Exploration of the Graph Isomorphism Problem

David Koplow, dkoplow@mit.edu

December 9, 2021

1 Abstract

The question of whether or not there is an efficient algorithm to determine if two graphs are isomorphic is unsolved. It is a subject of controversy: some believe one likely exists, while others believe it is impossible. While there are a number of solved sub-types of graphs, no algorithm has been proven to work for every case. In this paper I present a polynomial time algorithm that appears to generalize for non-strongly regular graphs. If there is a polynomial time solution to make it work on strongly-regular graphs as well, it could mean that GI=P.

2 Introduction and Motivation

This UROP is a continuation of research I began on my own about six months ago. The question about whether or not there exists a polynomial time algorithm to determine if any two arbitrary graphs are isomorphic is unsolved. The problem description is very straightforward so when the lack of a solution was mentioned in a course I took last semester, I was surprised. During the end of the semester and over the summer I began thinking about novel methods for identifying if two graphs are isomorphismic in polynomial time. I ran thousands of tests on dozens of different types of graphs, and my algorithm seemed to be able to discover a mapping of each test. So, I reached out to Professor Williams to ask about methods to more rigorously test my algorithm. She gave me a number of suggestions for graphs to try that many known algorithms that aren't polynomial time struggle with. While at first my algorithm was able to solve these with relative ease, as I tested higher node counts I encountered specific graphs where my algorithm fails. The goal of this UROP was to further develop my algorithm and explore these problematic graphs in more detail to better understand why the original iteration of my algorithm didn't work. In this paper, I detail the most recent form of my algorithm and my findings about these edge cases.

The Graph Isomorphism Problem is one great theoretical and practical importance. Graphs are one of the most common and expressive forms of data representation, and are used in fields ranging from social media, and circuit design, even to chemical engineering.[3][2] Yet, we have yet to find an algorithm that is able to efficiently determine if the structure of two graphs with different labelings are the same.

The problem is also of great theoretical concern because of its place in the polynomial time hierarchy. Algorithems that depend on the speed of determining if two graphs are isomorphic are in the Graph Isomorphism (GI) Complexity class.[1] If GI were in fact not equal to P, then $NP \neq P$, putting an end to one of the most famous questions in all of computer science. On the other hand, if $GI = P$, then it would open many possibilities in industry

3 The Algorithm

3.1 Overview

The mapping between two isomorphic graphs can be checked in polynomial time. Thus, any mapping produced between two non isomorphic graphs can be invalidated in polynomial time if the mapping can be generated in polynomial time. So, the problem can be simplified to discovering a valid mapping between two isomorphic graphs, G1 and G2.

I first process the graphs in such a way that the initial labeling and order of edges doesn't matter. This is done through a modified version of breadth-first search (BFS) from the perspective, or *view*, of each node. The search can be used to find the nodes at each *depth* from the root which will allow us to calculate the *dominance* of each node from that view. This search is done in $O(N+E)$ time, the dominance calculation for one node is done in $O(N^2 log(N))$ time, and there are $O(N)$ nodes, so this step takes $O(N^3 log(N))$ time.

While generating the dominances, I can produce dominance layouts for each node and group them together using a hash table, which doesn't increase the asymptotic run time. If there are any groups that contain only one view from G_1 and G_2, I know that the roots of both views must map together. If not, then I can randomly match two nodes in smallest group available. I can use this information to find more nodes that map together.

Using this information, I then update the mapping and recalculate the dominance of each view using the new mapping to generate new mapping groups for the views of all nodes that have not yet been mapped. This process will repeat at most N times because for every iteration, at least one node is mapped. As a result, the overall algorithm is $O(N^4 log(N))$. But, in most cases, only one or two loops will be needed, suggesting the average run time of this algorithm is $O(N^3 log(N))$. This run time varies depending on the type of graph being analyzed. If the weights of edges are confined to being integers, I can use an $O(N)$ sorting algorithm when calculating the dominance, making the over all run time $O(N^4)$.

