On Finding the Number of Graph Automorphisms

Robert Beals*
IAS and DIMACS

Richard Chang†
University of Maryland Baltimore County

William Gasarch‡
University of Maryland College Park

Jacobo Torán§
University of Ulm

March 5, 1996

Abstract

In computational complexity theory, a function f is called $b(n)$-enumerable if there exists a polynomial-time function which can restrict the output of $f(x)$ to one of $b(n)$ possible values. This paper investigates #GA, the function which computes the number of automorphisms of an undirected graph, and GI, the set of pairs of isomorphic graphs. The results in this paper show the following connections between the enumerability of #GA and the computational complexity of GI.

1. #GA is $\exp(O(\sqrt{n \log n}))$-enumerable.
2. If #GA is polynomially enumerable then GI $\in$ R.
3. For $\epsilon < \frac{1}{2}$, if #GA is n^{ϵ}-enumerable then GI $\in$ P.

*Institute For Advanced Study, Princeton, New Jersey, 08540, and DIMACS, Rutgers University, Piscataway, NJ, 08855-1179. (email: beals@math.ias.edu). Supported by an NSF Mathematical Sciences Postdoctoral Fellowship and by the Alfred P. Sloan foundation.

†Dept. of Computer Science and Electrical Engineering, University of Maryland Baltimore County, Baltimore, MD 21228, USA (email: chang@cs.umbc.edu). Supported in part by National Science Foundation grant CCR-9309137 and by the University of Maryland Institute for Advanced Computer Studies.

‡University of Maryland Institute for Advanced Computer Studies and Dept. of Computer Science, University of Maryland College Park, College Park, MD 20742, USA (email: gasarch@cs.umd.edu). Supported in part by National Science Foundation grant CCR-9301339.

§University of Ulm, Theoretische Informatik, D-89069 Ulm, Germany. (email: toran@informatik.uni-ulm.de,). The work of this author was done while he was at Dept. Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, Pau Gargallo 5, E-08028 Barcelona, Spain. Supported in part by the European Community through the Espirit BRA Program (project 7141, ALCOM II).

1 Introduction

The Graph Isomorphism problem has a special place in computational complexity theory. The set GI consists of all pairs of graphs that are isomorphic to each other. GI is known to be in NP but not NP-complete unless the Polynomial Hierarchy collapses [13, 26], a condition which violates the usual intractability assumptions. Nevertheless, there is no known polynomial-time algorithm to solve the isomorphism problem on general graphs even though some progress has been made towards polynomial-time algorithms in special cases (most notably for planar graphs [15, 16] and bounded degree graphs [21]). Thus, the Graph Isomorphism problem belongs to a short list of problems in NP that are suspected to be neither decidable in polynomial time nor NP-complete. In fact, the exact complexity of GI remains an open problem. For example, it is not known whether GI can be solved in randomized polynomial time or whether GI is contained in the class NP $\cap$ co-NP.

The current state of knowledge on the complexity of GI depends on the two-round interactive protocol for Graph Non-Isomorphism [13], a technique which we exploit in this paper. However, even before this proof was discovered, it was suspected that GI could not be NP-complete because counting the number of graph isomorphisms has roughly the same complexity as deciding the existence of an isomorphism [22]. In contrast, the counting versions of typical NP-complete problems tend to be much harder than the decision versions. The proof that counting graph isomorphisms is relatively "easy" also demonstrated a close connection between the structure of graph isomorphisms and graph automorphisms (isomorphisms between a graph and itself). The results in this paper add another link to this connection.

In computational complexity theory, a function is called $b(n)$-enumerable[1] if a polynomial-time function can determine a restricted range for the function. For example, *a priori* the #GA function, which computes the number of automorphisms in a graph, may output any value from 1 to $n!$ for a graph with n vertices. However, we will show that #GA can take on only one of $\exp(O(\sqrt{n \log n}))$ values — i.e., we will show that #GA is $\exp(O(\sqrt{n \log n}))$-enumerable. The other main results in this paper show that if #GA is "easy" in the sense of enumerability then there is a corresponding decrease in the complexity of GI. Namely:

- If #GA is polynomially enumerable then GI can be recognized in randomized polynomial time.
- For $\epsilon < \frac{1}{2}$, if #GA is n^{ϵ}-enumerable then GI can be recognized in deterministic polynomial time.

Currently, GI does not seem to be solvable in polynomial time using either randomized or deterministic computations. Hence, these results could also be interpreted as results on the non-enumerability of #GA.

The rest of the paper is organized as follows. In Section 2, we provide some technical background and formal definitions for the terms used in this paper. In Section 3, we construct a graph gadget that allows us to combine many instances of GI into one instance of #GA.

[1] not to be confused with recursive enumerability in recursive function theory or countable (denumerable) sets in set theory.

In Sections 4 and 5, we present the results connecting the enumerability of #GA and the complexity of GI. Finally, in Section 6 we give an upper bound on the enumerability of #GA.

2 Preliminaries

In this paper, we will work with many complexity classes. We assume that the reader is familiar with P and NP, the classes of languages recognized by deterministic and nondeterministic polynomial-time Turing machines. We will use PH to denote the Polynomial Hierarchy and R to denote the class of languages recognized by probabilistic polynomial-time Turing machines with bounded one-sided error. We refer the reader to standard textbooks [3, 4] in complexity theory for explanations on the relationships among these classes.

An instance of the Graph Isomorphism problem (GI) is a pair of undirected graphs (G, H). Without loss of generality, the vertices of the graphs are labelled 1 through n. The pair (G, H) is an element of GI if there exists a bijection f from the vertices of G to the vertices of H that preserves the edge relations — i.e., (u, v) is an edge in the graph G if and only if $(f(u), f(v))$ is an edge in H. In this case, f is called an isomorphism between G and H and we write $G \simeq H$. Note that we may think of f as a permutation of the set $\{1, \ldots, n\}$. Whereas GI is a set, or alternatively a decision problem, #GI is a function, or a counting problem. The value of the function #GI on input (G, H) is the number of isomorphisms from G to H.

An instance of the Graph Automorphism problem (GA) is a single graph G. The graph G is an element of GA if G has a non-trivial automorphism — i.e., an isomorphism between G and itself other than the identity function. Analogously, the function #GA computes the number of automorphisms on G. It is often more convenient to work with GA instead of GI, because the set of automorphisms of a graph forms a group under composition.

Clearly, the set GI is an element of NP because an NP machine can guess a permutation and check that the permutation is indeed an isomorphism between two graphs. As we have mentioned before, GI is known to be incomplete for NP unless the Polynomial Hierarchy collapses. The complexities of #GI, GA and #GA can be estimated based upon their relationship to GI. For example, GA reduces to GI by a many-one polynomial-time reduction. Therefore, GA is also an element of NP and cannot be complete for NP unless PH collapses. Clearly, GI reduces to #GI because knowing the number of isomorphisms certainly tells you whether one exists. In addition, one can compare the complexities of these problems as oracles. Using the group structure of GA, one can show that $\mathrm{P}^{\mathrm{GI}} = \mathrm{P}^{\mathrm{GA}} = \mathrm{P}^{\#\mathrm{GI}}$ [22], [18, Theorem 1.24]. Thus, treated as oracles for P, the problems GI, GA and #GI have essentially the same complexity.[2]

The incompleteness of GI also shows that $\mathrm{P}^{\#\mathrm{GI}}$ cannot contain any NP-complete problems unless PH collapses. This result sets the Graph Isomorphism problem apart from the NP-complete problems. For example, consider the satisfiability problem SAT and the corresponding counting problem #SAT, which outputs the number of satisfying assignments of a Boolean formula. SAT is of course NP-complete, so $\mathrm{P}^{\mathrm{SAT}} = \mathrm{P}^{\mathrm{NP}}$. However, it also known that $\mathrm{P}^{\#\mathrm{SAT}}$ contains the entire Polynomial Hierarchy [27]. Thus, the complexity of #SAT is

[2]Of course, $\mathrm{P}^{\mathrm{GA}} \subseteq \mathrm{P}^{\#\mathrm{GA}}$, but whether $\mathrm{P}^{\#\mathrm{GA}} \subseteq \mathrm{P}^{\mathrm{GA}}$ remains an open problem.

much higher than the complexity SAT, whereas the complexity of #GI is at the same level as that of GI.

