The Graph Isomorphism Problem: An Introduction

By

Glen Woodworth

A Project Submitted in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in

Mathematics

University of Alaska Fairbanks

November 1, 2023

APPROVED:

Abstract

The graph isomorphism (GI) problem has plagued mathematicians and theoretical computer scientists for decades. The statement of the problem is simple: given any two graphs G and G', is G isomorphic to G'? However, no algorithm (currently) exists that can answer this question in $O(p(n))$ (polynomial time) for *any* two graphs, where n is the number of vertices in each graph. Babai presented an approach to GI that solves the problem in $\exp\left((\log n)^{\mathcal{O}(1)}\right)$ *Babai* [2016, 2018]. This bound is called *quasipolynomial* time; it is the best known bound for solving the graph isomorphism problem for any two graphs.

Babai's 2016 manuscript is a difficult read for many, even for those who have a graph theory or algebra background. Much of the machinery the *Babai* [2016] paper used was either developed specifically for solving GI over the last forty years or novel to that paper, which means understanding Babai's quasipolynomial result starts with specialized results from the 1980s, in particular, *Luks* [1982]. Therefore, the goal of this project is to give a rigorous introduction to Babai's quasipolynomial result, including an overview of motivating ideas and useful prerequisite knowledge. No graph theory knowledge is required, but general knowledge of mathematics at the graduate level is assumed.

Table of Contents

List of Figures

Acknowledgments

Special Thanks!!!

1 Introduction

This section contains preliminaries (1.1) and the definition of the GI problem (1.2). Section 2 introduces an algorithm for determining that two graphs are not isomorphic based on Weisfeiler-Lehman (WL) canonical refinement. Section 3 introduces the String Isomorphism Problem and shows how to change the GI problem into the SI problem. Section 4 contains an introduction to *Luks* [1982], including some of the algebra unerpinning Babai's approach. Last, Section 5 is the conclusion.

1.1 Preliminaries

Definition 1.1: A *simple graph* is an ordered pair $G = (V, E)$ where V is a set, called the vertices of G, and E is a set of unordered pairs of vertices from V, called the edges of G, such that for all $u \in V$, $\{u, u\} \notin E$.

Definition 1.2: For a graph $G = (V, E)$, if $\{u, v\} \in E$, we say u and v are *adjacent* or *neighbors* and we can write $u \leftrightarrow v$.

Definition 1.3: A *drawing* of a graph G is a geometric representation of G in the plane using points and line segments where each vertex of G is represented by exactly one unique point and a line segment connects two points P and Q in the drawing if and only if the vertices P and Q represent, respectively, are adjacent in G.

Example 1.1: We construct two simple graphs, G_1 and G_2 where

$$G_1 = (\{A, B, C, D, E, F\}, \{\{A, B\}, \{B, C\}, \{C, D\}, \{C, E\}, \{C, F\}\}) \text{ and}$$
$$G_2 = (\{1, 2, 3, 4\}, \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}\}).$$

See Figure 1.1. for a drawing of each of these graphs.

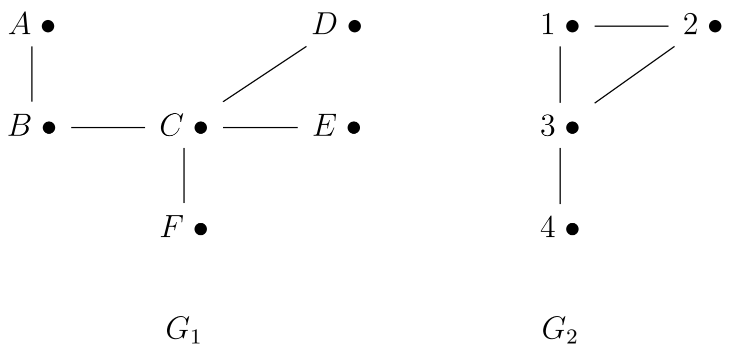


Figure 1.1: Examples of graphs

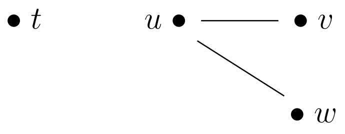


Figure 1.2: Adjacent and non-adjacent vertices

Remark: Graphs may have any number of vertices. However, unless otherwise specified, suppose any graph presented in this paper has a finite vertex set.

Definition 1.4: Let u be a vertex in a graph $G = (V, E)$. The *neighborhood of* u in G, denoted $N(u)$, is a subset of V containing all of the vertices in V that are adjacent to u in G. The cardinality of $N(u)$ is the *degree* of the vertex u, denoted $\deg(u)$.

Example 1.2: Consider the graph in Figure 1.2. There are two edges in this graph, $\{u, v\}$ and $\{u, w\}$. Thus, we may write $u \leftrightarrow v$ and $u \leftrightarrow w$. Note $N(u) = \{v, w\}$ so that $\deg(u) = 2$. Since t is not adjacent to any vertices, we may write $N(t) = \emptyset$ and $\deg(t) = 0$.

Definition 1.5: Given a graph $G = (V, E)$ where $V = \{1, 2, \ldots, n\}$, the *adjacency matrix* $A(G) = (a_{ij})$, where $a_{ij} = 1$ if vertex i and j are adjacent, otherwise $a_{ij} = 0$. If G is undirected and simple, $a_{ij} = a_{ji}$ and $a_{ii} = 0$.

Definition 1.6: Two graphs, $G = (V, E)$ and $G' = (V', E')$, are *isomorphic*, denoted

$G \cong G'$, if there exists a bijection $f : V \to V'$ such that for every $u, v \in V$, $\{u, v\} \in E$ if and only if $\{f(u), f(v)\} \in E'$. If such a function f exists, we call f a *graph isomorphism* from G to G' (if $G = G'$, we call f a *graph automorphism*).

Remark: In case we want the adjacency matrix of a graph $G = (V, E)$ where $|V| = n$ but $V \neq \{1, 2, \ldots, n\}$, there exists bijection $\phi : V \to \{1, 2, \ldots, n\}$ so that for $E' = \{\{\phi(u), \phi(v)\} : \{u, v\} \in E\}$ where $G' = (\{1, 2, \ldots, n\}, E')$, $G \cong G'$. Define $A(G) := A(G')$.

Definition 1.7: The *symmetric group* over a set A, $|A| = n$, is denoted $\mathrm{Sym}(A)$ when we require the elements of A explicitly or simply S_n otherwise.

Definition 1.8: If $A = (a_{ij})$ is an $(n \times m)$-matrix, $g \in \mathrm{Sym}(n)$ and $h \in \mathrm{Sym}(m)$, define $g^{-1}Ah = (a_{gi,hj})$ *Weisfeiler* [1976].

Remark: In the previous definition, $g^{-1}Ah$ is a matrix obtained by permuting the rows and columns of A by g and h, respectively.

Proposition 1.1: Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic if and only if there exists $\sigma \in \mathrm{Sym}(|V_1|)$ such that $\sigma^{-1}A(G_1)\sigma = A(G_2)$.

Proof. $\Rightarrow$ Suppose $G_1 \cong G_2$. Without loss of generality, $|V_1| = |V_2| = n$. There exist $A(G_1)$ and $A(G_2)$. For $k = 1, 2$, $A(G_k)$ induces a natural bijection $\phi_k : V_k \to \{1, 2, \ldots, n\}$ as in the remark above. For $u, v \in V_k$ where $i = \phi_k(u)$, $j = \phi_k(v)$, and $a_{ij} \in A(G_k)$

$$\{u, v\} \in E_k \text{ if and only if } a_{ij} = 1. \tag{1.1}$$

Since $G_1 \cong G_2$, there exist bijection $f : V_1 \to V_2$ such that $\{u, v\} \in E_1$ if and only if $\{f(u), f(v)\} \in E_2$. Define $\sigma := \phi_2 \circ f \circ \phi_1^{-1}$. Evidently $\sigma : \{1, 2 \ldots, n\} \to \{1, 2, \ldots, n\}$ is bijective as a composition of bijections. Thus, $\sigma \in S_n$. Let $a_{ij} \in A(G_1)$. There exist

$u, v \in V_1$ such that $u = \phi_1^{-1}(i)$ and $v = \phi_1^{-1}(j)$. Then $\sigma(i) = \phi_2(f(u))$ and $\sigma(j) = \phi_2(f(v))$. Since $\{u, v\} \in E_1$ if and only if $\{f(u), f(v)\} \in E_2$, applying (1.1) yields

$$a_{ij} = 1 \iff \{f(u), f(v)\} \in E_2. \tag{1.2}$$

Applying (1.1) again yields

$$\{f(u), f(v)\} \in E_2 \iff a_{\phi_2(f(u)),\phi_2(f(v))} = 1. \tag{1.3}$$

Since $\sigma(i) = \phi_2(f(u))$ and $\sigma(j) = \phi_2(f(v))$, we have

$$a_{ij} = 1 \iff a_{\sigma(i),\sigma(j)} = 1.$$

Hence, $\sigma^{-1}A(G_1)\sigma = A(G_2)$.

$\Leftarrow$ This direction mirrors the proof shown above and is left to the reader.

