UGP Report

The Graph Isomorphism Problem

Farzan Adil Byramji　　　　Mohd Talib Siddiqui

Supervisor: Dr. Nitin Saxena
Department of Computer Science and Engineering
Indian Institute of Technology Kanpur

Contents

1 Introduction

The computational complexity of the Graph Isomorphism (GI) problem has been an open problem in the area of computer science for a long time. When Karp published his paper on NP-complete combinatorial problems, he stated GI as an open problem. GI is one of the few problems that is in the class NP, but it is not known to be either NP-complete or in the class P. It is believed unlikely that GI is NP-complete since that would lead to the collapse of the polynomial hierarchy to the second level.

In our report, we first look at a combinatorial approach that gives us the color refinement algorithm. It is an incomplete isomorphism test, that deploys a simple iteration to update the colors of nodes based on the colors of its neighbours. Despite being a simple classification algorithm, it was shown that almost all graphs can be distinguished by it.

A generalisation of the color refinement algorithm came later, called the Weisfeiler-Leman Algorithm, that coloured tuples of nodes in a graph rather than a single one. It was much more powerful than colour refinement but was still an incomplete isomorphism test. However it became exceedingly difficult to construct graphs that defied this test.

Significant advancement on GI were made by Babai and Luks when they introduced computational group theoretic ideas in this domain. Luks [Luk82] presented a reduction to string isomorphism problem for bounded degree case, and provided a recursive algorithm for string isomorphism that worked in n^{c_d} time in case of graphs with degree bounded by d where c_d is a constant depending only on d. We present the algorithm and reduction of Luks in our report. These group-theoretic ideas when combined with combinatorial ideas of Zemlyachenko were used by Luks to give an $\exp(O(\sqrt{n \log n}))$ algorithm for general graphs, sketched in [BKL83].

A related problem is that of Hypergraph Isomorphism. In [Luk99], it was shown that for a dense input representation, Hypergraph Isomorphism is in polynomial time. So the input has length $\Omega(2^n)$. More generally, it was shown that for hypergraphs on n vertices isomorphism can be tested in $O(c^n)$ time for some constant c. As corollaries, we obtain that testing structural equivalence and isomorphism of Boolean functions given by truth tables is in polynomial time. This article also contained simple algorithms for Graph Isomorphism and Coset Intersection which run in simply exponential time $O(c^n)$ for a constant c.

Later, Babai [Bab16] developed on this and used several new group theoretic ideas to reduce the complexity to quasi-polynomial time. The foundation of Babai's algorithm is also the divide and conquer strategy of Luks, with some other group theoretic and combinatorial mechanisms. We sketch some of the main ideas of Babai's algorithm towards the end.

2 Preliminaries

In this section, we fix some notation, and give basic background on some concepts.

2.1 Graphs

For a graph X, the vertex set is denoted be $\mathcal{V}(X)$, and the edge set is denoted by $\mathcal{E}(X)$. We abuse the notation for graphs to denote $|X| = |\mathcal{V}(X)|$. A *subgraph* of X, Y is a graph with $\mathcal{V}(Y) \subseteq \mathcal{V}(X)$, and $\mathcal{E}(Y) \subseteq \mathcal{E}(X)$. The set of neighbours for a vertex $v \in \mathcal{V}(X)$, is given by $\mathcal{N}(v) = \{u | \{u, v\} \in \mathcal{E}(X)\}$. The degree of a vertex v is just the size of its neighbour set, $\deg(v) = |\mathcal{N}(v)|$.

We say that $h : \mathcal{V}(X) \longrightarrow \mathcal{V}(Y)$ is a *homomorphism* from graph X to graph Y if, $\{u, v\} \in \mathcal{E}(X)$ implies that $\{h(u), h(v)\} \in \mathcal{E}(Y)$. h is an *isomorphism* if h is bijective and $\{u, v\} \in \mathcal{E}(X)$ iff $\{h(u), h(v)\} \in \mathcal{E}(Y)$. We use the notation $X \cong Y$ to say X and Y are isomorphic. We define a *vertex colouring* of a graph by $\chi : \mathcal{V}(X) \longrightarrow C$, where C is the set of colours. Homomorphisms and isomorphisms must preserve vertex colourings for vertex coloured graphs. We denote a multiset by double curly braces $\{\!\{.\}\!\}$.

Next, we define *Johnson Graphs.* $J(k, t)$ is a with $t \geq 1$, and $k \geq 2t + 1$, is a graph with vertex set $V = \{v_T | T \subseteq \Delta, |T| = t\}$, where $|\Delta| = k$. There is an edge between two vertices v_T and v_S iff $|T \setminus S| = 1$.

2.2 Groups

Let G be a finite group. If H is a subgroup of G, we write $H \leq G$. The set of (right) cosets of H is denoted by $G/H = \{Hg \mid g \in G\}$. The index of H in G is denoted by $|G : H|$, which by Lagrange's theorem, is $|G|/|H|$. If H is a normal subgroup of G, we use the notation $H \trianglelefteq G$ and $H \triangleleft G$ if H is a proper normal subgroup of G.

For a group G, a chain of subgroups

$$G = G_0 \geq G_1 \geq \cdots \geq G_n = \{1\}$$

is called a subnormal series if for each $i \in [n]$, $G_{i-1} \trianglerighteq G_i$. Moreover if $G_i \neq G_{i-1}$ and G_{i-1}/G_i is simple for all $i \in [n]$, a subnormal series is called a composition series of G. For a composition series of G, the quotient groups G_{i-1}/G_i are called composition factors of G. The Jordan-Hölder theorem states that the composition factors are independent of the composition series up to isomorphism and permuting the factors. A proof can be found in most group theory texts (for instance, [Mil21, Chapter 6]). The Jordan-Hölder theorem will be used implicitly in 5.1 while reasoning about composition series. The *socle* $\mathrm{Soc}(G)$ of the group G is defined as the product of its minimal normal subgroups.

We will mostly be dealing with permutation groups in this domain. Thus, we define a *permutation group* acting on a set Ω, to be a subgroup $\Gamma \leq \mathrm{Sym}(\Omega)$ of the symmetric group on it. For symmetric groups, we also use the notation S_n for the symmetric group over a domain of n elements. Similarly, A_n for $\mathrm{Alt}(\Omega)$, alternating group over n elements. The degree of the permutation group is defined by the size of domain, $|\Omega|$ in this case. For any $\gamma \in \Gamma$, and any $\alpha \in \Omega$, we define the notation α^γ, to denote the image of the element α in the permutation γ. The *orbit* of an $\alpha \in \Omega$, denoted by $\alpha^\Gamma := \{\alpha^\gamma | \gamma \in \Gamma\}$, the elements α can be permuted to under any permutation in Γ. The group Γ is said to be transitive if the orbit of an element (consequently all elements) in Ω is Ω itself.

The *stabiliser* of some element $\alpha \in \Omega$, is denoted by $\Gamma_\alpha := \{\gamma \in \Gamma \mid \alpha^\gamma = \alpha\}$, which is a subgroup of Γ, and all its permutation, fix α. Similarly, for a set A, we define the *pointwise stabiliser*, $\Gamma_{(A)} := \{\gamma \in \Gamma | \forall \alpha \in A, \alpha^\gamma = \alpha\}$. For a subset $A \subseteq \Omega$, define the notation $A^\gamma := \{\alpha^\gamma | \alpha \in A\}$. Now,using this, we define a *setwise stabiliser* for $A \subseteq \Omega$, as the subgroup $\Gamma_A := \{\gamma \in \Gamma | A^\gamma = A\}$. The set A is said to be Γ-invariant if $\Gamma_A = \Gamma$. Now, for $A \subseteq \Omega$, we define notation $\Gamma[A] :]\{\gamma[A] | \gamma \in \Gamma\}$, the induced action of the group Γ on A, where $\gamma[A]$ is the permutation restricted to A.

Now, we see some definitions and notations which we encounter while dealing with transitive permutation groups. We will assume $\Gamma \leq \mathrm{Sym}(\Omega)$ to be transitive. A non-empty set $B \subseteq \Omega$, is a *block* if it holds for all $\gamma \in \Gamma$ that either $B^\gamma = B$ or $B^\gamma \cap B = \phi$. It is easy to see that Ω itself, and all the singleton subsets of Ω are blocks. We say that the group Γ is *primitive* if there are no non-trivial blocks of it. If we have a block $B \subseteq \Omega$ then a *block system* can be built as $\mathfrak{B} := \{B^\gamma | \gamma \in \Gamma\}$. Since we are dealing with permutation groups, we can infer that $\mathfrak{B}$ is an equipartition of the domain Ω. The subgroup that stabilises all the blocks setwise is defined as $\Gamma_{(\mathfrak{B})} := \{\gamma \in \Gamma | \forall B \in \mathfrak{B}, B^\gamma = B\}$. We can also define a natural action of Γ on the block system $\mathfrak{B}$, as $\Gamma[\mathfrak{B}] \leq \mathrm{Sym}(\mathfrak{B})$, which permutes blocks.

If we have two block systems, $\mathfrak{B}$ and $\mathfrak{B}'$, then we define, $\mathfrak{B} \preceq \mathfrak{B}'$, read as $\mathfrak{B}$ *refines* $\mathfrak{B}'$, if for every block $B \in \mathfrak{B}$, we have a block $B' \in \mathfrak{B}'$, such that $B \subseteq B'$. We denote $\mathfrak{B} \prec \mathfrak{B}'$, if $\mathfrak{B} \preceq \mathfrak{B}'$ and $\mathfrak{B} \neq \mathfrak{B}'$. A block system $\mathfrak{B}$ is called a *minimal block system* if there is no non-trivial block system $\mathfrak{B}'$ such that $\mathfrak{B} \prec \mathfrak{B}'$. $\mathfrak{B}$ is minimal iff $\Gamma[\mathfrak{B}]$ is primitive. If $\Delta \leq \Gamma$, then we say that a set $T \subseteq \Gamma$ is a *transversal* for Δ, if $|T| = |\Gamma|/|\Delta|$, and $\{\Delta\delta | \delta \in T\} = \{\Delta\gamma | \gamma \in \Gamma\}$. Intuitively, the transversal of a subgroup is a set that contains one representative of all cosets of the subgroup, and no other element.

Now, we define what we refer to as *Johnson Groups.* Let m and $t \leq \frac{m}{2}$, and denote $\binom{[m]}{t}$ be the set of all t element subsets of the set $[m] = \{1, 2, \ldots, m\}$. Let $S_m^{(t)} \leq \mathrm{Sym}(\binom{[m]}{t})$ be the natural induced action of S_m over $\binom{[m]}{t}$. Similarly, let

$A_m^{(t)} \leq \mathrm{Sym}(\binom{[m]}{t})$, denote the natural induced action of A_m on the set $\binom{[m]}{t}$. We refer to $S_m^{(t)}$ and $A_m^{(t)}$ as Johnson Groups.

2.3 Strings

A *string* is defined as a map $\mathfrak{x} : \Omega \longrightarrow \Sigma$, where Ω and Σ are finite sets. The set Σ is called the *alphabet*. We will be working with permutation groups which are subgroups of the symmetric group on Ω. A permutation γ can be applied on the string $\mathfrak{x}$ in the following manner:

$$\mathfrak{x}^\gamma : \Omega \longrightarrow \Sigma$$
$$\mathfrak{x}^\gamma(\alpha) = \mathfrak{x}\left(\alpha^{\gamma^{-1}}\right)$$

Let $\mathfrak{y}$ be another string, we say that γ is a isomorphism from $\mathfrak{x}$ to $\mathfrak{y}$, if $\mathfrak{x}^\gamma = \mathfrak{y}$. We write this isomorphism as $\gamma : \mathfrak{x} \cong \mathfrak{y}$. We define a $\Gamma-$*isomorphism*, as a permutation $\gamma \in \Gamma$ such that $\gamma : \mathfrak{x} \cong \mathfrak{y}$. The set of such Γ-*isomorphisms* is denoted by $\mathrm{Iso}_\Gamma(\mathfrak{x}, \mathfrak{y})$, such that:

$$\mathrm{Iso}_\Gamma(\mathfrak{x}, \mathfrak{y}) := \{\gamma \in \Gamma | \mathfrak{x}^\gamma = \mathfrak{y}\}$$

An instance of the *String Isomorphism Problem*, is given by two strings $\mathfrak{x}\mathfrak{y} : \Omega \longrightarrow \Sigma$, and a generating set for $\Gamma \leq \mathrm{Sym}(\Omega)$, and it is a *yes* instance if the two strings are Γ-*isomorphic* and a *no* instance otherwise.

The Γ-automorphism group of $\mathfrak{x}$ is denoted as $\mathrm{Aut}_\Gamma(\mathfrak{x}) := \mathrm{Iso}_\Gamma(\mathfrak{x}, \mathfrak{x})$. It is easy to see that for an arbitrary $\gamma \in \mathrm{Iso}_\Gamma(\mathfrak{x}, \mathfrak{y})$, we have:

$$\mathrm{Iso}_\Gamma(\mathfrak{x}, \mathfrak{y}) = \mathrm{Aut}_\Gamma(\mathfrak{x})\gamma$$

3 The WL Algorithm

In this section, we will see some incomplete isomorphism tests. First, we will look at the color refinement algorithm, which is a very simple isomorphism test. We then generalise it to k dimesnsions and state the k dimensional Weisfeiler-Leman Algorithm. The classical WL algorithm, was 2 dimensional, and the k dimensional generalisation came later.

3.1 The Color Refinement Algorithm

Color refinement is more of a combinatorial approach to distinguish non-isomorphic graphs. It relies heavily on the degree and number of neighbours coloured with a colour iteratively.

In colour refinement, initially we assume a uniform colouring of vertices. Then, the colouring assumed is refined iteratively by counting the number of neighbours with each colour and updating the colours.

Let χ_1, χ_2 be two vertex colorings of a graph G. χ_1 *refines* χ_2, denoted by $\chi_1 \preceq \chi_2$, if for all $u, v \in \mathcal{V}(G)$, $\chi_1(u) = \chi_1(v) \implies \chi_2(u) = \chi_2(v)$. We say that the colorings are equivalent, if $\chi_1 \preceq \chi_2$ and $\chi_2 \preceq \chi_1$.

