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 . . . 3

2.2 "Refine M" Procedure . . . 3

2.3 Full Algorithm . . . 3

3 Implementation of the Algorithm **3**

3.1 Python Implementation . . . 3

3.2 Initial C++ Implementation . . . 4

3.3 Improved C++ Implementation . . . 4

3.4 Optimizations through OpenMP . . . 4

3.5 Difficulties . . . 4

4 Performance **4**

4.1 Python Implementation . . . 4

4.2 C++ Implementation . . . 5

5 Conclusion **5**

6 Appendix **5**

6.1 Python Code . . . 5

6.1.1 Graph Class . . . 5

6.1.2 Subgraph Isomorphism Algorithm . . . 8

6.1.3 Main . . . 10

6.2 C++ Code . . . 15

6.2.1 Graph Class . . . 15

6.2.2 Subgraph Isomorphism Algorithm . . . 15

6.2.3 Main . . . 15

6.3 C++ Code with OpenMP . . . 15

6.3.1 Graph Class . . . 15

6.3.2 Subgraph Isomorphism Algorithm 15
6.3.3 Main 15

References **15**

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

The paper begins, as many do, with a definition of terms that will be used throughout. We have repeated them here for the sake of clarity later on. We begin by noting that the algorithm models the search for a possible mapping by having all possible mappings as nodes on a tree, on which a depth-first search is performed to find a solution.

The subgraph isomorphism problem is defined as the problem of finding all isomorphisms between a graph $G_\alpha = (V_\alpha, E_\alpha)$ and subgraphs of another graph $G_\beta = (V_\beta, E_\beta)$, with (V_α, E_α) and (V_β, E_β) being the set of vertices and edges of G_α and G_β, respectively. The number of vertices and edges of G_α and G_β, respectively, are (p_α, q_α) and (p_β, q_β).

The *adjacency matrix* for graph G_α, $[a_{ij}]$, is defined by:

$$a_{ij} = \begin{cases} 1 & \text{if } i \neq j \text{ and } i, j \text{ share an edge} \\ 0 & \text{otherwise} \end{cases}$$

The adjacency matrix for graph G_β, $[b_{ij}]$, is defined equivalently.

Notably, we have the $p_\alpha \times p_\beta$ matrix M, which is a binary matrix of assignments (mappings) from V_α to V_β. That is, if m_{ij} is 1, then vertex i in graph G_α could possibly be mapped to vertex j in G_β. Otherwise m_{ij} is 0. M has a few interesting properties as a result, which will soon be apparent.

Finally, we come to d, which is simply algorithm's current depth in the search tree. The algorithm starts at $d = 0$, and terminates when $d = p_\alpha$. The matrix M at a particular depth d is denoted as M_d, with solutions to the working instance of the subgraph isomorphism problem being the set of assignments from V_α to V_β, contained within matrices M_{p_α}.

2.1 Naive Approach

In the most basic algorithm

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 conecting 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

5 Conclusion

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.

6 Appendix

6.1 Python Code

6.1.1 Graph Class

```
"""
Graph module
only has a class
"""
class Vertex:
    """
    Vertex Class
    """
    def __init__(self, label = "", data="", visited=False):
        self.label = label
        self.data = data
        self.visited = visited


class Graph:
```

```
    """
    Graph class
    """
    def __init__(self, vertices=None, edges=None, adjacencies=None):
        if vertices is None:
            self.vertices = []
            self.n_vertices = lambda : len(self.vertices)
        else:
            self.vertices = vertices
        if edges is None:
            self.edges = []
        else:
            self.edges = edges
        if adjacencies is None:
            self.adjacencies = []
        else:
            self.adjacencies = adjacencies

        self.__update_adjacencies()

    #Private, don't want anyone calling this by accident.
    def __update_adjacencies(self):
        for edge in self.edges:
            self.adjacencies.append({edge[0], edge[1]})

    def add_vertex(self, vertex=Vertex()):
        """ Add vertex to graph
        """
        self.vertices.append(vertex)

    def remove_vertex(self, vertex):
        """ Remove vertex from graph
```

```
            along with edges which contain it
        """
        self.vertices.remove(vertex)
        # list comprehension
        # TLDR remove all elements in self.edges which contain vertex
        # we need this since edges is a list of tuples, not of single elements
        self.edges.remove([edge for edge in self.edges if vertex in edge])
        # see above
        self.adjacencies.remove([adj for adj in self.adjacencies if vertex in adj])

    def add_edge(self, source, dest, weight):
        """ Add edge connecting source and dest
        along with weight
        """
        if (source or dest) not in self.vertices:
            self.vertices.append(source)
            self.vertices.append(dest)

        self.edges.append([source, dest, weight])
        self.adjacencies.append({source, dest})

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

    # for debugging
    def print_graph(self):
        """ Prints information about the graph
        """
        print("Graph (vertices, edges, adjacencies):")
```

```
        print(self.vertices)
        print(self.edges)
        print(self.adjacencies)
```

Listing 3: Python graph class.

