Graph Isomorphism is in SPP ⋆

V. Arvind * and Piyush P Kurur [1]

Institute of Mathematical Sciences, Chennai 600113, India

Abstract

We show that Graph Isomorphism is in the complexity class SPP, and hence it is in $\oplus$P (in fact, in Mod_kP for each $k \geq 2$). These inclusions for Graph Isomorphism were not known prior to membership in SPP. We derive this result as a corollary of a more general result: we show that a *generic problem* FIND-GROUP has an $\mathrm{FP}^{\mathrm{SPP}}$ algorithm. This general result has other consequences: for example, it follows that the *hidden subgroup problem* for permutation groups, studied in the context of quantum algorithms, has an $\mathrm{FP}^{\mathrm{SPP}}$ algorithm. Also, some other algorithmic problems over permutation groups known to be at least as hard as Graph Isomorphism (e.g. coset intersection) are in SPP, and thus in Mod_kP for each $k \geq 2$.

Key words: Graph Isomorphism, Counting Classes, SPP, Lowness.

1 Introduction

The Graph Isomorphism problem —of testing if two graphs are isomorphic— is a well-studied algorithmic problem in the class NP. Formally, the decision problem GI (for Graph Isomorphism) is defined as:

$$\mathrm{GI} = \{\langle X_1, X_2\rangle \mid X_1 \text{ and } X_2 \text{ are isomorphic graphs}\}.$$

⋆ A preliminary version of the results in this paper were presented at the IEEE FOCS 2002 conference.

* Corresponding author.
Email addresses: arvind@imsc.res.in (V. Arvind), ppk@imsc.res.in (Piyush P Kurur).

[1] Present Address: Department of Computer Science and Engineering, I.I.T, Kanpur, Kanpur 208016, India.

It is an outstanding open problem in computational complexity whether Graph Isomorphism has a polynomial-time algorithm. This problem has stimulated a great deal of research in algorithms and complexity over the years. There is strong evidence that Graph Isomorphism is not NP-complete. In [2] (also see [5]) it was shown that Graph Nonisomorphism is in AM implying that GI is in NP $\cap$ coAM. It follows that GI cannot be NP-complete unless the polynomial hierarchy collapses to Σ_2^p [10,29]. Schöning, who introduced the notion of lowness in complexity theory, pointed out in [29] that GI is *low* for Σ_2^p. I.e. GI is powerless as oracle for Σ_2^p.

Subsequently, it was shown in [22] that GI is also *low* for the counting complexity class PP (PP is the language class corresponding to #P). This result is proven using the machinery of GapP functions introduced in the seminal paper by Fenner, Fortnow, and Kurtz [13] on gap-definable counting classes. The study of counting complexity classes is an area of research in structural complexity theory motivated by Valiant's class #P (see e.g. [13]). Intuitively, counting complexity classes are defined by suitable restrictions on the number of accepting and rejecting paths in nondeterministic Turing machines. In [13] the languages classes SPP and LWPP are introduced as generalizations of Valiant's class UP. It is shown in [13] that UP $\subseteq$ SPP $\subseteq$ LWPP, and LWPP is low for PP.

After Shor's breakthrough quantum polynomial-time algorithms for integer factoring and discrete log [31] a natural question is whether Graph Isomorphism is in BQP (the class of problems solvable in quantum polynomial time). The hidden subgroup problem was formulated to generalize Shor's algorithmic technique. In particular, Graph Isomorphism can be seen as an instance of the hidden subgroup problem.

How does the class BQP relate to standard complexity classes defined using classical Turing machines? Fortnow and Rogers [15] show that BQP is contained in the counting complexity class AWPP (definitions follow). Thus, in a sense, we can also think of BQP as a counting class.

1.1 Summary of new results

In this paper, we show that Graph Isomorphism is in the class SPP. This was left as an open question in [22] (also see [13]). As a consequence it follows that GI is in and low for $\mathrm{Mod}_k\mathrm{P}$ for each $k \geq 2$. Previously, only a special case of Graph Isomorphism, namely Tournament Isomorphism, was known to be in $\oplus$P. [2]

[2] Tournament Isomorphism in $\oplus$P follows because any tournament has an odd number of automorphisms. There are special cases of Graph Isomorphism, e.g.

What we prove is a more general result: we show that a generic problem FIND-GROUP is in $\mathrm{FP}^{\mathrm{SPP}}$ as a consequence of which GI and several other algorithmic problems on permutation groups that are not known to have polynomial-time algorithms turn out to be in SPP. In particular, as another corollary, we show that the hidden subgroup problem (HSP) over permutation groups is in $\mathrm{FP}^{\mathrm{SPP}}$. The hidden subgroup problem is of interest in the area of quantum algorithms.

Outline of the $\mathrm{FP}^{\mathrm{SPP}}$ *algorithm*

To indicate how the proof of our main theorem will proceed, we give a broad outline of the $\mathrm{FP}^{\mathrm{SPP}}$ algorithm for the specific problem of computing a generator set for the automorphism group $G = \mathrm{Aut}(X)$ of a graph X on n vertices (this problem is polynomial-time equivalent to GI). Since G is a subgroup of S_n, it has the following tower of subgroups

$$\mathbf{1} = G^{(n-1)} \leq G^{(n-2)} \leq \ldots \leq G^{(1)} \leq G^{(0)} = G,$$

where $G^{(i)}$ is the subgroup of G that fixes the points $1, 2, \ldots, i$.

Our algorithm will compute a generator set for G by computing the coset representatives of $G^{(i)}$ in $G^{(i-1)}$ for each i. Starting with $G^{(n-1)}$, the algorithm will compute what is known as a strong generator set for $G^{(i)}$ in decreasing order of i until finally it computes a strong generator set for $G^{(0)} = G$.

If G were given by its generator set as input, then it is well-known that a strong generator set for G can be computed in polynomial time. These ideas were developed in [32,16] to design a polynomial-time membership test for permutation groups. These ideas play an important role in the design of our algorithm. For our problem notice that we do not have access to a generator set for $G = \mathrm{Aut}(X)$. Indeed, a generator set for $\mathrm{Aut}(X)$ is what the algorithm has to compute. Our algorithm will use an NP oracle to access elements of G from different subgroups in the above tower. An important aspect that yields the $\mathrm{FP}^{\mathrm{SPP}}$ bound is that the queries made by the algorithm to the NP oracle are carefully chosen. A key procedure we use here is a polynomial-time algorithm for finding the lexicographically least element in a coset Hg of a permutation group $H \leq S_n$ and $g \in S_n$.

The plan of the paper is as follows: in the next section we explain notation and give preliminary definitions and results, particularly concerning SPP and

---

Graph Isomorphism for bounded-degree graphs or bounded genus graphs, that have polynomial-time algorithms.

related counting complexity classes. In Sections 3 and 4 we develop the ingredients leading to the proof of our main result that there is an $\mathrm{FP}^{\mathrm{SPP}}$ algorithm for the FIND-GROUP problem, and derive as corollary that GI is in SPP. In Sections 5 and 6 we give further applications of the main result. Finally, we state some open problems.

2 Preliminaries

Following standard notation, we use Σ to denote the alphabet $\{0, 1\}$ and Σ^* denotes the set of all finite strings over Σ. The length of a string $x \in \Sigma^*$ is denoted by $|x|$. Let $\mathbb{Z}$ denote the set of integers.

