Coherent Configurations and the Graph Isomorphism Problem

Gabriel Stauth

Abstract. Following the recent article "Graph Isomorphism in Quasipolynomial Time" by László Babai [1], this paper examines some of the properties of combinatorial configurations that arise in the study of the Graph Isomorphism problem. Specifically, their work includes a key discussion of an algorithm known as Weisfeiler-Leman canonical refinement. This algorithm, when given any combinatorial configuration, will produce in polynomial time a canonical refinement of that configuration that is guaranteed to be coherent. We explore the mathematical underpinnings of this algorithm, along with a look at some of the basic ideas connecting the algorithm to its application in the problem of determining graph isomorphism.

A Math 501 Project
under the direction of
Dr. John S. Caughman

Submitted in partial fulfillment of the requirements for the degree of
Master of Science in Mathematics
at
Portland State University

December 4, 2017

Contents

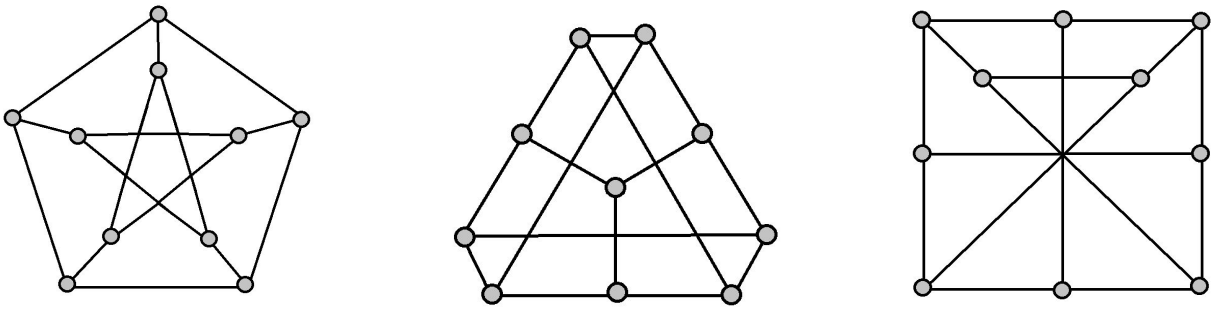
I. Introduction

In this 501 project, we consider a number of topics from the recent article "Graph Isomorphism in Quasipolynomial Time" by László Babai [1]. In doing so, we will examine a number of properties of combinatorial configurations that arise in the study of the Graph Isomorphism problem. Specifically, Babai's work includes a key discussion of an algorithm known as Weisfeiler-Leman canonical refinement. This algorithm, when given any combinatorial configuration, will produce -- in polynomial time -- a canonical refinement of that configuration that is guaranteed to be coherent. We explore the mathematical underpinnings of this algorithm, and we will consider some of the basic ideas connecting the algorithm to its applications in the problem of determining graph isomorphism.

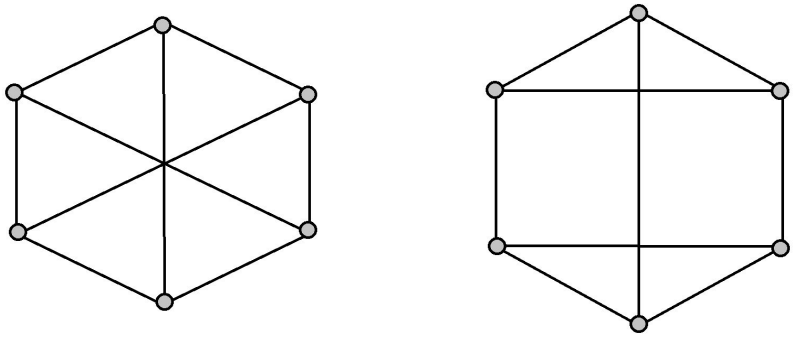
II. The Graph Isomorphism problem

First we define the problem of graph isomorphism. A **graph** G=(V,E) is an ordered pair where V is a finite set (whose elements are called **vertices**), and E is a set of 2-element subsets of V (called **edges**). When two vertices form an edge, we say they are **adjacent**. For basic terminology regarding graphs, we refer the reader to the fine textbooks by West [8] and Harary [5]. It is worth noting that our discussion also occasionally refers to a number of special types of graphs, such as directed graphs, multigraphs, or weighted graphs. Rather than attempt to include a comprehensive list of these definitions, we again refer the expositions found in the textbooks mentioned above.