$\square$

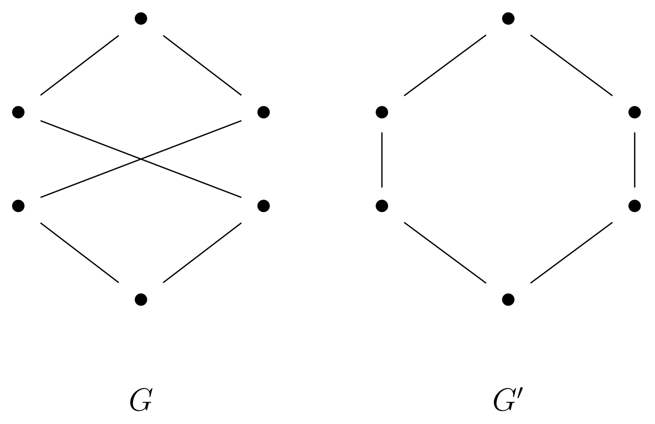


Figure 1.3: $G \cong G'$

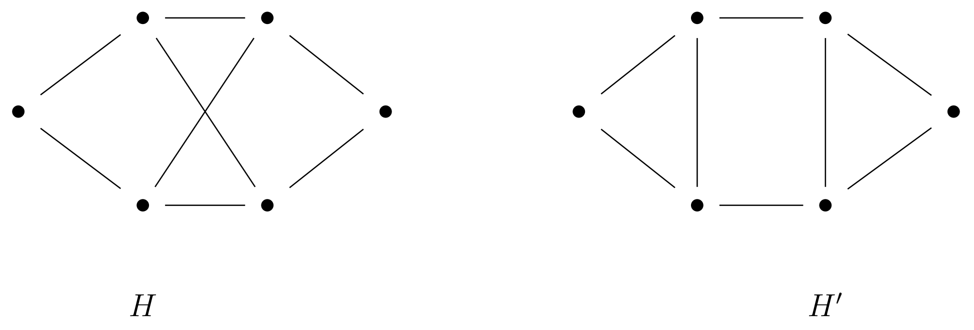


Figure 1.4: H is not isomorphic to H'. Observe that H' contains a 3-cycle and H does not.

Example 1.3: One way to think about two graphs being isomorphc is that two graphs are isomorphic if a drawing of one graph can be rearranged to another drawing without breaking or removing any edges to produce the other graph. See Figure 1.3 and Figure 1.4.

If two graphs are isomorphic, then they share every graph property, other than possibly the names of vertices. The contrapositive of that statement is as follows. If there exist any graph property that two graphs do not share, except the names of vertices, then those two graphs are *not* isomorphic. Therefore, knowledge of some properties of two graphs G and G' can sometimes allow us to easily determine that G and G' are not isomorphic. To that end, the following definitions are useful.

Definition 1.9: The *degree sequence* of a graph $G = (V, E)$ is a monotonic increasing

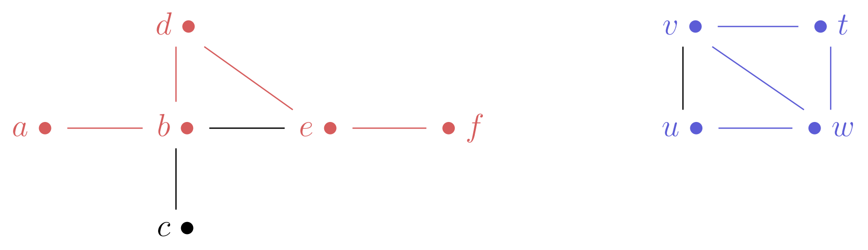


Figure 1.5: A disconnected graph $G = (V, E)$ where $V = \{a, b, c, d, e, f, u, v, t, w\}$ with a path and a walk identified

sequence of integers $d_1, d_2, \dots, d_n$ where $|V| = n$ and $d_i = deg(v_i)$ for all $v_i \in V$.

Definition 1.10: A *walk* in a graph $G = (V, E)$ is a list $(v_1, v_2, \dots, v_k)$ of vertices such that for $2 \leq i \leq k$, $\{v_{i-1}, v_i\} \in E$.

Example 1.4: In Figure 1.5, (u, w, t, v, w) is an example of a walk, shown in blue.

Definition 1.11: A $u, v-$*path* in a graph $G = (V, E)$ is a walk from vertex u to vertex v that has no repeated vertices.

Definition 1.12: The *length* of a $u, v-$path containing n vertices is $n - 1$ (i.e., the length of a path is the number of edges in the path). A *longest* $u, v-$path in a graph G is one containing a maximum number of vertices where u and v can be any vertices in G satisfying $u \neq v$. A $u, v-$path with length k is called P_k.

Example 1.5: Consider the graph in Figure 1.5. Define paths $P_4 = (a, b, d, e, f)$ and $P_3 = (u, w, t, v)$, where P_4 is shown in red and P_3 is a subset of the walk from Example 1.4 shown in blue.

Definition 1.13: A *n-cycle* (denoted C_n) is a walk on n vertices of the form

$$(v_1, v_2, v_3, \dots v_{n-1}, v_1)$$

such that for all $i, k \in \{1, 2, 3, \ldots, n-1\}$ where $i \neq k$, $v_i \neq v_k$.

Example 1.6: Consider the walk (u, w, t, v, w) shown in blue in Figure 1.5. Notice that the walk (w, t, v, w) contained in this blue walk is a $3-$cycle (C_3).

Definition 1.14: A *subgraph* of a graph $G = (V, E)$ is a graph $H = (V', E')$ where $V' \subset V$ and $E' \subset E$. If H is a subgraph of G, we write $H \subset G$.

Definition 1.15: A graph $G = (V, E)$ is *connected* if there exists a $u, v-$path for all distinct $u, v \in V$. Otherwise, we say G is *disconnected.*

Example 1.7: The graph in Figure 1.5 is not connected because there does not exist a $f, u-$path.

Example 1.8: Construct a subgraph $H = (V, E)$ of the graph shown in Figure 1.5 using the path P_4 shown in red by defining $V = \{a, b, d, e, f\}$ and $E = \{\{a, b\}, \{b, d\}, \{d, e\}, \{e, f\}\}$.

Definition 1.16: A *component* of a graph G is a connected subgraph $C \subset G$ that is not a subgraph of any other connected subgraph of G.

Example 1.9: Since the graph in Figure 1.5 is disconnected, there exists at least two components in that graph. Evidently, there are two: one component is the subgraph on the left, $K = (\{a, b, c, d, e, f\}, \{\{a, b\}, \{b, d\}, \{b, c\}, \{b, e\}, \{d, e\}, \{e, f\}\})$ and the other is the subgraph on the right, $W = (\{u, v, t, w\}, \{\{u, w\}, \{u, v\}, \{v, w\}, \{v, t\}, \{t, w\}\})$.

Definition 1.17: A simple graph $G = (V, E)$ where $|V| = n$ is *complete*, denoted $G = K_n$, if $|E| = n(n-1)/2$.

Example 1.10: See Figure 1.6 for an example of K_4.

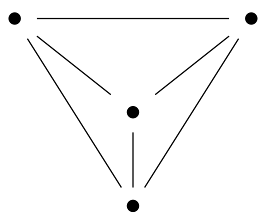
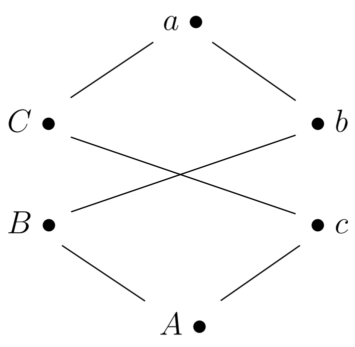


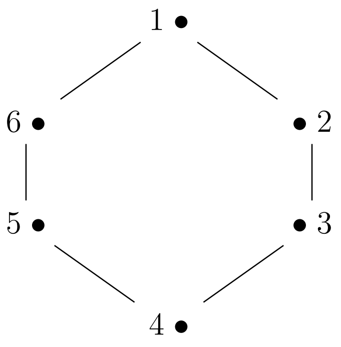
Figure 1.6: K_4

1.2 The Graph Isomorphism Problem

Definition 1.18: Given two graphs $G = (V, E)$ and $G' = (V', E')$, the *graph isomorphism problem* is to determine if there exists a graph isomorphism from G to G'.

Example 1.11: Given the two graphs $G = (V, E)$ and $G' = (V', E')$ shown below, we construct a bijection $f : V \to V'$ such that f is a graph isomorphism.