As usual, the class of languages computable in polynomial time is denoted by P, and the class of polynomial-time computable functions is denoted by FP. The class of languages computable in nondeterministic polynomial time is denoted by NP. Other basic notions from complexity theory that we require in this paper can be found in standard textbooks such as Balcázar et al's texts [7,8]. We now focus on definitions of counting complexity classes, with particular emphasis on gap-definable classes, and give a brief description of some of their properties relevant to the present article. Details can be found in [13–15,12].

2.1 SPP and other Counting Complexity Classes

Fenner, Fortnow and Kurtz defined gap-definable functions [13] using which they examined several counting complexity classes like PP, $\mathrm{C}_{=}\mathrm{P}$, $\mathrm{Mod}_k\mathrm{P}$, and SPP.

Definition 1 *A function $f : \Sigma^* \to \mathbb{Z}$ is said to be* gap-definable *if there is a nondeterministic polynomial time Turing machine M such that, for each $x \in \Sigma^*$, $f(x)$ is the difference between the number of accepting paths and the number of rejecting paths of M on input x. More precisely, if $acc_M(x)$ denotes the number of accepting paths and $rej_M(x)$ the number of rejecting paths of M on input x, then*

$$f(x) = acc_M(x) - rej_M(x).$$

Let GapP denote the class of gap-definable functions [13]. For each nondeterministic polynomial time Turing machine M let gap_M denote the GapP function defined by it.

Recall that a language L is in UP if there is a nondeterministic polynomial-time Turing machine M accepting L such that M has at most one accepting path on any input. The class UP was defined by Valiant in [33] and it captures the complexity of 1-way functions.

The complexity class SPP introduced in [13] is the GapP analogue of UP. The class LWPP, also introduced in [13], contains SPP. We recall their definitions.

Definition 2

(1) A language L is in SPP *if there is a nondeterministic polynomial-time Turing machine M such that*

$$x \in L \ \textit{implies} \ \textit{gap}_M(x) = 1,$$
$$x \notin L \ \textit{implies} \ \textit{gap}_M(x) = 0.$$

(2) A language L is in LWPP *if there are a nondeterministic polynomial-time Turing machine M and an* FP *function g such that*

$$x \in L \ \textit{implies} \ \textit{gap}_M(x) = g(1^{|x|}),$$
$$x \notin L \ \textit{implies} \ \textit{gap}_M(x) = 0.$$

In either case we say that L is accepted *by the machine M.*

We note that UP $\subseteq$ SPP $\subseteq$ LWPP. The standard counting complexity classes PP and Mod_kP can also be defined using gap-definable functions.

Definition 3 [13]

(1) A language L is in PP *if there is a nondeterministic polynomial-time Turing machine M such that*

$$x \in L \iff \textit{gap}_M(x) > 0.$$

(2) A language L is in Mod_kP *(for $k \geq 2$) if there is a nondeterministic polynomial-time Turing machine M such that*

$$x \in L \iff \textit{gap}_M(x) \neq 0 \ (\textit{mod } k).$$

Indeed, the above definitions are examples of a general notion of gap-definable complexity classes introduced and studied in [13]. It is shown in [13] that SPP is the minimal gap-definable class in a certain sense.

By relativizing the nondeterministic polynomial-time Turing machines we can define the relativized class GapP^A, for oracle $A \in \Sigma^*$. Thus, we can define the relativized complexity classes SPP^A, PP^A, and Mod_kP^A.

The notion of lowness was first introduced in complexity theory by Schöning in [28]. We recall the definition.

Definition 4 *Let $\mathcal{C}$ be a relativizable complexity class. We say that a language $A \in \Sigma^*$ is* low *for $\mathcal{C}$ if $\mathcal{C}^A = \mathcal{C}$.*

In particular we are interested in languages that are low for the class PP. We summarize as a theorem some properties of SPP from [13] related to lowness.

Theorem 5 [13]

(1) Every language in SPP *(indeed, even in the larger class* LWPP*) is low for* PP*. More precisely,* $\mathrm{PP}^{\mathrm{LWPP}} = \mathrm{PP}$.
(2) $\mathrm{SPP} \subseteq \mathrm{Mod}_k\mathrm{P}$ *for all* $k \geq 2$*. Moreover,* $\mathrm{SPP}^{\mathrm{SPP}} = \mathrm{SPP}$.

We note here that Graph Isomorphism was shown to be low for PP in [22] by proving that it is in LWPP. It is also shown in [22] that GA (testing if a given graph has a nontrivial graph automorphism) is in SPP. It is known that GA is polynomial-time reducible to GI, but the converse is open.

Recall that BPP denotes the class of languages with polynomial-time randomized algorithms with error probability bounded by, say, 1/3. The class BPP is also known to be low for PP [21].

The complexity class AWPP was introduced in [14]. More recently, Fenner [12] has shown a sort of gap amplification property for AWPP which yields the following neat definition for this class.

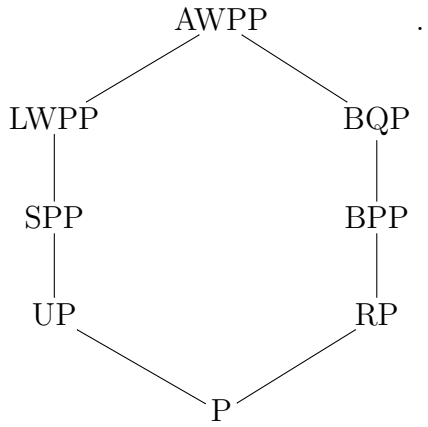
Definition 6 *A language L is in the class* AWPP *if there is a nondeterministic polynomial-time Turing machine M and a polynomial p such that for all $x \in \Sigma^*$*

$$x \in L \ \textit{ implies } \ 2/3 \leq \frac{gap_M(x)}{2^{p(|x|)}} \leq 1,$$
$$x \notin L \ \textit{ implies } \ 0 \leq \frac{gap_M(x)}{2^{p(|x|)}} \leq 1/3.$$

The class AWPP generalizes both BPP and SPP, and it is shown in [14] that every language in AWPP is low for PP.

Let BQP denote the class of languages that have quantum polynomial-time algorithms with bounded error probability (say 1/3). To complete the picture relating these classes, Fortnow and Rogers in [15] have shown that BQP is contained in AWPP and hence BQP is also low for PP.

It is interesting to note that NP ∩ co-NP is *not* known to be low for PP. Here is a diagram that shows the containments between the complexity classes discussed here.



Although no containment is known between BQP and SPP, it is interesting to compare these classes in terms of natural problems they contain. In the present paper we show that Graph Isomorphism and the hidden subgroup problem for permutation groups are in SPP. These problems have resisted efficient deterministic or randomized algorithms, but they are still considered as problems likely to have polynomial-time quantum algorithms. On the other hand, $\mathrm{FP}^{\mathrm{SPP}}$ contains Integer Factoring and Discrete Log that have polynomial-time quantum algorithms. In fact, these problems are even in $\mathrm{FP}^{\mathrm{UP}}$. Also, as $\mathrm{P}^{\mathrm{SPP}} = \mathrm{SPP}$, notice that the class $\mathrm{FP}^{\mathrm{SPP}}$ is essentially SPP: for $f \in \mathrm{FP}^{\mathrm{SPP}}$ and input x, the bits of $f(x)$ can be computed in SPP. A similar closure property holds for BQP.