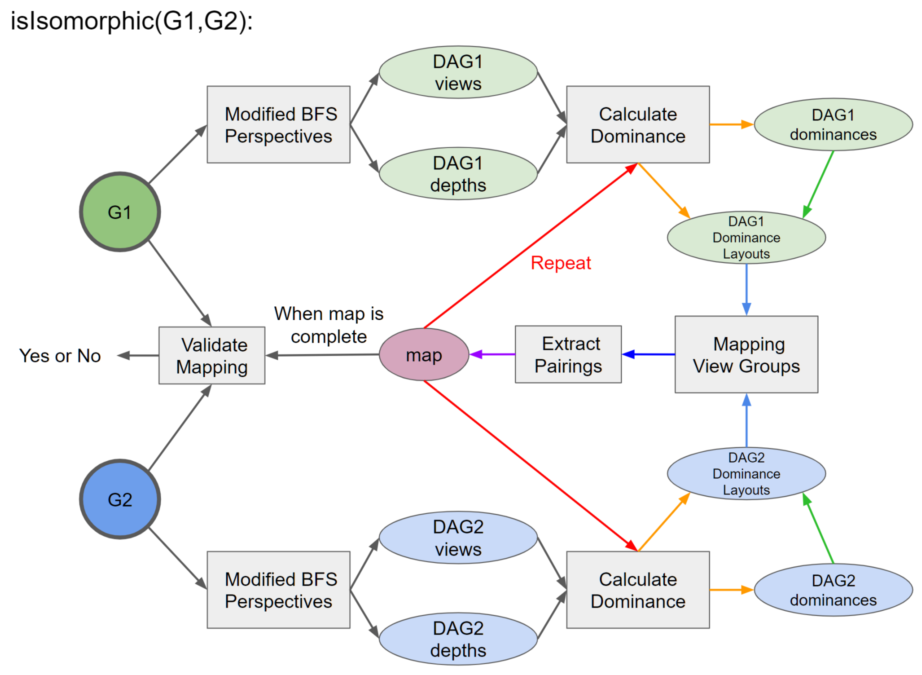

Figure 1: Overview of algorithm flow chart

3.2 Views

In this section I explain the initial BFS searches in more detail. I first need to define a data structure *DAGNode*, which stores five characteristics: id, index, depth, children, side-edges, and back-edges. The id is an integer representing to the original labeling of that node in the inputted graph. *index* indicates if the node has already been mapped. It is initialized to -1, and set to a value i when it is the ith node to be mapped. *depth* stores the depth at which the node was discovered from the modified BFS search described below, initialized to infinity. *children*, *side-edges*, and *back-edges* are lists of the indices of all incoming and outgoing edges from the node to another node, its *neighbors*, with a deeper depth, the same depth, and a shallower depth respectively. All three are initialized to empty arrays.

First I initialize a list of DAGNodes, one for each node in the graph I am searching. Let the *root* of the view be the node from which I want to calculate the view. I set the depth of the DAGNode of the root to 0. Let A be a list of the node ids I am currently searching, initially containing only the root. Let B be a list of the node ids I am going to search in the next step. I look through every node in A. For each, I look at the edges connected to it. Given an edge from the node currently being examined, n_1, to another node, n_2, the edge is given the following classification: child if the depth of $n_1 > n_2$, side-edge if depth of

$n_1 = n_2$, and back-edge if depth of $n_1 > n_2$. n_2 is appended to the respective list in the DAGNode of n_1. If n_2 is a child of n_1 and its depth is not equal to that of $n_1 + 1$, I add it to B, and set its depth to that of $n_1 + 1$. Then I set the A = B and erase B. I repeat this process until B is empty after all nodes in A have been examined.

Next, I group all nodes at each depth into a list, then collect all of those lists in a dictionary which is called *depths*.

This step is $O(N + E)$ as there are N nodes, and E edges, each of which is examined at most twice. For dense graphs this is on the order of $O(N^2)$, but it is still in polynomial time.

3.3 Dominance of a View

In this section I describe how I calculate the *dominance* dictionary of a view, which will allow us to create a *dominance layout*. The *dominance layout* is the representation of the view that is independent of initial labeling.