We denote the initial coloring by χ_0, which colors every vertex to be of the same color, and the coloring generated after the i^{th} iteration to be χ_i. The color refinement updates the coloring as follows:

$$\chi_i(v) = (\chi_{i-1}(v), \mathcal{M}_i(v))$$

where $\mathcal{M}_i(v)$ stores the colours of the neighbors in a multiset.

$$\mathcal{M}_i(v) = \{\!\!\{\chi_{i-1}(w) | w \in \mathcal{N}(v)\}\!\!\}$$

The color refinement algorithm refines the coloring in each iteration until a stable coloring is obtained, i.e., two consecutive iterations return equivalent colorings, which happens in at most n iterations. We can then compare the partition of vertices given by the colors. Since this is an incomplete isomorphism test, if two graphs do not have the same color partitions in their stable coloring, then they are not isomorphic. But we cannot infer anything when the partitions match.

The Color Refinement algorithm can be made to run in $\mathcal{O}((n+m)\log n)$ time where $|\mathcal{V}(G)| = n$ and $|\mathcal{E}(G)| = m$[CC82].

The algorithm, despite being very simple, is still a powerful tool. In essence, Babai, Erdos and Selkow [BES80] showed that color refinement algorithm can identify almost all graphs in the limit.

3.2 The k-WL algorithm

The k-dimensional Weisfeiler Leman algorithm is a direct generalisation of the color refinement algorithm. Instead of refining a coloring of vertices, we refine coloring of k-tuples of vertices. We use colorings of graph G that colors k-tuples:

$$\chi^k : \mathcal{V}(G) \longrightarrow C$$

where C is a set of colors.

Initially all the k-tuples are colored with their *atomic types.* $\operatorname{atp}(G, \overline{v})$ for a tuple $\overline{v}$ in the graph G, describes the isomorphic nature of the induced subgraph on the tuple. One way of formally defining atomic type to for a tuple $(v_1, \dots, v_k)$, is by a pair of $k \times k$ Boolean matrices, one of which is an adjacency, and one describes equalities among the elements of the tuple. Defining atomic types like this, it is easy to see that $\operatorname{atp}(G, v_1 \dots v_k) = \operatorname{atp}(H, w_1 \dots w_k)$ for graphs G and H, if and only if f defined as $f(v_i) = w_i$ describes an isomorphism from the induced subgraph of G by $v_1 \dots v_k$ to the induced subgraph of H by $w_1 \dots w_k$.

For k-dimensional WL, initially all the k tuples are colored by their atomic types, i.e., $\chi_0^k(\overline{v}) = \operatorname{atp}(G, \overline{v})$ for all $\overline{v} \in \mathcal{V}(G)^k$. The refinement is done as follows:

$$\chi_i^k(\overline{v}) = \left(\chi_{i-1}^k(\overline{v}), \mathcal{M}_i^k(\overline{v})\right)$$

where $\mathcal{M}_i^k(\overline{v})$ is the generalisation of the multiset from the color refinement algorithm, defined as follows:

$$\mathcal{M}_i^k(\overline{v}) = \{\!\!\{ \left(\chi_{i-1}^k(\overline{v}[v/1]), \dots \chi_{i-1}^k(\overline{v}[v/k])\right) | v \in \mathcal{V}(G) \}\!\!\}$$

where $\overline{v}[v/i]$ represents the tuple $\overline{v}$ in which the i^{th} vertex is substituted by v.

A stable coloring of k tuples can be found in at most n^k refinement iterations. This is called a k-stable coloring and can be computed in $\mathcal{O}(n^k \log n)$ time [IS19].

It is not hard to see that the 1 dimensional version of this essentially is the color refinement algorithm. The k-WL is also an incomplete isomorphism test.

The graphs that are not distinguishable by the k-dimensional Wl algorithm are called k-indistinguishable. It is extremely difficult to construct non-isomorphic graphs that are k-indistinguishable for $k \geq 3$. However, the following result by Cai, Furer and Immerman constructs small graphs for each k.

Theorem 3.1 ([CFI92]). *For all $k \geq 1$ there are non-isomorphic 3-regular graphs G_k and H_k, that are k-indistinguishable, with $|G_k| = |H_k| = \mathcal{O}(k)$.*

4 Basic Permutation Group Algorithms

4.1 Schreier Sims Algorithm

In the domain of efficient permutation group algorithms, the notion of strong generating sets introduced by Sims is very useful. The Schreier-Sims Algorithm is one algorithm for creating a strong generating set. We will first define the notions of base and strong generating sets. The Schreier-Sims algorithm is presented from [Ser03].

A sequence of elements $B = (\beta_1, \dots, \beta_m)$, where $\beta_i \in \Omega \forall i \in [m]$ is called a *base*, if the pointwise stabiliser of all elements G_B contains only one element which is the identity permutation. We define a subgroup chain wrt base B as follows:

$$G = G^{(1)} \geq G^{(2)} \cdots \geq G^{(m+1)} = \{1\}$$

where $G^{(i)}$ is the pointwise stabiliser of elements $\beta_1, \dots \beta_{i-1}$, $G_{(\beta_1, \dots \beta_{i-1})}$. The base is called a non-redundant base if for every $i \in [m]$, $G^{(i+1)}$ is a proper subgroup of $G^{(i)}$.

Next, we define *strong generating sets*. A generating set S is a strong generating set (SGS for short) with respect to the base B if the following holds for all $i \in [m+1]$:

$$\langle S \cap G^{(i)} \rangle = G^{(i)}$$

Since the definition of strong generating set relies on the notion of a base, we write BSGS to refer to a base and a strong generating set for that base. Once we have an SGS, it is easy to compute the orbit $\beta_i^{G^{(i)}}$ using the orbit algorithm from the next section. This additionally gives a right transversal R_i of G_i/G_{i+1} for each $i \in [m]$. We will always require that the representative of the subgroup itself is the identity element 1.

By Lagrange's theorem, $|G| = \prod_{i=1}^{m} |G^{(i)} : G^{(i+1)}| = \prod_{i=1}^{m} |R_i|$. Since each coset in G^i/G^{i+1} (and thus each representative in R_i) corresponds to a point in the orbit $\beta_i^{G^{(i)}}$, $|\beta_i^{G^{(i)}}| = |R_i|$. This gives an efficient way of computing $|G|$ once we have a BSGS and corresponding right transversals. Moreover each element $g \in G$ can be written uniquely as $g = r_m r_{m-1} \dots r_2 r_1$ for some $r_i \in R_i$. These r_i's can be found by the following procedure called *sifting*.

The *sifting* procedure computes the r_i's for a given permutation σ in the following manner. Given a permutation σ, and transversals R_i, and base B, we can find the $r_i \in R_i$, such that $\sigma = r_m \dots r_1$. Let β^γ be the image of β under the permutation γ. Then, we start with computing the coset representative $r_r \in R_1$ such that $\beta_1^{r_1} = \beta_1^{\sigma}$. Then compute $\sigma_2 = \sigma r_1^{-1}$. Again, compute $r_2 \in R_2$ such that $\beta_2^{r_2} = \beta_2^{\sigma_2}$, and compute $\sigma_3 = \sigma_2 r_2^{-1}$ and so on. In the end, we have

$\sigma_{m+1} = \sigma r_1^{-1} r_2^{-1} \dots r_m^{-1}$ =id. The computed $r_1, \dots r_m$ along the way are the required coset representatives.

This procedure can also be easily modified to give a group membership algorithm. A permutation $h \in \mathrm{Sym}(\Omega)$ is a member of G if and only if the sifting procedure is able to find r_i, $i \in [m]$ and $gr_1^{-1} r_2^{-1} \dots r_m^{-1} = 1$, the identity permutation. The *siftee* of h is defined using the largest index $i \leq m+1$, that can be computed during sifting, $h_i = hr_1^{-1} \dots r_{i-1}^{-1}$ is called the siftee if we cannot find a suitable $r_i \in R_i$. If $h \in G$, then the siftee is identity.

Now we describe the Schreier-Sims algorithm which can find a base and strong generating set for a given permutation group (provided as a set of generators) in polynomial time. The following lemma due to Schreier gives a set of generators of a subgroup H of G if we have a right transversal R for the cosets and generators S of G. For a group element $g \in G$, $\bar{g} \in R$ is such that $g \in Hr$, i.e, $gr^{-1} \in H$.

Lemma 4.1. *Let $H \leq G = \langle S \rangle$ and R be a right transversal of G/H containing 1. Then the following set T generates H.*

$$T = \{rs(\overline{rs})^{-1} \mid r \in R, s \in S\}$$

Proof. It is clear that $T \subseteq H$ since for every $g \in G$, $g\bar{g}^{-1} \in H$.

So we only need to show that for every element $h \in H$, $h = t_1 t_2 \dots t_k$ for some $k \in \mathbb{N}$ and some $t_i \in T$ for all $i \in [k]$. Let $h \in H$. Since S generates G, $h = s_1 s_2 \dots s_l$ for some s_i's in S. We will inductively rewrite the expression so that at the end we use only elements from T.

More precisely, for every $i \in [l+1]$, we will define an element $h_i = t_1 t_2 \dots t_{i-1} r_i s_i s_{i+1} \dots s_l$, (where t_j's are in T, $r_i \in R$ and s_j's in S) such that $h_i = h_{i-1}$. The element h_i is $1 s_1 s_2 \dots s_l$ which is clearly equal to h. By induction, we will thus have $h_i = h$ for all i. For the induction step, suppose we have h_i for $1 \leq i \leq l$, then

$$\begin{aligned} h_{i+1} &= t_1 t_2 \dots t_{i-1} r_i s_i s_{i+1} \dots s_l \\ &= t_1 t_2 \dots t_{i-1} (r_i s_i (\overline{r_i s_i})^{-1}) \overline{r_i s_i} s_{i+1} \dots s_l \\ &= t_1 t_2 \dots t_{i-1} t_i r_{i+1} s_{i+1} \dots s_l \end{aligned}$$

where $t_i = r_i s_i (\overline{r_i s_i})^{-1}$ and $r_{i+1} = \overline{r_i s_i}$.

Finally $h_{l+1} = t_1 t_2 \dots t_l r$ for some t_i's in T and $r \in R$. Since $h_{l+1} = h \in H$ and $t_1 t_2 \dots t_l \in H$, $h(t_1 t_2 \dots t_l)^{-1} \in H$. As the representative of H in R is 1, we deduce $r = 1$. Therefore $h = t_1 t_2 \dots t_l$. □

The following lemma gives a criterion for recognizing a strong generating set.

Lemma 4.2. *Let $\{\beta_1, \dots, \beta_k\} \subseteq \Omega$ and $G \subseteq Sym(\Omega)$. For $1 \leq j \leq k+1$, let $S_j \subseteq G_{(\beta_1, \beta_2, \dots, \beta_{j-1})}$ and for $j \leq k$, $\langle S_j \rangle \geq \langle S_{j+1} \rangle$ holds. Also suppose $S_{k+1} = \emptyset$, $G = \langle S_1 \rangle$ and for all $1 \leq j \leq k$,*

$$\langle S_j \rangle_{\beta_j} = \langle S_{j+1} \rangle.$$

Then $(\beta_1, \ldots, \beta_k)$ is a base for G and $S = \cup_{j\in[n]} S_j$ is a strong generating set for G relative to the base $B = (\beta_1, \ldots, \beta_k)$.

Proof. The proof is by induction on k. If $k = 0$, the statement is trivial since in this case $G = \langle\emptyset\rangle = \{1\}$.

For the induction step, suppose the statement is true for $k - 1 \geq 0$. We are given $\{\beta_1, \beta_2, \ldots, \beta_k\}$ and S_j's as described in the statement. By the induction hypothesis, $S' = \cup_{j=2}^{k} S_j$ is an SGS for $\langle S_2\rangle$ relative to the base $B' = (\beta_2, \ldots, \beta_k)$. Define $G^{(i)} = G_{(\beta_1,\ldots,\beta_{i-1})}$ as done while defining strong generating sets. Now we need to verify that for $j \in [k]$, $\langle G^{(j)} \cap S\rangle = G^{(j)}$.

For $j = 1$, this holds since $G = \langle S_1\rangle$ and $G^{(1)} = G = \langle S_1\rangle \leq \langle S \cap G^{(1)}\rangle$. The other direction is obvious.

For $j = 2$, $G^{(2)} = G_{(\beta_1)} = \langle S_1\rangle_{(\beta_1)} = \langle S_2\rangle \leq \langle S \cap G^{(2)}\rangle$ since $S_2 \subseteq G^{(2)}$.

For $j > 2$, we shall make use of the induction hypothesis. Now $G^{(j)} = (G_{\beta_1})_{(\beta_2,\ldots,\beta_{j-1})} = \langle S_2\rangle_{(\beta_2,\ldots,\beta_{j-1})} = \langle S' \cap G^{(j)}\rangle \leq \langle S \cap G^{(j)}\rangle$.

This finishes the proof. □

The Schreier-Sims algorithm proceeds by maintaining a partial base $(\beta_1, \beta_2, \ldots, \beta_i)$ and associated sets S_j, $1 \leq j \leq i$. Throughout we shall maintain the following assumptions of Lemma 4.2. $S_j \in G_{(\beta_1,\ldots,\beta_{j-1})}$ for all $j \in [i]$ and $\langle S_j\rangle \geq \langle S_{j+1}\rangle$. S_1 will be the input generating set T of G. S_{i+1} will implicitly be the empty set. We say that the structure is up to date below level k, if for all $j > k$, $\langle S_j\rangle_{\beta_j} = \langle S_{j+1}\rangle$. The main aim of the algorithm is to ensure that the structure becomes up to date below level 0. Once this happens, Lemma 4.2 tells us that the union of the S_j's is an SGS with respect to the base $(\beta_1, \beta_2, \ldots, \beta_i)$.

In the beginning, β_1 is chosen to be any element of Ω which is not fixed by all of G, i.e. there is some generator in the input generating set U of G which moves the element to some other element. At this point, the structure is up to date below level 1.

