $|S(p_i)| = p_i^{e_i}$ for $1 \leq i \leq r$.*
2. *For $1 \leq i \leq r$, each Sylow subgroup $S(p_i)$ can be uniquely expressed as the direct product of cyclic groups of orders $p_i^{e_{i1}}, p_i^{e_{i2}}, \ldots, p_i^{e_{is_i}}$ such that $e_{i1} \geq e_{i2} \geq \ldots \geq e_{is_i}$ and $\Sigma_{j=1}^{s_i} e_{ij} = e_i$.*
3. *This decomposition of G is unique up to isomorphism.*

The above theorem implies that, for any abelian p-group G of order p^e, there is a *unique* sequence of natural numbers $e_1 \geq e_2 \geq \ldots \geq e_m$ such that $\Sigma_{1\leq j\leq m} e_j = e$ and G can be expressed as a direct product of m cyclic groups of respective orders p^{e_j}, $1 \leq j \leq m$. The sequence $(e_1, e_2, \ldots, e_m)$ is called the *type* of the p-group G. Since any abelian group can be decomposed into its Sylow subgroups (Theorem 4.2.1, we can extend the definition of the type to an arbitrary abelian group.

A consequence of Theorem 4.2.1 is the existence of special kind of generator sets, called *independent* generator sets, for any abelian group. Let G be an abelian group which is a direct product of n cyclic groups $C_1, C_2, \ldots, C_n$ generated by elements $g_1, g_2, \ldots, g_n$ respectively. Then the set $S = \{g_1, g_2, \ldots, g_n\}$ is a generator set for G. Since G is the direct product of $C_1, C_2, \ldots, C_n$, it follows that for any $g \in S$, $\langle g \rangle \cap \langle S - \{g\} \rangle = \{e\}$. We call a generator set with this property an independent generator set. Formally, we have the following definition.

Definition 4.2.2 Let G be a finite abelian group. An element $g \in G$, $g \neq e$, is said to be *independent* of a set X of elements of G if $\langle g \rangle \cap \langle X \rangle = \{e\}$. A generator set S of G is an *independent generator set* for G, if every $g \in S$ is independent of $S - \{g\}$.

As a consequence of Theorem 4.2.1, all finite abelian groups have independent generator sets. Also, the size of the generator set is logarithmic in the size of the group. Now we prove a key group-theoretic proposition about independent generator sets. For stating this, first we give some definitions.

Let G be an abelian group generated by a set $S = \{g_1, \ldots, g_k\} \subseteq G$ (in the sequel, we shall assume that the elements of a generator set are ordered according to some fixed order. Wherever necessary, we will prescribe the order). Since G is abelian, for any element $g \in G$, there is an ordered tuple of $(l_1, \ldots, l_k)$ (we call this tuple exponents), $1 \leq l_i \leq o(g_i)$ for all $1 \leq i \leq k$ such that, $h = \prod_{i=1}^{k} g_i^{l_i}$. In most of our algorithms in this chapter, a major computation involved is that of exponents. In some places such computations have to be carried out for abelian factor groups also. Motivated by this, we give the following definition.

Definition 4.2.3 Let G be an abelian group generated by a set $S \subseteq G$. Let $X \subseteq S$. Then for $g \in G$, a tuple $(l_x | x \in X)$ is called an *X-exponent* of g with respect to S, if for all $x \in X$, $1 \leq l_x \leq o(x)$ and there exists indices $\{l_y \mid y \in S - X\}$ such that $g = \prod_{s \in S} s^{l_s}$.

In most places the set S of the above definition will be clear from the context. In this case we omit the reference to S and simply say *X-exponent*.

Now we prove a proposition which guarantees the uniqueness of exponents for independent generator sets. This property of independent generator sets is crucial for us.

Proposition 4.2.4 *Let G be an abelian group and S be an independent generator set for G. Then for any $g \in G$ and $X \subseteq S$, the X-exponent of g is unique.*

Proof. Notice that it is enough to show that for any $g \in G$, the S-exponent is unique. Then it will follow from the definition that, for any $X \subseteq S$, the X-exponent of g is also be unique. Let $S = \{g_1, \ldots, g_n\}$. Suppose for some $g \in G$ the S-exponent of g is not unique. Let $(l_1, \ldots, l_n)$ and $(l'_1, \ldots, l'_n)$ be two S-exponent of g such that

$l_i \neq l'_i$. Then $g = g_1^{l_1} \dots g_n^{l_n} = g_1^{l'_1} \dots g_n^{l'_n}$. Since G is abelian, this implies that $g_i^{l_i - l'_i} \in \langle S - \{g_i\} \rangle$ which is a contradiction to the fact that S is independent since $g_i^{l_i - l'_i} \neq e$. ■

The usefulness of independent generator sets comes from the above proposition. To make this clear, suppose $S \subseteq B_m$ be a generating set for an abelian group. Also, assume that the orders of all the elements in the set S are given. Then, it is easy to design an NP machine which on input $\langle m, S, g \rangle$ tests whether $g \in \langle S \rangle$. The machine basically guesses an S-exponent of g and verifies in polynomial time that the guess is correct. Now if S is an independent generator set then the NP machine will have a *unique* accepting path if and only if $g \in \langle S \rangle$. This observation along with the fundamental theorem (Theorem 4.2.1) indicates that computing an independent generator set and the type of the abelian group may be sufficient for all the three problems of our interest. Hence our focus is to design a deterministic oracle algorithm for this computation. This algorithm is allowed to use an NP language as oracle with the promise that the queries made to this oracle are 1-guarded. The main functions of this NP language will be to aid the algorithm for exponent and order computations. Notice that the exponent computation is at least as hard as computing the discrete logarithm. The key factor that helps to make 1-guarded queries to these NP languages, is the above proposition.

In the next subsection, we concentrate on the construction of an independent generator set for an abelian group from an arbitrary generator set. For application to solvable groups, we will actually be interested in the construction of an independent generator set for abelian *factor* groups. But, for clarity of presenting the methods involved, we first give a detailed design of a deterministic oracle algorithm (we shall call it INDEPENDENTGENERATOR) which takes an arbitrary generator set for an abelian group and converts it into an independent generator set by making 1-guarded queries to an NP language. In next section, we explain how we can modify INDEPENDENTGENERATOR to incorporate the construction of an independent generator set for abelian factor groups.

4.2.1 Constructing an Independent Generator Set

This subsection is devoted to the proof of the following theorem.

Theorem 4.2.5 *Let $\mathcal{B} = \{B_m\}_{m>0}$ be a group family. Then there exists a deterministic oracle algorithm* INDEPENDENTGENERATOR *and a language $L_{ig} \in$ NP such that* INDEPENDENTGENERATOR *takes $\langle m, S\rangle$ as input, where $S \subseteq B_m$, and L_{ig} as oracle. It outputs $|\langle S\rangle|$, types of each Sylow subgroup of $\langle S\rangle$ and a set of independent generators for $\langle S\rangle$ if $\langle S\rangle$ is abelian and outputs* NOT ABELIAN *if $\langle S\rangle$ is not abelian. Furthermore,* INDEPENDENTGENERATOR *runs in time polynomial in $|\langle m, S\rangle|$, and makes only 1-guarded queries to L_{ig}.*

As mentioned before, for the construction of an independent generator set, we use a constructive version of the Theorem 4.2.1. So, before we go into the details of the design of INDEPENDENTGENERATOR, we briefly explain the steps involved in the proof of the fundamental theorem. The first part of the theorem follows from the Sylow theorems. The proof of the second part, is essentially the proof of the following technical lemma. This lemma gives a greedy strategy to compute an independent generator set from an arbitrary set of generators for any abelian p-group. A proof of this lemma is given in the appendix at the end of the thesis.

Lemma 4.2.6 ([Bur55]) *Let G be a finite abelian p-group. Let $g_1, g_2 \dots, g_i$ be i independent elements of G of orders $p^{m_1}, p^{m_2}, \dots, p^{m_i}$ respectively such that for all j, $1 \leq j \leq i$, the maximum order of any element in the factor group $G/\langle\{g_1, g_2 \dots, g_j\}\rangle$ is $p^{m_{j+1}}$. Let $g'_{i+1}\langle\{g_1, g_2 \dots, g_i\}\rangle$ be an element in the factor group $G/\langle\{g_1, g_2 \dots, g_i\}\rangle$ of order $p^{m_{i+1}}$. Further, let $(x_1, x_2, \dots, x_i)$ be the unique $\{g_1, g_2 \dots, g_i\}$-exponent of $(g'_{i+1})^{p^{m_{i+1}}}$. Then $p^{m_{i+1}}$ divides x_j for $1 \leq j \leq i$. Let $y_j = x_j/p^{m_{i+1}}$ for $1 \leq j \leq i$. Then $g_{i+1} = g'_{i+1} g_1^{-y_1} g_2^{-y_2} \dots g_i^{-y_i}$ is an element of G of order $p^{m_{i+1}}$ which is independent of $\{g_1, g_2 \dots, g_i\}$.*

The first step for computing an independent generator set in INDEPENDENT-GENERATOR, is the construction of generator sets for all the Sylow subgroups of $\langle S\rangle$. Once we have the generator sets for each of the Sylow subgroups of $\langle S\rangle$, then we can use the method given by Lemma 4.2.6 to convert each of them into independent generator sets. The union of the independent generator sets of its Sylow subgroups forms an independent generator set for the group $\langle S\rangle$.

The next group-theoretic lemma shows how to construct generator sets for the Sylow subgroups from a generator set of an abelian group. We give a proof of this in the appendix.

Lemma 4.2.7 *Let G be a finite group of order n and let $p_1^{e_1}p_2^{e_2}\dots p_r^{e_r}$ be the complete prime factorization of n. Let H be an abelian subgroup of G generated by the set $\{g_1, g_2 \dots, g_s\} \subseteq H$. Then for each j, the p_j-Sylow subgroup of H is generated by $X_j = \{g_i^{(n/p_j^{e_j})} \mid 1 \le i \le s\}$.*

From the statement of Lemma 4.2.6, it is clear that the two major computations involved in the construction of an independent generator set are the exponent and the order computations. We use a language $L_{ig} \in$ NP precisely to aid INDEPENDENTGENERATOR for these computations. Before we give the exact definition of the language L_{ig}, we state a group-theoretic proposition which will give us a method for computing the order. Since we will be interested in computing the order in factor groups, we state the proposition for factor groups. See the appendix for a proof.

Proposition 4.2.8 *Let G, H and K be finite groups such that $H, K < G$ and K is a normal subgroup of H. Let $|G| = n = p_1^{e_1}\dots p_r^{e_r}$ be the unique prime factorization of the order of G. Then for any element hK in H/K, the order $o(hK)$ is of the form $p_1^{d_1}\dots p_r^{d_r}$, where for all i; $1 \le i \le r$, d_i is the smallest integer j such that $(h^{n/p_i^{e_i}})^{p_i^j} \in K$.*

This proposition essentially says that the computation of order can be done in polynomial time, provided we have oracles for membership testing and factorization. Notice that computing large powers can be done in polynomial-time using the method of "doubling"; by computing the squares successively. Next we define the language L_{ig} which INDEPENDENTGENERATOR will be using as oracle for order and exponent computations.

Oracles for INDEPENDENTGENERATOR

Here we give the precise definition of the NP language L_{ig}. We define L_{ig} as a disjoint union of two prefix languages L_1 and L_2 in NP.

Let L be a language in NP accepted by an NP machine M. The prefix language of L with respect to M is defined as follows: $\mathit{Prefix}_M(L) = \{\langle x, y\rangle \mid \exists z \in \Sigma^*$ such that yz is an accepting path of M on input $x\}$. It is easy to see that $\mathit{Prefix}_M(L) \in$ NP. Moreover, if $L \in$ UP, then $\mathit{Prefix}_M(L) \in$ UP.

We use the following theorem from [FK92] for defining L_1. Let PRIMES denote the language consisting of prime numbers encoded in binary. We use the following theorem from [FK92].

Theorem 4.2.9 ([FK92]) PRIMES $\in$ UP.

Now, consider the following language L'_1.

$$L'_1 \triangleq \{\langle n, p_1, \ldots, p_k, e_1, \ldots, e_k\rangle \mid \forall i \leq k,\ p_i \in \text{PRIMES};\ p_1 < \ldots < p_k;\ n = \prod_{i=1}^{k} p_i^{e_i}\}$$

From the facts that PRIMES $\in$ UP and for any integer there exists a unique prime factorization, it follows that $L'_1 \in$ UP. Let M'_1 be a UP machine accepting L'_1. Define $L_1 = \mathit{Prefix}_{M'_1}(L'_1)$. We have $L_1 \in$ UP. Notice that it is easy to design a deterministic oracle polynomial-time machine which can compute the complete factorization of any integer by a prefix search using L_1 as oracle. We formally write these observations as a proposition.

Proposition 4.2.10 *The language $L_1' \in$ UP. Let M_1' be a UP machine accepting L_1'. Let $L_1 = Prefix_{M_1'}(L_1')$. Then $L_1 \in$ UP. Moreover, there exists a polynomial-time deterministic oracle machine* FACTOR *which takes an integer as input and computes its complete prime factorization using L_1 as oracle.*

Now we will define the language L_2. L_2 is defined as a prefix language of another language L_2' defined below. Let $\mathcal{B} = \{B_m\}_{m>0}$ be a group family.

$L_2' \triangleq \{\langle m, S, g\rangle \mid g \in B_m, S \subseteq B_m, \text{ and } \forall h \in S; \exists l_h \leq o(h) \text{ such that } g = \Pi_{h\in S} h^{l_h}\}$

The following proposition essentially shows how to make 1-guarded queries to L_2'.

Proposition 4.2.11 *There exists an NP machine M_2' witnessing L_2' in* NP. *Moreover, for those inputs $\langle m, S, g\rangle \in L_2'$ such that S is an independent generator set for the abelian group $\langle S\rangle$, M_2' will have a unique accepting path.*

Proof Sketch. Consider an NP machine M_2' which on input $\langle m, S, g\rangle$, first computes $|B_m|$, then guesses a string $\langle |B_m|, p_1, \ldots, p_k, e_1, \ldots, e_k\rangle$ and verifies that $B_m = \Pi_{i=1}^k p_i^{e_i}$ by simulating the UP machine M_1' given by the previous proposition. Notice that, at the end of this computation, M_2' will have the complete prime factorization of $|B_m|$ on a unique path. Next, using the factorization of $|B_m|$, M_2' computes $o(h)$ for all $h \in S$ in polynomial-time. For this computation it uses the method given in Proposition 4.2.8. That is, it computes d_i for all $i; 1 \leq i \leq r$ where d_i is the smallest j such that $(g^{n/p_i^{e_i}})^{p_i^j} = e$. Since $1 \leq d_i \leq e_i$ and e_i is bounded by a polynomial in m, for every i, each d_i can be computed in time bounded by a polynomial in the length of the input. After this computation, M_2' guesses indices l_h for each $h \in S$ such that $1 \leq l_h \leq o(h)$ and accepts if $g = \Pi_{h\in S} h^{l_h}$. Note that $(l_h | h \in S)$ is an S-exponent of g. Now, if S is an independent generator set for the abelian group $\langle S\rangle$, by Proposition 4.2.4, the S-exponent of g will be unique and hence only one of the guesses of M_2' leads to acceptance. ■

Before we proceed further, we make a few remarks about the language L'_2. Firstly, notice that if $\langle S\rangle$ is abelian then L'_2 is same as Membership Testing (for general groups this is a "weak Membership Testing" oracle since $L'_2 \subset$ Membership Testing). This can be a little confusing since one of our goals is testing membership. But, though we will be using this language as an oracle, we will be very careful about accessing this oracle. We will always make sure that whenever a query $\langle m, S, g\rangle$ is made to this oracle, S is an independent generator set for $\langle S\rangle$. This is acceptable for us because, from the above proposition, such queries will be 1-guarded.

Now define L_2 as $Prefix_{M'_2}(L'_2)$. Then $L_2 \in$ NP. Let M_2 be the corresponding NP machine accepting L_2. Given g and an independent generator set S for the abelian group $\langle S\rangle$, using L_2 as oracle we can prefix search for the X-exponent of g, where $X \subseteq S$. For each such query made to L_2, M_2 will have at most one accepting path. Thus, if S is an independent generator set for an abelian group G, then for any $X \subseteq S$ and $g \in G$, the unique X-exponent of g can be computed in deterministic polynomial-time by making only 1-guarded queries to L_2. These observations we state as a lemma.

Lemma 4.2.12 *Let $L_2 = Prefix_{M'_2}(L'_2)$. Then $L_2 \in$ NP. Moreover there exists a deterministic polynomial-time oracle algorithm* EXPONENT *which takes $\langle m, S, X, g\rangle$ as input and L_2 as oracle such that, if the input satisfies the promise that S is an independent generator set for the abelian group $\langle S\rangle$, $X \subseteq S$ and $g \in \langle S\rangle$, then* EXPONENT *outputs the unique X-exponent of g. Furthermore, for such inputs* EXPONENT *makes only 1-guarded queries to the oracle. The behavior of* EXPONENT *is unspecified if the input does not satisfy the promise.*

In the next lemma, we give a deterministic oracle algorithm ORDER for computing the order of any element in a factor group. ORDER uses the method described in the Proposition 4.2.8 for this computation. It uses the UP language L_1 for factorization. Again, Membership Testing is needed for computing the order in a factor group. But we will be computing the order in a factor group H/K only when an independent generator set for K is already computed. So once again we can use the language L'_2 in a 1-guarded manner for testing membership.

Lemma 4.2.13 *Let $\mathcal{B} = \{B_m\}_{m>0}$ be a group family. Then there exists a deterministic polynomial-time oracle algorithm* ORDER *which takes as input $\langle m, g, X, Y\rangle$, and $0L_1 \cup 1L'_2$ as oracle such that, if the input satisfies the promise that $X, Y \subseteq B_m$, Y is a normal subgroup of X, $g \in \langle X\rangle$ and Y is an independent generator set for $\langle Y\rangle$, then* ORDER *outputs $o(g\langle Y\rangle)$. Furthermore, for such inputs* ORDER *makes only 1-guarded queries to the oracle. The behavior of* ORDER *is unspecified if the input does not satisfy the promise.*

We have defined the languages that we will be using as oracles for *all* the algorithms in this chapter. We can unify these languages into a single language L_{ig} by defining $L_{ig} = 0L_1 \cup 1L_2$ (disjoint union of L_1 and L_2). Notice that all the languages L_1, L'_2 and L_2 are subsumed by L_{ig}. Sometimes, for clarity of presentation, we may split L_{ig} into its component languages.

The Algorithm IndependentGenerator

Now we are ready to give a formal description of IndependentGenerator and prove that it has the behavior as given in Theorem 4.2.5.

We first explain how the algorithm works and then give a formal description of it. Algorithm IndependentGenerator on input $\langle m, S\rangle$, first checks whether $\langle S\rangle$ is abelian. This can be done easily by checking whether $g_i g_j = g_j g_i$ for all $g_i, g_j \in S$. Then it computes $|B_m|$ and using the algorithm Factor as subroutine computes the complete factorization of $|B_m|$. Knowing the factorization $p_1^{e_1} p_2^{e_2} \dots p_r^{e_r}$ of $|B_m|$, it can compute a generator sets for all the Sylow subgroups of $\langle S\rangle$ using the method given in Lemma 4.2.7. Let X_j denote the generator set thus constructed for the p_j-Sylow subgroup.

The next step of INDEPENDENTGENERATOR is to convert the generator sets of the Sylow subgroups into independent generator sets. Lemma 4.2.6 provides a greedy strategy for this. Consider the p_j-group $\langle X_j \rangle$. Let it be of type $(m_1, m_2, \ldots, m_s)$. Let $g_1, g_2 \ldots, g_i$ be i independent elements of $\langle X_j \rangle$ so far constructed such that g_k is of order $p_j^{m_k}$ for $1 \leq k \leq i$. Suppose we could get an element $g'_{i+1} \in \langle X_j \rangle$ such that $g'_{i+1}\langle\{g_1, g_2 \ldots, g_i\}\rangle$ is of maximum order $p^{m_{i+1}}$ in the factor group $\langle X_j \rangle / \langle\{g_1, g_2 \ldots, g_i\}\rangle$. Then by the method of Lemma 4.2.6, we can convert g'_{i+1} to $g_{i+1} \in \langle X_j \rangle$ of order $p^{m_{i+1}}$ which is independent of $\{g_1, g_2 \ldots, g_i\}$ as follows. First compute the $\{g_1, g_2 \ldots, g_i\}$-exponent $(x_1, x_2, \ldots, x_i)$ of $(g'_{i+1})^{p_j^{m_{i+1}}}$ using the algorithm EXPONENT as subroutine. Then compute $g_{i+1} = g'_{i+1} g_1^{-y_1} g_2^{-y_2} \ldots g_i^{-y_i}$ where $y_k = x_k / p^{m_{i+1}}$, for $1 \leq k \leq i$. Then by Lemma 4.2.6, g_{i+1} is an element of $\langle X_j \rangle$ of order $p^{m_{i+1}}$ which is independent of $\{g_1, g_2 \ldots, g_i\}$, there by incrementing the partial independent set by one.

For implementing the above method, we need to compute an element $g'_{i+1} \in \langle X_j \rangle$ and $p^{m_{i+1}}$ such that $g'_{i+1}\langle\{g_1, g_2 \ldots, g_i\}\rangle$ is of maximum order $p^{m_{i+1}}$ in the factor group $\langle X_j \rangle / \langle\{g_1, g_2 \ldots, g_i\}\rangle$. For this the following two propositions can be employed.

The first proposition is an observation which is crucial for our algorithm. It says that, for an abelian p-group, any generator set contains an element of maximum order among all elements of the group. So for getting an element of maximum order, we need only to search in the generator set.

Proposition 4.2.14 *Let G be an abelian p-group generated by a set S. Let $(m_1, m_2, \ldots, m_s)$ be the type of G. If $g \in S$ is an element of maximum order among the elements of S, then g is of order p^{m_1}.*

Proof. G is a group of prime power order. Since $(m_1, m_2, \ldots, m_s)$ is the type of G, the maximum possible order of an element in G is p^{m_1}. If all elements of S are of order less than p^{m_1} then every element of G will be of order less than p^{m_1} since S

The next proposition allows us to apply the above observation for abelian factor groups. It follows from the group-theoretic fact that for any group G, G is isomorphic to $G/H \times H$ where H is a normal subgroup of G.

