A Brief Dive into Graph Isomorphism

Kevin Richard Koch

May 12, 2019

Abstract

In this paper I will explain my exploration of graph isomorphism as a problem in computer science, as well as an algorithm I discovered written by Ullmann in 1976. The algorithm explores a subset of the graph isomorphism problem, sub graph isomorphism. I will also go into some of the real world applications of Graph Isomorphism, as well as the history and future for the topic.

What is graph isomorphism

Graph Isomorphism, here after refereed to as GI, is a characteristic of a graph to be able to be rewritten in different ways, while still maintaining the same information.
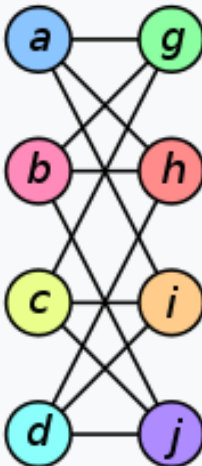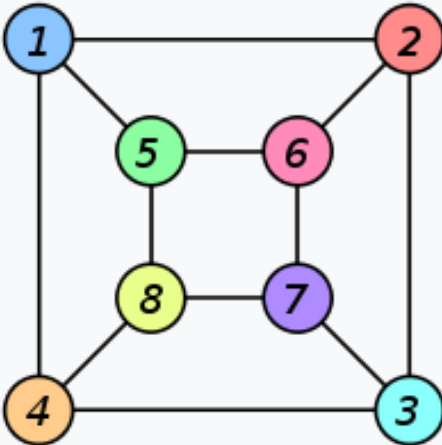

Figure 1: An example of an isomorphic graph

As you can see from the two graphs, even though they look very dissimilar, they are infarct the same graph. This property is extremely useful in many fields today. Chemists are able to look at the molecular makeup of different substances, and use GI to identify known atomic structures. This aspect of GI is so researched in fact that it is known as the "Chemist's Problem".

I should also take a moment to address what exactly is the Graph Isomorphism problem. Suppose that ω^1 and ω^2 are two graphs with n vertices, the GI problem asks if they are isomorphic or not. GI plays an important role in Computer Theory because it lies in the class known as NP, and is rated NP-Hard. It is unknown if it lies in P or NP-Complete, however most heuristic evidence stronly implies that it is not NP-Complete. Further evidence of this is the fact that if it was proven that this is NP-Complete, then the polynomial hierarchy would collapse. GI is known to be polynomial complete for graphs of bounded degree, bounded genus, bounded eigenvalue mutiplicity, and treewidth. The problem of counting the number of isomorphisms between two graphs is Turing reducible to the inversion of the graph itself. It is known that it has a time complexity of at most $\exp(O(n^{2/3}))$ for a graph with V(n). Also there are plenty of general algorithms for GI wheich are "practical in practice" meaning there are some worst case inputs which cause it to have a horrendius run time, however for most it is fine. [1, 2]

Now before we continue, I should define some terminology so it is known when referenced later on in the paper. A graph through this paper shall be known as an *combinatorial graph*. So, a graph G is a finite set of *vertices* plus a set of ordered or unordered pairs of vertices named *edges*. A set of vertices shall be noted as *V(G)* while a set of edges will be noted as *E(G)*. Graphs containing ordered pairs shall be known as *directed* and ones with unordered pairs shall be known as *undirected*. Graphs shall be assumed to be undirected unless otherwise noted. The number of edges of a vertex shall be known as a *valence* of the vertex. For a vertex, the number of head ends adjacent to a vertex is called the *indegree* of the vertex and the number of tail ends adjacent to a vertex is its *outdegree*. (How many nodes point to it, and how many point out.) A *connected component* is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph. A *loop* is an edge that connects a vertex to itself. And finally a *parallel edge* are two or more edges which are incident to the same two vertices.

Properties of Graph Isomorphism

For a graph to be isomorphic, it must fulfill several criteria. Not every graph that looks similar are isomorphic, one key feature of an isomorphic graph is that $V^1(G) = V^2(G)$, and $E^1(G) = E^1(G)$. There must exist a one to one mapping between the graph's vertices, and the following properties are satisfied, then the two graphs are said to be isomorphic.