Now suppose the structure is up to date below level k. By using Lemma 4.2, we have an SGS for $\langle S_{k+1}\rangle$ with respect to the base $(\beta_{k+1}, \ldots, \beta_i)$. The algorithm checks now if the structure is also up to date below level $k - 1$. It only needs to check if $\langle S_k\rangle_{\beta_k} \leq \langle S_{k+1}\rangle$ since $\langle S_k\rangle_{\beta_k} \geq \langle S_{k+1}\rangle$ holds by the invariants of the algorithm. For checking if S_{k+1} generates $\langle S_k\rangle_{\beta_k}$, it is sufficient to check if a set of generators of $\langle S_k\rangle_{\beta_k}$ is generated by S_{k+1}. We consider the Schreier generators of Lemma 4.1. We can do this since a transversal of $\langle S_k\rangle_{\beta_k}$ in $\langle S_k\rangle$ can be easily computed by the orbit and transversal computation of the next subsection. For each such Schreier generator, the algorithm checks if it lies in $\langle S_{k+1}\rangle$ by sifting since we have a BSGS for $\langle S_{k+1}\rangle$. If each Schreier generator is in $\langle S_{k+1}\rangle$, then we have verified that the structure is up to date below level $k - 1$.

On the other hand, if we find a Schreier generator which does not lie in $\langle S_{k+1}\rangle$, we add a non-trivial siftee to S_{k+1}. At this point, the structure is still up to date below level $k+1$. Now the algorithm checks if the structure is up to date below level k and continues in this manner until the structure is up to date below level 0. When $k = i$, we also pick β_{k+1} to be any point not fixed by the Schreier generator.

For the above algorithm to be efficient, we must not do any recomputation, such as checking Schreier generators that have previously been checked, creating the entire transversal from scratch, etc. Though there are several improvements that can be made to further speed up the above algorithm, it still runs in polynomial time. Now we analyze the time complexity of the above version of Schreier-Sims.

We shall need the following simple estimates on the size of B and the size of each S_i. Since only points which are moved are picked as base points, there can be at most $\lg |G|$ base points finally since for each base point, the size of the corresponding transversal is at least 2. Similarly since $S_i \leq G$ and each generator of S_i at least doubles the size of the subgroup generated by the previously added generators by Lagrange's theorem, $|S_i| \leq \lg |G|$.

The image computation for a base point takes $O(n)$ time when a new generator is added. The set S_k changes at most $\lg |G|$ times for $k > 1$. So the image computation for all base points in the entire algorithm takes $\mathcal{O}(|B| \lg |G| n + |T| n)$ time. The transversal computation for the orbit of a base point requires at most $n \times O(n)$ time since each permutation multiplication takes $O(n)$ time and the size of an orbit is at most n. Over all base points, $O(n^2 \lg |G|)$ time is spent in transversal computation.

The sifting of Schreier generators takes the most time. One sifting requires at most $\mathcal{O}(\lg |G|)$ permutation multiplications since $|B| \leq \lg |G|$. The total number of Schreier generators across all base points is $\sum_k |R_k||S_k| = \mathcal{O}(n \lg^2 |G| + |T| n)$ since each $|R_k| \leq n$ for all k and $|S_k| \leq \lg |G|$ for all $k > 1$. The total cost of the Schreier-Sims algorithm is therefore $\mathcal{O}(n^2 \lg^3 |G| + |T| n^2 \lg |G|)$.

4.2 Other subroutines

We also use some other subroutines in the algorithm we present for string isomorphism. We try to sketch them below for completeness. These are not the most efficient approaches but are conceptually simple.

Computing Orbits

We are given a set S, as a generator set for a permutation group such that $\langle S \rangle = G$ and an element $\alpha \in \Omega$. We need to find the orbit of α in the group G.

We construct a graph G, whose vertices denote the elements in the domain Ω. For every element δ in the generator set, we add an edge (β, β^δ), for all $\beta \in \Omega$.

After this construction, the problem essentially is a reachability problem. It is not hard to see that all the elements reachable from α in the graph G will be the orbit. We can use any graph reachability algorithm like depth-first search to compute the orbit.

We can label each edge by the permutation γ it is derived from. Now, while computing the orbit, if we discover a previously undiscovered vertex β from a vertex δ, via an edge labelled by σ, we call $\sigma\sigma'$ the *representative* of β, where σ' is the representative of δ. The base case being the representative of α being the identity permutation.

The graph construction will only take polynomial time, since for each permutation in S, we're adding at most n edges. The constructed graph will also have edges at most polynomially bounded, and we know that DFS runs in linear time. Hence, we can compute the orbit in polynomial time.

Compute a minimal block system

Given a set of generators S, we want to find a minimal block system for the group generated by them. Let S be the generator of the group G. The basic idea here is to find some block containing at least two elements from the domain.

Let us fix an element α from domain Ω. Now, pick $\beta \neq \alpha$ from the domain, and compute the orbit of $\{\alpha, \beta\}$ under the natural action of G on the unordered pairs of elements from Ω. Now, construct a graph with vertices as the domain Ω, such that (u, v) is an edge if $\{u, v\}$ was in the computed orbit of $\{\alpha, \beta\}$. The connected component containing α and β, is the smallest block that contains both α and β. If the graph is connected, then this is a trivial block, in which case, we start with another β not checked before. If this happens for all β, then the group is primitive, else, the connected components will form a block system of the group generated by the given generating set. Let us denote this block system by $\mathfrak{B} = \{B_1, \ldots, B_k\}$. Now, to get a minimal block system, we recursively call the minimal block system procedure on the block system, restricting the domain and the group action to blocks. At the end, since we have already checked if the group is primitive or not, we can obtain the minimal block system, by expanding all the sets in the returned system. For example, if the returned set was $\mathfrak{B}' = \{B'_1, \ldots, B'_l\}$, on the domain $\mathfrak{B}$, to get the minimal block system on the domain Ω, we just expand all the sets in the blocks B'_i.

To get the idea of this being a polynomial time procedure, it is easy to see all the computation is in polynomial time. Now, we want the number of recursive calls to be polynomial. To see that, we can easily see that the domain size always reduces in each recursive call, since the number of blocks in a non singleton block system is less than the number of elements in the domain. Thus the domain size strictly decreases in each recursive call, and hence the number of calls is also polynomial in the domain size.

Pointwise Stabiliser

This is a simple application of the Schreier-Sims approach. Let G be a permutation group with generating set T, $\alpha \in \Omega$ and suppose we have to compute G_α. If each element of T (and thus G) fixes α, then $G_\alpha = G$. So suppose G does not fix α. Now in the Schreier-Sims algorithm we pick α to be the first base point and then proceed as usual. Finally if the sets $S_1 = T, S_2, \ldots$ of generators are obtained, S_2 generates the stabiliser of α and $\cup_{i \geq 2} S_i$ is an SGS for G_α with base $(\beta_2, \beta_2, \ldots)$.

This idea easily generalizes to the pointwise stabilizer of a set of points. To do this, the elements of the set are made the initial base points and then we proceed as usual.

Setwise stabiliser of a block system

We define the setwise stabiliser of a block system as the subgroup $\Gamma_{(\mathfrak{B})}$, which stabilises all the blocks setwise.

Now, given a set of blocks $\mathfrak{B}$, a generating set S for the group G, we want to compute the generator set for the group that stabilises all the blocks in the set $\mathfrak{B}$.

Now, until we have only one block left in the set $\mathfrak{B}$, we do the following iterative procedure:

Pick a block B from the set $\mathfrak{B}$, and pick an $\alpha \in B$. Compute the generating set S_1, for the stabiliser of a. Let X be the set $\alpha^{\langle S \rangle} \cap B$, where $\alpha^{\langle S \rangle}$ is the orbit of α in $\langle S \rangle$. Let S_2 be the set of representatives for elements in the set X. We can remove the identity from this set, since we are considering generators. Now, set $S = S_1 \cup S_2$, and remove the block B from $\mathfrak{B}$.

In the end, we return S, as it remains at the end of all iterations. It is also not hard to see that the running time of this algorithm is polynomial.

The idea behind this procedure, is that for any element in the setwise stabiliser of blocks, it fixes all the blocks. So for all the blocks, and for any element in the block, either a permutation can keep it fixed, or map it to other elements in the block. Hence, our iterative procedure, constructs the generators by first adding the generators fixing an element in a block, and then adding permutations which are representatives and map the element to other elements inside the block.

Homomorphisms

A homomorphism $\varphi : G \to \mathrm{Sym}(\Delta)$ on a permutation group $G = \langle S \rangle \leq \mathrm{Sym}(\Omega)$ is given by the images $\varphi(s)$ for $s \in S$. For working efficiently with this homomorphism, we will consider G as a permutation group on $\Omega \cup \Delta$ (where Ω and Δ are disjoint). Specifically for an element $g \in G$, we will store the permutation $(g, \varphi(g))$ acting as g on Ω and as $\varphi(g)$ on Δ. Let G' denote the corresponding permutation group. Now the kernel of φ is precisely the pointwise stabilizer $G'_{(\Delta)}$.

To efficiently find images of elements in G, we shall keep a base $B_1 = (\beta_1, \beta_2, \ldots, \beta_m)$ and a corresponding strong generating set S_1, such that B_1 contains points only in Ω. Then to compute $\varphi(g)$, sift $(g, 1)$ to get $(1, h)$ as the siftee for some $h \in \mathrm{Sym}(\Delta)$. Then $\varphi(g) = h^{-1}$.

Similarly, if we want a preimage of $h \in \phi(G)$ or want to check membership in $\varphi(G)$, we keep a base $B_2 = (\beta_1, \beta_2, \ldots, \beta_m)$ and a corresponding strong generating set S_2, such that there is a $1 \leq k \leq m$ where $(\beta_1, \beta_2, \ldots, \beta_k)$ contains points only in Δ and the remaining base points lie in Ω. To check membership of $h \in \phi(G)$, sift $(1, h)$ until we have gone over all base points up to β_k. If the resulting siftee is of the form $(g, 1)$, accept, otherwise reject. For finding a preimage, we follow the same sifting procedure and a preimage of h is given by g^{-1}. The coset of all preimages is $\ker(\varphi)g^{-1}$.

5 Reduction to String Automorphism

We will describe a reduction for the graph isomorphism problem for connected trivalent case, to the problem of string automorphism. To do this, we will first reduce the isomorphism problem for connected trivalent graphs, to the problem of finding automorphisms of connected trivalent graphs with a distinguished edge. Let X be a connected trivalent graph, then, $\mathrm{Aut}_e(X)$, is the automorphism group of the graph X that preserves the edge e. Now, we have the following proposition:

Proposition 5.1 ([Luk82]). *Testing isomorphism of two connected trivalent graphs is poly-time reducible to the problem of determining generators for $Aut_e(X)$, where X is a graph and e is a distinguished edge.*

Proof. Let X_1, X_2 be two connected trivalent graphs. Now, we will fix an edge in the graph X_1. Let this be e_1. Now, we iterate over each edge in the graph X_2. Let e_2 be any edge of X_2. Now, we want to test if there is an isomorphism of X_1 and X_2 that maps edge e_1 to e_2. If we can do this for each e_2, we can determine if the graphs X_1 and X_2 are isomorphic. Now, to test whether there is an isomorphism from X_1 to X_2, which maps e_1 to e_2, we construct a new graph X. The new graph X, has the vertex set $\mathcal{V}(X_1) \cup \mathcal{V}(X_2) \cup \{v_1, v_2\}$, where v_1 and v_2 are new vertices inserted in the edges e_1 and e_2 respectively. Now, to finally construct the graph, we join v_1 and v_2 by an edge e. It is easy to see that X is connected and trivalent. Thus, the graphs X_1 and X_2 have an isomorphism that maps e_1 to e_2, if the group $\mathrm{Aut}_e(X)$ has an automorphism swapping v_1 and v_2. It is also trivial that if such an element in the automorphism group fixing e exists, then any set of generators of this group $\mathrm{Aut}_e(X)$, will contain an element that flips v_1 and v_2. □

Since we have the above reduction, it is sufficient to show a reduction for the problem of finding a generating set of $\mathrm{Aut}_e(X)$, for a connected trivalent graph X, and a distinguished edge e to the string isomorphism problem.

From here onward, we will work with a connected trivalent graph X with n vertices, and a distinguished edge e, for the reduction.

Now, we will define graphs X_r. X_r is the graph that is formed by edges and vertices of X, which are present in paths of length at most r, that pass through the edge e. Thus, X_1 is just the graph edge e and X_{n-1} is just X. Note that X_r is not the induced subgraph, in the manner that if some two vertices v_1 and v_2 are in X_r, and if there is an edge between them, it is not necessarily in the graph X_r.

Now, we will try to build an automorphism group for X_r's inductively. Let π_r be the induced homomorphism defined as follows:

$$\pi_r : \mathrm{Aut}_e(X_{r+1}) \longrightarrow \mathrm{Aut}_e(X_r)$$

where, for any $\sigma \in \mathrm{Aut}_e(X_{r+1})$, $\pi_r(\sigma)$ is just the restriction of σ to the smaller graph X_r. Now, this induced homomorphism helps us determine $\mathrm{Aut}_e(X_{r+1})$, if we can determine the following efficiently:

- $\mathcal{R}$ such that $\langle \mathcal{R} \rangle = K_r$, the kernel of π_r
- $\mathcal{S}$ such that $\langle \mathcal{S} \rangle = \pi_r(\mathrm{Aut}_e(X_{r+1}))$, the image of π

If we can do this, then we can determine a set $\mathcal{S}'$, such that $\pi_r(\mathcal{S}') = \mathcal{S}$, then we have $\langle \mathcal{R} \cup \mathcal{S}' \rangle = \mathrm{Aut}_e(X_{r+1})$. We will see that computing the generator for image, will eventually reduce to the string isomorphism problem. Let us define the set $V_r = \mathcal{V}(X_{r+1}) \setminus \mathcal{V}(X_r)$, to be the new vertices introduced in X_{r+1}. Now, since the graph is trivalent, it is easy to argue that any vertex in V_r will have at least one and at most three neighbours in $\mathcal{V}(X_r)$. Let us define a set A in the following manner:

$$A = \binom{\mathcal{V}(X_r)}{1} \cup \binom{\mathcal{V}(X_r)}{2} \cup \binom{\mathcal{V}(X_r)}{3}$$

which is essentially the set of subsets of $\mathcal{V}(X_r)$, of sizes one, two and three.