Proposition 4.2.15 *Let G be an abelian p-group of type $(m_1, m_2, \ldots, m_s)$. Let $H < G$ such that H is of type $(m_1, m_2, \ldots, m_i)$. Then G/H is of type $(m_{i+1}, m_{i+2}, \ldots, m_s)$.*

Finally, we give a formal description of the algorithm IndependentGenerator. We then argue that the behavior of the algorithm is as given in the Theorem. This will complete the proof of Theorem 4.2.5.

```
IndependentGenerator($m, S$)
if $\exists g_i, g_j \in S$ such that $g_i g_j g_i^{-1} g_j^{-1} \neq e$
  then output NOT ABELIAN;
$n \leftarrow |B_m|$;
$p_1^{e_1} p_2^{e_2} \ldots p_r^{e_r} \leftarrow$ Factor($n$);
for $i : 1 \leq i \leq r$
do Compute the generators set $X_i$ for the $p_i$-Sylow subgroup;
    /* Using Lemma 4.2.7 */
end-for
for $i : 1 \leq i \leq r$
do $S_i \leftarrow \phi$; $N_i \leftarrow 1$; $T_i \leftarrow ()$
    /* $S_i$ stands for the set of independent generators
    of $p_i$-Sylow subgroup so far constructed */
    while $\exists\ h \in X_i$ such that $\langle m, S_i, h \rangle \notin L_2'$
    do  Find an element $h_j \in X_i$ such that
        $p_i^{l_j} \leftarrow$Order($m, h_j, X_i, S_i$) is maximum;
        $(x_g \mid g \in S_i) \leftarrow$ Exponent($m, S_i, S_i, h_j$);
        $S_i \leftarrow S_i \cup \{h_j \Pi_{g \in S_i} g^{-x_g/p_i^{l_j}}\}$;
        $N_i \leftarrow p_i^{l_j}.N_i$;
        $T_i \leftarrow T_i \cup (l_j)$;
    end-while
end-for
Output $\cup_{i=1}^{r} S_i$ as the independent generator set;
Output $\Pi_{1 \leq i \leq r} N_i$ as the order;
Output $T_i$ as type of the $p_i$-Sylow subgroup of $\langle S \rangle$ for $1 \leq i \leq r$.
```

It is clear from the above discussions and the description of the algorithm that INDEPENDENTGENERATOR on input $\langle m, S\rangle$ outputs NOT ABELIAN if $\langle S\rangle$ is not abelian and outputs an independent generator set, order and type of $\langle S\rangle$ otherwise, in time polynomial in $|\langle m, S\rangle|$.

Now, we will see that the algorithm makes only 1-*guarded* queries to L_{ig}. The algorithm makes oracle calls in lines 4,13,15 and 16. In *line*-4, it makes a call to the language L_1 through the subroutine FACTOR which is 1-guarded since $L_1 \in$ UP. Now, it is clear from the above discussion, that whenever the algorithm enters the while-loop of *line*-13,the set S_i will be a set of independent elements. Hence, at *line*-13, the query $\langle m, S_i, h\rangle$ to L_2' will be 1-*guarded* from Proposition 4.2.11. The calls to L_{ig} at *lines* 15 and 16 are through the subroutines ORDER and EXPONENT. So these calls also will be 1-guarded since the generator set S_i involved in these subroutines calls are independent (see Lemma 4.2.12 and 4.2.13).

Upper Bounds for Abelian Group Problems

Here we shall illustrate how the algorithm INDEPENDENTGENERATOR can be modified to get upper bounds on the counting complexity of some of the problems over abelian black-box groups. Later in the chapter, we will extend these upper bounds for solvable groups.

Theorem 4.2.16 *Over any group family,* Membership Testing, Order Verification, *and* Group Isomorphism *for abelian black-box groups are in* SPP. *Hence these problems are low for* PP, $C_=$P *and the* Mod$_k$P *for* $k \geq 2$.

Proof. Firstly, we can assume that the groups encoded in the problem instance are abelian for all the three problems. Now consider Membership Testing. Let $\mathcal{B}$ be a group family. Consider an oracle NP machine M_{mb} which on input $\langle m, S, g\rangle$ converts the set S to an independent generator set if S generates an abelian group, by simulating the algorithm INDEPENDENTGENERATOR on input $\langle m, S\rangle$. Let S' be the independent generator set thus constructed. Then M_{mb} makes one query $\langle m, S', g\rangle$ to the language L'_2 and it accepts, producing a *gap*=1, if the answer is 'YES'. If the answer to the query is 'NO' then M branches into an accepting and a rejecting path, thus producing a zero *gap*. So, from Theorem 4.2.5 and Proposition 4.2.11, it follows that M_{mb} runs in time polynomial in the length of the input and makes only 1-guarded queries to L_{ig}. Since $\langle m, S', g\rangle \in L'_2$ if and only if $g \in \langle S'\rangle = \langle S\rangle$, *gap* produced by M_{mb} is 0 if $g \notin \langle S\rangle$ and *gap* is 1 if $g \in \langle S\rangle$. Now from Lemma 2.0.2, it follows that Membership Testing is in SPP.

To show that Order Verification is in SPP, notice that one of the outputs of INDEPENDENTGENERATOR on input $\langle m, S\rangle$ is $|\langle S\rangle|$. Hence, it is easy to modify INDEPENDENTGENERATOR to an oracle algorithm M_o, which on input $\langle m, S, n\rangle$ produces zero *gap* if $n \neq |\langle S\rangle|$ and *gap* 1 if $n = |\langle S\rangle|$. Hence, again from Theorem 4.2.5 and Lemma 2.0.2 it follows that Order Verification is in SPP.

Finally we show that Group Isomorphism for abelian groups is in SPP. Let $\langle m, S_1, S_2\rangle$ be an instance of Group Isomorphism. S_1 and S_2 generate abelian subgroups of $B_m \in \mathcal{B}$. Firstly, observe that any two abelian p-groups are isomorphic if and only if their types are identical. This follows from the fact that the type uniquely determines an abelian p-group up to isomorphism [Bur55]. Furthermore, two abelian groups $\langle S_1\rangle$ and $\langle S_2\rangle$ are isomorphic iff $|\langle S_1\rangle| = |\langle S_2\rangle|$, and for every prime factor p of $|\langle S_1\rangle|$, the p-Sylow subgroup of $\langle S_1\rangle$ is isomorphic to the p-Sylow subgroup of $\langle S_2\rangle$. Thus, in order to check that $\langle S_1\rangle$ and $\langle S_2\rangle$ are isomorphic, it is only required to verify that the p-Sylow subgroups of $\langle S_1\rangle$ and $\langle S_2\rangle$ respectively have identical types, for each prime factor p of $|\langle S_1\rangle|$. Hence, from Theorem 4.2.5 and Lemma 2.0.2 it follows that Group Isomorphism is in SPP. ■

4.3 Solvable Groups and Canonical Generator Sets

In the case of abelian groups, we have seen that the existence of an independent generator set played a crucial role in proving upper bounds for various computational problems. Motivated by this, we abstract out the essential structure that we need in a generator set which will help us in proving similar upper bounds for other classes of groups. More precisely we define the notion of *canonical generator set* for arbitrary classes of finite groups as a generalization to the notion of independent generator sets. Then we show the existence of canonical generator sets for the class of solvable groups.

Definition 4.3.1 Let $\mathcal{B} = \{B_m\}_{m>0}$ be any group family. Let $\mathcal{C}$ be a subclass of $\mathcal{B}$. The class of groups $\mathcal{C}$ has *canonical generator sets* if for every $G \in \mathcal{C}$, if $G < B_m$ then there is an *ordered* set $S = \{g_1, g_2, \ldots, g_s\} \subseteq G$ such that S generates G and each $g \in G$ can be uniquely expressed as $g = g_1^{l_1} g_2^{l_2} \ldots g_s^{l_s}$, where $1 \le l_i \le o(g_i), 1 \le i \le s$. Furthermore, $s \le q(m)$ for a polynomial q. S is called a *canonical generator set* for G.

From the above definitions, it is clear that if S is an independent generator set for an abelian group G then each $x \in G$ can be uniquely expressed as a product $\Pi_{g \in S} g^{i_g}$ for $1 \le i_g \le o(g)$, and it holds that $|S| \le \log(|G|)$. Thus for an abelian group, an independent generator set is a canonical generator set.

Note that the ordering of the elements of the canonical generator set is irrelevant for abelian groups since products commute. However, for classes of groups which are not abelian the ordering of elements in a canonical generator set has to be prescribed. Now we shall extend the notion of exponents to any class of groups with canonical generator sets.

Definition 4.3.2 Let $\mathcal{B} = \{B_m\}_{m>0}$ be any group family. Let $\mathcal{C}$ be a subclass of $\mathcal{B}$ which has canonical generator sets. Let G be an element of $\mathcal{C}$. Let S be a canonical generator set for G and let $X \subseteq S$. Then for $g \in G$, the tuple $(l_x | x \in X)$ is called the X-*exponent* of g with respect to S if for all $x \in X$, $1 \leq l_x \leq o(x)$ and there exists indices $\{l_y \mid y \in S - X\}$ such that $g = \prod_{s \in S} s^{l_s}$.

From the definition it follows that for $g \in G$ the X-exponent of g is unique. We now show that the class of solvable black-box groups for any group family has canonical generator sets.

Lemma 4.3.3 *Let $\mathcal{B} = \{B_m\}_{m>0}$ be a group family such that $|B_m| \leq 2^{q(m)}$ for a polynomial q. Let $G < B_m$ be a finite solvable group and $G = G_0 > G_1 > \ldots > G_{k-1} > G_k = e$ be the commutator series of G. Let $T_i = \{h_{i1}, h_{i2}, \ldots, h_{is_i}\}$ be a set of distinct coset representatives corresponding to an independent set of generators for the abelian group $H_i = G_{i-1}/G_i$. Then for any i, $1 \leq i \leq k$, the ordered set*[1] *$S_i = \bigcup_{j=i}^{k} T_j$ forms a canonical generator set for the group G_i and $|S_i| \leq q(m)$. Thus the class of solvable groups from $\mathcal{B}$ has canonical generator sets.*

Proof. Let $\mathcal{B} = \{B_m\}_{m>0}$ be a group family such that $|B_m| \leq 2^{q(m)}$ for a polynomial q. Let $G < B_m$ be a finite solvable group and $G = G_0 > G_1 > \ldots > G_{k-1} > G_k = \langle e \rangle$ be the commutator series of G. Recall that subgroups of solvable groups are solvable. We prove by induction on $k - i$, that for each $i : 1 \leq i \leq k$, $\bigcup_{j=i+1}^{k} T_j$ is a canonical generator set for G_i. For the base case, when $k - i = 1$ it clearly holds, since G_{k-1} is an abelian group and T_k is an independent set of generators for G_{k-1}. Suppose, as induction hypothesis, that $\bigcup_{j=i+1}^{k} T_j$ is a canonical generator set for G_i. Consider the group G_{i-1}. The factor group G_{i-1}/G_i is generated by the independent set of generators $X_i = \{h_{i1}G_i, h_{i2}G_i \ldots, h_{ik_i}G_i\}$. An element $g \in G_{i-1}$ belongs to exactly one right coset in G_{i-1}/G_i. Let that right coset be hG_i. Thus $g = hg'$ for

[1]The elements of the set $\bigcup_{j=i}^{k} T_j$ are ordered on increasing values of the index j, and lexicographically within each set T_j.

some $g' \in G_i$. Now, since X_i is an independent generator set of the abelian group G_{i-1}/G_i, there are unique indices $\{l_{i1}, l_{i2}, \ldots, l_{ik_i}\}$ such that $hG_i = h_{i1}^{l_{i1}} \ldots h_{ik_i}^{l_{ik_i}} G_i$. Consequently, $\{l_{i1}, l_{i2}, \ldots, l_{ik_i}\}$ are unique indices such that $g = hg' = h_{i1}^{l_{i1}} \ldots h_{ik_i}^{l_{ik_i}} g''$, where $g'' \in G_i$. By induction hypothesis, $\bigcup_{j=i+1}^{k} T_j$ is a canonical generator set for G_i, implying that g'' can be uniquely expressed as a product of powers of the elements in $\bigcup_{j=i+1}^{k} T_j$. We have proved the induction step that every $g \in G_{i-1}$ can be uniquely expressed as a product of powers of the elements in $\bigcup_{j=i}^{k} T_j$. Thus, $\bigcup_{j=i}^{k} T_j$ is a canonical generator set for G_{i-1}.

Finally we show that $q(m)$ bounds $|\bigcup_{i=1}^{k} T_i|$. Observe that since for each i, $1 \le i \le k$ the set X_i is an independent generator set for the quotient group G_{i-1}/G_i, it holds that $|T_i| \le \log(|G_{i-1}/G_i|)$. Therefore, it follows that $|\bigcup_{i=1}^{k} T_i| \le \sum_{1 \le i \le k} \log(|G_{i-1}/G_i|) = \log(|G|) \le q(m)$ since $|G| \le 2^{q(m)}$. ∎

In the rest of this section, we will consider the problem of constructing a canonical generator set from an arbitrary generator set for a solvable black-box group. Since from Lemma 4.3.3 a canonical generator set for a solvable group is an ordered collection of independent generator sets for the abelian factor groups of the commutator series of the solvable group, in the next subsection, we consider the problem of computing an independent generator set for an abelian factor group.

4.3.1 Constructing an Independent Generator Set for Abelian Factor Groups

Here we will design a deterministic oracle algorithm FACTINDGENERATOR which takes generator sets for two groups G and H such that H is normal in G and G/H is abelian, and computes coset representatives for an independent generator set for G/H. More precisely, we prove the following theorem.

Theorem 4.3.4 *Let $\mathcal{B} = \{B_m\}_{m>0}$ be a group family. Then there exists a deterministic oracle algorithm* FACTINDGENERATOR *such that* FACTINDGENERATOR *takes*

$\langle m, S, Y\rangle$ as input, where $S, Y \subseteq B_m$, and L_{ig} as oracle. Suppose the input $\langle m, S, Y\rangle$ satisfies the following promise:

1. *Y is a canonical generator set for the solvable group $\langle Y\rangle$.*
2. *$\langle Y\rangle$ is normal in $\langle S\rangle$ and $\langle S\rangle/\langle Y\rangle$ is abelian.*

Then FACTINDGENERATOR *outputs $|\langle S\rangle/\langle Y\rangle|$ and a set of independent generators for $\langle S\rangle/\langle Y\rangle$. Furthermore,* FACTINDGENERATOR *runs in time polynomial in $|\langle m, S, Y\rangle|$, and makes 1-guarded queries to L_{ig}. The behavior of* FACTINDGENERATOR *is unspecified if the input does not satisfy the promise.*

Notice here that FACTINDGENERATOR is using as oracle the same NP language L_{ig} defined in the previous section.

Let us recall the definition of the language L_2' which is one of the main components of L_{ig}.

$$L_2' \triangleq \{\langle m, S, g\rangle \mid g \in B_m, S \subseteq B_m, \text{ and } \forall h \in S; \exists l_h \leq o(h) \text{ such that } g = \Pi_{h \in S} h^{l_h}\}$$

Now we make an important observation that for an instance $\langle m, S, g\rangle$ of L_2', if S is a canonical generator set of a solvable group $\langle S\rangle$ then $\langle m, S, g\rangle \in L_2'$ if and only if $g \in \langle S\rangle$. That is, if a solvable group is presented by a canonical generator set, then Membership Testing can be done using the language L_2'. Moreover, the machine M_2' (given in Proposition 4.2.11) will have at most one accepting path on such instances. We state these observations as a proposition.

Proposition 4.3.5 *Let $\langle m, S, g\rangle$ be an instance of L_2' such that S is a canonical generator for the group $\langle S\rangle$. Then $\langle m, S, g\rangle \in L_2'$ if and only if $g \in \langle S\rangle$. Moreover, for those inputs $\langle m, S, g\rangle \in L_2'$ such that S is canonical generator set for the group $\langle S\rangle$, M_2' will have a unique accepting path.*

Because of the above property of L'_2, we have lemmas identical to Lemmas 4.2.12 and 4.2.13 for computing order and exponents over solvable groups also. We state them here.

Lemma 4.3.6 *Let $\mathcal{B} = \{B_m\}_{m>0}$ be a group family. Then there exists a deterministic polynomial-time oracle algorithm* EXPONENT *which takes $\langle m, S, X, g\rangle$ as input and L_2 as oracle such that, if the input satisfies the promise that S is a canonical generator set for the solvable group $\langle S\rangle$, $X \subseteq S$ and $g \in \langle S\rangle$, then* EXPONENT *outputs the unique X-exponent of g. Furthermore* EXPONENT *makes only 1-guarded queries to the oracle. The behavior of* EXPONENT *is unspecified if the input does not satisfy the promise.*

Lemma 4.3.7 *Let $\mathcal{B} = \{B_m\}_{m>0}$ be a group family. Then there exists a deterministic polynomial-time oracle algorithm* ORDER *which takes as input $\langle m, g, X, Y\rangle$, and $0L_1 \cup 1L'_2$ as oracle such that, if the input satisfies the promise that $X, Y \subseteq B_m$, Y is a normal subgroup of X, $g \in \langle X\rangle$ and Y is a canonical generator set for $\langle Y\rangle$, then* ORDER *outputs $o(g\langle Y\rangle)$. Furthermore* ORDER *makes only 1-guarded queries to the oracle. The behavior of* ORDER *is unspecified if the input does not satisfy the promise.*

Finally we come to the design of FACTINDGENERATOR. The design of FACTINDGENERATOR follows very closely that of INDEPENDENTGENERATOR. Observe that the Lemma 4.2.6, 4.2.7 and Proposition 4.2.14, which are important in the design of INDEPENDENTGENERATOR, are purely group-theoretic and hold for abelian factor groups also.

Before giving a formal description of FACTINDGENERATOR, we shall briefly explain how the algorithm works.

Let $\mathcal{B}$ be a group family. Let $G < B_m$ be a group and H be a normal solvable subgroup of G such that the factor group G/H is abelian. We are interested in computing an independent generator set for G/H. Let the group G/H be presented by a set X of coset representatives of generators of G/H and a canonical generator set Y for H. It is clear that if prime factorization of $|B_m|$ is known, then using method given by Lemma 4.2.7 we can compute a set of coset representatives for a generator set of the Sylow subgroups of G/H. So we shall assume that G/H is an abelian p-group for prime p. We use the method given by the Lemma 4.2.6 for converting X into a set of representatives for independent generator set for G/H.

Let G/H be of type $(m_1, \ldots, m_s)$ and $g_1H, \ldots, g_iH$ be a set of independent elements of G/H of orders $p^{m_1}, \ldots, p^{m_i}$ respectively. Firstly we have to find an element $g'_{i+1} \in X$ such that $g'_{i+1}H$ will be of order $p^{m_{i+1}}$ in the group $\langle\{g_1H, \ldots, g_iH\}\rangle$. Then for getting an element $g_{i+1}H$ independent of $g_1H, \ldots, g_iH$ of order $p^{m_{i+1}}$, we have to compute the $\{g_1H, \ldots, g_iH\}$-exponent of $(g'_{i+1}H)^{p^{m_{i+1}}}$.

Notice that, we have to do order and exponent computations over factor groups of G/H which itself is a factor group. Next we state an isomorphism theorem by which we can avoid computations over factor groups of factor groups. The theorem is elementary and can be seen in any standard text book in group theory [Hal59, Bur55].

Theorem 4.3.8 (Isomorphism Theorem) *Let H and K be normal subgroups of G and $H < K$. Then K/H is normal in G/H and the map $gH(K/H) \to gK$ from the factor group $(G/H)/(K/H)$ to G/K is an isomorphism of $(G/H)/(K/H)$ to G/K.*

Next we state a proposition which will help us in using the algorithm ORDER for order computations. The proof is implicit in the proof of Lemma 4.3.3 and is omitted here.

Proposition 4.3.9 *Let H, G be solvable and H be a normal subgroup of G such that G/H is abelian. Let $X \subseteq G$ be a set of coset representatives of an independent* *generator set for G/H and Y be a canonical generator set for H. Then the ordered set $S = X \cup Y$ is a canonical generator set for G, where elements from X are before those from Y in S.*

From the above two results, it follows that, the order of any element $gH \in G/H$ in the group $\langle\{g_1H, \ldots, g_iH\}\rangle$ is same as the order of $g \in G$ in the group $\langle\{g_1, \ldots, g_i\} \cup Y\rangle$. So the algorithm ORDER can be used to compute this. Moreover, since $\{g_1, \ldots, g_i\} \cup Y$ is a canonical generator set, the ORDER makes only 1-*guarded* queries to L_2'.

Now we see how we can use the algorithm EXPONENT to compute exponents.

Proposition 4.3.10 *Let G be a solvable group and $H < G$ be a normal subgroup of G such that G/H is abelian and Y be a canonical generator set for H. Let $\{g_1H, \ldots, g_iH\}$ be a set of independent elements of G/H. Then for any $gH \in \langle\{g_1H, \ldots, g_iH\}\rangle$, the $\{g_1H, \ldots, g_iH\}$-exponent of gH is same as the $\{g_1, \ldots, g_i\}$-exponent of g with respect to the canonical generator set $\{g_1, \ldots, g_i\} \cup Y$.*

Proof. Since $\{g_1H, \ldots, g_iH\}$ are independent, it follows that the $\{g_1H, \ldots, g_iH\}$-exponent of gH is unique. Let it be $(l_1, \ldots, l_i)$. Then $gH = (g_1H)^{l_1} \ldots (g_iH)^{l_i}$. This means that there is a unique $x \in H$ such that $g = g_1^{l_1} \ldots g_i^{l_i}x$. The proposition follows. ■

It follows that the algorithm EXPONENT can be used appropriately for the computation of exponents over a set of independent elements of a abelian factor group.

The rest of the computation is exactly same as that in INDEPENDENTGENERATOR. Finally we give the description of algorithm FACTINDGENERATOR. The proof that the algorithm behaves as given in Theorem 4.3.4 is very similar to that of Theorem 4.2.5.

```
FACTINDGENERATOR($m, S, Y$)
  $n \leftarrow |B_m|$;
  $p_1^{e_1} p_2^{e_2} \dots p_r^{e_r} \leftarrow$ FACTOR($n$);
  for $i : 1 \le i \le r$
  do Compute a set of coset representatives for the generator
      set $X_i$ for the $p_i$-Sylow subgroup of $G/H$;
      /* Using Lemma 4.2.7 */
  end-for
  for $i : 1 \le i \le r$
  do $S_i \leftarrow \phi$; $N_i \leftarrow 1$; $T_i \leftarrow ()$
      /* $S_i$ stands for the representatives for a set of independent
      generators for $p_i$-Sylow subgroup of $G/H$ so far constructed */
      while $\exists\, h \in X_i$ such that $\langle m, S_i \cup Y, h \rangle \notin L_2'$
      do  Find an element $h_j \in X_i$ such that
          $p_i^{l_j} \leftarrow$ORDER($m, h_j, X_i, S_i \cup Y$) is maximum;
          $(x_g \mid g \in S_i) \leftarrow$ EXPONENT($m, S_i, Y, h_j$);
          $S_i \leftarrow S_i \cup \{h_j \Pi_{g \in S_i} g^{-x_g/p_i^{l_j}}\}$;
          $N_i \leftarrow p_i^{l_j}.N_i$; $T_i \leftarrow T_i \cup (l_j)$
      end-while
  end-for
  Output $\cup_{i=1}^{r} S_i$ as representatives for an independent generator set for $G/H$;
  Output $\Pi_{1 \le i \le r} N_i$ as the order of $G/H$;
  Output $T_i$ as type of the $p_i$-Sylow subgroup of $\langle S \rangle / \langle Y \rangle$ for $1 \le i \le r$.
```

4.3.2 Constructing a Canonical Generator Set

Now we are ready to design a deterministic oracle algorithm (we shall call it CANONICALGENERATOR) which takes an arbitrary generator set of a solvable group G and converts it into a canonical generator set for G. The algorithm makes queries to the language L_{ig} defined earlier in this chapter. Again, for applications to our upper bound proofs, we will make sure that CANONICALGENERATOR makes only 1-guarded queries to L_{ig}. We state the behavior of CANONICALGENERATOR as a theorem. This subsection is devoted to the proof of this theorem.