The issue we wish to consider is to determine whether two given graphs A and B are actually the same graph, just displayed differently. Specifically, we recall that two graphs G=(V,E) and G'=(V',E') are said to be **isomorphic** if there exists a bijection f from V to V' such that two vertices x and y are adjacent in G if and only if f(x) and f(y) are adjacent in G'. For example, these three graphs are all isomorphic, as the reader may easily verify:



By contrast, the two graphs depicted below are not isomorphic:



Specifically, since the above graphs are not isomorphic, there is no way to label the vertices of these two graphs with corresponding labels in such a way that both graphs have the same adjacencies. This is left as a fun exercise for the reader, who will quickly discover that you can get fairly close, but there will always be obstructions to completion of such labellings. (A more experienced reader might notice structural differences in the graphs. For example, the graph on the left is bipartite and has girth 4, while the graph on the right is not bipartite and has girth 3. Since these facts and techniques are not needed in the remainder of the paper, we will not digress here to discuss them further.)

Traditionally, there have been probabilistic matching algorithms for constructing isomorphisms that work quickly most of the time but can have terrible worst case performance [2][3]. There are also algorithms that work quickly if certain restrictions are included, such as being planar or bipartite [2][3]. But for unrestricted worst case complexity, the best algorithms have performed no better than $e^{O(\sqrt{(n \log n)})}$ [4]. These difficulties have caused the graph isomorphism problem to be one of the few problems that have existed in the limbo between P and NP. Now an interesting side note is that it has been shown that if P≠NP then there is a problem in between [6]. So, as a result, one can observe that contrapositively, if we could show that there are no problems in between P and NP, then it must follow that P=NP.

We come now to the central advance made in Babai's paper, which is the topic of this project. His paper introduces an algorithm to correctly identify whether or not two graphs are isomorphic in (quasi-)polynomial time. This also reduces the String Isomorphism and Coset-Intersection problems as they both have polynomial conversions into Graph Isomorphism. This fact brings those problems into the closest complexity class to P, and leaves only a few known problems in limbo between P and NP.

In order to accomplish the construction of this new algorithm, Babai makes use of a combinatorial structure that is more general than standard graphs; these are structures known as coherent configurations. A configuration is essentially a special kind of partition of an NxN matrix that can be associated with a graph, and conforms to a number of special properties. The configuration of a graph can serve as a measure of symmetry -- highly symmetric graphs tend to have very simple coherent configurations, while asymmetric graphs have more complex ones.

III. Coherent Configurations

Given a finite set Ω (often {1,2...n}) , a **configuration** is a structure of the form $(\Omega;R_1,R_2....R_r)$ where each $R_i \subseteq \Omega \times \Omega$. We conventionally view the assignment of ordered pairs to the R_is as a coloring of Ω x Ω that satisfies the 3 conditions below:

1. Each color R_i is nonempty and together they partition Ω x Ω.
   (We remark that the condition that the R_i are nonempty is conventionally implied by the definition of a partition.)
2. The diagonal colors do not appear off the diagonal.
   (Vice versa, as well, so the diagonal colors and off diagonal colors are disjoint.)
3. Every color has a color that is its transpose.
   (A color could be its own transpose.)

The four small arrays below provide examples of configurations, where the letter in an entry denotes a color that is assigned to the ordered pair corresponding to that entry:

[A B]	[A B]	[A B C]	[A B B]
[B A]	[C D]	[C A B]	[C D C]
		[B C A]	[C B D]

(Note that the case where every element is unique is a configuration)

The three arrays below provide non-examples of configurations, for comparison:

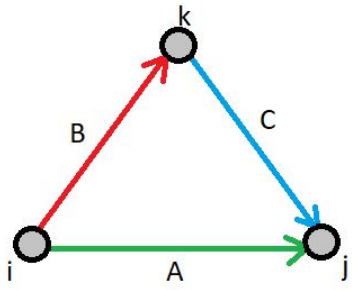
[A B]	[A A]	[A B C]
[B _]	[A A]	[B A C]
		[C B A]

The first of the three arrays above is not a configuration because the color classes do not form a complete partition. For the second array, we observe that a color on the diagonal appears off the diagonal, which is not allowed. And finally, for the third array, we observe that the transpose of color B does not match a color class exactly -- not A, B, or C.

To view this as a coloring of a graph, we would need to interpret the diagonal entries as colors of vertices, and since every entry is assigned a color, we would need to color both the edges and non-edges of a given graph. Alternatively, we may view a configuration as simply a coloring of the vertices and edges of a complete graph with loops.