6.1.2 Subgraph Isomorphism Algorithm

```
""" Subgraph Isomorphism
CPU Implementation
"""

import copy as cp # for deepcopy

def search(graph, subgraph, assignments, possible_assignments):
    """ Uses DFS to find instance of subgraph within larger graph
    """

    update_possible_assignments(graph, subgraph, possible_assignments)

    i = len(assignments)

    # Make sure that every edge between assigned vertices in the subgraph
    # is also an edge in the graph.

    # this loop calls graph.has_edge once for every edge in the subgraph
    for edge in subgraph.edges:
        if edge[0] < i and edge[1] < i:
            if not graph.has_edge(assignments[edge[0]], assignments[edge[1]]):
                return False

    # If all the vertices in the subgraph are assigned, then we are done.
    if i == subgraph.n_vertices():
```

```
            return True

        for j in possible_assignments[i]:
            if j not in assignments:
                assignments.append(j)

                # Create a new set of possible assignments, where graph node j
                # is the only possibility for the assignment of subgraph node i.
                new_possible_assignments = cp.deepcopy(possible_assignments)
                new_possible_assignments[i] = [j]

                if search(graph, subgraph, assignments, new_possible_assignments):
                    return True

                assignments.pop()

            possible_assignments[i].remove(j)
            update_possible_assignments(graph, subgraph, possible_assignments)

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

def update_possible_assignments(graph, subgraph, possible_assignments):
    """ docstring to suppress the warning
    """
    any_changes = True
    while any_changes:
```

```
        any_changes = False
        for i in range(0, subgraph.n_vertices()):
            for j in possible_assignments[i]:
                for adj in subgraph.adjacencies(i):
                    match = False
                    for vert in range(0, graph.n_vertices()):
                        # graph.has_edge gets called once
                        # for every vertex in the graph
                        # for every item in the subgraph's adjacencies
                        # for every possible assignments
                        # that is a huge number of calls to has_edge
                        # which is in itself an O(n) operation
                        # definitely room for improvement.
                        if adj in possible_assignments[adj] and \
                                graph.has_edge(j, vert):
                            match = True
                    if not match:
                        possible_assignments[i].remove(j)
                        any_changes = True
```

Listing 4: Python algorithm to find isomorphism.

6.1.3 Main

```
""" Main
"""
import sys #args
import getopt
from graph import Graph
from isomorphism import find_isomorphism
import cProfile

def main(argv):
```

```
    """ Main function
    """
    argc = len(argv)

    interactive, debug = handle_args(argc, argv)

    if interactive:
        print("Enter the graph as a space-separated adjacency list of the form\
                node1 node2 [edge_weight]:")
        filename = sys.stdin
    else:
        filename = argv[1]

    graph = import_data(filename, debug)

    if debug:
        graph.print_graph()

    if interactive:
        print("Enter the subgraph using the same format:")
        sub = sys.stdin
    else:
        sub = argv[2]

    subgraph = import_data(sub, debug)

    if debug:
        subgraph.print_graph()

    cProfile.runctx('find_isomorphism(graph, subgraph)', globals(), locals())

def import_data(filename, debug):
```

```
    """ Takes graph adjacency list written in file fd
        creates and returns graph object representing
        the graph in the file
        Note that edge weights are all 0 unless otherwise specified.
    """
    if filename == sys.stdin:
        fdesc = filename
    else:
        try:
            fdesc = open(filename, 'r')
        except IOError as err:
            sys.stderr.write("Error: Could not open file '%s'. Aborting.\n%s\n" \
                    % (filename, err))
            sys.exit(2)

    if debug:
        print(fdesc)

    graph = Graph()

    if debug:
        print("Reading file")
    for line in fdesc:
        vals = line.split()
        if len(vals) == 3:
            edge_weight = int(vals[2])
        elif len(vals) > 2:
            sys.stderr.write("Error reading file '%s'. Data should be in form \
                    'node1 node2 [edge_weight]', but line was %s\n" % line)
            sys.exit(3)
        else:
            edge_weight = 0
```

```
            graph.add_edge(int(vals[0]), int(vals[1]), edge_weight)

        if filename != sys.stdin:
            fdesc.close()

        return graph

def usage(name, err):
    """ Prints usage info
    """
    usg = """usage: %s [FILE] [OPTION]
    Options and arguments:
    FILE:               Space-separated file containing graph adjacency list. \
            If edge weights are not specified, all weights will be assumed 0.
    -h, --help:         Print this help message and exit.
    -i, --interactive:  Get space-separated graph adjacency list from stdin.
    -d, --debug:        Print debugging information.\n"""
    if err:
        sys.stderr.write(usg % name)
        return

    sys.stdout.write(usg % name)

def handle_args(argc, argv):
    """ get command line arguments, set variables accordingly
    """
    # TODO change to False for final deployment
    debug = True
    interactive = False

    # arguments should be either 2 files or the two files and some args
```

```
    # unless -i is specified in which case just the args
    if argc < 2:
        usage(argv[0], True)
        sys.exit(1)
    else:
        try:
            # using _ as a variable name is sort of the conventional way of saying
            # "we don't need or use this variable but i'm assigning it because
            # the method or api or whatever i'm using requires it"
            _, args = getopt.getopt(argv, "hid", ["help", "interactive", "debug"])
        except getopt.GetoptError:
            # print usage info and quit
            usage(argv[0], True)
            sys.exit(1)
        # handle args
        for arg in args:
            if arg in ("-h", "--help"):
                usage(argv[0], False)
                sys.exit()
            elif arg in ("-i", "--interactive"):
                interactive = True
                args.remove(arg)
            elif arg in ("-d", "--debug"):
                debug = True
                args.remove(arg)

    return interactive, debug

if __name__ == "__main__":
    main(sys.argv)
```

Listing 5: Python main function to process arguments and run the ismorphism algorithm.

6.2 C++ Code

6.2.1 Graph Class

6.2.2 Subgraph Isomorphism Algorithm

6.2.3 Main

6.3 C++ Code with OpenMP

6.3.1 Graph Class

6.3.2 Subgraph Isomorphism Algorithm

6.3.3 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`.