Now, we can define a neighbour function f as follows:

$$\begin{aligned} f :& V_r \longrightarrow A, \\ f(v) =& \{u \in \mathcal{V}(X_r) | \{u, v\} \in \mathcal{E}(X_{r+1})\} \end{aligned}$$

Now, to compute the kernel of π_r, we observe that for any $\sigma \in K_r$, we have that $f(v) = f(\sigma(v))$. This is because since σ is in the kernel, it should fix all vertices in X_r. Since σ is an automorphism on X_{r+1}, mapped vertices must have the same neighbours, because the f function gives the set of neighbours in X_r but all those vertices are fixed in σ. We observe that triplets of vertices v, v', v'' such that $f(v) = f(v') = f(v'')$ cannot exist. This is because,if such a triplet exists, then there is a vertex u in X_r that connects to v, v', v'', making the degree more than 3, since u will have at least one edge in X_r. Thus, at most, pairs of vertices with the same neighbours can exist. By the above two arguments, it is easy to infer that K_r would be the group generated by transpositions that contains two distinct vertices in V_r and have the same neighbour set. Thus this kernel computation can be done in polynomial time by just taking pairs of vertices in V_r, and comparing their neighbour sets.

Next we show how to compute the image of π. We start with some observations. Let $S \in A$ and $\sigma \in \mathrm{Aut}_e(X_{r+1})$. Then $|f^{-1}(S)| = |f^{-1}(\sigma(S))|$ since $\sigma \in \mathrm{Aut}_e(X_{r+1})$. Put another way, only the sets in A having preimages of the same size can be permuted by σ. Now let $x, y \in \mathcal{V}(X_r)$ and $\{x, y\} \in \mathcal{E}(X_{r+1}) \setminus \mathcal{E}(X_r)$. Then $\{\sigma(x), \sigma(y)\} \in \mathcal{E}(X_{r+1}) \setminus \mathcal{E}(X_r)$.

Interestingly these two simple conditions are also sufficient for an automorphism of X_r to lie in the image of π as is explained now. Let $\sigma : \mathcal{V}(X_r) \to \mathcal{V}(X_r)$ be an automorphism of X_r satisfying the following properties. For every $S \in A$, $|f^{-1}(S)| = |f^{-1}(\sigma(S))|$. For any $x, y \in \mathcal{V}(X_r)$, $\{x, y\} \in \mathcal{E}(X_{r+1}) \setminus \mathcal{E}(X_r)$ if and only if $\{\sigma(x), \sigma(y)\} \in \mathcal{E}(X_{r+1}) \setminus \mathcal{E}(X_r)$. To extend σ to an automorphism σ' of X_{r+1}, we need to find suitable images of vertices in V_r. The first condition readily allows us to do this since $\{f^{-1}(S) \mid S \in A\}$ forms a partition of V_r and we can map $f^{-1}(S)$ to $f^{-1}(\sigma(S))$ bijectively for each $S \in A$ with $f^{-1}(S)$ non-empty. From here, it is clear that the new edges between vertices in V_r and those in $\mathcal{V}(X_r)$ are preserved by σ'. The second property is required to ensure that all the new edges with both vertices in X_r are already preserved by σ.

So we only need to find the subgroup of automorphisms of X_r satisfying these two properties. This can be encoded as a string automorphism problem in the following way. The domain Ω is A. The alphabet Σ is $\{(i, b) \mid i \in \{0, 1, 2\}, b \in \{0, 1\}\}$. The string $s : A \to \Sigma$ is defined in the following way. For $T \in A$, $s(T) = (|f^{-1}(S)|, b)$ where $b = 1$ if $T \in \mathcal{E}(X_{r+1}) \setminus \mathcal{E}(X_r)$ and $b = 0$ otherwise. The group $G \leq \mathrm{Sym}(\Omega)$ is given by the induced action of $\mathrm{Aut}_e(X_r)$ on A. The subgroup $\langle \mathcal{S} \rangle H \leq G$ obtained as output from a string isomorphism subroutine can be easily converted to a subgroup of $\mathrm{Sym}(\mathcal{V}(X_r))$ by considering the restriction of each permutation to $\binom{\mathcal{V}(X_r)}{1}$. A pullback $\mathcal{S}'$ can be constructed by extending each permutation in $\mathcal{S}$ in the manner described previously.

In this way, we have shown how $\mathrm{Aut}_e(X_{r+1})$ can be computed using string isomorphism in the trivalent case. The above reduction easily generalizes to degree $d > 3$. We sketch the changes required for the general constant degree d case now. The kernel computation remains the same. Here a set S in $\mathcal{V}(X_r)$ of size at most d can be the neighbour set of at most $d - 1$ vertices in V_r and these vertices in V_r with the same neighbour set can be permuted arbitrarily. During the image computation, the string isomorphism instance has an alphabet of $2d$ size since for any $S \in \bigcup_{i=0}^{d-1} \binom{\mathcal{V}(X_r)}{i}$, $0 \leq |f^{-1}(S)| \leq d - 1$.

5.1 Groups arising in the reduction to String Isomorphism

We now look at the structure of the groups arising in the reduction discussed above, since the structure of these groups is crucial to the polynomial running time of the string isomorphism algorithm described in the next section.

First we discuss the simpler case of trivalent graphs. As argued earlier, the kernel K_r is generated by transpositions on disjoint pairs of vertices. So $K_r \cong \mathbb{Z}_2^k$ for some k. We claim that $\mathrm{Aut}_e(X_r)$ is an elementary abelian 2-group for each r. We prove this by induction on r. For $r = 1$, this is clear since $\mathrm{Aut}_e(X_1)$ is the group generated by the transposition swapping the ends of e. By induction, $\mathrm{Aut}_e(X_r)$ is an elementary abelian 2-group and so is the image of π_r since the image is a subgroup of $\mathrm{Aut}_e(X_r)$. Hence $\mathrm{Aut}_e(X_{r+1})$ is also an elementary abelian 2-group. The following lemma describes the primitive p-groups, and, in particular, primitive 2-groups.

Lemma 5.2. *Let P be a primitive p-subgroup of $Sym(\Omega)$. Then P is a cyclic group of order p acting on Ω of cardinality p.*

Proof. Let p^k be the order of P. We need to show that $k = 1$.

Fix an element $a \in \Omega$. Then P_a is a maximal subgroup of P. If not, then we $P_a \leq p^{k-2}$ and by the Sylow theorems, we can find $H < P$ such that $P_a < H$ and $|H| = p^{k-1}$. H cannot be transitive since if it were, we would be able to write every element of P as a product of an element from P_a and an element from H. So H cannot be transitive.

Consider the orbit a^H of a under H. We claim that $S = a^H$ is a block of imprimitivity for P. Let $g_1, g_2 \in P$ and suppose $S^{g_1} \cap S^{g_2} \neq \emptyset$. Suppose $b^{g_1} = c^{g_2}$ for some $b, c \in S$. Since $S = a^H$, suppose $a^{h_1} = b$ and $a^{h_2} = c$ for some $h_1, h_2 \in H$. Then $h_1 g_1 (h_2 g_2)^{-1} = h_1 g_1 g_2^{-1} h_2^{-1} \in P_a \subsetneq H$ implying $g_1 g_2^{-1} \in H$. Now consider any element $d \in S$. Then $d^{g_1} = d^{g_1 g_2^{-1} g_2}$. Since $d \in S$ and $g_1 g_2^{-1} \in H$, $d^{g_1 g_2^{-1}} \in S$ and thus $d^{g_1} \in S^{g_2}$. This shows that $S^{g_1} = S^{g_2}$. Hence S is a block of imprimitivity. But this contradicts the assumption that P is primitive. So our assumption that P_a is not a maximal subgroup is wrong.

Now that we have shown that P_a is a maximal subgroup of size p^{k-1}, we only need to show that the stabilizer P_a is trivial. By the orbit stabilizer theorem, $|\Omega| = p$. Any permutation in P_a can only have cycles of length less than p since a is fixed and its order must be a power of p. So the only possibility is that it must be the identity. This proves that $k = 1$ and hence $|P| = p$. Since p is prime, the only group of order p is the cyclic group. □

The lemma implies that a primitive 2-group just acts on 2 points.

Now we discuss the general case for degree d. The following definition describes the groups involved. The set Γ_d contains the groups whose composition factors are subgroups of S_d. We will show in the following lemmas (from [Luk82]) that for each r, $\mathrm{Aut}_e(X_r)$ lies in Γ_{d-1}.

Lemma 5.3. *If $N \trianglelefteq G$ and $G/N, N \in \Gamma_d$, then $G \in \Gamma_d$.*

Proof. Let $N \rhd N_1 \rhd \cdots \rhd N_k = \{1\}$ be a composition series of N. Let $G/N \rhd H_1/N \rhd H_2/N \rhd \cdots \rhd H_l/N = \{N\}$ be a composition series of G/N. Then we can combine the composition series to get $G \rhd H_1 \rhd \ldots H_{l-1} \rhd N \rhd N_1 \rhd \cdots \rhd N_{k-1} \rhd \{1\}$. The composition factors of G are given by the union of the composition factors of N and G/N since $(H_i/N)/(H_{i+1}/N) \cong H_i/H_{i+1}$ by the third isomorphism theorem. Since $G/N, N \in \Gamma_d$, this implies $G \in \Gamma_d$. □

The converse of the above lemma follows by essentially reversing the steps of the proof, starting from a composition series of G containing N and then extracting a composition series for N and a composition series for G/N.

The lemma suggests the following inductive approach (similar to the trivalent case) for showing that $\mathrm{Aut}_e(X_{r+1}) \in \Gamma_{d-1}$. For the induction step, it is sufficient to show that the kernel and image lie in Γ_d. The next two lemmas imply these statements.

Lemma 5.4. *The subgroups of S_d lie in Γ_d.*

Proof. Let $G \leq S_d$. Suppose N is a non-trivial normal subgroup of G and G/N is simple. By the previous lemma, it is sufficient to show that G/N is a subgroup of S_d since induction on the length of composition series gives $N \in \Gamma_d$. Let $G^{(i)} = G_{(1,2,\ldots,i-1)}$ for $1 \leq i \leq d$. Consider the following subgroup chain.

$$G^{(d)}N \leq G^{(d-1)}N \leq \cdots \leq G^{(1)}N$$

Note that $G^{(1)}N = G$ and $G^{(d)} = N$. By using the facts that $|G^{(i)}N/N| = |G^{(i)}/(G^{(i)} \cap N)|$ and $G^{(i+1)} \leq G^{(i)}$,

$$\frac{|G^{(i)}N|}{|G^{(i+1)}N|} = \frac{|G^{(i)}||N|}{|G^{(i)} \cap N|} \cdot \frac{|G^{(i+1)} \cap N|}{|G^{(i+1)}||N|} \leq \frac{|G^{(i)}|}{|G^{(i+1)}|} \leq d - (i-1) \leq d$$

where the second last inequality uses the fact that the cosets of $G^{(i+1)}$ in $G^{(i)}$ correspond to the images of i under $G^{(i)}$. Since N is a non-trivial subgroup of G, there is some $i \in [d-1]$ such that $G^{(i+1)}N \lneq G^{(i)}N = GN = G$. Then G acts on the set C containing cosets of $G^{(i+1)}N$ in $G^{(i)}N = G$ in the natural way. Note that N acts trivially on C since it is normal in G. Therefore this induces an action of G/N on C. Since $|C| > 1$ by the choice of i, this action of G/N is non-trivial since it is transitive. Since G/N is simple, the kernel of this action must therefore be trivial. So the action is faithful, which proves that $G/N \leq S_d$. □

The above lemma implies that the kernel of π_r is in Γ_{d-1} as we explain now. The computation of generators for the kernel shows that $K_r \cong S_{i_1} \times S_{i_2} \times \cdots \times S_{i_k}$ for some k where each i_k satisfies $i_k \leq d-1$. For $j \in [k]$, define the group $K_r^j = S_{i_j} \times S_{i_{j+1}} \times \cdots \times S_{i_k}$. Then

$$\{1\} \lhd K_r^k \lhd K_r^{k-1} \lhd \cdots \lhd K_r^2 \lhd K_r^1 = K_r$$

is a subnormal series of K_r, where each factor group K_r^{i+1}/K_r^i is S_{i_j} for some $j \in [k]$. By the lemma, the composition factors of each S_{i_j} are subgroups of S_{d-1}. Combining the composition series for each of the S_{i_j} using the above subnormal series gives a composition series of K_r showing that $K_r \in \Gamma_{d-1}$.

Lemma 5.5. *If $G \in \Gamma_d$ and $H \leq G$, then $H \in \Gamma_d$.*

Proof. Consider the following composition series of G

$$\{1\} \lhd G_k \lhd \ldots G_2 \lhd G_1$$

By taking intersection with H, we get the following subnormal series of H,

$$\{1\} \lhd G_k \cap H \lhd \ldots G_2 \cap H \lhd G_1 \cap H = H$$

The multiset of composition factors of H will be the union of the composition factors of the factor groups $G_i \cap H/G_{i+1} \cap H$. If we can show that each of these factor groups lies in Γ_d, we will be done. Since $G_i \cap H/G_{i+1} \cap H \cong G_i/G_{i+1}$ and $G \in \Gamma_d$, $G_i \cap H/G_{i+1} \cap H$ is a subgroup of Γ_d. This finishes the proof of the lemma. □

Since the image of a homomorphism is a subgroup, the above lemma implies that the image of π_r is in Γ_{d-1} as $\mathrm{Aut}_e(X_r) \in \Gamma_{d-1}$ by induction. Combining this with what was shown for the kernel and using Lemma 5.3, $\mathrm{Aut}_e(X_{r+1}) \in \Gamma_{d-1}$. Lemma 5.3 also implies that the induced action of a permutation group in Γ_{d-1} on a block system also belongs to Γ_{d-1} since the blockwise stabilizer is a normal subgroup.

6 Luks' Algorithm for String Isomorphism

Luks designed a recursive algorithm for the String Isomorphism problem. The algorithm presented by Luks [Luk82], uses two kinds of recursions. We will present the algorithm as presented by Grohe and Neuen [GN20].