Returning to graph automorphisms, we note that the value of $\#\mathrm{GA}(G)$ has several special properties. First, $\#\mathrm{GA}(G)$ must range from 1 to $n!$ because the identity function is always an automorphism and there are at most $n!$ permutations of the n vertices. Second, the set of automorphisms of G forms a subgroup of S_n, the set of all permutations of $\{1, \ldots, n\}$ under composition. This group structure can be exploited in many ways. For example, from LaGrange's Theorem, we know that $\#\mathrm{GA}(G)$ must divide $n!$, hence $\#\mathrm{GA}(G)$ cannot have factors larger than n. Thus, given $\#\mathrm{GA}(G)$ as input, it is possible to obtain a complete prime factorization of $\#\mathrm{GA}(G)$ in polynomial time. The following observations about #GA and prime numbers will be needed throughout the paper.

Lemma 2.1 *Let G be a graph with n vertices. For $i \geq 1$, let m_i be the ith smallest prime number larger than n.*

1. *$\#\mathrm{GA}(G)$ divides $n!$.*
2. *m_i does not divide $\#\mathrm{GA}(G)$.*
3. *There exists a prime p s.t. $m_i < p < 2m_i$.*
4. *For $n \geq 17$, $m_i \leq 2(n \log n + i \log i)$.*
5. *m_i can be computed in time $n^{O(1)} + i^{O(1)}$.*

Proof: Parts 1 and 2 follow from the preceding discussion. Part 3 is just Bertrand's Postulate [14] (that there exists a prime number between x and $2x$). Part 4 can be derived easily from a result of Rosser and Schoenfeld [24] which states that the number of primes less than x is between $x/\ln x$ and $1.25506x/\ln x$, for $x \geq 17$. (These are estimates for the constants in the Prime Number Theorem.) Part 5 follows from Part 4, because m_i is polynomial in n and i. Since we can list all the primes below a number x in time polynomial in x (not the length of x), m_i can be found in time polynomial in n and i. ∎

Thus, #GA cannot take on every value between 1 and $n!$ since some of these numbers cannot be the order of a subgroup of S_n. This leads us to "enumerability" as a measure of complexity. The concept of enumerability in computational complexity theory was introduced independently by Beigel [5] and by Cai and Hemachandra [8] then later modified by Amir, Beigel and Gasarch [1].

Definition 2.2 Let $b : \mathbb{N} \to \mathbb{N}$ be polynomially bounded. A function f is *$b(n)$-enumerable* if there exists a polynomial-time computable function g, such that for all x, $g(x)$ outputs a list of at most $b(|x|)$ values, one of which is $f(x)$. A function f is *poly-enumerable* if f is $b(n)$-enumerable for some polynomial b.

For super-polynomial $b : \mathbb{N} \to \mathbb{N}$, f is *$b(n)$-enumerable* if there exists a polynomial-time computable function g, such that for all x, $f(x)$ is one of the $b(|x|)$ values in the sequence $g(x, 0), g(x, 1), \ldots, g(x, b(|x|) - 1)$. (Here we assume that the second input to g is written in binary.)

Intuitively, we think of the enumerability of functions as a generalization of approximability. For example, suppose a function $f : \mathbb{N} \to \mathbb{N}$ is approximable within a factor of 2. Then, there is a polynomial-time computable function which, for all x, outputs a value y and guarantees that $y \leq f(x) \leq 2y$. Thus, the set of possible values of $f(x)$ is restricted to the numbers between y and $2y$. For enumerability, the set of possible values does not have to be an interval. Another difference between enumerability and approximability is that in approximability the number of possible values is "output sensitive" — i.e., the number of possible values of $f(x)$ depends directly on $f(x)$ rather than x. In addition, approximability is only meaningful when there is a natural total ordering on the range of the function whereas enumerability makes sense in a broader setting.

There do exist functions which cannot be polynomially enumerated unless some intractibility assumptions are violated. For example, Cai and Hemachandra showed that unless P = PP, the function #SAT is not n^ϵ-enumerable for $\epsilon < 1$.[3] This result was improved independently by Cai and Hemachandra [9] and by Amir, Beigel and Gasarch [1], who showed that P = PP if and only if #SAT is $p(n)$-enumerable for some polynomial p. Moreover, Amir, Beigel and Gasarch [1] proved that unless the Polynomial Hierarchy collapses to its fourth level, #SAT is not 2^{n^ϵ}-enumerable for $\epsilon < 1$. Since #SAT is clearly 2^n-enumerable, these results show tight upper and lower bounds on the enumerability of #SAT assuming that PH does not collapse.

In the present paper, we investigate the enumerability of #GA. Our motivation for studying the enumerability of #GA is twofold. First, the results mentioned above, combined with Toda's theorem that every set in PH reduces to #SAT [27], show that #GA cannot be #P-complete unless PH collapses to P^{NP} (actually P^{GI}). Therefore, the enumerability properties of #GA might be very different from those of #SAT. Also, connections between the enumerability of #GA and the complexity of GI might help us obtain a better classification of the Graph Isomorphism problem.

Throughout the paper, we use the number of vertices in a graph as the measure of the size of the input. We do this to simplify the terminology even though the length of the encoding of a graph could be as long as n^2. In certain cases, this convention does have an effect on our results. For example, Theorem 4.5 is stated for $\epsilon < 1/2$; without this convention, the statement would be $\epsilon < 1/4$.

3 Combining Lemma

In this section we show how to combine many instances of GI into one instance of #GA. This lemma will be used in the proofs of the main theorems of Sections 4 and 5. First, we need to define the notion of reductions between two functions (as opposed to sets).

Definition 3.1 (Krentel [19]) Let f and g be two functions. We say that f reduces to g, written $f \leq_{\mathrm{m}}^{\mathrm{p}} g$, if there exist two polynomial-time computable functions S and T such that

$$f(x) = S(x, g(T(x))).$$

[3] PP is the class of languages recognized by probabilistic polynomial-time Turing machines with unbounded two-sided error. Since PP contains NP, the conclusion P = PP is generally considered to be unlikely.

Intuitively, $f \leq_{\mathrm{m}}^{\mathrm{p}} g$ implies that f is easier than g, because g provides enough information for a polynomial-time function to compute f. These reductions are also called *metric reductions* in the literature.

To simplify our notation, we will also use the following notational device for generalized characteristic functions. For a set A and an ordered list $x_1, \ldots, x_r$ of instances of A, the function $\chi^A(x_1, \ldots, x_r)$ outputs a sequence of r bits such that the ith bit is 1 if and only if $x_i \in A$. Note that r does not have to be constant here.

