

# CANONICAL FORMS, OVER-COLORING, AND POLYTIME GRAPH ISOMORPHISM

ANDREW MEYER

California State University of Sacramento  
agm223@csus.edu

**Abstract.** The question of Graph Isomorphism’s (GI) true time complexity classification has remained a mystery for years. It is certainly in NP but uncertain whether or not it is in P, with many suggesting it is in fact NP-Intermediate. Despite many special case polytime solvers existing, a recent breakthrough in the field has only improved the problem’s time complexity to “quasi-polytime”  $O(n^{\log^c n})$ . This thesis proposes an “over-coloring” approach to canonical label construction that is  $O(n^4)$  and conjectures based on reasoning and empirical evidence that it is a bijective (1:1) map from graph symmetric groups to canonical labels. That is to say, the proposed function  $\text{overColor}(G)$  takes as input a graph  $G$  and outputs a canonical label for  $G$ : a label that is unique to only  $G$  and the set of all graphs isomorphic to  $G$ . Then, three comparison algorithms that can each be used to distinguish the labels in polytime, polytime, and factorial time respectively are provided. The first algorithm used for comparison is a valid polytime GI solver if and only if the function  $\text{overColor}(G)$  is invertible (1:1). The second allows for a naïve reconstruction of the permutation between input graphs. The third offers a true reconstruction of this permutation matrix that leverages canonical labels to prune the permutation search space. A proof for the reduction of polytime GI to the invertibility of  $\text{overColor}(G)$  is offered and future work is discussed for how this proof can be used to aid in solving polytime GI itself. Lastly, supporting experimental evidence for both random graphs and specific  $k$ -regular and Johnson graphs is offered to demonstrate  $\text{overColor}(G)$ ’s robustness in these spaces that suggests it may be 1:1.

**1. Introduction.** Graph isomorphism (GI), simply stated, is the test of whether two graphs are essentially the same graph. More formally, given the adjacency matrices of two graphs,  $G_1 = (V_1, E_1)$  and  $G_2 = (V_2, E_2)$ , does there exist a permutation matrix  $\phi$  such that  $(\phi)(G_1)(\phi^T) = G_2$ ? From a graph theory perspective this asks if there exist a bijection  $\phi$  between the vertex sets  $V_1$  and  $V_2$  such that for every edge  $e(v_a, v_b) \in E_1$  there exists a bijected edge  $e(\phi(v_a), \phi(v_b)) \in E_2$ ? GI is a well studied problem with many special case polytime solvers [3–5, 7, 11, 12] and for most practical purposes can be polytime solved by NAUTY, TRACES, and their derivatives [6]. However, no known universal polytime solver yet exists and despite  $GI \in NP$ , it is unknown whether or not  $GI \in P$ . Such problems are thought to fall into a possible NP-Intermediate (NPI) complexity class where they are members of  $NP - P$  but not NP-Complete [9]. Examples of other problems suspected to be in NPI include integer factorization as evidence by the best known algorithms remaining exponential [10, 14, 15], the closely related discrete log problem [15], and minimum circuit size [8].

Some examples of special case polytime GI solvers include instances where input graphs have bounded tree width [4], bounded genus [12], bounded degree [3], or are permutation graphs [5]. There are also instances of linear time solvers such as in the case of graphs that are planer embeddable [7]. However, the fastest known general case solver for GI runs in  $e^{O(\sqrt{n \log n})}$  time which is certainly exponential [2]. Also of note is a recently proposed solution to general GI that falls into the quasi-polytime class at  $O(n^{\log^c n})$  that drew a lot of attention in 2016 but was initially contested on its claimed time complexity [1]. Another interesting approach worth mentioning is invariant testing to solve GI that simply checks through a list of relatively easy to compute graph invariants. If enough invariants are checked with no unequal values between two graphs, they can be considered relatively isomorphic. However, this

is often used as a pre-process to more robust GI methods rather than the primary method.

This paper proposes a highly generalized  $O(n^4)$  polytime solver for the GI problem that covers all finite connected graphs; both weighted and unweighted, directed and undirected, with or without loops, and with or without multi-edges. The algorithm works by first constructing set canonical labels (SCL) for each rooted graph embedded in the input graphs using the `overColor()` function described in **8. Algorithm and Discussion**. Then comparison algorithms perform set comparisons on these SCLs, attempt a naive reconstruction of the permutation matrix  $\phi$ , or attempt a heuristic pruning of the permutation search space before using a backtracking approach to find  $\phi$ . This paper will discuss the pseudo code for these algorithms, the theory behind them, and experimental results obtained by testing these algorithms on semi-randomly generated graph pairs. The implemented PHP code, test suite, and results repository can be found in the appendices as well.

**2. Problem Statement.** Does there exist a polytime algorithm that can compute canonical forms of simple connected graphs? If so, can this algorithm be expanded to include graph loops, edge weights, directed edges, and multi-edges? Lastly, can the SCL canonical forms of two graphs be compared in polytime?

**3. Background.** A graph  $G$  is made up of a vertex set  $V$  and an edge set  $E$  and a graph's creation is often denoted as  $G = (V, E)$ . A common representation of a graph is a two dimensional,  $|V| \times |V|$  matrix in which each cell represents an edge/non-edge between vertices  $(v_a, v_b) \in G$  and each row and column represents a vertex  $v \in G$ . This representation is referred to as an adjacency matrix, which we will also denote as  $G$ . In the case of simple graphs, an adjacency matrix is symmetric and each cell contains a 0 for no edge or a 1 for edge. In the case of edge weighted graphs, the 1 is replaced by the edge weight  $w \in W$  (assumed to be non-zero) where  $W$  is the set of all weights in  $G$ . For directed graphs, the adjacency matrix becomes asymmetrical because some edge  $e(v_1, v_2)$  does not necessarily have a partnering edge  $e(v_2, v_1)$  as it would have had in the symmetric adjacency matrix of an undirected graph. A loop is a vertex that is both the origin and terminus of an edge, denoted as  $e(v_1, v_1)$ . A loop in an adjacency matrix is represented by an edge in a cell along the diagonal from the origin of the matrix. A multigraph is a graph that contains multi-edges. A multi-edge is a set of edges connecting the same two vertices and is represented in an adjacency matrix by an array in the cell corresponding to the vertices  $(v_a, v_b) \in G$ . Lastly, a sub-graph is simply a graph  $SG$  contained inside of another graph  $G$  such that  $SG \subseteq G$ .

A permutation matrix  $\phi$  is a matrix in which for any given row or column, only one of the cells contains a 1 while all others contains 0. This is the matrix representation of the bijection  $\phi$  between two isomorphic graphs. In order to permute a graph  $G_1$  in to  $G_2$  we matrix multiply such that  $G_2 = (\phi)(G_1)(\phi^T)$  where  $\phi^T$  is the transpose of  $\phi$ . The question of two graphs being isomorphic to each other is as simple as asking does there exist a matrix  $\phi$  for which this calculation can be performed? Although relatively straight forward, computing  $\phi$  is often hard to do in general.