The *dominance* of a *view* is a dictionary which maps every node to a number that represents a sort of priority which it has compared to other nodes at its depth. It is calculated in three steps, each of which can be modified to fit the exact characteristics of the graph being analyzed. The description below is for simple graphs as it is the easiest-to-understand version. The nature of the modifications required for other types of graphs is described in the example cases section.

The stages of the dominance calculation are as follows: compare *dominance* of children, break ties with *dominance* of side edges, and break further ties with *dominance* of back-edges.

Stage 1: I initialize the *dominance1* of every node to $2(N + 1)$. Starting at the deepest depth, I sort all of the children of each node at that depth by dominance. Then, for every node at that depth, I calculate its dominance by looping through every other node at that depth and determining which is *stronger*. If the node being examined is stronger then I add $2(N + 1)$ to its dominance. In Stage 1, node n_1 is *stronger* than n_2 if either it has a greater index, or n_1's most dominant child is more dominant than that of n_2. If it is a tie, then I compare the next most dominant children. If n_1 and n_2 have a different number of children, after all children of the one with the least children have been compared, the node with more children is considered stronger.

Stage 2: I sort all side edges of every node at the current depth by their *dominance1*. After calculating the *dominance1* for a given depth, I then calculate *dominance2*. *dominance2* is initialized to be equal to *dominance1*. I then look at the side edges of each node at that depth. Similar to Stage 1 I calculate *dominance2* by determining how many other nodes at that depth the examined node is *stronger* than. Here, however, stronger is calculated differently. Given two nodes at the depth n_1 and n_2, whichever has a larger *dominance1* is considered *stronger*. If it is a tie, then I look perform a similar check as I did in the *stronger* calculation of Stage 1, but this time using the side edges instead of the children. If n_1 is stronger than n_2, I add $N + 1$ to its *dominance*.

Stage 3: Stage 3 begins after the *dominance2* of all depths has been computed. I now go through the depths in the opposite direction: from shallowest to deepest. This is done because back-edges connect to nodes with lower depths. For each node at a given depth, I sort all back edges by their *dominance2*. I also initialize *dominance3* to the value stored in *dominance2*. Then, just as with Stage 1 and Stage 2, I compute *dominance3* by comparing each node at a depth to every other node at that depth to see which is *stronger*. Here, stronger is determined by the *dominance2* of the back-edges in the same manner as in Stage 1 and Stage 2. *dominance3* is the final stage and what the function returns, so for the rest of the paper it will just be referred to as *dominance*.

The *dominance layout* of a *view* is the depth groups sorted by dominance with the children, side-edges, and back-edges lists for every node sorted by dominance. Two *dominance layouts* from the views of n_1 and n_2 are considered equal if all of the following requirements are met:

1. The number and magnitude of depths are the same.
2. The number of nodes at each depth is the same.
3. The dominance, index, and color of the of ith node in a depth of n_1 is the same as the ith node of the depth of n_2.
4. The children, side-edge, and back-edge lists are the same length and the dominance, index, and color of the node at each index matches for corresponding nodes in the *dominance layout* of n_1 and n_2.

I ultimately decided to export all of these features in the form of an array for each view. This allowed me to multi-thread this processes, at the cost of accessing all of the information before being able to determine if two nodes have different dominance layouts. In the exported form, the array can then be hashed to reduce the comparison time. But, to ensure the accuracy of the program I refrained from using the hashing method in my tests. When I also removed type inference from my code, it ran about 500 percent faster than originally on larger graphs.

An edge can connect to a maximum of two nodes so $O(N^2)$ nodes to be sorted. 3N dominances need to be calculated for each view and there are 2N views. This means it takes $O(N^3 log(N))$ to calculate the *dominance3* for one iteration of the algorithm. There are a maximum of N iterations which would make this step take at most $O(N^4 log(N))$ time.