Now, we are ready to prove the Combining Lemma. This key lemma allows us to construct a graph $\mathcal{F}$ from q instances of the Graph Isomorphism problem, $(G_1, H_1), \ldots, (G_q, H_q)$, such that $\#\mathrm{GA}(\mathcal{F})$ provides enough information to determine in polynomial time whether G_i is isomorphic to H_i for each instance (G_i, H_i).

Lemma 3.2 (Combining Lemma) *There exist polynomial-time functions T and S such that $\chi^{\mathrm{GI}} \leq_{\mathrm{m}}^{\mathrm{p}} \#\mathrm{GA}$ via T and S. Furthermore, in the case where the ordered list $\mathcal{Q} = \langle (G_1, H_1), \ldots, (G_q, H_q) \rangle$ consists of pairs of graphs with n vertices, the following hold for the graph $\mathcal{F}$ output by $T(\mathcal{Q})$.*

1. *The $\mathcal{F}$ has $O(n^2 q \log n + n q^2 \log q)$ vertices.*
2. *The output of $S(\mathcal{Q}, \#\mathrm{GA}(\mathcal{F}))$ can be computed from $(n, q, \#\mathrm{GA}(\mathcal{F}))$.*

Proof: In the construction below, the running time of T will be polynomial in $|\mathcal{Q}|$ which is polynomial in $n + q$. This allows for the possibility that $\mathcal{Q}$ has lots of small graphs. In the first step of the construction, we find $m_1, \ldots, m_q$, the q prime numbers immediately following the number n. Let $r_i = (m_i + 1)/2$ (w.l.o.g. r_i is an integer). For each i, $1 \leq i \leq q$, we construct a graph C_i as follows. Take r_i copies of G_i, r_i copies of H_i and a complete graph with $2r_i$ new vertices $a_1, \ldots, a_{2r_i}$. Connect each vertex in the jth copy of G_i to a_j. Connect each vertex in the jth copy of H_i to a_{r_i+j}. (See Figure 1.) Call the resulting graph C_i.

Now, suppose that G_i is isomorphic to H_i. Then every automorphism of C_i can be formed by a permutation of the vertices in $\{a_1, \ldots, a_{2r_i}\}$ followed by an automorphism of each copy of G_i and H_i. Hence

$$\#\mathrm{GA}(C_i) = (2r_i)!(\#\mathrm{GA}(G_i))^{2r_i}.$$

Since $2r_i = m_i + 1$, the prime number m_i divides $\#\mathrm{GA}(C_i)$.

On the other hand, if G_i is *not* isomorphic to H_i then every automorphism of C_i can be formed by a permutation of the vertices in $\{a_1, \ldots, a_{r_i}\}$, followed by a permutation of the vertices in $\{a_{r_i+1}, \ldots, a_{2r_i}\}$ and the automorphisms of each copy of G_i and H_i. In this case,

$$\#\mathrm{GA}(C_i) = (r_i)!(r_i)!(\#\mathrm{GA}(G_i))^{r_i}(\#\mathrm{GA}(H_i))^{r_i}.$$

Since $r_i < m_i$ and $|G_i| = |H_i| = n < m_i$, the prime number m_i does not divide $\#\mathrm{GA}(C_i)$. In summary,

$$G_i \simeq H_i \iff m_i \text{ divides } \#\mathrm{GA}(C_i). \tag{1}$$

Let $\mathcal{F}$ be the disjoint union of all the C_i, for $1 \leq i \leq q$. The output of the function $T(\mathcal{Q})$ will be $\mathcal{F}$. Since each C_i has $(m_i + 1)(n + 1)$ vertices, the total number of vertices in $\mathcal{F}$ is

$$(n+1)\sum_{i=1}^{q}(m_i + 1) < q(n+1)(m_q + 1).$$

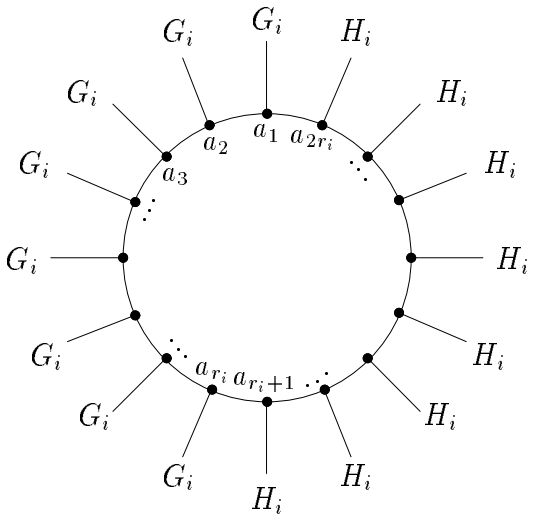

Figure 1: Combining r_i copies of G_i and H_i.

By Lemma 2.1 part 4, we know that $m_q \leq 2(n \log n + q \log q)$. Hence, we can bound the number of vertices in $\mathcal{F}$ by $O(n^2 q \log n + nq^2 \log q)$.

Now, we show how $\#\mathrm{GA}(\mathcal{F})$ can be used to compute $\chi^{\mathrm{GI}}(\mathcal{Q})$. By the construction of $\mathcal{F}$,

$$\#\mathrm{GA}(\mathcal{F}) = \prod_{i=1}^{q} \#\mathrm{GA}(C_i). \tag{2}$$

We also know that for all i, $1 \leq i \leq q$

$$G_i \simeq H_i \Rightarrow m_i \text{ divides } \#\mathrm{GA}(\mathcal{F}).$$

However, the converse may not hold because $\#\mathrm{GA}(C_j)$ for some $j > i$ may contain m_i as a factor. Thus, to determine whether $G_i \simeq H_i$ from $\#\mathrm{GA}(\mathcal{F})$, these extraneous m_i factors (if they exist) must be removed. To do this, we start with the last C_i, namely C_q. Since $m_j < m_q$ for all $j < q$, we know that

$$G_q \simeq H_q \iff m_q \text{ divides } \#\mathrm{GA}(\mathcal{F}).$$

Now, if $G_q \simeq H_q$ then C_q contributes an $(m_q + 1)!$ factor to $\#\mathrm{GA}(\mathcal{F})$; otherwise, C_q contributes an $(r_i)!(r_i)!$ factor to $\#\mathrm{GA}(\mathcal{F})$.

To determine whether $G_{q-1} \simeq H_{q-1}$ we need to remove the prime factors $> n$ from $\#\mathrm{GA}(\mathcal{F})$ that came from $\#\mathrm{GA}(C_q)$. Since $\#\mathrm{GA}(G_q)$ and $\#\mathrm{GA}(H_q)$ do not contain prime factors $> n$, the prime factors $> n$ in $\#\mathrm{GA}(C_q)$ come from either $(m_q + 1)!$ or $(r_q)!(r_q)!$

depending on whether $G_q \simeq H_q$ (which we have already determined). So, let

$$N_i = \begin{cases} \#\mathrm{GA}(\mathcal{F}) & \text{if } i = q \\ \dfrac{N_{i+1}}{(m_{i+1}+1)!} & \text{if } m_{i+1} \text{ divides } N_{i+1} \\ \dfrac{N_{i+1}}{(r_{i+1})!(r_{i+1})!} & \text{otherwise} \end{cases}$$

From the preceding discussion, it is clear that $G_i \simeq H_i \iff m_i$ divides N_i. Thus, the function S can compute $\chi^{\mathrm{GI}}((G_1, H_1), \ldots, (G_q, H_q))$ from $\#\mathrm{GA}(\mathcal{F})$, n and q, by finding $m_1, m_2, \ldots, m_q$ and calculating $N_1, N_2, \ldots N_q$. ∎

4 #GA and n^ϵ-enumerability

The main theorem in this section shows that it is unlikely for #GA to be n^ϵ-enumerable because for any $\epsilon < 1/2$, #GA is n^ϵ-enumerable if and only if GI can be recognized in polynomial time. We begin with a review of two constructions from the literature. The first one shows that the Graph Isomorphism problem is "self-computable," in the sense that given GI as an oracle, we can construct an isomorphism between two isomorphic graphs in polynomial time [18, 25]. We reproduce the proof of this well-known theorem because we need to make references to the construction in the proof and because we need to estimate the sizes of the graphs queried.