Notationally, we follow the examples above, and we say that an edge (i,j) has color C, corresponding to the (i,j) entry in the matrix. Also for vertices, we say that vertex v is color C corresponding to the (v,v) entry.

There is one more property necessary for a configuration to be coherent, but first we offer one more definition. In a configuration, we say a **detour** for a given color is a path of length two, whose endpoints form an edge that is assigned that given color. The **type** of the detour is simply the ordered pair of colors along the path. Pictured below, we see an example of a B,C-detour for color A.



Finally, to be **coherent,** a configuration must also satisfy the property below:

4. There exist an array of **structure constants** p^A_{BC} that satisfy:
   $(\forall$ colors A,B,C$)(\forall (i, j) \in A)$
   $$p^A_{BC} = |\{k : (i, k) \in B \text{ and } (k, j) \in C\}|.$$

In other words, this condition is saying that, regardless of which A-colored edge we look at, there will be exactly p^A_{BC} detours of type B,C.

Alternatively, there is an equivalent matrix definition:

4. There exists an r by r matrix of constants for each color where the (B,C) entry in matrix A is:

$$p^A_{BC} = |\{k : (i, k) \in B \text{ and } (k, j) \in C\}|, (\forall (i, j) \in A).$$

This is a complex and subtle property, so let us look at some simple examples. First off, note that a trivial two-color configuration exists of any size. We simply let every diagonal entry be assigned to one color (A) while we assign all of the off-diagonal entries to the other color (B). There are 8 structure constants in this case, as we will see in a moment. The array appears thus:

[A B B B]
[B A B B]
[B B A B]
[B B B A]

One way to think of this configuration is that color A corresponds to staying in place, and B corresponds to moving to another vertex. This is equivalent to a coloring of the vertices and edges of the complete graph on 4 vertices.

Now, with only 2 colors there are 8 structure constants, and here they are, all listed out:

$p^A_{AA}=1 \quad p^A_{AB}=0 \quad p^A_{BA}=0 \quad p^A_{BB}=3$

$p^B_{AA}=0 \quad p^B_{AB}=1 \quad p^B_{BA}=1 \quad p^B_{BB}=2$

To read these, remember: it is the number ways to get to where the top type of edge is going by taking a detour of the bottom edges' types.

So, for example, $p^A_{BA}=0$ because there is no way to get back where you started by taking a B edge and then an A edge.

Similarly, $p^B_{BB}=2$ because there are always two different B,B detours for any B edge (each using one of the two other vertices)

Another way to see these is with 'detour matrices':

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 3 \end{bmatrix}, \qquad B = \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix}$$

In these matrices, the rows and columns are naturally indexed by the colors of the coherent configuration, and each entry corresponds to a unique structure constant. Note that both 'detour matrices' have the same grand sum (total of all elements).

Another fairly trivial type of coherent configuration is one where every color is unique:

[A B C]
[D E F]
[H I J]

This trivial configuration is clearly coherent because each color only appears once, the structure constants are just the detours of each type for each color.

Since the number of structure constants is the cube of the number of colors, there are 729 structure constants in this example. However, 702 of them are 0. The remaining 27 are all 1, and there are three for each color: for example $p^A_{AA}=1$ $p^A_{BD}=1$ and $p^A_{CH}=1$. There are no other detours for A, so all its other structure constants are zero.

Here in this example we see that the detour matrices are cumbersome and sparse, so it's easier to list the nonzero entry pairs for each color like so:

A: AA BD CH
B: AB BE CI
C: AC BF CJ

These lists are easy to compute, as each is just a row and a column, and we will we show later that this is an efficient way to compute all of the structure constants.

Here is a non-example, namely, a configuration which is not coherent:

[A B B]
[C D C]
[C B D]

This is not a coherent configuration because the Bs adjacent to the A would indicate that $p^B_{AB}=1$ while the B in the bottom row would indicate that $p^B_{AB}=0$.

IV. Vertex Color Awareness and Degree Awareness

The examples above can provide inspiration to us regarding one of the important properties that coherent configurations must have.

Vertex Color Awareness: In a coherent configuration, every pair of edges that share a color must have the same-colored endpoints. For example, if a B-colored edge goes from an A-vertex to a C-vertex, then all B-colored edges go from an A-vertex to a C-vertex.

Proof: Assume two edges (i,j) and (k,l) have the same color A.