A common approach to tackling GI is to use canonical labeling schemes to give all graphs in the same symmetric group the same unique label and to ensure no other symmetric group of graphs shares that label, or a close approximation of this through refinement techniques [6]. In this way, GI can be reduced to a checking of these canonical labels. It should be noted, although refinement techniques are considered the best algorithms at present for efficient general case GI, they are still lower bounded

by exponential time complexity [13]. A symmetric group in this case is the set of all graphs that can be created given one graph  $G$  transformed via  $G_n = (\phi)(G)(\phi^T)$  for all possible  $\phi$  of appropriate size, and a symmetric group in this case is  $|V|!$  in size. As of now, no canonical labeling scheme has been created that can construct these labels in polytime in general. It should be noted, a canonical labeling approach to GI does not necessarily involve computing  $\phi$  nor does it necessarily include  $\phi$  as an output. Instead, it assigns labels to graphs such that iff two graphs are isomorphic, they share a label.

A rooted graph is simply a graph that has a vertex designated as the root of the graph. Given an unrooted graph  $G = (V, E)$ , there are  $|V|$  rooted sub-graphs that use the entire sets  $(V, E) \in G$ ; one for each vertex in  $G$ . Graph coloring is the process of assigning colors to vertices of  $G$  such that no vertex neighbors another vertex of the same color. Minimum coloring is the method in which this is done using the least number of colors. N-coloring is the process of attempting to color  $G$  using at most  $N$  colors. Over-coloring is the process of continually 2-coloring vertices in the same way as standard 2-coloring but doing so until the entire graph is monochromatic. That is to say, once 2-coloring is broken by adjacent vertices having the same color, continue coloring anyways and color over vertices that already have color assigned to them, as can be seen figure 3.1.

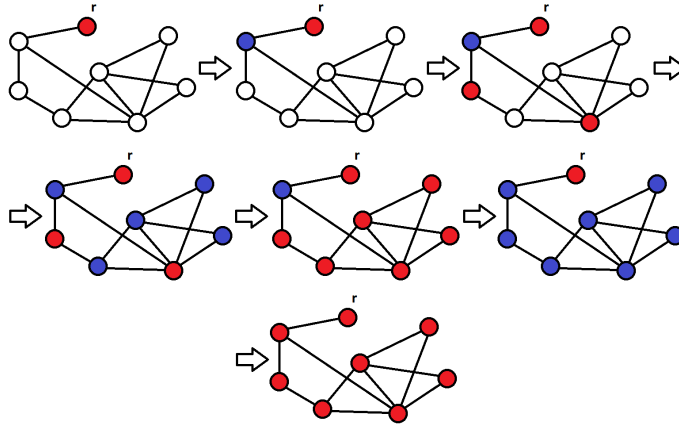


FIG. 3.1. Visualizing the over-coloring process

Note, over-coloring requires that an odd cycle exist in  $G$  for termination to occur per **Lemma 4.6**. Gamma matrices ( $\Gamma$ ) are produced via the `overColor()` process described in the section 8. **Algorithm and Discussion.** Essentially, they are codified representations of the step by step over-coloring of graphs, starting at each possible root. Note, a  $\Gamma$  matrix is equal to the return of `overColor(G)`. For simplicity sake, the variable  $\Gamma$  is used slightly differently in the `canonicalNameCompare()` function as explained in **Lemma 4.5**, but in all other cases, it is as described above. Degree or neighborhood of a vertex  $v$  is the number of adjacent vertices to  $v$ . Lastly, color influence (CI) is a time sensitive analogous to neighborhood of vertices in the over-color process. A vertex's CI during any given round of `overColor()`s while loop (line 8) is the number of its neighbors who are currently colored the same as the current rounds color variable. This is simply a measure of the number of neighbors that will color  $v$  during a particular round.

A regular graph is a graph whose vertices all have the same number of neighbors.

A  $k$ -regular graph is a graph for which all vertices have exactly  $k$  neighbors. A graph can be said to contain a symmetry if any two edges  $(e_1, e_2) \in G$  can be swapped and the resulting graph remains isomorphic to  $G$ . Three important function types are surjective, injective, and bijective. Surjective functions map all elements of the function's range to at least one element in its domain. An injective function maps all elements of a domain to exactly one element in its range. A bijective function is one that is both injective and surjective; mapping all elements of the function's domain to exactly one element in its range such that all elements in its range are accounted for. Also, bijective functions are by definition invertible which means given a function  $f$ , there exists a function  $f^{-1}$  such that  $f^{-1}(f(x)) = x$ .

A graph is said to be amenable if the surjective process of color refinement successfully canonically labels a graph. Color refinement and the more generalized individualization refinement are common techniques used in industry for GI. [6, 13, 16]. An example set for which no graph is amenable by color refinement is the set of  $k$ -regular graphs [16]. That is to say, given any two non-isomorphic,  $k$ -regular graphs with an equal number of vertices, color refinement fails to successfully identify them as non-isomorphic. Another problem area for existing GI solvers that rely on individualization refinement is for graphs that contain Johnson sub-graphs as a large proportion of the total graph [1]. A Johnson graph is constructed by first listing the set of all combinations of elements  $\{1, 2, \dots, n\}$  for an  $n$  choose  $k$ , denoted  $\binom{n}{k}$ . This list is the vertex set  $V \in G$ . Then, for each pair of vertices  $(v_1, v_2) \in V$ , iff the size of their intersection of elements  $i = k - 1$ , the elements share an edge  $e(v_1, v_2) \in E$ . Specifically for small  $k$  values, and especially for  $k = 2$ , these graphs, when embedded in slightly larger graphs tend to cause individualization refinement algorithms to fail or exceed their average time complexity of polytime [1]. Often, mountains of heuristics are used to combat these issues while using color refinement and more generalized individualization refinement, but in general these two graph classes seem to cause issues with these most broadly applicable base algorithms.

Another common technique worth mentioning but not important to this paper is the checking of invariant lists of input graphs. A graph invariant is a property that is unchanged regardless of how a graph is permuted. This means they are excellent for quickly ruling out trivial cases of non-isomorphic pairs of graphs. These algorithms are probabilistic in the sense that they check the input graphs for unequal invariants and if none are found, the algorithm says they are relatively isomorphic. To this date, no finite list of invariants have been found that can be computed in polytime and guarantee isomorphism if all are equal. Although this technique varies wildly in terms of accuracy and speed, it provides a framework for easily exchanging accuracy for speed and vice versa depending on the application.

**4. Lemmas.** This section will discuss the necessary prerequisites for section 5. **Conjectures** and section 6. **Proof.**

Lemma 4.1. If a connected simple graph  $G$  of at least 5 vertices is bipartite, its complement  $G^C$  will include an odd cycle and be connected so long as the original graph was not complete bipartite.

*Proof.* Suppose  $G$  is a connected simple bipartite graph. If one of its bipartitions contains at least three vertices  $(v_1, v_2, v_3)$ , the vertices  $(v_1, v_2, v_3)$  form a triangle in  $G^C$ . If  $G$  is not complete bipartite, there exists at least two vertices, one in each bipartition, that are not adjacent. This means they are connected in  $G^C$  which implies  $G^C$  must be connected.

Lemma 4.2. If two simple or loopy graphs  $G_1$  and  $G_2$  are isomorphic, their

complements  $G_1^C$  and  $G_2^C$  are also isomorphic.