Theorem 4.3.11 *Let $\mathcal{B} = \{B_m\}_{m \ge 0}$ be a group family. Then there is a deterministic oracle algorithm* CANONICALGENERATOR *such that* CANONICALGENERATOR *takes $\langle m, S\rangle$ as input and L_{ig} as oracle, and outputs a canonical generator set for $\langle S\rangle$ if $\langle S\rangle$ is solvable and outputs* NOT SOLVABLE *otherwise. Moreover,* CANONICALGENERATOR *runs in time polynomial in the length of the input and makes only 1-guarded queries to L_{ig}.*

Before going into the formal proof of the theorem, we give the basic ideas behind the proof. Firstly we shall see the usefulness of the algorithm FACTINDGENERATOR (Theorem 4.3.4) in the design of CANONICALGENERATOR. Let S be an arbitrary generator set for a solvable group $G < B_m$. Let $G = G_0 > \ldots > G_i > \ldots > G_k = \{e\}$ be the commutator series of G. (Notice that $k \leq p(m)$ where p is the polynomial bounding the length of the encoding of elements of the group in the group family. This follows from the fact that $|B_m| \leq 2^{p(m)}$ and Lagrange's theorem.) From Lemma 4.3.3, a canonical generator set for a solvable group is an ordered collection of independent generator sets for each abelian factor groups G_{i-1}/G_i. So, suppose we have given a set of generators S_i for each of the groups G_i in the commutator series, then we can construct a canonical generator set for G by a series of simulations of FACTINDGENERATOR on inputs $\langle m, S_{i-1}, S'_i\rangle$, for $k \geq i \geq 1$, starting from $i = k$ and decrementing i by one after each simulation. Here S'_i is the output of FACTINDGENERATOR on the previous simulation. So we have the following theorem.

Theorem 4.3.12 *Let $\mathcal{B} = \{B_m\}_{m\geq 0}$ be a group family. Then there is a deterministic oracle algorithm* CANONIZE *such that* CANONIZE *takes $\langle m, S_0, \ldots, S_k\rangle$, $S_i \subseteq B_m$ as input and L_{ig} as oracle. Suppose the input satisfies the promise that $\langle S_0\rangle$ is solvable and for all $0 \leq i \leq k$, S_i generates the i^{th} commutator subgroup of $\langle S_0\rangle$. Then* CANONIZE *outputs canonical generator sets for $\langle S_i\rangle$ for $0 \leq i \leq k$. Moreover,* CANONIZE *runs in time polynomial in the length of the input and makes only 1-guarded queries to L_{ig}. The behavior of* CANONIZE *is unspecified if the input does not satisfy the promise.*

In view of the above theorem, it is clear that the problem of computing a canonical generator set for any solvable group essentially boils down to the problem of computing a short generator set for the commutator subgroup of the group. The following proposition provides a method for this computation. A proof of this is given in the appendix.

Proposition 4.3.13 *Let G be a finite group generated by the set S. Then, the commutator subgroup of G is the normal closure of the set $\{ghg^{-1}h^{-1} \mid g, h \in S\}$ in G.*

The above theorem gives us the following easy polynomial-time oracle algorithm CommutatorSubgroup which takes $\langle m, S\rangle$ as input and Membership Testing as oracle and computes a generator set for the commutator group of $\langle S\rangle$.

```
CommutatorSubgroup($m, S$)
1  $X \leftarrow \{ghg^{-1}h^{-1} \mid g, h \in S\}$;
2  while $\exists g \in S;\ x \in X$ such that $\langle m, X, gxg^{-1}\rangle \notin$ Membership Testing
3  do $X \leftarrow X \cup gxg^{-1}$;
4  end-while
5  Output $X$.
```

It easily follows from Proposition 4.3.13 that the algorithm CommutatorSubgroup, on input $\langle m, S\rangle$ outputs a generator set for the commutator subgroup $\langle S\rangle'$. Let X_i be the set X at the beginning of the i^{th} iteration of the **while**-loop. If, after the i^{th} iteration, no new element is added to X_i, then X_i is output. Otherwise, if $X_{i+1} = X_i \cup \{g\}$, it follows from Lagrange's theorem that $|X_{i+1}| \geq 2|X_i|$. Hence the number of iterations of the **while**-loop is bounded by $p(m)$, the polynomial bounding the length of the encoding of the elements of the group in the group family.

A straightforward adaptation of the above algorithm for computing a generator set for the commutator group seems difficult because of the queries to Membership

Testing oracle. Suppose we can make sure that whenever a query $\langle m, X, g\rangle$ to Membership Testing is done, X is a canonical generator set for the solvable group $\langle X\rangle$, then we can replace Membership Testing oracle by L_2' and from Proposition 4.3.5 it will follow that query y will be 1-guarded. We make sure this promise, by constructing the commutator series incrementally.

The algorithm CANONICALGENERATOR works in stages. Let S_i^j denote the partial generator set for the i^{th} element in the commutator series of G_0 constructed at the end of *stage* $(j-1)$. At *stage 1* we have $S_0^1 = S$ and $S_i^1 = \{e\}$ for $1 \leq i \leq p(m)$, where p is the polynomial bounding the length of any element in the group family. Input to *Stage* j is the tuple $\langle i, S_i^j, \ldots, S_{p(m)}^j\rangle$ such that for $l > i$, S_l^j is a canonical generator set for the solvable group $\langle S_l^j\rangle$. During the computation of this stage, we will make sure that for all the queries made to L_{ig}, the groups involved in the queries will be only those for which a canonical generator set has been constructed, thereby making these queries 1-guarded. At the end of the stage, we update each S_i^j to S_i^{j+1} such that S_i^{j+1} is still a subgroup of G_i, the i^{th} commutator subgroup of G_0. To keep the running time within polynomial bound, we make sure that after $p(m)$ stages, there exists k, such that the k^{th} partial commutator subgroup doubles in size. Then from Lagrange's theorem, it will follow that the commutator series will be generated after $p(m)^3$ stages. We now formally prove the theorem.

Proof. *(of Theorem 4.3.11)* We first give a formal description of the algorithm CANONICALGENERATOR and then prove the correctness.

CANONICALGENERATOR uses oracle algorithms CHECKCOMMUTATOR and CANONIZE as subroutines. CHECKCOMMUTATOR takes as input $\langle m, X, Y\rangle$ such that $X, Y \subseteq B_m$ and checks whether $\langle Y\rangle$ contains the commutator subgroup of $\langle X\rangle$. This is done by first checking whether the commutators of all the elements in X are in $\langle Y\rangle$. If this is not the case, the algorithm returns such a commutator. Otherwise, it further checks whether $\langle Y\rangle$ is normal in $\langle X\rangle$. Notice that, to do this it is enough to verify that $\forall x \in X; y \in Y$, $xyx^{-1} \in \langle Y\rangle$. If this condition is false, the algorithm returns an element $xyx^{-1} \notin \langle Y \rangle$. If both the conditions are true, it follows from Proposition 4.3.13 that $\langle Y \rangle$ contains the commutator subgroup of $\langle X \rangle$.

CheckCommutator makes oracle queries to the language L'_2. (weak Membership Testing, defined in the section 4.2.1) for testing membership in $\langle Y \rangle$. It should be noted that, for CheckCommutator to work as intended, Y should be a canonical generator set for the group $\langle Y \rangle$. We will make sure that CanonicalGenerator makes calls to CheckCommutator with $\langle m, X, Y \rangle$ as input, only when Y is a canonical generator set for the solvable group $\langle Y \rangle$. A formal description of the subroutine CheckCommutator is given below.

```
CheckCommutator($m, X, Y$)
if $\exists x_1, x_2 \in X$, such that $\langle m, Y, x_1 x_2 x_1^{-1} x_2^{-1} \rangle \notin L'_2$
   then $g \leftarrow xyx^{-1}y^{-1}$;
         Return $g$.
   else  if $\exists x \in X, y \in Y$ such that $\langle 0^m, Y, xyx^{-1} \rangle \notin L'_2$
            then $g \leftarrow xyx^{-1}$;
                  Return $g$.
            else  $g \leftarrow$ YES;
                  Return $g$.
         end-if
end-if
```

The subroutine Canonize is the algorithm promised by Theorem 4.3.12 for computing a canonical generator set for a solvable black-box group G, given an arbitrary generator set for the commutator series of G. We use the notation $[\text{Canonize}()]_l$ to denote the generator set produced by Canonize for the l^{th} element G_l, in the commutator series of G.

Following is the description of the algorithm CanonicalGenerator. CanonicalGenerator makes queries L_{i_g} through the subroutines Canonize and CheckCommutator.

```
CANONICALGENERATOR($m, S$)
 1  Stage 0
 2  $S_1^0 \leftarrow S$; $S_i^0 \leftarrow \{e\}$ for $1 \le i \le p(m)$;
 3  $i \leftarrow 1$;
 4  Stage $j$ (Input to this stage is $\langle i, S_1^j, \ldots, S_{p(m)}^j \rangle$)
 5  $k \leftarrow i$;
 6  $g \leftarrow$ CHECKCOMMUTATOR($m, S_k^j, S_{k+1}^j$);
 7  while $g \neq$ YES
 8  do $S_{k+1}^j \leftarrow S_k^j \cup \{g\}$;
 9      $k \leftarrow k + 1$;
10      if $k = p(m)$
11         then Output NOT SOLVABLE.
12      end-if
13      $g \leftarrow$ CHECKCOMMUTATOR($m, S_k^j, S_{k+1}^j$);
14  end-while
15  if $k = 1$
16     then Output [CANONIZE($S_1^j, S_2^j, \ldots, S_{p(m)}^j$)]$_1$.
17     else  $S_l^{j+1} \leftarrow S_l^j$ for $1 \le l \le (k-1)$;
18           $S_l^{j+1} \leftarrow$ [CANONIZE($S_k^j, S_{k+1}^j, \ldots, S_{p(m)}^j$)]$_l$ for $k \le l \le p(m)$;
19           $i \leftarrow (k-1)$;
20            goto Stage $j+1$;
21  end-if
```

Now we are ready to prove the correctness of CANONICALGENERATOR. We first prove a series of claims, from which the correctness will follow easily.

Claim 4.3.13.1 *In the algorithm* CANONICALGENERATOR, *at any stage j, it holds that $\forall i; 1 \le i < p(m)$, $\langle S_{i+1}^j \rangle < \langle S_i^j \rangle'$.*

Proof. We prove this by induction on the stages. For the base case, when $j = 0$, it is clear that the claim holds. Assume that it is true for $(j-1)^{\text{th}}$ stage. Now consider S_{i+1}^j and S_i^j. Depending on how the sets S_{i+1}^j and S_i^j are updated in *lines*-17,18 of CANONICALGENERATOR, we have the following cases.

Case 1. $S_i^j = S_i^{j-1}$; $S_{i+1}^j = S_{i+1}^{j-1}$. In this case, from the induction hypothesis, it is clear that $\langle S_{i+1}^j \rangle < \langle S_i^j \rangle'$.

Case 2. $S_i^j = S_i^{j-1} \cup \{g_i\}$; $S_{i+1}^j = S_{i+1}^{j-1}$. From the induction hypothesis, it follows that $\langle S^j_{i+1}\rangle = \langle S^{j-1}_{i+1}\rangle < \langle S^{j-1}_i\rangle' < \langle S^j_i\rangle'$.

Case 3. $S^j_i = S^{j-1}_i$; $S^j_{i+1} = S^{j-1}_{i+1} \cup \{g_{i+1}\}$. The element g_{i+1} is added to the set S^{j-1}_{i+1} at *line*-8 of the algorithm, where g_{i+1} is the element returned by the subroutine CHECKCOMMUTATOR. Suppose g_{i+1} is a commutator of the set $S^j_i = S^{j-1}_i$. Then $g_{i+1} = xyx^{-1}y^{-1}$ for some elements $x, y \in S^j_i$. From induction hypothesis and the definition of the commutator subgroup of a group, it follows that $\langle S^j_{i+1}\rangle = \langle S^{j-1}_{i+1} \cup \{g_{i+1}\}\rangle < \langle S^j_i\rangle'$. On the other hand, suppose g_{i+1} is of the form xyx^{-1} for some $x \in S^j_i = S^{j-1}_i$ and $y \in S^{j-1}_{i+1}$. We have, $\langle S^{j-1}_{i+1}\rangle < \langle S^{j-1}_i\rangle' = \langle S^j_i\rangle'$. But we know that $\langle S^j_i\rangle'$ is normal in $\langle S^j_i\rangle$. So, in particular $g_{i+1} \in \langle S^j_i\rangle'$ and hence $\langle S^j_{i+1}\rangle < \langle S^j_i\rangle'$.

Case 4 . $S^j_i = S^{j-1}_i \cup \{g_i\}$; $S^j_{i+1} = S^{j-1}_{i+1} \cup \{g_{i+1}\}$. From induction hypothesis, we have $\langle S^{j-1}_{i+1}\rangle < \langle S^{j-1}_i\rangle'$. It follows that $\langle S^{j-1}_{i+1}\rangle < \langle S^{j-1}_i \cup \{g_i\}\rangle$. Now using a very similar argument as in **Case 3**, it is easy to show that $\langle S^j_{i+1}\rangle < \langle S^j_i\rangle'$.

Hence the claim. ■

Claim 4.3.13.2 *In* CANONICALGENERATOR, *the input* $\langle i, S^j_i, S^j_{i+1}, \ldots, S^j_{p(m)}\rangle$ *to any stage* j, *is such that for all* $i < t \le p(m)$, S^j_t *is a canonical generator for the solvable group* $\langle S^j_t\rangle$.

Proof. We prove this by induction. For $j = 1$, it is easily verified that the claim is true. Let us assume that the claim is true for j^{th} stage. Let $\langle i, S^j_i, \ldots, S^j_{p(m)}\rangle$ be the input to the j^{th} stage. Let the **while**-loop is exited through *line*-14 after l iterations with the value of g='YES'. (If the loop is exited through *line*-11, then there are no more stages to be considered). Then the value of $k = i + l$ and for $t > k$, S^j_t is not updated inside the loop, and hence by induction hypothesis, it remains a canonical generator set for the solvable group $\langle S^j_t\rangle$. Since the value of g = CHECKCOMMUTATOR(m, S^j_k, S^j_{k+1}) is 'YES' we have that $\langle S^j_k\rangle' < \langle S^j_{k+1}\rangle$. From Claim 4.3.13.1 we have $\langle S^j_{k+1}\rangle < \langle S^j_k\rangle'$. Hence $\langle S^j_{k+1}\rangle = \langle S^j_k\rangle'$. It follows that $\langle S^j_k\rangle$ is solvable and S^j_t for $k \le t \le p(m)$ are generator sets for the commutator series of $\langle S^j_k\rangle$. Hence at *line*-18, CANONIZE will output a canonical generator set for each of the elements in the commutator series of $\langle S^j_k\rangle$. At *line*-19, i is updated to $k-1$, and the input to the $j+1^{\text{th}}$ stage $\langle k-1, S_{k-1}^{j+1}, S_k^{j+1}, \ldots, S_{p(m)}^{j+1}\rangle$ where S_t^{j+1} is a canonical generator set for the solvable group $\langle S_t^{j+1}\rangle$ for $k \le t \le p(m)$. Hence the claim. ■

Claim 4.3.13.3 *In the algorithm* CANONICALGENERATOR, *for any stage j, it holds that $\exists i$ such that $|\langle S_i^{j+p(m)}\rangle| \ge 2|\langle S_i^j\rangle|$ if stage $j+p(m)$ exists.*

Proof. Notice that, if the algorithm at stage j enters the **while**-loop, then there $\exists i$ such that $S_i^{j+1} = S_i^j \cup g$ for a $g \notin \langle S_i^j\rangle$. So, it is enough to show that the **while**-loop is entered at least once after every $p(m)$ sages, if such a stage exists. Suppose the stage j is entered with the value of $i = i'$. It is clear from the algorithm that if the algorithm never enters the **while**-loop in the next $p(m)$ stages, at stage $(j + p(m) + 1)$, the value of $i = i' - (p(m) + 1) < 0$, for all $i' \le p(m)$, which is impossible, since the algorithm is terminated when the value of $i = 0$. Hence the claim. ■

To complete the proof of the theorem, first we shall see that the algorithm CANONICALGENERATOR runs in time polynomial in the length of the input. Observe that it is enough to show that the number of stages executed by the algorithm is bounded by a polynomial, since the number of iterations of the **while**-loop in *lines* 7-14 is bounded by $p(m)$. Now, the claim is that the the number of stages executed by the algorithm is bounded by $2p^3(m)$. Firstly, notice that for any $H < B_m$, $|H| \le 2^{p(m)}$. Hence for any j, $\prod_{i=1}^{p(m)} |\langle S_i^j\rangle| \le 2^{p^2(m)}$. Suppose the claim is false. Now from Claim 4.3.13.3 it follows that, $\prod_{i=1}^{p(m)} |\langle S_i^{j+p(m)}\rangle| \ge 2\prod_{i=1}^{p(m)} |\langle S_i^j\rangle|$. Hence $\prod_{i=1}^{p(m)} |\langle S_i^{2p^3(m)}\rangle| > 2^{p^2(m)}$, a contradiction.

Now we shall see that CANONICALGENERATOR makes only 1-guarded queries to L_{ig}. Let us first see that the queries to L_{ig} through CHECKCOMMUTATOR are 1-guarded. It is enough to show that whenever CANONICALGENERATOR calls CHECKCOMMUTATOR with argument $\langle m, S_k^j, S_{k+1}^j\rangle$ in *stage* j, S_{k+1}^j is a canonical generator set. But from Claim 4.3.13.2, the input $\langle i, S_i^j, S_{i+1}^j, \ldots, S_{p(m)}^j\rangle$ to any stage j, is such that for all $i < t \leq p(m)$, S^j_t is a canonical generator for the solvable group $\langle S^j_t \rangle$. Now, by inspecting the description of the algorithm, it follows that whenever CANONICALGENERATOR calls CHECKCOMMUTATOR with argument $\langle m, S^j_k, S^j_{k+1} \rangle$, S^j_{k+1} is a canonical generator set.

To see that the queries to L_{ig} through CANONIZE are 1-guarded, notice that calls to CANONIZE are made outside the **while**-loop. This means that CHECK-COMMUTATOR with input (m, S^j_k, S^j_{k+1}) returns YES. That is $\langle S^j_k \rangle' < \langle S^j_{k+1} \rangle$. Hence $\langle S^j_k \rangle' = \langle S^j_{k+1} \rangle$ from Claim 4.3.13.1. So it follows that calls to CANONIZE with argument $\langle S^j_i, S^j_{i+1}, \ldots, S^j_{p(m)} \rangle$ will be such that S^j_l for $i \leq l \leq p(m)$ will generate the commutator series of S^j_i for all i. It follows from Theorem 4.3.12 that queries to L'_2 will be 1-guarded.

Finally, we show that the above algorithm on input (m, S), outputs a canonical generator set for the group $G = \langle S \rangle$ if G is solvable and outputs NOT SOLVABLE otherwise. Now, observe that if $H_1 < H_2$ are two finite groups, $H'_1 < H'_2$. Hence it follows from Claim 4.3.13.1 that, $\langle S^j_i \rangle < G_i$ for any i at any stage j, where G_i is the i^{th} element in the commutator series of G. We know that after the execution of $2p^3(m)$ stages, the algorithm outputs either a set $X \subseteq B_m$ or NOT SOLVABLE. Suppose it outputs NOT SOLVABLE in stage j. This happens after the value of the variable inside the **while**-loop is assigned $p(m)$. From the description of the algorithm inside the loop, it follows that the group $\langle S^j_{p(m)} \rangle$ does not contain the commutator subgroup of $\langle S^j_{p(m)-1} \rangle$. But if G where solvable, then we know that $G_{p(m)} = \{e\}$ and since $\langle S^j_{p(m)} \rangle < G_{p(m)}$ from Claim 4.3.13.1, we have a contradiction.

Suppose the algorithm outputs a set $X \subseteq B_m$ at *line*-16 in stage j. Thus the value of the variable k is 1. Notice that, inside the **while**-loop, the value of k is only incremented. This implies that at stage j the **while**-loop is not entered (the value of i could not have become 0 at a previous stage). So input to stage j is $\langle 1, S^j_1, \ldots, S_{p(m)} \rangle$. From Claim 4.3.13.2, it follows that for all $2 \leq t \leq p(m)$, $\langle S^j_t \rangle$ is solvable and S^j_t is a canonical generator set for the group $\langle S^j_t \rangle$. From the value of g =YES and Claim 4.3.13.1, it follows that $\langle S_1^j\rangle' = \langle S_2^j\rangle$. Also, since $S_1^j = S$ for any stage j, it follows that S_i^j generates the i^{th} element in the commutator series of $\langle S\rangle = G$. Hence, from Theorem 4.3.12, it follows that $[\textsc{Canonize}(S_1^j, \ldots, S_{p(m)}^j)]_1$ is a canonical generator set for G. This completes the proof of the the theorem.

■

4.4 Solvable Group Problems are in SPP

Finally, in this section, we show that all the three basic problems over solvable groups we consider here are in SPP. Notice that, the problem of checking whether a black-box group is solvable or not is in the class SPP follows from Theorem 4.3.11 and Lemma 2.0.2. Now we show how the algorithm CANONICALGENERATOR can be modified to get the required upper bound.

Theorem 4.4.1 *Over any group family $\mathcal{B}$, the problems* Membership Testing, Order Verification, Group Isomorphism *over the subclass of solvable groups are in* SPP *and hence low for the classes* PP, C$_=$P*and* Mod$_k$P *for $k \geq 2$.*

Proof. The proof is very similar to the upper bound proofs for abelian groups given in Theorem 4.2.16. Consider Membership Testing. Let $\mathcal{B}$ be a group family. Consider an oracle NP machine M which on input $\langle m, S, g\rangle$ converts the set S to a canonical generator set if S generates a solvable group, by simulating the algorithm CANONICALGENERATOR on input $\langle m, S\rangle$. Let S' be the canonical generator set thus constructed. Then M makes one query $\langle m, S', g\rangle$ to the language L_2' and it accepts if the answer is 'YES'. If the answer to the query is 'NO' then M branches into an accepting and a rejecting path, thus producing a zero *gap*. So, *gap* produced by M is 0 if $g \notin \langle S'\rangle = \langle S\rangle$ and *gap* is 1 if $g \in \langle S'\rangle = \langle S\rangle$. Since S' is a canonical generator set, It follows that M makes only 1-*guarded* queries to L_{ig}. Also, M

runs in time polynomial in the input size. Hence from Lemma 2.0.2, it follows that Membership Testing$\in$ SPP.