We define f below.

$$a \mapsto 1, \qquad A \mapsto 4,$$

$$b \mapsto 6, \qquad B \mapsto 5,$$

$$c \mapsto 3, \qquad C \mapsto 2.$$

Evidently f is a bijection where f^{-1} is defined as follows.

$$\begin{aligned} 1 &\mapsto a, & 4 &\mapsto A, \\ 2 &\mapsto C, & 5 &\mapsto B, \\ 3 &\mapsto c, & 6 &\mapsto b. \end{aligned}$$

To check that f is a graph isomorphism, we need to check that for all $v_1, v_2 \in V$, $\{v_1, v_2\} \in E$ if and if $\{f(v_1), f(v_2)\} \in E'$. Below is a table of all possible ordered pairs created by choosing elements from V.

	a	b	c	A	B	C
a	(a,a)	(a,b)	(a,c)	(a,A)	(a,B)	(a,C)
b	(b,a)	(b,b)	(b,c)	(b,A)	(b,B)	(b,C)
c	(c,a)	(c,b)	(c,c)	(c,A)	(c,B)	(c,C)
A	(A,a)	(A,b)	(A,c)	(A,A)	(A,B)	(A,C)
B	(B,a)	(B,b)	(B,c)	(B,A)	(B,B)	(B,C)
C	(C,a)	(C,b)	(C,c)	(C,A)	(C,B)	(C,C)

We may think that there are $|V| \times |V| = 36$ pairs to check. However, since G and G' are simple graphs, we do not have to check each pair. Specifically, we know that neither G nor G' have loops, meaning no entry from the diagonal, such as $\{a, a\}$, is in E nor is $\{f(u), f(u)\}$ in E' for any $u \in E$. Moreover, since the edges of G and G' are undirected, that is the edges are unordered rather than ordered pairs, we need only check either the upper or lower triangle of the table. Therefore, there are only $\binom{6}{2} = 15$ entries to check. This last step of verification that f is a graph isomorphism is left to the reader.

The graph isomorphism problem (GI) has a straightforward statement. Given any two graphs, $G = \{V, E\}$ and $G' = \{V', E'\}$, solving the GI problem requires either proving the existence of a graph isomorphism from V onto V' satisfying the conditions in Definition 1.5 or showing no such function exists. There are some instances where solving the GI problem is as simple as the statement of the problem. If $V = V$ and $E = E'$, then the identity map is a graph isomorphism. If two graphs are both complete and have the same number of vertices, then they are isomorphic. On the other hand, there are many scenarios where it is easy to tell that a graph isomorphism between two graphs cannot exist. For example, since it is impossible to produce a bijection between two sets of different finite sizes, if $|V| \neq |V'|$ then G is not isomorphic to G'.

Another way to think of a graph isomorphism is as a bijection between two graphs that preserves degree. Thus, if G is a graph with two vertices of degree 3 while G' only has a single vertex of degree 3, G cannot be isomorphic to G'. In fact, every graph property must be shared between two graphs for there to exist a graph isomorphism between them. The following list contains some examples of this.

1. If G is connected and G' is not connected, then G and G' are not isomorphic
2. If the length of the longest path in G is not the same as length of the longest path in G', then G and G' are not isomorphic.
3. If G has a 3-cycle while G' does not, then G and G' are not isomorphic.
4. If two graphs have different degree sequences, then they are not isomorphic.

When the given graphs, G and G', share several common properties (e.g., connected) and standard measures (e.g., longest path), solving the GI problem is more difficult. The problem becomes even more difficult when the graphs involved are regular and/or highly symmetric without being complete graphs. However, the issue at hand is not the difficulty of solving the GI problem for two specific graphs. The goal of researching the GI problem is to produce an algorithm or prove an algorithm exists that can efficiently solve the GI problem for *any* two simple graphs.

1.3 Algorithms

Definition 1.19: Given a set of graphs Γ, an *algorithm for graph identification* is an algorithm with a domain of $\Gamma \times \Gamma$ that, given an input (G_1, G_2), produces a result of $+1$ if $G_1 \cong G_2$ and -1 if not *Weisfeiler* [1976].

Let $\mathbb{A}$ be the set of all graph identification algorithms. The cardinality of $\mathbb{A}$ is large, perhaps infinite. We need a way to focus on the algorithms we care about, which is often the fast ones. We compare algorithms using the following definitions.

Definition 1.20: A *step* in an algorithm is any computation unit that is independent of the problem size *Mehta and Sahni* [2004].

Definition 1.21: The *running time* for algorithm $\mathcal{A}$ for graph identification is a function $f : \mathbb{A} \times \mathbb{N} \to \mathbb{N}$ where $f(\mathcal{A}, n)$ is the maximum number of steps required by $\mathcal{A}$ to find the result for any pair of graphs in the domain of $\mathcal{A}$ where n denotes the number of vertices in each graph *Weisfeiler* [1976].

When discussing the running time of an algorithm for graph identification, we are interested in how the running time increases as the input size grows arbitrarily large. The following definition is a standard way of understanding the magnitude of a monotonic function, which is useful for comparing the running times of algorithms.

Definition 1.22: For any monotonic functions $f, g : \mathbb{N} \to \mathbb{N}$, we write $f(n) = \mathcal{O}(g(n))$ if there exists $c, n_0 > 0$ such that for all $n \geq n_0$

$$f(n) \leq cg(n).$$

In such a case, we say "*$f(n)$ is on the order of $g(n)$*."

Any algorithm for graph identification solves the GI problem. Since the preceeding defintion gives us a standard way to measure the speed of algorithms, it is reasonable to ask, "What is the fastest such algorithm?" This is exactly the question that researchers of the GI problem are trying to answer. The following definitions formalize this.

Definition 1.23: For all sufficiently large n, the *problem of graph identification* is to find a graph identification algorithm $\mathcal{A}$ with a domain of consisting of all pairs of graphs on n vertices such that, for any other such algorithm $\mathcal{B}$,

$$f(\mathcal{A}, n) \leq f(\mathcal{B}, n)$$

Weisfeiler [1976].

Definition 1.24: A graph identification algorithm $\mathcal{A}$ *solves the problem of graph identification* if it satisfies the conditions of the preceding definition.

Definition 1.25: Given an algorithm for graph identification $\mathcal{A}$ that solves the problem of graph identification, $f(\mathcal{A}, n)$ is the *time complexity* of the GI problem.

Remark: The distinction between the GI problem and the problem of graph identification is for the benefit of the reader. Other authors may say "a solution to the GI problem" to mean "a solution to the problem of graph identification" or "a solution to the time complexity of the GI problem" in our context.

Using the preceding defintions, so long as we do not care about running time, it is a straightforward task to develop a graph identificaiton algorithm. We return to this is in Section 2. One of the goals of research into the Graph Isomorphism problem is to prove that there exists an algorithm for graph identification that runs in polynomial time or to prove that no such algoithm exists.

Definition 1.26: A graph identification algorithm for graphs on n vertices has *polynomial runtime* if for some $k > 0$, its running time is $\mathcal{O}(n^k)$.

Definition 1.27: A *quasipolynomial* function is a function of the form $\exp(p(\log n))$ for some polynomial p *Babai* [2018].

Remark: To help understand the maginutude of such fuctions, note

$$\mathcal{O}(\exp(p(\log n))) = \exp((\log n)^{\mathcal{O}(1)}) = n^{(\log n)^{\mathcal{O}(1)}}.$$

Definition 1.28: A problem P is *polynomial time reducible* to another problem P' if there exist an alogrithm with polynomial running time transforming P into P'.

No polynomial time graph identification algorithm has been found, nor has the graph identification problem been solved. That is, we have no proof that we have found an optimal graph identification algorithm, which would solve the problem of Definition 1.21. Regardless, the GI problem is thought to be of polynomial time complexity. Driving this intuition is not only Babai's breakthrough result, but also the fact that polynomial running time algorithms exist for broad classes of graphs. For example, the upper bound for the time complexity of the GI problem restricted to planar graphs with n vertices is $\mathcal{O}(n)$ *Hopcroft and Wong* [1974]. Progress has been made on GI in general by reducing GI to other problems in polynomial time, such as to the String Isomorphism problem (SI) (see Section 3). The most recent advance with respect to the the time complexity of the general GI problem is a result in *Babai* [2016, 2018], which, via a polynomial time reduction to SI, proves constructively that it is possible to solve GI for any pair of simple graphs in *quasi-polynomial time* (with respect to the length of an input string or the number of vertices).

Definition 1.29: An algorithm for graph identification $\mathcal{A}$ has *quasi-polynomial* run time if

$$f(\mathcal{A}, n) = \exp(p(\log n))$$

Babai [2016, 2018].

Babai's algorithm is the culmination of decades of research into algebraic graph theory and combinatorics with a focus on determining the time complexity of the GI problem. This project presents an introduction to three core areas required for understanding his algorithm.

The first area is canonical refinement, first presented for the purpose of graph identification in *Weisfeiler* [1976]; *Weisfeiler and Leman* [1968] where the Weisfeler-Leman canonical refinement process (WL) is presented, including examples as well as a few algorithms related to graph identification. WL is at the core of most theoretical and practical graph identification algorithms, including *Babai* [2016, 2018]. We present WL in the next section.

The second is the String Isomorphism problem (SI). In *Luks* [1982], the Graph Isomorphism problem is reduced to the String Isomorphism problem in polyomial time. In *Babai* [2016, 2018], the main theorem (Theorem 2.1) is that string isomorphism can be tested in quasipolynomial time. When combined with *Luks* [1982], the result is a proof that GI can be solved in quasipolynomial time as well. We present the SI with examples in Section 3.

