Permutation groups and the graph isomorphism problem

Sumanta Ghosh and Piyush P Kurur
Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur,
Kanpur, Uttar Pradesh, India 208016
{smghosh,ppk}@cse.iitk.ac.in

February 11, 2014

1 Introduction

One of the core ideas in mathematics is the notion of an isomorphism, i.e. structure preserving bijections between mathematical objects like groups, rings and fields. A natural computational question is to decide, given two such objects as input, whether they are isomorphic or not. In the context of undirected finite graphs, this problem is called the graph isomorphism problem and is the subject matter of this article. Informally, we say that two graphs are isomorphic if they are the same up to a renaming of their vertices, i.e. we have a bijection between the vertex sets that preserve the adjacency relation of edges. Many other isomorphism problems for explicitly presented finite mathematical structures like groups, for example, reduce to the graph isomorphism problem. Also, many problems that arise in practice, like studying the structure of chemical compounds, are essentially graph isomorphism in disguise. Hence, understanding this problem computationally is important.

There are efficient programs and libraries (see NAUTY for instance) that can solve large instances of graph isomorphism that arise in practice. However, there are no known polynomial-time algorithm for the general case. In complexity theory, ever since the notion of NP-completeness has been formalised, the graph isomorphism problem has had an important status as it is believed to be a natural example of a problem of intermediate complexity [12, Chapter 7], i.e. neither in P nor NP-complete: It is know that the graph isomorphism problem is in the complexity class co-AM [5], a randomised version of co-NP, and its NP hardness will result in the collapse of the polynomial hierarchy [5, 24]. Furthermore, Köbler et al. [15] showed that graph isomorphism is low for the counting class PP by showing its membership in LWPP. This was further improved by Arvind and Kurur [1] to SPP. As a result graph isomorphism is in and low for various counting complexity classes like classes like $\oplus$P etc. Thus, under rea-

sonable complexity theoretic assumptions, graph isomorphism is *not* NP-hard. Ladner [18] proved the existence of an infinite hierarchy of problems of intermediate complexity assuming that P is different from NP. The graph isomorphism problem, for reasons stated above, is believed to be a natural example.

In this article, we study graph isomorphism and related problems. There is now a vast literature on graph isomorphism and we really cannot do justice to the topic in such a short article. For a detailed study of graph isomorphism, mainly from a complexity theoretic view point, we refer the reader to the excellent book by Köbler et al. [16]. This paper concentrates on one of the aspects of the graph isomorphism problem, namely its intimate connection to permutation group algorithms. Permutation groups arise in the study of graph isomorphism problem because of its close relation to the graph automorphism problem. For a graph X, the automorphisms, i.e. isomorphisms from X to itself, forms a group under function composition. We can identify this group as a subgroup of the set of all permutations of the vertex set $V(X)$. Automorphisms, thus are symmetries of the graph. The computational problem of computing a generating set of the automorphism group is equivalent to the graph isomorphism problem [22]. Most algorithms for graph isomorphism that make use of permutation group theory makes use of this connection. Understanding the automorphism group of a graph is also in tune with what is now a guiding principle in much of modern mathematics: understanding objects by understanding their symmetries.

2 Preliminaries

We briefly review the group theory required for this article mainly to fix notation and convention. For details please refer any standard book on group theory, for example Marshall Hall [13]. The *trivial group* that contains only the identity is denoted by 1. For a group G, we use the notation $H \leqslant G$ (or $G \geqslant H$) to denote that H is a subgroup of G. The *right coset* of the subgroup H of G associated with an element $g \in G$ is the set $Hg = \{hg | h \in H\}$. The set of all right cosets form a partition of G and any subset T of G that has a unique coset representative from each right coset is called a *right transversal* of H in G. Analogously, we define left cosets and left transversals. In general, the right coset Hg and the left coset gH are different. We say that H is a *normal subgroup* if $gH = Hg$. We use the notation $H \trianglelefteq G$ (or $G \trianglerighteq H$) to denote that H is a normal subgroup of G.

A *simple group* is a group that has no nontrivial normal subgroups. A *composition series* of a group G is a tower of subgroups $G = G_0 \trianglerighteq G_1 \trianglerighteq \ldots \trianglerighteq G_t = 1$ such that each of the factor groups G_i/G_{i+1}, called the *composition factors*, are simple. The Jordan-Hölder theorem states that for any group G, its composition series is essentially unique, i.e. any two composition series are of equal length and the list of composition factors are equal up to a reordering. *Solvable groups* are those whose composition factors are abelian.

The set of all permutation of n elements forms a group called the *symmetric group* which we denote by S_n. In algorithmic settings, it is often useful to make

the domain of n elements explicit: For a finite set Ω, the set $\mathrm{Sym}(\Omega)$ denote the group of permutations on Ω. By a *permutation group* on Ω we mean a subgroup of the symmetric group $\mathrm{Sym}(\Omega)$. As is customary, we use Wielandt's notation [28]: Let α be any element of Ω and let g be a permutation in $\mathrm{Sym}(\Omega)$, the image of α under g is denoted by α^g. The advantage of this notation is that it follows the familiar laws of exponentiation: $(\alpha^g)^h = \alpha^{gh}$. We can extend this notation to (i) subsets of permutations: $\alpha^A = \{\alpha^g | g \in A\}$, or to (ii) subsets of Ω: $\Sigma^g = \{\alpha^g | \alpha \in \Sigma\}$. In particular, for a permutation group on Ω, the set α^G is called the *G-orbit* of α. Given any two elements α and β of Ω the G-orbits α^G and β^G are either disjoint or are the same. Thus, orbits of G partition the underlying set Ω. A subset Σ of G is said to be *G-stable* if $\Sigma^g = \Sigma$. Clearly any G-orbit is G-stable. In general, a G-stable set is a union of G-orbits.

Let G be a permutation group acting on the set Ω and let Σ be a subset of Ω. The *point-wise stabiliser* of Σ is the subgroup of all g in G that is trivial on Σ, i.e. $\alpha^g = \alpha$ for all α in Σ. The *setwise stabiliser* is the subgroup that fixes the set Σ as a whole, i.e it is the subgroup of all g in G such that $\Sigma^g = \Sigma$.

A permutation group G is said to be *transitive* if the entire set Ω is a single orbit. Equivalently, G is transitive if for any two elements α and β in Ω there is a permutation g in G such that $\alpha^g = \beta$. For a transitive permutation G on Ω, a subset Σ is said to be a *G-block* if for any permutation g in G, the set Σ^g is either *identical to* or *disjoint from* the set Σ. Any singleton set is a block and so is the entire set Ω. These blocks are called the *trivial blocks* of G. For a transitive permutation group G on Ω and a permutation g in G, the set Σ^g is a G-block whenever Σ itself is one. Such a block Σ^g is called a *conjugate block* of Σ. The family of conjugate blocks $\{\Sigma^g | g \in G\}$ forms a partition of the set Ω which is called the *block system* associated with the block Σ. A permutation group that has no nontrivial block is called a *primitive permutation group*. An example of a primitive group is the group $\mathrm{Sym}(\Omega)$. We have the following lemma about block systems which is more or less direct from the definition.