To show that Order Verification is in SPP, notice that the algorithm CANONICALGENERATOR can be easily modified to get an algorithm M_{ov} so that on input $\langle m, S\rangle$, it computes $|\langle S\rangle|$, by making only 1-guarded queries to L_{ig}. Hence, again by Lemma 2.0.2, it follows that Order Verification is in SPP.

Finally we show that Group Isomorphism for solvable groups is in SPP. Let $\langle m, S_1, S_2\rangle$ be an instance of Group Isomorphism. Let Let $\langle S_1\rangle = G_0 > G_1 > \ldots > G_k = \langle e\rangle$ and $\langle S_2\rangle = H_0 > H_1 > \ldots > H_k = \langle e\rangle$ be the two commutator series. $\langle S_1\rangle$ and $\langle S_2\rangle$ are isomorphic iff for each $i : 1 \leq i \leq k$, the abelian factor groups G_{i-1}/G_i and H_{i-1}/H_i are isomorphic. G_{i-1}/G_i and H_{i-1}/H_i are isomorphic if the types of G_{i-1}/G_i and H_{i-1}/H_i are the same. Now, the algorithm CANONICALGENERATOR can be modified to get a polynomial-time oracle Turing machine M_{is} which computes the type of each of the abelian factor groups involved in the commutator series of the solvable groups given in the input, making only 1-guarded queries to L_{ig} and producing a *gap*=1 if $\langle S_1\rangle$ is isomorphic to $\langle S_2\rangle$ and a *gap*=0 otherwise. It follows that Group Isomorphism is in SPP. ■

4.5 Summary

In this chapter we considered the complexity of three basic computational problems over solvable black-box groups namely Membership Testing, Order Verification and Isomorphism Testing. These problems are neither known to be in P nor known to be hard for NP. We showed that these problems are in the low complexity counting class SPP. Hence these problems are low for many counting classes and unlikely to be hard for NP. The upper bound is built upon a constructive version of the fundamental theorem of finite abelian groups. For extending the result from abelian groups to solvable groups, we extend the concept of an independent generator set of an abelian group to a similar, special kind of generator set (we call it canonical generator set) for solvable groups.

This concludes the first part of the thesis. In the next chapter, we will study the problem of computing a generator set for a group if we are provided with a membership testing oracle. We pose this problem in learning-theoretic framework and study its complexity in the *teaching assistant* model of learning.

Chapter 5

Exact Learning via Teaching Assistants

5.1 Introduction

In this and the next chapter we focus on the issue of classifying the representation classes with respect to the complexity of exactly learning them. For this purpose, we propose a refinement of Angluin's model of learning, called the teaching assistant model of exact learning. Here we give a brief survey on the known results in this area and motivate the need for the new model.

In Angluin's model, a natural method for quantifying the complexity of a representation class is to consider the type and number of queries that a learner has to ask the teacher in order to learn any concept in the class. Hence, characterizing the power of various queries (mainly membership and equivalence) in Angluin's model has been of interest among researchers. Angluin [Ang90] showed that polynomially many equivalence queries are insufficient for learning representation classes which have a combinatorial property called approximate fingerprints. Applying this result she showed that deterministic finite automata (DFA), context free languages (CFL), conjunctive normal form formulas (CNF) and disjunctive normal form formulas (DNF) are not learnable with polynomially many equivalence queries. In [Gav93], Gavaldà proved that the nonexistence of approximate fingerprints is actually a sufficient condition for learnability with polynomially many equivalence queries. In particular, he showed that the representation class of CIRCUITS (boolean functions represented by logical circuits) are polynomial-time equivalence query learnable with the computational aid of an oracle in polynomial-time hierarchy. This shows that CIRCUITS are, in a sense, easier to learn than classes like DFAs, NFAs, CFGs, CNFs or DNFs. An improvement on the complexity of the oracle used for learning CIRCUITS is given in [BCG$^+$96]. More recently, Hellerstein et al. [HPRV96] have shown that a representation class is learnable with polynomially many membership and equivalence queries, if and only if the representation class has *polynomial-size certificates*. A similar characterization has also been given in [Heg95].

Another interesting approach towards analyzing the complexity of exact learning via queries was proposed by Watanabe and Gavaldà [Wat90, WG94]. They used ideas from structural complexity theory for this purpose. In [WG94], a notion of *machine types* is defined in order to capture various types of queries in Angluin's model. The authors also show a close relation between the polynomial-time query learnability of representation classes and the complexity of some related *representation finding* problems.

The results mentioned above help us in classifying various classes according to the difficulty of exactly learning them. However, the classification could be finer. For example, how do we compare two representation classes that are both exactly learnable with polynomially many equivalence queries but not learnable with polynomially membership queries? It could be the case that one of these classes is easier than the other because the full power of equivalence queries may not be needed to learn one although it is required to exactly learn the other. Our main interest in this chapter is to explore this possibility.

In this and the next chapter we are mainly interested in the learnability of some algebraic classes in the teaching assistant model. To motivate this model, we consider the learnability of three representation classes; SYM of permutation groups, LS(p) of linear spaces over finite fields and the class 3-CNF, in Angluin's model. In the next section, we give some essential notations and definitions from computational learning theory. We also give the precise definitions of the representation classes of interest to us. Then we formally define Angluin's model and prove upper and lower bounds on the learnability of the above-mentioned representation classes in this model. Finally we give the definition of the teaching assistant model in detail and prove upper bounds on learning these algebraic classes in this model. In the next chapter we further study the learnability of various subclasses of the above-mentioned representation classes and prove some absolute lower bounds on the learnability of these classes in this new model.

5.2 Learning Theory: Notations and Definitions

A *representation class* $\mathcal{P}$ is a tuple $\langle R, \mu, t\rangle$ where t is a polynomial, $R \subseteq \Sigma^*$ and μ is a collection $\mu = \{\mu_n\}_{n\in\mathbf{N}}$ such that $\mu_n : R \to 2^{\Sigma^{t(n)}}$ is a many-one mapping. Any element in the range of μ is called a *concept* of $\mathcal{P}$. The set of all concepts of $\mathcal{P}$ is called the *concept class* represented by the representation class $\mathcal{P}$. When there is no confusion, we denote the concept class represented by $\mathcal{P}$ also by $\mathcal{P}$. For any n, let $\mathcal{P}_n$ denote the concepts of $\mathcal{P}$ in $\Sigma^{t(n)}$. For any concept $c \in \mathcal{P}_n$ let $size(c)$ denote $\min\{|r| : \mu_n(r) = c\}$. Let $\mathcal{P}_{n,m}$ denote the concepts of $\mathcal{P}$ in $\Sigma^{t(n)}$ which have representations of size less than or equal to m. For two representation classes $\mathcal{P} = \langle R, \mu, t\rangle$ and $\mathcal{P}' = \langle R', \mu', t'\rangle$, $\mathcal{P}'$ is said to be a *subclass* of $\mathcal{P}$, denoted by $\mathcal{P}' \subseteq \mathcal{P}$, if $R' \subseteq R$, $t' = t$, and $\mu' = \mu$ when restricted to R'. A representation class is *honest* if the following properties hold: given a string $y \in \Sigma^*$ and $n \in \mathbf{N}$, it can be checked in polynomial time whether y is a valid representation of some concept in $\mathcal{P}_n$. Given input $\langle 0^n, r, x\rangle$, it can be decided in polynomial time whether x is in $\mu_n(r)$. In this thesis, we will be interested only in honest representation classes.

Here, we give the precise definitions of three representation classes of interest to us. In Chapter 6, we also consider some subclasses of these representation classes.

Representation classes SYM, LS(p) and 3-CNF

The *symmetric group* S_n consists of all permutations on $\{1,\ldots,n\}$ with permutation composition as group operation. Subgroups of S_n are called *permutation groups.* Now let us fix the encoding of permutations. Let the n-tuple $\sigma = (i_1, \ldots, i_n)$ represents the permutation $j \to i_j$ in S_n. Let $m = \lceil logn \rceil$ and the binary representation of i_j be $x_{j1} \ldots x_{jm}$. Then we represent σ with the binary string

$$x_{11}x_{11}\ldots x_{1m}x_{1m}01x_{21}x_{21}\ldots x_{2m}x_{2m}01\ldots 01x_{n1}x_{n1}\ldots x_{nm}x_{nm}$$

of length $2nm + 2(n-1)$. It is clear that the group operations can be performed in time linear in n.

Let SYM $= \langle R, \{\mu_n\}_{n\geq 1}, t\rangle$ denote the representation class of permutation groups where $R = \cup_{n\geq 1} R_n$, and $r \in R_n$ encodes a set of permutations of S_n, namely, it is a concatenation of some strings, each encoding a permutation as explained above. For $r \in R_n$, $\mu_n(r)$ is the subgroup of S_n generated by permutations encoded in r. Here, $t(n) = 2n\lceil \log n \rceil + 2(n-1)$. In [FHL80], it is shown that checking for membership of an element in a permutation group given by a generator set can be done in polynomial time. Hence it follows that the representation class SYM is honest.

Now we define the class LS(p). For a fixed prime p, let F_p denote the finite field of size p. Let the n-fold direct sum of F_p be denoted by F_p^n. A vector $u \in F_p^n$ is represented as an n-tuple $(u[1], \ldots, u[n])$ for $u[i] \in F_p$. We can use the pairing function as in the case of permutation groups to encode this tuple whose length is $t(n) = 2n\lceil \log p \rceil + 2(n-1)$. LS($p$) contains subspaces of F_p^n represented by a set of vectors which spans the subspace. More formally, LS(p) $= \langle R, \{\mu_n\}_{n\geq 1}, t\rangle$. Here $R = \cup_{n\geq 1} R_n$, where $r \in R_n$ encodes a set of vectors of F_p^n. For $r \in R_n$, $\mu_n(r)$ is the vector space spanned by the set of vectors encoded in r. Since testing whether a given vector is in the span of a set of vectors can be done in polynomial time, it follows that LS(p) is an honest representation class.

Finally, 3-CNF= $\langle R, \{\mu_n\}_{n\geq 1}, t\rangle$, where $R = \cup_{n\geq 1} R_n$, and $r \in R_n$ encodes a boolean function from $\{0,1\}^n$ to $\{0,1\}$ in conjunctive normal form where each clause has at most 3 literals. Here $t(n) = n$.

Now we are ready to give a detailed definition of Angluin's model of exact learning and consider the learnability of the above-defined representation classes in this model.

5.3 Angluin's Model of Exact Learning

Let $\mathcal{P} = \langle R, \mu, t\rangle$ be a representation class. Angluin's model of learning [Ang88] consists of a teacher and a deterministic learner. The learner α on input $\langle 0^n, 0^l\rangle$, has to output a representation $r \in R$ such that $\mu_n(r) = c$ if $size(c) \leq l$, for the *target* concept c selected by the teacher after some finite number of computation steps. During the course of learning α can make two types of queries to the teacher, namely *equivalence queries* and *membership queries*. To the queries made by the learner, the teacher answers consistent with the target concept he/she has selected. To make an equivalence query, α writes down a string $h \in R$. The teacher responds to this, by answering 'Yes' if $\mu_n(h) = c$, or gives a *counter example* $x \in \mu_n(h)\triangle c$, if $\mu_n(h) \neq c$. These kinds of equivalence queries are called *proper* equivalence queries. If the query y made by the learner is not in R, then y is said to be an *improper* equivalence query. In this thesis, by an equivalence query we always mean a proper equivalence query.

To make a membership query, α writes down a string $x \in \Sigma^{t(n)}$. The teacher responds to this, by answering 'Yes' if $x \in c$ and 'No' if $x \notin c$. The learner's output is unspecified if $size(c) > l$. This kind of learning is called *bounded learning* as opposed to unbounded learning where the length parameter l is not given to the learner.

Now we look at some standard complexity measures on resources used by the learner. Let $\mathcal{P}$ be a representation class. Then $\mathcal{P}$ is said to be polynomial-query learnable, if there exists a polynomial q and a learner α, such that for all n and l, and for all $c \in \mathcal{P}_n$, the learner on input $\langle 0^n, 0^l \rangle$ with c as target concept such that $size(c) \leq l$, outputs a representation of c by making at most $q(n+l)$ queries to the teacher. $\mathcal{P}$ is said to be polynomial-time learnable, denoted by FP-learnable, if there is a polynomial q and a learner α, such that for all n and l, and for all $c \in \mathcal{P}_n$, the learner on input $\langle 0^n, 0^l \rangle$ with c as target concept such that $size(c) \leq l$, outputs a representation of c within $q(n+l)$ time steps.

Notice that any upper bound on the time complexity is an upper bound on the query complexity. Hence if $\mathcal{P}$ is polynomial-time learnable then it is also polynomial-query learnable.

Before we go into the details of our results, we briefly explain the need for considering bounded learning as opposed to unbounded learning. In unbounded learning the learner is not given the length parameter l as a part of the input. Hence for polynomial-time unbounded learnability the learner is allowed to run for a number of time steps polynomial in $n + size(c)$. Though bounded learnability is weaker than unbounded learnability, we restrict ourselves to bounded learnability because it is difficult to deal with learners whose running time is sensitive to the output, particularly when we consider learners which are nondeterministic oracle Turing machines. (We will be dealing with nondeterministic learners as a technical tool in the next chapter. The exact definition of these learners are given there.) For instance, consider in the unbounded learning setting, a nondeterministic learner α learning a representation class $\mathcal{P}$. Let α take 0^n as input and run in time $q(n + size(c))$ where $c \in \mathcal{P}_n$ is the target concept. We can ensure that nondeterministic computation paths of α which outputs a representation of c are of length exactly $q(n + size(c))$. But on those computational paths where α is not going to output a string, cannot be timed to run exactly $q(n + size(c))$ steps, because the machine simply does not know when to stop on that path.

Bounded and unbounded learnability coincide for those representation classes $\mathcal{P}$

which have representations of size bounded by a polynomial in n for all concepts in $\mathcal{P}_n$, (in a complexity-theoretic sense $\mathcal{P}$ can be viewed as a subclass of the nonuniform complexity class P/poly). We note that both the classes SYM and LS(p) have this property. To see this for the class SYM, we recall Lagrange's theorem which states that if G is a finite group and $H < G$ then $|G|$ is divisible by $|H|$. It can be easily deduced from Lagrange's theorem that every subgroup $G < S_n$ has a generator set of cardinality bounded by $n \log n$. Hence, each concept $c \in \mathrm{SYM}_n$ has a representation of size polynomial in n. It holds for LS(p) also. As a consequence, bounded learnability coincides with unbounded learnability when we consider polynomial-time learnability of subclasses of SYM and LS(p). So, in this case we can do away with the parameter 0^l from the learner's input which denotes the bound on the size of the target concept.

5.3.1 Complexity of Learning SYM, LS(p) and 3-CNF in Angluin's Model

In this subsection we show both upper and lower bounds on the learnability of the classes SYM, LS(p) and 3-CNF in Angluin's Model. While the upper bound is obtained by a simple application of Lagrange's theorem, for proving the lower bound we need to count the number of cyclic groups generated by p-cycles of S_p for prime p.

An m-cycle is a permutation $\sigma \in S_n$ such that there are indices $i_1, \ldots, i_m$, $1 \leq i_j \leq n$: $\sigma(i_j) = i_{j+1 (\mathrm{mod}\ m)}$, and $\sigma(i) = i$ if $i \neq i_j$ for $1 \leq j \leq m$. It is easy to note that the order of any m-cycle is m. We have the following proposition on the number of cyclic groups generated by p-cycles. See the appendix for a proof.

Proposition 5.3.1 *Let p be a prime. Then there are $(p-2)!$ cyclic subgroups of S_p generated by p-cycles.*

Theorem 5.3.2 *The representation classes* SYM, LS(p) *and* 3-CNF *are*

1. FP*-learnable using only equivalence queries in Angluin's model.*

2. *not* FP*-learnable using only membership queries in Angluin's model.*

Proof. (1). It is shown in [Ang88] that the class 3-CNF is FP-learnable using only equivalence queries. We give here a learner for the class SYM which makes at most $n \log n$ queries to output a generator set for a subgroup of S_n. A learner for LS(p) which makes at most n queries to output a basis for a subspace of F_p^n can be designed very similarly. We omit the proof of this.

We give the description of the learner EquivLearn for SYM. As mentioned earlier, we omit the length parameter l from the input to the learner.

```
EquivLearn($0^n$)
1  $S \leftarrow \{e\}$;
2  while Answer to query $S$ is 'No'
3  do $g \leftarrow$ COUNTER EXAMPLE ;
4       $S \leftarrow S \cup \{g\}$;
5  return $S$.
```

Consider the i^{th} iteration of the **while** loop. Let $S = \{g_1, \ldots, g_{i-1}\}$ be the generator set constructed so far. From the fact that G is a group and the identity $e \in G$, it follows that $\langle S \rangle < G$. So the counterexample g_i given by the teacher at the i^{th} iteration is in G but not in $\langle S \rangle$ thereby growing the generator set by one. To show that the **while**-loop is executed at most $n \log n$ times, it is enough to show that at most $n \log n$ counter examples are sufficient. Form Lagrange's theorem it follows that if G and H be two groups such that $H < G$ and $g \in G$ but $g \notin H$, then $|\langle H, g \rangle| \geq 2|H|$. Hence addition of an element into the set S, grows the size of the group $\langle S \rangle$ by at least twice the previous size. The result follows.

(2). Now we show that none of these classes are FP-learnable using only membership queries. Note that to show this it is enough to show that these are not polynomial query learnable using only membership queries. Again, we prove it for the class SYM. The proof is by an adversary argument. Suppose there is a learner α which on input 0^n has to output a generator set for a subgroup S_n by making only membership queries. Let s be the polynomial bounding the number of queries asked by α. Choose large enough prime p such that $(p-2)! > s(p)+1$. The adversary keeps all the $(p-2)!$ p-cyclic groups with him. Let $\{q_1, q_2 \ldots, q_{s(p)}\}$ be the set of membership queries made by α. For each membership query $q_i \neq e$ by α, the adversary answers 'No', and the adversary answers 'Yes' if $q_i = e$. Now, from Proposition 5.3.1, and the fact that $(p-2)! > s(p)+1$, it follows that the adversary can consistently answer in this manner and still there will be at least 2 p-cyclic groups which are consistent with the answers to the queries. Hence the learner will be unable to decide which one of the cyclic groups to output.

A very similar adversary argument can be given for showing the lower bounds for both LS(p) and 3-CNF (in the case of LS(p) the adversary keeps all 1-dimensional liner spaces of F_p^n and in the case of 3-CNF the adversary can keep all the conjunction of n literals). ■

These upper and lower bound results indicate that the complexity of exactly learning these three representation classes in Angluin's model are very similar. Our next aim is to distinguish the learnability of these classes. The intuition is that the classes SYM and LS(p) are algebraic and hence by exploiting the structure of these classes, we may be able to get better upper bounds on their learnability. For this purpose, in the next section we introduce a new exact learning model called the teaching assistant model of exact learning. Then in the last section of this chapter we prove new upper bounds on the learnability of these classes in this new model.

5.4 The Teaching Assistant Model of Exact Learning

Before we give formal definitions of the main ideas in the new model, we give some intuitive explanations. As mentioned earlier, apart from the learner and the teacher the new ingredient of this model is an intermediate agent between the learner and the teacher, called a teaching assistant. We define a teaching assistant as a set of strings whose membership in the set depends on the concept of interest. Then we define the notion of oracle Turing machines accepting these sets. These oracle Turing machines are allowed to make membership queries to the concept of interest (the concept fixed by the teacher). By varying the acceptance criteria of these Turing machines, we get different teaching assistant classes. We will be interested in acceptance criteria which are analogous to known complexity classes. A learner learning a representation class can ask membership queries to a pre-defined teaching assistant. Now the complexity of a representation class can be quantified by the complexity of the teaching assistant using which a learner (usually a deterministic oracle Turing machine) can learn the concept (output a representation for the concept). We move onto the formal definitions of the teaching assistant model.

Definition 5.4.1 Let $\mathcal{P}$ be a representation class. Let $Q : \mathrm{N}\times\mathrm{N}\times\Sigma^*\times 2^{\Sigma^*} \rightarrow \{0,1\}$ be a predicate. A tuple $\langle n,l,x,c\rangle \in \mathrm{N}\times\mathrm{N}\times\Sigma^*\times 2^{\Sigma^*}$, is said to be $\mathcal{P}$-valid if $c \in \mathcal{P}_n$ and $size(c) \leq l$. The *teaching assistant* defined by Q w. r. t. the representation class $\mathcal{P}$ is the set
$L(\mathcal{P}) = \{\langle n,l,x,c\rangle | \langle n,l,x,c\rangle$ is P-valid and $Q(n,l,x,c)\}$.

In the above definition notice that, consistent with the notion of bounded learnability, we have the parameter l as part of $\mathcal{P}$-valid tuples.

Now we formalize the notion of learning via teaching assistants. Let $\mathcal{P} = \langle R,\mu,t\rangle$ be a representation class and $L(\mathcal{P})$ be a teaching assistant for $\mathcal{P}$. As in Angluin's model, learners are deterministic oracle Turing machine transducers. The learning algorithm comprises of a learner and teaching assistant pair. The teacher selects a target concept $c \in \mathcal{P}_n$. The learner α, on input $\langle 0^n, 0^l \rangle$, has to output a representation $r \in R$ such that $\mu_n(r) = c$ and $size(c) \leq l$, after some finite steps of computation. During the course of the computation α can query its teaching assistant, say $L(\mathcal{P})$. Notice that when α wants to query the teaching assistant about $\langle n, l, x, c \rangle$, it is enough that α communicates x to the teaching assistant, since the other parameters are implicit. For such a query x by the learner, the learner receives the answer 'Yes' if $\langle n, l, x, c \rangle \in L(\mathcal{P})$ and 'No' if $\langle n, l, x, c \rangle \notin L(\mathcal{P})$. $\mathcal{P}$ is said to be learnable with assistant $L(\mathcal{P})$ if there is a learner α for $\mathcal{P}$ with queries to $L(\mathcal{P})$ as teaching assistant and we say α learns $\mathcal{P}$ using teaching assistant $L(\mathcal{P})$.

