Project 3 - Graph Isomorphisms

Owen Dennis
Spring 2020
COSC 320 - 002
5/11/2020

Abstract

For this project, I decided to construct two algorithms. The first was implemented to test whether or not two graphs are isomorphic to one another via some mapping. The second tests whether or not there exists some number of subgraphs of one graph that are isomorphic to another graph. These functions are useful in many applicable areas concerning comparisons, such as identifying and matching biochemical structures in hopes of discovering evolutionary mechanisms in molecular networks.

Introduction

The term *isomorphism* stems from the Ancient Greek words *isos*, meaning "equal", and *morphe*, meaning "form" or "shape". In other words, it is used mainly in mathematics to describe the equality of two shapes. The key word her is shape, because what remains different are the values that the elements of the structure hold. To show this equality, there needs to be a way to transform one value from the first structure to a value in the second while still preserving the first structure's correlation between values. If there is a way, or mapping, to do this, then the two structures are said to be isomorphic to one another.

In this project we will focus specifically on how to define and test the isomorphism, or lack there of, between two graphs. To explain this mathematically, we must first describe the notation we will use. We will denote our first graph as $G_1 = (V_1, E_1)$ and our second as $G_2 = (V_2, E_2)$, where V_i and E_i are the sets of vertices and edges of G_i, respectively. The set of edges E_i is the collection of all edges in G_i, each denoted as (u_i, v_i), where $u_i, v_i \in V_i$.

We must now define a function $\phi : (V_1 \cup V_2) \to K$, for some arbitrary set K, as the *node label function*[3]. In terms of the operations of this function, two vertices u and v are said to be equal if $\phi(u) = \phi(v)$, meaning they get mapped to the same value. Thus, two graphs G_1 and G_2 are said to be isomorphic[3] if there exists a bijective mapping $m : V_1 \to V_2$ such that

$$\forall v \in V_1, \phi(v) = \phi(m(v)) \text{ and} \tag{1}$$

$$\forall u, v \in V_1, (u, v) \in E_1 \iff (m(u), m(v)) \in E_2 \tag{2}$$

Now, the second aspect of this project deals with subgraph isomorphism, which is very similar to isomorphisms. We will use the same notation and node label function as before for this definition. One graph G_1 is said to be an *induced subgraph isomorphism*[3] of G_2 if there exists an injective mapping $m : V_1 \to V_2$ such that

$$\forall v \in V_1, \phi(v) = \phi(m(v)) \text{ and } \tag{3}$$

$$\forall u, v \in V_1, (u, v) \in E_1 \implies (m(u), m(v)) \in E_2 \tag{4}$$

It is easy to see the small differences between the two definitions. With respect to being an isomorphism, to be and induced subgraph isomorphism, the mapping doesn't need to be surjective and the edge correlation property on (4) is only one way whereas (2) is an if and only if statement.

In addition, notice that the definition we are using is for a induced subgraph isomorphism, which is a specific type of subgraph isomorphism. In our case, we are defining a subgraph $G' = (V', E')$ of a graph $G = (V, E)$ as a graph where $V' \subset V$ and $E' \subset E$. The set $E - E'$ consists of elements (u, v), where $u, v \in V - V'$. We will be using these notations and definitions for the rest of our paper.

Topic Details

The Graph Isomorphism Problem

We will be tackling two problems in the computer science community: the Graph Isomorphism Problem and the Induced Subgraph Isomorphism problem. The Graph Isomorphism Problem says that first, for two graphs G_1 and G_2, $|V_1| = |V_2|$ and $|E_1| = |E_2|$. Since the isomorphic mapping is a bijection, the number of elements in both sets must be equal. The Graph Isomorphism Problem asks the following question: "Determine whether there exists a bijective edge-invariant vertex mapping (isomorphism) $f : V_1 \to V_2$".[2] This is basically the definition of a graph isomorphism.