3.4 Mapping Groups

The dominance layouts of the two graphs can be used to find which nodes map to each other. This process involves first grouping the indices of all *views* which have the same *dominance layout* from G1 and G2. These groups are then categorized by the number of *views* that share a *dominance layout*. If there are dominance layouts in G1 that only have one copy of in G2, these nodes must match if the graph is isomorphic. Otherwise, the algorithm looks at the group with the smallest cardinality, and tries matching the node for the first view in G1 with the first in G2. If this leads to an invalid mapping, which can be detected in one step, the algorithm tries the next possible node in G2.

If there are any dominance layouts in G1 that aren't in G2, or the cardinally of the number of views with a certain dominance layout in G1 is different than G2, then no mapping exists, so the graphs aren't isomorphic.

This method only uses information about the structure of the graph and known pairings to determine groupings. So, eventually I came up with a new method of grouping that in addition uses information about nodes that could pair with the current graph information, even if they aren't definitely known to pair. This was done by coloring nodes that exist in a single group with a unique color, and then including the color as part of the dominance calculation. This method proved more efficient than the original because it made fewer mistakes that had to be back tracked and it required less memory allocations. However, it ultimately didn't solve the failure cases in certain strongly regular graphs.

3.5 Information Extraction and Map Updates

After determining a pair of nodes that likely map, other pairs can be extracted by comparing their dominance layouts. The views of both nodes are compared, and any corresponding nodes that have yet to be mapped and share a unique dominance for all nodes at its depth are also mapped to each other. New mapping information can be used to differentiate nodes which originally had the same dominance layouts in G_1 and G_2.

Because there are N nodes in a graph, this process takes $O(N)$ time. A maximum of 2N nodes need to have their information extracted over the course of one iteration so it takes $O(N^2)$ time. The $O(N^3 log(n))$ time of calculating the dominance layouts for one iteration is the most important factor so if it takes all N iterations to complete, this algorithm takes a total of $O(N^4 log(N))$ time.

4 Example Cases

Below are a few example cases illustrating how this algorithm can be generalized to different graph types. If a graph is already labeled, assign each label a number and run the algorithm. If it is unlabeled, assign each node a random distinct number from 1 through N and run the algorithm.

4.1 Simple Graphs

This will work correctly with the basic algorithm, or any other version provided when the graph is adapted to fit that type. For example, to use the Digraph algorithm described below, every edge in the simple graph should be replaced with one edge going in and one going out, both with weights of 1.

4.2 Digraphs

To make the program run faster (though not asymptotically for dense graphs), nodes should be stored in such a way that they capture the weights of all in and out edges to and from that node. This can be done by adding an extra dimension to the adjacency matrix such that the first element of the edge from node i to node j represents the weight of an in edge, and the second element represents the weight of an out edge.

For digraphs to work, two modifications need to be made to the algorithm. The first is that I must add a step at each *stronger* function in Stages 1 through 3, to break ties with the weight of that edge. Additionally, all sorting done to children, side-edges, and back-edges done in the dominance calculation must first sort with dominance as the most significant feature, then edge weight as the least significant feature. Out edges should be compared before in edges. The second modification is that when checking if two dominance layouts are equivalent, the in and out edge weights must also match for any nodes with the same depth and dominance.

5 Test Cases

After implementing the algorithms described above, I tested them out on thousands of graphs. I tried graphs ranging from 5 to 500 nodes. Exactly how the edges were connected varied depending on the type of graph. All experiments followed the predicted polynomial time trend. The types of graphs tried and the corresponding success of the algorithm on it can be seen in Table 1.

5.1 Random Dense Graphs

Dense graphs with random edges can be very useful when determining the performance of the algorithm. They are easy to generate and make for useful test cases for a general graph. Random dense graphs are also one of the easiest types of graphs for a graph isomorphism algorithm to function on because they are highly asymmetrical, making them good candidates for assessing the average performance of the algorithm.