*Proof.* Given two simple or loopy isomorphic graphs  $G_1$  and  $G_2$  and their bijection  $\phi_a$ , it follows that  $G_2 = (\phi_a)(G_1)(\phi_a^T)$  by the definition of isomorphism. Let us define graph complement as the function  $f(G) = G'$  where each 0 in the adjacency matrix of  $G$  is set to 1, and each 1 originally in  $G$  is set to zero (assume no weights).  $f$  is obviously invertible because  $f(f(G)) = G$ . This implies it is bijective and that the bijection  $\phi_b$  such that  $(\phi_b)(G_1^C)(\phi_b^T) = G_2^C$  is equal to  $\phi_a$ . Because a bijection  $\phi_b$  exists,  $G_1^C$  and  $G_2^C$  must be isomorphic.

**Lemma 4.3.** Given two simple complete bipartite graphs  $G_1$  and  $G_2$  with an equal number of vertices and  $|V_1| > 4$ , if the size of the two bipartitions  $(b_1, b_2) \in G_1 = |b_1|, |b_2|$  and the sizes of bipartitions  $(b_3, b_4) \in G_2 = |b_3|, |b_4|$ , then iff  $|b_1|$  and  $|b_2| \neq |b_3|$  nor  $|b_4|$ ,  $G_1$  and  $G_2$  cannot be isomorphic.

*Proof.* Given a simple complete bipartite graph  $G_m$ , the number of edges can be calculated by  $|E_m| = |b_x| * |b_y|$ . If  $|b_1|$  and  $|b_2| \neq |b_3|$  nor  $|b_4|$ , then  $|b_1| * |b_2| \neq |b_3| * |b_4|$  if  $|b_1| + |b_2| = |b_3| + |b_4|$  and  $|b_1| + |b_2| = |V_1| > 4$ . Thus,  $(G_1, G_2)$  are isomorphic iff  $|b_1| * |b_2| = |b_3| * |b_4|$  and  $|b_1| + |b_2| = |b_3| + |b_4|$  for  $|V_1| > 4$ . This means because we know  $|V_1| = |V_2|$ , we know  $G_1$  and  $G_2$  are isomorphic if the graphs are simple complete bipartite.

Note: If the isomorphism of the simple complete bipartite graphs is established and only the edge weights can break  $G_1$  and  $G_2$ 's isomorphism, uniformly inserting edges for all non-edges in  $G_1$  and  $G_2$  that all share a unique edge weight outside of the domain of edge weights originally in  $G_1$  and  $G_2$  preserves any bijective properties  $G_1$  and  $G_2$  may have. This is demonstrated by lines 18 and 19 in `GI()` and results in the succesful classifications described in section 7. **Experimental results.**

**Lemma 4.4.** The bijection  $\phi_a$  between isomorphic graphs  $G_1$  and  $G_2$  is the same as the bijection  $\phi_b$  between  $\Gamma_1$  and  $\Gamma_2$  where  $\Gamma_a = \text{overColor}(G_a)$ . For a better understanding of `overColor()`, see **Lemmas 4.6, 4.7, Conjecture 5.1** and **Algorithm 8.3**.

*Proof.* The string contained in each cell  $c(a, b) \in \Gamma_1$  corresponds to how vertex  $v_b$  was colored by `overColor( $G_1$ )` when vertex  $v_a$  was selected as the root. The string contained in each cell  $c(\phi(a), \phi(b)) \in \Gamma_2$  corresponds to how vertex  $\phi(v_b)$  was colored by `overColor( $G_2$ )` when vertex  $\phi(v_a)$  was selected as the root. Because  $v_a$  and  $v_b$  are isomorphically equivalent to  $\phi(v_a)$  and  $\phi(v_b)$ , the edge  $e(v_a, v_b)$  is equivalent to  $e(\phi(v_a), \phi(v_b))$ . This means the edge cell  $c(a, b) \in G_1$  being equivalent to the edge cell  $c(\phi(a), \phi(b)) \in G_2$  implies the cell  $c(a, b) \in \Gamma_1$  is equivalent to the cell  $c(\phi(a), \phi(b)) \in \Gamma_2$  if `overColor()` is deterministic. Since `overColor()` is obviously deterministic due to lack of any randomization methods, this holds true.

Evidence to support this can be seen in lines 7 and 8 of `canonicalNameCompare()` and the resulting succesful classification rate shown in section 7. **Experimental results.** If  $\phi_a \neq \phi_b$ , this action would cause the arrays A and B in `cannonicalNameCompare()` to contain different elements even when  $G_1$  is isomorphic to  $G_2$  and  $\phi_a \neq$  the identity matrix. This implies  $\phi_a = \phi_b$  when  $G_1$  and  $G_2$  are isomorphic.

**Lemma 4.5.** Given a gamma matrix  $\Gamma_1 = \text{overColor}(G_1)$ , concatenating  $G_1$ 's adjacency matrix via  $G_1 \oplus \Gamma_1$  preserves the canonical property of  $\Gamma_1$  while including the required edge weight information needed to determine isomorphism of edge weighted graphs.

*Proof.* Following from lemma 4.4, the bijection  $\phi_a$  between  $G_1$  and  $G_2$  is the same as the bijection  $\phi_b$  between  $\Gamma_1$  and  $\Gamma_2$ . This implies  $(\phi_a)((G_1) \oplus (\Gamma_1))(\phi_a^T) = (\phi_b)((G_1) \oplus (\Gamma_1))(\phi_b^T)$ .

Note: Matrix concatenation in this case is the concatenation of each cell  $M_1(a, b)$

of one matrix with the corresponding cell  $M_2(a, b)$  of another.

Lemma 4.6. The function  $\text{overColor}(G)$  will always terminate when given a connected, finite graph  $G$  that contains an odd cycle.

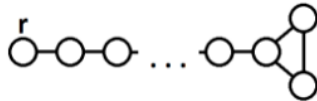
*Proof.* If a graph  $G$  has an odd cycle, there must exist at least two adjacent vertices  $(v_1, v_2)$  equidistant from a third vertex  $v_3$ . This implies regardless of the root  $v_3 \in G$  selected,  $\text{overColor}(G)$  will eventually cause  $G$  to exist in a state where two neighboring vertices  $(v_1, v_2) \in G$  have the same color given  $G$  contains an odd cycle. This monochromatic group of vertices is  $\mu$ . On every cycle of  $\text{overColor}()$ 's while loop after  $\mu$  is formed, all members of  $\mu$  and all neighbors of vertices in  $\mu$  will form the monochromatic group  $\mu$ . Because  $G$  is connected finite, all vertices  $v \in G$  must eventually join  $\mu$  which eventually results in  $|\mu| = |V|$  which implies  $G$  is monochromatic. Then  $\text{overColor}(G)$  cycles to the next starting root. Because there are a finite number of roots to select from,  $\text{overColor}(G)$  must eventually terminate.

Note: In the case of graphs with loops,  $\mu$  initial may only contain one vertex that neighbors itself.

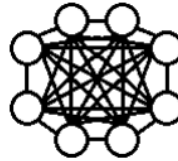
Lemma 4.7. The function  $\text{overColor}(G)$  will always terminate in  $O(n^4)$  when given a connected finite graph  $G$  that contains an odd cycle and  $n = |V|$ .

*Proof.* The loose upper bound  $O(n^4)$  can be obtained by combining the two worst case scenarios for input graphs.