Using C++, we have implemented a program called "isIsomorphic" to take an object of the Graph class and then, using another graph as a parameter, returning a Boolean value with regards to whether the two graphs are isomorphic or not. If they are, the program would provide a mapping from the first to the second graph's vertices, and therefore answering the Graph Isomorphism Problem's question. The pseudo-code for the function follows:

```
bool isIsomorphic(Graph $G_1$, Graph $G_2$)
1       if $G_1$.vertices.size $\neq$ $G_2$.vertices.size
```

```
        **return** false
    **if** $G_1$.numEdges $\neq$ $G_2$.numEdges
        **return** false
    **declare** vector perm
    **for** itr = $G_2$.vertices.begin to $G_2$.vertices.end
        perm.push_back(itr$\rightarrow$first)
    **sort** perm
    **do**
        **declare** map $m$
        index = 0
        **for** itr = $G_1$.vertices.begin to $G_1$.vertices.end
            $m$.insert(itr$\rightarrow$first,perm[index])
            index++
        **declare** Graph temp
        **for** itr = $G_2$.vertices.begin to $G_2$.vertices.end
            temp.addVertex(itr$\rightarrow$first)
        **for** itr = $G_1$.vertices.begin to $G_1$.vertices.end
            itr2 = $m$.find(itr$\rightarrow$first)
            **for** $i$ = 0 to itr$\rightarrow$second.size
                itr3 = $m$.find(itr$\rightarrow$second[$i$])
                temp.addEdge(itr2$\rightarrow$second,itr3$\rightarrow$second)
        **if** temp == $G_2$
            print "Mapping:"
            **for** itr = $m$.begin to $m$.end
                print i$\rightarrow$first + " " + i$\rightarrow$second
            **return** true
    **while** next_permutation(perm.begin,perm.end)
    **return** false
```

The function above first makes sure the number of vertices and edges between the two graphs are equal. It then creates a vector to hold all of the vertices of G_2 to permeate through after sorting it. Then, the function loops through every permutation of the the vertices of G_2. In each loop, there is a map variable that defines a new mapping dependent upon the loop's permutation. After the mapping is created, a temporary graph is made to mimic the structure of G_1, but with the vertices change depending on the loop's mapping. After the temporary graph is created via the mapping, the function checks to see if it is equal to G_2. If it is, we have found our mapping and can safely say that the G_1 and G_2 are isomorphic to each other. If no map is found , then the final result is false. In other words, zero mappings exist where G_1 and G_2 are isomorphic.

Although this algorithm solves the problem, its complexity is far from efficient because it has to run through every permutation of the vertices in G_1. To clarify, lines 1 through 5 run in constant time, lines 6 and 7 are $O(|V_2|)$, line 8 is in $O(|V_2| \cdot log|V_2|)$, and the loop of lines 9 to 28 run through every permutation of V_2, which takes $O((|V_2|)!)$ time. Within the loop, lines 10, 11, and 15 are all in constant time, lines 12, 13, and 14 are in $O(|V_1|)$, lines 16 and 17 are in $O(|V_2|)$, and lines 18 through 22 are in $O(|V_1| + |E_2|)$ because it has to run through every vertex and edge to create the temporary graph. In total, the run time of this function is:

$$\begin{aligned} &O(2 + |V_2| + |V_2| \cdot log|V_2| + (|V_2|)! \cdot (3 + |V_1| + |V_2| + |V_1| + |E_2|)) && (5)\\ &= O(2 + |V_2| \cdot (2 + log|V_2|) + (|V_2|)! \cdot (3 + 3|V_1| + |E_2|)) && (6)\\ &= O((|V_1|)! \cdot (|V_1| + |E_1|)) && (7)\end{aligned}$$

Thus, the time complexity of this function is $O((|V_1|)! \cdot (|V_1| + |E_1|))$.