The graphs were generated by defining a graph of n vertices and linking any vertex with each other with a probability of p. In the experiments below, $p = 0.5$. The performance of the algorithm as a function of the number of nodes is shown in Figure 2. The points seem to stay well under the theoretical worst case bound. A noticeable feature of the graph is the slight curve up in the time as N gets large, which theoretically wouldn't be expected in a polynomial time algorithm. But, it seems too gradual to be a result of an exponential or pseudo-polynomial function. Rather, it appears that this is a result of physical limitations of my computer. My the program doesn't have enough RAM to store all of the graph view layouts, so frequent memory allocations need to be made, slowing the performance. Additionally, as the program runs for longer, the operating system temporarily pauses the execution of the program to run

Type	Undirected	Digraphs	Weighted
Binary Trees	YES	YES	YES
Bipartite	YES	YES	YES
Cayley Graphs*	YES	YES	YES
Clique	YES	YES	YES
Complete	YES	YES	YES
Connected Clique Sub-Graphs	YES	YES	YES
Connected Regular Sub-Graphs	YES	YES	YES
Connected Star Sub-Graphs	YES	YES	YES
Connected Strongly Regular Sub-Graphs*	NO	NO	YES
Double Binary Trees	YES	YES	YES
Random Dense*	YES	YES	YES
Random Sparse	YES	YES	YES
Regular	YES	YES	YES
Star	YES	YES	YES
Strongly Regular*	NO	NO	YES
Wheel	YES	YES	YES

Table 1: Table of graph types attempted and if all trials succeeded.

other necessary programs. Additionally, even at it's highest rate, it is still less than the theoretical bound.

Experiments were also conducted in which the terminal node of two edges in the graph were swapped in one of the isomorphic graphs, to produce non-isomorphic graphs. The algorithm correctly identified all these graphs as non isomorphic.

5.2 Strongly Regular Graphs and Sub-Graphs

Strongly regular graphs were both taken from data base, and also generated by Sage Math in their python library and then imported in Julia and experimented on.[5]

Strongly Regular graphs seem to be the only type of graph that this algorithm truly struggles with. Strongly regular graphs are defined by four parameters: v, k, λ, and μ. v indicates the number of vertices in the graph. k represents the degree of each node in the graph. λ is the number of neighbors shared by any vertex with each adjacent vertex. μ represents the number of common neighbors among any two non adjacent neighbors.[8]