Lemma 4.1 *There exists a polynomial-time Turing machine using* GI *as an oracle which finds an isomorphism between two graphs, if the graphs are isomorphic.*

Proof: We prove that GI is "self-computable" by constructing a mapping between the vertices of two isomorphic graphs G and H using GI as an oracle. This "self-computable" property is similar to the self-reducibility of SAT. In the first stage of the construction, we find a vertex i_1 in H such that there is an isomorphism between G and H mapping vertex 1 in G to vertex i_1 in H. This is accomplished by trying all n vertices in H exhaustively and asking the GI oracle the n questions:

> Is there an isomorphism from G to H mapping vertex 1 to vertex i_1?

These questions can be transformed into queries to GI by attaching cliques with $n+1$ vertices to vertex 1 in G and vertex i_1 in H. Thus, any isomorphism between the transformed graphs must map vertex 1 in G to vertex i_1 in H. If such an isomorphism exists, the remaining $n-1$ stages of the construction assign vertices $2, \ldots, n$ in G to the vertices in H under the restriction that vertex 1 maps to vertex i_1. In stage k of the construction, vertex k in G and i_k in H will be attached to a clique with $n+k$ vertices. ∎

Remark: It is convenient to think of the procedure described in the preceding proof as a *self-reduction tree*. The root of the tree, level 0, is labelled with the graphs (G, H). Each vertex at level k has $n - k$ children which represent the $n - k$ possible assignments of vertex k in G to the $n - k$ remaining vertices in H. These vertices are labelled with the corresponding transformed graphs. This tree has $n!$ leaves, so we cannot construct the entire tree in polynomial time. However, at the leaves of the tree, every vertex of G is assigned to some vertex of H. Thus, in polynomial time we can determine whether the mapping represented by a leaf is indeed an isomorphism between G and H.

In the proof of Theorem 4.5 below, our strategy is to traverse the self-reduction tree from the top down. Since the tree has exponentially many paths, we will need to identify some of the paths as dead-ends. The following combinatorial lemmas [7, 23] will help us prune the tree and maintain a polynomial bound on the running time of the tree traversal.

Definition 4.2 For a collection $\mathcal{C}$ of sets and a set X, we say X *separates* C if for all $S, S' \in \mathcal{C}$, $S \neq S' \Rightarrow S \cap X \neq S' \cap X$.

Lemma 4.3 *For a collection $\mathcal{C}$ of sets, with $|\mathcal{C}| = n \geq 1$, there exists a set X that separates $\mathcal{C}$ where $|X| \leq n - 1$.*

The lemma below adapts Lemma 4.3 to show that if we have ℓ vectors in $\{0,1\}^\ell$, then the vectors can be uniquely identified by their values at $\ell - 1$ coordinates. Thus, one of the coordinates is not needed to distinguish the vectors from each other. In the following, we use $(\vec{b})_i$ to denote the ith component of a vector $\vec{b} \in \Sigma^\ell$.

Lemma 4.4 *Let $m \leq \ell$ and $\vec{b}_1, \ldots, \vec{b}_m \in \{0,1\}^\ell$. There exists a coordinate k such that for all $\vec{b}_i \neq \vec{b}_j$, there exists $t \neq k$ such that $(\vec{b}_i)_t \neq (\vec{b}_j)_t$. Moreover, k can be found in time polynomial in ℓ.*

Proof: It suffices to prove the case where $m = \ell$. Use Lemma 4.3 where $\mathcal{C}$ is the collection of subsets of $\{1, \ldots, \ell\}$ represented by the bit vectors $\vec{b}_1, \ldots, \vec{b}_\ell$. Let k be an element not contained in the separator X. Since X is a separator, each pair of bit vectors must differ at coordinates other than k. The coordinate k can be found in time polynomial in ℓ because we can simply try all possible values for k and check each pair of $\vec{b}_i$ and $\vec{b}_j$. This takes time $O(\ell^4)$. ∎

We are now ready to prove the main result of this section. The techniques used in this proof and in Lemma 4.4 are derived from results on enumerability and self-reducibility by Amir, Beigel and Gasarch [1].

Theorem 4.5 *For $\epsilon < 1/2$, the function* #GA *is n^ϵ-enumerable if and only if* GI $\in$ P.[4]

[4]Recall that the n in "n^ϵ-enumerable" is the number of vertices in the graph. If n is the length of the encoding of the graph, we would need to further restrict $\epsilon < 1/4$.

Proof: If GI $\in$ P, then #GA is also computable in polynomial time using group theoretic arguments [18]. In this case, #GA would be 1-enumerable. Thus, we only need to show that if #GA is n^ϵ-enumerable, then GI $\in$ P.

Given two graphs G and H with n vertices, we search the self-reduction tree described above in stages. We maintain a list $\mathcal{Q}$ of pairs of graphs from the self-reduction tree. Initially, $\mathcal{Q}$ contains just the pair (G, H). Throughout the tree-pruning procedure we maintain the invariant that $G \simeq H$ if and only if $\mathcal{Q}$ contains a pair of isomorphic graphs (i.e., $\chi^{\mathrm{GI}}(\mathcal{Q})$ is not all zeroes). Also, the size of the list $\mathcal{Q}$ will always be polynomially bounded. In the beginning of every stage of the tree pruning, we take each pair of graphs in $\mathcal{Q}$ and replace it with its children in the self-reduction tree. We continue the replacement until $\mathcal{Q}$ has at least $q(n)$ pairs (for $q(n) \geq n$ to be determined below).

Let $\mathcal{Q}' = \langle (G_1, H_1), \ldots, (G_{q(n)}, H_{q(n)}) \rangle$ be the first $q(n)$ pairs in $\mathcal{Q}$. Let m be an upper bound on the size of these graphs. We apply the Combining Lemma to construct the graph $\mathcal{F}$ which has at most $r = mq(n)^2 \log q(n)$ vertices. Then, we use the enumerator for #GA on $\mathcal{F}$ to obtain a list of r^ϵ numbers one of which is #GA($\mathcal{F}$). The function S in the Combining Lemma converts these numbers into a list of r^ϵ vectors $\vec{b}_1, \ldots, \vec{b}_{r^\epsilon}$ in $\{0,1\}^{q(n)}$, one of which is $\chi^{\mathrm{GI}}(\mathcal{Q}')$. Now, suppose that $\vec{b}_i \neq 0^{q(n)}$ for all i, $1 \leq i \leq r^\epsilon$. Then, we know that $\chi^{\mathrm{GI}}(\mathcal{Q}') \neq 0^{q(n)}$, so G must be isomorphic to H. Thus, we can halt the pruning procedure and accept. In the remaining case, we may assume that $0^{q(n)}$ is one of the vectors in $\vec{b}_1, \ldots, \vec{b}_{r^\epsilon}$. We will pick $q(n)$ below so that $r^\epsilon \leq q(n)$. Then, Lemma 4.4 gives us a coordinate k such that for $\vec{b}_i \neq \vec{b}_j$, the vectors $\vec{b}_i$ and $\vec{b}_j$ differ on a coordinate other than k. Now, it cannot be the case that (G_k, H_k) is the only isomorphic pair of graphs in $\mathcal{Q}'$, because in that case $\chi^{\mathrm{GI}}(\mathcal{Q}') = 0^{k-1}10^{q(n)-k}$, hence $\chi^{\mathrm{GI}}(\mathcal{Q}')$ can only be distinguished from $0^{q(n)}$ using the kth coordinate. Thus, the pruning process can safely remove the pair (G_k, H_k) from the list $\mathcal{Q}$ and still guarantee that if $\mathcal{Q}$ contains an isomorphic pair before pruning, then it also does after pruning.