As of right now, "the Graph Isomorphism Problem is neither known to be in P nor known to be NP-complete; instead, it seems to hover between the two categories."[4] However, there have been recent breakthroughs on moving closer to P than ever before. In 2015, László Babai, a professor at the University of Chicago, published a paper saying that that he has an algorithm "outside [the] borders [of P], in the suburbs rather than the central city".[4] He said to not expect a proof of the Graph Isomorphism Problem in P time anytime soon, but believes we are on our way to getting there. In fact, most computer scientists believe that the problem is P while only some believe it to be NP-complete.[4]

The Induced Subgraph Isomorphism Problem

As said before, the second function we implemented was for the Induced Subgraph Isomorphism Problem. This problem first says that given two graphs G_1 and G_2, $|V_1| \leq |V_2|$ and $|E_1| \leq |E_2|$. For this project, we did not ask for $|V_1| \leq |V_2|$ and $|E_1| \leq |E_2|$, instead $|V_1| < |V_2|$ and $|E_1| < |E_2|$. This is because we have already dealt with graph isomorphisms and are looking for only induced subgraph isomorphisms. The Induced Subgraph Isomorphism Problem asks the following question: "Find an edge-invariant injective function $f : V_1 \rightarrow V_2$."[2] This is basically the definition of an induced subgraph isomorphism.

Using C++, we have implemented a program called "isSubIsomorphic" to take an object of the Graph class and then, using another graph structure with less vertices and edges as a parameter, return a Boolean value with regards to whether the graph has one or more subgraphs isomorphic to the second one or not. If there is at least one, the program would provide a mapping for each one, setting the vertices of each subgraph to the vertices of the second graph's vertices, and thus answering the Induced Subgraph Isomorphism Problem.

However, before we get into that function, we need to introduce another helper function that the algorithm uses. This function is recursive and is called "GetSubsets" and, given the vertices of G and an integer n, returns a 2D vector of integers of all possible subsets of V of the size n. This function is used to return all possible vertex subvectors of the first graph in the "isSubIsomorphic" function. The pseudo-code for the function follows:

```
**void** getSubsets(vector<vector<int>>& subsets, int $n$, int start, int $l$, bool* used)
    **if** $l == n$
        **declare** vector $s$
        **declare** int $i = 0$, $m = 0$
        **for** itr = $G$.vertices.begin to $G$.vertices.end
            **if** used[$i$++]
                **if** $m$++ $< n$
                    $s$.push_back(itr→first)
        subsets.push_back($s$)
    **else if** start == $G$.vertices.size
        **return**
    **else**
        used[start] = true
        getSubsets(subsets,$n$,start,$l$+1,$used$)
        used[start] = false
        getSubsets(subsets,$n$,start+1,$l$,$used$)
```

The function above takes in, by reference, the 2D vector of integers, a Boolean array of all trues, and several integers for recursion. The recursive step creates all permutations of a specific size of the vertices of G by creating a subvector containing the current element and then one without the element. Once that subvector is of the right size, it is added to the list of subsets. Once all permutations of a certain size have been added, the function finishes.

The function for "isSubIsomorphic" is as follows:

```
**bool** isSubIsomorphic(Graph $G_1$, Graph $G_2$, int count)
    **if** $G_1$.vertices.size $\leq$ $G_2$.vertices.size
        **return** false
    **if** $G_1$.numEdges $\leq$ $G_2$.numEdges
```

```
4           return false
5       declare subIso = false and used[$G$.vertices.size]
7       for $i = 0$ to $G_1$.vertices.size
8           used[$i$] = true
9       declare 2D vector subsets
10      getSubsets(subsets, $G_2$.vertices.size, 0, 0, used)
11      for $i = 0$ to subsets.size
12          declare Graph temp
13          for $j = 0$ to subsets.size
14              temp.addVertex(subsets[$i$][$j$])
15          for itr = temp.vertices.begin to temp.vertices.end
16              itr2 = $G_1$.vertices.find(itr$\rightarrow$first)
17              for $j = 0$ to itr2$\rightarrow$second.size
18                  itr3 = temp.vertices.find(itr$\rightarrow$second[$j$])
19                  if itr3 $\neq$ temp.vertices.end
20                      temp.addEdge(itr1$\rightarrow$first,itr2$\rightarrow$second[$j$])
21          if temp.isIsomorphic($G_2$)
22              subIso = true
23              count++
24      return subIso
```