Only certain combinations of these parameters actually produce graphs that

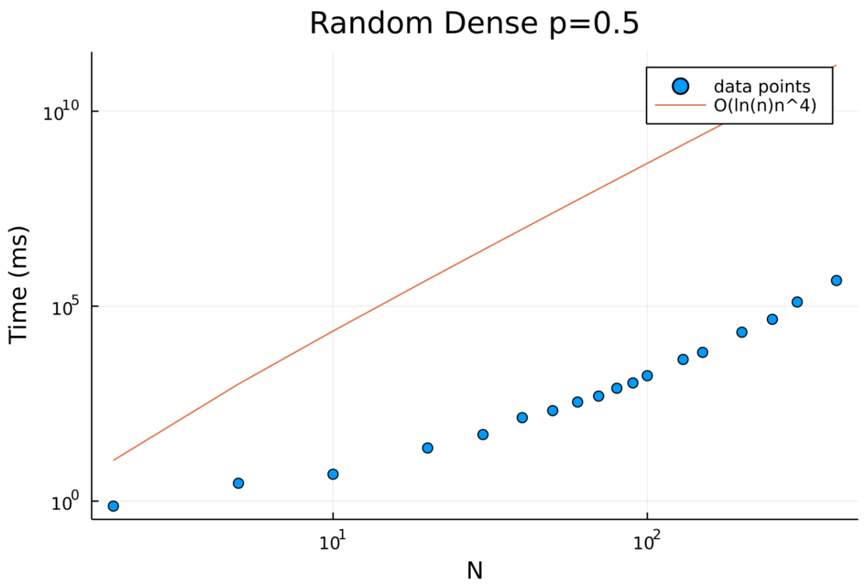

Figure 2: Performance of algorithm on random dense graphs

exist, and some can represent a set of many non-isomorphic graphs. I tested the first few hundred of these parameter sets. While the vast majority of isomorphisms for these graphs were correctly discovered, there were some edge cases, which became more frequent as the complexity of the graphs increased. For example, the first graph that the program wasn't able to correctly calculate had a parameter set of (125, 96, 74, 72), but graphs with all similar parameters that existed functioned correctly. The next problematic graph wasn't until the parameter set (156, 125, 100, 100). After researching at length for mentions of strongly regular graphs with these parameters in literature, consulting annotations in strongly regular graph databases, and reaching out to an expert in the field, Professor Josh Grochow at the University of Colorado, I have been unable to determine what about these graphs makes them special.[6] While it may have something to do the degrees of symmetry in the graphs, this can't be the only factor, as many isomorphisms for more complex graphs are able to be verified correctly. However, this does play a partial role, since changing even a single edge or node in the graphs of the edge cases lead to the algorithm to performing correctly.

Interestingly, I was able to produce more graphs that my algorithm failed on by connecting all nodes of a complex strongly regular graph, that originally was correctly processed, to a single super node. I could also produce incorrect cases when I took a strongly regular graph that normally worked, but added edges of a new weight to all nodes that weren't originally connected. This hints that the issue might in part be a result of limitations in how deep the dominance layouts get. Perhaps, there isn't enough distinction in the side edges. This led me to think about how I could extract more information out of the graph. This is when I realized that in the original form of the algorithm, I wasn't including information about possible matching groups in the dominance calculation. As an experiment, I decided to add a color feature to the SmartDAGNodes, and use this to store what matching group the node was in. Then, was able to include the color as a factor in the dominance calculation. As described in the Mapping Groups section, this seemed to improved the speed of matching, but didn't prevent the problematic cases. This may be a result of the first match still being incorrect because all nodes are perceived as having the same dominance. But, as of now I have been unsuccessful at figuring out a way to better make this first step.

Experiments were also done on graphs with strongly regular sub-graphs. A number of scenarios were tried. First, the parameters of the two sub-graphs were varied. Either they had different parameters, or the same parameters. The sub-graphs with same parameters could either be isomorphic or non-isomorphic. Different forms of connectivity were also tested. The graphs could be unconnected, connected with a single edge, or in the case of the graphs with the same parameters, an edge was added in between each corresponding node in the sub-graphs. None of these caused problems for the algorithm, unless one of the sub-graphs was itself an edge case.

5.3 Cayley Graphs

Like strongly regular graphs, Cayley graphs have many degrees of symmetry, making them hypothetically a very difficult type of graph for a graph ismorphism algorithm. To test my algorithm, I pulled samples of Cayley graphs for every n that was available on the House of Graphs database, that were then converted from the g6 format to an adjacency matrix using Sage Math.[**hog**] The resulting graph was then loaded into Julia, and my algorithm was run on it. Isomorphisms for all graphs pulled from the data base were correctly identified, and the performance of my algorithm on the Cayley graphs seems to be far better than the theoretical upper bound. As can be seen in Figure 3, the actual run times seem to follow a $O(n^2)$ trend. Specifically, the slope of the best fit line for the set of data produced by $(ln(x), ln(y))$ where x is the number of nodes in each graph and y is the time of execution, is 0.732528, suggesting the original data is well fit by a polynomial of degree $e^{0.732528} = 2.080$. This is likely a result of the the Cayley graphs being fairly sparse compared to the random Dense graphs and Strongly Regular graphs, while also not being nearly symmetric enough to really confuse the algorithm.

6 Conclusion

The past semester has helped me make significant progress in better understanding the limitations of my algorithm and the complexities of the Graph Isomorphism Problem. Through successful experiments with new types of graphs,