Algorithm 1 Luks' Algorithm : StrIso($\mathfrak{x}, \mathfrak{y}, G, \gamma, W$)

```
1: if $\gamma \neq$ id permutation then
2:     return StrIso($\mathfrak{x}, \mathfrak{y}^{\gamma^{-1}}, G, \mathrm{id}, W)\gamma$
3: end if
4: if $|W| == 1$ then
5:     if $\mathfrak{x}(\alpha) == \mathfrak{y}(\alpha)$ for $\alpha \in W$ then
6:         return $G$
7:     else
8:         return $\phi$
9:     end if
10: end if
11: Pick $\alpha$ from $W$
12: $W' \leftarrow$ orbit of $\alpha$
13: if $W \neq W'$ then
14:     return StrIso($\mathfrak{x}, \mathfrak{y}$, StrIso($\mathfrak{x}, \mathfrak{y}, G, \mathrm{id}, W'), W \setminus W'$)
15: end if
16: Compute minimal block system $\mathfrak{B}$ of action of $G$ on $W$
17: Compute $\Delta = G_{(\mathfrak{B})}$, subgroup that stabilises all the blocks
18: Compute transversal $T$ of $\Delta$ in $G$
19: return $\cup_{\gamma \in T}$StrIso($\mathfrak{x}, \mathfrak{y}, \Delta, \gamma, W$)
```

Algorithm 1, is the Luks' string Isomorphism algorithm. Let us define some notation to better understand the logic behind the two recursions used by Luks. Let $\mathfrak{x}, \mathfrak{y}$ be two strings $\mathfrak{x}, \mathfrak{y} : \Omega \longrightarrow \Sigma$. Let $K \subseteq \mathrm{Sym}(\Omega)$ be a set of permutations and $W \subseteq \Omega$ be a window, then we define:

$$\mathrm{Iso}_K^W(\mathfrak{x}, \mathfrak{y}) := \{\gamma \in K | \forall \alpha \in W : \mathfrak{x}(\alpha) = \mathfrak{y}(\alpha^\gamma)\}$$

In our context, we will maintain that K is a coset of some $G \leq \mathrm{Sym}(\Omega)$ and our window W is G-invariant. Let $K = G\gamma$, for a representative γ of the coset K, then:

$$\mathrm{Iso}_K^W(\mathfrak{x}, \mathfrak{y}) = \mathrm{Iso}_{G\gamma}^W(\mathfrak{x}, \mathfrak{y}) = \mathrm{Iso}_G^W(\mathfrak{x}, \mathfrak{y}^{\gamma^{-1}})\gamma$$

We can exploit this relation, and consider cases in which we are only dealing K which are groups. Thus, we can solve the problem of string isomorphism restricting K to cosets, by only solving the problem for K which are subgroups.

The input to the algorithm are two strings $\mathfrak{x}, \mathfrak{y}$, a set of generators for a group G, and a representative γ (in case K is a coset), and a window W. The first step of the algorithm 1 is to check if the representative is identity permutation or not, else we permute the second string, and make K a group.

Note that for the top call of the recursion, the window W is the full domain Ω, because we ant to compare the whole string. We will maintain the G-invariant nature of W throughout. The two searchable entities here are the window W, and the group G.

In the first kind of recursion, we split the window. This is done when the group in consideration is not transitive on W. Then, we pick an element α, and compute its orbit W_α. Then, we can restrict the group G to $\mathrm{Iso}_G^{W_\alpha}(\mathfrak{x}, \mathfrak{y})$, and our window to $W \setminus W_\alpha$. This is just basically doing string isomorphism on the substrings $\mathfrak{x}[W_\alpha]$ and $\mathfrak{y}[W_\alpha]$ (where $\mathfrak{x}[W_\alpha]$ is the string $\mathfrak{x}$ with domain restricted to W_α), and restricting our set of permutation set on which these are isomorphic. It is clear that:

$$\mathrm{Iso}_G^{W}(\mathfrak{x}, \mathfrak{y}) = \mathrm{Iso}_{K'}^{W \setminus W_\alpha}(\mathfrak{x}, \mathfrak{y}) \text{ where } K' = \mathrm{Iso}_G^{W_\alpha}(\mathfrak{x}, \mathfrak{y})$$

It is easy to see that in each recursion call, the window is always invariant over the set of permutations we are searching in.

The other recursion is when the group G is transitive over W. In this case, we cannot split the window to test string isomorphisms on substrings. Therefore, in this case, we restrict the set of permutations we are trying to find an isomorphism over. Now, let Δ be the subgroup that stabilises all the blocks of some minimal block system $\mathfrak{B}$ of G, i.e., $\Delta = G_{(\mathfrak{B})}$. Now, let T be the transversal of Δ, and $G[\mathfrak{B}]$ be the induced action of G on the minimal block system. We can see that:

$$\mathrm{Iso}_G^{W}(\mathfrak{x}, \mathfrak{y}) = \cup_{\gamma \in T} \mathrm{Iso}_{\Delta\gamma}^{W}(\mathfrak{x}, \mathfrak{y})$$

Let $t = |T|$. It is easy to see that $t = |G[\mathfrak{B}]|$, which is a primitive group since $\mathfrak{B}$ is a minimal block system. This holds true because, the transversal has representatives of cosets of Δ, and Δ stabilises each of the blocks, so each permutation in the transversal, permutes the blocks, which is exactly the group action of G on the minimal block system $\mathfrak{B}$.

The top call of the algorithm is done with a group G and permutation identity. The window W remains G invariant because, if we employ the first type of recursion, then we already know that the orbits are G invariant. Otherwise, the window is already Δ invariant since $\Delta \leq G$ under which W was invariant. And the first step after employing the recursion is to just just apply the representative, and again the window remains invariant of the group passed.

To analyse the time complexity of the algorithm, let us assume $n = |\Omega|$ and let us denote the number of recursive calls to the procedure by f. Thus, if we employ the first type of reduction, we have:

$$f(n) \leq f(n - n_1) + f(n_1)$$

where n_1 is the length of orbit computed. For the second recursion, if $|\mathfrak{B}| = b$, then we can easily see that the orbit of any element in Δ is going to be at most length n/b. Thus, there are going to be t problems with window size at most n/b, and they will employ the first type of recursion because Δ is not transitive. Thus, we have:

$$f(n) \leq t \cdot b \cdot f\left(\frac{n}{b}\right)$$

This is because, all subroutines in section 4.2 were polytime algorithms. The gap here is that, we also need a transversal for the setwise stabiliser of the block system. This can be done in the following manner. Let S be a generating set of the group G, and let $\mathfrak{B}$ be a minimal block system on this. For any permutation σ, let π be the corresponding induced action of σ in the group $G[\mathfrak{B}]$. Construct S' to be the set of generators induced on this, i.e., $S' = \{\pi_1, \ldots, \pi_k\}$. Now, we use Scheier Sims algorithm on this generator to get an SGS and also store the transversals of the subgroup chain defined by the base. A slight modification is that, whenever we operate on a permutation π, we store the corresponding σ. For example, if we have to do $\pi_i\pi_j$ anywhere in Schreier Sims, we store the corresponding $\sigma_i\sigma_j$ for it. Doing this, we obtain transversals as well as an SGS for the induced group $G[\mathfrak{B}]$ and also a pre-image for G corresponding to all the permutations considered. Now, using the stored transversals, we generate the whole group from the SGS, and store the corresponding permutations in G. This constructed set of permutations in G, will be the transversal for the setwise stabiliser of the block system $\mathfrak{B}$. It is not hard to see that the procedure described runs in time poly($|G[\mathfrak{B}]|$), which is in turn poly(t).

The basic idea behind the above procedure is that the setwise stabiliser of blocks, Δ, fixes all the blocks, so if we take the induced action of permutations on the group $G[\mathfrak{B}]$, then all the permutations in $G[\mathfrak{B}]$ permute the blocks as a whole, which is exactly what is present in the transversal of Δ. Hence, we only need to construct all permutations in $G[\mathfrak{B}]$ and store one pre-image for all of them in G, which will be the required transversal.

Thus, the time complexity of Luks' string isomorphism algorithm heavily depends on the size of primitive groups involved in the recursion. For the graph isomorphism problem with bounded degree d, we have already seen the reduction which leads us to groups that are in $\hat{\Gamma}_d$. Using a result from Babai, Cameron, Palfy [BCP82], we have:

Theorem 6.1. *There is a function g such that for every primitive permutation group $G \in \hat{\Gamma}_d$, we have that $|G| \leq n^{g(d)}$, where n is the size of domain of permutation group.*

In our case, the domain size is b, since the primitive group is over the minimal block system, we have $t = b^{g(d)}$, we have:

$$f(n) \leq b^{\mathcal{O}(g(d))} \cdot f\left(\frac{n}{b}\right)$$

From this, we can infer that the number of recursive calls to the procedure are polynomially bounded. Using the fact that all other subroutines are polynomial time procedures, one call to Luks' procedure takes polynomially bounded time.

Thus, for the bounded degree case, the graph isomorphism problem can be solved in polynomial time.

7 Babai's Algorithm for String Isomorphism

We have seen two kinds of recursions in Luks' algorithm previously. We will refer to them as follows:

Orbit-by-orbit processing For a group G in consideration, and a G-invariant window W, if G is not transitive over W, we can split W into several $G-$invariant windows as orbits, $W = W_1 \cup W_2 \cdots \cup W_d$. We recurse in the following way:

- $L \leftarrow G$
- for $i = 1 \ldots d$
    - $L \leftarrow \mathrm{Iso}_L(\mathfrak{x}^{W_i}, \mathfrak{y}^{W_i})$

This is a very efficient recurrence, because we only incur a small overhead and we get smaller instances of the problem.

$$f(n) \leq \sum_{i=1}^{d} f(n_i) + \text{overhead}$$

Subgroup Descent Given a subgroup $H \leq G$, we can descent to H, and obtain $|G : H|$ instances of H isomorphism problems. However, get $|G : H|$ number of problems of the same size, but on H instead of G.

$$\mathrm{Iso}_G(\mathfrak{x}, \mathfrak{y}) = \cup_{\gamma \in R} \mathrm{Iso}_{H\gamma}(\mathfrak{x}, \mathfrak{y})$$

where R is the set of right coset representative for H.

This can be inefficient when the index is large.

7.1 Bottlenecks for Luks' Algorithm

In Luks' algorithm, we descend to a subgroup, which stabilises all the blocks of a minimal block system $\mathfrak{B}$. Let us denote the subgroup by $G_{\mathfrak{B}}$. Now, the index $|G : G_{(\mathfrak{B})}|$ is equal to the size of the induced action of group on the blocks $G[\mathfrak{B}]$. We know that $G[\mathfrak{B}]$ is primitive since $\mathfrak{B}$ is minimal. Hence, if the size of this group is large, we incur a large multiplicative cost for subgroup descent, since we reduce our group to $G_{(\mathfrak{B})}$, but we get $|G[\mathfrak{B}]|$ instance of SI of the same size. Thus, large primitive groups create the bottleneck for efficient Luks' recursion.

Theorem 7.1. *(Cameron[Cam81], Maróti[Mar02]) For $n \geq 25$, if G is a primitive group with $|G| \geq n^{1+\log n}$ then G is a Cameron Group. $G \leq S_n$ is a Cameron group with parameters $s, t \geq 1$ and $k \geq \max(2t, 5)$ if for some $s, t \geq 1$ and $k \geq 2t$ we have $n = \binom{k}{t}^s$, the socle of G is isomorphic to A_k^s (which just the direct product of A_k s times) and $\left(A_k^{(t)}\right)^s \leq G \leq S_k^{(t)} \wr S_s$ (wreath product).*

Babai showed the equivalence of such Cameron groups to Johnson groups in the following result.

Theorem 7.2. *(Babai, [Bab15]) Let $G \leq S_n$ be a primitive group of order $|G| \geq n^{1+\log n}$, where n is greater than some absolute constant. Then there is a polynomial-time algorithm to compute a normal subgroup $N \trianglelefteq G$ of index $|G : N| \leq n$, and a block system $\mathfrak{B}$ such that $N[\mathfrak{B}]$ is permutationally equivalent to $A_m^{(t)}$ for some $m \geq \log n$.*

This theorem effectively says that the hard groups that Luks' recursion cannot handle are (essentially) Johnson Groups. The algorithmic part of the theorem is due to Babai, Luks and Seress [BLS87].

Now, we explain the procedure **Reduce-to-Johnson**, as described by Babai [Bab15]. The procedure takes input a group $G \leq \mathrm{Sym}(\Omega)$, and strings $\mathfrak{x}, \mathfrak{y} : \Omega \to \Sigma$ and gives output either the isomorphism group $\mathrm{Iso}_G(\mathfrak{x}, \mathfrak{y})$, or we get a new $G, \mathfrak{x}, \mathfrak{y}$, with G transitive, and a block system $\mathfrak{B}$ on which $G[\mathfrak{B}]$ is equivalent to a Johnson Group.

Reduce to Johnson

- if $G \leq \mathrm{Aut}(\mathfrak{x})$:
  - if $\mathfrak{x} = \mathfrak{y}$ then return G
  - else return ϕ
- if $|G| < C_0$, where C_0 is some constant, chosen to deem the combinatorial results usable, then we can compute $\mathrm{Iso}_G(\mathfrak{x}, \mathfrak{y})$ by bruteforce
- if G is intransitive, we can continue Luks' reduction using orbit by orbit processing
- Now if G is transitive, compute a minimal block system $\mathfrak{B}$, let $|\mathfrak{B}| = n_0$ the subgroup that stabilises all blocks $G_{(\mathfrak{B})}$ and let $G[\mathfrak{B}]$ be the induced action of G on the block system $\mathfrak{B}$
- if $|G[\mathfrak{B}]| < n_0^{1+n_0}$ then we can continue with subgroup descent to $G_{(\mathfrak{B})}$
- Now, since $G[\mathfrak{B}]$ is a primitive group of order $\geq n_0^{1+\log n_0}$, then using Theorem 7.1, reduce $G[\mathfrak{B}]$ to a Johnson group. Since by the theorem, the index is at most n_0, the multiplicative cost incurred is at most n_0.
- return $G, \mathfrak{B}(\text{modified}), G[\mathfrak{B}](\text{reduced to Johnson}), \mathfrak{x}, \mathfrak{y}$

Now we define a *giant representation.* A giant representation of a group H, is a homomorphism $\varepsilon : H \to \mathrm{Sym}(\Gamma')$, for some domain Γ' such that $H^{\varepsilon} \geq \mathrm{Alt}(\Gamma')$.

