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

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.[4] 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,[4] 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.[2]

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

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}. M_0 is generated by the following formula:

$$m_{ij} = \begin{cases} 1 & \text{if degree}(j \in G_\beta) \geq \text{ degree}(i \in G_\alpha) \\ 0 & \text{otherwise} \end{cases}$$

Each step of the computation consists of setting all but one of the values of one of the rows of M to 0, and then checking whether exactly one 1 exists in each row. If it does, we have found an isomorphism, since each vertex of G_α has been mapped to exactly one vertex of G_β. If it does not, then we continue onto another row of M and check again. If we have reached the point where we cannot perform the operation on a row of M without violating the condition that each *column* of M contains at most a single 1 (a vertex of G_β cannot be mapped from multiple vertices of G_α), then we backtrack to a previous version of M and try to change the column that we set to 1 in the current row. After exhausting all columns, we conclude there cannot be an isomorphism, since at least one vertex of G_α cannot be mapped to any vertex of G_β.[4]

2.2 "Refine M" Procedure

In the most basic algorithm, the entire tree is searched, including branches which cannot possibly contain a solution at any depth. This is a large number of M matrices to evaluate, so Ullmann introduces a "refine M" procedure to reduce the number that must be checked by eliminating nodes of the tree which cannot contain a solution.

The procedure is actually quite simple. We iterate through each 1 m_{ij} in M and check whether, for each neighbor of the vertex in G_α corresponding to i, there exists at least one possible assignment from it to a neighbor of the vertex in G_β corresponding to j. If this condition does not hold, the 1 is changed to a 0. Since doing so may render other 1s invalid, we repeat the refine procedure on the modified M until running the procedure does not change it any further, indicating that the refinements do not produce any new invalid 1s. This obviously involves several layers of nested loops, adding considerable overhead to the overall algorithm. However, this overhead is very little considering the number of possibilities that are subsequently eliminated, making it a very worthwhile operation to perform regardless.

This refine M procedure is run both on the initial M as well as each of M_d, greatly reducing how many iterations we must perform to see that a particular mapping is invalid.

3 Implementation of the Algorithm

The full source code for all versions can be found in the appendix.

3.1 Python Implementation

The Python code is based largely on an implementation of the Ullmann algorithm provided by an answer on the website *StackOverflow*.[1] It would not run in its original form, so a few adjustments were needed.

```python
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. It calls the function `search`, which performs the actual work of finding an isomorphism.

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 mostly due to the usage of the `has_edge` method of the graph class, shown below.

```python
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).[3] Thus, the `has_edge` method of the Python implementation is O(n).

3.2 Initial C++ Implementation

The initial C++ implementation was based entirely on the Python implementation. Indeed, it represented our best attempt to "translate" the Python code to C++. As one might imagine, it was riddled with compilation errors. That said, we did manage to get it to run, and analysis indicated it performed little better than the Python equivalent. We suspected this was because it retained the same issue that the Python implementation had: calling `has_edge`, an O(n) method, countless times. This motivated a significant improvement.

3.3 Improved C++ Implementation

The improved C++ implementation represents a significant milestone in our project, as it not only improved upon the clarity of the isomorphism algorithm, but also resulted in a much cleaner graph class, with edges implemented as an adjacency matrix. This makes the `has_edge` function O(1), as it consists of simply looking up whether the given vertices share an edge by checking the matrix. This makes its repeated use in `refine_possible_assignments` much more acceptable, a fact which is reflected in the performance.

We also have stored vertices in such a way that each vertex is also stored along with its degree, allowing quick and easy lookup of a vertex's degree.

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.

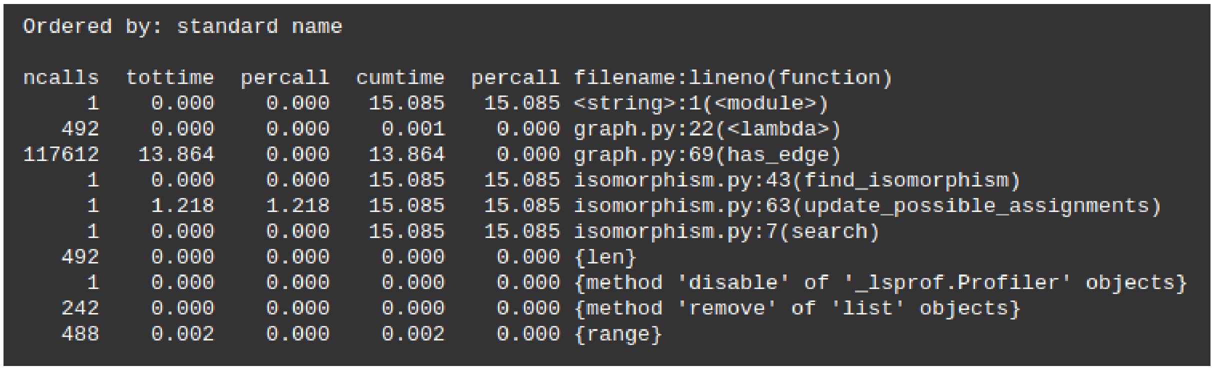
4 Performance

Since the OpenMP optimizations are still in progress, we do not yet have performance information for the OpenMP version of our project.