Let $\mathcal{P}$ be a representation class. Let α be a learner which learns $\mathcal{P}$ using a teaching assistant $L(\mathcal{P})$. For an input $\langle 0^n, 0^l \rangle$ and c as the target concept, the time complexity of the learner is the number of steps it takes before it writes down a representation for c. $\mathcal{P}$ is said to be deterministic polynomial-time learnable (in short, FP-learnable) with a teaching assistant $L(\mathcal{P})$, if there is a polynomial q and a deterministic learner α learning $\mathcal{P}$ using $L(\mathcal{P})$ such that for all n, and for all $c \in \mathcal{P}_n$, on input $\langle 0^n, 0^l \rangle$, the time complexity of α is bounded by $q(n+l)$.

In order to quantify the complexity of teaching assistants we define certain teaching assistant classes with respect to a given representation class. The teaching assistant classes we define here are the ones analogous P, NP, Σ_2^p, SPP and LWPP.

Definition 5.4.2 Let $\mathcal{P}$ be a representation class and $L(\mathcal{P})$ be a teaching assistant.

1. $L(\mathcal{P})$ is said to be in the class $\Sigma_2^p(\mathcal{P})$ if there exists a polynomial-time Σ_2 oracle machine which for any $\mathcal{P}$-valid tuple $\langle n, l, x, c \rangle$, on input $\langle 0^n, 0^l, x \rangle$ uses c as oracle and accepts $\langle 0^n, 0^l, x \rangle$ if and only if $\langle n, l, x, c \rangle \in L(\mathcal{P})$

2. $L(\mathcal{P})$ is said to be in the class $\mathrm{P}(\mathcal{P})$ if there exists a polynomial-time deterministic oracle Turing machine M which for any $\mathcal{P}$-valid tuple $\langle n, l, x, c \rangle$, on

input $\langle 0^n, 0^l, x\rangle$ uses $c \in \mathcal{P}_n$ as oracle and accepts $\langle 0^n, 0^l, x\rangle$ if and only if $\langle n, l, x, c\rangle \in L(\mathcal{P})$.

3. $L(\mathcal{P})$ is said to be in the class NP($\mathcal{P}$) if there exists a polynomial-time nondeterministic oracle Turing machine M which for any $\mathcal{P}$-valid tuple $\langle n, l, x, c\rangle$, on input $\langle 0^n, 0^l, x\rangle$ uses $c \in \mathcal{P}_n$ as oracle and accepts $\langle 0^n, 0^l, x\rangle$ if and only if $\langle n, l, x, c\rangle \in L(\mathcal{P})$.

4. $L(\mathcal{P})$ is said to be in the class SPP($\mathcal{P}$) if there exists a polynomial-time nondeterministic oracle Turing machine M which for any $\mathcal{P}$-valid tuple $\langle n, l, x, c\rangle$, on input $\langle 0^n, 0^l, x\rangle$ uses c as oracle and produces a *gap*=1 if $\langle n, l, x, c\rangle \in L(\mathcal{P})$ and *gap*=0 if $\langle n, l, x, c\rangle \notin L(\mathcal{P})$.

5. $L(\mathcal{P})$ is said to be in the class LWPP($\mathcal{P}$) if there exists a polynomial-time computable function f and a polynomial time nondeterministic oracle Turing machine M which for any $\mathcal{P}$-valid tuple $\langle n, l, x, c\rangle$, on input $\langle 0^n, 0^l, x\rangle$ uses c as oracle and produces a *gap*=$f(n)$ if $\langle n, l, x, c\rangle \in L(\mathcal{P})$ and *gap*=0 if $\langle n, l, x, c\rangle \notin L(\mathcal{P})$.

In all the above definitions, the behavior of machines accepting the teaching assistant is not specified on inputs which are not $\mathcal{P}$-valid.

The following containments among teaching assistant classes follow directly from the above definitions.

Proposition 5.4.3 *For any representation class $\mathcal{P}$*

- $\mathrm{P}(\mathcal{P}) \subseteq \mathrm{SPP}(\mathcal{P}) \subseteq \mathrm{LWPP}(\mathcal{P})$.
- $\mathrm{P}(\mathcal{P}) \subseteq \mathrm{NP}(\mathcal{P}) \subseteq \Sigma_2^p(\mathcal{P})$

Finally, we have the following definition of learning with an assistant from a particular teaching assistant class.

Definition 5.4.4 Let $\mathcal{P}$ be a representation class and $\mathcal{C}$ be any assistant class defined with respect to $\mathcal{P}$. We say a representation class $\mathcal{P}$ is FP-learnable with a $\mathcal{C}$-assistant if there exists a teaching assistant $L(\mathcal{P}) \in \mathcal{C}(\mathcal{P})$ and a learner α, such that $\mathcal{P}$ is FP-learnable with $L(\mathcal{P})$.

Remark. The notion of learning via teaching assistants is somewhat similar to the notion of self-producible circuits studied in [GW93]. (This is a generalization of self-p-producible circuits introduced by Ko [Ko85] and further studied by Balcázar and Book [BB86].) For comparing the two notions, we give some definitions. For any reltivizable function complexity class $\mathcal{F}$, a language $L \in$ P/poly is said to have $\mathcal{F}$-self-producible circuits, if an advice for L can be computed in $\mathcal{F}(L)$. The motivation for this definition was to study the relative complexity of producing circuits for languages in P/poly. The main focus in [GW93] was on proving lower bounds. For example, it is shown that there exists a language $A \in$ P/poly which is not FP-self-producible. We are interested in producing representations (learning) for a class of concepts (as oppose to a language in [GW93]). We need learners which work for *all* the concepts in the class. Hence intuitively, it may be difficult (easier) to prove upper (respectively, lower) bound results in our framework compare to the one in [GW93]. For example, for proving the above-mentioned lower bound in [GW93], the authors have to diagonalize against all polynomial time reductions. On the other hand, an easy adversary argument is enough for showing the existence of a representation class that is not FP-learnable with membership queries.

For the sake of comparing the teaching assistant model with Angluin's model, in the next subsection we give a brief sketch of how to simulate equivalence and membership queries in the new model.

5.4.1 Comparison with Angluin's model

In this section we first prove a universal upper bound on the learnability of any representation class. Then we show the relationship between Angluin's model and the teaching assistant model for specific assistant classes.

Theorem 5.4.5 *For any representation class $\mathcal{P}$, $\mathcal{P}$ is* FP*-learnable with a Σ_2^p-assistant.*

Proof Let $\mathcal{P} = \langle R, \mu, t\rangle$ where $\mu = \{\mu_n\}_{n\geq 1}$ be any representation class. Define a teaching assistant as follows: $L(\mathcal{P}) = \{\langle n, l, x, c\rangle$ is $\mathcal{P}$-valid$|\ \exists y$ such that $|xy| \leq l$ and $\mu_n(xy) = c\}$. We define a Σ_2^p machine M accepting $L(\mathcal{P})$. M on input $\langle 0^n, 0^l, x\rangle$ existentially guesses y of length bounded by $l - |x|$ and then universally verifies whether $\mu_n(xy) = c$ by universally guessing a string $z \in \Sigma^{t(n)}$ and verifying whether $z \in \mu_n(xy)$ if and only if $z \in c$. Since $\mathcal{P}$ is honest, checking membership in $\mu_n(xy)$ can be done in polynomial time, given xy. Membership of z in c can be done by querying the oracle c.

Now we give the description of the learner α for $\mathcal{P}$ which prefix searches for a representation of the target concept using teaching assistant $L(\mathcal{P})$. Let $c \in \mathcal{P}_n$ be the target concept and l be such that $size(c) \leq l$.

```
Σ₂ᵖ-LEARNER($0^n, 0^l$)
1  $i \leftarrow l$;
2  while $\langle n, i-1, \lambda, c\rangle \in L(\mathcal{P})$
3  do $i \leftarrow i - 1$;
4  $x \leftarrow \lambda$;
5  for $j \leftarrow 1$ to $i$
6  do if $\langle n, i, x1, c\rangle \in L(\mathcal{P})$
7         then $x \leftarrow x1$;
8         else $x \leftarrow x0$;
9   Output $x$ .
```

We now compare Angluin's model with the teaching-assistant model. We show that membership queries are characterized by P-assistants. On the other hand, the combination of membership and equivalence queries can be simulated with NP-assistants. However, we show that NP-assistants are strictly more powerful than the combination of membership and equivalence queries. In fact we prove in this section that it is not possible to capture the power of proper equivalence queries in the teaching-assistant model.

Theorem 5.4.6 *For any representation class $\mathcal{P}$, it is* FP*-learnable with membership queries if and only if it is* FP*-learnable with a* P*-assistant.*

Proof Let α be an FP-learner for $\mathcal{P} = \langle R, \mu, t\rangle$ where $\mu = \{\mu_n\}_{n\geq 1}$ which uses membership queries to the teacher. Define a teaching assistant $L(\mathcal{P})$ as follows. $L(\mathcal{P}) = \{\langle n, l, x, c\rangle | x \in c\}$. It is obvious that $L(\mathcal{P}) \in \mathrm{P}(\mathcal{P})$. Let $c \in \mathcal{P}_n$ be the target concept and let l be such that $size(c) \leq l$. Consider a learner α' which behaves as follows. α' on input $\langle 0^n, 0^l\rangle$ simulates α on input$\langle 0^n, 0^l\rangle$. Whenever α makes a membership query x, α' makes a query $\langle n, l, x, c\rangle$ to $L(\mathcal{P})$ and treats the answer as the answer to the membership query for α and continues the simulation of α. It is easily verified that α' outputs a representation of c in time polynomial in $n + l$, if and only if α outputs a representation of c in time polynomial in $n + l$.

To show the other containment, let β be a FP-learner which learns $\mathcal{P}$ using teaching assistant $L(\mathcal{P}) \in \mathrm{P}(\mathcal{P})$. Let M be the polynomial-time oracle Turing machine witnessing $L(\mathcal{P}) \in \mathrm{P}(\mathcal{P})$. Let $c \in \mathcal{P}_n$ be the target concept and let l be such that $size(c) \leq l$. Consider a learner β' which on input $\langle 0^n, 0^l\rangle$ and c as target concept, simulates β. Whenever β makes a query $\langle n, l, x, c\rangle$ to $L(\mathcal{P})$, β' writes down $\langle 0^n, 0^l, x\rangle$ on its work tape and start simulating M on input $\langle 0^n, 0^l, x\rangle$. During this simulation of M, whenever M makes a membership query y to c, β' makes a membership query y to the teacher and treats the answer to this as the answer to the query y of M. Now, β' outputs a representation of c in time polynomial in $n + l$, if and only if β outputs the same representation of c in time polynomial in $n + l$.

Theorem 5.4.7 *For any representation class $\mathcal{P}$, If $\mathcal{P}$ is* FP*-learnable with equivalence and membership queries, then $\mathcal{P}$ is* FP*-learnable with an* NP*-assistant.*

Proof. Let α be an FP-learner for $\mathcal{P} = \langle R, \mu, t\rangle$ where $\mu = \{\mu_n\}_{n\geq 1}$ which uses membership and equivalence queries to the teacher. Define a teaching assistant $L(\mathcal{P})$ as follows.

$L(\mathcal{P}) = \{\langle n, l, x, c\rangle |$ if $x = 1y$ implies $y \in c$, and if $x = 0y\#z$ implies there is a $w \in \Sigma^{p(n)-|y|}$ such that $yw \in c\triangle\mu_n(z)\}$.

It is easy to see that $L(\mathcal{P}) \in \mathrm{NP}(\mathcal{P})$, since $\mathcal{P}$ is honest. Let $c \in \mathcal{P}_n$ be the target concept and l be such that $size(c) \leq l$. Consider a learner α' which behaves as follows. α' on input $\langle 0^n, 0^l\rangle$ simulates α on input $\langle 0^n, 0^l\rangle$. Whenever α makes a membership query x, α' makes a query $\langle n, l, 1x, c\rangle$ to $L(\mathcal{P})$ and takes the answer as answer to the membership query of α and proceeds. Whenever α makes an equivalence query x, α' first makes $\langle n, l, 0\#x, c\rangle$ to $L(\mathcal{P})$. If the answer to this query is 'No', then α' outputs x. Otherwise α' uses $L(\mathcal{P})$ to prefix search for a string $u \in c\triangle\mu_n(x)$ and uses u as counterexample and proceeds. It is easy to see that α' outputs a representation of c in time polynomial in $n + l$, if and only if α outputs a representation of c in time polynomial in $n + l$. ■

Let $\mathcal{P}$ and $\mathcal{P}'$ be concept classes such that $\mathcal{P}'$ is a subclass of $\mathcal{P}$. Although it intuitively appears reasonable that $\mathcal{P}'$ should be as easy to learn as $\mathcal{P}$, such is not the case in Angluin's model [Ang88]. While 3-CNF is FP-learnable with only equivalence queries, the subclass of 3-CNF consisting of concepts of cardinality exactly 1 (SINGLETONS) is not FP-learnable with membership and equivalence queries. It is known that this anomaly is caused by equivalence queries: more precisely, the anomaly arises from the fact that in Angluin's model equivalence queries must be representations of concepts that *belong* to the concept class. On the other hand, we observe that in the teaching-assistant model this anomaly does not arise by virtue of the fact that the teaching assistant makes only membership queries to the teacher. We state this as a proposition. The proof is by a very straightforward adversary argument.

Proposition 5.4.8 [Ang88] *There exist honest representation classes $\mathcal{P}$ and $\mathcal{P}'$ such that $\mathcal{P}'$ is a subclass of $\mathcal{P}$, and $\mathcal{P}$ is* FP*-learnable with equivalence queries but $\mathcal{P}'$ is not* FP*-learnable with equivalence and membership queries in Angluin's model.*

Remark. We observe here that the teaching assistant model does not have the above anomaly. It follows from the definitions of a subclass and learning with teaching assistants that if $\mathcal{P}$ is FP-learnable with a $\mathcal{C}$-assistant ($\mathcal{C}$ is any of the above-defined teaching assistant classes) then $\mathcal{P}'$ is also FP-learnable with a $\mathcal{C}$-assistant. Hence, any upper bound on learning $\mathcal{P}$ is an upper bound on learning $\mathcal{P}'$ and any lower bound on learning $\mathcal{P}'$ is a lower bound on learning $\mathcal{P}$. We summarize this as a proposition.

Proposition 5.4.9 *Let $\mathcal{P}$ and $\mathcal{P}'$ be representation classes such that $\mathcal{P}'$ is a subclass of $\mathcal{P}$. Let $\mathcal{C}$ be any of the above-defined teaching assistant classes. Then if $\mathcal{P}$ is* FP*-learnable with a $\mathcal{C}$-assistant, $\mathcal{P}'$ is also* FP*-learnable with a $\mathcal{C}$-assistant.*

From Propositions 5.4.8 and 5.4.9, we have the following interesting corollary.

Corollary 5.4.10 *There is no teaching assistant class $\mathcal{C}$ such that the teaching assistant model captures equivalence query learnability in Angluin's model.*

In the next section we prove upper bounds on the learnability of the two algebraic concepts SYM and LS(p) in the teaching assistant model.

5.5 Upper Bounds on Learning SYM and LS(p)

In this section, we show some of the main results of this part of the thesis. First we show that the representation class SYM is FP-learnable with an LWPP-assistant.

Then we show that the class LS(p) is FP-learnable with an SPP-assistant. These results indicate that LS(p) may be easier to learn that the class SYM. Finally we show a lower bound on the learnability of the class 3-CNF. More precisely we show that 3-CNF is not FP-learnable with an LWPP-assistant (SPP-assistant) unless NP $\subseteq$ LWPP (respectively NP $\subseteq$ SPP).

Remark. We would like to observe here that as in the case of Angluin's model, without loosing any generality, the length-bound parameter l in various definitions of the teaching assistant model also can be omitted when we are considering representation classes which have short representations (polynomially bounded) for all the concepts. Since all the algebraic classes we are interested in have this property, we will not mention this parameter for various learning algorithm that we design here, either as part of the input to a learner or as part of tuples defining a teaching assistant with respect to these classes.

Learning SYM with LWPP-assistant

We prove the following theorem.

Theorem 5.5.1 *The representation class* SYM *is* FP-*learnable with an* LWPP-*assistant.*

Before we go into the proof, we give some definitions from the theory of permutation groups. Please recall the basic group-theoretic definitions given in Chapter 2.

Let $G < S_n$. For any $i; 0 \leq i \leq n$, the set $\{g \in G \mid \forall j \leq i : g(j) = j\}$ forms a subgroup of G. Denote this subgroup by $G^{(i)}$. This definition gives rise to the following chain of subgroups of G: $\{e\} = G^{(n)} < G^{(n-1)} < \cdots < G^{(0)} = G$. Now consider the left cosets of $G^{(i)}$ in $G^{(i-1)}$. It is easy to verify that two elements $g_1, g_2 \in G^{(i-1)}$ are in the same left coset of $G^{(i)}$ iff $g_1(i) = g_2(i)$. A set $T_i = \{g_{i1}, g_{i2}, \ldots, g_{id_i}\}$ of distinct (left) coset representatives of $G^{(i)}$ in $G^{(i-1)}$ is called a *transversal*. Notice that $G^{(i-1)} = \bigcup_{j=1}^{d_i} g_{ij}G^{(i)}$, where $d_i = |T_i|$. This means that any element $g \in G^{(i-1)}$ can be uniquely expressed as $g_{ij}h$ for some $h \in G^{(i)}$ and $1 \leq j \leq d_i$. It follows that the set $\bigcup_{i=1}^{n} T_i$ generates G. We call such a generator set a *strong generator set* for G.

For any set $X \subseteq \{1, 2, \ldots, n\}$, let $G_{[X]}$ denote the subgroup $\{g \in G \mid g(i) = i \ \forall i \in X\}$. For any permutation $\sigma \in S_n$ let $G_{[\sigma,X]}$ denote the set $\{g \in G \mid g(i) = \sigma(i) \ \forall i \in X\}$. Note that $G_{[\sigma,X]} = \sigma G_{[X]}$.

Now we state some useful properties of permutation groups in the following lemma. A sketch of the proof is given in the appendix.

Lemma 5.5.2 *Let $G < S_n$.*

1. *$|G^{(i)}| = \prod_{j>i} d_j$ for each i, $1 \leq i \leq n$.*
2. *The number of strong generator sets for $G^{(i)}$ is $\prod_{j=i+1}^{n} |G^{(j)}|^{d_j}$ for each i, $1 \leq i \leq n$.*
3. *Let $X \subseteq \{1, 2, \ldots, n\}$ and $\sigma \in S_n$. Then $|G_{[\sigma,X]}| = |G_{[X]}|$*

Now we formally give the proof of the Theorem 5.5.1

Proof. (*of Theorem 5.5.1*) Let G be the target group. Define the assistant $L(\mathrm{SYM})$ as:
$L(\mathrm{SYM}) = \{\langle 0^n, k, i, j_1, \ldots, j_k, G\rangle \mid \exists g \in G$ such that g fixes 1 to i; $g(i+l) = j_l$ for $1 \leq l \leq k\}$. We first show that this set defines an LWPP-assistant by constructing a nondeterministic machine which produces zero *gap* if the input string is not in the set and produces a $gap = (n!)^{2n^2+1}$ if it is in the set. Then we give a FP-learner which uses $L(\mathrm{SYM})$ as assistant for prefix searching for a strong generator set of any target group.

Claim 5.5.2.1 *There is a nondeterministic oracle machine M such that given a group $G < S_n$ as oracle it has the following behavior:*

- *If* $\langle 0^n, k, i, j_1, \ldots, j_k, G\rangle \in L(\mathrm{SYM})$ *then* M *on input* $\langle 0^n, k, i, j_1, \ldots, j_k\rangle$ *produces a gap* $= (n!)^{2n^2+1}$.
- *If* $\langle 0^n, k, i, j_1, \ldots, j_k, G\rangle \notin L(\mathrm{SYM})$ *then* M *on input* $\langle 0^n, k, i, j_1, \ldots, j_k\rangle$ *produces zero gap.*

Proof For ease of exposition, we will give a structured description of machine M. Intuitively speaking, M simulates another machine N on all computation paths on which a "correct" guess is made. We need to introduce the new machine N (invoked as subroutine by M) in order to ensure that the gap produced by M is the easily computable quantity stated in the claim. We will also further structure the description of N by designing another machine N' which is a subroutine to N. All these machines access G as oracle. The main work involved is in the design of N which has the behavior that on input $\langle 0^n, i\rangle$ it produces a *gap* $= (n!)^{2n^2+1}/|G^{(i)}|$. The following is the description of machine M.

```
DESCRIPTION OF $M(0^n, k, i, j_1, \ldots, j_k)$
1  Guess $g \in S_n$;
2  if $g \in \mu_n(r)$ and $g(j) = j;\ \forall j; 1 \le j \le i$ and $g(i+l) = j_l;\ \forall l; 1 \le l \le k$
3     then  Simulate Machine $N$ on input $\langle 0^n, i+k\rangle$ ;
4     else  Branch into two paths ;
5           accept on one path and reject on the other path ;
6  end-if
```

Now we argue that M has the claimed properties (assuming the behavior of N which we later establish). Suppose $\langle 0^n, k, i, j_1, \ldots, j_k, G\rangle \notin L(\mathrm{SYM})$. Then there exists no $g \in G^{(i)}$ such that $g(i+l) = j_l$ for $1 \le l \le k$. In this case the **if** test in *line* 2 of M fails for all guesses and M produces zero *gap*. Now, suppose $\langle 0^n, k, i, j_1, \ldots, j_k, G\rangle \in L(\mathrm{SYM})$. It follows from part 3 of Lemma 5.5.2 that the number of guessed permutations g that fulfill the **if** test is $|G^{(i+k)}|$. Now, suppose N on input $\langle 0^n, i+k\rangle$ produces $gap = (n!)^{2n^2+1}/|G^{(i+k)}|$. Since N is being simulated on $|G^{(i+k)}|$ paths, the total gap produced by M is $((n!)^{2n^2+1}/|G^{(i+k)}|) \times |G^{(i+k)}| = (n!)^{2n^2+1}$.

Now the task is to design N so that it has the desired behavior. Before we describe N we describe its subroutine N'.

DESCRIPTION OF $N'(0^n, i, j)$

```
1  Guess $g \in S_n$;
2  if $g \in G^{(i-1)}$ such that $g(i) = j$
3     then if $g \in G$
4              then accept.
5     else  reject.
6  end-if
```

DESCRIPTION OF $N(0^n, i)$

```
1  Guess an encoding $X_i = \langle T_{i+1}, \ldots, T_n \rangle$ of a strong generator set for $G^{(i)}$
2  if $X_i$ does not correctly encodes a strong generator set produce a $gap = 0$
3  for $g \in X_i$;  if $g \notin G^{(i)}$ produce a $gap = 0$
4   Using $X_i$ compute $\alpha = \prod_{n \geq k > i} |T_k|$
5   Using $X_i$ compute $\beta = \prod_{n \geq j > i} (\prod_{n \geq k > j} |T_k|)^{|T_j|}$
6   Branch into $(n!)^{n^2+1}/\alpha\beta$ and continue
7   Using $X_i$ construct the set $S_i$
8   Compute $\gamma = \prod_{n \geq j > i} (\prod_{n \geq k > j} |T_k|)^{n-j-|T_j|+1}$
9   Branch into $(n!)^{n^2}/\gamma$ paths
10  Produce a gap of $\prod_{(j,k) \in S_i} ((\prod_{n \geq l > j} |T_l|) - acc_{N'}\langle 0^n, j, k \rangle)$
```

Observe that N' has the following behavior: If there is $g \in G^{(i-1)}$ such that $g(i) = j$ then there are $|G^{(i)}|$ accepting paths, otherwise N has no accepting paths.