1. The first worst case is the greatest number of cycles  $\text{overColor}(G)$ 's while loop will execute before  $\mu$  is formed. A depiction of this worst case input graph can be seen in graph 1 below. If  $n$  is the number of vertices in  $G$ ,  $\text{overColor}(G)$ 's while loop will have  $n - 1$  cycles before  $\mu$  is formed by the two vertices furthest distance from  $r$ . Which implies  $2n$  until termination, which is  $O(n)$ .
2. The second worst case is when in a given cycle, every vertex in  $G$  must color its neighbors, and each of these vertices neighbors  $n - 1$  vertices. In this scenario,  $O(n^2)$  colorings would occur in this cycle. This is the case of a complete graph, as can be seen in graph 2 below.



Graph 1



Graph 2

FIG. 4.1. Worst case graphs

Thus, if  $O(n)$  cycles are required to achieve a monochromatic graph, and on each cycle,  $O(n^2)$  coloring operations occur, then  $O(n^3)$  operations will occur after all cycles are run. It should be obvious  $G$  could not be both of these cases and thus  $\text{overColor}(G)$ 's while loop could not take more than the number of operations required to terminate in this hypothetical. Now, if the above scenario is expanded such that it is run for every possible root  $r \in G$ ,  $n$  cycles of  $O(n^3)$  operations will occur. Hence,  $\text{overColor}(G)$  can be naively upper bounded by  $O(n^4)$ .

Lemma 4.8. In  $\text{GI}()$ , any directed graph  $G$  is equivalent to a directed graph  $G$  in which every uni-directional edge in  $G$  is complemented with a new edge in  $G$ ,

connected to the same vertices but pointed in the opposite direction whose weight is outside of the domain of weights for edges originally in  $G$ .

*Proof.* Given a graph  $G_1$  whose edge weights form the set  $W$ , let us define the function  $f$  that takes every cell  $c(a, b) \in G_1$  that contains a non-zero value and puts in the cell  $c(b, a) \in G_1$  the value  $x \notin W$  if  $c(b, a)$  does not already contain a non-zero value. Let us then define  $f^{-1}$  as the function that looks at  $G_1$  and for every cell  $c(a, b)$  that contains the value  $x$ , replaces it with 0. Because  $f$  is invertible per  $G_1 = f^{-1}(f(G_1))$  it is bijective and  $\phi_a = \phi_b$  for  $(\phi_a)(G_1)(\phi_a^T) = G_2$  and  $(\phi_b)(f(G_1))(\phi_b^T) = f(G_2)$ . Thus, for any directed graph  $G$ , transforming  $G$  such that  $G = f(G)$  as a pre-process for  $\text{GI}()$  maintains any isomorphism or lack thereof for inputs of  $\text{GI}()$ .

Lemma 4.9. In  $\text{GI}()$ , multigraphs are equivalent to edge weighted graphs with no multi-edges.

*Proof.* Given a graph  $G_1$  whose edge weights form the set  $W$ , let us define the function  $f$  that takes every cell  $c(a, b) \in G_1$  that contains an array of multi-edges and replaces the array in  $c$  with a string that equals the weight-sorted ordering of the multi-edges in  $c$ , separating each weight in the string by  $x \notin W$ . Let us then define  $f^{-1}$  as the function that looks at  $G_1$  and for every cell  $c(a, b) \in G_1$  that contains a string with a character  $x$ , replaces it with an array of edge weights equal to that string exploded by the character  $x$ . Because  $f$  is invertible per  $G_1 = f^{-1}(f(G_1))$  it is bijective and  $\phi_a = \phi_b$  for  $(\phi_a)(G_1)(\phi_a^T) = G_2$  and  $(\phi_b)(f(G_1))(\phi_b^T) = f(G_2)$ . Thus, for any directed graph  $G$ , transforming  $G$  such that  $G = f(G)$  as a pre-process for  $\text{GI}()$  maintains any isomorphism or lack thereof for inputs of  $\text{GI}()$ .

Lemma 4.10. The SCLs, or  $\Gamma$  matrices of two graphs can be compared in polytime.

*Proof.* Logically, a column  $v$  in a  $\Gamma$  matrix represents a particular vertex  $v$  selected as a root for a round of  $\text{overColor}()$  and each element  $u$  in that column is how the vertex  $u$  was colored throughout  $\text{overColor}()$  in the round in which  $v$  was the root. If **Conjecture 5.1** holds, the set of strings contained in these cells are unique to only graphs in the same symmetric group which means sorting the column does not destroy the uniqueness of the information. This means if each column of  $\Gamma$  is sorted, the uniqueness of the set of elements of  $\Gamma$  is maintained. Thus, comparing two  $\Gamma$  matrices can be done via an  $O(n^2 \log n)$  sort where each of the  $n$  columns is sorted individually. Then each column concatenates its elements into strings where each element is separated by some character  $y \notin (W + x)$ . These strings are then placed in arrays A and B for  $\Gamma_1$  and  $\Gamma_2$  respectively. Lastly, sort A and B in  $O(n \log n)$ . Afterwards, an  $O(n)$  loop string compares the elements of A and B for equality.

Lemma 4.11. If  $G$  is monochromatic, all edges have necessarily been traversed during  $\text{overColor}(G)$  except for edges connected to the vertices colored in the final round prior to  $G$  becoming monochromatic. If one additional round of over-coloring is performed, all edges must have been traversed at least once.

*Proof.* Let us say  $\text{overColor}()$ s while loop round number is  $d$  and the vertex furthest from root  $r$  is distance  $x$  from  $r$ . When  $d = 1$ , all edges connected to  $r$  are traversed. When  $d = 2$ , all edges of vertices distance 1 from  $r$  are traversed. In general, when  $d = n$ , all edges of vertices distance  $n - 1$  from  $r$  are traversed. When  $d = x$ , all edges of vertices  $x - 1$  are traversed. This means on round  $x + 1$ , all edges of vertices distance  $x$  from  $r$  are traversed.

Lemma 4.12. Leaving the arrays A and B unsorted in the last step of the process described in **Lemma 4.10** allows for the recreation of  $\phi$  assuming  $G_1$  and  $G_2$  are isomorphic.

*Proof.* String 1 in A corresponding to  $\Gamma_1$  matching string x in B corresponding to  $\Gamma_2$  means the original permutation matrix's first row contained a 1 in position x

(starting from 1). Repeating this until all strings are matched reconstructs  $\phi$ . If  $G_1$  is not isomorphic with  $G_2$ , A and B should not contain the same set of elements if **Conjecture 5.1** holds. However, if  $G_1$  is isomorphic to  $G_2$ , A and B are guaranteed to contain the same set of elements due to **Lemma 4.4**'s proof of bijection preservation.

Note: If a graph  $G$  has a symmetry to it such that two of its own vertices can be swapped and the resulting graph  $G'$  is isomorphic with  $G$ , these vertices will produce the same string in  $G$ 's  $\Gamma$ . This means even if two graphs are isomorphic, rebuilding  $\phi$  as described above may result in an invalid permutation. To correct this, an algorithm would need to check every possible bijection of  $\Gamma_1$  and  $\Gamma_2$  to find the original  $\phi$  that permutes  $G_1$  into  $G_2$ . The naive approach as described in the above proof can be seen in **Algorithm 10.4**. A backtracking algorithm that heuristically prunes the original permutation search space using  $\Gamma$  and then checks all remaining possible bijections can be seen in **Algorithm 10.5, 10.6**.

Lemma 4.13. The size of the pruned  $\phi$  space **Algorithm 4.6** searches through has a factorial growth rate that is less than or equal to  $n!$  where  $n = |G|$ .