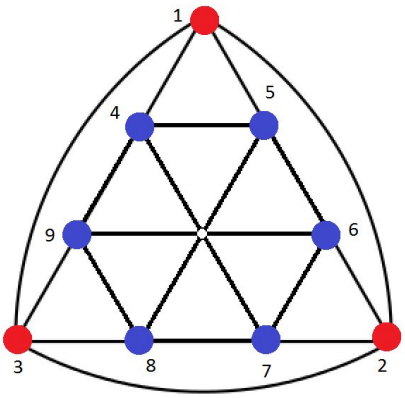
First we show that the starting vertices i and k must be the same color. To this end, let vertex i have color B. Now it must be the case that $p^{A}_{BA}>0$ because we have edge (i,j) of color A, and a detour that corresponds to the path (i,i) then (i,j) which is a detour of color type BA. Therefore, since edge (k,l) is also color A, there must be at least one BA-detour to get from k to l. Since B is a diagonal color, it corresponds to staying in place, so the only possible BA-detour is (k,k) then (k,l). This detour forces vertex k to have color B. Thus i and k have the same color.

A similar argument will show that the end vertices also have the same color. (Equivalently, we can make this argument by invoking the transpose property)

-QED

Next we wish to consider a somewhat larger example that illustrates a number of the other features that can be found in coherent configurations. Below we see another regular graph, and here, each vertex has 4 edges.

```
[A B B C C D D D D]
[B A B D D C C D D]
[B B A D D D D C C]
[E F F G H I J I J]
[E F F H G J I J I]
[F E F I J G H I J]
[F E F J I H G J I]
[F F E I J I J G H]
[F F E J I J I H G]
```



This example has two colors of vertices A and G, and 5 types of edges, B,C,E,H and J. There are also 3 types of non-edges: D, F and I. This gives us a total of 10 colors, which means that there are 1000 structure constants to verify. For fun, the reader is invited to check these by brute force.

Obviously, verifying all of them directly by hand is impractical, especially because 90% of them are zero. Indeed, even automating this task is quite slow on larger graphs (though still a tractable task on order of ~O(n^7)). Instead let us consider each color in turn.

A: this color is for the three vertices that are mutually adjacent and are part of two triangles (of types AAA and AGG)
G: this color is for the other six vertices, they are only included in one triangle each (AGG).
B: this is the color of the A,A edges
C,E: these are the colors of the AG and GA edges, respectively.
D,F: these are the colors of the non-edges between the two vertex colors.
H: these are the colors of the GG edges used in the AGG triangles.
I: these are colors of the non-edges between G vertices.
J: the color of the GG edges not used in a triangle.

In this example you can see Vertex Color Awareness at work. This property is why CE and DF are inverses of each other, instead of being one color.

This example is also large enough to see another important property of coherent configurations.

Degree Awareness: If two vertices share a color, they have the same out-degrees of each color and the same in-degrees of each color.

In our example above, notice that every A vertex has 1A, 2Bs, 2Cs, and 4Ds going out and every G vertex has 1E, 2Fs, 1G, 1H, 2Is, and 2Js. Similar observations hold for edges coming in to these colors.

Proof: Pick a color A with edge (i,j). Now, by the transpose property (3), let the inverse of A have color B. Let vertex i have color C. It follows that $P^C_{AB}=k>1$, because this is the number of ways to step out and back along an A edge. Indeed, this k is the out-degree of vertex i with respect to color A. By vertex color awareness, all these edges start with color C and therefore all have the same out-degree.

-QED

These properties give us some powerful indicators to look for in determining whether a configuration is coherent. However, these properties alone are not sufficient to force coherency. For example, any regular graph (uniform degree) has these properties, but may not be coherent.

V. The Detour List Lemma

Let us now consider the 'detour matrices'. First we make note of a property that I am calling the Detour List Lemma.

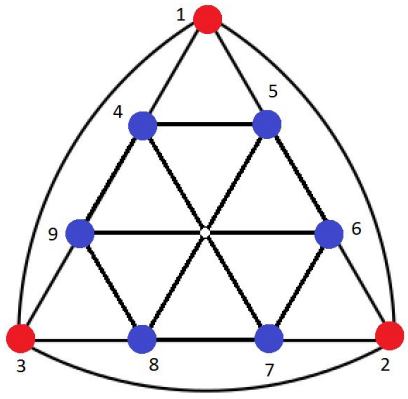
Detour List Lemma: The total of all of the entries in the detour matrix of any given color (its grand sum) is n, the number of vertices in the graph. Accordingly, the detour matrix may be represented by a list of length n, rather than by an (r x r)-matrix.