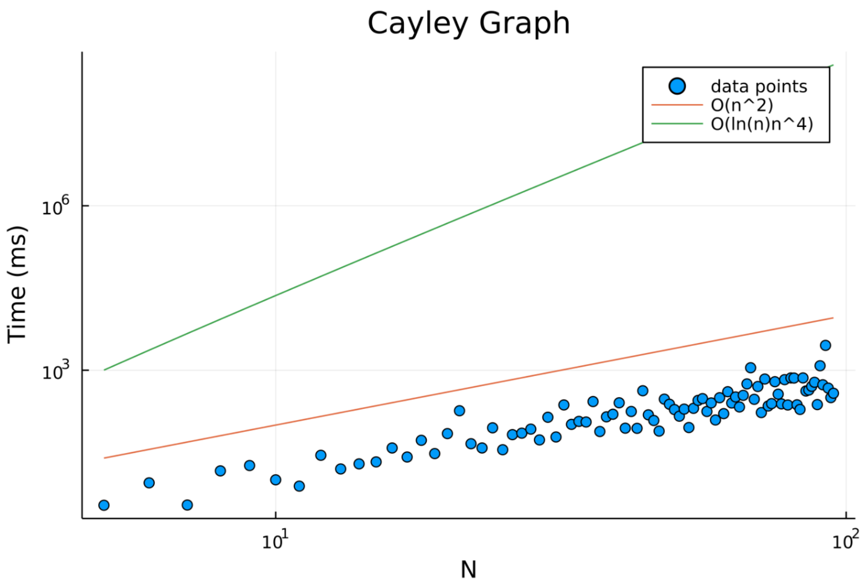

Figure 3: Performance of algorithm on Cayley graphs

and the discovery of new cases that the algorithm doesn't work on, I have been able narrow down some of the characteristics that cause it to fail. Namely, very high symmetry and very high connectivity. I have also been able to use this information to make improvements to the efficiency of my algorithm, and while in its current form, it still does not work for all strongly regular graphs, my work this semester has supplied me with a direction for future research. It's possible that one method of better understanding how my algorithm fails is by studying other algorithms for estimating or identifying isomorphisms, such as Weisfeiler-Leman, and seeing if my algorithm is subsumed by it.[7] In the case of the Weisfeiler-Leman algorithm, my algorithm seems to function very differently. At each iteration, instead of only considering a node's nearest neighbors, my algorithm looks at the entire graph from the perspective of that node. Additionally, instead of generating a single canonical representation of the both input graphs and comparing them as in the Weisfeiler-Leman method, my method tries to match nodes as it proceeds. My algorithm also doesn't compare the union of edges of combinations of non-adjacent nodes, however, as is the case in Weisfeiler-Leman-k algorithm. Although they function differently, it is still possible that my algorithm is subsumed by Weisfeiler-Leman-k for some low dimensional k.[6] This is a path for further research. Another path for future research is exploring in more detail how my algorithm behaves on other types of graphs. While I have done preliminary experiments with hyper graphs and multi graphs, more work is needed to assess the algorithm's true performance.

All things considered, this project has been a very valuable experience. I am grateful to Professor Williams for proving me with this opportunity to further pursue it in the form of this UROP, and plan to continue research into it on my own in the coming months.[4]

References

[1] Harm Derksen. "The Graph Isomorphism Problem and approximate categories". In: *Journal of Symbolic Computation* 59 (2013), pp. 81–112.

[2] Y E Cho Henry S Baird. "An artwork design verification system". In: *Proceedings of the 12th Design Automation Conference* (1975), pp. 414–420.

[3] Christophe-André Mario Irniger. *Graph Matching – Filtering Databases of Graphs Using Machine Learning Techniques*. 2005.

[4] Associate Professor Virginia Vassilevska Williams at the Massachusetts Institute of Technology. personal communication. 2021.

[5] Ted Sepnce. *Strongly Regular Graphs on at Most 64 Vertices*. Ed. by University of Glasgow Department of Mathematics. URL: `http://www.maths.gla.ac.uk/~es/srgraphs.php`.

[6] Assistant Professor Josh Grochow at the University of Colorado. personal communication. 2021.

[7] B.Yu. Weisfeiler and A.A. Leman. "The Reduction of a Graph to Canonical Form and the Algebra which Appears Therein". In: *Nauchno-Technicheskaya Informatsia* 9 (1968), pp. 12–16.

[8] Eric W. Weisstein. *Strongly Regular Graph*. URL: `https://mathworld.wolfram.com/StronglyRegularGraph.html`.