We continue removing items from $\mathcal{Q}$ until it has fewer than $q(n)$ pairs of graphs. Then we proceed to the next stage. After at most n stages, the pairs in $\mathcal{Q}$ are leaves of the self-reduction tree, so we can compute $\chi^{\mathrm{GI}}(\mathcal{Q})$ in polynomial time. By the invariant we have maintained, $G \simeq H$ if and only if $\chi^{\mathrm{GI}}(\mathcal{Q})$ is not all zeroes. Thus, we have shown that GI $\in$ P.

Finally, we need to show that by picking $q(n)$ to be n^α where $\alpha > 1/(1-2\epsilon)$, we can guarantee that $r^\epsilon \leq q(n)$. (The constant α is positive since $\epsilon < 1/2$.) From the construction of the self-reduction tree in Lemma 4.1, we know that m is $O(n^2)$ since the graphs G_i and H_i consist of n original vertices and cliques of size $n+1$ through $2n$. So,

$$r^\epsilon \leq (cn^2 \cdot q(n)^2 \cdot \log q(n))^\epsilon = (cn^2 \cdot n^{2\alpha} \cdot \alpha \log n)^\epsilon.$$

Thus, $r^\epsilon < n^{2\epsilon+2\alpha\epsilon+\delta}$ for all $\delta > 0$. From our choice of α, we know that

$$2\epsilon + 2\alpha\epsilon < 1 + 2\alpha\epsilon < \alpha.$$

Therefore, $r^\epsilon \leq q(n)$. ∎

Since it is generally believed that GI $\notin$ P, the preceding theorem can also be interpreted as a lower bound on the enumerability of #GA. We can also use the theorem to obtain a lower bound on the bounded query complexity of #GA. The bounded query classes are defined as follows.

Definition 4.6 Let $j(n)$ be a function and X be a set. A function f is in $\mathrm{PF}^X_{j(n)\text{-T}}$ if there exists a polynomial-time oracle Turing machine which computes f using no more than $j(n)$ queries to X on inputs of length n.

Counting the number of oracle queries has been established as a useful complexity measure. For example, the number of queries to an NP oracle can be used to characterize the complexity of approximating NP-optimization problems [11, 12]. The following fact shows that there is an intimate connection between the enumerability of a function and the bounded query complexity of that function.

Fact 4.7 (Beigel [6, Lemma 3.2]) *Let f be any function and $j(n)$ be a polynomial-time computable function. The following are equivalent:*

1. *There exists X such that $f \in \mathrm{PF}^X_{j(n)\text{-T}}$.*
2. *f is $2^{j(n)}$-enumerable.*

Using Fact 4.7 we can obtain a lower bound on the number of queries needed to compute #GA, assuming that GI $\notin$ P.

Corollary 4.8 *Let $\epsilon < 1/2$. If there exists an X such that* #GA $\in \mathrm{PF}^X_{\epsilon \log n\text{-T}}$ *then* GI $\in$ P.

5 #GA and poly-enumerability

Assuming that GI $\notin$ P, the main result in the previous section is a "non-enumerability" result. In general, we would like to prove stronger non-enumerability results for #GA. For example, Amir, Beigel and Gasarch [1] were able to prove that #SAT is not 2^{n^ϵ}-enumerable for $\epsilon < 1$ unless the Polynomial Hierarchy collapses. We cannot use their machinery for the case of #GA, because it turns out that #GA is actually $\exp(O(\sqrt{n \log n}))$-enumerable. Instead, we adapt the techniques of Goldwasser, Micali and Rackoff [13] to show that #GA cannot be poly-enumerable unless GI $\in$ R.[5]

Goldwasser, Micali and Rackoff showed that Graph Non-Isomorphism, the complement of GI, can be recognized by a two-round interactive protocol. We briefly review this protocol in order to motivate the proof of Theorem 5.2. There are two parties involved in the interactive protocol: an all-powerful prover and a randomized polynomial-time verifier. The verifier asks the prover to convince him that the input graphs (G, H) are *not* isomorphic as follows. In secret, the verifier randomly picks one of G and H along with a random permutation π. He applies π to either G or H, whichever one he picked, and obtains a new graph X. Then, the verifier asks the prover whether X is a permuted version of G or of H. If G and H are not isomorphic, the all-powerful prover simply checks if $G \simeq X$ or if $H \simeq X$, and provides the appropriate answer. On the other hand, if G and H are isomorphic, X can be a permuted version of either graph. In that case, the prover can only provide the correct answer with probability one half. Thus, if $G \not\simeq H$, there exists a prover who can always convince the

[5] We were motivated in part by Lozano and Torán [20, Theorem 5.1] who also used this technique in their proof.

verifier to accept. Conversely, if $G \simeq H$, no prover (even one that "lies") can convince the verifier to accept with greater than 50 percent probability.

In the proof below, our randomized algorithm for GI will play the role of the verifier and the enumerator for #GA will play the role of the prover. However, unlike the prover in an interactive protocol, the enumerator provides several answers at once.[6] Thus, it is possible for the enumerator to give two answers at the same time: one that corresponds to $G \simeq H$ and one to $G \not\simeq H$. We cope with this situation by emulating the interactive protocol many times in parallel. Thus, instead of picking one permutation π and forming one permuted graph X, we pick $q(n)$ permutations $\pi_1, \ldots \pi_{q(n)}$ and form $q(n)$ graphs $X_1, \ldots, X_{q(n)}$ each of which is a permutation of either G or H.

Notation 5.1 For each permutation $\pi \in S_n$, we use $\pi(G)$ to denote the graph obtained by re-labelling the vertices of G using π. Given two permutations $\pi, \rho \in S_n$, we define $\pi \circ \rho \in S_n$ to be the functional composition of π and ρ — i.e., $(\pi \circ \rho)(G) = \pi(\rho(G))$.

Theorem 5.2 *If* #GA *is poly-enumerable then* GI $\in$ R.