As mentioned before, SPP is contained in and is low for the complexity classes $\mathrm{Mod}_k\mathrm{P}$, $\mathrm{C}_=\mathrm{P}$, and PP. Also, SPP has other nice properties (see [13] for details). For instance, SPP is characterized exactly as the class of languages low for GapP. In summary, SPP can be seen as the GapP analogue of UP and is a robust complexity class.

Let M be a nondeterministic polynomial-time oracle Turing machine. Suppose A is a language in NP accepted by some NP machine N. We say that M^A makes *UP-like queries* to the oracle A if on all inputs x, $M^A(x)$ makes *only* such queries y for which $N(y)$ has *at most* one accepting path. Effectively, it is like M having access to a UP oracle. We now state a useful variant of a result from [22,23].

Theorem 7 ([22]) *Let M be a nondeterministic polynomial-time oracle machine with oracle $A \in \mathrm{NP}$ such that M^A makes UP-like queries to A then the function $h(x) = gap_{M^A}(x)$ is in* GapP.

The following lemma, which is a straightforward consequence of Theorem 7 and of Theorem 5, is in a form useful for this paper.

Lemma 8

- *Suppose L is in* SPP^A *accepted by the nondeterministic polynomial-time oracle machine M^A with oracle $A \in \mathrm{NP}$ (i.e. $x \in L$ implies that $gap_{M^A}(x) = 1$, and $x \notin L$ implies that $gap_{M^A}(x) = 0$), such that the machine M^A makes UP-like queries to A, then L is in* SPP.
- *Suppose a function $f : \Sigma^* \to \Sigma^*$ is in* FP^A *(i.e. f is computed by a polynomial-time oracle transducer M^A) where $A \in \mathrm{NP}$, such that the machine M^A makes UP-like queries to A, then f is in* $\mathrm{FP}^{\mathrm{SPP}}$.

2.2 Permutation group preliminaries

The set of all permutations on the set $[n] = \{1, 2 \ldots, n\}$ is a group under composition of permutations. This group is the *symmetric group of degree n* and is denoted by S_n. A *permutation group* on the set $[n]$ is a subgroup of S_n.

We use letters $g, h, \ldots, \sigma, \tau, \pi, \ldots$ with subscripts and superscripts to denote elements of S_n and i, j and k for the elements of the set $[n]$. Subsets and subgroups of S_n are denoted by capital letters A, G, H etc. For two groups G and H, we write $H \leq G$ to denote that H is a subgroup of G (not necessarily a proper subgroup).

We use the following standard notation in permutation group theory [35,25]. For $g \in S_n$ and $i \in [n]$, we denote by i^g the image of i under permutation g. The composition $g_1 g_2$ of permutations $g_1, g_2 \in S_n$ is defined *left to right*: i.e. applying g_1 first and then g_2. More precisely, $i^{g_1 g_2} = (i^{g_1})^{g_2}$ for all $i \in [n]$. For $A \subseteq S_n$ and $i \in [n]$ we denote the set $\{i^g | g \in A\}$ by i^A. In particular, if $A \leq S_n$ then i^A is the *orbit* of i under the action of A on $[n]$.

If $G \leq S_n$ then for each $i \in [n]$, we let $G^{(i)}$ denote the subgroup $\{g \in G \mid j^g = j$ for each $j \in [i]\}$. $G^{(i)}$ is called the *pointwise stabilizer* of $[i]$ in G.

The identity permutation is denoted by 1 (we use 1 to denote the identity of all groups) and the subgroup consisting of only 1 is denoted $\mathbf{1}$. The permutation group *generated* by a subset A of S_n is the smallest subgroup of S_n containing A and is denoted by $\langle A \rangle$.

For the algorithmic problems considered in this paper, we assume that a permutation π in S_n is presented as the ordered sequence $(1^\pi, 2^\pi, \ldots, n^\pi)$. Further, we assume that subgroups of S_n are presented by generator sets.

Let G be a group and H be a subgroup of G. For $\varphi \in G$ the subset $H\varphi = \{\pi\varphi : \pi \in H\}$ of G is a *right coset* of H in G. Two right cosets of H in G are either disjoint or identical. Thus, the right cosets of H in G form a partition of G. When G is finite this partition is finite and can be written as $G = H\varphi_1 \cup H\varphi_2 \cup \ldots \cup H\varphi_k$. Each right coset of H has cardinality equal to $|H|$ and the set $\{\varphi_1, \varphi_2, \ldots, \varphi_k\}$ is a set of *distinct coset representatives* of H in G.

As developed by Sims [32], pointwise stabilizers are fundamental in the design of algorithms for permutation group problems. The structure used is the chain of stabilizers subgroups in G given by: $\mathbf{1} = G^{(n)} \leq G^{(n-1)} \leq \ldots \leq G^{(1)} \leq G^{(0)} = G$. Let C_i be a complete set of right coset representatives of $G^{(i)}$ in $G^{(i-1)}, 1 \leq i \leq n$. Then $\bigcup_{i=1}^{n-1} C_i$ forms a generator set for G. Such a generator set is called a *strong generator set* for G [32,16]. Any $g \in G$ has a unique factorization $g = g_1 g_2 \ldots g_n$, with $g_i \in C_i$.

We now recall two basic algorithmic results concerning permutation groups that are essential ingredients in the proof of our main result in Section 4. These algorithms are originally due to Sims [32], and the polynomial-time analysis is from [16]. Further details can be found in the survey article by Luks [25] and the monograph by Hoffman [18].

Theorem 9 *Given as input the generator set S for a permutation group $G \leq S_n$, the following two basic algorithmic tasks can be implemented in time polynomial in n*

(1) For each element $i \in [n]$, its orbit $i^G = \{i^g \mid g \in G\}$, can be computed in polynomial time. Furthermore, for each j in the orbit i^G we can compute in polynomial time an element $g \in G$ such that $i^g = j$.
(2) The tower of subgroups $\mathbf{1} = G^{(n)} \leq G^{(n-1)} \leq \ldots \leq G^{(1)} \leq G$ can be computed in time polynomial in n. (I.e. the right coset representative sets C_i for the groups $G^{(i)}$ in $G^{(i-1)}, 1 \leq i \leq n$ can be computed in polynomial time giving a strong generator set for each $G^{(i)}$ including G).

3 Computing the least element of a right coset

We define the lexicographic ordering $\prec$ of permutations in S_n induced by the natural order of $[n]$ as follows: For two permutations $\pi \neq \tau \in S_n$ we say that $\pi \prec \tau$ if for some $i \in [n]$ we have

$$i^\pi < i^\tau \text{ and } \quad j^\pi = j^\tau \text{ for } 1 \leq j \leq i-1.$$

Clearly, this is a total order on S_n. Writing a permutation π as the ordered

sequence $(1^\pi, 2^\pi, \ldots, n^\pi)$ this is clearly the natural lexicographic ordering on these sequences with the sequence $(1, 2, \ldots, n)$ as the least element of S_n and the sequence $(n, n-1, \ldots, 1)$ as the last element of S_n.