Proof: Pick any color, A, and any edge (i,j) in A. Now look at all of the detours of the form (i,k)(k,j). First, note that there are exactly n of them, as there are precisely n options for k, and each is a valid detour. Now, each of those detours has an associated ordered pair of colors, the colors of (i,k) and (k,j), respectively. Each pair corresponds to increasing the count in the 'detour matrix' of A by one, as the entry is defined to be the total number of detours of that color pair.

-QED

This is another, much more compact, way to display the structure of a coherent configuration: First we recall the color matrix as before:

```
[A B B C C D D D D]
[B A B D D C C D D]
[B B A D D D D C C]
[E F F G H I J I J]
[E F F H G J I J I]
[F E F I J G H I J]
[F E F J I H G J I]
[F F E I J I J G H]
[F F E J I J I H G]
```



Then, instead of the ten 10x10 very sparse matrices that we would need, we can simply list the elements:

```
A:  AA  BB  BB  CE  CE  DF  DF  DF  DF
B:  AB  BA  BB  CF  CF  DE  DE  DF  DF
C:  AC  BD  BD  CG  CH  DI  DI  DJ  DJ
D:  AD  BC  BD  CI  CJ  DG  DH  DI  DJ
E:  EA  FB  FB  GE  HE  IF  IF  JF  JF
F:  EB  FA  FB  GF  HF  IE  IF  JE  JF
G:  EC  FD  FD  GG  HH  II  II  JJ  JJ
H:  EC  FD  FD  GH  HG  IJ  IJ  JI  JI
I:  ED  FC  FD  GI  HJ  IG  II  JH  JJ
J:  ED  FC  FD  GJ  HI  IH  IJ  JG  JI
```

This can be compacted even more by listing the number of duplications to:

```
A:  AA 2BB 2CE 4DF
B:  AB  BA  BB 2CF 2DE 2DF
C:  AC 2BD  CG  CH 2DI 2DJ
D:  AD  BC  BD  CI  CJ  DG  DH  DI  DJ
E:  EA 2FB  GE  HE 2IF 2JF
F:  EB  FA  FB  GF  HF  IE  IF  JE  JF
G:  EC  FD  FD  GG  HH 2II 2JJ
H:  EC  FD  FD  GH  HG 2IJ 2JI
I:  ED  FC  FD  GI  HJ  IG  II  JH  JJ
J:  ED  FC  FD  GJ  HI  IH  IJ  JG  JI
```

Potentially, this representation could be very useful for large graphs with few colors. Either way, this reduces what would otherwise be r^3 entries to the n*r nonzero ones.

VI. The Weisfeiler-Leman Algorithm

In this section, we reach the critical point of describing the Weisfeiler-Leman Algorithm by which any given configuration can be refined (as a partition) to a configuration that is coherent.

We begin with the algorithm itself. These few lines are the main work of the W-L refinement process. We start with an n x n color matrix with all positive entries, and an empty n x n x n+1 structure for the detour lists.

```
for(int i=0;i<n;i++){
   for(int j=0;j<n;j++){
      detourLists[i][j][0]=colors[i][j];
      for(int k=0;k<n;k++){
         detourLists[i][j][k+1]=C*colors[i][k]+colors[k][j];
      }
      Arrays.sort(detourLists[i][j]);
   }
}
```

The first two for-loops record each edge's original color on the bottom layer of the data structure. The third nested loop finds all the detours of that edge by iterating through all the vertices, and then stores that list of detours with the original color. The constant C is chosen to be large enough that the sorting works (greater than the number of colors, so at least n^2+1).

Once all of the detours have been found, then each the list for each edge is sorted. Because there are n^2 of these lists, each of length n+1, and because sorting is O(n log(n)), this whole process takes $O(n^3 \log(n))$ time.

The only other work is to rename the colors based on the detour lists, so that edges with the same list have the same color, and vice versa. This is done with a new linked list of the detour lists encountered, comparing the detour list of any edge to everything in the list so far. If it is there, then that position on the list is your new color; otherwise, add your detour list to the end and take that position as your new color.[1]

Fact: If no colors changed, then the resulting stable configuration is coherent.

Proving the statement above is a bit subtle. After all, it is easy to show that a coherent configuration is stable, but is something stable a coherent configuration?

First note that, because the input is a configuration, it already satisfies the first 3 properties. In other words, we just need to show the property regarding the existence of structure constants holds. Next, when the output is stable, that means that the colors do not change. This, in turn, means that their detour lists do not change. Since detour lists are equivalent to structure constants (by the above lemma), all the structure constants remain the same. Because the structure constants are, well, constant, that means they actually exist as constants. But this is the final requirement for a configuration to be coherent.