Proof: Assuming that #GA is $p(n)$-enumerable via an enumeration function g, we will construct a randomized polynomial-time algorithm to decide whether the input graphs G and H are isomorphic. Let n be the number of vertices in G and H and let $q(n) \geq n$ be a polynomial to be specified later. As discussed above, we randomly pick $q(n)$ permutations $\pi_1, \ldots, \pi_{q(n)}$ from S_n and for each π_i permute either G or H. Our choice of applying π_i to either G or H can be represented by a single bit. So, a bit vector $\vec{b} \in \{0,1\}^{q(n)}$ and the permutations $\pi_1, \ldots, \pi_{q(n)}$ fully specify our random choices. Let b_i be the ith bit of $\vec{b}$ and X_i be the result of applying the permutation π_i; that is:

$$X_i = \begin{cases} \pi_i(H) & \text{if } b_i = 0; \\ \pi_i(G) & \text{if } b_i = 1. \end{cases}$$

Now, consider the instances of the graph isomorphism problem: $(G, X_1), \ldots, (G, X_{q(n)})$. Note that if G *is not* isomorphic to H then

$$\chi^{\mathrm{GI}}((G, X_1), \ldots, (G, X_{q(n)})) = \vec{b},$$

since G is isomorphic to X_i only when $X_i = \pi_i(G)$. On the other hand, if G *is* isomorphic to H, then our choice of applying π to G or H does not change whether G is isomorphic to X_i. So, in this case,

$$\chi^{\mathrm{GI}}((G, X_1), \ldots, (G, X_{q(n)})) = 1^{q(n)}.$$

Next, we use the Combining Lemma on $(G, X_1), \ldots, (G, X_{q(n)})$ to construct a graph $\mathcal{F}$ such that $\#\mathrm{GA}(\mathcal{F})$ provides enough information to compute $\chi^{\mathrm{GI}}((G, X_1), \ldots, (G, X_{q(n)}))$. If we could compute $\#\mathrm{GA}(\mathcal{F})$ directly, then we can immediately determine whether $G \simeq H$ by checking whether $\#\mathrm{GA}(\mathcal{F})$ corresponds to the case where $\chi^{\mathrm{GI}}((G, X_1), \ldots, (G, X_{q(n)}))$ is

[6] Another difference is that the enumerator cannot "lie" in the same manner as the prover because one of the answers it provides must be the correct value of #GA.

$\vec{b}$ or $1^{q(n)}$.[7] However, we cannot compute $\#\mathrm{GA}(\mathcal{F})$ directly, so we use the enumerator for $\#\mathrm{GA}(\mathcal{F})$ instead. Let T and S be the reduction functions from the Combining Lemma. (We used T on $(G, X_1), \ldots, (G, X_{q(n)})$ to obtain the graph $\mathcal{F}$.) Since we assume that $\#\mathrm{GA}$ is $p(n)$-enumerable, in polynomial time we can compute $g(\mathcal{F})$, a set of polynomially many values one of which is $\#\mathrm{GA}(\mathcal{F})$. For each value in $g(\mathcal{F})$, we use the S function from the Combining Lemma to determine a possible value for $\chi^{\mathrm{GI}}((G, X_1), \ldots, (G, X_{q(n)}))$. Call the set of all such values

$$POSS = \{\ S(n, q(n), N) \mid N \in g(\mathcal{F})\ \}.$$

If $G \not\simeq H$, then $\vec{b} = \chi^{\mathrm{GI}}((G, X_1), \ldots, (G, X_{q(n)}))$. Thus, $\vec{b}$ must be an element of $POSS$, since $\chi^{\mathrm{GI}}((G, X_1), \ldots, (G, X_{q(n)}))$ is always an element of $POSS$. On the other hand, if $G \simeq H$, then $\vec{b} \in POSS$ occurs with very low probability (proven below). Therefore, the strategy for our randomized algorithm is to accept (G, H) if and only if $\vec{b} \notin POSS$. The following summarizes the algorithm to determine whether $G \simeq H$.

1. Randomly pick $\vec{b} \in \{0,1\}^{q(n)}$ and $\pi_1, \ldots, \pi_{q(n)} \in S_n$.
2. Use the Combining Lemma to construct $\mathcal{F} = T((G, X_1), \ldots, (G, X_{q(n)}))$.
3. Use g to generate the possible values of $\#\mathrm{GA}(\mathcal{F})$.
4. Compute the set $POSS = \{S(n, q(n), N) \mid N \in g(\mathcal{F})\}$.
5. If $\vec{b} \notin POSS$ then output YES, otherwise output NO.

If $G \not\simeq H$, then the algorithm above outputs NO with probability 1. It remains to be proven that if $G \simeq H$ then the algorithm above outputs YES with high probability. Intuitively, it is unlikely for $\vec{b} \in POSS$ when $G \simeq H$ because $POSS$ is completely determined by $\mathcal{F}$ and the same $\mathcal{F}$ can be the result of exponentially many random choices. We prove this formally by partitioning the set of all random choices of the algorithm into blocks of $2^{q(n)}$ random choices. Each random choice within a block has a distinct $\vec{b}$ but produces the same graph $\mathcal{F}$. Thus, within each block, the probability that $\vec{b} \in POSS$ is very low. The blocks are defined as follows.

Assume that G is isomorphic to H. Since permutations are invertible, for every graph Y isomorphic to G, there exists a permutation ρ such that $\rho(H) = Y$. Now, fix a sequence of permutations $\sigma_1, \ldots, \sigma_{q(n)} \in S_n$ and let $Y_i = \sigma_i(G)$. Let $\rho_1, \ldots, \rho_{q(n)}$ be the corresponding permutations such that $\rho_i(H) = Y_i$. Let $\vec{b}$ be any bit vector chosen by our randomized algorithm. Then, there exists a choice of $\pi_1, \ldots, \pi_{q(n)}$ such that the graph X_i constructed in the algorithm is exactly Y_i, namely:

$$\pi_i = \begin{cases} \rho_i & \text{if } b_i = 0; \\ \sigma_i & \text{if } b_i = 1. \end{cases}$$

Furthermore, if we fix an isomorphism τ from H to G, the permutation π_i is completely determined by b_i and σ_i — i.e., $\pi_i = \sigma_i \circ \tau$ if $b_i = 0$ and $\pi_i = \sigma_i$ if $b_i = 1$. Thus, we

[7] Except for the pathological case where we chose $\vec{b} = 1^{q(n)}$, but this occurs with low probability.

can associate the permutations $\sigma_1, \ldots, \sigma_{q(n)}$ with a block of $2^{q(n)}$ random choices for the randomized algorithm (since there are $2^{q(n)}$ different bit vectors $\vec{b}$). To see that every choice of $\vec{b}$ and $\pi_1, \ldots, \pi_{q(n)}$ corresponds to some block, simply note that we can set $\sigma_i = \pi_i \circ \tau^{-1}$ if $b_i = 0$ and $\sigma_i = \pi_i$ if $b_i = 1$. This will again guarantee that $Y_i = X_i$ for every i. Therefore, the blocks do form a partition of the set of all random choices made by the algorithm.

Finally, observe that for each of the $2^{q(n)}$ distinct bit vectors $\vec{b}$ in a block, the same instances of GI, $(G, X_1), \ldots, (G, X_{q(n)})$, are constructed by the randomized algorithm. Thus, the same set *POSS* is generated. Since *POSS* has $p(r(n))$ elements and since $p(r(n))$ is polynomially bounded, the probability that a randomly chosen $\vec{b}$ is not an element of *POSS* is at least $1 - p(r(n))/2^{q(n)}$. Thus, for $q(n)$ large enough, the randomized algorithm will accept with high probability in the case that $G \simeq H$. ∎

As before, we can translate the non-enumerability of #GA into lower bounds on its bounded query complexity using Fact 4.7.

Corollary 5.3 *If there exists an X such that* $\#\mathrm{GA} \in \mathrm{PF}^X_{O(\log n)\text{-T}}$ *then* $\mathrm{GI} \in \mathrm{R}$.