Lemma 2.1. *Let G be a transitive permutation group on Ω and let Σ be a block. Let N denote the subgroup of G that setwise stabilises all the elements in the Σ-block system $\mathcal{B}(\Sigma) = \{\Sigma^g | g \in G\}$. Then N is a normal subgroup of G and G/N acts as a permutation on Σ-block system $\mathcal{B}(\Sigma)$. In addition, if Σ is a maximal G-block then this action of the group G/N is primitive.*

In algorithms that deal with permutation groups, we need a succinct way to encode them which we now describe. Any permutation of Ω can be presented by an array of $\#\Omega$ elements and hence can be encoded as a string of size $O(n \lg n)$. A permutation group is presented via a list of permutations that generate the group. It is a well known fact that any group G has a generating set of size less than $\lceil \lg \#G \rceil$ and hence this presentation of permutation group is reasonable. Thus, we assume that the input size, for an algorithm that takes a generating set S of a permutation group G on Ω, is $\#S + \#\Omega$. Similarly, an algorithm that is expected to produce a permutation group as output, should output a generating set of size polynomial in $\#\Omega$. For example, the strong generating set that we describe in the next section, is of size at most $\#\Omega^2$.

By a graph we mean an undirected graph, i.e. a finite set of vertices and an edge set which is a subset of unordered pairs of vertices. We use $V(X)$ and $E(X)$ to denote the set of vertices and the set of edges of a graph X respectively. A bijection f from $V(X)$ to $V(Y)$ is an *isomorphism* if for every two vertices u and v of X, the unordered pair $\{u, v\}$ is an edge of X if and only if $\{f(u), f(v)\}$ is an edge of Y. An *automorphism* of a graph X is an isomorphism from the graph to itself. The set of automorphism of a graph X, denoted by $\mathrm{Aut}\,(X)$, form a group under composition. In fact, $\mathrm{Aut}\,(X)$ is a permutation group on $V(X)$.

In the article, we assume that a graphs of n vertices is encoded as an n^2-bit strings that represent its $n \times n$ adjacency matrix. We now define the *graph isomorphism problem*

Problem 2.2 (Graph isomorphsim problem)**.** *The* graph isomorphism problem *(*GI *for short) is defined as follows: Given two undirected graphs X and Y via their adjacency matrix, decide whether they are isomorphic.*

The counting version of the graph isomorphism problem, denoted by #GI*, is the problem of computing the number of isomorphism between the two input graphs (0 when they are not isomorphic).*

Graph isomorphism problem is closely related to the *automorphism problem* that we define next.

Problem 2.3 (Automorphism problem)**.** *The* automorphism problem *(*AUT *for short) is the problem of computing a strong generating set of the automorphism group* $\mathrm{Aut}\,(X)$ *of an input graph X.*

Mathon [22] proved that the problems GI, #GI and AUT are all polynomial-time Turing reducible to each other. Therefore, in the setting of permutation group algorithms, it is often the automorphism problem that is attacked for solving graph isomorphism.

A graph X is said to be *rigid* if it has no nontrivial automorphism, i.e. if $\mathrm{Aut}\,(X)$ is the trivial group. We now define the graph rigidity problem.

Problem 2.4 (Graph rigidity problem)**.** *Given an input graph X via its adjacency matrix, check whether the graph is rigid.*

Clearly, an oracle for the automorphism problem, or by Mathon's result [22], the graph isomorphism problem, is sufficient to decide the rigidity of a graph. However, the other direction is open.

Open problem 2.5. *Is the graph rigidity problem polynomial-time equivalent to the graph isomorphism problem.*

An important variant of graph isomorphism is the isomorphism of coloured graphs. For this article, a *c-colouring* of a graph X, where c a positive integer, is a map from the vertex set $V(X)$ to the set of integers $1, \ldots, c$. Given a c-colouring ψ, the ith *colour class* is subset $\psi^{-1}(i)$ of $V(X)$. A *coloured graph* is a tuple (X, ψ) of a graph X and colouring ψ. We often suppress the colouring ψ

when it is understood from the context and just denote the coloured graph by X. Given two c-coloured graphs (X, ψ) and (Y, φ), an isomorphism f between the underlying graphs X and Y is a *coloured graph isomorphism* if it respects the vertex colours, i.e. for any vertex v of X, $\psi(v) = \varphi(f(v))$. An automorphism of a coloured graph is analogously defined. Clearly coloured graph isomorphism generalises graph isomorphism as we can assume an ordinary graph as 1-coloured graph. In the other direction, coloured graph isomorphism polynomial-time Turing reduces to the graph isomorphism problem. The key idea is the following *gadget construction*. For a coloured graph X, we construct a new graph $\tilde{X}$ by first adding, for each colour class i, a long path L_i (say of length $n + i + 1$). We then connect all the vertices of the colour class i to one of the end points of L_i. Given coloured graphs X and Y, any isomorphism between the modified graphs $\tilde{X}$ and $\tilde{Y}$ forces the vertices in a given colour class of X to be mapped to the vertices of the same colour class in Y due to the graph gadgets L_i. Therefore, the coloured graphs X and Y are isomorphic if and only if the modified graphs $\tilde{X}$ and $\tilde{Y}$ are isomorphic. The rigidity problem and the automorphism problem generalise naturally to coloured graphs as well.

The graph isomorphism problem and the automorphism problem can be defined for directed graphs as well. It turns out that these variants are polynomial-time Turing reducible to the undirected case. Therefore, in this article, we mostly concentrate on undirected graph isomorphism. Nonetheless, from the perspective of the isomorphism problem, there is an important subclass of directed graphs called *tournaments* that we define below.

Definition 2.6. *A directed graph X is a* tournament *if for every two distinct vertices u and v, exactly one of the directed edge (u, v) or (v, u) exists in $E(X)$.*

The automorphism group of a tournament cannot have a 2-cycle (why?), and hence has to be of odd order. This forces it to be *solvable* by Feit-Thompson theorem [9]. This property has been exploited by Babai and Luks [3] to give significantly efficient algorithms for tournament isomorphism.

3 Basic polynomial-time algorithms

In this section, we mention some well known polynomial-time algorithms for permutation group problems. The very first polynomial-time algorithm is the algorithm to compute the orbits of a permutation group. Let S be a generating set of the permutation group G then define a relation $\alpha \to_S \beta$ if there exists a g in S such that $\alpha^g = \beta$. It is easy to see that the symmetric, transitive closure of the relation $\to_s$ gives us all the G-orbits. We can thus compute the orbits efficiently by computing reachability.