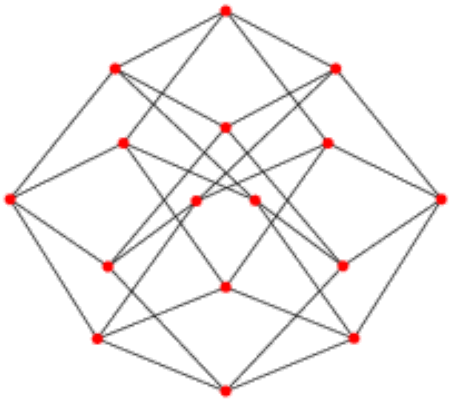
Some other properties of this algorithm are worth noting.

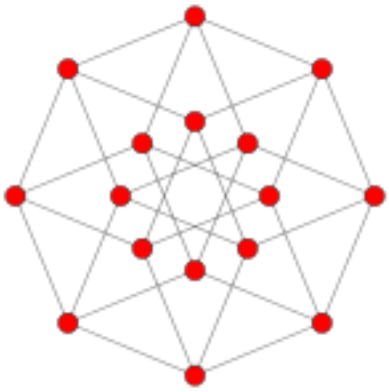
- The number of colors can only increase, because the color from the previous step is carried over.
- If the number of colors stays the same, that means that no colors changed.
- Thus, to terminate, one only needs to check the number of colors.
- Furthermore, since there is a maximum number of colors n^2 the algorithm must finish before then.
- In practice it finishes much faster than that, often needing only 2-3 iterations.

[1] This also means that colors are introduced in order, so reading left to right, every new color will be the one immediately after the previous new color. This prevents getting stuck in a loop relabeling a coherent configuration.

VII. Examples and Implementation

How does this actually affect the original graph isomorphism problem? Well two isomorphic graphs will have isomorphic coherent configurations (and, for the curious, there are indeed examples of nonisomorphic graphs that give the same coherent configuration). However that puts us back where we started, right? Actually no, while matching coherent configurations cannot prove definitively that two graphs are isomorphic, it can show that two graphs are not isomorphic. For example, consider the hypercube/tesseract and the Hoffman graph:





These two graphs have a lot in common: they have 16 vertices, each of degree 4; both are bipartite, and they are even cospectral [6]. But ,when they are run through the algorithm, we get:

A B C C C D D D E E E F G H H H B A D D D C C C E E E G F H H H C D A C C B D D E H H F G E E H C D C A C D B D H E H F G E H E C D C C A D D B H H E F G H E E D C B D D A C C E H H G F E E H D C D B D C A C H E H G F E H E D C D D B C C A H H E G F H E E I I I J J I J J K L L M M L L N I I J I J J I J L K L M M L N L I I J J I J J I L L K M M N L L O P O O O P P P Q Q Q R S Q Q Q P O P P P O O O Q Q Q S R Q Q Q J J I I J I I J L L N M M K L L J J I J I I J I L N L M M L K L J J J I I J I I N L L M M L L K 3 iterations	A B C B B C D C C D E D B C D C B A B C C B C D D C D E C B C D C B A B D C B C E D C D D C B C B C B A C D C B D E D C C D C B B C D C A B C B B C D C C D E D C B C D B A B C C B C D D C D E D C B C C B A B D C B C E D C D C D C B B C B A C D C B D E D C C D E D B C D C A B C B B C D C D C D E C B C D B A B C C B C D E D C D D C B C C B A B D C B C D E D C C D C B B C B A C D C B B C D C C D E D B C D C A B C B C B C D D C D E C B C D B A B C D C B C E D C D D C B C C B A B C D C B D E D C C D C B B C B A 2 iterations

So we have a clear difference in the two configurations produced, therefore the graphs are not isomorphic. The highly symmetric hypercube (right) only has 5 colors in its configuration, while the Hoffman graph has almost 20.

The algorithm was implemented in java, and is included in its entirety in the appendix. This was verified on a variety of examples such as the ones displayed in this paper. All examples are also in an appendix.

VIII. Conclusion

In conclusion, we have presented a polynomial time algorithm for generating coherent configurations from arbitrary graphs. These configurations can serve as an isomorphism rejection tool and are also used extensively in the rest of the paper to verify local areas and extend to the whole graph. In his paper, Babai recursively gives a vertex a unique color and runs this refinement on each of those graphs. In this way, we see that coherent configurations serve as a useful tool for detecting symmetries and as a way to quickly reject nonisomorphic graphs. Indeed, the Graph Isomorphism Problem is also important for the ramifications it has to the P=NP problem. If is is provably impossible to solve in P then P≠NP, while on the other hand, if we prove that there are no problems like this one in between P and NP, then we can safely conclude that P=NP.