*Proof.* Given a graph  $G$  with  $m$  vertices that satisfy the condition described in **Lemma 4.12**'s note, there are  $m!$  ways in which those vertices could be interchanged and thus  $m!$  potential  $\phi$ 's to check. If  $G$  contains a set  $M$  of different vertex sets that all satisfy the same condition, the size of the  $\phi$  search space  $|\Phi| = \prod(M_x!)$  which implies  $|\Phi| \leq n!$  where  $|\Phi| \rightarrow n!$  as any  $|m| \in M \rightarrow n$ .

Note: The larger  $|\Phi|$  becomes, the more  $G$  can be said to be symmetric, which can potentially be exploited for further heuristic reductions of the search space (see Appendix something - Authors Notes).

## 5. Conjecture. Conjecture 5.1. overColor() is a bijective function.

Firstly, the empirical evidence reported in section 7. **Experimental Results** demonstrates overColor() is at least "good" at uniquely assigning SCLs to graphs according to their symmetric groups. If overColor() is in fact a proper canonical labeling function, it would mean overColor() bijectively maps symmetric groups of simple and looped graphs into the SCL space of its outputs.

Reasoning to support this claim is as follows: Iff  $G_a$  is not isomorphic to  $G_b$ , there must exist an edge  $e_1(v_a, v_b) \in G_a$  that does not have a corresponding edge  $e_2(\phi(v_a), \phi(v_b)) \in G_b$  by definition of isomorphism. What overColor( $G$ ) is doing is traversing all edges in a graph  $G$  and recording for every vertex when it is pathed to by the over-coloring process starting at each root  $r$ . Like in the process described in **Lemma 4.11**, every edge must be traversed at least once for overColor()'s while loop to exit. This means the existence of  $e_1 \in G_1$  causes at least one path the over-coloring process takes during overColor( $G_1$ ) to be different than any path taken by overcolor( $G_2$ ) for  $G_2$ .

Now, if our permutation matrix is the identity matrix, let us salt  $G_2$  such that  $G_1$  is not isomorphic to  $G_2$ . The way in which we salt is to transplant an edge from the vertex pair  $(v_a, v_b)$  to the vertex pair  $(v_c, v_d)$  such that the pairs are distinct from each other and do not result in a graph symmetry with each other. This means  $e_1(v_a, v_b) \in G_1$  is no longer equivalent to  $e_2(v_a, v_b) \in G_2$ . As an aside, if  $G_1$  and  $G_2$  were still isomorphic after salting,  $e_1(v_a, v_b)$  would still have a bijected edge  $e_2(v_c, v_d)$ . Because the over-coloring process must traverse  $e_1$  at some point during overColor( $G_1$ ) per **Lemma 4.11**, one of the paths taken to  $v_a$  and/or  $v_b$  must be different from any path taken by the over-coloring process in overColor( $G_2$ ) for either vertex of  $e_2$ . This means the CI of at least one of the vertices of  $e_1 \in G_1$  must be different at some point during overColor() from either of the vertices in  $e_2$ . This means the string contained



at either  $\Gamma_1[r][a]$  and/or  $\Gamma_1[r][b]$  must be different than the string contained in either  $\Gamma_2[r][a]$  and/or  $\Gamma_2[r][b]$ . This implies the set of strings contained in column  $r$  of  $\Gamma_1$  must necessarily be different than the set of strings contained in any column of  $\Gamma_2$ , so long as the salting process broke  $G_2$  from  $G_1$ 's symmetric group. Therefore,  $\text{overColor}()$  is injective because only in the case of graphs being isomorphic can  $e_1(v_a, v_b)$  have an equivalent  $e_2(\phi(v_a), \phi(v_b))$ . Finally, because  $\text{overColor}()$  is injective, and surjective **Lemma 4.6**,  $\text{overColor}()$  is bijective for simple and loopy graphs.

If  $\text{overColor}()$  can formally be shown to be invertible, it is automatically shown to be bijective. Conversely, showing it is not invertible proves  $\text{overColor}()$  is not a proper canonical labeling scheme and is only “good” at distinguishing symmetric groups in its labeling process.

**6. Proof.** Given two simple connected graphs with or without loops  $G_1$  and  $G_2$ , either both graphs are bipartite, neither are bipartite, or one is bipartite and one is not. Given the first case, the two graphs must be inverted for  $\text{overColor}()$  to terminate per **Lemma 4.6**. We know because of **Lemma 4.1** this complement of a graph guarantees an odd cycle exists in  $\text{overColor}()$ 's input graph and that this graph complement guarantees a connected graph except when the original graph was complete bipartite. This is a valid operation in an isomorphism test because of **Lemma 4.2**. We know  $\text{overColor}()$  will produce some label for  $G_1$  and  $G_2$  in  $O(n^4)$  time because of **Lemma 4.6** and **Lemma 4.7**. If  $\text{overColor}()$  is shown to be invertible, we also know the labeling produced will be canonical due to **Conjecture 5.1**. These two labels can then be compared in polytime to determine isomorphism of  $G_1$  and  $G_2$  because **Lemma 4.10**. For the second case, the graphs need not be inverted because an odd cycle already exists in both and are simply plugged into  $\text{compareCanonicalNames}(G_1, G_2)$  from  $\text{GI}()$ . The third case is trivial because one graph contains an odd cycle and the other does not. This implies they cannot be isomorphic. Lastly, in the event the original graphs were complete bipartite, have the same edge counts, and have the same vertex counts, they must be isomorphic per **Lemma 4.3**.

Assuming **Conjecture 5.1**: Given a simple graph with loops, the graph is obviously not bipartite and can therefore be covered under  $\text{GI}()$  because of **Lemma 4.6**. Given a weighted graph, concatenating the adjacency matrix and its gamma matrix provides a valid canonical labelling because of **Lemma 4.4** and **Lemma 4.5**. For directed graphs, bijectively transforming them to their equivalent weighted forms via the process described in **Lemma 4.8** provides valid canonization through the use of  $\text{GI}()$  as well, again because of **Lemma 4.4** and **Lemma 4.5**. Multigraphs can be bijectively mapped to weighted non-multigraphs because **Lemma 4.9** which implies valid canonization through  $\text{GI}()$  once again because of **Lemma 4.4** and **Lemma 4.5**. Lastly, **Lemma 4.3** simply provides justification for the weighted complete bipartite graph scenario in  $\text{GI}()$  which can be seen on lines 18 and 19 of  $\text{GI}()$ . Given the above and because **Lemma 4.7**, graph isomorphism for connected simple, loopy, weighted, directed, and/or multigraphs is polytime solvable if **Conjecture 5.1** can be proved.