Lemma 3.1. *There is a polynomial-time algorithm, which given a generating set S of a permutation group G on Ω and an $\alpha \in \Omega$, computes the orbit α^G.*

Many permutation group algorithms follows the general scheme of first reducing the problem to the transitive case by finding all the orbits of the group

using the above lemma, and then restricting the group to the orbit. This is followed by a divide can conquer that is done on the blocks of the transitive action of the group. Thus, finding the blocks of a transitive permutation group is a crucial step in various algorithms. Let G be a transitive permutation group over Ω. Fix any two elements α and β in Ω and consider the graph $X_{\alpha,\beta}$ whose vertices are Ω and edges are $\{\alpha,\beta\}^G$. Let Σ be the smallest G-block containing both α and β then Sim's observed [25] that vertices in any connected component C of the graph $X_{\alpha,\beta}$ is a G-block in the block system $\{\Sigma^g | g \in G\}$ associated with Σ. By running this algorithm on all pairs one can compute the set of minimal (as well as maximal) blocks of G-blocks.

Lemma 3.2. *There is a polynomial-time algorithm that takes as input the generating set of a transitive permutation group G on Ω and decides whether G is primitive or not. If the input group G is not primitive, then it computes a minimal (or maximal) G-block system.*

We already argued that a generating set of a group is a natural way to present a permutation group. A *strong generating set* is a special generating set of a permutation group that makes many computational tasks easy. Consider a permutation group G on the set Ω. Fix an ordering $\{\alpha_1, \ldots, \alpha_n\}$ on the set Ω and let $G^{(i)}$ denote the subgroup of G that fixes the first i elements of Ω, i.e. the subgroup of all elements g of G such that $\alpha_j^g = \alpha_j$ for all $1 \leq j \leq i$. Consider the *tower* $G = G^{(0)} \geqslant \ldots \geqslant G^{(n-1)} = 1$ of subgroups of G. Let C_i be any set of permutations that has exactly one element from each right coset of $G^{(i)}$ in $G^{(i-1)}$, i.e. C_i is a *right transversal* of $G^{(i)}$ in $G^{(i-1)}$. Given any permutation g in G, there is an unique element, say h_1, in C_1 which is in the same right coset of $G^{(1)}$ as that of g. It is easy to see that $g' = gh_1^{-1}$ is in $G^{(1)}$. Continuing this argument with g' and the group $G^{(1)}$, it is easy to see that any element g can be expressed as a product $h_1 \ldots h_{n-1}$, $h_i \in C_i$. In fact, if the transversals C_i's are fixed, the above product representation is unique. Thus, $\cup_i C_i$ forms a generating set of G which we call the *strong generating* set of G. Many computational tasks become trivial once the strong generating set is calculated. For example, the uniqueness of the product representation of g shows that the order of the group $\#G$ is the product of the sizes $\prod_i \#C_i$.

We now describe the algorithm to compute the strong generating set of a permutation group that was given in its complete form by Furst et al. [11] based on ideas from Sims [26]. It is based on the following lemma due to Schreier and hence it (and similar algorithms) are some times called *Schreier-Sims algorithm.*

Lemma 3.3 (Schreier's Lemma)**.** *Let G be a group and H be a subgroup of G. Let T be any right transversal of H in G that contains 1 as a coset representative. For each g in G, let $\overline{g}$ denote the unique coset representative of Hg in T. Let S be a generating set for G then set*

$$S' = \{ts(\overline{ts})^{-1} | t, s \in S\}$$

generates the group H

Let S be the generating set of a permutation group G. The main idea of the algorithm is that once we have a right transversal C_1 of $G^{(1)}$ in $G^{(0)} = G$, we can use Schreier's Lemma 3.3 to compute the Schreier generating set for $G^{(1)}$. We then recursively compute the the strong generating set for $G^{(1)}$. At each stage of the algorithm, we compute the right transversal C_{i+1} and recurse on the Schreier generating set of $G^{(i+1)}$ obtained in that stage.

The right transversal C_1 is computed by starting with the set $T_0 = 1$ and inductively compute T_{i+1} as follows: T_{i+1} is the union of T_i and a subset of T_iS such that T_{i+1} does not contain any redundant representative of same right coset of $G^{(1)}$, i.e. T_{i+1} does not contain two distinct elements g_1 and g_2 such that $\alpha_1^{g_1} = \alpha_1^{g_2}$. If at some point $T_{i+1} = T_i$, we stop the procedure. Since the set C_1 can at most have n elements this procedure has to terminate in polynomially many steps. The actual algorithm [11] can be significantly more efficient by computing all the transversals C_i's simultaneously through a sifting procedure. We summarise all the polynomial-time solvable tasks that uses the Schreier-Sims procedure in the following lemma.

Lemma 3.4 (Furst, Hopcroft, and Luks)**.** *There are polynomial-time algorithms for the computational tasks:*

1. *computing a strong generating set,*
2. *computing the order of a permutation group,*
3. *checking the membership of a permutation $g \in Sym(\Omega)$ in a given permutation group G.*

The Schreier-Sims algorithm can be generalised to find the generating set of a subgroup H of a permutation group G, given indirectly by a membership oracle, provided the index $\frac{\#G}{\#H}$ is small. We state this in the next lemma.

Lemma 3.5. *There is algorithm that takes as input a permutation group G on Ω via a generating set S and computes the generating set of a subgroup H of G given via a membership oracle, i.e. a procedure to test whether a given element g of G is actually an element of H. The algorithm takes time polynomial in $\#S$, $\#\Omega$ and the index $\frac{\#G}{\#H}$.*

Consider a permutation group G on Ω and let Δ be any subset of Ω. The *point-wise stabiliser* of the set Δ, which we denote by $G(\Delta)$ is the subgroup of all elements of G that fix every element of G, i.e. $G(\Delta) = \{g | \delta^g = \delta,\ \forall \delta \in \Delta\}$. It is easy to see that finding the point-wise stabiliser of any subset of Ω can be done in polynomial-time by adapting the Schreier-Sims algorithm.

4 Divide and conquer algorithms for permutation groups

We now illustrate a general technique that is used in many permutation group algorithms by giving an algorithm to find the setwise stabiliser for special groups.

Although superficially similar to point-wise stabiliser, computing the setwise stabiliser is a different ball game. It is at least as hard as graph isomorphism: For a graph X, consider the group $G = \mathrm{Sym}\,(V(X))$ acting on the set $\Omega = \binom{V(X)}{2}$. The automorphism group of the graph X is the set-wise stabiliser of the subset $E(X)$ of Ω. The setwise stabiliser problem is a variant of a more general problem which we define below.