In this section we describe a simple polynomial-time algorithm that takes as input a permutation group $\langle A \rangle = G \leq S_n$ and a permutation $\sigma \in S_n$ and computes the lexicographically least element of the right coset $G\sigma$ of G in S_n. This algorithm is a crucial ingredient in the proof of the main theorem in the next section.

Theorem 10 *There is a polynomial-time algorithm that takes as input a permutation group $\langle A \rangle = G \leq S_n$ and a permutation $\sigma \in S_n$ and computes the lexicographically least element of the right coset $G\sigma$.*

PROOF.

We describe the algorithm and then argue its correctness.

Input: $G \leq S_n$, $\sigma \in S_n$;
Output: Lexicographically least element in $G\sigma$;
Let $G^{(n)} \leq G^{(n-1)} \leq \ldots \leq G^{(1)} \leq G^{(0)} = G$ be the tower of subgroups of G where, by Theorem 9, the generator set for each $G^{(i)}$ and the strong generator set for G can be computed in polynomial time;
$\pi_0 = \sigma$;
  for $i := 0$ **to** $n-1$
    find the element y in $(i+1)^{G^{(i)}}$ such that y^{π_i} is minimum;
    (* This can be done in polynomial time as the entire orbit $(i+1)^{G^{(i)}}$ of $i+1$ in $G^{(i)}$, which is a set of size at most $n-i$, can be computed in polynomial time by applying Theorem 9, and finding the minimum in the orbit takes linear time. *);
    Let $g_i \in G^{(i)}$ be such that $(i+1)^{g_i} = y$;
    (* By Theorem 9, g_i can be computed in polynomial time *);
    $\pi_{i+1} := g_i \pi_i$;
  endfor;
Output π_n;

Since $\pi_0 = \sigma$ and $G^{(n-1)} = \{1\}$, it suffices to prove the following claim in order to show that the algorithm computes the lexicographically least element of $G\sigma$.

Claim 11 *For all $0 \leq i < n-1$ the lexicographically least element of $G^{(i)}\pi_i$ is in $G^{(i+1)}\pi_{i+1}$.*

Proof of Claim. By definition, $\pi_{i+1} = g_i \pi_i$, where g_i is in $G^{(i)}$ such that g_i

maps $i+1$ to $y \in (i+1)^{G^{(i)}}$ and such that $y^{\pi_i} = x$ is the minimum element in $\{z^{\pi_i} \mid z \in (i+1)^{G^{(i)}}\}$. Since $G^{(i)}$ fixes each element in the set $[i]$ and since $g_i \in G^{(i)}$, we can see that for every $1 \leq k \leq i$, for each $g \in G^{(i)}$ and $h \in G^{(i+1)}$, we have $k^{h\pi_{i+1}} = k^{\pi_{i+1}} = k^{g_i\pi_i} = k^{\pi_i} = k^{g\pi_i}$. In particular if ρ is the lex-least element of $G^{(i)}\pi_i$, every element in $G^{(i+1)}\pi_{i+1}$ agrees with ρ on the first i elements.

Furthermore, for each $g \in G^{(i+1)}$ notice that $(i+1)^{g\pi_{i+1}} = (i+1)^{\pi_{i+1}} = (i+1)^{g_i\pi_i} = x$, where x is defined above. It is clear that $G^{(i+1)}\pi_{i+1}$ is precisely the subset of $G^{(i)}\pi_i$ each of whose elements maps $i+1$ to x. Together with the fact that $(i+1)^{\rho} = x$ (by the lex-least property of ρ), we get the desired conclusion.

By induction and the above claim it follows that the lex-least element of $G\sigma = G^{(0)}\pi_0$ is in $G^{(n)}\pi_n = \{\pi_n\}$. Thus, π_n is the desired lexicographically least element of $G\sigma$. □

The polynomial-time algorithm of Theorem 10 can be generalised to compute the lexicographically least element of $\tau G\sigma$.

Corollary 12 *There is a polynomial-time algorithm that takes as input a permutation group $\langle A\rangle = G \leq S_n$ and two permutations $\tau, \sigma \in S_n$, and computes the lexicographically least element of $\tau G\sigma$. In particular, the lexicographically least element of a left coset τG can also be computed in polynomial time.*

PROOF. Notice that $\tau G\sigma = \tau G\tau^{-1}\tau\sigma$ and $\tau G\tau^{-1}$ is a subgroup of S_n with generating set $\{\tau g\tau^{-1} \mid g \in A\}$. The result follows directly from Theorem 10 applied to the group $\tau G\tau^{-1}$ and the permutation $\tau\sigma$. □

4 Graph Isomorphism in SPP

We are ready to prove the main theorem of the paper. Recall that the Graph Isomorphism problem is the following decision problem: GI = $\{(X_1, X_2) \mid X_1$ and X_2 are isomorphic$\}$. A related problem is AUTO which is a functional problem: given a graph X as input the problem is to output a strong generator set for $\mathrm{Aut}(X)$. It is well-known from the result of Mathon [26] (see e.g. [23]) that GI and AUTO are polynomial-time Turing equivalent.

Thus, in order to show that GI $\in$ SPP it suffices to show that AUTO $\in \mathrm{FP}^{\mathrm{SPP}}$. In other words, it suffices to show that there is a deterministic polynomial-

time Turing machine M with oracle $A \in \mathrm{SPP}$ that takes a graph X as input and outputs a strong generator set for $\mathrm{Aut}(X)$.

We observe here that the problem AUTO itself is one among a class of problems, each of which we will show is in $\mathrm{FP}^{\mathrm{SPP}}$ by giving such an algorithm for the following *generic* problem FIND-GROUP which we formally describe below:

Let $\mathcal{G}_n$ denote the set of all subgroups of S_n, for each n. Let $\mathcal{G}$ denote the union $\bigcup \mathcal{G}_n$. The FIND-GROUP problem is defined by a function

$$f : \Sigma^* \times 0^* \longrightarrow \mathcal{G},$$

where to each pair $\langle x, 0^n \rangle$ in the domain, the image $f(\langle x, 0^n \rangle)$ is a subgroup of S_n. When the function f is fixed and n is given, it is more convenient notation to denote $f(\langle x, 0^n \rangle)$ by G_x.

Furthermore, for each subgroup $f(\langle x, 0^n \rangle)$ we assume that we have an efficient membership test. More precisely, we assume that we have access to a procedure $\mathrm{MEMB}(x, g)$, that takes x and $g \in S_n$ as input, and evaluates to **true** if and only if $g \in G_x$ *in time polynomial in* n *and* $|x|$. The FIND-GROUP problem is to compute a strong generator set for G_x given $\langle x, 0^n \rangle$ as input.

The problem FIND-GROUP is generic in the sense that for different functions f we get different problems. For instance, in the case of AUTO, for each n vertex graph X, encoded as $x \in \Sigma^*$, we can define $f(\langle X, 0^n \rangle) = \mathrm{Aut}(X)$ and for $m \neq n$ we can define $f(\langle X, 0^m \rangle)$ as the trivial subgroup $\mathbf{1}$ of S_m. The function $\mathrm{MEMB}(x, g)$ is polynomial-time computable as checking whether $g \in S_n$ is in $\mathrm{Aut}(X)$ can be done in time polynomial in n.