References

[1] László Babai: *Graph Isomorphism in Quasipolynomial Time.* arXiv:1512.03547v2, 2017.

[2] Brendan D. McKay and Adolfo Piperno: *Practical Graph Isomoprhism, II*. arXiv:1301.1493, 2013.

[3] Adolfo Piperno: *Search Space Contraction in Canonical Labeling of Graphs*. arXiv:0804.4881, 2008, v2 2011.

[4] Babai, L., Kantor, W. M. and Luks, E. M. 1983. *Computational complexity and the classification of finite simple groups*. In: Proceedings of the 24th Annual Symposium on the Foundations of Computer Science, 162–171.

[5] Harary, Frank, 1921-2005. *Graph Theory*. New York : Addison-Wesley, 1969.

[6] Hofstadter, Douglas R., 1945-. *Gödel, Escher, Bach : An Eternal Golden Braid*. New York :Basic Books, 1979.

[7] Weisstein, Eric W. "Hoffman Graph." From MathWorld--A Wolfram Web Resource

[8] West, Douglas B. *Introduction to Graph Theory*. 2nd Ed., New York : Pearson, 2001.

Appendix

```
import java.util.Arrays;
import java.util.LinkedList;

Public static void CoherentConfig(int[][] edges) {
	//Input: adjacency matrix
	//Output: coherent configuration
	int n=edges.length;
	 int[][] colors= new int [n][n];

	 int C=n*n+1; //max number of colors
	 int[][][] lists= new int [n][n][n+1];

	 for(int i=0;i<n;i++){
		for(int j=0;j<n;j++){
			colors[i][j]=edges[i][j]+1;//force nonzero
		}
		colors[i][i]=3;
	 }


	 int iter=0;
	 boolean Done=false;
	 while(!Done&&iter<1000){ //10 iterations is excessive, 1000 is a fair upper bound
		System.out.println(iter);
		for(int i=0;i<n;i++){
			for(int j=0;j<n;j++){
				System.out.print((char)(colors[i][j]+64));//convert int to capital letter (+95 for lowercase +64 upper)
									//will break with many colors (+32 for max symbols ~ 200)
				System.out.print(" ");
			}
			System.out.println();
		}
		Done=true;
		iter++;
		for(int i=0;i<n;i++){//O(n) multiplier for each loop
			for(int j=0;j<n;j++){
				for(int k=0;k<n;k++){
					lists[i][j][k+1]=C*colors[i][k]+colors[k][j]; //O(1) work in innermost loop
					//find 'detours' note that C is just a hack to fake ordered pairs
				}
				lists[i][j][0]=colors[i][j];//this will stay first after sorting as each color is >0
				Arrays.sort(lists[i][j]);//sort each location's list. This is O(n log n) complexity
			}
		}//Total complexity is O(n)*O(n)*O(n log n)= O(n^3 log n)
		//now how many distinct lists do we have?
		LinkedList<int[]> types=new LinkedList<int[]>();
		types.add(lists[0][0]);

		for(int i=0;i<n;i++){
			for(int j=0;j<n;j++){
				int temp=colors[i][j];
				colors[i][j]=NodupeInsert(types,lists[i][j])+1;//force nonzero
				if(temp!=colors[i][j]){
					Done=false;
				}
```

```
                        }
                    }
                    if(Done){//when done print out the detour lists
                        System.out.println();
                        for(int i=0;i<types.size();i++){
                            int[] l=types.get(i);
                            for(int j=0;j<n+1;j++){
                                System.out.print((char)((l[j]/C)+64) +""+ (char)((l[j]%C)+64) + "  " );
                            }
                            System.out.println();
                        }
                        System.out.println();
                    }

            }
            System.out.println("Coherent!");
    }

    public static int NodupeInsert(LinkedList<int[]> types, int[] list){//helper function
            //assigns colors to detour lists
             int index;
             for(index=0;index<types.size();index++){
                    if(Arrays.equals(list,types.get(index)))
                        return index;
             }
             types.add(list);

             return index;
    }


////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//examples:


            int[][] hexatriag={     {0,1,1,1,1,0,0,0,0},
                                    {1,0,1,0,0,1,1,0,0},
                                    {1,1,0,0,0,0,0,1,1},
                                    {1,0,0,0,1,0,1,0,1},
                                    {1,0,0,1,0,1,0,1,0},
                                    {0,1,0,0,1,0,1,0,1},
                                    {0,1,0,1,0,1,0,1,0},
                                    {0,0,1,0,1,0,1,0,1},
                                    {0,0,1,1,0,1,0,1,0}, };//hexatriagonal example

             int[][] hoffman={
                                     {0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0},