Finally, **Lemma 4.12** provides an alternative to the conjecture dependence above. If a  $\phi$  can be found such that  $(\phi)(G_1)(\phi^T) = G_2$ ,  $G_1$  and  $G_2$  are obviously isomorphic. One way it can be found is by simply finding a bijection between sets A and B from **Lemma 4.10**. Typically, only one exists and thus a  $\phi$  is trivially constructible within polytime per **Lemma 4.12**. However, as stated in the note of **Lemma 4.12**, this is not always the case. For these instances, an algorithm must search a set of possible  $\phi$ s to find the specific  $\phi$  that satisfies  $(\phi)(G_1)(\phi^T) = G_2$ . A backtracking algorithm is presented in **Algorithms 10.5, 10.6** to do just that. However, **Lemma 4.13** suggests

its worst case runtime appears outside polytime. Thus, the comparison algorithm offered in **Algorithms 10.2** is polytime as shown in **Lemmas 4.6, 4.7** but is only conjectured to be a proper solver. The comparison algorithm offered in **Algorithms 10.4** is polytime, guarantees correctness if it returns true, but can potentially return a false negative if the circumstance described in **Lemma 4.12** occurs. Lastly, the comparison algorithm offered in **Algorithms 10.5, 10.6** guarantees total accuracy, but through **Lemma 4.13** and empirical testing has been found to not be within polytime, although it can be seen through **7. Experimental Results** that its average runtime is equivalent to that of the other two algorithms.

**7. Experimental Results.** The procedure used in testing  $\text{GI}()$  for random graph pair cases was as follows: randomly create a graph  $G_1$  following an Erdős-Rényi like approach to graph generation, maintaining a rough graph density of between 0.25 and 0.75. Then, permute the graph to create  $G_2$ . Afterwards, randomly chose on 50/50 odds to salt  $G_2$  such that it is no longer isomorphic to  $G_1$  and record which choice was made. Lastly, call  $\text{GI}(G_1, G_2)$  and compare the result to the expected outcome recorded when choosing to salt or not. Then repeat this process ad nauseam. The types of graphs generated for testing include simple, simple directed, unweighted bipartite, weighted bipartite, complete bipartite, loopy, loopy weighted, loopy weighted multi, and loopy weighted directed multigraphs. The salting patterns used include edge transplant, edge removal, and edge weight change. In the case of the edge transplant salt, transplanting would be iterated randomly anywhere from 1 to 10 times. Across roughly one million test cases for graphs varying in size from 25 to 150 vertices, zero cases were found where  $\text{GI}()$  returned an incorrect answer. Plots of the time and space complexity for these test cases can be seen below. Note, the time complexity is much better than  $O(n^4)$  in practice because of the naivety of the upper bound given in **Lemma 4.7**. From the data gathered during testing specifically, it can be estimated  $\text{GI}()$  is upper bounded by approximately  $n^{2.5}$ .

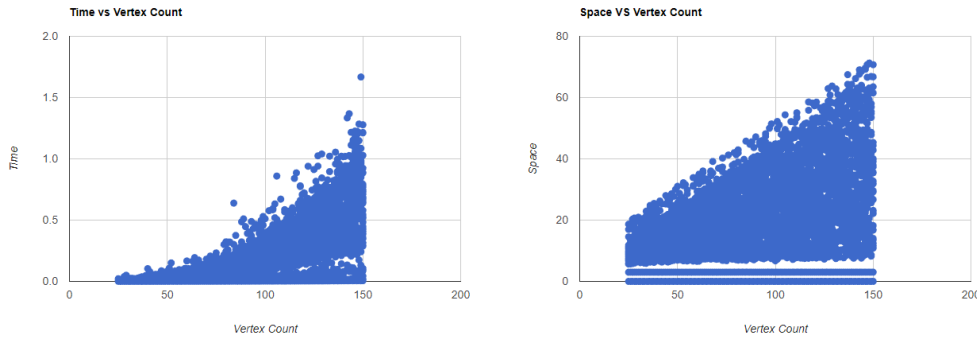


FIG. 7.1. *Time vs vertex count and Space vs vertex count*

For targeted testing, two specific test cases were looked at: Johnson sub-graphs and  $k$ -regular graphs. These two were chosen because they appear to have the highest problem incidence among leading  $\text{GI}$  solvers. The following targeted tests also only test for false positives because these are the tests for which existing solvers take issue. Also, due to the deterministic nature of  $\text{overColor}()$ , it has become apparent while testing that there is no need to actually test for true positives as the algorithm is

incapable of producing a false negative **Lemma 4.4, 4.5**. Lastly, the following two targeted examples were run using **Algorithm 10.2** specifically because it is the one **Conjecture 5.1** is concerned with.

In the case of k-regular graphs, the following procedure was used: generate two k-regular graphs  $G_1 = K(n, k), G_2 = K(n, k)$  for  $n = rand(40, 60)$  and  $k = rand(4, 5)$ . These numbers were chosen because of the computational complexity of random graph generation for k-regular graphs for large n and k values. Because  $G_1$  and  $G_2$  are random k-regular graphs, it is highly unlikely they are isomorphic, so for testing they were assumed to be non-isomorphic pairs while remaining k-regular with equal numbers of vertices and edges. Next,  $GI(G_1, G_2)$  is called and its output recorded. Any instance in which the output returned is "true," it can be said the algorithm failed to properly identify the pair as not isomorphic. It should also be noted, finding any case in which the proposed algorithm successfully identifies non-isomorphic pairs of k-regular graphs of equal vertex and edge counts demonstrates its superiority to standard color refinement because it fails in every such case. However, because this set's history of complication in GI, many random graphs within this set are tested for. [Results pending (currently at 400 graph pair)] See fig. 7.2 below for the time and space complexity charts (like 7.1).

[Insert charts here once testing is done]

In the case of Johnson graphs, the following procedure was used: generate a Johnson graph  $G = J(n, k)$  for  $n = rand(10, 20)$  and  $k = 2$ . Copy this graph to graphs  $G_1, G_2$  and then create a random binary array  $B_1$  of length  $|G| + 1$ . Append  $B_1$  to the adjacency matrix of  $G_1$  by appending each row  $r \in G_1$  with element  $e_r \in B_1$ . Then push  $B_1$  to  $G_1$  so that it is the new last row of  $G_1$ 's adjacency matrix. Next, randomly shuffle  $B_1$  to make  $B_2$  such that  $B_1 \neq B_2$  and perform the same process as described for  $G_1, B_1$  with  $G_2, B_2$ . This produces two graphs  $G_1, G_2$  that have an exceptionally small likely hood of being isomorphic while still both containing the same Johnson sub-graph that constitutes the majority of their structures. Then run a test  $GI(G_1, G_2)$  and record the result. Any instance in which the output returned is "true," it can be said the algorithm failed to properly identify the pair as not isomorphic. [Results pending (currently at 25,000 graph pairs)] See fig. 7.3 below for the time and space complexity charts (like 7.1).

[Insert charts here once testing is done]

[I'll also update the charts for the initial testing from the first paragraph of this section once I get more complete results from the new 10.4 and 10.5 canonicalName-Compare() algorithms. Currently at about 75,000 test cases that focus on simple, loopy, and bipartite graphs. This more directed focus is due to the meaninglessness of testing other cases (multi, weighted, directed, etc...) because of **Lemma 4.5, 4.8, 4.9**]

Lastly, im going to add a paragraph or two about interesting patterns i've found in the visualizations of the algorithms. If you remember from the summer, there's some pretty neat stuff to be seen in the charts of time, space, complexity and such.

**8. Algorithm and Discussion.** See **Appendix something - Implementation** for the actual implementation of these algorithms.

The function  $GI(G_1, G_2)$  is the actual GI solver called to determine if two graphs are isomorphic. Its primary function is to pre-process the input graphs for canonicalNameCompare() to determine if they are obviously not isomorphic due to differing edge or vertex counts, if one contains an odd cycle and the other does not, and to perform the processes describe in **Lemma 4.8** and **Lemma 4.9**. It then goes through some minor tree logic to determine if the input graphs should be inverted and does further pre-processing in the case of the input graphs being complete bipartite per **Lemma 4.3** to account for edge weighting. At the end of the tree logic, it calls canonicalNameCompare( $G_1, G_2, invert$ ).