Remark. An $\mathrm{FP}^{\mathrm{SPP}}$ algorithm for FIND-GROUP allows us to show at one stroke that, apart from GI, several other permutation group problems are in SPP. In particular, we show in the next section that the hidden subgroup problem for permutation groups has an $\mathrm{FP}^{\mathrm{SPP}}$ algorithm.

Theorem 13 *There is an* $\mathrm{FP}^{\mathrm{SPP}}$ *algorithm for the* FIND-GROUP *problem.*

PROOF. Let $\langle x, 0^n \rangle$ be an input instance of FIND-GROUP. The goal is to compute a strong generator set for $G_x \leq S_n$ using MEMB as subroutine. As we have fixed the input, we will sometimes drop the subscript and write G instead of the group G_x.

Our goal is to design an $\mathrm{FP}^{\mathrm{SPP}}$ algorithm for finding the coset representatives of $G^{(i)}$ in $G^{(i-1)}$ for each i in the tower of subgroups $\mathbf{1} = G^{(n-1)} \leq G^{(n-2)} \leq$

$\ldots \leq G^{(1)} \leq G^{(0)} = G$. Starting with $G^{(n-1)}$, which is trivial, the algorithm will build a strong generator set for $G^{(i)}$ in decreasing order of i until finally it computes a strong generator set for $G^{(0)} = G$. Thus, it suffices to describe how the algorithm will compute the coset representatives of $G^{(i)}$ in $G^{(i-1)}$ assuming that a strong generator set for $G^{(i)}$ is already computed.

We first introduce a definition and notation. A *partial permutation* on the set $[n]$ is an injective function $\pi : I \longrightarrow [n]$, where the domain I of π is a subset of $[n]$. Thus, π is any function that can be extended to a permutation in S_n. We say that a partial permutation φ *extends* π if the domain of φ contains I and $i^{\varphi} = i^{\pi}$ for all $i \in I$. Let $\pi : I \longrightarrow [n]$ be a partial permutation and let $i \in [n] \setminus I$. We denote by $\pi[i \mapsto j]$ the unique partial permutation that extends π to the domain $I \cup \{i\}$ by mapping i to j.

For a subgroup $H \leq S_n$ and $g \in S_n$ let lex−least(Hg) denote the lexicographically least permutation in the coset Hg. We next define a language in NP to which our main algorithm will make UP-like queries:

$$L = \{\langle x, 0^n, S, i, j, \pi\rangle \mid S \subseteq G_x^{(i)}, \pi \text{ is a partial permutation that fixes each of } 1, \ldots, i-1 \text{ and } i^{\pi} = j, \text{ and there is a } g \in G_x^{(i-1)} \text{ such that } g \text{ extends } \pi \text{ and } g = \text{lex−least}(\langle S\rangle g)\}.$$

Partial permutation π is part of instance $\langle x, 0^n, S, i, j, \pi\rangle$, as we will use L as an oracle to do a prefix search for the lexicographically least $g \in G_x^{(i-1)}$ such that $i^g = j$. We now describe an NP machine N that accepts L.

Description of Machine N ;
Input: $\langle x, 0^n, S, i, j, \pi\rangle$;
Verify using MEMB that $S \subseteq G_x^{(i)}$;
  Guess $g \in S_n$;
    if $g \in G_x^{(i-1)}$ and $i^g = j$ and g extends π and $g = \text{lex−least}(\langle S\rangle g)$
    then ACCEPT
    else REJECT;

Clearly, N is an NP machine that accepts L. The crucial point is that if $i^g = j$ then for every element $h \in \langle S\rangle g$, $i^h = j$. Also, using the algorithm in Theorem 10 the lexicographically least element of $\langle S\rangle g$ can be computed in polynomial time.

Claim 14 *If $\langle S\rangle = G^{(i)}$ then the number of accepting paths of N on input $\langle x, 0^n, S, i, j, \pi\rangle$ is either 0 or 1. In general, on input $\langle x, 0^n, S, i, j, \pi\rangle$, N has either 0 or $\frac{|G^{(i)}|}{|\langle S\rangle|}$ accepting paths.*

Proof of Claim. Suppose $\langle x, 0^n, S, i, j, \pi\rangle \in L$ and $\langle S\rangle = G^{(i)}$. Notice that

if for some $g \in G^{(i-1)}$ we have $i^g = j$ (for $j > i$), then $\langle S \rangle g$ consists of all elements in $G^{(i-1)}$ that map i to j. Thus the unique guess in S_n made by N that leads to acceptance is the lexicographically least element of $\langle S \rangle g$. On the other hand, if $\langle S \rangle$ is a proper subgroup of $G^{(i)}$ then $G^{(i)}g$ can be written as a disjoint union of $|G^{(i)}|/|\langle S \rangle|$ many right cosets of $\langle S \rangle$. Thus, in general N would have $|G^{(i)}|/|\langle S \rangle|$ many accepting paths if $\langle x, 0^n, S, i, j, \pi \rangle$ is in L.

We are now ready to describe an FP^L algorithm for FIND-GROUP. The algorithm is designed so it queries L for a $\langle x, 0^n, S, i, j, \pi \rangle$ *only if* $\langle S \rangle = G^{(i)}$, thereby ensuring that it makes only UP-like queries to L. Finally, by Lemma 8 we can convert this algorithm to an $\mathrm{FP}^{\mathrm{SPP}}$ algorithm.

(* FP^L algorithm CONSTRUCT($\langle x, 0^n \rangle$ *);
$C_i := \emptyset$ for every $0 \leq i \leq n-2$;
(* C_i will finally be a complete set of coset representatives of $G^{(i+1)}$ in $G^{(i)}$ *).
$D_i := \emptyset$ for every $0 \leq i \leq n-2$;
$D_{n-1} = 1$;
(* D_i will finally be a strong generator set for $G^{(i)}$ for each i. *)
  for $i := n-1$ **downto** 1
    (* D_i is already computed at the beginning of the i^{th} iteration
    and at the end of the i^{th} iteration we have D_{i-1} *)
    Let $\pi : [i-1] \to [n]$ be the partial permutation
    that fixes all elements from 1 to $i-1$;
    (* in case $i = 1$ this is the everywhere undefined partial permutation *)
      for $j := i+1$ **to** n
      $\pi' := \pi[i \mapsto j]$;
      (* π' extends π to $[i]$ by mapping i to j *)
      if $\langle x, 0^n, D_i, i, j, \pi' \rangle \in L$ **then**
      (* There is an element in $G^{(i-1)}$ that maps i to j. We will find
      it by a prefix search that extends the partial permutation π' *)
      for $k := i+1$ **to** n
        find the element ℓ not in the range of π' such that
        $\langle x, 0^n, D_i, i, j, \pi'[k \mapsto \ell] \rangle \in L$;
        $\pi' := \pi'[k \mapsto \ell]$;
        (* At this point π' will be a permutation in S_n *)
      endfor
      $C_{i-1} := C_{i-1} \cup \{\pi'\}$;
      endif
    endfor
    (* At this point C_{i-1} is a complete set of coset
    representatives of $G^{(i)}$ in $G^{(i-1)}$ *)
  $D_{i-1} = D_i \cup C_{i-1}$;
Output D_0