The function above makes sure the number of vertices and edged in the given graph is greater than that of the first one. Then it sets up the variables to be used in the "getSubsets" function. Once the function has been called and the subvectors of the size $|V_2|$ are created, the function runs a loop on every subvector. For each loop, a temporary graph is created with the current subvector's values as its vertices. Then, a nested for loop runs on each vertex in the new graph and looks at the edges in the first graph that contain the current vertex. If the other vertex in that edge is found in the new graph's vertex set, then that edge is added to the new graph. If that second vertex is not found, then that edge is passed over and not added. Once the graph has fully been created, the "isIsomorphic" function is run with it and the second graph in the parameters of the function. If the function comes out true, then the mapping is printed out, the returned subIso Boolean variables is set to true, and the number of subgraph isomorphisms found is incremented.

The complexity of this function is worse than even the Graph Isomorphism Problem's because it has to call the "isIsomorphic" function in each of its permutations. To clarify, lines 1 through 6 and 9 are all in constant time, line8 7 and 8 are in $O(|V_1|)$, line 10 is in $O(|V_1|C|V_2|)$, or $O(|V_1|$ choose $|V_2|)$, and the loop of lines 11 through 23 are in $O(|V_1|C|V_2|)$.

This is because there are "$|V_1|$ choose $|V_2|$" permutations that come out of calling the "getSubsets" function. Line 12 is in constant time, lines 13 and 14 are in $O(|V_1|C|V_2|)$, and lines 15 through 20 are $O(|V_2|+|E_2|$ which is an estimate if we are assuming that the temporary graph has the same number of vertices and edges as $|V_2|$. Lastly, the lines 21 though 23 are $O((|V_2|)! \cdot (|V_2|+|E_2|))$ because that is the complexity of of the "isIsomorphic" function. In total, the run time of this function is:

$$\begin{aligned} &O(4+|V_1|+2|V_1|C|V_2| \cdot (1+|V_1|C|V_2|+(|V_2|+|E_2|)+((|V_1|)! \cdot (|V_1|+|E_1|)) &(8)\\ &=O(2|V_1|C|V_2| \cdot (|V_1|C|V_2|+((|V_1|)! \cdot (|V_1|+|E_1|)) &(9)\\ &=O(|V_1|C|V_2| \cdot (|V_1|)! \cdot (|V_1|+|E_1|)) &(10)\end{aligned}$$

Thus, the time complexity of this function is $O(|V_1|C|V_2| \cdot (|V_1|)! \cdot (|V_1|+|E_1|))$.

A Real-World Application

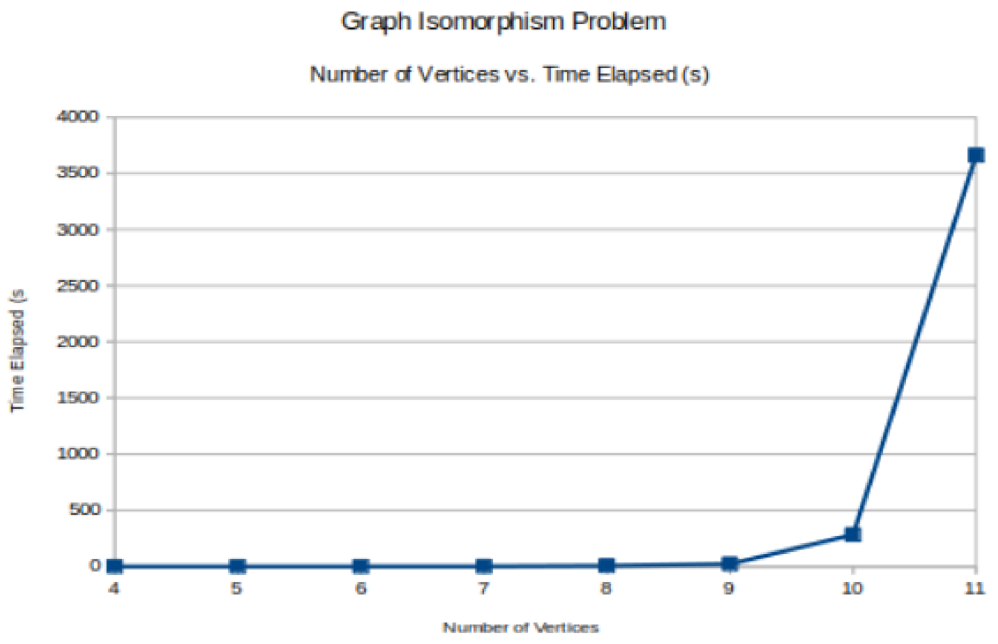
Subgraph isomorphisms might even have better and more abundant uses in everyday life than graph isomorphisms. There is one application of subgraphs that is very useful in interesting, and that is its use of identifying and matching biochemical structures. "Subgraph isomorphism algorithms are intensively used by biochemical tools."[1] In fact, you can look at a biochemical structure as a graph consisting of molecular components (the vertices) and the relationship between them (the edges).