---

**Algorithm 8.1**  $GI(G_1, G_2)$

---

```

1: set  $x$  and  $y \notin W_1$  or  $W_2$  such that  $x \neq y$ 
2: if  $|E_1| \neq |E_2|$  or  $|V_1| \neq |V_2|$  then
3:   return false
4: else
5:   if  $G_1$  and/or  $G_2$  has multi edges then
6:     Perform process described in Lemma 4.9, using  $x$  as separator.
7:   end if
8:   if  $G_1$  and/or  $G_2$  is directed then
9:     Perform process described in Lemma 4.8, using  $x$  for weighting.
10:  end if
11:  if isNotConnected( $G_1$ ) or isNotConnected( $G_2$ ) then
12:    return invalid input
13:  end if
14:  if isNotBipartite( $G_1$ ) and isNotBipartite( $G_2$ ) then
15:    return canonicalNameCompare( $G_1, G_2$ , false)
16:  else if isBipartite( $G_1$ ) and isBipartite( $G_2$ ) then
17:    if isCompleteBipartite( $G_1$ ) and isCompleteBipartite( $G_2$ ) then
18:      Perform process described in Lemma 4.3's note, using  $y$  for weighting
19:      return canonicalNameCompare( $G_1, G_2$ , false)
20:    else
21:      canonicalNameCompare( $G_1, G_2$ , true)
22:    end if
23:  else
24:    return false
25:  end if
26: end if

```

---

The function canonicalNameCompare( $G_1, G_2, invert$ ), called by  $GI()$ , is responsible for calling overColor() to build the SCLs for the input graphs  $G_1, G_2$  and then comparing their SCLs for equivalence. It inverts the input graphs in the event the boolean "invert" is true because **Lemma 4.1**, **Lemma 4.2**, and **Lemma 4.6**. After calling  $\Gamma_1 = \text{overColor}(G_1)$  and  $\Gamma_2 = \text{overColor}(G_2)$ , it concatenates the original input graphs to the results of overColor() because **Lemma 4.4** and **Lemma 4.5**. Lastly, because SCLs are dependent on set equivalence in which order does not matter, the  $\Gamma$  matrices are sorted so that string compare can be used to determine equivalence as described in **Lemma 4.10**.

**Algorithm 8.2** canonicalNameCompare( $G_1, G_2, invert$ )

---

```

1:  $OG_1 = G_1$ 
2:  $OG_2 = G_2$ 
3: if  $invert$  then
4:    $G_1 = \text{graphComplement}(G_1)$ 
5:    $G_2 = \text{graphComplement}(G_2)$ 
6: end if
7:  $\Gamma_1 = \text{Matrix concatenate } OG_1 \text{ and overColor}(G_1)$ 
8:  $\Gamma_2 = \text{Matrix concatenate } OG_2 \text{ and overColor}(G_2)$ 
9:  $A = B = \text{array}()$ 
10: Perform the process described in Lemma 4.10 using  $A$  and  $B$  as the arrays
11: if  $A == B$  then
12:   return true
13: else
14:   return false
15: end if

```

---

Lastly,  $\text{overColor}(G)$  is the process described in **Lemma 4.6**, **Lemma 4.7**, and **Conjecture 5.1**. It over-colors an input graph and records this step by step process in  $\Gamma$ . It then returns  $\Gamma$ . Note,  $root$  is a vertex in  $G$  and the notation “ $\text{Array}[root]$ ” refers to the column position of  $root$  in the adjacency matrix  $G$ .  $Color$  can be thought of as being either red or blue (1 or 0). The notation “ $color^{-1}$ ” refers to swapping “color” to the opposite color of its current color; either from 1 to 0 or from 0 to 1. The notation “ $x := y$ ” means set  $x$  equal to  $x$  concatenate  $y$ . Finally, “ $v.c$ ” means the current color of the vertex  $v$ .

**Algorithm 8.3** overColor( $G$ )

---

```

1:  $\Gamma = \text{2D array}()$ 
2: for each vertex  $root \in G$  do
3:   Clear all coloring from  $G$ 
4:    $color = 1$ 
5:    $root.c = color$ 
6:    $\Gamma[root][root] := root.c$ 
7:    $lock = \text{false}$ 
8:   while  $G$  is not monochromatic or  $lock$  do
9:     if  $G$  is not monochromatic then
10:        $lock = \text{true}$ 
11:     end if
12:     for each vertex  $i \in G$  where  $i.c == color$  do
13:       for each neighbor  $n$  of  $i$  do
14:          $n.c = color^{-1}$ 
15:          $\Gamma[root][n] := n.c$ 
16:       end for
17:     end for
18:      $color = color^{-1}$ 
19:   end while
20: end for
21: return  $\Gamma$ 

```

---

The additional functions 4 and 5 offer alternatives to function 2 as described in **Lemmas 4.12, 4.13**. Function 4 is a naive polytime approach to reconstructing  $\phi$  that relies on the non-existence of graph symmetries that produce a disordering of vertex bijections between two sets A and B. Function 5 does not rely on this assumption but sacrifices time complexity such that it is no longer polytime. Function 6 is the actual backtracking algorithm used by function 5 to reconstruct  $\phi$ . Both functions 4 and 5 are constructed very similarly to function 2 and only differ in the criteria necessary to return true. They all still rely on function 3 to perform the most crucial part of the over all GI algorithm; over-coloring.

---

**Algorithm 8.4** canonicalNameCompare2( $G_1, G_2, invert$ )

---

```

1:  $OG_1 = G_1$ 
2:  $OG_2 = G_2$ 
3: if  $invert$  then
4:    $G_1 = \text{graphComplement}(G_1)$ 
5:    $G_2 = \text{graphComplement}(G_2)$ 
6: end if
7:  $\Gamma_1 = \text{Matrix concatenate } OG_1 \text{ and overColor}(G_1)$ 
8:  $\Gamma_2 = \text{Matrix concatenate } OG_2 \text{ and overColor}(G_2)$ 
9:  $A = B = \text{array}()$ 
10:  $tmpA = tmpB = \text{array}()$ 
11: Perform process described in Lemma 4.10 using  $A$  and  $B$  as arrays but copy  $A$ ,
     $B$  to  $tmpA$ ,  $tmpB$  prior to the final sort.
12: if  $A == B$  then
13:    $tmpG = \text{Perform process described in Lemma 4.12 using } tmpA \text{ and } tmpB \text{ as}$ 
    arrays, with output being the permuted graph  $(\phi)(G_1)(\phi^T)$ 
14:   if  $OG_1 == tmpG$  then
15:     return true
16:   else
17:     return false
18:   end if
19: else
20:   return false
21: end if