Problem 4.1 (Colour preserving subgroup)**.** *Let G be a permutation group on Ω which is partitioned into k-colour classes $\mathcal{C} = \{C_i\}_{i=1}^{k}$. Compute the subgroup of G that stabilises each of the colour class C_i, i.e. compute the subgroup $\{g \in G | C_i^g = C_i\}$*

The setwise stabiliser problem is the special case when the number of colours is 2. While we cannot expect a polynomial-time algorithm for this problem in general without solving the graph isomorphism problem, for special groups, we can solve the above in polynomial-time. For example, if we know that the input group G is solvable then we have a polynomial-time algorithm. In fact, the polynomial-time algorithm of Luks [19] for trivalent graphs uses such a subroutine as the group that occurs there is a 2-group and hence is solvable.

We now given a sketch of the algorithm, detail of which can be found in the paper by Luks [19]. To avoid notation clutter we fix an input group G and the colouring $\mathcal{C}$ of Ω. We say that a permutation g preserves colours of all elements in the subset Σ if for all $\alpha \in \Sigma$, α and α^g are in the same colour class. Let H be a subgroup of G and Σ an H-stable subset of Ω. We denote $\mathrm{CP}\,(H, \Sigma)$ to be the subset of H that preserves the colours of elements of Σ. Our task is to compute $\mathrm{CP}\,(G, \Omega)$. For the divide and conquer algorithm to work, we need to generalise the problem to cosets of permutation groups: We need to compute $\mathrm{CP}\,(Hg, \Sigma)$ for the coset Hg of the subgroup H of G where the set Σ is H-stable. Note that Σ is not necessarily stabilised by elements of Hg.

The set $\mathrm{CP}\,(Hg, \Sigma)$ has the following crucial properties which follows more or less directly from the definitions.

Lemma 4.2. *1. The set $\mathrm{CP}\,(H, \Sigma)$ is a subgroup of H.*

2. The set $\mathrm{CP}\,(Hg, \Sigma)$ is either empty or is a coset of the group $\mathrm{CP}\,(H, \Sigma)$.

3. Suppose Σ is the disjoint union $\Sigma_1 \uplus \Sigma_2$ both of which are H-stable then $\mathrm{CP}\,(Hg, \Sigma) = \mathrm{CP}\,(\mathrm{CP}\,(Hg, \Sigma_1), \Sigma_2)$.

It is crucial that $\mathrm{CP}\,(Hg, \Sigma)$ is a coset (item 2 in the above lemma) because we can then succinctly represent the set by giving a generating set of $\mathrm{CP}\,(H, \Sigma)$ and the coset representative.

We are now ready to give the outline of the divide and conquer algorithm for computing $\mathrm{CP}\,(Hg, \Sigma)$.

Reduction to transitive case Let $\Sigma' \subset \Sigma$ be any H-orbit. We first compute the group $\mathrm{CP}\,(Hg, \Sigma')$ which is the transitive case of the above problem. Let $\Sigma = \Sigma' \uplus \Sigma''$. We use the fact that $\mathrm{CP}\,(Hg, \Sigma) = \mathrm{CP}\,(\mathrm{CP}\,(Hg, \Sigma'), \Sigma'')$.

Transitive case For this case H acts transitively on Σ. Let Δ be a *maximal* H-block of H and let $\mathcal{B}(\Delta) = \{\Delta_1, \ldots, \Delta_k\}$ be the associated block system. Let N be the normal subgroup of H that fixes all the blocks $\mathcal{B}(\Delta)$ setwise. Then we have $H = \uplus_x Nx$ as a disjoint union of cosets of N. We can then compute the set $\mathrm{CP}(Hg, \Sigma)$ by taking the union of all the cosets $\mathrm{CP}(Nxg, \Sigma)$ which are not empty.

If the number of cosets Nx are polynomially bounded then we can compute a generating set for N using Lemma 3.5. It then amounts to recursively computing the polynomially many cosets $\mathrm{CP}(Nxg, \Sigma)$ and combining the non-empty ones. The number of cosets Nx that is considered in the transitive case is the same as the order of the quotient group H/N. Since the block Δ that we choose is the maximal block, the quotient group H/N, as a permutation group on the set $\mathcal{B}(\Delta)$, is a primitive group (See 2.1). Thus, we need a bound on the size of a primitive permutation group. While the order of a primitive permutation group on n elements can be exponential in n, consider the case of the primitive group S_n for example, for solvable primitive permutation groups, a result by Pálfy [23, Theorem 1] gives us the polynomial bound we are looking for.

Theorem 4.3 (Pálfy)**.** *There are absolute constants C and c such that any solvable primitive permutation group on Ω is of size less than $C \cdot \#\Omega^c$.*

The above bound has a generalisation to groups with bounded non-abelian composition factors: Let Γ_d denote the class of groups such that each composition factor is either abelian or is isomorphic to a subgroup of S_d. Babai et al. [4] generalised the Pálfy's bound to the class Γ_d.

Theorem 4.4 (Babai, Cameron, and Pálfy)**.** *There are absolute constants C and c such that for any positive integer d, any primitive permutation group on Ω in the class Γ_d is of size less than $C\#\Omega^{cd}$.*

As a result, the colour preserving subgroup problem is solvable in polynomial-time for groups that are in the class Γ_d.

Lemma 4.5. *Colour preserving subgroup problem is solvable in polynomial-time for the class of solvable groups and the class of groups in the family Γ_d for constant d.*

We now discuss a natural context where the colour stabiliser problem for groups in the class Γ_d occurs. Consider the graphs of valence d, i.e. all vertices are of degree less than or equal to d. Luks [19] gives a polynomial-time algorithm for this class of graphs by reducing it to the colour preserving subgroup problem where the input group G is in the class Γ_{d-1}. We quickly give a sketch.

Fix the two input graphs graphs X_1 and X_2. We assume that the graphs are connected, otherwise we run the algorithm for each pair of connected components. Furthermore, we restrict our attention to checking whether X_1 and X_2 are isomorphic via an isomorphism that maps a particular edge e_1 of X_1 to an edge e_2 of X_2: We just need to repeat the procedure for all such pairs of edges to know whether X_1 and X_2 are isomorphic.

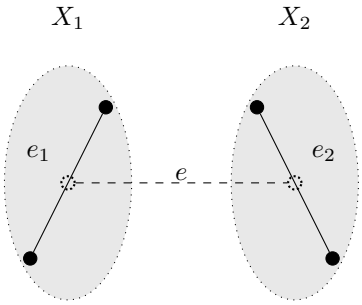

Figure 1: Luks gadget for bounded valence graphs

Consider the new graph which we denote by Z which is essentially the disjoint union of the graph X_1 and X_2 with an additional edge e that connects the mid points of e_1 and e_2 (see figure 4). If $d > 2$ is the maximum degree of any vertex in the input graph then Z also has degree bounded by d. Luks algorithm for bounded valence computes the group subgroup $\mathrm{Aut}\,(Z)_e$ of $\mathrm{Aut}\,(Z)$ that maps the auxiliary edge e to itself. It is clear that the input graphs X_1 and X_2 are isomorphic if and only if there is at least one element in $\mathrm{Aut}\,(Z)_e$ (and therefore in any generator set) that flips the edge e. The algorithm then proceeds to compute the group $\mathrm{Aut}\,(Z)_e$. First the graph Z is layered as follows: Let the ith layer of Z, denoted by Z_i, be the sub graph which contains all the edges (as well as the end points) at a distance i from the auxiliary edge e. In particular, the graph Z_0 consists of just the edge e and its end points. All automorphisms of Z that stabilises the edge e has to preserve this layered structure. The group $\mathrm{Aut}\,(Z)_e$ is then computed by inductively computing the groups $G_i = \mathrm{Aut}\,(Z_i)_e$.