The third is an algorithm for graph identification from *Luks* [1982], which pioneered a divide-and-conquer method. The domain of this algorithm is graphs where every vertex has degree less than or equal to some bound $d \geq 0$ *Luks* [1982]. In *Babai and Luks* [1983]; *Babai et al.* [1983], that result is combined with a combinatorial result from *Zemlyachenko et al.* [1982] to produce an upper bound of $\exp(\mathcal{O}(\sqrt{n \log n}))$ for the time complexity of the GI problem *Babai* [2018]. This algorthim is the backbone of *Babai* [2016, 2018]. Specifically, Babai's algorithm is identical Luks' algorithm until a bottleneck is reached. Babai's major contribution lies in dealing with the bottleneck, but in order to understand Babai's algorithm as a whole, we must understand Luks' first. We present an intrdouction to the main result of *Luks* [1982] in Section 4.

Remark: There are practical tools for efficiently solving the GI problem for two graphs, even if they have thousands or in some cases, millions, of vertices, see “Nauty” introduced in *McKay et al.* [1981].

2 Weisfeler-Lehman

2.1 Definitions

Definition 2.1: A *partial ordering* of a set S is a relation $<$ on S satisfying the following for all a, b, c in S:

1. Irreflexive: not $a < a$.
2. Asymmetric: If $a < b$ then not $b < a$.
3. Transitive: If $a < b$ and $b < c$ then $a < c$.

For all $a, b \in S$, if $a < b$ or $b < a$, we say a and b are *comparable*; otherwise,we say a and b are *incomparable.*

Definition 2.2: Given a graph $G = \{V, E\}$, a *canonical labeling* of V is a partial ordering of V such that if vertices a and b are incomparable, then there is a graph automorphism sending a to b *Weisfeiler and Leman* [1968].

Definition 2.3: Given a graph $G = \{V, E\}$ and an algorithm $\mathcal{A}$ for obtaining a canonical labeling of G, a *canonical form* of G is its adjacency matrix, $A(G)$, with respect to a canonical labeling of its vertices, denoted $Canon_{\mathcal{A}}(G)$ *Weisfeiler and Leman* [1968].

2.2 A Canonization Algorithim

WL is a classic and important method for partitioning the vertices of a graph G based on the automorphism group of G *Weisfeiler* [1976]; *Weisfeiler and Leman* [1968]. Since the original 1968 paper, the power of WL has been improved. These improvements, called k-WL, where k is called the dimension, can identify a larger class of graphs. For our purposes, an understanding of the original WL, also known as 1-WL, is sufficient. The usefulness of WL depends primarily on the answers to the following questions. Give any finite graph G:

1. Does a canonical labelling for G exist and will WL produce one?

2. Is a canonical labelling for G unique?

The answer to (1) is yes. In *Weisfeiler and Leman* [1968], it is proved that WL produces a canonical labelling for any finite graph G. With respect to (2), the answer is no, which will be shown shortly. However, producing a canonical labelling is still useful. For many pairs of graphs, after they are each reduced to a canonical form using the same method, it is often straightforward to check that they are not isomorphic. WL or any method for producing a canonical labelling of a graph, gives rise to an algorithm for graph identification, as we will see shortly *Weisfeiler* [1976].

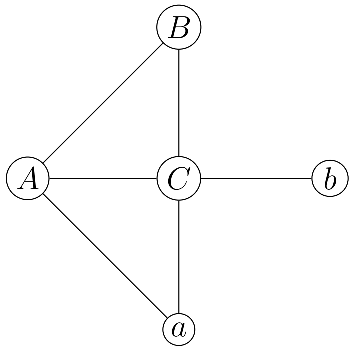
The following definition is useful for understanding WL.

Definition 2.4: Let $G = \{V, E\}$ be a graph and $\{l_1, l_2, \ldots, l_n\}$ be a partition of V into n classes. For any $k \in \{1, 2, \ldots, n\}$ and any vertex $u \in l_k$ the **characteristic vector** of u, denoted $cv(u)$, is a $(n+1)$-tuple defined by $cv(u) = (k, u_1, u_2, \ldots, u_n)$ where $u_i \in \mathbb{Z}_{\geq 0}$ for $i \in \{1, 2, \ldots, n\}$ is the number of neighbors u has in class l_i *Weisfeiler and Leman* [1968].

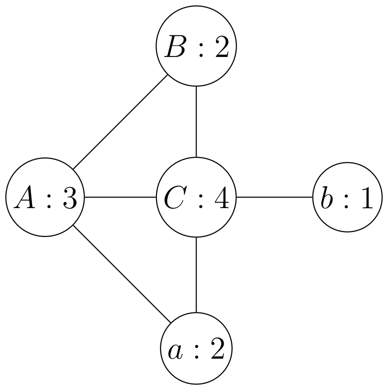
Remark: Evidently $n \leq |V| + 1$.

To reduce a graph to a canonical form, *Weisfeiler and Leman* [1968] propose the following iterative procedure. Initially, each vertex u is labelled by its degree. These labels determine the initial class for each vertex. Note that there is a natural order on these classes, as they are integers. An iteration of the algortihm proceeds as follows. Generate a characterstic vector for each vertex. Next, order the set of characteristic vectors lexicographically. Last, starting with the next highest integer that has not been used as a label, relabel the vertices based on the lexicographic order. Iterate until the classes stabilize, that is, until subsequent iterations would not change the classes. For any simple graph, this algorithm will always terminate and produce a canonical labelling *Weisfeiler and Leman* [1968]. The following example demostrates the initialization and one iteration of this proceudre.

Example 2.1: Consider the following graph $G = \{V, E\}$ where $V = \{A, B, C, a, b\}$ and $V = \{(A, B), (A, C), (A, a), (B, C), (C, a), (C, b)\}$.



We label each vertex in G by degree.

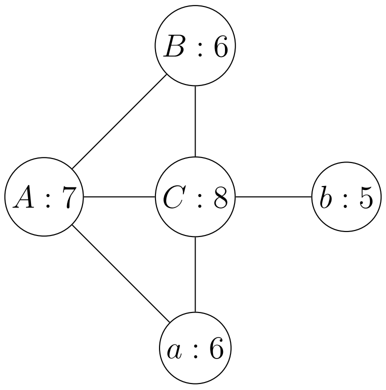


Our partition of V is $P = \{l_1, l_2, l_3, l_4\}$ where $l_1 = \{b\}$, $l_2 = \{a, B\}$, $l_3 = \{A\}$, and $l_4 = \{C\}$. As an example, note $N(A) = \{B, C, a\}$. Therefore, A has two neighbors in l_2, namely, a and B, and one neighbor in l_4, C. Thus, $cv(A) = (3, 0, 2, 0, 1)$. The reader is encouraged to construct the characterstic vectors for the remaining vertices. To check your work, the results are shown in the table below.

vertices	A	B	C	a	b
cv	(3,0,2,0,1)	(2,0,0,1,1)	(4,1,2,1,0)	(2,0,0,1,1)	(2,0,0,0,1)

The last step of an iteration is to order the vectors lexicographically. We assign a new label to each vertex based on this ordering. Since 5 is the next integer after our initial 4 classes, first vertex in the order receives a new label of 5. Continue until every vertex has been relabelled. For example, vertex b with characteristic vector $(2,0,0,0,1)$ receives a new label of 5. The result of applying this ordering and relabelling process to the vertices and characteristic vectors of this example is shown in the table and graph below.

vertices	A	B	C	a	b
cv	(3,0,2,0,1)	(2,0,0,1,1)	(4,1,2,1,0)	(2,0,0,1,1)	(2,0,0,0,1)
new label	7	6	8	6	5



We formalize the steps in the following algorithm.

WL

Input: $G = \{V, E\}$

Initialize: Label vertices using degree

Iterate: Produce characteristic vectors, order lexicographically and relabel

Output: Graph G with a canonical labelling of the vertex set

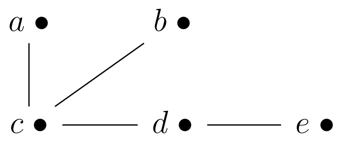
While it is likely clear that comparable vertices (those with different labels) in the output of WL are different, if two vertices u and v end up with the same label, it is not obvious that there exists an automorphism sending u to v. However, this non-trivial result is shown in *Weisfeiler and Leman* [1968]. Hence, we can be sure that WL results in a canonical labelling for any simple graph.

Remark: In *Weisfeiler and Leman* [1968], it is shown that WL will produce a canonical labelling for graphs with loops and/or multiple edges, not just simple graphs.

The following example applies WL Canonical Labeling until a canonical labelling is produced.

Example 2.2: Given the graph G below, we follow WL canonical labelling until the classes stabilize.