The machine N produces the required *gap* as follows. On input $\langle 0^n, i \rangle$, N tries to compute $|G^{(i)}|$ nondeterministically with G as oracle. For that it first guesses an encoding of a strong generator set $X_i = \langle T_{i+1}, \ldots, T_n \rangle$ of $G^{(i)}$ and verifies that it actually encodes a strong generator set of some subgroup of S_n. By making queries to G it verifies that the elements encoded in X_i are all in $G^{(i)}$. The bad case that still remains to be handled is that the guessed set X_i does not generate all of $G^{(i)}$ but only a *proper* subgroup of it. On computation paths where a generator set for a proper subgroup of $G^{(i)}$ is guessed, it must produce zero *gap*. We now give details.

Let $X_i = \langle T_{i+1}, \ldots, T_n\rangle$ where T_j, for $(i+1) \leq j \leq n$, encodes in lexicographic order a transversal of $G^{(i)}$ in $G^{(i-1)}$. (We use T_i to denote both a transversal of $G^{(i)}$ in $G^{(i-1)}$ and its lexicographic encoding.) Let $S_i = \{\langle j, k\rangle \mid (i+1) \leq j \leq k \leq n$ and $\nexists\ g \in X_i \cap G^{(j-1)}$ such that $g(j) = k\}$. (By '$g \in X_i$' we mean g is encoded in X_i.) Let K_i denotes the number of strong generator sets for $G^{(i)}$.

Let us analyze the behavior of N. It is clear from the description of N that after *line* 3 only those computation paths matter which correctly encode a subgroup of $G^{(i)}$. All other paths contribute zero *gap* to the overall *gap* produced by N. The remaining steps of N ensure that those paths on which the guessed X_i generates a proper subgroup of $G^{(i)}$ contribute zero *gap*.

First, let ρ be a computation path on which the guessed X_i encodes a strong generator set for $G^{(i)}$. We now show that N produces $gap = (n!)^{2n^2+1}/|G^{(i)}|$. By part 1 of Lemma 5.5.2, α computed in *line* 4 is $|G^{(i)}|$. Also, from part 2 of Lemma 5.5.2, β computed in *line* 5 is K_i, which is the number of strong generator sets for $G^{(i)}$. So, in *line* 6 ρ contributes $gap = (n!)^{n^2+1}/(|G^{(i)}|K_i)$. (Notice that $(n!)^{n^2+1}$ is chosen to be divisible by the denominator.) In *line* 9, each such path again branches into $(n!)^{n^2}/\gamma$ paths. So at this point the path ρ has contributed $(n!)^{2n^2+1}/(|G^{(i)}|K_i\gamma)$ paths. In *line* 10, since X_i generates $G^{(i)}$ it follows that $acc_{N'}(\langle 0^n, j, k\rangle) = 0$, for each $\langle j, k\rangle \in S_i$. Hence the *gap* produced in *line* 10 is $\prod_{\langle j,k\rangle \in S_i}(\prod_{n \geq l > j} |T_l|)$ which, by rearranging terms, is easily seen to be $\prod_{n \geq j > i}(\prod_{n \geq k > j} |T_k|)^{(n-j-|T_j|+1)} = \gamma$. Therefore, at end of *line* 10 ρ contributes a $gap = \gamma((n!)^{2n^2+1}/|G^{(i)}|K_i\gamma)$. Since there are K_i different paths ρ, one for each strong generator set for $G^{(i)}$, the total contribution to the *gap* from the K_i generator sets adds up to $(n!)^{2n^2+1}/|G^{(i)}|$.

Now, let ρ be a computation path on which the guessed set X_i generates a proper subgroup of $G^{(i)}$. We show that this path contributes a $gap = 0$. Notice that in order to prove that ρ contributes $gap = 0$, it suffices to show that the *gap* produced in *line* 8 is zero. Observe that corresponding to X_i there is a least $j < i$, say j_0, such that T_{j_0} is not a transversal of $G^{(j_0)}$ in $G^{(j_0-1)}$. Therefore, there exists $k \leq n$ and $g \in G$ such that $\langle j_0, k\rangle \in S_i$, $g \in G^{(j_0-1)}$, and $g(j_0) = k$. On the other hand, since j_0 is the largest such index $\bigcup_{n\geq l>j_0} T_l$ is a strong generating set for $G^{(j_0)}$. Putting the preceding two statements together, we can see that machine N' on input $\langle 0^n, j_0, k\rangle$ has $|G^{(j_0)}|$ accepting paths. Furthermore, since $\prod_{n\geq l>j_0} |T_l| = |G^{(j_0)}| = acc_{N'}(\langle 0^n, j_0, k\rangle)$, it holds that the product $\prod_{\langle j,k\rangle\in S_i}(acc_{N'}(\langle 0^n, j, k\rangle) - (\prod_{n\geq l>j} |T_l|)) = 0$. Hence the gap produced on path ρ is zero. Finally, notice that the *gap* prescribed in *line* 10 is indeed producible from the closure properties of GapP. This proves Claim 5.5.2.1.

Claim 5.5.2.2 *There is a polynomial time deterministic learning algorithm* LEARNSYM *with* $L(\text{SYM})$ *as assistant which, given a target concept* G *from the class* SYM *outputs a strong generator set for* G.

Proof The objective of the learner LEARNSYM is to compute a strong generator set as a union $\bigcup_{i=1}^{n} T_i$, where T_i is a transversal of $G^{(i)}$ in $G^{(i-1)}$, for $1 \leq i \leq n$. To this end, the learner first computes, with the help of $L(\text{SYM})$, the set of pairs $\{\langle i, j\rangle \mid \exists \ g \in G$ such that $g \in G^{(i-1)}$ and $g(i) = j\}$. For each such pair $\langle i, j\rangle$ it invokes a subroutine CONSTRUCT on input $\langle 0^n, i, j\rangle$ in order to compute the lexicographically first $g \in G^{(i-1)}$ such that $g(i) = j$. This is done by a prefix search with the help of $L(\text{SYM})$. It is easy to see that the set of all these computed elements g is a strong generator set. We give a formal description of this simple learning algorithm below.

```
LEARNSYM($0^n$)
1  for pairs $\langle i,j\rangle; 1 \le i \le j \le n$ do
2    if $\langle 0^n, 1, i-1, j, G\rangle \in L(\mathrm{SYM})$
3      then  CONSTRUCT($0^n, i, j$);
4    end-if
5  end-for
```

```
CONSTRUCT($0^n, i, j$)
1  $k \leftarrow 1$
2  repeat
3          Find first $l_k; i < l_k \le n$ such that
4          $\langle 0^n, k+1, i-1, j, x_1, \ldots, x_{k-1}, l_k, G\rangle \in$L(SYM);
5          $x_k \leftarrow l_k$;
6          $k \leftarrow k+1$;
7     until $k = n-i$
8  return $(1, \ldots, i-1, j, x_1, \ldots, x_{n-i})$.
```

It is easy to see that LEARNSYM has time complexity $O(n^4)$. ■

Learning LS(p) with SPP-assistant

Now we show the following upper bound on learning the class LS(p).

Theorem 5.5.3 *For any prime p, the representation class* LS(p) *is* FP*-learnable with an* SPP*-assistant.*

For the proof we need to give some notations and definitions from linear algebra. We give only definitions that are essential for our proof. For more basic definitions and results please refer to any standard linear algebra book (see for example [HK71]).

Let U be a vector space over a field F. We denote the zero vector of U by $\mathbf{0}$. U is called nontrivial if $U \neq \{\mathbf{0}\}$. Let $V < U$ denote that V is a subspace of U. For $X \subseteq U$ let $\langle X\rangle$ denote the subspace of U spanned by X. The direct sum of U and V is denoted by $U \oplus V$, and the n-fold direct sum of F is denoted by F^n. For the proof of the Theorem 5.5.3, we need to recall some results form linear algebra. The following result easily follow from basic definitions.

Proposition 5.5.4 *Let $W < V < U$ be finite dimensional vector spaces over a field F. Let A be a basis for U and B a basis for W. Let A' be a maximal subset of A such that $A' \cup B$ is linearly independent. Then $A' \cup B$ is a basis for U. Moreover, $V = W \oplus (\langle A' \rangle \cap V)$.*

Let F be a field, and U be a nontrivial subspace of F^n. Let $U^{(i)}$ denote the subspace $\{u \in U \mid \text{for } 1 \leq j \leq i: \ u[j] = 0\}$ of U, for $1 \leq i \leq n$, and let $U^{(0)}$ denote U. The *0-index* of U is the maximum k such that $U^{(k)}$ is nontrivial. We have the following proposition. See appendix for a proof.

Proposition 5.5.5 *Let F be a field and U be a nontrivial subspace of F^n. If k is the 0-index of U then $U^{(k)}$ is a 1-dimensional subspace.*

Proof. (*Of Theorem 5.5.3*) We first define an assistant $L(\mathrm{LS}(p))$ and show that $\mathrm{LS}(p)$ is FP-learnable with assistant $L(\mathrm{LS}(p))$. Then we prove that $L(\mathrm{LS}(p)) \in \mathrm{SPP}(\mathrm{LS}(p))$.

On input 0^n, let $V < F_p^n$ be the target subspace. Let vectors F_p^n be uniformly encoded in strings of length $t(n)$ for polynomial t (recall the definition of $\mathrm{LS}(p)$). Define $L(\mathrm{LS}(p))$ as: $L(\mathrm{LS}(p)) \triangleq \{\langle 0^n, A, i, y, V\rangle | i \text{ is the 0-index of } (\langle A \rangle \cap V); \exists z \in \Sigma^{t(n)} \text{ with } yz \in \langle A \rangle \cap V; yz \neq \mathbf{0}\}$.

We first give an intuitive idea of how the learner works. The aim of the learner is to construct a basis for the target subspace V. Suppose we have constructed an independent set of vectors $B \subseteq V$. We use $L(\mathrm{LS}(p))$ to prefix search for a vector $v \in V - \langle B \rangle$ as follows: Let A be a fixed basis for F_p^n (for example, the standard basis of unit vectors). Construct $A' \subseteq A$ such that $A' \cup B$ is a basis for F_p^n. (This can be done with some rank computations, which can be carried out in polynomial time.) Next, compute the 0-index of the space $(\langle A' \rangle \cap V)$ with $L(\mathrm{LS}(p))$ as oracle, by making queries of the form $\langle 0^n, A', i, \epsilon, V\rangle$ for $1 \leq i \leq n$. We now construct v using prefix search with queries to $L(\mathrm{LS}(p))$. Notice that if $\langle B \rangle \neq V$

then by Proposition 5.5.4 $\langle A' \rangle \cap V$ is nontrivial and every nonzero element in it is independent of vectors in $\langle B \rangle$. Therefore, including v in B increases the dimension of $\langle B \rangle$. This process of including new vectors in $\langle B \rangle$ will therefore terminate within n steps. We now formally describe the learning algorithm.

```
LEARNLINEAR($0^n$)
 1  $B \leftarrow \{0\}$;
 2  $A \leftarrow \{e_1, \ldots, e_n\}$ (* Standard basis for $F_p^n$ *) ;
 3  Construct $A' \subseteq A$ such that $B \cup A'$ is maximally independent ;
 4  $i \leftarrow 1$;
 5  while $\langle 0^n, A', i, \lambda, V \rangle \notin L(\mathrm{LS}(p))$
 6  do $i \leftarrow i + 1$;
 7  end-while
 8  if $i = n + 1$
 9     then output B and stop .
10  end-if
11  $y \leftarrow \lambda$;
12  for $j \leftarrow 1$ to $t(n)$
13  do if $\langle 0^n, A', i, y1, r \rangle \in L(\mathrm{LS}(p))$
14          then $y \leftarrow y1$;
15          else $y \leftarrow y0$;
16      end-if
17  end-for
18  $B \leftarrow B \cup \{y\}$;
19   goto 3;
```

We now show that $L(\mathrm{LS}(p)) \in \mathrm{SPP}(\mathrm{LS}(p))$. We construct a GapP machine M which on input $\langle 0^n, A', i, y \rangle$, uses V as oracle, and has the property that $M(0^n, A', i, y)$ produces $gap = 1$ if $\langle 0^n, A', i, y, V \rangle \in L(\mathrm{LS}(p))$ and produces $gap = 0$ otherwise.

To this end we define another NP machine N (which uses V as oracle). On input $\langle 0^n, A', i, y, k \rangle$ the machine N guesses $p - 1$ distinct nonzero vectors $v_1, \ldots, v_{p-1} \in \langle A' \rangle^{(k)}$ in lexicographically increasing order, such that y is a prefix of v_1 and N accepts along this path iff each $v_i \in V$. It follows from the definition of 0-index that if k is more than the 0-index of $A' \cap V$ then N has no accepting paths. Furthermore, if k is the 0-index then, from Proposition 5.5.5, N has a unique accepting path.

Let $acc_N(\langle 0^n, A', i, y, k\rangle)$ be the number of accepting paths of N on input $\langle 0^n, A', i, y, k\rangle$. We now design the desired machine M which uses N.

Description of $M(0^n, A, i, y)$

```
1   Produce a gap of $gap_N(\langle 0^n, A', i, y, i\rangle) \cdot \prod_{k=i+1}^{n}(1 - acc_N(\langle 0^n, A', i, y, k\rangle))$.
```

Notice that on input $\langle 0^n, A', i, y\rangle$, if i is not the 0-index then M has a $gap = 0$. Furthermore, if i is the 0-index and y is the prefix of the guessed vector v_1, then M has $gap = 1$ otherwise M has $gap = 0$. Hence $L(\mathrm{LS}(p))$ is in $\mathrm{SPP}(\mathrm{LS}(p))$. M can produce the *gap* prescribed in *line* 1 above by appropriate simulations of machine N. ■

Hardness of learning 3-CNF

Now we show a complexity-theoretic evidence that it is unlikely that 3-CNFs are learnable with an SPP-assistant or an LWPP-assistant. We prove the following theorem.

Theorem 5.5.6 *If* 3-CNF *is* FP*-learnable with an* SPP*-assistant, then* NP $\subseteq$ SPP. *If* 3-CNF *is* FP*-learnable with an* LWPP*-assistant, then* NP $\subseteq$ LWPP.

Proof. We prove that if 3-CNF is FP-learnable with an SPP-assistant, then NP $\subseteq$ SPP. The proof of the second statement is identical and is omitted. For proving this, we show that if 3-CNF has a learning algorithm with an SPP-assistant, then the co-NP-complete language TAUT= $\{\langle 0^n, 0^l, f\rangle | f$ encodes 3-CNF on n variables of size $\leq l$ and f is a tautology$\}$ is in $\mathrm{P}^{\mathrm{SPP}}$ which is SPP [FFK94]. Since SPP is closed under complement, the result follows.

Let α be an FP-learner, which on input $\langle 0^n, 0^l\rangle$ learns an n-variable 3-CNF of size less than or equal to l with an SPP-assistant $L($ 3-CNF$) = \{\langle n, l, y, c\rangle | y \in$

$\Sigma^*; Q(n, l, y, c)\}$ accepted by a machine N, where Q is the predicate defining the SPP-assistant. Define a language $A = \{\langle 0^n, 0^l, y, f\rangle | y \in \Sigma^*; f$ encodes a 3-CNF formula and $Q'(n, l, y, f)\}$ where the predicate Q' is such that if $Q'(n, l, y, f)$ is true then N on input $\langle 0^n, 0^l, y\rangle$ with f as target formula has $gap = 1$ and if $Q'(n, l, y, f)$ is false then N will have a $gap = 0$. It is easy to see that $A \in$ SPP. Consider the deterministic machine M, for accepting TAUT, which on input $\langle 0^n, 0^l, f\rangle$ does the following. M simulates α on input n two times, sequentially. In the first simulation whenever α makes a query $\langle n, l, y, c\rangle$ to $L($ 3-CNF$)$, M makes a query $\langle 0^n, 0^l, y, f\rangle$ to A. In the next simulation whenever α makes a query $\langle n, l, y, c\rangle$ to $L($ 3-CNF$)$, M makes a query $\langle 0^n, 0^l, y, T\rangle$ to A (T is the trivial formula *true*). M accepts $\langle 0^n, 0^l, f\rangle$ if the output of M in both the cases are the same. Now we crucially observe that since α is a deterministic machine it outputs the same string in both simulations iff f is a tautology. Theorem follows. ∎

5.6 Summary

In this chapter we investigated the complexity of exact learning some algebraic representation classes. Our major interest was in learning the class of permutation groups and linear spaces over finite fields. First we observed that Angluin's model is insufficient for a fine classification of the complexity of these representation classes. Hence, we proposed a refinement of Angluin's model called the teaching assistant model of exact learning. To classify the complexity of learning various representation classes in the teaching assistant model, we introduced the notion of teaching assistant classes and defined some teaching assistant classes analogous to well-known complexity classes. Among these, the teaching assistant classes analogous to the complexity classes SPP and LWPP were of importance. As our main results, we showed that permutation groups are polynomial-time learnable with LWPP-assistants and linear spaces are polynomial-time learnable with SPP-assistants. We also showed that it is unlikely that the representation class 3-CNF is polynomial time learnable with any of these assistant classes. These results formalize the intuition the complexity of learning 3-CNF is quite different from that of learning either permutation groups or linear spaces. Notice that in Angluin's model, it is not possible to bring out this difference.

In the next chapter we continue our investigations on the complexity of exact learning. We define more assistant classes and prove some absolute separations among these classes. We use subclasses of the representation class SYM for showing these separations.

Chapter 6

Separating Teaching Assistant Classes

In the previous chapter, we introduced various assistant classes and used these classes to give finer upper bounds on learning some algebraic representation classes. As mentioned earlier, the different assistant classes that we have defined are in exact analogy to standard complexity classes. Furthermore, the previous chapter indicates that some of these teaching assistant classes have some correspondence with exact learnability in Angluin's model. Motivated by the possible finer classification of representation classes, and again in analogy with complexity classes, in this chapter we define and study some more teaching assistant classes. Particularly, we define assistant classes corresponding to the classes UP ∩ co-UP, UP and NP ∩ co-NP. Using some subclasses of the representation class SYM, we show some separations that are possible among various teaching assistant classes that we consider.

Before going into the proofs, we summarize the containments and separations that we show among various teaching assistant classes in the following figure. For the completeness sake, we also compare various types of queries in Angluin's model. In the last chapter we have shown that the membership queries in Angluin's model are completely characterized by P-assistants in the teaching assistant model. It is shown in [Ang88] that the class k-CNF is FP-learnable with only equivalence queries in Angluin's model. It is also shown there that the representation class monotone-

DNF (class of monotone boolean functions represented by monotone DNF formulas) is FP-learnable with equivalence and membership queries. In [AHK93], it is shown that the class of monotone Read-once formulas (class of monotone boolean functions representable by monotone formulas in which all the literals appear exactly once) are FP-learnable with only membership queries.

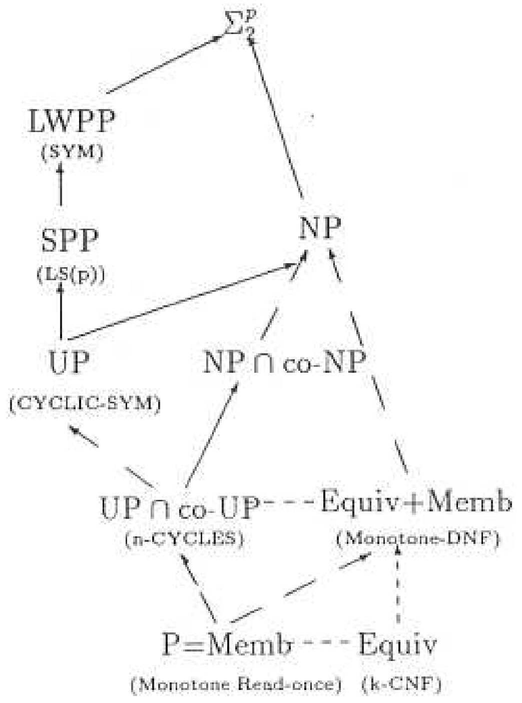

Figure 1. Inclusions and separations of different teaching assistant classes[1].

In the next section we define three more teaching assistant classes analogous to complexity classes NP ∩ co-NP, UP ∩ co-UP and UP. We also prove machine characterizations of some of these classes. These characterizations are used to show some more lower bounds in the teaching assistant model.

[1]In the figure a bold line with arrow indicates inclusion, a dashed line with arrow indicates proper inclusion, and a dashed line without arrow indicates separation in both directions. For each teaching assistant class $\mathcal{C}$ in the picture, a representation class that is FP-learnable with $\mathcal{C}$-assistant is given in brackets under the assistant class. Formal definitions of these classes are given later.

6.1 More Assistant Classes

Let us first define the assistant classes of interest to us in this chapter.

Definition 6.1.1 Let $\mathcal{P}$ be a representation class and $L(\mathcal{P})$ be a teaching assistant.

1. $L(\mathcal{P})$ is said to be in the class NP($\mathcal{P}$)$\cap$co-NP($\mathcal{P}$) if there exists a polynomial-time non-deterministic oracle Turing machine M which for any $\mathcal{P}$-valid tuple $\langle n, l, x, c\rangle$, on any path outputs 'accept', 'reject' or '?' such that: on input $\langle 0^n, 0^l, x\rangle$ uses c as oracle and outputs 'accept' on at least one of its paths and does not output 'reject' on any path, if $\langle n, l, x, c\rangle \in L(\mathcal{P})$ and outputs 'reject' on at least one of its paths and does not output 'accept' on any path, if $\langle n, l, x, c\rangle \notin L(\mathcal{P})$.

2. $L(\mathcal{P})$ is said to be in the class UP($\mathcal{P}$) if there exists a polynomial-time non-deterministic oracle Turing machine M which for any $\mathcal{P}$-valid tuple $\langle n, l, x, c\rangle$, on input $\langle 0^n, 0^l, x\rangle$ uses c as oracle and accepts $\langle 0^n, 0^l, x\rangle$ if and only if $\langle n, l, x, c\rangle \in L(\mathcal{P})$ with the promise that M has at most one accepting path on any input.