                                     {0,0,0,0,0,0,0,0,1,1,1,0,1,0,0,0},
                                     {0,0,0,0,0,0,0,0,1,0,0,1,0,1,1,0},
                                     {0,0,0,0,0,0,0,0,0,1,0,1,0,1,0,1},
                                     {0,0,0,0,0,0,0,0,0,0,1,1,0,0,1,1},
                                     {0,0,0,0,0,0,0,0,1,0,0,0,1,1,1,0},
                                     {0,0,0,0,0,0,0,0,0,1,0,0,1,1,0,1},
                                     {0,0,0,0,0,0,0,0,0,0,1,0,1,0,1,1},
                                     {1,1,1,0,0,1,0,0,0,0,0,0,0,0,0,0},
                                     {1,1,0,1,0,0,1,0,0,0,0,0,0,0,0,0},
                                     {1,1,0,0,1,0,0,1,0,0,0,0,0,0,0,0},
```

```
                    {1,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0},
                    {0,1,0,0,0,1,1,1,0,0,0,0,0,0,0,0},
                    {0,0,1,1,0,1,1,0,0,0,0,0,0,0,0,0},
                    {0,0,1,0,1,1,0,1,0,0,0,0,0,0,0,0},
                    {0,0,0,1,1,0,1,1,0,0,0,0,0,0,0,0},
                    };

int[][] tessaract={
                    {0,1,0,1,1,0,0,0,0,0,0,0,1,0,0,0},
                    {1,0,1,0,0,1,0,0,0,0,0,0,0,1,0,0},
                    {0,1,0,1,0,0,1,0,0,0,0,0,0,0,1,0},
                    {1,0,1,0,0,0,0,1,0,0,0,0,0,0,0,1},
                    {1,0,0,0,0,1,0,1,1,0,0,0,0,0,0,0},
                    {0,1,0,0,1,0,1,0,0,1,0,0,0,0,0,0},
                    {0,0,1,0,0,1,0,1,0,0,1,0,0,0,0,0},
                    {0,0,0,1,1,0,1,0,0,0,0,1,0,0,0,0},
                    {0,0,0,0,1,0,0,0,0,1,0,1,1,0,0,0},
                    {0,0,0,0,0,1,0,0,1,0,1,0,0,1,0,0},
                    {0,0,0,0,0,0,1,0,0,1,0,1,0,0,1,0},
                    {0,0,0,0,0,0,0,1,1,0,1,0,0,0,0,1},
                    {1,0,0,0,0,0,0,0,1,0,0,0,0,1,0,1},
                    {0,1,0,0,0,0,0,0,0,1,0,0,1,0,1,0},
                    {0,0,1,0,0,0,0,0,0,0,1,0,0,1,0,1},
                    {0,0,0,1,0,0,0,0,0,0,0,1,1,0,1,0},
                    };


int[][] prism={                     {0,1,1,1,0,0},
                                    {1,0,1,0,1,0},
                                    {1,1,0,0,0,1},
                                    {1,0,0,0,1,1},
                                    {0,1,0,1,0,1},
                                    {0,0,1,1,1,0},}; //prism example


int[][] hazard={                    {0,1,1,1,1,1,1},
                                    {1,0,1,0,0,0,0},
                                    {1,1,0,0,0,0,0},
                                    {1,0,0,0,1,0,0},
                                    {1,0,0,1,0,0,0},
                                    {1,0,0,0,0,0,1},
                                    {1,0,0,0,0,1,0}, }; //hazard example

int[][] halin={                     {0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                                    {1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                                    {1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                                    {0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                                    {0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
                                    {0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                                    {0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                                    {1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                                    {0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
                                    {0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                                    {0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0},
                                    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0},
                                    {0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0},
                                    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0},
```

```
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1},
{0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0},}; //no symmetries

int[][] k33={       {0,0,0,1,1,1},
                    {0,0,0,1,1,1},
                    {0,0,0,1,1,1},
                    {1,1,1,0,0,0},
                    {1,1,1,0,0,0},
                    {1,1,1,0,0,0}, };// K3,3
```