Input:

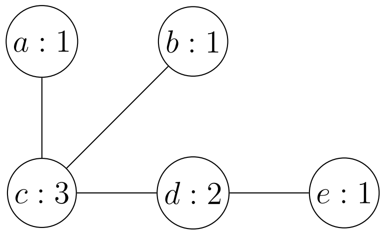


Ordering the vertices of G alphabetically, the adjaceny matrix of G is as follows.

$$A(G) = \begin{array}{c|ccccc} & a & b & c & d & e \\ \hline a & 0 & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 1 & 0 & 0 \\ c & 1 & 1 & 0 & 1 & 0 \\ d & 0 & 0 & 1 & 0 & 1 \\ e & 0 & 0 & 0 & 1 & 0 \end{array}$$

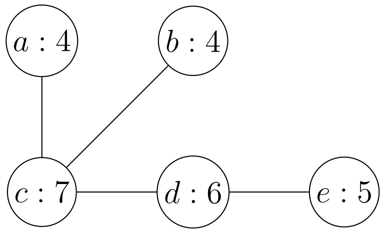
I. Initialize:

classes	label
{c}	3
{d}	2
{a,b,e}	1



II. Generate Characteristic Vectors & Relabel:

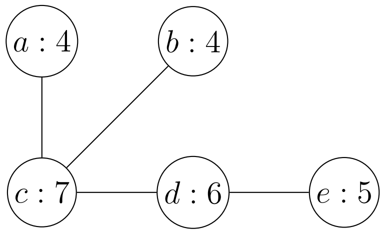
classes	cv	relabel
{c}	(3,2,1,0)	7
{d}	(2,1,0,1)	6
{e}	(1,0,1,0)	5
{a,b}	(1,0,0,1)	4



III. Iterate:

Since the classes stay the same, the process terminates here.

classes	cv
{c}	(7,2,0,1,0)
{d}	(6,0,1,0,1)
{e}	(5,0,0,1,0)
{a,b}	(4,0,0,1,0)



The following is a canonical form of G generated by WL.

$$Canon(G) = \begin{array}{c|ccccc|} & a & b & e & d & c \\ a & 0 & 0 & 0 & 0 & 1 \\ b & 0 & 0 & 0 & 0 & 1 \\ e & 0 & 0 & 0 & 1 & 0 \\ d & 0 & 0 & 1 & 0 & 1 \\ c & 1 & 1 & 0 & 1 & 0 \end{array}$$

Notice that a and b are in the same class, that is, a and b are incomparable. Since the relevant result from *Weisfeiler and Leman* [1968] mentioned above, ensures that this is a canonical form of G. Hence, there exist a graph automorphism sending a to b. The function, $f : \{a, b, c, d, e\} \to \{a, b, c, d, e\}$, defined by

$$f(u) = \begin{cases} b, \ u = a \\ a, \ u = b \\ u, \ u \neq a, b \end{cases}$$

is such a function.

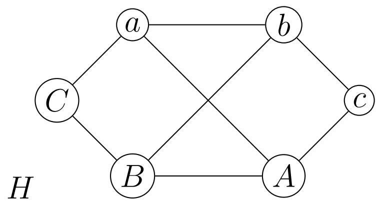
Remark: Since WL iteratively produces partitions of vertex sets, it is also called *color refinement*. The color of a vertex is its class at any given iteration.

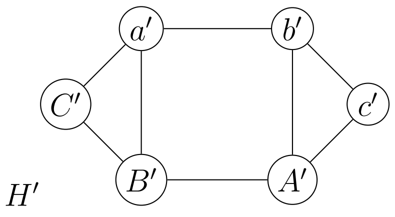
2.3 A Non-Isomorphism Test

As mentioned in the previous subsection, we can use a canonization algorithm to produce a graph identification algorithm. However, unless we add extra machinery, canonization is not sufficient for graph identification. In that case, we can only hope to produce an algorithm that can determine two graphs are not isomorphic. In what follows, we present such an algorithm based on WL, the *WL Non-Isomorphism Test*, adopted from *Shervashidze et al.* [2011]. This test differs from WL in two key ways. First, the input of WL Non-Isomorphism Test is two graphs rather than a single graph. WL is applied to each graph at the same time, where new vertex labels for both graphs come from same alphabet. Second, at initialization, a label string is generated for each graph G, denoted $s(G)$, using the initial labels as follows. The k^{th} entry in a label string for a graph G is the number of vertices of G with label k. After each iteration, a new label string is generated based on the new labels, then concatenated to the end of the old string. If the graphs have different strings at any step then the algorithm terminates immediately, returning that the two graphs are not isomorphic.

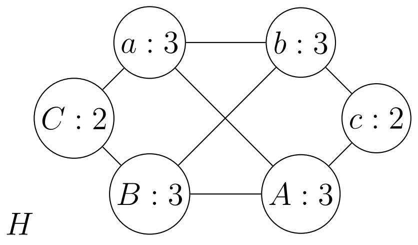
```
                 **WL Non-Isomorphism Test**
      Input: Graphs $G = \{V, E\},\ G' = \{V', E'\}$
 Initialize: Label the vertices using degree
             Generate and compare label strings
             If strings are not equal, terminate
    Iterate: Produce characteristic vectors for all vertices in $G$ and $G'$
             Order characteristic vectors, relabel
             If classes did not change, terminate
             Generate new label string
             If strings are not equal, terminate
     Output: A graph isomorphism from $G$ to $G'$ does not exist
             or NULL
```

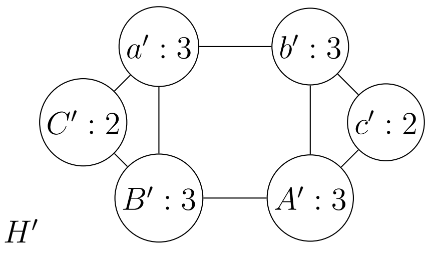
Example 2.3: This example applies the WL Non-Isomorphism Test to H and H'.





I. Initialize: Label the vertices by their degree and construct label strings.





classes	label
{a,b,A,B}	3
{c,C}	2
{}	1

$s(H) = (024)$

classes	label
{a',b',A',B'}	3
{c',C'}	2
{}	1

$s(H') = (024)$

II. Generate Characteristic Vectors & Relabel:

Notice that the labels do not change from the previous step, so the process terminates after relabelling.

classes	cv	label
{a,b,A,B}	(3,0,1,2)	5
{c,C}	(2,0,0,2)	4
{}		

classes	cv	label
{a',b',A',B'}	(3,0,1,2)	5
{c',C'}	(2,0,0,2)	4
{}		

Recall that H' has a 3 cycle while H does not. Thus, H and H' are not isomorphic. Since the algorithm terminates on a step where the two graphs have the same number of classes and the same number of vertices in each class, the WL Non-Isomorphism Test reports nothing. It is unable to determine that H and H' are not isomorphic.

Definition 2.5: An algorithm *distinguishes* two graphs G and G' if the algorithm can determine that G and G' are not isomorphic.

In the previous example, we say that the Naive WL isomorphism Test does not *distinguish* H and H'.

2.4 Simple Algorithms for Graph Identification based on WL

For each of the following algorithms, the domain is all simple graphs on n vertices. Two slow but straightforward algorithms are presented. The input for each algorithm is two simple graphs $G_k = (V_k, E_k)$, $k = 1, 2$.

2.5 Brute Force

After initialization, the brute force algorithm outlined below operates as follows. Pick $\sigma \in \text{Sym}(V_1)$. Construct $\sigma(G_1)$ based on the following definition.

Definition 2.6: For a graph $G = (V, E)$ and $\sigma \in \text{Sym}(V)$, define $\sigma(V) := \{\sigma(v) : v \in V\}$, $\sigma(E) := \{\{\sigma(u), \sigma(v)\} : \{u, v\} \in E\}$, and $\sigma(G) := (\sigma(V), \sigma(E))$.

Iterate this process until $A(\sigma(G_1)) = A(G_2)$ is true or $\text{Sym}(V_1)$ is exhausted.

Brute Force Graph Identification Algorithm

```
Input: $G_1 = (V_1, E_1),\ G_2 = (V_2, E_2)$
Initialize: Construct $A(G_1),\ A(G_2)$ and generate $\text{Sym}(V_1) := \{\sigma_k\}_{k=0}^{|V_1|!}$
Iterate: On the $k^{th}$ iteration, for $\sigma_k \in \text{Sym}(V_1)$ compute $A(\sigma_k(G_1))$
         and evaluate $A(\sigma_k(G_1)) = A(G_2)$.
         If $A(\sigma_k(G_1)) = A(G_2)$ True, return $+1$
Output: $+1$ if returned during Iterate; $-1$ otherwise
```

Remark: On graphs with large order, even the initialization step of this algorithm is prohibively time consuming because generating $\mathrm{Sym}(V_1)$ takes at least $|V_1|!$ steps. However, the rest of the algorithm is even slower. Slow enough that the time lost in the initialization step is inconsequential in the running time analysis of the algorithm as a whole.