We claim that a call to the FP^L algorithm CONSTRUCT($\langle x, 0^n \rangle$) outputs a strong generator set D_0 for the group $G = G_x$. We show this by induction. Initially, $D_{n-1} = 1$ clearly generates $G^{(n-1)} = \mathbf{1}$. Suppose at the beginning of the i^{th} iteration it holds that D_i is a strong generator set for $G^{(i)}$. It suffices to show that at the end of the i^{th} iteration $D_{i-1} = D_i \cup C_{i-1}$ is a strong generator set for $G^{(i-1)}$. For each $j : i+1 \leq j \leq n$, the query $\langle x, 0^n, D_i, i, j, \pi' \rangle \in L$ checks if there is an element in $G^{(i-1)}$ that maps i to j. The subsequent prefix search with queries to L computes the lexicographically least element in $G^{(i-1)}$ that maps i to j. Furthermore, by Claim 14, as D_i generates $G^{(i)}$, all queries made to L are UP-like. Thus, at the end of the i^{th} iteration C_{i-1} is a complete set of coset representatives for $G^{(i)}$ in $G^{(i-1)}$ and hence D_{i-1} is a strong generator set for $G^{(i-1)}$. Thus at the end D_0 is a a strong generator set for G. Therefore, we have an FP^L algorithm problem for FIND-GROUP.

Finally, since the FP^L algorithm makes only UP-like queries to the NP oracle L, it follows from Lemma 8 that FIND-GROUP has an $\mathrm{FP}^{\mathrm{SPP}}$ algorithm. □

Remark. Let UPSV denote the class of functions $f : \Sigma^* \longrightarrow \Sigma^*$ for which there is a nondeterministic polynomial-time transducer M that on each input x has a *unique* accepting path on which it outputs $f(x)$. We note that using UPSV there is alternative description of our $\mathrm{FP}^{\mathrm{SPP}}$ algorithm for FIND-GROUP: we can first design an $\mathrm{UPSV}^{\mathrm{SPP}}$ algorithm, where the prefix search that we do in CONSTRUCT($\langle x, 0^n \rangle$) is replaced by directly guessing a permutation in the right coset (consisting of elements that fix 1 to $i-1$ and map i to j) and rejecting along all paths on which we do not guess the lexicographically least element of the coset. Then, by a general prefix search argument we can see that $\mathrm{FP}^{\mathrm{SPP}}$ and $\mathrm{UPSV}^{\mathrm{SPP}}$ are the same and hence conclude that FIND-GROUP is in $\mathrm{FP}^{\mathrm{SPP}}$.

As we already noted, GI and AUTO are polynomial-time equivalent and AUTO, being an instance of FIND-GROUP has an $\mathrm{FP}^{\mathrm{SPP}}$ algorithm by Theorem 13. Since $\mathrm{SPP}^{\mathrm{SPP}} = \mathrm{SPP}$ and $\mathrm{SPP} \subseteq \mathrm{Mod}_k\mathrm{P}$ for each $k \geq 2$, the next corollary is an immediate consequence.

Corollary 15 *Graph Isomorphism is in* SPP *and hence in* $\mathrm{Mod}_k\mathrm{P}$ *for every* $k \geq 2$.

5 Hidden subgroup problem

We recall the general definition of the hidden subgroup problem.

Definition 16 *The* hidden subgroup problem HSP *has an input instance a*

finite group G (presented by a finite generator set) and we are given (in the form of an oracle) a function f from G to some finite set X such that f is constant and distinct on different right cosets of a hidden subgroup H of G. The problem is to determine a generator set for H.

Many natural problems like Graph Isomorphism, Integer Factoring etc, can be cast as a special case of HSP. An efficient quantum algorithm for the general problem will result in efficient quantum algorithm for all these. Based on suitable generalizations of Shor's technique [31], the above problem has efficient quantum algorithms for the case when G is an abelian group (see e.g. [27] for an exposition). However, the status of HSP is open for general nonabelian groups, except for some special cases where it is settled (see, e.g. [17,19]). In particular, even when we restrict attention to G being the permutation group S_n, it is not known if HSP has quantum polynomial time algorithms except in special cases.

Independently, it is shown by Fortnow and Rogers [15] that the class BQP of languages that have polynomial-time quantum algorithms is closely connected with language classes that are low for PP. In particular, it is shown in [15] that BQP $\subseteq$ AWPP where AWPP is a language class that generalizes both BPP and LWPP.

Theorem 17 [15] BQP $\subseteq$ AWPP *and hence* BQP *is low for* PP.

In this section we show as a corollary to Theorem 13 that there is an $\mathrm{FP}^{\mathrm{SPP}}$ algorithm for the HSP problem over permutation groups.

Theorem 18 *There is an* $\mathrm{FP}^{\mathrm{SPP}}$ *algorithm for the* HSP *problem over permutation groups, and hence* HSP *over permutation groups is low for* PP, GapP, $\oplus$P, $\mathrm{C}_{=}\mathrm{P}$ *etc.*

PROOF. We are given (in the form of an oracle) a function f from S_n to a finite set X such that f is constant and distinct on different right cosets of a hidden subgroup H of S_n. The $\mathrm{FP}^{\mathrm{SPP}}$ will first compute $f(1)$ with one query to f. Now, notice that f gives a membership test for the unknown subgroup H, because a permutation $g \in S_n$ is in H if and only if $f(g) = f(1)$. Thus we essentially have a membership test as required for the FIND-GROUP problem of Theorem 13. The result now follows by invoking the algorithm described in the proof of Theorem 13. Lowness for PP also follows as SPP is low for PP. $\square$

Using the $\mathrm{FP}^{\mathrm{SPP}}$ algorithm for the FIND-GROUP problem we can show that other algorithmic problems on permutation groups [25] which are not known to

have polynomial-time algorithms are also in SPP. Among the different problems mentioned in [25] we pick the following two examples as most other problems are known to be polynomial time reducible to these.

The input instance to the CONJ-GROUP problem consists of three subgroups $\langle S \rangle = G$, $\langle S_1 \rangle = H_1$, and $\langle S_2 \rangle = H_2$ of S_n, and the problem is to determine if there is a $g \in G$ such that $gH_1g^{-1} = H_2$ (i.e. H_1 and H_2 are G-conjugate).

A closely related problem NORM has input instance two subgroups G and H of S_n, and the problem is to determine a generator set for the normalizer subgroup $N_G(H) = \{g \in G \mid gHg^{-1} = H\}$. Just as GI and AUTO are polynomial-time equivalent, it turns out that CONJ-GROUP and NORM are also polynomial-time equivalent [25].

Theorem 19 *The problem* NORM *is in* $\mathrm{FP}^{\mathrm{SPP}}$ *and* CONJ-GROUP *is in* SPP.

PROOF. We show that NORM is an instance of FIND-GROUP. The theorem will follow as a direct consequence of Theorem 13. It suffices to observe that given subgroups $\langle S \rangle = G$ and $\langle T \rangle = H$ of S_n, testing if $g \in N_G(H)$ (i.e. $gHg^{-1} = H$) can be carried out in polynomial time. More precisely, it is clear that $gHg^{-1} = H$ if and only if $gtg^{-1} \in H$ for every $t \in T$, which can be checked in polynomial time by Theorem 9. □

As already mentioned, a consequence of the above theorem is that several other decision problems in permutation groups (e.g. coset intersection, double coset equality, set transporter) which are polynomial-time many-one reducible to CONJ-GROUP are also in SPP.