It is easy to see that we can obtain a homomorphism $\varphi : G \to \mathrm{Sym}(\Gamma)$, for some domain Γ, with $|\Gamma| = m$. This is a *giant representation* of G. This is because there is a trivial homomorphism from G to $G[\mathfrak{B}]$, and $G[\mathfrak{B}]$ is isomorphic to a Johnson Group. Johnson groups, $S_m^{(t)}$ and $A_m^{(t)}$, in turn are isomorphic to symmetric and alternating groups on domain size m. Note that we will use the term giant representation and giant homomorphism interchangeably.

We call this domain Γ the *ideal domain.*

7.2 Ideal Domain

Now we discuss some things about the ideal domain as described in the previous section. Now, again we have two cases if we consider the automorphism group $\mathrm{Aut}_G(\mathfrak{x})$.

If $\mathrm{Aut}_G(\mathfrak{x})$ admits a giant representation, i.e., for a giant representation $\varphi : G \to \mathrm{Sym}(\Gamma)$ of G, we have $\mathrm{Aut}_G(\mathfrak{x})^{\varphi} \geq \mathrm{Alt}(\Gamma)$, then we have a way to *lift* permutations from the ideal domain to the original domain.

However, when the automorphism group does not admit a giant representation, then we want to find an encasing subgroup of $\mathrm{Aut}_G(\mathfrak{x})$, such that $H \leq \mathrm{Sym}(\Gamma)$ and $\mathrm{Aut}_G(\mathfrak{x})^{\varphi} \leq H$.

Now, in our algorithm, in the second case, what we want is to find homomorphisms to encasing subgroups, which have giant representations in a significantly smaller domain. Thus, we are trying to shrink our ideal domain in every step. This is because $m = |\Gamma|$ goes in the exponent of the time complexity. Hence, our goal is to bring down m upto $\mathrm{poly}\log n$ to achieve a quasipolynomial bound.

Our target recurrence would be as follows, with n and $m = |\Gamma| \leq n$:

$$f(n, m) \leq q(n) f(n, 9/10m)$$

i.e., with $q(n)$ multiplicative cost, we want to shrink the ideal domain size by a constant factor. We decide on a threshold value $m_0 = \mathrm{poly}\log n$, beyond which we bruteforce on the ideal domain (lifting, described later) to reduce the original domain by a constant factor. This is equivalent to performing strong Luks reduction to reduce to the kernel of the giant homomorphism. This has a multiplicative cost of $m_0!$ which is quasipolynomial in n. Thus for $m \leq m_0$:

$$f(n, m) \leq q(n) f(9/10n, 9/10n)$$

Now, if the multiplicative costs that we incur during these steps is quasipolynomial in n, then the recurrence solves to quasipolynomial time.

7.3 Lifting

The main group-theoretic theorem used in Babai's algorithm is described next. We shall need the following definition.

Now, we define the notion of *affected points.* Given a giant homomorphism $\varphi : G \to \mathrm{Sym}(\Gamma)$ where $G \leq \mathrm{Sym}(\Omega)$, a point $\alpha \in \Omega$ is said to be affected if $G_\alpha^\varphi \ngeq \mathrm{Alt}(\Gamma)$. Also, if $\beta^\sigma = \alpha$ for some $\beta \in \Omega$ and $\sigma \in G$, then $G_\beta = \sigma G_\alpha \sigma^{-1}$. Therefore, $G_\beta^\varphi = \sigma^\varphi G_\alpha^\varphi (\sigma^{-1})^\varphi$. Thus, if α is unaffected so is β. We can infer that either all points of an orbit are affected or none of them are. If all of them are affected, we call that orbit an *affected orbit.*

Theorem 7.3. *Let $\varphi : G \to Sym(\Gamma)$ be a giant homomorphism. Let $n = |\Omega|, k = |\Gamma|$. Let U be the set of points of Ω unaffected by φ.*

1. *(Unaffected Stabilisers lemma) Assume $k > \max\{8, 2+\log_2 n\}$. Then $(G_{(U)})^\varphi \geq Alt(\Gamma)$. In particular, $U \neq \Omega$ (at least one point of Ω is affected).*
2. *(Affected Orbit lemma) Assume $k \geq 5$. If Δ is an affected G-orbit, then $\ker(\varphi)$ is not transitive on Δ. Each orbit of $\ker(\varphi)$ in Δ has length at most Δ/k.*

We do not go into the proof of the Unaffected Stabilisers here, but it uses the O'Nan-Scott theorem describing the structure of maximal permutation groups and CFSG via Schreier's hypothesis. Schreier's hypothesis states that the outer automorphism group of any simple group is solvable. The second statement in Part 1 follows from the first as we argue now. If $U = \Omega$, by the first statement $(G_{(\Omega)})^\varphi \geq \mathrm{Alt}(\Gamma)$. But $G_{(U)} = \{\mathrm{id}\}$ which cannot be mapped to $\mathrm{Alt}(\Gamma)$ since $k > 8$. So there must be an affected point.

Part 2 has an elementary proof.

Proof of the Affected Orbit lemma. Let $x \in \Delta$ and $N := \ker(\varphi)$. The orbit of x under the action of N is

$$|x^N| = \frac{|N|}{|N_x|} = \frac{|N|}{|N \cap G_x|} = \frac{|NG_x|}{|G_x|} = \frac{|G : G_x|}{|G : NG_x|} = \frac{|\Delta|}{|G^\varphi : (G_x)^\varphi|}$$

Since x is affected, $(G_x)^\varphi \ngeq \mathrm{Alt}(\Gamma)$. Now if $G^\varphi = \mathrm{Alt}(\Gamma)$, any proper subgroup of $\mathrm{Alt}(\Gamma)$ has index at least k since $k \geq 5$ so that $\mathrm{Alt}(\Gamma)$ is simple. If $G^\varphi = \mathrm{Sym}(\Gamma)$, any subgroup of $\mathrm{Sym}(\Gamma)$ which is not $\mathrm{Alt}(\Gamma)$ also has index at least k. So $|x^N| = \frac{|\Delta|}{|G^\varphi : (G_x)^\varphi|} \leq \frac{|\Delta|}{k}$. □

Suppose $\varphi : G \to \mathrm{Sym}(\Gamma)$ is a giant homomorphism. Let $\bar{g} \in \mathrm{Alt}(\Gamma)$. Since φ is a giant homomorphism, there is some $g \in G$ such that $\varphi(g) = \bar{g}$. Now we want to find $g' \in \mathrm{Aut}_G(\mathfrak{x})$ such that $\varphi(g') = \bar{g}$. The following lifting procedure does this.

Algorithm 2 Lifting

```
1: $K := \ker(\varphi)$
2: find $g$ such that $\varphi(g) = \bar{g}$
3: return $\mathrm{Aut}_{Kg}(\mathfrak{x})$
```

The set of automorphisms mapped to $\bar{g}$ is given by $\mathrm{Aut}_G(s) \cap Kg = \mathrm{Aut}_{Kg}(\mathfrak{x}) = \mathrm{Iso}_K(\mathfrak{x}, \mathfrak{x}^{g^{-1}})g$. So we have reduced finding the coset of automorphisms which map to $\bar{g}$ to a single K-isomorphism instance. Every K-orbit has size at most n/m by the Affected Orbit lemma.

Now we see how this can be used to determine $\mathrm{Aut}_G(\mathfrak{x})$ when $\mathrm{Aut}_G(\mathfrak{x})^\varphi \geq \mathrm{Alt}(\Gamma)$. Let S be a set of generators of $\mathrm{Alt}(\Gamma)$. For instance, one can take the set of all 3-cycles in Γ. Lift each element of S to $\mathrm{Aut}_G(\mathfrak{x})$. If any of these is empty, $\mathrm{Aut}_G(\mathfrak{x})^\varphi \ngeq \mathrm{Alt}(\Gamma)$. If all of them lift, $\mathrm{Aut}_G(\mathfrak{x})^\varphi \geq \mathrm{Alt}(\Gamma)$. To check if $\mathrm{Aut}_G(\mathfrak{x})$ is actually generated by the lifts, lift a transposition on Γ to $\mathrm{Aut}_G(\mathfrak{x})$. If it also lifts, $\mathrm{Aut}_G(\mathfrak{x})^\varphi = \mathrm{Sym}(\Gamma)$ and $\mathrm{Aut}_G(\mathfrak{x})$ is generated by the lifts of S and the transposition. If the transposition does not lift, $\mathrm{Aut}_G(\mathfrak{x})^\varphi = \mathrm{Alt}(\Gamma)$ and $\mathrm{Aut}_G(\mathfrak{x})$ is generated by the lifts of S.

7.4 Canonical Structures on the ideal domain

We start by abstractly defining the notion of a canonical structure. Later we describe what kinds of structures we will mainly be using. Let f be a function on strings $\mathfrak{x} : \Omega \to \Sigma$ which associates a structure $f(\mathfrak{x})$ on Γ satisfying the following property. For every pair of strings $\mathfrak{x}, \mathfrak{y}$ and $\sigma \in \mathrm{Iso}_G(\mathfrak{x}, \mathfrak{y})$, σ^φ is an isomorphism between $f(\mathfrak{x})$ and $f(\mathfrak{y})$. Then we say that $f(\mathfrak{x})$ is canonically obtained from $\mathfrak{x}$. Note that since every isomorphism of strings transforms into an isomorphism of the corresponding canonical structures, we can try restricting the set of potential isomorphisms of given input strings by lifting isomorphisms between the corresponding canonical structures. If the canonical structures are not isomorphic, then we already know that the strings are not isomorphic. Otherwise we have shrunk the coset inside which we were looking for isomorphisms.

The two main kinds of canonical structures which will appear repeatedly throughout Babai's algorithm are colourings and equipartitions. A colouring of the ideal domain Γ is simply a function $c : \Gamma \to [l]$ where l is the number of colours. Equivalently it is an ordered partition of Γ. An equipartition of Γ is a partition $\{\Gamma_i \mid i \in [l]\}$ where each part is of the same size (for each $i \in [l]$, $|\Gamma_i| = |\Gamma|/l$).

We say that we have a good colouring, if all the colour classes are at most 90% of the size of the domain we are working on. We will now describe how obtaining a good colouring or an equipartition helps in reducing the domain size. Let $f(\mathfrak{x})$ and $f(\mathfrak{y})$ be canonical structures associated with strings $\mathfrak{x}$ and $\mathfrak{y}$ respectively. These will be colourings of the domain possibly with some additional structure if the colouring is not good.

Algorithm 3 Align

```
1: If $f(\mathfrak{x})$ and $f(\mathfrak{y})$ are not $G^{\varphi}$-isomorphic, reject.
2: Pick any $\bar{\sigma} \in \mathrm{Iso}_{G^{\varphi}}(f(\mathfrak{x}), f(\mathfrak{y}))$ and $\sigma \in \varphi^{-1}(\bar{\sigma})$.
3: Set $\mathfrak{y} = \mathfrak{y}^{\sigma^{-1}}$ and $G = \varphi^{-1}(\mathrm{Aut}(f(\mathfrak{x})))$.
4: if $f(\mathfrak{x})$ is not good then
5:     if the dominant colour class $\Delta$ in $f(\mathfrak{x})$ has an equipartition then
6:         Let $\Gamma'$ be the set of parts of this equipartition.
7:     else
8:         $\Delta$ is the vertex set of a Johnson graph $J(s,t)$, then associate $\Delta$ with $\binom{\Gamma'}{t}$ where $|\Gamma'| = s$.
9:     end if
10:    Set $\Gamma = \Gamma'$
11:    Update $\varphi$ to be the composition of the original $\varphi$ and the natural action of the automorphism group on the additional structure to $\mathrm{Sym}(\Gamma')$
12: end if
```

We now look at some more details related to the Align procedure. The isomorphism test in the first line can be performed efficiently since $G^{\varphi} = \mathrm{Sym}(\Gamma)$ or $G^{\varphi} = \mathrm{Alt}(\Gamma)$. By what was explained earlier, $\mathfrak{x}$ and $\mathfrak{y}$ cannot be G-isomorphic if $f(\mathfrak{x})$ and $f(\mathfrak{y})$ are not G^{φ}-isomorphic. Then we lift a G^{φ}-isomorphism between $f(\mathfrak{x})$ and $f(\mathfrak{y})$ to G and perform a shift appropriately. In this way, $\mathrm{Iso}_G(\mathfrak{x}, \mathfrak{y}) \subseteq \mathrm{Aut}_{G'}(\mathfrak{x}, \mathfrak{y}^{\sigma^{-1}})$ where $G' = \varphi^{-1}(\mathrm{Aut}(f(\mathfrak{x})))$ is the updated G. If the colouring $f(\mathfrak{x})$ is good, G is intransitive with each orbit having length at most $0.9n$. So in this case, we have shrunk the actual domain significantly. In the other cases, we shrink the ideal domain. When there is a non-trivial equipartition on the dominant colour class, since each block has size at least 2, the new domain Γ' has size at most $m/2$. The Johnson graph case is explained in detail later.

7.5 Combinatorial techniques

Suppose we have a canonically embedded regular graph on the ideal domain Γ. Then we want to find good colourings or equipartitions on it. However, some graphs are resistent to good colouring or partition. One case being Johnson graphs, which are resilient to combinatorial techniques used in these cases.

But, we see that Johnson graphs give a very large reduction in the domain size. Suppose we can find a Johnson graph $J(m', t)$, for $t \geq 2$. We know that the automorphism group is $S_{m'}^{(t)}$ which is isomorphic to $S_{m'}$. Thus, we can reduce our ideal domain to m'. It is easy to see that:

$$\mathrm{Aut}_G(\mathfrak{x}) \to \mathrm{Sym}(\Gamma) \to \mathrm{Aut}(J(m',t)) \to S_{m'}$$
$$m' < 1 + \sqrt{2m}$$

Thus, reduction from m to m' is very efficient. If we keep getting Johnson graphs on new domains, the size shrinks very quickly t poly $\log n$, in just $\log\log n$ rounds.