We now evaluate the run time of this algorithm. First, the adjacency matrix for each graph is created and $\mathrm{Sym}(V_1)$ is generated. Without loss of generality, suppose $|V_1| = |V_2| = n$. Since the adjacency matrix is symmetric and has zeros on the diagonal, creating the adjacency matrices requires $2\binom{n}{2} + 2 = \mathcal{O}(n^2)$ steps. To generate $\mathrm{Sym}(V_1)$, we first label the vertices in V_1 using integers $1, 2, \ldots, n$ then apply an algorithm with an optimal run time of $\mathcal{O}((n+1)!)$ (see *Heap* [1963]; *Johnson* [1963]). Next, in the kth iteration, the algorithm checks if $A(\sigma_k(G_1)) = A(G_2)$. This step requires $\binom{n}{2} = \mathcal{O}(n^2)$ comparisons to ensure each unique vertex pair is checked. Since we must iterate over every possible permutation, we repeat this step $n!$ times. As n gets large, the dominating factor with respect to steps is $n!\binom{n}{2}$. We can estimate the runtime as follows.

$$n!\binom{n}{2} = \frac{n!n!}{(n-2)!2} = \frac{n!(n)(n-1)}{2} = \mathcal{O}((n+2)!).$$

Thus, the worst-case runtime of this algorithm is $\mathcal{O}((n+2)!)$. This should intuitively make sense because there are $n!$ possible permutations and $\binom{n}{2}$ entries to check for each permutation, where $\binom{n}{2} \approx n^2$ for large n. See Appendix for a formal proof.

Although the Brute Force Graph Identification Algorithm (BF) is an upper bound on the complexity of the GI, this bound is useful only as a tool for understanding the vastness of the problem. An algorithm with a factorial run time is essentially useless.

2.6 Brute Force with WL

If we combine BF with an algorithm to produce a canonical labelling, we may reduce the number of permutations we need to check, resulting in a reduction in overall running time. This subsection presents a brute force algorithm for graph identification using WL to produce a canonical labelling. We call this algorithm BFWL.

BFWL is a natural step towards a faster graph identification algorithm. The idea is to use WL to reduce the number of permutations we need to check by brute force. Input the graphs into WL Non-Isomorphism Test, but store the canonical labelling that is produced. If WL Non-Isomorphism Test reports that the graphs are not isomorphic, terminate. Otherwise, the algorithm begins BF, but with a restricition on the elements of Sym(V_1) to check. We are only interested in the permutations that preserves the class of every vertex. Specifically, we want $\sigma \in$Sym(V_1) such that for all the vertex classes l and for all $v \in V_1$, v is in class l if and only if $\sigma(v)$ is in class l.

Brute Force with WL Algorithm

```
      Input: $G_1 = (V_1, E_1),\ G_2 = (V_2, E_2)$
 Initialize: Run WL Non-Isomorphism Test, but store the canonical labelling
             Construct canon($G_k$), $k = 1, 2$ using the canonical labelling
             Generate $H \leq$ Sym($V_1$) such that for all $\sigma \in H$,
             $\sigma$ respects the classes of the canonical labelling.
    Iterate: On the $k^{th}$ iteration, for $\sigma_k \in H$ compute $A(\sigma_k(G_1))$
             and evaluate $\sigma_k(canon(G_1)) = canon(G_2)$.
             If $\sigma_k(canon(G_1)) = canon(G_2)$ True, return $+1$
     Output: $+1$ if returned during Iterate; $-1$ otherwise
```

We now evaluate the running time of Brute Force with WL Algorithm (BFWL). Since BFWL only chance at improving on BF is by the success of WL at reducing the amount of permutations to check, we consider several cases, including the extremal cases (worst and best possible) as well as a general case, each based on the number of canonical classes produced by WL. In each case, we suppose that WL results in the same amount of classes and the number of vertices per class for canon(G_1) and canon(G_2), for otherwise each case would terminate immediately after WL, reporting G_1 is not isomorphic to G_2.

For the first case, suppose the output of WL is a single class. If each graph has this same single class, this is the worst possible result. Any vertex may be sent to any other vertex by a permutation while still respecting the canonical classes. The impact is that BFWL must search the entire permutation group, which means the running time will be equivalent to BF.

For the second case, suppose the number of canonical classes is equal to the number of vertices, that is, every vertex receives their own class. This is the best possible case because any graph isomorphism between two graphs must preserve WL canonical classes. Therefore, for any vertex v in G_1 with canonical class l, there exist only one vertex u in G_2 that a grah isomorphism from G_1 to G_2 could send v to. Hence, if the two graphs are not identified as non-isomorphic by WL Non-Isomorphism test, this case ensures there exist exactly one function from G_1 to G_2 that could be a graph isomorphism. Since this check takes only $\binom{n}{2}$ steps (where n is the number of vertices of the input graphs), the running time is equivalent to WL Non-Isomorphism test.

For a third, more general case, suppose the number of canonical classes is equal to m such that $m|n$ where n is the number of vertices of the input graphs. If we also require that the classes be equal in size, the running time of BFWL becomes

$$m\left(\frac{n}{m}\right)!\binom{n}{2}.$$

Depending on the size of m, this can be a large or a small speed up over BF. For example, if n is even and $m = 2$ where each class has size $n/2$, we have

$$2\left(\frac{n}{2}\right)!\binom{n}{2} = 2\frac{(n/2)!n!}{(n-2)!2} = (n/2)!(n)(n-1) = \mathcal{O}((n/2)!n^2).$$

3 The String Isomorphism Problem

3.1 A graph as a string

Following Luks [1982], a graph can be represented as a binary string using the indicator function for adjacency relations in the graph. Let $\Omega = [1, \ldots, n]$. Let $G = \{\Omega, E\}$ be an undirected, simple graph. Let $\binom{\Omega}{2}$ denote the set of all unordered pairs in Ω (Babai 2018). Let $\delta_G : \binom{\Omega}{2} \to \{0, 1\}$ be the indicator function for the adjacency relations in G, defined by

$$\delta_G(\{x,y\}) = \begin{cases} 1, & \{x,y\} \in E(G) \\ 0, & \{x,y\} \notin E(G) \end{cases}.$$

Then δ_G is the binary string representation of G.

Definition 3.1: $\mathrm{Sym}(\Omega)$ (denoted S_Ω) is the group of all bijections $f : \Omega \to \Omega$.

From Definition 1.6, we know that two graphs G and G' are isomorphic if there exists a bijection $f : V(G) \to V(G')$ such that for all $u, v \in V(G)$, $\{u, v\} \in E(G)$ if and only if $\{f(u), f(v)\} \in E(G')$. Since all of the graphs we are discussing are labelled using $[n]$, every isomorphism between two graphs is a function from S_n. In order to define the string isomorphism problem, the following definition is important.

Definition 3.2:
$S_n^{(2)} = \left\{\sigma \in \mathrm{Sym}\ \left(\binom{\Omega}{2}\right) : \exists f \in S_n \text{ s.t. } \forall \omega = \{u,v\} \in \binom{\Omega}{2},\ \sigma(\omega) = \{f(u), f(v)\}\right\}.$

Notice that $S_n^{(2)} \subseteq \mathrm{Sym}(\binom{\Omega}{2})$. There is an added restriction on the bijections in $S_n^{(2)}$ compared to those in $\mathrm{Sym}(\binom{\Omega}{2})$. Every permutation of unordered pairs (i.e. permutation of edges) in $S_n^{(2)}$ can be obtained using a permutation in S_n (i.e. a permutation of the vertices). When $0 < n \leq 3$, $S_n^{(2)}$ $=\mathrm{Sym}(\binom{\Omega}{2})$. When $n > 3$, the two groups are no longer equal. Consider the following example.

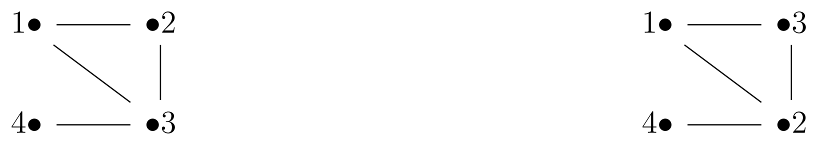
Example 3.1: Let

$$\sigma = \begin{pmatrix} \{1,2\} & \{1,3\} & \{1,4\} & \{2,3\} & \{2,4\} & \{3,4\} \\ \{1,3\} & \{1,2\} & \{1,4\} & \{2,3\} & \{2,4\} & \{3,4\} \end{pmatrix}.$$

Notice that $\sigma \in \mathrm{Sym}\big(\binom{4}{2}\big)$. To see that $\sigma \notin S_4^{(2)}$, we must show that there is no function $f \in S_4$ such that for all $u, v \in [4]$, $\sigma(\{u,v\}) = \{f(u), f(v)\}$. Although there are 24 elements in S_4, there is only one element in S_4 (see f_1 below) that results in mapping $\{1,2\} \to \{1,3\}$ and $\{1,3\} \to \{1,2\}$. We have

$$f_1 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 3 & 2 & 4 \end{pmatrix}.$$

Notice that $\sigma(\{3,4\}) = \{3,4\} \neq \{2,4\} = \{f_1(3), f_1(4)\}$. Since this is the only possible bijection for σ to satisfy the conditions of $S_4^{(2)}$, we know $\sigma \notin S_4^{(2)}$. To make this point more clear, consider the graphs on four vertices below.



(a) G (b) $f_1(V(G))$

Figure 3.1: G and the result of applying f_1 to V(G).