3. $L(\mathcal{P})$ is said to be in the class UP($\mathcal{P}$)$\cap$co-UP($\mathcal{P}$) if there exists a polynomial-time nondeterministic oracle Turing machine M which for any $\mathcal{P}$-valid tuple $\langle n, l, x, c\rangle$, on any path outputs 'accept', 'reject' or '?' such that: on input $\langle 0^n, 0^l, x\rangle$ uses c as oracle and outputs 'accept' on a unique path and does not output 'reject' on any path if $\langle n, l, x, c\rangle \in L(\mathcal{P})$, and outputs 'reject' on a unique path and does not output 'accept' on any path, if $\langle n, l, x, c\rangle \notin L(\mathcal{P})$.

In all the above definitions, the behavior of machines accepting the teaching assistants is not specified on inputs which are not $\mathcal{P}$-valid.

Remark. Since we are interested in the query complexity (in an information-theoretic sense) of teaching assistants, it is some times useful to define relativized teaching assistant classes by allowing Turing machines accepting teaching assistants to have oracle access to a language, say $A \subseteq \Sigma^*$. Notice that this language aids the machine only in computation (we term such an oracle a *computational oracle*) and not in retrieving information about the target concept. Let $\mathcal{P}$ be a representation class. Then the assistant class $\mathcal{C}(\mathcal{P})$ relative to a computational oracle $A \subseteq \Sigma^*$ is denoted by $\mathcal{C}(\mathcal{P})^A$.

The following refinement of Proposition 5.4.3 from the previous chapter is direct from the definitions.

Proposition 6.1.2 *For any representation class* $\mathcal{P}$

- $\mathrm{P}(\mathcal{P}) \subseteq \mathrm{UP}(\mathcal{P}) \cap \text{co-UP}(\mathcal{P}) \subseteq \mathrm{UP}(\mathcal{P}) \subseteq \mathrm{SPP}(\mathcal{P}) \subseteq \mathrm{LWPP}(\mathcal{P})$.
- $\mathrm{P}(\mathcal{P}) \subseteq \mathrm{UP}(\mathcal{P}) \cap \text{co-UP}(\mathcal{P}) \subseteq \mathrm{NP}(\mathcal{P}) \cap \text{co-NP}(\mathcal{P}) \subseteq \mathrm{NP}(\mathcal{P}) \subseteq \Sigma_2^p(\mathcal{P})$

6.1.1 Learning with NP $\cap$ co-NP and UP $\cap$ co-UP assistants

Machine characterizations of learners which use assistants from NP$\cap$co-NP and UP$\cap$ co-UP assistant classes will be handy in many of our proofs, especially for proving lower bounds. We next show these characterizations. For this purpose, we extend the deterministic learners in Angluin's model to nondeterministic learners. We consider two types of nondeterministic learners; NPSV learners and UPSV learners. Let $\mathcal{P}$ be a representation class. A NPSV (UPSV) learner is a nondeterministic oracle Turing machine transducer which on input $\langle 0^n, 0^l \rangle$ and $c \in \mathcal{P}$ as target concept, has to output a representation $r \in R$ such that $\mu_n(r) = c$ (if $size(c) \leq l$) for the concept c, on at least one (respectively, *exactly* one) of its paths. Moreover, the learner should output the *same* representation on all the paths on which it is outputting. Also, the running time of the transducer must be bounded by a polynomial. As in the case of teaching assistant classes, we also allow these learners to have access to a computational oracle whenever necessary.

In the next theorem, we show machine characterizations of the above-defined assistant classes. The proof of the theorem is an adaptation of the proof that a function $f \in \mathrm{FP}^{\mathrm{NP}\cap\text{co-NP}}$ if and only if f is NPSV computable (implicitly in [BLS84]).

Theorem 6.1.3 *For any representation class $\mathcal{P}$, and any language $A \subseteq \Sigma^*$*

1. *$\mathcal{P}$ is NPSV-learnable with membership queries with the aid of a computational oracle A if and only if $\mathcal{P}$ is* FP*-learnable with an* $(\mathrm{NP} \cap \text{co-NP})^A$*-assistant.*
2. *$\mathcal{P}$ is* UPSV*-learnable with membership queries with the aid of a computational oracle A if and only if $\mathcal{P}$ is* FP*-learnable with an* $(\mathrm{UP} \cap \text{co-UP})^A$*-assistant.*

Proof. We prove part 1 of the above theorem. The second part can be proved very similarly.

Let $\mathcal{P} = \langle R, \{\mu_n\}_{n\geq 1}, t\rangle$ be any representation class and A be a computational oracle. Let β be an FP-learner for $\mathcal{P}$ using a teaching assistant $L(\mathcal{P})$ in $(\mathrm{NP}(\mathcal{P}) \cap \text{co-NP}(\mathcal{P}))^A$. Let M^A be a nondeterministic oracle Turing machine witnessing $L(\mathcal{P})$ in $(\mathrm{NP}(\mathcal{P}) \cap \text{co-NP}(\mathcal{P}))^A$. Let $c \in \mathcal{P}_n$ be the target concept and l be such that $size(c) \leq l$. Consider the following learner β' which on input $\langle 0^n, 0^l\rangle$ simulates the learner β on input $\langle 0^n, 0^l\rangle$. Whenever β makes a query $\langle n, l, x, c\rangle$ to $L\mathcal{P}$, β' start simulating M on input $\langle 0^n, 0^l, x\rangle$ by guessing a path ρ of M. During the simulation of path ρ of M by β, whenever M makes a query y to A, β' also makes query y to A. Whenever M makes a query z to the target concept c, β' makes a membership query z to the teacher and treats the answer as the answer to M's query z. If the path ρ outputs '?', then β' abandons that path. If the path ρ outputs 'accept' then β' treats this as answer 'Yes' to the query $\langle n, l, x, c\rangle$ of β and proceeds. If the path ρ outputs 'reject' then β' treats this as answer 'No' to the query $\langle n, l, x, c\rangle$ of β and proceeds. Now, on any single simulation of M by β', the paths which are not abandoned by β' has the same answer, it clear that, in the end, β' running in time bounded by a polynomial in $n + l$, outputs the same string as output by β.

To show the containment in the other direction, let $\mathcal{P} = \langle R, \{\mu_n\}_{n\geq 1}, t\rangle$ be any representation class that is NPSV-learnable using membership queries. Let α be an NPSV-learner for $\mathcal{P}$ which uses A as computational oracle. Let q be the polynomial bounding the running time of α. Without loss of generality, we can assume that for input $\langle 0^n, 0^l\rangle$, all the paths of α runs for exactly $q(n+l)$ time steps. Consider the following teaching assistant $L(\mathcal{P}) = \{\langle n, l, (i, b), c\rangle | \alpha$ on input $\langle 0^n, 0^l\rangle$, with c as target concept, and A as computational oracle, runs for $q(n+l)$ steps and outputs a string y whose i^{th} bit is $b\}$. Now we show that $L(\mathcal{P}) \in (\text{NP}(\mathcal{P}) \cap \text{co-NP}(\mathcal{P}))^A$. Consider the following oracle Turing machine M^A which on input $\langle 0^n, 0^l, (i, b)\rangle$ simulates the learner α for $q(n+l)$ steps using c as oracle. On any path ρ of α, if α makes a query y to A, M^A also makes query y to A. Whenever α makes a membership query z to the teacher, M^A makes a membership query z to the target concept c and treats the answer as the answer to α's query z. At the end of $q(n+l)$ steps of computation, on the paths where α does not output anything, M^A outputs '?' and on the paths where α outputs a string y M^A outputs 'accept' if i^{th} bit of y is b and outputs 'reject' if i^{th} bit of y is not b or $|y| < i$. It is easy to verify that M^A witnesses $L(\mathcal{P}) \in (\text{NP}(\mathcal{P}) \cap \text{co-NP}(\mathcal{P}))^A$. Now we give an FP-learner α' for $\mathcal{P}$ using teaching assistant $L(\mathcal{P})$. Let $c \in \mathcal{P}_n$ be the target concept and l such that $size(c) \leq l$.

```
NP ∩ co-NP-LEARNER $\alpha'(0^n, 0^l)$
1  $i \leftarrow 1$;
2  while $\langle n, l, (i, b), c\rangle$ or $\langle n, l, (i, \bar{b}), c\rangle \in L(\mathcal{P})$
3  do if $\langle n, l, (i, b), c\rangle \in L(\mathcal{P})$
4          then $y \leftarrow yb$;
5          else  $y \leftarrow y\bar{b}$;
6      end-if
7      $i \leftarrow i + 1$;
8  end-while
9  Output $y$.
```

It is clear that α' constructs the representation of c which is output by α. Also, the running time of α' is bounded by $O(q(n+l))$. ■

What is the difference between NP$\cap$co-NP-assistants and UP$\cap$co-UP-assistants? We show in the next theorem that information-theoretically they have the same power. More precisely, we show that UP $\cap$ co-UP-assistants with computational oracles in NP are as powerful as NP $\cap$ co-NP-assistants. Thus, separating these assistant classes in the polynomial-time learnability sense is as hard as separating P and NP.

Computational difficulty of separating UP$\cap$co-UP and NP$\cap$co-NP assistants

Here we prove the following theorem.

Theorem 6.1.4 *Let $\mathcal{P}$ be any representation class. If $\mathcal{P}$ is* FP*-learnable with an* (NP $\cap$ co-NP)*-assistant, then there is language $B \in$* NP *such that $\mathcal{P}$ is* FP*-learnable with a* $(\text{UP} \cap \text{co-UP})^B$*-assistant.*

Proof. We use the machine characterization of FP-learnability using NP $\cap$ co-NP-assistants and UP $\cap$ co-UP-assistants proved in Theorem 6.1.3. Let $\mathcal{P} = \langle R, \{\mu_n\}_{n\geq 1}, \ell\rangle$ be any representation class which is FP-learnable using NP$\cap$co-NP-assistant. Then by Theorem 6.1.3, there is an NPSV-learner α, which learns $\mathcal{P}$ using only membership queries. Consider the language (not to be confused with a teaching assistant) $B = \{\langle 0^n, 0^l, x, y\rangle | \exists$ a path ρ of α , which is lexicographically lesser than x, and α on input $\langle 0^n, 0^l\rangle$, $\mu_n(y)$ as target concept, outputs a string on $\rho\}$. We first show that B is in NP. Consider a nondeterministic oracle machine N which on input $\langle 0^n, 0^l, x, y\rangle$ first guesses a path ρ, which is lexicographically smaller than x, of α on input $\langle 0^n, 0^l\rangle$ and $\mu_n(y)$ as target concept and simulates α on that path. Since the representation class is honest, checking membership in $\mu_n(y)$ can be done in polynomial-time. So whenever α makes membership queries z to the target concept $\mu_n(y)$, N computes the answer to this query using the representation y (part of input) of $\mu_n(y)$. Hence $B \in$ NP.

To complete the proof, consider a UPSV-learner α', which on input $\langle 0^n, 0^l \rangle$ and target concept c, simulates the NPSV-learner α on input $\langle 0^n, 0^l \rangle$ and target concept c. If α on a computation path x outputs a string y as a representation of the target concept, α' makes a query $\langle 0^n, 0^l, x, y \rangle$ to B. If the answer is 'No', α' outputs y. If the answer to the query is 'Yes', α' halts without output. It is clear that α' outputs only on the unique lexicographically first path on which α outputs. Hence α' is a UPSV-learner with membership queries and B as oracle for $\mathcal{P}$. From Theorem 6.1.3, it follows that $\mathcal{P}$ is FP-learnable using a $(\mathrm{UP} \cap \text{co-UP})^B$-assistant with the help of an NP oracle. ■

The next corollary is immediate.

Corollary 6.1.5 *If there exists a representation class $\mathcal{P}$ that is* FP*-learnable with an* NP $\cap$ co-NP*-assistant, and not* FP*-learnable with a* UP $\cap$ co-UP*-assistant, then* $\mathrm{P} \neq \mathrm{NP}$.

6.2 Learning Subclasses of SYM

In this section we show more results on the learnability of algebraic representation classes n-CYCLES and CYCLIC-SYM. These are subclasses of SYM $= \langle R, \{\mu_n\}_{n \geq 1}, t \rangle$. By restricting elements in R to satisfy some pre-specified property, we define these subclass of SYM as follows. $\mathcal{P}' = \langle R', \{\mu'_n\}_{n \geq 1}, t' \rangle$ be a subclass of SYM. $\mathcal{P}'$ is the class CYCLIC-SYM if R' consists of those sets of permutations which generate a *cyclic* group. $\mathcal{P}'$ is the class n-CYCLES if R' consists of a single n-*cycle* (please recall the definition of an n-*cycle* from the previous chapter). It is clear that n-CYCLES$\subseteq$CYCLIC-SYM$\subseteq$SYM. Since SYM is honest, it follows that all these classes are honest.

Next we show upper and lower bounds on learning these subclasses in teaching assistant model. These results are relatively easier to prove. The purpose of these results is to illustrate the separations that are possible among various teaching assistant classes that we have defined.

First we show upper and lower bounds on learning n-CYCLES.

Learning n-CYCLES

We show the following theorem.

Theorem 6.2.1 *The class n-*CYCLES *is not* FP*-learnable with P-assistants. It is* FP*-learnable with a* UP $\cap$ co-UP*-assistant.*

Proof. Notice that FP-learning with membership query in Angluin's model is same as FP-learning with a P-assistant. Now, the proof of the first part of the theorem is already given as the proof of the Theorem 5.3.2 given in the previous chapter, namely the class SYM is not FP-learnable with membership queries.

For the proof of the second part, we need the following elementary fact about the number of generators of a group generated by an n-cycle. To be more precise, let $\phi(n) = |\{i | 1 \le i \le n; gcd(i,n) = 1\}|$, for $n \in \mathbf{N}$. For each n-cyclic group $G < S_n$, there are precisely $\phi(n)$ n-cycles σ such that $G = \langle \sigma \rangle$.

Now, we describe a UPSV-learner that learns n-CYCLES with membership queries. On input 0^n, the UPSV-learner first computes $\phi(n)$, then guesses a lexicographically ordered set of $\phi(n)$ n-cycles, and finally makes membership queries corresponding to each guessed n-cycle. The learner outputs the first of the guessed n-cycles, if answers to all the queries are 'Yes'. From Theorem 6.1.3, it follows that n-CYCLES is FP-learnable with a UP $\cap$ co-UP-assistant ■

We now turn to the learnability of the class CYCLIC-SYM.

Learning CYCLIC-SYM

We show that while CYCLIC-SYM is not learnable with NP $\cap$ co-NP-assistants, it is FP-learnable with a UP-assistant. We first prove the lower bound.

Theorem 6.2.2 *The class* CYCLIC-SYM *is not* FP-*learnable with* NP $\cap$ co-NP-*assistants.*

Proof. To show this lower bound, notice that it suffices to show that CYCLIC-SYM is not NPSV-learnable with membership queries. Suppose α is an NPSV-learner for CYCLIC-SYM. Let s be a polynomial such that $s(n)$ upper bounds the number of membership queries asked by $\alpha(0^n)$. Choose a prime p such that $(p-2)! > s(p)$. Consider the computation of $\alpha(0^p)$, with target group $\langle e \rangle$. Let ρ be a computation path on which α outputs a representation for $\langle e \rangle$. Furthermore, let $(q_1, q_2 \ldots, q_{s(p)})$ be the set of elements of S_p queried by α on the path ρ. Since the target group is $\langle e \rangle$ the answers to all queries except e is 'No'. Since $(p-2)! > s(p)$, Proposition 5.3.1 implies that there is a p-cycle g such that $\{q_i \mid 1 \leq i \leq s(p)\} - \{e\} \cap \langle g \rangle = \emptyset$. Now consider the same computation path ρ for $\alpha(0^p)$ for target group $\langle g \rangle$. Notice that the answers to queries $(q_1, q_2 \ldots, q_{s(p)}, e)$ is the same whether the target group is $\langle e \rangle$ or $\langle g \rangle$. Therefore, α ends up outputting a representation for $\langle e \rangle$ on path ρ, even for the target group $\langle g \rangle$. This contradicts the definition of NPSV-learnability. ∎

Now we give an upper bound on learning CYCLIC-SYM. For the proof of this theorem we require the following simple group-theoretic results. See the appendix for a proof.

Proposition 6.2.3 *Let G be a cyclic subgroup of S_n.*

1. *If G is of order p^k, for prime p, then $p^k \leq n$.*
2. *Let $|G|$ has the prime factorization $p_1^{e_1} \ldots p_l^{e_l}$. Then the number of elements in G of order $p_i^{k_i}$ is $p_i^{k_i} - p_i^{k_i - 1}$, for each i and $k_i \leq e_i$.*

Theorem 6.2.4 CYCLIC-SYM *is* FP-*learnable with a* UP-*assistant.*

Proof. For CYCLIC-SYM we define a teaching assistant as follows:

$L(\text{CYCLIC} - \text{SYM}) = \{\langle 0^n, p, k, x, G\rangle | G < S_n, p^k \leq n, p$ is a prime, $\exists y \in \Sigma^+$ xy encodes the lex. first $g \in G : o(g) = p^k\}$.

Now we describe a machine ACCEPT-L(CYCLIC-SYM) that witnesses L(CYCLIC-SYM) in UP(CYCLIC − SYM).

```
ACCEPT-$L$(CYCLIC-SYM)($0^n, p, k, x, G$)
 1  $K \leftarrow p^k - p^{k-1}$;
 2  if $p^k > n$ or $p$ not a prime
 3    then reject
 4  end-if
 5  Guess $K$ distinct strings $x_1, \ldots, x_K$ in lexicographical order
 6  for $i = 1$ to $K$
 7  do if $x_i$ does not encode a permutation of $S_n$
 8         then reject
 9         else  Compute the order $o_i$ of the element encoded by $x_i$
10               if $o_i \neq p^k$
11                 then reject
12               end-if
13       end-if
14  end-for
15  if $\exists y$ such that $x_1 = xy$
16    then accept
17    else  reject
18  end-if
```

It is easy to see that the machine accepts the teaching assistant L(CYCLIC-SYM). Now we will see that the above non-deterministic machine has the required time bound and unambiguous behavior. Let G be the target cyclic group. *line* 2 can be obviously done in time polynomial in n. Since $p^k \leq n$ the nondeterministic step in *line*-5 can also be done in time polynomial in n. Computing the order of $g \in S_n$, in *line*-9 can be done in time polynomial in n by writing the cycle decomposition of g and computing the *lcm* of the cycle lengths in the decomposition. Now, from

Proposition 6.2.3, if there is an element of order p^k, then there are $p^k - p^{k-1}$ elements of order p^k in G. Hence at most *one* non-deterministic guess can be an accepting path after completing the **for**-loop. It follows that L(CYCLIC-SYM)$\in$ UP(CYCLIC-SYM).

Now we describe the CYCLICLEARNER for CYCLIC-SYM, which uses the assistant L(CYCLIC-SYM). Let G be the target concept. We first give an intuitive idea of how CYCLICLEARNER it works. Let $p_1^{e_1} \ldots p_l^{e_l}$ be the prime factorization of $|G|$, where G is the target cyclic group. Let G_i denote the unique subgroup of G of order $p_i^{e_i}$, $1 \leq i \leq l$. If g_i is a generator of G_i, then it is easy to see that $\prod g_i$ generates G. CYCLICLEARNER exploits precisely this property: using the assistant L(CYCLIC-SYM) it computes a generator g_i for each G_i and then finally computes a generator for G by multiplying the g_i's. The formal description follows.

```
CYCLICLEARNER($0^n$)
σ ← e;
for primes $p \leq n$
do Compute the largest index $d$ such that $p^d \leq n$;
    while $\langle 0^n, p, d, \lambda, G\rangle \notin L$(CYCLIC-SYM)
    do $d \leftarrow d - 1$;
    end-while
    (* Using $L$(CYCLIC-SYM) prefix search for $g_p$ of order $p^d$ *)
    $z \leftarrow \lambda$;
    while $\langle 0^n, p, d, z0, G\rangle$ or $\langle 0^n, p, d, z1, G\rangle \in L$(CYCLIC-SYM)
    do if $\langle 0^n, p, d, z0, G\rangle \in L$(CYCLIC-SYM)
          then $z \leftarrow z0$;
          else  $z \leftarrow z1$;
    end-while
    $g_p \leftarrow z$;
    $\sigma \leftarrow \sigma g_p$;
end-for
output $\sigma$.
```

6.3 Summary

The main focus of this chapter was on separating some of the teaching assistant classes that we defined in this and the previous chapters of the thesis. We introduced subclasses of the representation class SYM for showing some of the separations. (The separations shown are depicted in Figure 1.) The main purpose of defining these subclasses was to illustrate the possibility of separations among the teaching assistant classes. However, it will be interesting to see whether there are other representation classes studied in the literature which witness these separations.

Chapter 7

Conclusion

In this thesis, we investigated the structural complexity of some computational group-theoretic problems. The main results shown in this thesis provide more insights into the complexity of these problems.

In Chapter 4 of the thesis, we investigated the complexity of three basic computational group-theoretic problems; Membership Testing, Order Verification and Isomorphism Testing. All these problems are computationally difficult; there is no polynomial time algorithm for any of these problems over black-box groups. But, it is also not known whether they are hard for NP. In Chapter 4, we showed that these problems over solvable black-box groups are in the counting class SPP. This upper bound of SPP implies that these problems will not provide additional power as oracle to counting classes like PP, $C_=P$ and Mod_kP. This is an indication in support of the belief that these problems are unlikely to be hard for NP. Another important aspect of this result is that these problems provide more examples of natural problems in SPP which are not known to be in P.

There is one major open question that arises from this investigation. Extended the above upper bound to problems defined over general black-box group. We believe that the problems we considered here are low for PP even over general black-box groups. The methods we used for solvable groups will not extend for the general case. We feel that new methods based on classification theorems about finite simple groups may be required for the proof of this. A more general question is to show membership of other natural problems in the complexity classes like SPP or LWPP.

In Chapters 5 and 6 of the thesis, our focus was on the complexity of learning some algebraic representation classes. Since Angluin's model of exact learning is not enough for a fine classification of their complexity, we proposed a new exact learning model called the teaching assistant model. For classifying the complexity of learning in this model, we introduced the notion of teaching assistant classes. Using these notions we were able to show that complexity of learning permutation groups and linear spaces over finite fields are quite different from that of learning 3-CNF. In Chapter 6 we considered separating teaching assistant classes using subclasses of the representation class of permutation groups.

We mention some open problems. Firstly, it will be interesting to see whether there are representation classes that can be learned with an LWPP-assistant but not with an SPP-assistant. Is it true that class SYM is polynomial-time learnable with an SPP-assistant? As a first step towards answering this, it will be interesting if one can show that the subclass of SYM consisting only of abelian groups as concepts, can be learned using an SPP-assistant.

The teaching assistant model for learning is as yet of only theoretical interest. As the next step, it will be interesting to develop more learning algorithms in this model for various other representation classes (not necessarily algebraic), and explore further the relationship between the teaching assistant model and other well-known models of learning.

Bibliography

[AHK93] D. Angluin, L. Hellerstein, and M. Karpinski. Learning read-once formulas with queries. *Journal of the Association for Computing Machinery*, 40(1):185–210, 1993.