However, the good part is, that the only hard cases for good colourings or equipartitions are Johnson Graphs. This is by the Split-or-Johnson routine described by Babai.

Theorem 7.4. *(Split-or-Johnson) Given a nontrivial regular graph on* Γ*, at quasipolynomial multiplicative cost, we can find one of the following:*

1. *a good canonical colouring (*$\forall$ *colour classes* ≤ 0.9*, or*
2. *a canonical equipartition on the large colour class* > 0.9*, or*
3. *a canonically embedded Johnson graph on the large colour class* > 0.9

We can only invoke the Split-or-Johnson lemma when we have a canonically embedded regular graph on the ideal domain Γ. However, we cannot directly deduce a regular canonical structure on Γ. What we get is a canonical k-ary relation on Γ by the local certificates routine (described later).

A k-ary relation $\mathfrak{X} = (\Gamma, R)$ is such that $R \subseteq \Gamma^k$. We call $\Delta \subseteq \Gamma$ a symmetric set if $\mathrm{Sym}\Delta \leq \mathrm{Aut}(\mathfrak{X})$. Using this, let us define the symmetry defect. The symmetry defect of $\mathfrak{X}$ is $\min\{|T| \mid \Gamma \setminus T$ is a symmetric set for $\mathfrak{X}\}$. Note that a graph is just a binary relation.

On the k-ary relations given by local certificates, we can invoke the following lemma called *Design Lemma* due to Babai:

Theorem 7.5. *(Design Lemma) Given a k-ary relation with a symmetry defect* $\geq \frac{1}{10}$ *one can find, at* $m^{O(k)}$ *multiplicative cost, one of the following:*

1. *a good canonical colouring (*$\forall$ *colour classes* ≤ 0.9*, or*
2. *a canonical equipartition on the large colour class* > 0.9*, or*
3. *a canonically embedded non-trivial regular graph on the large colour class* > 0.9

In our case, the k-ary relation given by the local certificates algorithm is such that $k = O(\log n)$, and thus, the multiplicative cost remains quasipolynomial. We can also see that we do not need the condition on the symmetry defect in the Split-or-Johnson statement, because non-trivial regular graphs already have a symmetry defect of at least $\frac{1}{2}$.

7.6 Local Certificates

We now look at the core group-theoretic subroutine used in Babai's algorithm. Informally the idea is to consider small test sets on the ideal domain and look at the action of automorphisms fixing the test set. Considering such local pieces of information, we can combine them to obtain information about the whole automorphism group. This local-to-global approach crucially relies on the group-theoretic facts mentioned earlier.

A test set $T \subseteq \Gamma$ has size $|T| = k > \max\{8, 2 + \log_2 n\}$ and $k = O(\log n)$. We also assume that $m > 10k$ since otherwise we can apply brute force. Given a test set T, denote by G_T the setwise stabiliser of T in G, $G_T = \{\sigma \in G \mid \sigma^\varphi \in \mathrm{Sym}(\Gamma)_T\}$. Let $\psi_T : G_T \to \mathrm{Sym}(T)$ be the homomorphism defined by $\sigma^{\psi_T} = \sigma^\varphi|_T$. A test set is said to be *full* if $\mathrm{Aut}_{G_T}(\mathfrak{x})^{\psi_T} \geq \mathrm{Alt}(T)$. We need to compute whether a given test set is full or not. Moreover we need meaningful certificates of fullness or non-fullness. These are defined next. A *fullness certificate* for a full test set T is a subgroup $K(T) \leq \mathrm{Aut}_{G_T}(\mathfrak{x})$ such that $K(T)^{\psi_T} \leq M(T)$. A non-fullness certificate is a subgroup $M(T) \leq \mathrm{Sym}(T)$ which is not a giant such that $\mathrm{Aut}_{G_T}(\mathfrak{x})^{\psi_T} \leq M(T)$. The local certificates algorithm decides if T is full and returns an appropriate certificate. For $W \subseteq \Omega$, let $A(W) = \mathrm{Aut}_{G_T}(\mathfrak{x}^W)$, the automorphism group of the partial string $\mathfrak{x}^W$ in G_T.

Algorithm 4 Local Certificates

```
 1: $W := \emptyset$
 2: **while** $W \neq \mathrm{Aff}(A(W))$ and $A(W)^{\psi_T} \geq \mathrm{Alt}(T)$ **do**
 3:     $W = \mathrm{Aff}(A(W))$
 4:     Update $A(W)$
 5: **end while**
 6: **if** $A(W)^{\psi_T} \geq \mathrm{Alt}(T)$ **then**
 7:     **return** "full", $A(W)_{(U)}$ where $U = \Omega \setminus W$
 8: **else**
 9:     **return** "non-full", $A(W)^{\psi_T}$
10: **end if**
```

The details of how to update $A(W)$ will be given later. We first argue correctness of the algorithm. First note that in the beginning $A(W) = G$ since W is empty. Therefore $A(W)^{\psi_T} \geq \mathrm{Alt}(T)$ since $A(W)$ is a giant. Also by the Unaffected Stabilizers lemma, there is at least one point affected by $A(W) = G$. Therefore the while loop is entered at least once. Since each orbit is either affected or unaffected, the window W is always a union of orbits. Moreover the set of affected points can only grow, since $A(G)$ becomes smaller and thus the corresponding pointwise stabilisers can also only get smaller. There can be at most $n = |\Omega|$ many iterations of the while loop, since in every iteration the window W grows.

Now suppose the if condition is satisfied. By the termination condition for the while loop, it must be that $W = \mathrm{Aff}(A(W))$. So the set $U = \Omega \setminus W$ is unaffected by $A(W)$. Now we use the Unaffected Stabilizers lemma to deduce that $A(W)_{(U)}^{\psi_T} \geq \mathrm{Alt}(T)$. Moreover $A(W)_{(U)} \leq \mathrm{Aut}_{G_T}(\mathfrak{x})$ since $A(W)$ only contains automorphisms of $\mathfrak{x}^W$ and each point outside W is fixed by permutations in $A(W)_{(U)}$. Therefore $A(W)_{(U)}$ is indeed a fullness certificate.

In the other case, $A(W)^{\psi_T} \not\geq \mathrm{Alt}(T)$. Since $\mathrm{Aut}_{G_T}(\mathfrak{x}) \leq A(W)$, $\mathrm{Aut}_{G_T}(\mathfrak{x})^{\psi_T} \not\geq \mathrm{Alt}(T)$. Also $\mathrm{Aut}_{G_T}(\mathfrak{x})^{\psi_T} \leq A(W)^{\psi_T}$, so $A(W)^{\psi_T}$ is a non-fullness certificate.

For the update step, let W_{old} be the window before it is changed.

Algorithm 5 Updating $A(W)$

```
1: $N := A(W_{\mathrm{old}})_{(T)}$
2: $L := \emptyset$
3: for $\bar{\sigma} \in A(W_{\mathrm{old}})^{\psi_T}$ do
4:     pick $\sigma \in A(W_{\mathrm{old}})$ such that $\sigma^{\psi_T} = \bar{\sigma}$
5:     $L(\bar{\sigma}) = \mathrm{Aut}_{N\sigma}^{W}(\mathfrak{x})$
6:     $L = L \cup L(\bar{\sigma})$
7: end for
8: $A(W) = L$
```

In the first line N is the kernel of the map $\psi_T : A(W_{\mathrm{old}}) \to \mathrm{Sym}(T)$. The set L will eventually contain the generators of $A(W)$. The main idea used in the algorithm is the following decomposition of $A(W_{\mathrm{old}})$ into cosets:

$$A(W_{\mathrm{old}}) = \cup_{\bar{\sigma}} N\sigma$$

Now we perform Luks descent. This proves correctness. For efficiency, note that by the Affected Orbit lemma, each orbit of N in W has length n/k. The number of $\bar{\sigma}$'s we go over is at most $k!$. So overall the recomputation of $A(W)$ requires at most $n \cdot k!$ instances of String Isomorphism calls on windows of length at most n/k.

Theorem 7.6 (Aggregate Certificates)**.** *Let* $\max\{8, 2+\log_2 n\} < k < m/10$. *Then at a multiplicative cost of* $m^{O(k)}$, *we can either find a good canonical colouring of* Γ, *or a good canonically embedded* k*-ary relational structure with relative symmetry defect* $\geq 1/2$ *or reduce the determination of* $\mathit{Iso}_G(s_1, s_2)$ *to* $n^{O(1)}$ *instances of size* $\leq 2n/3$.

We now give a rough idea about how the AggregateCertificates routine proceeds, but no details are provided here. First compute local certificates for all $\binom{\Gamma}{t}$ test sets. Also run CompareLocalCertificates on pairs of test sets (this is similar to the LocalCertificates routine). Now look at the subgroup F generated by the fullness certificates. If the nontrivial orbits of F in Γ cover at least 0.1 fraction and no orbit has length greater than 0.9 fraction of all points, we have a good canonical colouring. Otherwise if there is a large orbit, we check if F^{φ} is a giant. If yes, we can try using lifting ideas to break the problem into smaller instances of string isomorphism. Otherwise the degree of transitivity of F^{φ} is small. In this case, we can find a structure with high symmetry defect by individualizing a few points. In the remaining case, where most points in Γ are fixed, we can find a canonically embedded k-ary relational structure with relative symmetry defect $\geq 1/2$.

7.7 Master Algorithm

The following algorithm assumes that we have performed the procedure Reduce-to-Johnson and have the ideal domain Γ, the giant homomorphism φ and a set of blocks $\Phi = \{B_t \mid T \in \binom{\Gamma}{t}\}$ on which G acts as a Johnson group. Following Babai [Bab15], whenever a good colouring is returned, we perform orbit by orbit processing on the strings and then recurse on the smaller windows. If some other structure is returned, we continue with the algorithm.

The procedure ProcessHighSymmetry is described now. Let C be the unique symmetric class of size $\geq |\Gamma|/2$. If $\mathfrak{x}$ has such a class $C_{\mathfrak{x}}$ but $\mathfrak{y}$ does not or the sizes of such classes $C_{\mathfrak{x}}$ and $C_{\mathfrak{y}}$ in $\mathfrak{x}$ and $\mathfrak{y}$ are different, we can reject. Otherwise let $\bar{\sigma}$ be a permutation sending $C_{\mathfrak{x}}$ to $C_{\mathfrak{y}}$. Now we can align by setting $\mathfrak{y} = \mathfrak{y}^{\sigma}$ where $\sigma^{\varphi} = \bar{\sigma}$, so that $C_{\mathfrak{x}} = C_{\mathfrak{y}}$. Using the partition $\{C_{\mathfrak{x}}, \Gamma \setminus C_{\mathfrak{x}}\}$, we can colour each point $x \in \Omega$ based on the number of points in $C_{\mathfrak{x}}$ contained in x. Since we are in the primitive case, each point is a block which corresponds to a k-subset of Γ. Now each colour class corresponding to sets having at most $k-1$ elements of $C_{\mathfrak{x}}$ has size at most $n/2$ (which can be checked by a small calculation). Let C_0 be the remaining colour class in Ω consisting of sets all of whose elements lie in C.

Now we have reduced the problem to the group $H = \varphi^{-1}(\mathrm{Alt}(\Gamma)_C)$ and we need to find $\mathrm{Iso}_H(\mathfrak{x}, \mathfrak{y})$. This further reduces to two instances with group $H' = \varphi^{-1}(\mathrm{Alt}(C) \times \mathrm{Alt}(\Gamma \setminus C))$, $\mathrm{Iso}_H(\mathfrak{x}, \mathfrak{y}')$ for $\mathfrak{y}' = \mathfrak{y}$ and $\mathfrak{y}' = \mathfrak{y}^{\sigma}$ where σ is a permutation which is the product of a transposition on C and a transposition on $\Gamma \setminus C$. As C is a symmetric class, any permutation $\sigma \in \varphi^{-1}(\mathrm{Alt}(C))$ leaves $\mathfrak{x}^{C_0}$ unchanged. Therefore if $\mathfrak{x}^{C_0} \neq \mathfrak{y}^{C_0}$, we can reject. Otherwise they are the same and we can recurse on the other colour classes each of which has size less than $n/2$.

We next explain how to find $\mathrm{Iso}_G(\mathfrak{x}, \mathfrak{y})$ when $t = 1$. In this case, we first check if $\mathfrak{x}$ and $\mathfrak{y}$ contain each letter in Σ with the same frequency. If not, we can reject. Otherwise we find a permutation $\sigma \in G$ such that $\mathfrak{x}^{\sigma} = \mathfrak{y}$. This is always possible if $G = \mathrm{Sym}(\Omega)$. If $G = \mathrm{Alt}(\Omega)$, such a permutation always exists as long as some letter appears at least twice. If not, there is at most one isomorphism. Now suppose we have such a permutation σ. Then we only need $\mathrm{Aut}_G(\mathfrak{x}, \mathfrak{x})$. This

Algorithm 6 Babai's SI

```
 1: if $m \leq (\log n)^3$ then
 2:     apply Luks descent to reduce to kernel of action on the blocks
 3: else
 4:     set $N = \ker(\varphi)$
 5:     if $G$ primitive (each block $B_T$ is a singleton) then
 6:         if t $= 1$ then
 7:             return $\mathrm{Iso}_G(\mathfrak{x}, \mathfrak{y})$
 8:         else
 9:             view $\mathfrak{x}, \mathfrak{y}$ as edge-coloured $t$-uniform hypergraphs $H(\mathfrak{x})$ and $H(\mathfrak{y})$ on $\Gamma$
10:             if relative symmetry of $H(\mathfrak{x}) < 1/2$ then
11:                 Perform ProcessHighSymmetry on $H(\mathfrak{x})$
12:             else
13:                 Apply Design Lemma, followed by Split-or-Johnson if required
14:                 Align
15:             end if
16:         end if
17:     else
18:         Run AggregateCertificates
19:         if a canonical $k$-ary structure is returned then
20:             Apply Design Lemma, followed by Split-or-Johnson if required
21:         end if
22:         Align
23:     end if
24: end if
```

is a direct product of symmetric groups on positions having the same letter if $G = \mathrm{Sym}(\Omega)$. If $G = \mathrm{Alt}(\Omega)$ we need to only consider such even permutations.

