Graph Isomorphisms

Mitchell Krystofiak, Travis Gopaul

12 May 2021

Abstract

This project studies the Graph Isomorphism problem, which aims to detect isomorphisms between graphs and currently has no known algorithms that run in polynomial time. The analysis below attempts to create a similar algorithm by testing if a graph is a permutation of another graph. Base cases rule out obvious non-isomorphic graphs and spends most of it's time permuting the adjacency lists of vertices in each graph.

Keywords: graph isomorphism, polynomial, permutation, adjacency list

1 Introduction

An isomorphism between two graphs is a bijection between the sets of vertices that preserves adjacency. Let G and H be graphs and the vertices of each graph be denoted as V. We define a bijection $f : V(G) \to V(H)$ such that for any two vertices u and v of G, if they are adjacent in G, the graphs are isomorphic if and only if $f(u)$ and $f(v)$ are adjacent in H (McKay and Piperno, 2004).

In the theoretical study of graph isomorphisms, the problem is that it is not yet possible to detect these isomorphisms in polynomial time. There have been many attempts and published claims, yet the problem remains NP-incomplete. The fastest proven running time, which has stood for three decades is $e^{O(\sqrt{nlogn})}$ (McKay and Piperno, 2004). From the many attempts, we have the nauty algorithm, which used automorphisms to prune the search, and saucy, which reimplemented the automorphism group subset of nauty using sparse data structures. This gave saucy the ability to detect some types of automorphisms (McKay and Piperno, 2004). Automorphisms are special isomorphisms of graphs where the permutation of edges remains the same. These algorithms use a method known as canonical labelling, which places vertex labels in a manner that does not depend on where they were before. After the canonical labeling operation, two graphs that are isomorphic become identical (McKay and Piperno, 2013).

2 Algorithm

The algorithm we have developed involves similar methods to the nauty, saucy and Traces algorithms and uses the canonical labelling operation. When graphs are put through the isomorphism checker, they are first put through a serires of base cases. In the graph structure, there are variables to record the number of vertices and edges. The isomorphism checker also checks the degree sequences of each graph. The degree sequences are a list of the number of edges attached to each vertex. For example, if a graph G with vertices $V = [1, 2, 3]$, if 1 has 1 edge, 2 has 2 edges, and 3 has 1 edge, then the degree sequence of G would be $[1, 1, 2]$. Using the fact that graphs are only isomorphic if the degree sequences are equal and the vertex and edge counts are equal, the code below tries to eliminate obvious non-isomorphic graphs before checking for permuations between the vertex sets:

```
bool is-iso(Graph H)
    iso = false
    degG = degree sequence of G
    degH = degree sequence of H
    if (G.vertexCount != H.vertexCount)
        iso = false
    else if (G.edgeCount != H.edgeCount)
        iso = false
    else if (degG != degH)
        iso = false
```

The degree sequences are handled by separate member functions, where one is used to return a vector for comparisons and one is used to return a mapping from the vertices and their number of adjacent edges. To compare them, another member function reads in the degree sequence vectors from two graphs:

```
vector degreeSeqVect(int sort)
    vector degree
    int count = 0
    for i in G.vertices
        count = 0
        for j in i.second
            count++
        degree.push-back(count)
    if sort == 1
        sort(degree)
    return degree

map degreeSeqVect()
    map degree
    int count = 0
```

```
    int j = 0
    for i in G.vertices
        count = 0
        for j in i.second
            count++
        degree.insert(i,count)
    return degree

bool degreeSeqComp(vector a, vector b)
    if a.size != b.size
        return false
    for i in a
        if a[i] != b[i]
            return false
    return true
```

After the two graphs pass the base cases, the actual permutation function begins. First, the graphs are canonically labeled, as mentioned above using the algorithm:

```
map zeroMap()
    map degree
    bool force-break = false
    int j = 0
    vector list
    for i in G.vertices
        list.push-back(i.first)
        degree.insert(j,i-¿second)
        j++
    for i in degree
        j = 0
        for k in i.second
            j = 0
            while(*k != list[j])
                j++
                if (j == list.size)
                    force-break = true
                if force-break == true
                    print "Error!"
                *k = j
    return degree
```

Now that we have our graphs canonically labeled, we can permute on the two graphs. Based on our above theory, the two graphs, if they are isomorphic, they should be equal. Below, the last part of the is-iso method calls the permutation method:

```
is-iso(Graph H)
    ...
    else
        map mapG = G.zeroMap()
        map mapH = H.zeroMap()
        map a = degreeSeqMap(mapG)
        map b = degreeSeqMap(mapH)
        if ((iso = permutation(a, b, mapG, mapH)
            print "Isomorphic!"
            print "Not isomorphic!"
    return iso

bool permutation(map a, map b, map X, map Y)
    bool perm = false
    int count = 0
    vector found
    for i in a
        perm = false
        while (!perm)
            for j in b
                vector iterator it
                it = find(found, j.first)
                if (i.first = j.first; i.second == i.second; it ∉ found)
                    tempG = X[i.first]
                    tempH = Y[j.first]
                    perm = is-permutation(tempG, tempH)
                    if (perm)
                        count++
                        found.push-back(j.first)
            perm = true
    return ((count == a.size);(count == found.size)) ? true: false
```

After going through the long cycle associated with finding isomorphisms and ruling out non-isomorphisms, the end of the permutation will ultimately decide if two graphs are isomorphic by means of canonical labeling. The algorithm is not as efficient as the one mentioned above, however it is in some variation of $O(n!)$ time.

3 Conclusion

Upon the completion of our algorithm, it was clear to see how complex the graph isomorphism problem truly is. We examined the approaches used in the nauty algorithms and its derivatives and found that while our approach was

similar in nature, our solution took little regard for time complexity given the abstractions needed to simply solve the problem.

The ability to determine if an isomorphism exists between graphs has important implications in various scientific fields due to the heavy use of graphs to represent data. In mathematical chemistry, this technique can be used to determine molecular symmetry between graphs of chemical compounds. Graph isomorphism can be used in circuit design in electronic design automation. Graph drawing computer vision are also fields of research where isomorphism can be applied to compare image symmetry (Liotta et al., 2006). The graphs in these fields are significantly larger than the scope of what our program is designed to handle and makes evident the importance of an efficient graph isomorphism detection algorithm.

4 References

Giuseppe Liotta, Roberto Tamassia, and Ioannis G Tollis. (2006). Graph Algorithms And Applications 5. World Scientific.

Brendan D. McKay, Adolfo Piperno. (2004). Practical graph isomorphism, II. Journal of Symbolic Computation, Volume 60. 94-112.
https://doi.org/10.1016/j.jsc.2013.09.003.

Brendan D. McKay, Adolfo Piperno.(2013) nauty and Traces User's Guide (Version 2.5).
http://users.cecs.anu.edu.au/ bdm/nauty/nug25.pdf

5 Appendix

All reference code, including the graph class structure, it's components and methods, are including in the Project3.zip file attached to this pdf. All important code has been pseudoed in the algorithm section of this paper. In addition, all of this code and a detailed README file can be found on github at this link: https://github.com/TWGopaul/GraphIsomorphism.