Inductively, Luks proves that G_i's are in the class Γ_{d-1} as follows: Let H_{i+1} denote the subgroup of G_i that is obtained by restricting elements of G_{i+1} on Z_i and let K_{i+1} be the associated kernel, i.e. the subgroup of G_{i+1} that is trivial when restricted to Z_i. For a fixed vertex u in layer i (i.e. in the graph $Z_i \setminus Z_{i-1}$ is connected to at most $d-1$ vertices in the layer $i+1$ and these vertices have to be mapped within themselves by elements of K_{i+1}. Therefore, K_{i+1} is a subgroup of a product of m many copies of the symmetric groups S_{d-1} for some positive integer m. The quotient group G_{i+1}/K_{i+1} is the group H_{i+1}, which itself is a subgroup G_i, a group in the class Γ_{d-1}. This is possible only if G_{i+1} is in Γ_{d-1}: Consider any composition series of G_{i+1} which passes through the normal subgroup K_{i+1}. The composition factors are either composition factors of H_i or that K_{i+1}.

Having computed G_i the algorithm computes G_{i+1} by computing (1) a generating set for the kernel K_{i+1} and (2) a set of elements of G_{i+1} whose restriction to G_i generates H_i. It is this inductive step that requires the solution of colour preserving subgroup problem and luckily the input group (G_i in our case) turns out to be in the class Γ_d and hence solvable by the algorithm in Lemma 4.5. Thus we have the following theorem.

Theorem 4.6 (Luks)**.** *Consider the family $\mathcal{G}_d$ of graphs whose vertices are of degree bounded by d. There is a $n^{O(d)}$ algorithm to decide isomorphism of graphs in $\mathcal{G}_d$.*

The current fastest algorithm for graph isomorphism [29] is based on a valence reduction step together with the application of the above theorem of Luks for bounded valence graphs. Therefore, any improvement on the bounded valence case will improve the state of the art for the general graph isomorphism problem.

5 Lexicographically least permutations

We now mention some results that make use of the ordering of permutations induced by the ordering on the domain Ω. Firstly note that a total ordering on the set Ω gives a total ordering on $\mathrm{Sym}\,(\Omega)$: for distinct permutations g and h, $g < h$ if at the first (in the ordering on Ω) element α in Ω where they differ, we have $\alpha^g < \alpha^h$. We call this ordering the *lexicographic ordering* on the permutations. Under this ordering the lexicographically least permutation is the identity permutation. The first problem that we study is the problem of computing the lexicographically least element in a coset.

Problem 5.1 (lexicographically least in a coset)**.** *Given a permutation group G on Ω as a set of generators and an arbitrary permutation x on Ω, compute the lexicographically least element in the coset Gx.*

Clearly if x is in G then the coset is the group G itself and the lexicographically least element of the coset is the identity element. We now sketch a polynomial-time algorithm for this problem.

Let α be the least element of Ω. The set of images of α under permutations in the coset Gx is given by the set $\alpha^{Gx} = (\alpha^G)^x$ which can be computed easily once the orbit α^G of α is computed. Clearly, the lexicographically least element of Gx should map α to the least element β of α^{Gx}. We can also compute, using the transitive closure algorithm for orbits, an element x_1 in the coset Gx such that $\alpha^{x_1} = \beta$. Therefore the lexicographically least element of Gx is also the lexicographically least element in the coset $G_\alpha x_1$ as this coset is precisely the set of elements of Gx that maps α to β. A similar algorithm can be give for left cosets xG as well. We thus have the following lemma:

Lemma 5.2. *Computing the lexicographically least element in a coset can be done in polynomial-time.*

The above lemma is a key step in proving that the graph isomorphism problem is in the complexity class SPP.

Definition 5.3 (SPP)**.** *For a non-deterministic polynomial-time Turing machine let* $\mathrm{gap}(M, x)$ *denote the* difference *in the number of accepting paths and rejecting paths of M on the input x. A language L is in the complexity class*

SPP if there is a polynomial time non-deterministic Turing machine M such that for all strings x in the language L, the $\mathrm{gap}(M, x)$ *is 1 and for all strings not in the language L,* $\mathrm{gap}(M, x)$ *is 0.*

Languages in SPP are believed to be of low complexity and are unlikely to be NP-hard. In particular, any *gap definable complexity* [10] class not only contain SPP but also derive no extra computational power with a language in SPP as oracle, i.e. SPP is *low* for all these complexity classes. Gap definable complexity classes [10] are counting classes defined using GapP functions, i.e. functions that are differences of accepting and rejecting paths of an NP machine, and contain many common counting complexity class like PP and $\oplus$P etc.

The main idea involved in the proof is to design a polynomial-time algorithm A that makes queries to an NP language L with some restriction on the queries that A makes to L. We design a non-deterministic polynomial time machine M for L such that for all queries the algorithm A makes, the machine M has at most one accepting path. Such an oracle machine can be converted to an SPP algorithm, i.e. an NP machine whose gap function is the characteristic function of GI, using standard techniques [15].

The base algorithm A is an inductive algorithm that builds the strong generating set of the automorphism group by computing the group G_i of all automorphisms that fix the first i vertices of the graph. To compute G_{i-1} from G_i, the algorithm has to query the NP-language L which essentially checks, given a $j > i$, whether there is an automorphism that maps i to j. The base polynomial-time machine can then find one such by doing a prefix search. However, we need to design an NP machine M for L such that for all queries asked by A, there is at most one accepting path. This we achieve as follows: The algorithm A also provides to L the generator set of G_i, i.e. queries to L are (encoding of) pairs $\langle S, j\rangle$ where S is a generating set of G_i at the i-th stage. We know that if there is an automorphism, say g in G_{i-1}, that maps i to j then the set of all such automorphisms form the coset $G_i g$. The machine M essentially guess the automorphism g that maps i to j if it exists and accepts only if g is the lexicographically least permutation in $G_i g$. Since there is only one such guess g, we know that for all queries that the algorithm A makes to L the machine M has at most one accepting path. The SPP result then follows as mentioned above.

Theorem 5.4 (Arvind and Kurur)**.** *The graph isomorphism problem is in SPP.*

While computing the lexicographically least element in a coset has an efficient algorithm, consider the following generalisation to a double coset.