[All86] E. Allender. The complexity of sparse sets in P. In *Proc. of the 1st IEEE Structure in Complexity Theory Conference*, pages 1–11, 1986. Lecture Notes in Computer Science # 223.

[Ang88] D. Angluin. Queries and concept learning. *Machine Learning*, 2:319–342, 1988.

[Ang90] D. Angluin. Negative results for equivalence queries. *Machine Learning*, 5:121–150, 1990.

[Ang92] D. Angluin. Computational learning theory: survey and selected bibliography. In *Proc. of the 24th ACM Symposium on Theory of Computing*, pages 351–369, 1992.

[AV96] V. Arvind and N. V. Vinodchandran. The complexity of exactly learning algebraic concepts. In *Proc. of the 7th International Workshop on Algorithmic Learning Theory*, pages 100–112. Springer-Verlag, 1996. Lecture Notes in Artificial Intelligence # 1160.

[AV97a] V. Arvind and N. V. Vinodchandran. Exact learning via teaching assistants. In *Proc. of the 8th International Workshop on Algorithmic*

Learning Theory, pages 291–306. Springer-Verlag, 1997. Lecture Notes in Artificial Intelligence # 1316.

[AV97b] V. Arvind and N. V. Vinodchandran. Solvable black-box group problems are low for PP. *Theoretical Computer Science*, 180:17–47, 1997.

[Bab85] L. Babai. Trading group theory for randomness. In *Proc. of the 17th ACM Symposium on Theory of Computing*, pages 421–429, 1985.

[Bab92] L. Babai. Bounded round interactive proofs in finite groups. *SIAM Journal of Discrete Mathematics*, 5:88–111, 1992.

[BB86] J. Balcázar and R. Book. Sets with small generalized Kolmogorov compelxity. *Acta Informatica*, 23:679–688, 1986.

[BCF⁺95] L. Babai, G. Cooperman, L. Finkelstein, E. Luks, and Á. Seress. Fast monte carlo algorithms for permutation groups. *Journal of Computer and System Sciences*, 50:296–308, 1995.

[BCG⁺96] N. Bshouty, R. Cleve, R. Gavaldà, S. Kannan, and C. Tamon. Oracles and queries that are sufficient for exact learning. *Journal of Computer and System Sciences*, 52:421–433, 1996.

[BDG88] J. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity – I & II.* Springer Verlag, Berlin Hiedelberg, 1988.

[BG92] R. Beigel and J. Gill. Counting classes: Thresholds, parity, mods and fewness. *Theoretical Computer Science*, 103:3–23, 1992.

[BHZ87] R. Boppana, J. Hastøad, and S. Zachos. Does co-NP have short interactive proofs? *Information Processing Letters*, 25:127–132, 1987.

[BLS84] R. Book, T. Long, and A. Selman. Quantitative relativization of complexity classes. *SIAM Journal on Computing*, 13:461–487, 1984.

[BLS87] L. Babai, E. Luks, and Á. Seress. Permutation groups in NC. In *Proc. of the 19th ACM Symposium on Theory of Computing*, pages 409–420, 1987.

[BRS95] R. Beigel, N. Reingold, and D. Spielman. PP is closed under intersection. *Journal of Computer and System Sciences*, 50(2):191–202, 1995.

[BS84] L. Babai and E. Szemerédi. On the complexity of matrix group problems I. In *Proc. of the 25th IEEE Symposium on Foundations of Computer Science*, pages 229–240, 1984.

[Bur55] W. Burnside. *Theory of Groups of Finite Order.* Dover Publications, INC, 1955.

[CF93] G. Cooperman and L. Finkelstein. Combinatorial tools for computational group theory. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Vol. 11, 1993.

[CH90] J. Cai and L. Hemachandra. On the power of parity polynomial time. *Mathematical Systems Theory*, 23(2):95–106, 1990.

[FFK94] S. Fenner, L. Fortnow, and S. Kurtz. Gap-definable counting classes. *Journal of Computer and System Sciences*, 48:116–148, 1994.

[FHL80] M. Furst, J. E. Hopcroft, and E. Luks. Polynomial time algorithms for permutation groups. In *Proc. of the 21st IEEE Symposium on Foundations of Computer Science*, pages 36–45, 1980.

[FK92] M. Fellows and N. Koblitz. Self-witnessing polynomial time complexity and prime factorization. In *Proc. of the 7th Structure in Complexity Theory Conference*, pages 107–110, 1992.

[For97] L. Fortnow. Counting complexity. *Complexity Theory Retrospective II, Springer Verlag*, 1997.

[FR96] L. Fortnow and N. Reingold. PP is closed under truth-table reductions. *Information and Computation*, 124(1):1–6, 1996.

[Gav93] R. Gavaldà. On the power of equivalence queries. In *Proc. of the EUROCOLT*, pages 193–203, 1993.

[Gil77] J. Gill. Computational complexity of probabilistic complexity classes. *SIAM Journal on Computing*, 6:675–695, 1977.

[GJ78] M. Garey and D. Johnson. *Computers and Intractability: A guide to the theory of* NP*-Completeness.* Freeman Sanfransisco, 1978.

[GS84] S. Grollmann and A. Selman. Complexity measure for public-key cryptosystems. In *Proc. of the 25th IEEE Symposium on Foundations of Computer Science*, pages 495–503, 1984.

[Gup95] S. Gupta. Closure properties and witness reduction. *Journal of Computer and System Sciences*, 50(3):412–432, 1995.

[GW93] R. Gavaldà and O. Watanabe. On the computational complexity of small descriptions. *SIAM Journal on Computing*, 22:1257–1275, 1993.

[Hal59] M. Hall. *The Theory of Groups.* Macmillan, New York, 1959.

[Heg95] Hegedüs. Generalized teaching dimensions and the query complexity of learning. In *Proc. of the 8th ACM Conference on Computational Learning Theory*, pages 108–117, 1995.

[Her90] U. Hertrampf. Relations among mod classes. *Theoretical Computer Science*, 74:325–328, 1990.

[HK71] K. Hoffmann and R. Kunz. *Linear Algebra.* Prentice Hall Inc, 1971.

[HPRV96] L. Hellerstein, K. Pillaipakkamnatt, V. Raghavan, and D. Vilkins. How many queries are needed to learn? *Journal of the Association for Computing Machinery*, 43(5):840–862, 1996.

[Kan85] W. Kantor. Sylow's theorem in polynomial time. *Journal of Computer and System Sciences*, 30:359–394, 1985.

[KL90] W. Kantor and E. Luks. Computing in quotient groups. In *Proc. of the 22nd ACM Symposium on Theory of Computing*, pages 524–534, 1990.

[Ko85] K. Ko. Continuous optimization problems and a polynomial hierarchy of real functions. *Journal of Complexity*, 1:210–231, 1985.

[Köb95] J. Köbler. On the structure of low sets. In *Proc. of the 10th IEEE Symposium on the Structure in Complexity Theory*, pages 246–261, 1995.

[KST92] J. Köbler, U. Schöning, and J. Torán. Graph isomorphism is low for PP. *Journal of Computational Complexity*, 2:301–310, 1992.

[KSTT92] J. Köbler, U. Schöning, S. Toda, and J. Torán. Turing machines with few accepting paths and low sets for PP. *Journal of Computer and System Sciences*, 44(2):272–286, 1992.

[Lad75] R. Ladner. On the structure of polynomial time reducibility. *Journal of the Association for Computing Machinery*, 22(1):155–171, 1975.

[Lub81] A. Lubiw. Some NP-complete problems similar to graph isomorphism. *SIAM Journal on Computing*, 10:11–21, 1981.

[Luk87] E. Luks. Computing the composition factors of a permutation groups in polynomial time. *Combinatorica*, 7:87–99, 1987.

[Luk92] E. Luks. Computing in solvable matrix groups. In *Proc. of the 33rd IEEE Symposium on Foundations of Computer Science*, pages 111–120, 1992.

[OH93] M. Ogiwara and L. Hemachandra. A complexity theory for feasible closure properties. *Journal of Computer and System Sciences*, pages 295–325, 1993.

[Pap94] C. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, 1994.

[PZ83] C. H. Papadimitriou and S. K. Zachos. Two remarks on the complexity of counting. In *Proceedings of the Sixth GI Conference of Theoretical Computer Science*, pages 269–276. Springer-Verlag, 1983. Lecture Notes in Computer Science #145.

[Sch83] U. Schöning. A low and a high hierarchy with in NP. *Journal of computer and System Sciences*, 27:14–28, 1983.

[Sch88] U. Schöning. Graph isomorphism is in the low hierarchy. *Journal of Computer and System Sciences*, 37:312–323, 1988.

[Sim70] C. Sims. Computational methods in the study of permutation groups. In *Computational Problems in Abstract Algebra*, pages 176–77. Ed. John Leech, Pergamon press, 1970.

[Sim75] J. Simon. *On some central problems in computational complexity*. PhD thesis, Cornell University, Ithaca, Newyork, 1975.

[TO92] S. Toda and M. Ogiwara. Counting classes are at least as hard as the polynomial-time hierarchy. *SIAM Journal on Computing*, 21(2):316–328, 1992.

[Tod91] S. Toda. PP is as hard as the polynomial-time hierarchy. *SIAM Journal on Computing*, 20(5):865–877, 1991.

[Tor88] J. Torán. An oracle characterization of the counting hierarchy. In *Proc. of the 3rd IEEE Structure in Complexity Theory Conference*, pages 213–223, 1988.

[Val76] L. G. Valiant. Relative complexity of checking and evaluating. *Information Processing Letters*, 5:20–23, 1976.

[Val79] L. G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8:189–201, 1979.

[Val84] L. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

[Vin97] N. V. Vinodchandran. Improved lowness results for solvable black-box group problems. In *Proc. of the 17th International Conference on the Foundations of Software Technology and Theoretical Computer Science*, pages 220–234, 1997. Lecture Notes in Computer Science # 1346.

[Wag86] K. W. Wagner. The complexity of combinatorial problems with succinct input representation. *Acta Informatica*, 23:325–356, 1986.

[Wat90] O. Watanabe. A formal study of learning via queries. In *Proc. of the 17th International Colloquium on Automata, Languages and Programming*, pages 139–152. Springer-Verlag, 1990. Lecture Notes in Computer Science # 443.

[WG94] O. Watanabe and R. Gavaldà. Structural analysis of polynomial time query learnability. *Mathematical Systems Theory*, 27:231–256, 1994.

Appendix

In this appendix we give the proofs of several group-theoretic and linear algebraic facts that we used in the thesis.

Lemma 4.2.6 *Let G be a finite abelian p-group. Let $g_1, g_2 \ldots, g_i$ be i independent elements of G of orders $p^{m_1}, p^{m_2}, \ldots, p^{m_i}$ respectively such that for all j, $1 \le j \le i$, the maximum order of any element in the factor group $G/\langle\{g_1, g_2 \ldots, g_j\}\rangle$ is $p^{m_{j+1}}$. Let $g'_{i+1}\langle\{g_1, g_2 \ldots, g_i\}\rangle$ be an element in the factor group $G/\langle\{g_1, g_2 \ldots, g_i\}\rangle$ of order $p^{m_{i+1}}$. Further, let $(x_1, x_2, \ldots, x_i)$ be the unique $\{g_1, g_2 \ldots, g_i\}$-exponent of $(g'_{i+1})^{p^{m_{i+1}}}$. Then $p^{m_{i+1}}$ divides x_j for $1 \le j \le i$. Let $y_j = x_j/p^{m_{i+1}}$ for $1 \le j \le i$. Then $g_{i+1} = g'_{i+1} g_1^{-y_1} g_2^{-y_2} \ldots g_i^{-y_i}$ is an element of G of order $p^{m_{i+1}}$ which is independent of $\{g_1, g_2 \ldots, g_i\}$.*

Proof. From the statement of the lemma we have $(g'_{i+1})^{p^{m_{i+1}}} = g_1^{x_1} g_2^{x_2} \ldots g_i^{x_i}$. First we show that $p^{m_{i+1}}$ divides x_j for $1 \le j \le i$. Fix a j; $1 \le j \le i$. Let us focus on the group generated by $\{g_1, g_2 \ldots, g_j\}$. Since any element of the factor group $G/\langle\{g_1, g_2 \ldots, g_j\}\rangle$ is of order at most p^{m_j} we have ${g'_{i+1}}^{p^{m_j}} \in \langle\{g_1, g_2 \ldots, g_j\}\rangle$. But notice that $m_j \ge m_{i+1}$. Hence

$$\begin{aligned}(g'_{i+1})^{p^{m_j}} &= ({g'_{i+1}}^{p^{m_{i+1}}})^{p^{m_j - m_{i+1}}} \\ &= (g_1^{x_1} \ldots g_i^{x_i})^{p^{m_j - m_{i+1}}} \in \langle\{g_1, g_2 \ldots, g_j\}\rangle.\end{aligned}$$

Now, since for all k, $j \le k \le i$, g_k is independent of $\{g_1, g_2 \ldots, g_i\} - \{g_k\}$, we have $(g_k^{x_k})^{p^{m_j - m_{i+1}}} = e$. In particular $(g_j^{x_j})^{p^{m_j - m_{i+1}}} = e$. But the order of g_j is p^{m_j} and it should divide $x_j(p^{m_j - m_{i+1}})$. Hence $p^{m_{i+1}}$ should divide x_j.

To show that $g_{i+1} = g'_{i+1} g_1^{-y_1} g_2^{-y_2} \ldots g_i^{-y_i}$ is of order $p^{m_{i+1}}$ in G and independent of $\{g_1, g_2 \ldots, g_i\}$, first notice that g_{i+1} and g'_{i+1} are in the same coset of $\langle\{g_1, g_2 \ldots, g_i\}\rangle$ in G. Hence it is enough to show that $(g_{i+1})^{p^{m_{i+1}}} = e$. But, since $p^{m_{i+1}} y_j = x_j$ and $(g'_{i+1})^{p^{m_{i+1}}} = g_1^{x_1} \ldots g_i^{x_i}$, the result follows. ∎

Lemma 4.2.7 *Let G be a finite group of order n and let $p_1^{e_1}p_2^{e_2}\dots p_r^{e_r}$ be the complete prime factorization of n. Let H be an abelian subgroup of G and is generated by the set $\{g_1, g_2 \dots, g_s\} \subseteq H$. Then for each j, the p_j-Sylow subgroup of H is generated by $X_j = \{g_i^{(n/p_j^{e_j})} \mid 1 \le i \le s\}$.*

Proof. It is not hard to see that for each j, the group $\langle X_j\rangle$ generated by X_j is a subgroup of the (unique) p_j-Sylow subgroup of H. Therefore, it is enough to show that each generator of H can be generated using products of appropriate elements from the different X_j. Consider a generator g_l. Let $o(g_l) = n_l$. Since $\gcd(n_l, \Sigma n/p_j^{e_j}) = 1$, it follows that g_l is in the cyclic group generated by $\prod_{j=1}^{r} g_l^{n/p_j^{e_j}}$.

■

Proposition 4.2.8 *Let G, H and K be finite groups such that $H, K < G$ and K is a normal subgroup of H. Let $|G| = n = p_1^{e_1}\dots p_r^{e_r}$ be the unique prime factorization of the order of G. Then for any element hK in H/K, the order $o(hK)$ is of the form $p_1^{d_1}\dots p_r^{d_r}$, where for all i; $1 \le i \le r$, d_i is the smallest integer j such that $(h^{n/p_i^{e_i}})^{p_i^j} \in K$.*

Proof. We shall prove the result for the case when the group K is trivial. The proof for the general case is identical. Let h be any element of H and for $1 \le i \le r$, d_i be the smallest integer j such that $(h^{n/p_i^{e_i}})^{p_i^j} = e$. Let us denote the number $n/p_i^{e_i}$ by N_i, for $1 \le i \le r$. Also denote $p_1^{d_1}\dots p_r^{d_r}$ by N. We first show that $o(h)$ divides N. Since for each i, $h^{N_i p_i^{d_i}} = e$, we have $o(h)$ divides $N_i p_i^{d_i}$ for each i; $1 \le i \le r$. This means that $o(h)$ divides $\gcd(\{N_i p_i^{d_i} \mid 1 \le i \le r\})$, which is the same as N. Hence $o(h)$ divides N.

To show that N divides $o(h)$, assume that $o(h) < N$. But we already have $o(h)$ divides N. Hence there exists a prime p_i; $1 \le i \le r$, such that $h^{N/p_i} = e$. But this means that $(h^{N_i})^{p_i^{d_i-1}} = e$. This contradicts the definition of d_i. Hence N divides $o(h)$. The result follows. ■

Proposition 4.3.13 *Let G be a finite group generated by the set S. Then, the commutator subgroup of G is the normal closure of the set $\{ghg^{-1}h^{-1} \mid g, h \in S\}$ in G.*

Proof. Let N be the normal closure of the set $X = \{ghg^{-1}h^{-1} \mid g, h \in S\}$ in G and G' be the commutator subgroup of G. Recall that $G' = \langle\{ghg^{-1}h^{-1} \mid g, h \in G\}\rangle$. It is a well known group-theoretic fact that G' is normal in G and G/G' is abelian. Moreover if H is a normal subgroup of G such that G/H is abelian, then $G' < H$. Now, from the definition of N and G' and the fact that G' is normal in G, it follows that $N < G'$. Hence to show that $N = G'$, it is enough to show that G/N is abelian. Firstly observe that G/N is generated by $\{gN \mid g \in S\}$. But for all $g, h \in S$, $(gN)(hN)(gN)^{-1}(hN)^{-1} = N$. Hence the result. ∎

Proposition 5.3.1 *Let p be a prime. Then there are $(p-2)!$ cyclic subgroups of S_p generated by p-cycles.*

Proof. The number of p-cycles in S_p is $(p-1)!$. Let g be any arbitrary p-cycle. Since p is a prime, all the $p-1$ elements except identity in $\langle g \rangle$ are p-cycles and all of them generate the group $\langle g \rangle$. Also, if $h \notin \langle g \rangle$ is any p-cycle, then from Lagrange's theorem it follows that $\langle g \rangle \cap \langle h \rangle = \{e\}$ where e is the identity permutation. Hence there are $(p-1)!/(p-1) = (p-2)!$ cyclic subgroups in S_p generated by p-cycles. ∎

Lemma 5.5.2 *Let $G < S_n$.*

1. *$|G^{(i)}| = \prod_{j>i} d_j$ for each i, $1 \le i \le n$.*

2. *The number of strong generator sets for $G^{(i)}$ is $\prod_{j=i+1}^{n} |G^{(j)}|^{d_j}$ for each i, $1 \le i \le n$.*

3. Let $X \subseteq \{1, 2, \ldots, n\}$ and $\sigma \in S_n$. Then $|G_{[\sigma,X]}| = |G_{[X]}|$

Proof. (1). From the definitions it follows that for any i, $G^{(i)}$ is the disjoint union of d_{i+1} right cosets of $G^{(i+1)}$ in $G^{(i)}$. Hence, $|G^{(i)}| = d_{i+1}|G^{(i+1)}|$. The result follows by induction.

(2). Consider the chain of subgroups $\{e\} = G^{(n)} < G^{(n-1)} < \ldots < G^{(0)} = G$. Then the set $\bigcup_{i=1}^{n} T_i$ forms a strong generator set for G where T_i is a complete right transversal of $G^{(i)}$ in $G^{(i-1)}$. Observe that for $1 \leq i \leq j \leq n$, if there exists a permutation in G that fixes 1 to $i-1$ and maps i to j, then there is an entire right coset of $G^{(i)}$ in $G^{(i-1)}$ that fixes 1 to $i-1$ and maps i to j. Consequently, any one element of this right coset can be included in T_i. Thus for each pair $\langle i, j\rangle$, $i \leq j$, there are $|G^{(i)}|$ choices for inclusion of a generator in T_i that fixes 1 to $i-1$ and maps i to j. Hence if d_i is the cardinality of T_i then there are $|G^{(i)}|^{d_i}$ complete right transversals of $G^{(i)}$ in $G^{(i-1)}$. Hence the result.

(3). The result follows from the observation that for any σ, $G_{[\sigma,X]}$ is a right coset of $G_{[X]}$ in G. ∎

Proposition 5.5.5 *Let F be a field and U be a nontrivial subspace of F^n. If k is the 0-index of U then $U^{(k)}$ is a 1-dimensional subspace.*

Proof. Let us recall the definition of 0-index. For $1 \leq i \leq n$, $U^{(i)}$ is the subspace $\{u \in U \mid \text{for } 1 \leq j \leq i: \ u[j] = 0\}$ of U. $U^{(0)}$ denotes U. The 0-index of U is the maximum k such that $U^{(k)}$ is nontrivial.

Define V_i to be the subspace $\{v \in F_n \mid v[j] = 0; \text{ for } 1 \leq j \leq n\}$ of F^n and let V_0 denote F^n. It is clear that for $1 \leq j \leq n$, V_i is of dimension $n - i$. Now, we have $U^{(i)} = U \cap V_i$.

Since k is the 0-index of U we have $U^{(k+1)}$ is trivial but $U^{(k)}$ is nontrivial. Suppose the dimension of $U^{(k)}$ is greater than 1. Let u_1 and u_2 be two linearly independent nonzero vectors in $U^{(k)}$. Let a_k be a vector in some fixed basis of V_k that is not in V_{k+1}. Then there exists nonzero scalars $\alpha_1, \alpha_2 \in F$ and vectors $v_1, v_2 \in V_{k+1}$ such that

$$\begin{aligned} u_1 &= v_1 + \alpha_1 a_k \\ u_2 &= v_2 + \alpha_2 a_k \end{aligned}$$

Now consider the vector $u = \alpha_2 u_1 - \alpha_1 u_2$. This is a nonzero vector (since u_1 and u_2 are independent) which is identical to $\alpha_2 v_1 - \alpha_1 v_2$ and hence in V_{k+1}. But we already have $u \in U$. Hence $u \in U^{(k+1)}$ which is a contradiction to the fact that $U^{(k+1)}$ is trivial. ■

Proposition 6.2.3 *Let G be a cyclic subgroup of S_n.*

1. *If G is of order p^k, for prime p, then $p^k \leq n$.*
2. *Let $|G|$ has the prime factorization $p_1^{e_1} \dots p_l^{e_l}$. Then the number of elements in G of order $p_i^{k_i}$ is $p_i^{k_i} - p_i^{k_i - 1}$, for each i and $k_i \leq e_i$.*

Proof. The first part immediately follows from the fact if $o(\sigma) = p^k$, for $\sigma \in S_n$ and prime p, then the cycle decomposition of σ has a cycle of length p^k. For the second part, recall that if G is cyclic and d divides $|G|$ there is a unique cyclic subgroup of G of order d. Also, any cyclic group of order d has $\phi(d)$ generators, where ϕ is the Euler totient function. The proposition follows since $\phi(p_i^{k_i}) = p_i^{k_i} - p_i^{k_i - 1}$ for $1 \leq i \leq l$. ■