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 between 13 and 16 seconds to determine if an isomorphism existed between a specified graph a subgraph of the Facebook graph specified earlier.

We used the Python `cProfile` library to measure the performance of our Python implementation. The documentation is available at `https://docs.python.org/3.5/library/profile.html`. A typical run of the `find_isomorphism` function timed by `cProfile` produces the following output:



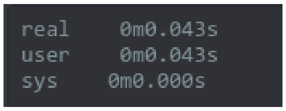
```
Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000   15.085   15.085 <string>:1(<module>)
   492    0.000    0.000    0.001    0.000 graph.py:22(<lambda>)
117612   13.864    0.000   13.864    0.000 graph.py:69(has_edge)
     1    0.000    0.000   15.085   15.085 isomorphism.py:43(find_isomorphism)
     1    1.218    1.218   15.085   15.085 isomorphism.py:63(update_possible_assignments)
     1    0.000    0.000   15.085   15.085 isomorphism.py:7(search)
   492    0.000    0.000    0.000    0.000 {len}
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
   242    0.000    0.000    0.000    0.000 {method 'remove' of 'list' objects}
   488    0.002    0.000    0.002    0.000 {range}
```

Figure 1: Sample run of the Python implementation of subgraph isomorphism.

4.2 C++ Implementation

The C++ implementation, due to the improvement of `has_edge`'s runtime, runs orders of magnitude more quickly than the Python implementation; running in much less than a second where the Python implementation took over ten. We timed it using the Linux `time` utility, which takes a command as an argument, runs the command, and reports the amount of time taken for the command to complete. In our case, we run `time ./main`, adding the appropriate arguments, and `time` reports the time the program took to run and exit. While this adds a bit more overhead than the Python version (we time the entire program rather than a single function call), it is clear this makes little difference, given the huge improvement over the Python version.



```
real    0m0.043s
user    0m0.043s
sys    0m0.000s
```

Figure 2: Timed sample run of the C++ implementation of subgraph isomorphism.

5 Conclusion

Over the coming two weeks, we should have completed the OpenMP parallelizations and thoroughly tested the performance of the resulting program. We will also be much more in-depth in the explanation of the Ullmann algorithm. We also hope to use a more precise method of getting performance information for the C++ implementation, as our current method is rather volatile.

6 Appendix

6.1 Python Code

6.1.1 Graph Class

```python
"""
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

```
#include <algorithm> //Replace, remove, find
#include <iostream>
#include "graph.h"

using std::remove;
using std::find;
using std::get;
using std::cout;
using std::map;
using std::unordered_set;

struct CompareFirst
{
  CompareFirst(int val) : val_(val) {}
  bool operator()(const std::pair<int,int>& elem) const {
    return val_ == elem.first;
  }
  private:
    int val_;
};

Graph::Graph () {
  this->vertices = vector<pair<int, int> >();
  this->edges = vector<vector<int> >();
  this->adjacencies = vector<unordered_set<int> >();
  this->vertex_indices = map<int, int>();
  this->vertex_vals = map<int, int>();
}

void Graph::add_vertex(pair<int, int> vertex){
  vertices.push_back(vertex);
  // add mappings
  vertex_indices[vertex.first] = vertices.size()-1;
  vertex_vals[vertices.size()-1] = vertex.first;
  // add row with edges.size()+1 zeros
  edges.push_back(vector<int>(edges.size()+1,0));
  // add a set for its neighbors
  adjacencies.push_back(unordered_set<int>());
}

void Graph::add_edge(int source, int dest, int weight){
  // weight should be non-zero

  if (this->get_index(source) < 0) {
    this->add_vertex(pair<int, int>(source, 0));
  }

  if (get_index(dest) < 0) {
    this->add_vertex(pair<int, int>(source, 0));
  }
  // increase vertex degree
  vertices[this->get_index(source)].second++;
  vertices[this->get_index(dest)].second++;

