

Milestone Report — Optimization of Subgraph Isomorphism Algorithm

by Nelson Batista, Max Inciong, Francesca Truncale

Senior Project II

Professor Jianting Zhang

Fall 2017

Contents

1	Introduction	1
1.1	Background	1
1.2	Motivation	1
1.3	Data Set	2
2	Ullmann Algorithm	2
2.1	Naive Approach	2
2.2	“Refine M” Procedure	2
2.3	Full Algorithm	2
3	Implementation of the Algorithm	2
3.1	Python Implementation	2
3.2	Initial C++ Implementation	3
3.3	Improved C++ Implementation	3
3.4	Optimizations through OpenMP	3
3.5	Difficulties	4
4	Performance	4
4.1	Python Implementation	4
4.2	C++ Implementation	4
5	Appendix	5
5.1	Python Code	5
5.1.1	Graph Class	5
5.1.2	Subgraph Isomorphism Algorithm	5
5.1.3	Main	5
5.2	C++ Code	5
5.2.1	Graph Class	5
5.2.2	Subgraph Isomorphism Algorithm	5
5.2.3	Main	5
5.3	C++ Code with OpenMP	5
5.3.1	Graph Class	5
5.3.2	Subgraph Isomorphism Algorithm	5

5.3.3	Main	5
References		5

1 Introduction

The goal of this project is to understand and optimize an algorithm for solving the NP-complete problem of subgraph isomorphism.

1.1 Background

An *isomorphism* between two graphs is a mapping from the vertices of one graph, say G , to another graph, say H , such that any two vertices which are adjacent in G are mapped to vertices which are adjacent in H . The subgraph isomorphism problem, in turn, is the problem of determining whether an isomorphism exists between a graph G and any of the subgraphs of a larger search graph, H . The issue that makes the subgraph isomorphism problem so difficult to solve for a particular pair of graphs is cycling through the main graph and expanding each node within the graph and checking if each of the generated subgraphs formed from each node is isomorphic to the control graph.

Fortunately, many attempts exist to solve the subgraph isomorphism problem. Among the earliest of these is the famous *Ullmann Algorithm*, proposed by Julian R. Ullmann in his 1976 paper, *An Algorithm for Subgraph Isomorphism*. He first describes a basic, naive approach to finding a mapping from the vertices of a graph to a subgraph of a larger graph. He goes on to define a “refine” procedure to dramatically reduce the number of possible mappings that must be checked.[3] These will be covered in greater detail in the next section.

The goal of our project was to take this algorithm and improve its performance. Subgraph isomorphism is an expensive problem to solve, and finding multiple possible isomorphisms from one graph to subgraphs of another is even more expensive, so even slight improvements in the algorithm’s performance is likely to lead to large performance improvements in any large-scale program which needs to use it often.

1.2 Motivation

Our reasoning for choosing this topic for our project is an apparent lack of especially efficient algorithms for solving the subgraph isomorphism problem. Solutions to the subgraph isomorphism problem are often used to detect similarities in chemical compounds,[3] which may shed light on some of their properties. If this process needs to be done, for example, on a very large database of chemical compounds, or less frequently for several very large compounds, the process may take a very long time to complete. It would be very beneficial to improve the performance of this algorithm, even if only slightly, so that these sorts of use cases can still be handled in a more reasonable amount of time.

1.3 Data Set

The primary graph used to test the algorithm and its performance is an undirected graph consisting of “friends lists” from the social media website, Facebook. Each node of the graph represents a unique user, and, if two nodes are adjacent, then their corresponding users are friends on Facebook.[1]

The graph contains 4,039 nodes and 88,234 edges. The relatively large size of the graph makes it well-suited to use for collection of performance data. As we shall see, the Ullmann algorithm takes a rather long time to complete with a graph of this size, allowing any optimizations and/or parallelizations to be readily apparent.

2 Ullmann Algorithm

2.1 Naive Approach

2.2 “Refine M” Procedure

2.3 Full Algorithm

3 Implementation of the Algorithm

3.1 Python Implementation

The

```
def find_isomorphism(graph, subgraph):  
  
    assignments = []  
    possible_assignments = [[True]*graph.n_vertices() \  
                             for i in range(subgraph.n_vertices())]  
    if search(graph, subgraph, assignments, possible_assignments):  
        return True  
    return matches
```

Listing 1: Function used to determine whether an isomorphism exists.