8 Conclusion

We have looked at some of the main algorithms for the graph isomorphism problem. Much more work has been done on restricted classes on graphs and understanding the power of the different approaches. We note that in practice, several instances of GI can be handled efficiently by programs such as `nauty` and `traces`. We refer the reader to [MP14] for more on practical graph isomorphism. In this section, we briefly look at some connections of GI with other isomorphism problems.

Group Isomorphism

Given two groups G_1, G_2 explicitly via their Cayley tables, are G_1 and G_2 isomorphic? This is the Group Isomorphism problem (GpI). There is a simple polynomial time reduction from GpI to GI. In fact, isomorphism of any kind of first-order structures (using functions, relations, constants) reduces to GI [Mil79]. Here we describe the reduction from GpI to GI.

For a group G, construct graph $X = (V, E)$ where $V = \{v_g \mid g \in G\} \cup \{a_{(g,h)}, b_{(g,h)}, c_{(g,h)}, d_{(g,h)} \mid (g,h) \in G \times G\}$. So if $|G| = n$, $|V| = n + 4n^2$. For each pair $(g,h) \in G \times G$, we add the following 6 undirected edges to X:

$$\begin{gathered}(v_g, a_{(g,h)}), (v_h, b_{(g,h)}), (v_{gh}, d_{(g,h)}),\\ (a_{(g,h)}, b_{(g,h)}), (b_{(g,h)}, c_{(g,h)}), (c_{(g,h)}, d_{(g,h)}).\end{gathered}$$

Now the reduction to GI transforms groups G_1 and G_2 into graphs X_1 and X_2. It is clear that any isomorphism $\varphi : G_1 \to G_2$ gives an isomorphism $\psi : X_1 \to X_2$ as follows

$$\begin{aligned}\psi(v_g) &= v_{\varphi(g)}\\ \psi(a_{(g,h)}) &= a_{(\varphi(g),\varphi(h))}\\ \psi(b_{(g,h)}) &= b_{(\varphi(g),\varphi(h))}\\ \psi(c_{(g,h)}) &= c_{(\varphi(g),\varphi(h))}\\ \psi(d_{(g,h)}) &= d_{(\varphi(g),\varphi(h))}.\end{aligned}$$

For the other direction, we first claim that any isomorphism $\psi : X_1 \to X_2$ must map c vertices of X_1 to c vertices of X_2. Any c vertex has degree 2, one neighbour (a d vertex) with degree 2 and another neighbour (a b vertex) with degree 3. Note that no v, a, b or d vertex satisfies these conditions if $n > 1$. (If $n = 1$, there is a unique group of order n, so there is nothing to check.) So a c vertex must be mapped to a c vertex. Now the neighbours of a c vertex must also be mapped to corresponding neighbours and similarly for the neighbours of neighbours. So the map ψ must preserve the type of a vertex: a, b, c, d or v.

Define $\varphi : G_1 \to G_2$ in the following way. For all $g \in G_1$, if $\psi(v_g) = v_h$, define $\varphi(g) = h$. Since φ is a bijection from X_1 and X_2, φ is a bijection from G_1 to G_2. To see that it preserves the group relations, suppose $g_1g_2 = g_3$. Then we have a corresponding gadget in X_1 with vertices $a_{(g_1,g_2)}, b_{(g_1,g_2)}, c_{(g_1,g_2)}, d_{(g_1,g_2)}$. These are mapped by ψ to $a_{(h_1,h_2)}, b_{(h_1,h_2)}, c_{(h_1,h_2)}, d_{(h_1,h_2)}$ respectively for some $(h_1, h_2) \in G_2 \times G_2$. Then v_{g_1}, v_{g_2}, v_{g_3} are mapped to v_{h_1}, v_{h_2}, v_{h_3} respectively where $h_3 = h_1h_2$. Therefore $\varphi(g_1g_2) = \varphi(g_1)\varphi(g_2)$.

So we have a polynomial time reduction from GpI to GI. Combined with Babai's algorithm, this gives a quasipolynomial algorithm for GpI. However this is unnecessary. For decades now, an $n^{\log n+O(1)}$ algorithm for GpI has been known ([Mil78] where the algorithm is attributed to Tarjan). This relies on the fact that any group of order n has a generating set of size at most $\log_2 n$. Given such a generating set, we only need to check all possible maps from this generating set to G_2 and check if any of them extend to an isomorphism. This check can be performed in polynomial time. Improving the complexity to $n^{o(\log n)}$ for GpI is open.

We briefly state now how combinatorial approaches like Weisfeiler-Leman have been useful for understanding the group isomorphism problem. Brachter and Schweitzer [BS20] studied three WL-type algorithms in the context of GpI and the notion of WL-dimension. They show that these WL-type algorithms are equivalent in the sense that the WL-dimension with respect to different algorithms only differ by a constant factor. One of these algorithms performs the above reduction transforming a group to a graph, performs k-WL on the obtained graph and then pulls back the obtained colouring to the group. Another version applies the ideas of k-WL to the group itself, while the third does the same while also taking the subgroup generated by a tuple into account. They also construct non-isomorphic graphs having some common isomorphism invariants which are distinguished by WL. Grochow and Levet [GL21] showed that WL for groups also captures isomorphism for several classes of groups for which different polytime tests were known.

Chattopadhyay, Torán and Wagner [CTW13] proved that Graph Isomorphism is strictly harder than Group Isomorphism under a weak notion of reductions. Specifically they showed that GI is not $\mathrm{ACC}^0[p]$ reducible to GpI even when the reduction is allowed to be randomized. The rough idea for this is as follows. First it is shown that GpI can be decided by a restricted class of polynomial size nondeterministic circuits, while this restricted class of circuits cannot compute Parity. Since Parity is AC^0 reducible to GI, this implies that there is no AC^0 reduction from GI to GpI.

Recently Dietrich and Wilson [DW22] showed that Group Isomorphism is nearly-linear time for *most* orders. More formally they showed that there is a dense subset $Y \subseteq \mathbb{N}$ and a deterministic Turing machine that decides GpI for input size $n \in Y$ in time $O(n^2(\log n)^c)$ for some constant c. Here a dense subset

Y is a subset satisfying $\lim_{n\to\infty} \frac{|Y\cap[n]|}{n} = 1$. It is worth emphasizing, however, that this does *not* show that most instances of GpI are efficiently solvable since there are “many” groups of orders excluded from Y. One class excluded from Y is that of prime powers, and it is believed that groups with prime power orders form the bottleneck for efficient group isomorphism testing.

Boolean function Isomorphism

In this subsection, we consider the problem of deciding when to Boolean functions given as input are ‘equivalent’. Here one needs to define the notion of equivalence and how the functions are encoded. For instance, if the notion of equivalence is both functions being the same and each function is described as a formula, this problem is coNP-complete since a certificate for two functions not being the same is an input where they differ and TAUT obviously reduces to this problem in polynomial time. Now suppose we relax the notion of equivalence to isomorphism: there is some permutation $\sigma \in S_n$ such that $f(x) = g(\sigma x)$. Then this is the problem of Formula Isomorphism (FI). Agrawal and Thierauf [AT00] showed that FI cannot be Σ_2^p-complete unless the polynomial hierarchy collapses.

Instead of a succinct representation as a formula, suppose each function is given by its truth table. So the input has length $\Omega(2^n)$, where n is the number of variables. Luks [Luk99] showed that this can be done in $O(c^n)$ time. If we also allow flipping some of the input bits in our notion of equivalence, we get the problem structural equivalence of functions which can also be solved in the same time. Structural equivalence of f and g can be expressed as the existence of a permutation matrix $P \in M(\mathbb{F}_2, n)$ and a vector $v \in \mathbb{F}_2^n$ such that $f(x) = g(Px+c)$. This motivates looking at equivalence under the action of general and general-affine linear groups. One of the open problems in [Luk99] is to find a polynomial time algorithm for isomorphism testing under the action of these groups. A polynomial time algorithm for GI would give rise to such an isomorphism test for the general linear group since there is a polynomial time Karp reduction from this problem to GI. We note that Babai’s algorithm is not useful here since the exponent there is greater than 2 while testing equivalence via enumeration requires only $O(2^{cn^2}) = O(N^{c\log N})$ time where $N = 2^n$.

SI/GI on groups with restricted composition factors

For groups in $\widehat{\Gamma}_d$, we have already seen that Luks’ algorithm terminates in $n^{O(d)}$ time. Let us define a slightly general SI instance, the *Set-of-Strings Isomorphism Problem.* It takes input two sets of strings $\mathfrak{X} = \{\mathfrak{x}_1, \ldots, \mathfrak{x}_m\}$ and $\mathfrak{Y} = \{\mathfrak{y}_1, \ldots, \mathfrak{y}_m\}$, where $\mathfrak{x}_i, \mathfrak{y}_i : \Omega \to \Sigma$ are strings and a group $G \leq \mathrm{Sym}(\Omega)$ and asks whether there is some $\gamma \in G$ such that $\mathfrak{X}^\gamma = \{\mathfrak{x}_1, \ldots, \mathfrak{x}_m\} = \mathfrak{Y}$.

This is interesting because the Hypergraph isomorphism problem for $\widehat{\Gamma}_d$ groups is polynomial time reducible to the Set-of-Strings isomorphism problem for $\widehat{\Gamma}_d$ groups under many one reductions. It is also interesting to see that the Set-of-Strings problem on $\widehat{\Gamma}_d$ groups is solvable in $(n+m)^{\operatorname{poly}\log d}$ time, which is better than a trivial bound using Luks.

There are various classes of graphs that can be reduced to SI on groups with restricted composition factors. One such example is *t-CR-bounded* graphs. A graph (G, χ), where χ is a vertex colouring, is a t-CR-bounded graph if, the colouring χ can be converted to a discrete colouring, i.e., a colouring in which each vertex has its own colour, by applying the following two operations repeatedly in any order:

- apply color refinement algorithm, or
- For a color class of size less or equal to t, assign discrete colors

For this class of graphs, a polynomial time Turing reduction from GI to Set-of-Strings Isomorphism problem under $\widehat{\Gamma}_t$ groups is known. And as a corollary, the Isomorphism of t-CR-bounded graphs can be decided in $n^{\operatorname{poly}\log t}$ time.

Similarly, a number of results with restricted group structures or graph classes are known. For a more intricate survey of these results, we refer the reader to [GN20].

References

[AT00] Manindra Agrawal and Thomas Thierauf. The formula isomorphism problem. *SIAM Journal on Computing*, 30(3):990–1009, 2000.

[Bab15] László Babai. Graph isomorphism in quasipolynomial time, 2015.

[Bab16] László Babai. Graph isomorphism in quasipolynomial time. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 684–697, 2016.

[BCP82] László Babai, Peter J Cameron, and Péter P Pálfy. On the orders of primitive groups with restricted nonabelian composition factors. *Journal of Algebra*, 79(1):161–168, 1982.

[BES80] László Babai, Paul Erdos, and Stanley M Selkow. Random graph isomorphism. *SIaM Journal on computing*, 9(3):628–635, 1980.

[BKL83] László Babai, William M Kantor, and Eugene M Luks. Computational complexity and the classification of finite simple groups. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, pages 162–171. IEEE, 1983.

[BLS87] L. Babai, E. Luks, and A. Seress. Permutation groups in nc. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, page 409–420, New York, NY, USA, 1987. Association for Computing Machinery.

[BS20] Jendrik Brachter and Pascal Schweitzer. On the weisfeiler-leman dimension of finite groups. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 287–300, 2020.

[Cam81] Peter J Cameron. Finite permutation groups and finite simple groups. *Bulletin of the London Mathematical Society*, 13(1):1–22, 1981.

[CC82] Alain Cardon and Maxime Crochemore. Partitioning a graph in $O(|A| \log_2 |V|)$. *Theor. Comput. Sci.*, 19(1):85–98, 1982.

[CFI92] Jin-Yi Cai, Martin Fürer, and Neil Immerman. An optimal lower bound on the number of variables for graph identification. *Combinatorica*, 12(4):389–410, 1992.

[CTW13] Arkadev Chattopadhyay, Jacobo Torán, and Fabian Wagner. Graph isomorphism is not ac0-reducible to group isomorphism. *ACM Transactions on Computation Theory (TOCT)*, 5(4):1–13, 2013.

[DW22] Heiko Dietrich and James B Wilson. Group isomorphism is nearly-linear time for most orders. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 457–467. IEEE, 2022.

[GL21] Joshua A Grochow and Michael Levet. Weisfeiler-leman for group isomorphism: Action compatibility. *arXiv preprint arXiv:2112.11487*, 2021.

[GN20] Martin Grohe and Daniel Neuen. Recent advances on the graph isomorphism problem. *arXiv preprint arXiv:2011.01366*, 2020.

[IS19] Neil Immerman and Rik Sengupta. The k-dimensional Weisfeiler-Leman algorithm. *arXiv preprint arXiv:1907.09582*, 2019.

[Luk82] 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.

[Luk99] Eugene M Luks. Hypergraph isomorphism and structural equivalence of boolean functions. In *Proceedings of the thirty-first annual ACM symposium on Theory of computing*, pages 652–658, 1999.

[Mar02] Attila Maróti. On the orders of primitive groups. *Journal of Algebra*, 258(2):631–640, 2002.

[Mil78] Gary L Miller. On the nlog n isomorphism technique (a preliminary report). In *Proceedings of the tenth annual ACM symposium on theory of computing*, pages 51–58, 1978.

[Mil79] Gary L Miller. Graph isomorphism, general remarks. *Journal of Computer and System Sciences*, 18(2):128–142, 1979.

[Mil21] James S. Milne. Group theory (v4.00), 2021. Available at www.jmilne.org/math/.

[MP14] Brendan D McKay and Adolfo Piperno. Practical graph isomorphism, ii. *Journal of symbolic computation*, 60:94–112, 2014.

[Ser03] Ákos Seress. *Permutation group algorithms*. Number 152. Cambridge University Press, 2003.