6 Parallel queries to NP

In this section we discuss an application of our main theorem Theorem 13 to a different problem concerning Graph Isomorphism.

We first recall the definitions of two important function classes. Let $\mathrm{FP}_{||}^{\mathrm{NP}}$ denote the class of functions computable in polynomial time with *parallel* queries to an NP oracle. Likewise, let $\mathrm{FP}^{\mathrm{NP}}[\log]$ denote the class of functions computable in polynomial time with *logarithmically many* adaptive queries to an NP oracle. In contrast to the decision problem setting where $\mathrm{P}_{||}^{\mathrm{NP}} = \mathrm{P}^{\mathrm{NP}}[\log]$, it is believed to be unlikely that $\mathrm{FP}_{||}^{\mathrm{NP}} = \mathrm{FP}^{\mathrm{NP}}[\log]$. Indeed, it is

shown in [11,30,9] that $\mathrm{FP}_{||}^{\mathrm{NP}} = \mathrm{FP}^{\mathrm{NP}}[\log]$ implies NP = RP. It is useful to recall the proof of this result. Let SAT denote the set of satisfiable Boolean formulas. The key idea in the proof is that given a boolean formula F with a unique satisfying assignment, the satisfying assignment can be computed in $\mathrm{FP}_{||}^{\mathrm{NP}}$. Thus, if $\mathrm{FP}_{||}^{\mathrm{NP}} = \mathrm{FP}^{\mathrm{NP}}[\log]$, we can find a satisfying assignment of F in polynomial time by enumerating the polynomially many candidates (given by all possible answers to the logarithmically many queries) and testing. Since SAT is randomly many-one reducible to USAT (the set of boolean formulas with unique satisfying assignment), the collapse result $\mathrm{FP}_{||}^{\mathrm{NP}} = \mathrm{FP}^{\mathrm{NP}}[\log]$ implies NP = RP follows.

A question that has remained open is whether we can derive the collapse NP = P from the same assumption. The paper by Jenner and Torán [20] contains a detailed investigation of this question.

In general, we could ask which problems in NP are in P as a consequence of the assumption $\mathrm{FP}_{||}^{\mathrm{NP}} = \mathrm{FP}^{\mathrm{NP}}[\log]$. Consider a language $L \in \mathrm{NP}$ defined by a set A in P and a polynomial bound p as follows:

$$x \in L \iff \exists y \in \Sigma^* \ : \ |y| \leq p(|x|) \text{ and } \langle x, y \rangle \in A.$$

Given an $x \in L$ the problem of computing a witness $y \in \Sigma^*$ such that $|y| \leq p(|x|)$ and $\langle x, y \rangle \in A$ is the *search problem* corresponding to L. Of course, the search problem depends on the set A. Suppose L has the property that this search problem can be solved in $\mathrm{FP}_{||}^{\mathrm{NP}}$. Then, analogous to the discussion above regarding SAT, it is easy to see that $\mathrm{FP}_{||}^{\mathrm{NP}} = \mathrm{FP}^{\mathrm{NP}}[\log]$ implies that L is in P. Using the $\mathrm{FP}^{\mathrm{NP}}[\log]$ machine for the search problem, in polynomial time we can simply enumerate the entire set of polynomially many candidate witnesses and check if there is a y among them such that $\langle x, y \rangle \in A$. Thus we have the following.

Proposition 20 *Suppose $L \in \mathrm{NP}$ has a corresponding search problem that can be solved in $\mathrm{FP}_{||}^{\mathrm{NP}}$. Then $\mathrm{FP}_{||}^{\mathrm{NP}} = \mathrm{FP}^{\mathrm{NP}}[\log]$ implies that L is in P.*

A natural example for such a language L is the Graph Automorphism problem GA as shown in [24]. Thus, $\mathrm{FP}_{||}^{\mathrm{NP}} = \mathrm{FP}^{\mathrm{NP}}[\log]$ implies GA is in P [24].

For Graph Isomorphism, however, it is open if the search problem can be solved in $\mathrm{FP}_{||}^{\mathrm{NP}}$. Thus the above proposition is not applicable. Nevertheless, we will show that if $\mathrm{FP}_{||}^{\mathrm{NP}} = \mathrm{FP}^{\mathrm{NP}}[\log]$ then GI is in P as a consequence of Theorem 13 and another general proposition similar to Proposition 20.

We recall the definition of promise problems.

Definition 21 [11] *A promise problem is a pair of sets (Q,R). A set L is called a solution of the promise problem (Q,R) if for all* $x \in Q$, $x \in L \Leftrightarrow x \in R$.

A promise problem of particular interest is $(1\mathrm{SAT}, \mathrm{SAT})$, where 1SAT contains precisely those Boolean formulas which have at most one satisfying assignment. Observe that any solution of the promise problem $(1\mathrm{SAT}, \mathrm{SAT})$ has to agree with SAT in the formulas having a unique satisfying assignment as well as in the unsatisfiable formulas. By the results of Selman [30,11] we know that $\mathrm{FP}_{||}^{\mathrm{NP}} = \mathrm{FP}^{\mathrm{NP}}[\log]$ implies that every solution to the promise problem $(1\mathrm{SAT}, \mathrm{SAT})$ is in P.

Proposition 22 *Suppose* $L \in \mathrm{NP}$ *is accepted by a deterministic polynomial-time oracle machine M with access an* NP *oracle A such that M makes* only *UP-like queries to A. Then* $\mathrm{FP}_{||}^{\mathrm{NP}} = \mathrm{FP}^{\mathrm{NP}}[\log]$ *implies that L is in* P.

PROOF. Recall that an oracle query q to A is UP-like if the NP machine for A has at most one accepting path on input q. Since all oracle queries made by the machine M to NP oracle A are UP-like, we can replace the oracle with any solution to the promise problem $(1\mathrm{SAT}, \mathrm{SAT})$: let f denote the standard parsimonious reduction from A to SAT. Then, each query q to A is transformed to a SAT query $f(q)$ which will be correctly answered by any solution to $(1\mathrm{SAT}, \mathrm{SAT})$. But the promise problem $(1\mathrm{SAT}, \mathrm{SAT})$ is in P by the assumption $\mathrm{FP}_{||}^{\mathrm{NP}} = \mathrm{FP}^{\mathrm{NP}}[\log]$. Thus, it follows that L is also in P. □

We can now easily derive our claimed result for Graph Isomorphism, and HSP for permutation groups.

Theorem 23 $\mathrm{FP}_{||}^{\mathrm{NP}} = \mathrm{FP}^{\mathrm{NP}}[\log]$ *implies that the* FIND-GROUP *problem for permutation groups can be solved in polynomial time. Hence, it follows that* $\mathrm{FP}_{||}^{\mathrm{NP}} = \mathrm{FP}^{\mathrm{NP}}[\log]$ *implies Graph Isomorphism is in* P *and it implies that the hidden subgroup problem for permutation groups is in* P.