Problem 5.5 (Lex-least in a double coset)**.** *Given the generating sets of permutation groups G and H on a totally ordered set Ω and an arbitrary permutation x on Ω, compute the lexicographically least element in GxH.*

The problem of computing the lex-least element in a double coset is intimately connected to the problem of graph canonisation which we define below.

Definition 5.6 (Canonical forms for graphs)**.** *Consider the class $\mathcal{G}(\Omega)$ of all graphs on the vertex set Ω. A function* **CF** *on $\mathcal{G}(\Omega)$ is a* canonical form *if it satisfies the following properties:*

1. *For every graph X in $\mathcal{G}(\Omega)$, $\mathbf{CF}(X)$ is isomorphic to X.*
2. *If X and Y are two isomorphic graphs then $\mathbf{CF}(X)$ is the same as $\mathbf{CF}(Y)$.*

In other words, a canonical form **CF** picks a unique representative from each isomorphism class of graphs on Ω. Clearly graph isomorphism is solvable given a canonisation procedure. Therefore, one way of attacking the graph isomorphism problem is to give fast canonisation procedure. For many classes of graphs, Babai and Luks [3] gave an efficient canonisation procedure which is also currently the best general purpose algorithms. In particular, they were able to give an $O\left(n^{c \log n}\right)$ for tournaments. This canonisation procedure makes use of the fact that tournaments have a solvable automorphism group. They also show how Luks' polynomial-time algorithm for bounded valance [19] can be modified and extended to a canonisation algorithm with essentially the same running time.

As opposed to computing the lexicographically least element in a coset, computing it in a double coset is known to be NP-hard [21, Theorem 5.1] even when one of the group is solvable. However, in many contexts, particularly in relation with graph isomorphism and canonisation, we have some freedom to choose the underlying ordering of the set Ω. Can we reorder the set Ω so as to make it possible to apply the divide and conquer technique similar to that of the colour preserving subgroup problem that we saw in the previous section? Indeed this is the case provided the reordering is "compatible" with the divide and conquer structure of the group G. Firstly, we need to generalise the lexicographically least element as follows: Consider an ordering $<$ on Ω. Let Δ be a G-stable set. We consider the restriction of the order $<$ on the set Δ. This gives a partial order on elements of permutations, we say that $g < h$ if for the least δ in Δ on which g and h differ, we have $\delta^g < \delta^h$. Under this restricted ordering there will be multiple lexicographically minimal elements. We now describe how to build a new ordering $\prec$ on Ω under which it is feasible to compute the lexicographically least element of the double coset GxH.

Ordering the orbit Fix an ordering between the G-orbits by picking say the least element in each of them. If Ω_1 and Ω_2 are two orbits such that $\Omega_1 < \Omega_2$ in the above chosen order, then we set every element of Ω_1 to be less than that of Ω_2 under the new ordering $\prec$. The motivation of this reordering is the following: Let $\Omega = \Omega_1 \uplus \Omega_2$ then computing the lexicographically least element with respect to the new ordering $\prec$ can be done by first computing the lexicographically minimal elements with respect to the restriction of $\prec$ on Ω_1 and then from them picking the lexicographically least element with respect to the restriction of $\prec$ on Ω_2.

Ordering within orbits When G is transitive, we do the reordering with respect to the blocks. We pick a maximal G-block Δ in a canonical way.

The Δ block system partition the set Ω so reorder it pretty much the same way as in the previous case using the Δ block system instead. If N denotes the normal subgroup of G that fixes all the blocks in the Δ block system, we can recursively find the lex-least elements in double cosets $NgxH$ and find the minimal ones out of it. If G is in the class Γ_d, the number of sub-problems are polynomially bounded by Theorem 4.4.

The above reordering can be formalised in terms of the *structure forest* of the group G. The structure forest is the collection of *structure trees* one for each G-orbit. For an orbit Σ, the structure tree is a tree where the leaves are elements of Σ. Each internal node v is associated with a G-block Δ_v with the following properties.

1. For any child u of v, the G-block Δ_u is a maximal block contained in Δ_v.
2. If $\{u_1, \ldots, u_k\}$ are the set of children of v then the blocks Δ_{u_i}'s are all conjugates of each other and partition the parent block Δ_v.

This structure forest captures the divide and conquer on G. The elements of Ω can be reordered once we compute the structure forest of G: The structure trees are ordered in the order of the least element in them. For each internal node v and children u_1 and u_2, $u_1 \prec u_2$ if the least element of the associated block in u_1 is smaller than that of u_2 according to the original ordering $<$. This will finally give an ordering $\prec$ on the entire set Ω. Thus, we have the following theorem (See the survey article by Luks [21] for details).

Theorem 5.7. *Given a totally ordered set Ω, permutation groups G and H on Ω and a permutation x on Ω. Suppose G is in the class Γ_d then in time polynomial in n we can compute a new ordering $\leq_G$ such that computing the lex-least element of the double coset GxH can be done in $n^{O(d)}$.*

Notice that we do not have any restriction whatsoever on the group H.

6 Structure of primitive groups

In the last two sections, we saw how bounds on the order of primitive permutation group can be crucial in the runtime analysis of various divide and conquer algorithms for permutation groups. We now mention how knowing the actual structure of primitive groups are computationally useful. This section is mainly motivated by the study of *bounded colour class graph isomorphism problem.*

Problem 6.1 (Bounded colour class graph isomorphism)**.** *Fix a constant b. Given two coloured graph X and Y such that the number of vertices in any given colour class is bounded by b decide whether the graphs are isomorphic.*

We abbreviate this problem as BCGI_b. This restricted graph isomorphism problem does have polynomial-time algorithm but what about fast parallel algorithms? Luks [20] answered this question affirmatively by giving a reduction

to a restricted point-wise stabiliser problem and solving it in NC. Further careful analysis by Arvind et al. [2] showed that the problem lies in the $\mathrm{Mod}_k\mathrm{L}$ hierarchy. Together with the hardness for this class [27], we have a fairly tight classification of this variant of graph isomorphism.

We now explain the overall structure of the algorithm for BCGI_b. Both Luks [20] and Arvind et al. [2] reduce the bounded colour graph isomorphism problem to a restricted version of point-wise stabiliser problem which we now define.

Problem 6.2 (bounded orbit point-wise stabiliser problem)**.** *Given as input a set Ω, a subset Δ of Ω and a permutation group G on Ω such that the G-orbits are all of cardinality bounded by a constant c. Compute generating set of the point-wise stabiliser subgroup $G(\Delta)$.*

We abbreviate this problem as PWS_c. As mentioned before, the above problem is solvable in polynomial-time. However, in this context, we are interested in providing a parallel algorithm. What needs to be exploited is that the G-orbits are bounded and thus G is actually a subgroup of a product of small symmetric groups, the symmetric groups on each of the G-orbits.