Notice that $\{2,4\} \in E(f_1(V(G)))$ but $\{2,4\} \notin E(G)$

Definition 3.3: Let δ_1 and δ_2 be string representations of undirected, simple graphs G_1 and G_2, respectively. Then δ_1 and δ_2 are $S_n^{(2)}$-*isomorphic*, denoted $\delta_1 \cong \delta_2$, if there exists $\sigma \in S_n^{(2)}$ such that $\delta_1 \circ \sigma = \delta_2$.

Given binary strings δ_1 and δ_2, both with length n, the *string isomorphism problem* asks: Does there exist $\sigma \in S_n^{(2)}$ such that $\delta_1 \circ \sigma = \delta_2$? The following example answers this question

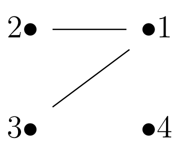
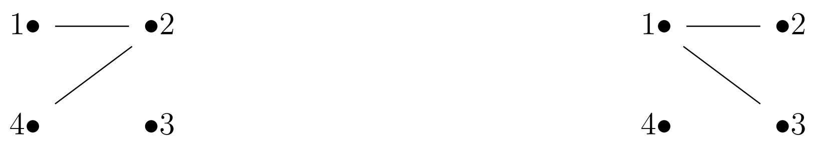


Figure 3.3: G_3, with vertex set $f_2(V(G_1))$.

for a binary string with length 6.



(a) G_1 (b) G_2

Figure 3.2: Two isomorphic graphs on 4 vertices.

Example 3.2: Consider graphs G_1 and G_2 in Figure 3qgis .2. In this case, we consider $\Omega = [1, 2, 3, 4]$. A bijection that produces an isomorphism between G_1 and G_2 is f_2,

$$f_2 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 1 & 4 & 3 \end{pmatrix}.$$

If we apply f_2 to $V(G_1)$, the resulting graph is G_3 (Figure 2.3). We now construct the binary string representation of each graph. We have

$$\delta_{G_1} = \begin{pmatrix} \{1,2\} & \{1,3\} & \{1,4\} & \{2,3\} & \{2,4\} & \{3,4\} \\ 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

and

$$\delta_{G_2} = \begin{pmatrix} \{1,2\} & \{1,3\} & \{1,4\} & \{2,3\} & \{2,4\} & \{3,4\} \\ 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

We also have

$$\text{Sym}\ (\Omega) = S_4.$$

To show that $\delta_{G_1} \cong \delta_{G_2}$, we must find $\sigma \in S_4^{(2)}$ such that $\delta_{G_1} \circ \sigma = \delta_{G_2}$. We simply record the effect on $E(G_1)$ of applying f_2 to $V(G_1)$. We have

$$\sigma = \begin{pmatrix} \{1,2\} & \{1,3\} & \{1,4\} & \{2,3\} & \{2,4\} & \{3,4\} \\ \{1,2\} & \{2,4\} & \{2,3\} & \{1,4\} & \{1,3\} & \{3,4\} \end{pmatrix}.$$

Then

$$\delta_{G_1} \circ \sigma = \begin{pmatrix} \{1,2\} & \{1,3\} & \{1,4\} & \{2,3\} & \{2,4\} & \{3,4\} \\ \{1,2\} & \{2,4\} & \{2,3\} & \{1,4\} & \{1,3\} & \{3,4\} \\ 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} = \delta_{G_2},$$

as desired. Hence, $\delta_{G_1} \cong \delta_{G_2}$.

Lemma 3.1: $S_n^{(2)}$ is a subgroup of $\mathrm{Sym}(\binom{[n]}{2})$.

Proof. Let $\sigma, \eta \in S_n^{(2)}$. Since $S_n^{(2)} \subseteq \mathrm{Sym}\left(\binom{[n]}{2}\right)$, $\sigma, \eta \in \mathrm{Sym}\left(\binom{[n]}{2}\right)$. Moreover, $\mathrm{Sym}\left(\binom{[n]}{2}\right)$ is a group. Hence, $\sigma \circ \eta \in \mathrm{Sym}\left(\binom{[n]}{2}\right)$. Now, by definition there exists $f_\sigma,\ f_\eta \in S_n$ such that for all $\omega = \{u, v\} \in \mathrm{Sym}(\binom{[n]}{2})$, $\sigma(\{u,v\}) = \{f_\sigma(u), f_\sigma(v)\}$ and $\eta(\{u,v\}) = \{f_\eta(u), f_\eta(v)\}$. Since $f_\sigma, f_\eta \in S_n$, $f_\sigma \circ f_\eta \in S_n$. Let $\omega = \{u,v\} \in \mathrm{Sym}(\binom{[n]}{2})$. Consider $\sigma \circ \eta(\{u,v\})$. We have

$$\sigma \circ \eta(\{u,v\}) = \sigma(\{f_\eta(u), f_\eta(v)\}) = \{f_\sigma \circ f_\eta(u), f_\sigma \circ f_\eta(v)\}.$$

Hence, $\sigma \circ \eta \in S_n^{(2)}$. Lastly, consider $f_\sigma^{-1} \in S_n$. Let $\sigma^{-1}(\{u,v\}) = \{f_\sigma^{-1}(u), f_\sigma^{-1}(v)\}$. Hence, $\sigma^{-1} \in S_n^{(2)}$. Observe,

$$\sigma \circ \sigma^{-1}(\{u,v\}) = \sigma(\{f_\sigma^{-1}(u), f_\sigma^{-1}(v)\}) = \{f_\sigma \circ f_\sigma^{-1}(u), f_\sigma \circ f_\sigma^{-1}(v)\} = \{u,v\}.$$

By a similar argument, $\sigma^{-1}\sigma(\{u,v\}) = \{u,v\}$. Hence, $S_n^{(2)}$ is a subgroup of $\mathrm{Sym}(\binom{[n]}{2})$. □

Lemma 3.2: Two graphs are isomorphic if and only if their string representations are $S_n^{(2)}$-isomorphic.

Proof. Let δ_{G_1} and δ_{G_2} be binary string representations of undirected graphs G_1 and G_2, respectively.

$\Rightarrow$

Suppose $G_1 \cong G_2$. By Definition 1.6, there exists bijection $g : V(G_1) \to V(G_2)$ such that for all $u, v \in V(G_1)$, $\{u, v\} \in E(G_1)$ if and only if $\{g(u), g(v)\} \in E(G_2)$. Notice that $g \in S_n$. Let $\sigma : \binom{[n]}{2} \to \binom{[n]}{2}$ where $\sigma(\{u, v\}) = \{g(u), g(v)\}$. Then $\sigma \in S_n^{(2)}$. Let $u, v \in [n]$. Consider $\delta_{G_2} \circ \sigma(\{u, v\})$. We have

$$\delta_{G_2} \circ \sigma(\{u, v\}) = \delta_{G_2}(\{g(u), g(v)\}) = \delta_{G_1}(\{u, v\}).$$

The last equality follows because $\{u, v\} \in E(G_1)$ if and only if $\{g(u), g(v)\} \in E(G_2)$. This means $\delta_{G_1}(\{u, v\}) = 1$ if and only if $\delta_{G_2}(\{g(u), g(v)\}) = 1$. Hence, $\delta_{G_2} \cong \delta_{G_1}$.

$\Leftarrow$

Suppose $\delta_{G_1} \cong \delta_{G_2}$. Note that for any $\sigma \in S_n^{(2)}$, there exist $f_\sigma \in S_n$ such that $\sigma(\{u, v\}) = \{f_\sigma(u), f_\sigma(v)\}$ for any $\{u, v\} \in \binom{[n]}{2}$. Since $\delta_{G_1} \cong \delta_{G_2}$, by Definition 3.3, there exists $\hat{\sigma} \in S_n^{(2)}$ such that $\delta_{G_1} \circ \hat{\sigma} = \delta_{G_2}$. We have

$$\delta_{G_2}(\{u, v\}) = \delta_{G_1} \circ \hat{\sigma}(\{u, v\}) = \delta_{G_1}(\{f_{\hat{\sigma}}(u), f_{\hat{\sigma}}(v)\}).$$

This means that $\delta_{G_2}(\{u, v\}) = 1$ if and only if $\delta_{G_1}(\{f_{\hat{\sigma}(u)}, f_{\hat{\sigma}(v)}\}) = 1$. In other words, $\{u, v\} \in E(G_2)$ if and only if $\{f_{\hat{\sigma}(u)}, f_{\hat{\sigma}(v)}\} \in E(G_1)$. Hence, $G_1 \cong G_2$. $\square$

4 Introduction to Luks' Algorithm

4.1 Definitions

The following defintions come directly from *Luks* [1982]

Definition 4.1: A subset G of $\mathrm{Sym}(A)$ is said to *stabilize* $B \subseteq A$ if for all $\sigma \in G$, $\sigma(B) = B$. In such a case, we call the set $G_B = \{\sigma \in G : \sigma(B) = B\}$ the *stabilizer* of B in G.

Definition 4.2: If $G \subseteq \mathrm{Sym}(A)$ acts on $B \subseteq A$ and there exist $b \in B$ such that $B = \{\sigma(b) : \sigma \in G\}$, we say G acts *transitively* on B.