6 Subexponential enumeration

The results of the preceding section can be interpreted as lower bounds on the enumerability of #GA since it seems unlikely that GI ∈ P or GI ∈ R. In this section we provide an upper bound on the enumerability of #GA by showing that #GA is $\exp(O(\sqrt{n \log n}))$-enumerable. Our enumerator will be oblivious, that is, $g(A, i)$ will only depend on n and i. We think of i as being an encoding of the order of a permutation group A of degree n. We must show that such an encoding exists such that i takes space $O(\sqrt{n \log n})$ and the function $i \mapsto |A|$ is computable in polynomial time.

Lemma 6.1 *Let A be a subgroup of S_n. Let p_i be the ith prime. Then $|A|$ must be of the form $\prod_{i=1}^{m} p_i^{d_i}$ where*

1. *for all i, $d_i \leq n$.*
2. *$m \leq n/\ln n + o(n/\ln n)$.*

Proof: Let m and $d_1, \ldots, d_m$ be such that $|A| = \prod_{i=1}^{m} p_i^{d_i}$. Since $|A|$ is a divisor of $n!$, we know that for all i, $p_i^{d_i}$ divides $n!$. However, for any prime p, the highest power of p dividing $n!$ is p^d, where

$$d = \sum_{j \geq 1} \left\lfloor \frac{n}{p^j} \right\rfloor < n \sum_{j \geq 1} \frac{1}{p^j} = \frac{n}{p-1} \leq n.$$

(This formula simply counts the number of positive integers up to n which are divisible by p, p^2, et cetera.) In particular, all the prime factors of $|A|$ are $\leq n$. The Prime Number Theorem states that

$$\lim_{n \to \infty} \frac{\pi(n)}{n/\ln n} = 1,$$

where $\pi(n)$ is the number of primes less than or equal to n. Thus, $m \leq n/\ln n + o(n/\ln n)$. ∎

Theorem 6.2 #GA *is* $\exp(O(\sqrt{n \log n}))$*-enumerable via an oblivious enumerator.*

Proof: Let G be a graph. The set of automorphisms of G is a subgroup of S_n, hence $\#\mathrm{GA}(G)$ must be of the form specified in Lemma 6.1. We show how to enumerate all possible sizes of the subgroups of S_n. We describe this in the form of a compressed encoding of orders of subgroups of S_n, such that the encoding lengths are $O(\sqrt{n \log n})$ and given the encoding of $|A|$, we can compute $|A|$ in polynomial time. One method would be to simply write down the binary representations of the d_i, for $1 \leq i \leq \pi(n)$. By Lemma 6.1 the number of bits used would be $O(n)$, which is too many. However, we will use this technique for "small" primes.

Let $k = \sqrt{n \log n}$. Then $\pi(k) = O(k/\log n) = O(\sqrt{n/\log n})$. For $A \leq S_n$, the first part of the encoding of A will consist of the binary representations of the d_i for $p_i \leq k$. This uses $O(\pi(k) \log n) = O(\sqrt{n \log n})$ bits, as desired. To encode the exponents of the larger primes, we must use detailed knowledge of the possible orders of subgroups of S_n. However, our task will be greatly simplified by the fact that we can now ignore the small primes.

Let Ω denote $\{1, \ldots, n\}$. For $x \in \Omega$, x^A denotes *the A-orbit of* x, i.e., the set of all images of x under the action of A. We say that an orbit is *trivial* if it has one element, and we say that A is *transitive* if Ω is an orbit. In any case, the A-orbits partition Ω. We describe two divide-and-conquer techniques based on this partition. Let $\Delta = x^A$ be an A-orbit, and let A_x denote the subgroup $\{a \in A \mid x^a = x\}$ of those permutations which fix x. It is well known that

$$|A| = |A_x| \cdot |\Delta|,$$

so if $|\Delta|$ has only prime factors $\leq k$, we may replace A by A_x for the purposes of the second part of the encoding. We henceforth assume that all nontrivial orbits of A have orders divisible by some prime larger than k (so in particular there are at most $\sqrt{n/\log n}$ nontrivial orbits). Actually, we make the more general assumption that for any proper subgroup B of A, the index $|A|/|B|$ is divisible by a prime larger than k.

Now suppose that $|\Delta| = m$, and let B be the subgroup of S_m obtained from A by ignoring the action outside of Δ. Let A_Δ denote the subgroup of A which fixes every point of Δ. Then $|A| = |B| \cdot |A_\Delta|$, and to encode $|A|$ it suffices to first encode $|B|$ and then recursively encode $|A_\Delta|$. The number of orbits Δ which need to be considered is $O(\sqrt{n/\log n})$, so we may use only $O(\log n)$ bits to encode $|B|$. To do this, we make great use of the fact that B is transitive.

Now suppose that $\Delta_1, \ldots, \Delta_r$ is a partition of Δ with $1 < r < m$ such that B permutes the Δ_i. If several such partitions exist, then choose one that minimizes r. Then the partition is called a system of imprimitivity for B, and the Δ_i are called blocks of imprimitivity. Since B acts transitively on the set of blocks, B has a subgroup of index r, from which we conclude that r is divisible by some prime $p > k$. Let N be the subgroup of B that fixes the blocks, i.e. any $x \in N$ sends each Δ_i to itself. Let K be the subgroup of S_r obtained from B by considering the action on the blocks. Then $|B| = |N| \cdot |K|$. In addition, every orbit of N has order less than k, so $|N|$ is a product of primes at most k. Thus, to encode $|B|$, it suffices to encode $|K|$. By minimality of r, K is *primitive*, i.e., preserves no nontrivial partition of the permutation domain.

Much is already known about the structure of primitive groups. The O'Nan–Scott Lemma [10, Theorem 4.1] classifies them into several types. Many of these are ruled out by our

assumption that the index of any proper subgroup of K is divisible by some prime larger than k (where k^2 is bigger than r, the size of the permutation domain). In fact, we are left in the case that K has a simple normal subgroup T. Further, either $|T| = r$ and K is a subgroup of the automorphism group of $T \times T$, or K is a subgroup of the automorphism group of T. Following the Classification of Finite Simple groups, much is known about the permutation representations of finite simple groups [2]. In particular, either T is one of polynomially many alternating or classical groups (each of which can be described by a string of length $O(\log n)$), or $|K|$ is polynomially bounded. In the first case $|K|/|T|$ is polynomially bounded, so in either case $O(\log n)$ bits suffice to describe the order of K.

We summarize the encoding of $|A|$. First, we write down the binary representations of the exponents of the primes in $|A|$ for all primes $p \leq k$. Then, we write down a sequence of $O(n/k)$ pairs $(\langle T \rangle, N)$, where $\langle T \rangle$ is an $O(\log n)$ length name of a group T which is either the trivial group or a classical group with a permutation representation of degree $\leq n$ and N is a positive integer (written in binary) which is at most n^c for some constant c. For primes $p > k$, the p-part of $|A|$ is the product of the p-parts of the N's and the orders of the T's. The order of T is given by an explicit formula, so $|A|$ can be computed in polynomial time from its encoding. The first part of the encoding uses $O(\pi(k) \log n)$ bits; the second part of the encoding uses $O((n/k) \log n)$ bits. Both of these quantities are $O(\sqrt{n \log n})$ as desired. ∎

7 Discussion

Several open problems remain on the enumerability of #GA. We have shown that if #GA is poly-enumerable, then GI $\in$ R. For SAT, we know that if #SAT poly-enumerable, then P = PP [1, 9] which implies that SAT $\in$ P. For #GA, it remains open whether #GA being poly-enumerable could imply that GI $\in$ P. It might even be possible to show that if #GA is 2^{n^ϵ}-enumerable for some $\epsilon < 1/2$, then GI $\in$ P. Such a theorem would not violate our upper bound that #GA is $\exp(O(\sqrt{n \log n}))$-enumerable. Note that an analogous result for #SAT, that #SAT is 2^{n^ϵ}-enumerable implies SAT $\in$ P, is not known.