To see the connection of the PWS_c and BCGI_b isomorphism we consider the equivalent automorphism problem which we denote by AUT_b.

Lemma 6.3 (Luks)**.** *The AUT_b problem logspace reduces to $\mathrm{PWS}_{2^{b^2}}$ problem.*

Here is the sketch of this reduction. Let X be the coloured graph and let $C_1, \ldots, C_m$ denote the the colour classes into which the vertex set $V(X)$ is partitioned. The automorphism group is a subgroup of the product group $G = \prod_i \mathrm{Sym}\,(C_i)$. We expressed the automorphism group $\mathrm{Aut}\,(X)$ as a point-wise stabiliser of G on its action on a different set Ω that we construct as follows: Define the set $C_{i,j}$ to be the set of unordered pairs $\{u, v\}$ where $u \in C_i$ and $v \in C_j$ and let $E_{i,j}$ be the subset of edges of X between the colour classes C_i and C_j. Define the set $\Omega_{i,j}$ to be the power set of $C_{i,j}$ then the edge sets $E_{i,j}$ are actually *points* or element of $\Omega_{i,j}$. Consider the natural action of G on the union $\Omega = \cup_{i,j}\Omega_{i,j}$ and let Δ be the subset of all the *points* $E_{i,j}$ of Ω. It is easy to see that the point-wise stabiliser of G with respect to the subset Δ is actually the automorphism group $\mathrm{Aut}\,(X)$. Notice that each of the set $\Omega_{i,j}$ are G-stable and hence the orbits of G are at most of size $2^{\binom{b}{2}}$.

Both Luks [20] and Arvind et al. [2] then solve the PWS_c problem. While the actual algorithms of Luks [20] and Arvind et al. [2] are fairly technical, we attempt to explain the essence of the algorithm and the permutation group theory involved in those results.

The group G in question can be seen as a product of groups $G = \prod_i G_i$ where G_i is the action of G on the ith orbit. For each of the groups G_i, compute a special *normal series* $G_i = N_{i,0} \rhd \ldots \rhd N_{i,t}$ and let N_k denote the produce $\prod_i N_{i,k}$. The algorithm does a divide and conquer to compute $N_k(\Delta)$ going one level at a time. The base case of this divide and conquer is when the group $N_{i,s}$ hits a socle. The *socle* of a group G is the group generated by the set of all minimal normal subgroups of G. The main group theoretic result that is used is

the O'Nan-Scott theorem (See the book by Dixon and Mortimer [6] for a proof of this result) on the structure of the socle of a primitive permutation group and its point-wise stabiliser.

For the details of the algorithm, we refer the reader to the conference paper of Arvind et al. [2]. A detailed version is available in the Ph.D thesis of Kurur [17, Chapter 5]

7 Representation of groups on graphs

In this section, we look at the group representability problem. This problem was defined and studied by Dutta and Kurur [7] to explore the connection between graph isomorphism and permutation group algorithms from a representation theoretic point of view. Representation theory is the study of homomorphisms from a group to the group $\mathrm{GL}(V)$, the automorphisms of a vector space V. In the context of graph isomorphism, we would like to understand homomorphisms between groups and automorphisms of graphs.

Definition 7.1. *A* representation *of a group G* on a *graph X is a homomorphism from the group G to the automorphism group* $\mathrm{Aut}\,(X)$ *of the graph X.*

There is always a trivial representation that sends all the elements of the group to the identity automorphism. What we are interested in is a non-trivial homomorphisms. The main problem of interest in this section is the following [7].

Problem 7.2 (Group representability problem)**.** *Given a group G and a graph X, decide whether G has a non-trivial representation on X.*

The hardness of the problem depends on how the group G is presented. We assume, unless otherwise mentioned, that G is provided to the algorithm via a multiplication table. Therefore, one can assume that the input size is $\#G+\#V(X)$. In studying its connection to graph isomorphism, we can restrict the problem in two ways: (1) restrict the groups to come from a natural class like for example solvable or abelian or (2) restrict the class of graphs to say planar graphs or trees.

The very first result that we have in this context is the following [7].

Lemma 7.3 (Dutta and Kurur)**.** *The graph isomorphism problem is log-space many-one reducible to the abelian group representability problem.*

The main idea behind the proof is the following: Consider an instances of graph isomorphism where we want to check whether the graphs X and Y are isomorphic. We assume they are connected and have n vertices. For a prime p, consider the graph Z which is the disjoint union of $p-1$ copies of X and 1 copy of Y. Suppose that the group $\mathrm{Aut}\,(Z)$ has a p-cycle say g. Consider any vertex u of Z such that $u^g \neq u$. It is easy to see that the orbit of u under the cyclic group generated by g has to have p-elements. Furthermore, if any

two of the elements in this orbit is in the same connected component of Z then the entire orbit is. If the prime p is chosen to be greater than n then such a p-cycle necessarily has to permutes the components as each of the connected components of Z are of cardinality at most $n < p$. This is only possible if some copy of X in Z is mapped to the copy of Y and hence X and Y have to be isomorphic. Conversely, for any prime p, if X and Y are isomorphic, then the group $\mathrm{Aut}(Z)$ has a p-cycle. Thus, to decide whether X and Y are isomorphic, we need to check the group representability of the additive group of $\mathbb{Z}/p\mathbb{Z}$, on the graph Z for some prime p greater than the number of vertices in X. By Bertrand's postulate (it is actually a theorem but the name seems to be stuck) there is always a prime p between n and $2n$ which we chose for this purpose.

Notice that for the previous lemma, all we needed is to pick a prime p such that $\mathrm{Aut}(X)$ does not have p-cycle. Recall that tournaments have odd order automorphism group and hence we have the following result.

Theorem 7.4. *Tournament isomorphism is reducible to $\mathbb{Z}/2\mathbb{Z}$ representability.*

In this context, we have the following open problem.

Open problem 7.5. *Is graph isomorphism reducible to $\mathbb{Z}/2\mathbb{Z}$-representability (or for that matter any fixed group representability).*

What about the other direction, i.e. reduction from representability to isomorphism? Dutta and Kurur [7] prove the following result for solvable group representability.

Lemma 7.6 (Dutta and Kurur)**.** *The representability problem for solvable groups is polynomial-time Turing reducible to graph isomorphism.*

For a group G, let G' be the commutator subgroup. The main idea is the following group theoretic fact.

Lemma 7.7. *A solvable group G is representable on X if and only if there is a prime p that divides both the orders G/G' and $\#\mathrm{Aut}(X)$.*

From the above lemma it follows that to check representability for solvable groups, all we need is a way to compute the orders $\#G/G'$ and that of $\#\mathrm{Aut}(X)$. Clearly the former can be computed easily as the group is presented as a multiplication table and the latter using an oracle to the automorphism problem (or equivalently the graph isomorphism problem). We can even assume that the group is presented as a permutation group because there are efficient algorithms to compute the commutator subgroup of a permutation group [11, Theorem 4].