Definition 4.3: If $G \subseteq \mathrm{Sym}(A)$ acts transitively on A, B is a nonempty proper subset of A, and for all $\sigma, \tau \in G$, $\sigma(B) = \tau(B)$ or $\sigma(B) \cap \tau(B) = \emptyset$, then we say B is a *G-block*.

Remark: An alternate but equivalent definition for a G-block is that B is a G-block if for all $\sigma \in G$, $\sigma(B) = B$ or $\sigma(B) \cap B = \emptyset$.

Definition 4.4: If B is a G-block, we call $\{\sigma(B) : \sigma \in G\}$ a *G-block system in A*.

Remark: Given a G-block system, G acts transitvely on the blocks of the system.

Definition 4.5: If there are no G-blocks in a G-block system in A of size > 1, we say G acts *primitvely* on A

Definition 4.6: A G-block system is *minimal* if G acts primitvely on the blocks.

Remark: It is the number of blocks that is minimal.

Proposition 4.1: If for a finite set A, $G \subseteq \mathrm{Sym}(A)$ acts transitively on A and B is a G-block on A containing the element a in A, then G_B defined by $G_B = \{\sigma \in G : \sigma(B) = B\}$

is a subgroup of G containing the stabilizer of a in G, $G_a = \{\sigma \in G : \sigma(a) = a\}$.[*Dummit and Foote*, 2004, 117]

Proof. Let $\sigma, \tau \in G_B$. Observe,

$$\begin{aligned}(\sigma\tau^{-1})(B) &= \sigma(\tau^{-1}(B)) = \sigma(\tau^{-1}(\tau(B))) \\ &= \sigma((\tau^{-1}\tau)(B)) = \sigma(B) = B.\end{aligned}$$

Thus, $G_B \leq G$. Let $\gamma \in G_a$. Since B is a G-block on A, $\gamma(B) = B$ or $\gamma(B) \cap B = \emptyset$. Since $\gamma(a) = a$ and $a \in B$, we know $\gamma(B) = B$. Hence, $G_a \leq G_B$. $\square$

Proposition 4.2: For any finite set A where $G \subseteq \text{Sym}(A)$ acts transitively on A, G is primitive on A if and only if for all $a \in A$ G_a is a maximal subgroup of G [*Dummit and Foote*, 2004, 117].

Proof. $\Rightarrow$

Let $a \in A$. Suppose $G_a \subseteq H \subseteq G$. Consider $B = \{ha : h \in H\}$. We show that B is a G-block on A. Let $\sigma \in G$. Note $\sigma(B) = \{\sigma(ha) : h \in H\}$ and suppose there exist $b \in \sigma(B) \cap B$. Then there exist $h_1, h_2 \in H$ such that $h_1a = b$ and $\sigma(h_2a) = b$. Hence, $a = h_1^{-1}(\sigma(h_2a))$, but then $h_1^{-1}\sigma h_2 \in G_a \subseteq H$. Observe, $\sigma = h_1h_1^{-1}\sigma h_2h_2^{-1} \in H$. Therefore, $\sigma(B) = B$, which means B is a G-block. Since G acts primitively on A, either $B = \{a\}$ or $B = A$. If $B = \{a\}$, then $H \subseteq G_a$, so $H = G_a$. On the other hand, suppose $B = A$. Let $g \in G$. Since $B = A$, there exist $h \in H$ such that $ga = ha$. We have $h^{-1}ga = a$, which means $h^{-1}g \in G_a \leq H$. Therefore, $g = hh^{-1}g \in H$, as desired. Hence, $G = H$.

$\Leftarrow$

Let $a \in A$. Suppose B is a G-block in A containing a. By Proposition 4.1, $G_B = \{\sigma \in G : \sigma(B) = B\}$ is a subgroup containing G_a. Since G_a is a maximal subgroup, either $G_B = G_a$ or $G_B = G$. Suppose $G_B = G_a$. $\square$

Lemma 4.1: Let P be a transitive p-subgroup of $\mathrm{Sym}(A)$ with $|A| > 1$ and $\mathcal{B}$ be a minimal P-block system. Then

1. $|\mathcal{B}| = p$,
2. the stabilizer of $\mathcal{B}$ in P, denoted P', has index p in P.

Luks [1982].

Proof. Since $\mathcal{B}$ is a minimal P-block system, P acts primitively on the blocks. By 4.2, for all $B \in \mathcal{B}$, P_B is a maximal subgroup of P. The action of P on A induces an action on $\mathcal{B}$. Then there exist homomorphism $\phi : P \to \mathrm{Sym}(\mathcal{B})$, where $\ker \phi = P'$.By the First Isomorphism Theorem,

$$P/P' \cong \mathrm{Im}(\phi) \leq \mathrm{Sym}(\mathcal{B}).$$

Note any permutation group is primitive if and only if it is transtivie and its point stabililzer is a maximal subgroup. Moreover, all maximal subgroups of a finite p-group have index p and are normal. Thus, any primitive p-subgroup of $\mathrm{Sym}(\mathcal{B})$ has order p and acts regularly on $\mathcal{B}$. Hence, $|\mathcal{B}| = |P/P'| =$ the number of blocks $= p$.(Professor Derek Holt on Stack Exchange https://math.stackexchange.com/questions/2912055/question-about-lemma-on-primitive-groups) $\square$

5 Appendix

5.1 Running time

Theorem 5.1: Brute Force Graph Identification has runtime $\mathcal{O}((n+2)!)$.

Proof. We write out the steps and the frequency of each step. Observe,

	Steps	Frequency	Total
Initialize	$(n+1)! + n^2$	1	$n^2 + (n+1)!$
Iterate	$\binom{n}{2}$	$n!$	$\binom{n}{2}(n!)$
Output	1	1	1

Summing the entries of the Total column, we have

$$f(n) = n^2 + (n+1)! + \binom{n}{2}(n!) + 1.$$

Let $n_0 = 10$ and suppose $n > n_0$. Observe,

$$\begin{aligned} f(n) &= (n+1)! + \binom{n}{2}(n!) + 1 = (n+1)! + \frac{n!n!}{(n-2)!2} + 1 = (n+1)! + \frac{n!(n)(n-1)}{2} + 1 \\ &\leq (n+1)! + n!(n)(n-1) \leq (n+1)! + (n+2)! \leq 2((n+1)!). \end{aligned}$$

Hence, $f(n) = \mathcal{O}((n+2)!)$ where $c = 2$. □

5.2 Babai's Algorithm

The following is an short overview of *Babai* [2016, 2018]. Babai's goal is to use a "Divide-and-Conquer" recursive method, pioneered in *Luks* [1982].

1. Reduce GI to the more general String Isomorphism Problem (SI) (see Section III)

2. Solve SI in quasipolynomial time via a combination of new results as well as the algo-

rithm from *Luks* [1982] and Weisfeiler-Lehman.

References

Babai, L. (2016), Graph isomorphism in quasipolynomial time [extended abstract], in *Proceedings of the Forty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '16, p. 684–697, Association for Computing Machinery, New York, NY, USA, doi: 10.1145/2897518.2897542.

Babai, L. (2018), Group, graphs, algorithms: the graph isomorphism problem, in *Proceedings of the International Congress of Mathematicians: Rio de Janeiro 2018*, pp. 3319–3336, World Scientific.

Babai, L., and E. M. Luks (1983), Canonical labeling of graphs, in *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pp. 171–183.

Babai, L., W. M. Kantor, and E. M. Luks (1983), Computational complexity and the classification of finite simple groups, in *24th Annual Symposium on Foundations of Computer Science (Sfcs 1983)*, pp. 162–171, IEEE.

Dummit, D. S., and R. M. Foote (2004), *Abstract algebra*, vol. 3, Wiley Hoboken.

Heap, B. (1963), Permutations by interchanges, *The Computer Journal*, *6*(3), 293–298.

Hopcroft, J. E., and J.-K. Wong (1974), Linear time algorithm for isomorphism of planar graphs (preliminary report), in *Proceedings of the sixth annual ACM symposium on Theory of computing*, pp. 172–184.

Johnson, S. M. (1963), Generation of permutations by adjacent transposition, *Mathematics of computation*, *17*(83), 282–285.

Luks, E. M. (1982), Isomorphism of graphs of bounded valence can be tested in polynomial time, *Journal of computer and system sciences*, *25*(1), 42–65.

McKay, B. D., et al. (1981), Practical graph isomorphism.

Mehta, D. P., and S. Sahni (2004), *Handbook of data structures and applications*, Chapman and Hall/CRC.

Shervashidze, N., P. Schweitzer, E. J. Van Leeuwen, K. Mehlhorn, and K. M. Borgwardt (2011), Weisfeiler-lehman graph kernels., *Journal of Machine Learning Research*, *12*(9).

Weisfeiler, B. (1976), On construction and identification of graphs, springer lecture notes, 558.

Weisfeiler, B., and A. Leman (1968), The reduction of a graph to canonical form and the algebra which appears therein, *nti, Series*, *2*(9), 12–16.

Zemlyachenko, V., N. Korneenko, and R. Tyshkevich (1982), The isomorphism problem for graphs, *Zap. Nauch. Seminarov LOMI AN SSSR*, *118*, 83–158.