While no polynomial-time algorithms for GI or #GA have been discovered, algorithms with subexponential running time do exist. For example, there exists an algorithm with time complexity $\exp(O(\sqrt{n \log n}))$ which computes the automorphism group and its generators [2] (this is harder than solving GI and #GA). This algorithm combines the techniques of several authors including Babai, Luks, and Zemlyachenko. The bound on the running time is the same as the upper bound on the enumerability of #GA that we achieved in Section 6. This striking observation brings up the possibility of the following time-enumeration trade-off. By allowing the enumerator to run for longer than polynomial time, say time $\exp(O((n \log n)^a + \log n))$, it might be possible to achieve $\exp(O((n \log n)^b))$-enumerability for $a + b = 1/2$. Note that the case $a = 1/2$ is given by the subexponential-time algorithm mentioned above [2], and the case $a = 0$ is our upper bound. The case $0 < a < 1/2$ is open.

We remark that for oblivious enumeration, our upper bound is tight up to some logarithmic factors in the exponent. To see this, let $k = \sqrt{n \log n}$ as in Section 6. Since $k\pi(k) = O(n)$, the sum of the primes up to k is $O(n)$. By scaling k by a constant factor, we may achieve that $\pi(k) = \Omega(\sqrt{n / \log n})$ and the sum of the primes less than k is $\leq n$. So, for

any subset S of the primes $p \leq k$, there is a permutation group of degree n whose order is the product of the primes in S (the group is generated by disjoint cycles of prime lengths). This gives us $\exp(\Omega(\sqrt{n/\log n}))$ different orders of permutation groups of degree n. It is straightforward to obtain the same lower bound for the number of distinct orders of automorphism groups of graphs of degree n. Thus, oblivious enumeration seems to have reached its limit, but it remains open whether or not a clever use of polynomial-time computable graph properties would yield a better, non-oblivious enumerator.

Finally, we note that the reduction of Section 5 can be used even if the enumeration is not polynomial. (It would yield a randomized algorithm for GI which has super-polynomial running time.) Roughly speaking, the reduction can be used as long as the enumerability is less than about $\exp(n^{1/2-\epsilon})$ (so, not surprisingly, an oblivious enumerator is useless for this). In order to beat the subexponential running time of the existing algorithm [2], the enumerability would have to be about $\exp(n^{1/4-\epsilon})$ (however, the enumerator would not have to be restricted to polynomial time).

Acknowledgements

The authors would like to thank László Babai for helpful comments on the topics of this paper, and Dave Mount for proofreading.

References

[1] A. Amir, R. Beigel, and W. I. Gasarch. Some connections between bounded query classes and non-uniform complexity. In *Proceedings of the 5th Structure in Complexity Theory Conference*, pp. 232–243, July 1990. A much expanded version has been submitted to *Information and Computation* and is available via ftp://ftp.cs.yale.edu/pub/beigel/.

[2] L. Babai, W. M. Kantor, and E. M. Luks. Computational complexity and the classification of finite simple groups. In *Proceedings of the 24th IEEE Symposium Foundations of Computer Science*, pp. 162–171, 1983.

[3] J. L. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity I*, volume 11 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1988.

[4] J. L. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity II*, volume 22 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1990.

[5] Richard Beigel. *Query-Limited Reducibilities*. PhD thesis, Stanford University, 1987. Also available as Report No. STAN-CS-88-1221.

[6] R. Beigel. A structural theorem that depends quantitatively on the complexity of SAT. In *Proceedings of the 2nd Structure in Complexity Theory Conference*, pp. 28–32, June 1987.

[7] R. Beigel. Bounded queries to SAT and the Boolean hierarchy. *Theoretical Computer Science*, 84(2):199–223, July 1991.

[8] J. Cai and L. A. Hemachandra. Enumerative counting is hard. *Information and Computation*, 82(1):34–44, July 1989.

[9] J. Cai and L. A. Hemachandra. A note on enumerative counting. *Information Processing Letters*, 38(4):212–219, 1991.

[10] P. J. Cameron. Finite permutation groups and finite simple groups. *Bulletin of the London Mathematical Society*, 13:1–22, 1981.

[11] R. Chang. On the query complexity of clique size and maximum satisfiability. In *Proceedings of the 9th Structure in Complexity Theory Conference*, pages 31–42, June 1994. To appear in *Journal of Computer and System Sciences*.

[12] R. Chang, W. I. Gasarch, and C. Lund. On bounded queries and approximation. Technical Report TR CS-94-05, Department of Computer Science, University of Maryland Baltimore County, April 1994. To appear in *SIAM Journal on Computing*.

[13] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.

[14] G. Hardy and E. Wright. *An introduction to the theory of numbers*. Clarendon Press, Oxford, 1979. Fifth Edition. The first edition was in 1938.

[15] J. E. Hopcroft and R. E. Tarjan. A V^2 algorithm for determining isomorphism of planar graphs. *Information Processing Letters*, pages 32–34, 1971.

[16] J. E. Hopcroft and J. K. Wong. A linear time algorithm for isomorphism of planar graphs. In *ACM Symposium on Theory of Computing*, pages 172–184, 1974.

[17] R. Karp and R. Lipton. Some connections between nonuniform and uniform complexity classes. In *Proceedings of the 12th ACM Symposium on Theory of Computing*, pp. 302–309, 1980.

[18] J. Köbler, U. Schöning, and J. Torán. *The Graph Isomorphism Problem: Its Structural Complexity*. Progress in Theoretical Computer Science. Birkhauser, Boston, 1993.

[19] M. W. Krentel. The complexity of optimization problems. *Journal of Computer and System Sciences*, 36(3):490–509, 1988.

[20] A. Lozano and J. Torán. On the nonuniform complexity of the Graph Isomorphism problem. *Complexity Theory: Current Research*, pp. 245–273, 1993. Edited by K. Ambos-Spies, S. Homer, and U Schöning. Shorter version in *Proceedings of the 7th Structure in Complexity Theory Conference*, pp. 118–129, June 1992.

[21] E. Luks. Isomorphism of bounded valence can be tested in polynomial time. *Journal of Computer and System Sciences*, 25:42–65, 1982.

[22] R. Mathon. A note on the graph isomorphism counting problem. *Information Processing Letters*, 8:131–132, 1979.

[23] J. C. Owings, Jr. A cardinality version of Beigel's Nonspeedup Theorem. *Journal of Symbolic Logic*, 54(3):761–767, September 1989.

[24] J. B. Rosser and L. Schoenfeld. Approximate formulas for some functions of prime numbers. *Illinois Journal of Mathematics*, 6:64–94, 1962.

[25] C. P. Schnorr. Optimal Algorithms for Self-Reducible Problems. In *Proceedings of the 3rd International Conference on Automata, Language, and Programming* (ICALP), pp. 322-337, 1976.

[26] U. Schöning. Probabilistic complexity classes and lowness. *Journal of Computer and System Sciences*, 39:84–100, December 1989.

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