With a biochemical network as the focus, scientists try to identify, order, and classify motifs, or the "simple building blocks of complex networks".[1] These motifs give insight into the evolutionary methods of mechanisms (such as proteins). To obtain motifs from a network, scientists have to look at every possible subgraph of the network, classify each type of subgraph they find, and sort out the subgraphs that appear in high frequency.[1] The middle step in this process is the part that would use the "isSubIsomorphic" function, except other, more efficient, algorithms have been implemented to solve the same problem.

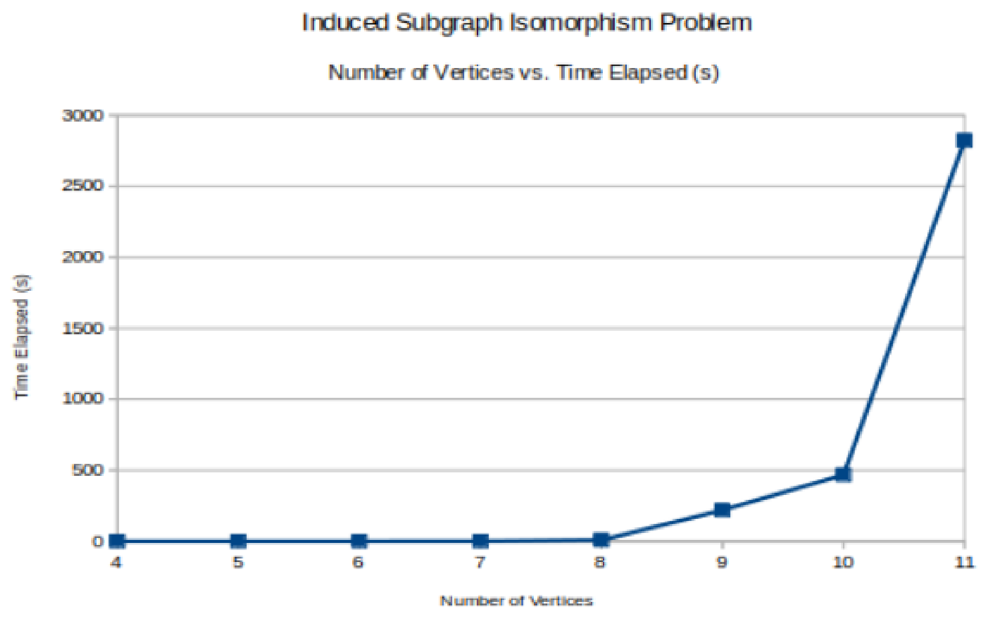
An example of a more efficient algorithm orders the looping of vertex permutations based on their lexicographic ordering, which is derived from how vertices compare to their surrounding neighbors. This process increases the likelihood of finding a subgraph isomorphism much earlier in the algorithm and can save it from running an unnecessary number of loops.[1]