Thus as far as group representability is concerned, as long as we restrict the problem to solvable groups, we are within the realm of graph isomorphism. However, even for the simplest of the non-solvable case we do not have a satisfactory answer:

Open problem 7.8 (A_5 representability problem)**.** *Given a graph X decide whether there is a subgroup of $\mathrm{Aut}(X)$ which is isomorphic to the alternating group of A_5.*

The importance of A_5 here is that it is the smallest example of a non-solvable group. Since A_5 is a simple group, non-trivial homomorphisms from it to $\mathrm{Aut}\,(X)$ can only be injections.

Torán [27] showed that graph isomorphism is hard for a lot of parallel complexity classes like $\oplus$L etc. An important open problem in the context of graph isomorphism is whether it is hard for the complexity class P (under suitable reductions). If this is the case, it would give evidence that it is unlikely to have efficient parallel algorithms for graph isomorphism. We would like to pose the same question for the group representability problem

Open problem 7.9. *Is the group representability problem hard for the complexity class* P*.*

Are there reasons to believe that the group representability problem is harder than graph isomorphism ? We really do not know. However, for the restricted case of representability on trees, we already have a difficulty. Graph isomorphism on trees can be done in polynomial time. In fact, even for planar graphs isomorphism testing can be done in linear time [14] or, if one is interested in the space bounded classes, in logarithmic space [8]. In contrast, Dutta and Kurur [7] proved the following result for representability on trees.

Theorem 7.10 (Dutta and Kurur)**.** *The problem of group representability on trees is Turing equivalent to the problem of testing, given an integer n in unary and group G via multiplication table, whether there is a non-trivial homomorphism to the symmetric group S_n or not.*

We call the problem of checking whether a group G has a homomorphism to S_n as *permutation representability problem* and is motivated by Cayley's theorem that states that every finite group is a subgroup of a symmetric group. However, finding the smallest n for which G is a subgroup seems to be hard although we admit that there are no known hardness result for the above problem.

8 Conclusion

In this article, we discussed the complexity of some permutation group algorithms and its close connection to graph isomorphism. Most of these algorithms perform a divide and conquer and it is here the structure of permutation groups plays a crucial role. Of particular interest are permutation group theoretic structures like orbits and blocks whose computation allows us to often reduce the general case to the primitive case. Also in most of these cases the primitive case is solvable if the group is known to be in some special class like solvable or Γ_d. This makes use of bounds on the sizes of primitive groups or, in some cases, their explicit structure. We also saw how these classes naturally arose in study of restricted versions of graph isomorphism. We therefore believe that a better understanding of permutation groups and is relation to graph isomorphism is crucial in pinning down the computational complexity of this elusive problem.

References

[1] V. Arvind and Piyush P Kurur. Graph Isomorphism is in SPP. In 43^{rd} *Annual Symposium of Foundations of Computer Science*, pages 743–750. IEEE, November 2002.

[2] V. Arvind, Piyush P Kurur, and T. C. Vijayaraghavan. Bounded color multiplicity Graph Isomorphism is in the $\#L$ hierarchy. In *20th Conference on Computational Complexity (CCC 2005)*, pages 13–27. IEEE, June 2005.

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

[4] László Babai, Peter J. Cameron, and P. P. Pálfy. On the order of primitive groups with restricted nonabelian composition factors. *Journal of Algebra*, 79:161–168, 1982.

[5] R. Boppana, J. Hastad, and S. Zachos. Does co-NP have short interactive proofs, May 1987.

[6] John D. Dixon and Brian Mortimer. *Permutation Groups*. Number 163 in Graduate texts in mathematics. Springer-Verlag, 1991.

[7] Sagarmoy Dutta and Piyush P. Kurur. Representating groups on graphs. *CoRR*, abs/0904.3941, 2009.

[8] Samir Dutta, Nutan Limaye, Prajakta Nimbhorkar, Thomas Thierauf, and Fabian Wagner. Planar graph isomorphism is in logspace. In *Proceedings of the IEEE Conference on Computational Complexity*, pages 203–124, 2009.

[9] Walter Feit and John G. Thompson. Solvability of groups of odd order. *Pacific Journal of Mathematics*, 13(3):775–1027, 1963. URL `http://projecteuclid.org/euclid.pjm/1103053943`.

[10] Stephen A. Fenner, Lance Fortnow, and Stuart A. Kurtz. Gap-definable counting classes. In *Structure in Complexity Theory Conference*, pages 30–42, 1991. URL `citeseer.nj.nec.com/fenner92gapdefinable.html`.

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

[12] M. Garey and D. Johnson. *Computers and Intractability: A guide to the theory of NP-completeness*. W. H. Freeman, 1979.

[13] Marshall Hall Jr. *The Theory of Groups*. The Macmillan Company, New York, first edition, 1959.

[14] J. E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs (preliminary report). In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, STOC '74, pages 172–184, New York, NY, USA, 1974. ACM.

[15] Johannes Köbler, Uwe Schöning, and Jacobo Torán. Graph isomorphism is low for PP. *Computational Complexity*, 2(4):301–330, 1992. URL `citeseer.nj.nec.com/obler92graph.html`.

[16] Johannes Köbler, Uwe Schöning, and Jacobo Torán. *The Graph Isomorphism Problem: Its Structural Complexity*. Birkhauser, 1993.

[17] Piyush P Kurur. *Complexity upper bounds using permutation group theory*. PhD thesis, Institute of Mathematical Sciences, Chennai, India, 2006.

[18] Richard E. Ladner. On the structure of polynomial time reducibility. *Journal of the ACM*, 22(1):155–171, 1975.

[19] Eugene M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of Computer and System Sciences*, 25(1):42–65, 1982.

[20] Eugene M. Luks. Parallel algorithms for permutation groups and graph isomorphism. In *Proceedings of the IEEE Foundations of Computer Science*, pages 292–302. IEEE Computer Society, 1986.

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

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

[23] P. P. Pálfy. A polynomial bound for the orders of primitive solvable groups. *Journal of Algebra*, pages 127–137, July 1982.

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

[25] C. C. Sims. Graphs and finite permutation groups. *Mathematische Zeitschrift*, 95:76–86, 1967.

[26] C. C. Sims. Some group theoretic algorithms. *Topics in Algebra*, 697: 108–124, 1978.

[27] Jacobo Torán. On the hardness of graph isomorphism. *SIAM Journal of Computing*, 33(5):1093–1108, 2004.

[28] Helmut Wielandt. *Finite Permutation Groups*. Academic Press, New York, 1964.

[29] V. N. Zemlyachenko, N. M. Korneenko, and R. I. Tyshkevich. Graph isomorphism problem. *Journal of Soviet Mathematics*, 29:1426–1481, 1985.