PROOF. As a consequence of Theorem 13 it follows that FIND-GROUP has a polynomial-time oracle algorithm that makes only UP-like queries to an NP oracle A. Thus, by Proposition 22 it follows that $\mathrm{FP}_{||}^{\mathrm{NP}} = \mathrm{FP}^{\mathrm{NP}}[\log]$ implies FIND-GROUP can be solved in polynomial time. Consequently, Graph Isomorphism and the hidden subgroup problem for permutation groups are in P under the assumption $\mathrm{FP}_{||}^{\mathrm{NP}} = \mathrm{FP}^{\mathrm{NP}}[\log]$. □

7 Concluding remarks

In this paper we have shown that Graph Isomorphism is in SPP. We have also shown that several other problems on permutation groups are in SPP. All these results are byproducts of the $\mathrm{FP}^{\mathrm{SPP}}$ algorithm for the problem FIND-GROUP. We would like to know if better upper bounds can be shown for the complexity of special cases of graph isomorphism especially tournament isomorphism. Specifically, is tournament isomorphism in UP? It is known that the automorphisms of a tournaments forms a solvable group and has odd order. Can this additional property be somehow exploited?

A related problem is Graph Canonization. Let f be a function from the family of finite graphs, $\mathcal{G}$, to itself. We say that f is a *canonization* if for every $X \in \mathcal{G}$, $f(X) \cong X$ and for every $X_1, X_2 \in \mathcal{G}$, $f(X_1) = f(X_2)$ iff $X_1 \cong X_2$. There is an $O(n^{log n})$ algorithm for Tournament Isomorphism by giving a canonization procedure for tournaments [4]. The complexity of Graph Canonization is intriguing. The only known upper bound for the problem is $\mathrm{FP}^{\mathrm{NP}}$. It is known that Graph Isomorphism is polynomial-time reducible to Graph Canonization. Is the converse true, at least for tournaments? Is Graph Canonization for tournaments low for PP?

In order to study the complexity of group-theoretic problems in a general setting, Babai and others in [6,5,3], have developed a theory of black-box groups. The main results in [6,5,3] were to put several natural problems in $\mathrm{NP} \cap \mathrm{coAM}$ or $\mathrm{AM} \cap \mathrm{coAM}$. However, lowness for PP has been addressed only for the case of *solvable* black-box groups in [1,34], where many of these problems are shown to be in SPP. It is interesting to ask if our approach of showing membership in SPP via finding the lexicographically least element in a coset can be generalized to black-box groups. More precisely, what is the complexity of finding a canonical element in the right coset of a black-box group?

Acknowledgment. We are grateful to the referees for useful remarks and suggestions that have helped improve the presentation.

References

[1] V. Arvind and N. V. Vinodchandran. Solvable black-box group problems are low for PP. *Theoretical Computer Science*, 180(1–2):17–45, 1997.

[2] L. Babai. Trading group theory for randomness. In *17th Annual Symposium on Foundations of Computer Science*, pages 421–429. ACM, 1985.

[3] L. Babai. Bounded round interactive proofs in finite groups. *SIAM journal of Discrete Mathematics*, 5(1):88–111, February 1992.

[4] L. Babai and E. M. Luks. Canonical labeling of graphs. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, pages 171–183, 1983.

[5] L. Babai and S. Moran. Arthur-Merlin games: a randomized proof system, and a hierarchy of complexity classes. *Journal of Computer and System Sciences*, 36:254–276, 1988.

[6] L. Babai and E. Szemeredi. On the complexity of matrix group problems I. In *Proceedings of the* 24^{th} *IEEE Foundations of Computer Science*, pages 229–240, 1984.

[7] J. L. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity I*, volume 11 of *ETACS monographs on theoretical computer science*. Springer-Verlag, Berlin, 1988.

[8] J. L. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity II*, volume 22 of *ETACS monographs on theoretical computer science*. Springer-Verlag, Berlin, 1990.

[9] R. Beigel. NP-hard sets are P-superterse unless R = NP, January 04 1995.

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

[11] S. Even, A. L. Selman, and Y. Yacobi. The complexity of promise problems with applications to public-key cryptography. *Information and Control*, 61(2):159–173, May 1984.

[12] S. A. Fenner. PP-lowness and a simple definition of AWPP. *Theory of Computing Systems*, 36(2):199–212, 2003.

[13] S. A. Fenner, L. J. Fortnow, and S. A. Kurtz. Gap-definable counting classes. In *Structure in Complexity Theory Conference*, pages 30–42, 1991.

[14] S.A. Fenner, L.J. Fortnow, S. A. Kurtz, and L. Li. An oracle builder's toolkit. In *SCT: Annual Conference on Structure in Complexity Theory*, pages 120–131, 1993.

[15] L. J. Fortnow and J. D. Rogers. Complexity limitations on quantum computation. In *IEEE Conference on Computational Complexity*, pages 202–209, 1998.

[16] M. L. Furst, J. E. Hopcroft, and E. M. Luks. Polynomial-time algorithms for permutation groups. In *IEEE Symposium on Foundations of Computer Science*, pages 36–41, 1980.

[17] S. Hallgren, A. Russel, and A. Ta-Shma. Normal subgroup reconstruction and quantum computing using group representation. In *Proceedings of the* 32^{nd} *ACM Symposium on Theory of Computing*, pages 627–635, Portland, Oregon, 21-23 May 2000.

[18] C. M. Hoffmann. *Group-Theoretic Algorithms and Graph Isomorphism.* Springer, Berlin, Heidelberg, 1982.

[19] G. Ivanyos, F. Magniez, and M. Santha. Efficient quantum algorithms for some instances of the non-abelian hidden subgroup problem. In 13^{th} *ACM Symposium on Parallel Algorithms and Architectures*, pages 263–270, 2001.

[20] B. Jenner and J. Torán. Computing functions with parallel queries to NP. *Theoretical Computer Science*, 141(1–2):175–193, 1995.

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

[22] J. Köbler, U. Schöning, and J. Torán. Graph isomorphism is low for PP. *Computational Complexity*, 2(4):301–330, 1992.

[23] J. Köbler, U. Schöning, and J. Torán. *The Graph Isomorphism Problem: Its Structural Complexity.* Birkhauser, 1993.

[24] A. Lozano and J. Torán. *On the Non-Uniform Complexity of the Graph Isomorphism Problem.* Cambridge University Press, 1992.

[25] E. M. Luks. Permutation groups and polynomial time computations. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 11:139–175, 1993.

[26] R. Mathon. A note on graph isomorphism counting problem. *Information Processing Letters*, 8(3):131–132, 15 March 1979.

[27] M Mosca. *Quantum Computer algorithms.* PhD thesis, Oxford University, 1999.

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

[29] U. Schöning. Graph isomorphism is in the low hierarchy. In *Symposium on Theoretical Aspects of Computer Science*, pages 114–124, 1987.

[30] A. L. Selman. A taxonomy of complexity classes of functions. *Journal of Computer and System Sciences*, 48(2):357–381, 1994.

[31] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.

[32] C. C. Sims. Computational methods in the study of permutation groups. *Computational problems in Abstract Algebra*, pages 169–183, 1970.

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

[34] N. V. Vinodchandran. Counting complexity of solvable black-box group problems. *SIAM Journal of Computing*, 33(4):852–869, 2004.

[35] H. Wielandt. *Finite Permutation Groups.* Academic Press, New York, 1964.