The main function is `find_isomorphism`, which is depicted above. The function `search` is a modified Depth First Search, which is used to find the instance of a subgraph within a larger graph.

The `search` function runs the `update_possible_assignments` function. This particular function has been the subject of refining over the course of the project. At the time of the python implementation, it was the most significant bottleneck.

This was partially due to the usage of the `has_edge` method of the graph class, shown below.

```
def has_edge(self, vert1, vert2):  
    """ Checks if edge connecting vert1 and vert2 is in the graph  
    """  
    #if adjacent, there's an edge  
    return ({vert1, vert2} in self.adjacencies)
```

Listing 2: Code for the `has_edge` method of our graph class.

This method calls the built-in `in` function of Python lists, which has a runtime complexity of $O(n)$.^[2] Thus, the `has_edge` method of the Python implementation is $O(n)$.

3.2 Initial C++ Implementation

3.3 Improved C++ Implementation

3.4 Optimizations through OpenMP

OpenMP utilizes the multiple cores of a computer's processor in order implement multithreading to run related tasks in parallel. We intend to use it where the bottlenecks in our algorithm are to improve the algorithm's performance. The default number of cores for many computers is four. We intend to place the parallelisms where there tend to be a large number iterations, specifically in loops. This means that, theoretically, the program should run four times faster than without multithreading, using our earlier assumption of four processor cores allotted to the algorithm. It is more likely that it will be approximately 3 to 3.5 times faster, since we may not be able to completely parallelize the bottlenecks as intended. Additionally, using OpenMP, as is the case with any parallelization method, comes with a certain amount of overhead in separating the work to be done. There is also the issue of load balancing: making sure that each thread is performing approximately the same amount of computation so that no thread is idling.

The most important place to place the OpenMP is in the `refine_possible_assignments` function. This function is where the majority of the algorithm's computation takes place, as it has several layers of nested loops, each iterating through vectors which may be very large, depending on the sizes of the graphs being examined. Ideally, we want to balance the loads such that each processor core does an equal amount of

work, so as to maximize the amount of parallelization done. However, while we may simply parallelize every iterative process, that might not be efficient enough. Ideally, we would further parallelize in nodes of the search tree which have many possible assignments.

3.5 Difficulties

4 Performance

4.1 Python Implementation

The Python implementation was very inefficient, but it was necessary for us to understand what we had to do for a C++ implementation. It took us between 13 and 16 seconds to determine if an isomorphism existed between a specified graph a subgraph of the Facebook graph specified earlier.

4.2 C++ Implementation

Process We wanted to understand the algorithm starting from pseudocode. First we gathered pseudocode to determine the algorithm for subgraph isomorphism. From there, we wanted to transcribe it into python, and then into C++ and from there parallelize it using threads. After using the python implementation, we decided to utilize something known as the Ullman algorithm for our C++ implementation. We have also since changed our plan to utilize CUDA and GPGPU to utilizing OpenMP

Graph Class The graph class that we ended up using is an unweighted directed graph. This means that when searching for isomorphisms, we have to take into consideration the direction of the nodes rather than simply the connections within each graph. The graphs are unweighted, which means the isomorphism function does not take into account the values contained within each of the nodes.

5 Appendix

5.1 Python Code

5.1.1 Graph Class

5.1.2 Subgraph Isomorphism Algorithm

5.1.3 Main

5.2 C++ Code

5.2.1 Graph Class

5.2.2 Subgraph Isomorphism Algorithm

5.2.3 Main

5.3 C++ Code with OpenMP

5.3.1 Graph Class

5.3.2 Subgraph Isomorphism Algorithm

5.3.3 Main

References

- [1] J. McAuley and J. Leskovec. “Learning to Discover Social Circles in Ego Networks”. In: *NIPS* (2012). URL: <https://snap.stanford.edu/data/egonets-Facebook.html> (visited on 11/26/2017).
- [2] *TimeComplexity*. URL: <https://wiki.python.org/moin/TimeComplexity> (visited on 11/26/2017).
- [3] Julian R. Ullmann. “An Algorithm for Subgraph Isomorphism”. In: *Journal of the ACM* 23.1 (Jan. 1976), pp. 31–42. DOI: 10.1145/321921.321925.