Test Results

To physically show the growth of the two functions, we took many tests and documented the results. The independent variable is the number of vertices and the dependent variable is the time elapsed (in seconds) when running the program. To test the full run through of the isomorphism program, we used two example graphs that were know to be not isomorphic. Also, each point is an average of three trials. The graph is listed below:



The next graph comes from our testing of the subgraph program. Again, the goal of these tests were to run every case of the isomorphism check to a false output. We also set a parameter that $|V_2| = |V_1| - 1$ for a greater chance of getting no subgraph isomorphisms. Again, each point is the average of three trials. Here is the graph:



The table that created the charts is listed in the Appendix for further, more exact, documentation. If you notice, the subgraph isomorphism algorithm actually runs faster than the graph isomorphism algorithm, which contradicts the complexity we described earlier. We believe this is a result of the subgraph creation, where some graphs have a different number of edges than the second graph in the function. If this is the case, the algorithm runs in constant time by failing the second condition. Thus, we have come to the conclusion that the complexities we found earlier were only in the worst case. It is hard to create examples where the isomorphism function runs fully every single time, but if you were to check the tables, you can initially see the faster growing subgraph function. The first seven trials show a fast growing subgraph algorithm, but the graph isomorphism runs slower in the last trial, where there are more chances for the subgraph function to "cut corners".

Conclusion

In conclusion, while the Subgraph Isomorphism Problem is NP-complete, the Graph Isomorphism Problem is neither P nor NP-complete. Computer scientists have continued to discover algorithms that solve the problem in almost P, but have not found one yet.

In this project, the algorithm created to solve the Graph Isomorphism Problem was in time $O((|V_1|)! \cdot (|V_1| + |E_1|))$, which is an awful complexity. However, the goal of this project was to solve the problem in any way possible, and that is what we did. On the other hand, the algorithm created to solve the Induced Subgraph Isomorphism Problem was $O(|V_1|C|V_2| \cdot (|V_1|)! \cdot (|V_1| + |E_1|))$, which is even worse then the prior solution. Again, the problem was solved in any way possible, which, again, was our goal.

Our example of a real-world application of subgraph isomorphisms was identifying and matching biochemical structures in hopes of discovering evolutionary mechanisms in molecular networks. The process that scientists work though to achieve this goal takes advantage of the Subgraph Isomorphism Problem in an efficient way. This highly-valuable use sheds more light on the importance of finding an algorithm in P (which seems almost inevitable). Are the obstacles computer scientists facing a part of something we have never seen before, or from a known topic but being used in the wrong way? Time will only tell...

Bibliography

1. Bonnici, Vincenzo, et al. “A Subgraph Isomorphism Algorithm and Its Application to Biochemical Data.” BMC Bioinformatics, vol. 14, no. Suppl 7, 2013, doi:10.1186/1471-2105-14-s7-s13.

2. Calude, Cristian S., et al. "QUBO Formulations for the Graph Isomorphism Problem and Related Problems." Theoretical Computer Science, vol. 701, 2017, pp. 54–69., doi:10.1016/j.tcs.2017.04.016.

3. Jüttner, Alpár, and Péter Madarasi. "VF2++—An Improved Subgraph Isomorphism Algorithm." Discrete Applied Mathematics, vol. 242, 2018, pp. 69–81., doi:10.1016/j.dam.2018.02.018.

4. Klarreich, Erica, and Quanta Magazine. "Algorithm Solves Graph Isomorphism in Record Time." Quanta Magazine, 14 Dec. 2015, www.quantamagazine.org/algorithm-solves-graph-isomorphism-in-record-time-20151214/.

Appendix

Graph Isomorphism Problem Test Results

Number of Vertices in $\|V_1\|$	**Time Elapsed (s)**
4	0.0000017492
5	0.0193298735
6	0.1414863333
7	1.34095
8	6.346767
9	21.65313
10	282.783
11	3661.04

Induced Subgraph Isomorphism Problem Test Results

Number of Vertices in $\|V_2\|$	**Time Elapsed (s)**
4	0.00005164067
5	0.00780063
6	0.05700633
7	0.45618453
8	7.84194
9	218.01967
10	467.80067
11	2822.68