1. $V^1(G) = V^2(G)$

2. $E^1(G) = E^1(G)$

3. The indegree and outdegree should be equal

4. The number of connected components should be the same

5. The number of loops should be the same

6. The number of parallel edges should be the same

The Chemist's problem

This problem is actually identifying what molecule is what. Say you're given the molecule for water, which is commonly known as H_2O. You would have to analyze different structures and try and match it with water. So efficient graph isomorphism algorithms are super convenient here, since molecules can be made into a graph form where the bonds would be the edges, and the vertexes would be the different atoms that make up the compound. [3]

Miller then goes on to describe a process where he determines certificates for different equivalence relations. A certificate is defined as a graph that has an isomorphism, and a deterministic certificate if f is a certificate and is computable in polynomial time. Further, a succinct certificate is a certificate which is computable in NP time. This brings us to an open question presented in the article of what is the relation of the following four properties other than 1 implies 2 implies 4 and 1 implies 3 implies 4:

1. $<A, \equiv>$ has deterministic certificates.

2. equivalence of A over $\equiv$ is in P.

3. $<A, \equiv>$ has succinct certificates

4. equivalence of A over $\equiv$ is in $NP \cap \overline{NP}$

(where $\equiv$ is an equivalence relation over a set A).

It is not known (as of this paper)if graph isomorphism satisfies any of the above four conditions. Sine polynomial time reduciability preserves all of the conditions, a positive solution for graph isomorphism would imply a positive solution for structures. In fact, graph isomorphism isn't known to satisfy any of the above conditions. This again leads us back to the discussion of isomorphism was found to be polynomial reducible, all the classes would fall apart. [3]

The Algorithms

In researching graph isomorphism, I discovered that there is a multitude of different facets of the graph isomorphism problem, and I was overwhelmed with the plethora of options out there for me to research, so I ended up taking a bit of a shotgun approach, studying graph isomorphism as a broad topic, then diving deeper into some of the sub problems that come as a result of graph isomorphism. I read in one of the papers that graph isomorphism gained so much popularity within the computer science fields when it was first being researched that it was even referred to as the "four-color disease". [4]

So my research eventually lead me to the following brute force algorithm. From the paper I read on the topic, it looks like they made an incorrect note in their pseudocode, and as such I have fixed that for this paper.

So a brute force algorithm for graph isomorphism, is exponential in the amount of time it takes to process. The following algorithm has a run time of $O(N^2 - N!)$ where n is the number of vertices in the two graphs we test on. The procedure is simple, enumerate and test all permutations of the vertices of one of the two graphs(in this case graph 2), until an isomorphism mapping is found. If it exhausts every possible permutation then the two graphs are not isomorphic. Notes for the following algorithms, the boolean array 'used' marks which vertices have been mapped. The array perm records the vertex mapping. For all $V \in$ G1, perm[x] contains the label of the mapped vertexes in G2. [5]

Algorithm 1 BruteForceWrap

1: **procedure** BRUTEFORCEWRAP
2:      isomorphic = BruteForce(N-1, G)   ▷ calls upon brute force to find if isomorphic is true or false
3:      **if** $isomorphic = TRUE$ **then**
4:          write "Graphs are Isomorphic."
5:      **else**
6:          write "Graphs are not Isomorphic."

Algorithm 2 Brute Force

1: **procedure** BRUTEFORCE(*level*)
2:     result = FALSE
3:     **if** *level* = −1 **then**
4:         result = CheckEdges(*G*)
5:     **else**
6:         i = 0
7:         **while** $i < N$ & $result = FALSE$ **do**   ▷ Runs until you run out of vertexes or you find a match
8:             **if** $used[i] = FALSE$ **then**
9:                 used[i] = TRUE   ▷ keeps track of what nodes you've visited during recursion
10:                 perm[level] = i
11:                 result = BruteForce(*level* − 1)
12:                 used[i] = FALSE   ▷ Resets state after we are done with it
13:             i = i + 1
14:     return result

Algorithm 3 Check Edges

1: **procedure** CHECKEDGES
2:     diff = TRUE
3:     **for** $x = 0$ to $N − 1$ **do**
4:         y = 0
5:         **while** $y < N$ & $diff = TRUE$ **do**
6:             **if** $adj\_matrix1[x][y] \neq adj\_matrix2[perm[x]][perm[y]]$ **then**
7:                 diff = TRUE
8:             y = y + 1
9:     return diff

Bibliography

[1] Harm Derksen. The graph isomorphism problem and approximate categories. *Journal of Symbolic Computation*, 59:81 – 112, 2013.

[2] Stephen G. Hartke and Andrew Radcliffe. Mckays canonical graph labeling algorithm. *Contemporary Mathematics book series*, 479, 02 2013.

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

[4] Ronald C. Read and Derek G. Corneil. The graph isomorphism disease. *Journal of Graph Theory*, 1(4):339–363, 1977.

[5] Robert W.; Tolley, Tracy R.; Franceschini and Mikel D. Petty. "graph isomorphism algorithms: Investigation of the graph isomorphism problem". Technical report, Institute for Simulation and Training., 1995.