```

---

---

**Algorithm 8.5** canonicalNameCompare3( $G_1, G_2, invert$ )

---

```

1:  $OG_1 = G_1$ 
2:  $OG_2 = G_2$ 
3: if  $invert$  then
4:    $G_1 = \text{graphComplement}(G_1)$ 
5:    $G_2 = \text{graphComplement}(G_2)$ 
6: end if
7:  $\Gamma_1 = \text{Matrix concatenate } OG_1 \text{ and overColor}(G_1)$ 
8:  $\Gamma_2 = \text{Matrix concatenate } OG_2 \text{ and overColor}(G_2)$ 
9:  $A = B = \text{array}()$ 
10:  $tmpA = tmpB = \text{array}()$ 
11: Perform process described in Lemma 4.10 using  $A$  and  $B$  as arrays but copy  $A$ ,
     $B$  to  $tmpA$ ,  $tmpB$  prior to the final sort.
12: if  $A == B$  then
13:    $used = 1D \text{ array of } -1\text{s with size} = |A|$ 
14:    $\phi = 2D \text{ array of } 0\text{s with size} = |OG_1|$ 
15:   return  $\text{bt}(tmpA, tmpB, OG_1, OG_2, \phi, used, 0)$ 
16: else
17:   return false
18: end if

```

---



---

**Algorithm 8.6**  $\text{bt}(A, B, G_1, G_2, \phi, used, row)$ 


---

```

1: if  $row == |G_1|$  then
2:   if  $(\phi)(G_1)(\phi^T) \neq G_2$  then
3:     return false
4:   else
5:     return true
6:   end if
7: else
8:   for  $col = 0; col < |B|; col++$  do
9:     if  $used[col] == -1$  and  $A[row] == B[col]$  then
10:       $tmp\phi = \phi$ 
11:       $tmpUsed = used$ 
12:       $tmp\phi[row][col] = 1$ 
13:       $tmpUsed[col] = row$ 
14:      if  $\text{bt}(A, B, G_1, G_2, tmp\phi, tmpUsed, row + 1)$  then
15:        return true
16:      end if
17:    end if
18:  end for
19:  return false
20: end if

```

---

**9. Conclusion.** The question of Graph Isomorphisms time complexity being NPI or P has been around for years. Many techniques exist that allow specific cases of GI to fall into P. A common approach to tackling GI involves searching for a permutation matrix that permutes one graph into the other. However, another common approach involves canonical labeling. This technique relies on unique labels assigned to the symmetric groups of graphs so that any two graphs that are isomorphic share the same label, and no two symmetric groups share the same label. This idea was used as the premise for the function `overColor()`. This function can generate canonical labels for non-bipartite, connected simple or loopy graphs. It was also shown that in the case input graphs are bipartite, `GI()` can use the graph complements of input graphs for testing isomorphism. Due to the bijective nature of `overColor()` proposed in **Conjecture 5.1**, the pre-processed adjacency matrices of weighted graphs, directed graphs, and multigraphs were shown to be concatable to the SCL of their more simplified simple or loopy representations. It was shown that `overColor()` is polytime at  $O(n^4)$ .

Three comparison algorithms were presented that rely on `overcolor()` in their assessment of isomorphism. The first, 10.2, simply compares SCLs directly and is polytime. The second, 10.4, attempts a naive reconstruction of  $\phi$  but is clearly only a near GI solver, and is also polytime. Third, functions 10.5 and 10.6, present a backtracking approach that exhaustively searches  $\phi$  space to test for isomorphism while heuristically pruning its search by leveraging the canonical properties of `overColor()`'s SCLs, but sacrifices speed such that only its average case is polytime while its worst case is factorial. Lastly, experimental evidence was provided to demonstrate `GI()` using `overColor()` to compare graphs for isomorphism. In doing so, it can be observed that real world performance of `GI()` is actually better than  $O(n^4)$ , estimated to be roughly  $n^{2.5}$  for the sample cases tested for. Also, specific test cases on sets of k-regular graphs and Johnson graphs were evaluated using the comparison algorithm 10.2 which demonstrates these common problematic graph types for GI could also be handled by the proposed algorithm. These experimental results suggest that `overColor()` is a bijective function and if **Conjecture 5.1** can be proven, this would imply graph isomorphism is polytime solvable for connected simple, loopy, weighted, directed, and multigraphs.

**10. Future Work.** First and foremost, showing `overColor()` to be invertible would be the most interesting thing to pursue. However, other interesting results and paths have revealed themselves during the research involved with this paper. Regarding optimizations, `overColor()`, most of `canonicalNameCompare()`, and even parts of `GI()` can all be parallelized because of the lack of dependencies in the functions. `OverColor( $G_1$ )` and `overColor( $G_2$ )` can be done in parallel for example. Also, each iteration of the outermost forloop of `overColor()` can be done in parallel. And the list goes on.

Another optimization is in the canonical label comparison step. If a matrix  $\Gamma_a$  contains a certain set of elements in a column, I strongly suspect another matrix  $\Gamma_b$  can only contain that same column set if their respective graphs  $G_1$  and  $G_2$  are isomorphic. This is because that column is essentially the canonical label for the rooted graph corresponding to that column's vertex. Two graphs with vertex and edge counts equal to  $|V|$  and  $|E|$  can only have isomorphic rooted graphs of sizes  $|V|$  and  $|E|$  if they themselves are isomorphic. Thus, after sorting  $A$  and  $B$ , comparing only the first element in each should be enough to show the whole arrays are equivalent. This also majorly improves storage performance of these canonical labels for practical



purposes such as in graph databases. If only the first elements are needed to conduct the isomorphism check, only the first elements need to be stored.

Another area of interest is in reducing the time complexity offered in this paper;  $O(n^4)$ . The upper bound given here was only intended to establish the algorithm as polytime and was not concerned with a more refined time complexity assignment. Lastly, I also strongly suspect the process over-color itself can be simplified to some sort of dynamic programming algorithm that only records some equivalent to vertex CIs anytime a vertex is pathed to for the first time along a unique path, for every unique path starting from root  $r$ . This function may also be easier to show invertibility if it is in fact possible.

## REFERENCES

- [1] L. BABAI, *Graph isomorphism in quasipolynomial time*, 2015.
- [2] L. BABAI AND W. M. KANTOR, *Computational complexity and the classification of finite simple groups*, 1983.
- [3] L. BABAI AND E. M. LUKS, *Canonical labeling of graphs*, 1983.
- [4] H. L. BODLAENDER, *Polynomial algorithms for graph isomorphism and chromatic index on partial  $k$ -trees*, 1990.
- [5] C. J. COLBOURN, *On testing isomorphism of permutation graphs*, 1981.
- [6] B. D. MCKAY AND A. PIPERNO, *Practical graph isomorphism ii*, 2013.
- [7] D. EPPSTEIN, *Subgraph isomorphism in planar graphs and related problems*, 1999.
- [8] V. KABANETS AND J.-Y. CAI, *Circuit minimization problem*, 2000.
- [9] R. E. LADNER, *On the structure of polynomial time reducibility*, 1975.
- [10] H. W. LENSTRA, *Factoring integers with elliptic curves*, 1987.
- [11] D. LOKSHANOV AND M. PILIPCZUK, *Fixed-parameter tractable canonization and isomorphism test for graphs of bounded treewidth*, 2014.
- [12] G. MILLER, *Isomorphism testing for graphs of bounded genus*, 1980.
- [13] D. NEUEN AND P. SCHWEITZER, *An exponential lower bound for individualization-refinement algorithms for graph isomorphism*, 2017.
- [14] C. POMERANCE, *A tale of two sieves*, 1996.
- [15] P. W. SHOR, *Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer*, 1999.
- [16] G. R. V. ARVIND AND O. VERBITSKY, *Graph isomorphism, color refinement, and compactness*, 2015.