  // source -> dest
  // we'd need a flag or something for directed graphs
  // somewhere
  edges[this->get_index(source)][this->get_index(dest)] = weight;
  edges[this->get_index(dest)][this->get_index(source)] = weight;
  // add dest/source to that set
  // once again, consider directed graphs
  this->add_neighbor(source, dest);
  this->add_neighbor(dest, source);
}

bool Graph::has_edge(int vert1, int vert2){
  // just got this bad boy down to O(1)
  // you're welcome
  return (edges[vert1][vert2] != 0);
}

int Graph::get_index(int value) {
  if (vertex_indices.find(value) != vertex_indices.end()) {
    return vertex_indices[value];
  }

  return -1;
}

int Graph::get_value(int index) {
  if (vertex_vals.find(index) != vertex_vals.end()) {
    return vertex_vals[index];
  }

  return -1;
}


unordered_set<int> Graph::neighbors(int vertex) {
  return this->adjacencies[this->get_index(vertex)];
}

void Graph::add_neighbor (int vert, int neighbor) {
  this->neighbors(vert).insert(this->vertices[this->get_index(neighbor)].first);
```

```
}

// for debugging
void Graph::print_graph() {
  cout << "Begin graph" << std::endl;
  cout << "-------------------------------\n";
  cout << "Vertices: " << vertices << std::endl;
  cout << "Edges: " << edges << std::endl;
  cout << "-------------------------------\n";
  cout << "End graph" << std::endl;
}
```

Listing 6: C++ graph class.

6.2.2 Subgraph Isomorphism Algorithm

```
#include <vector>
#include <utility>
#include <unordered_set>
#include <algorithm> //for std::find
#include "isomorphism.h"

using std::find;

vector < pair<int,int> > *find_isomorphism (Graph &sub, Graph &graph) {

  // assignments_tree[i] specifies the possible_assignments at depth i
  vector <vector <vector <bool> > > assignments_tree;
  // generate M0
  vector < vector<bool> > possible_assignments = create_possible_assignments(sub, grap
  // return the address of this vector, or null if no isomorphism was found

  //This doesn't work, assignments is out of scope. Throw it on the heap.
```

```
//vector <pair<int, int> > assignments = vector<pair<int, int> >(0);

vector <pair<int, int> > *assignments = new vector <pair<int, int> >;


// columns_used[i] = true iff column i in M has been used at current stage of comput
vector <bool> columns_used (possible_assignments[0].size(), false);

// column_depth[d] = k if column k was used at depth d
vector<int> column_depth (possible_assignments.size(), 0);

ssize_t i, j, k, pa_n = possible_assignments.size(), pb_n = possible_assignments[0].

int depth = 0;
columns_used[depth] = 0;

// refine the possible assignments using the initial set of possible assignments
// refine M0
if (!refine_possible_assignments(sub, graph, possible_assignments)) {
  // if refine_possible_assignments returns false, then there is no possible isomorp
  // so we can just stop here.
  return NULL;
}

// check if there's a j such that possible_assignments[d][j] = 1 and f[j] = 0
// 2
// if there is no j such that possible_assignments[d][j] == true and f[j] == false g
two:
if (find(possible_assignments[depth].begin(),
      possible_assignments[depth].end(), true) != possible_assignments[depth].end()
    && find(column_depth.begin(), column_depth.end(), false) != column_depth.end())
  // there exists such a j
```

```
    assignments_tree.push_back(possible_assignments);
    if (depth == 0) {
      k = column_depth[depth];
    }

    else {
      k = 0;
    }
  }

  else {
    // we found no such j
    goto seven;
  }

  // 3
  // increment k until we find a possible assignment for this row
  three:
  for (; !possible_assignments[depth][k] || columns_used[k]; k++) ;

  // found one.
  for (j = 0; j < pb_n; j++) {
    // set all of the columns in this row to 0 except that first assignment we just fo
    if (j != k) {
      possible_assignments[depth][j] = false;
    }
  }

  if(!refine_possible_assignments(sub, graph, possible_assignments)) {
    // 5
    five:
    int r;
    r = k+1;
    for (; r < pb_n; r++) {
      if (possible_assignments[depth][j] && !columns_used[j]) {
        possible_assignments = assignments_tree[depth];
        goto three;
      }
    }
    goto seven;

    // 6
    six:
    column_depth[depth] = k;
    columns_used[depth] = true;
    depth++;
    goto two;

    // 7
    seven:
    if (depth == 1) {
      return NULL;
    }

    else {
      columns_used[depth] = false;
      depth--;
      possible_assignments = assignments_tree[depth];
      k = column_depth[depth];
      goto five;
    }

  }
```

```
  // 4
  four:
  if (depth < pa_n) {
    goto six;
  }

  else {
    // isomorphism found. add it
    for (int i = 0; i < pa_n; i++) {
      for (int j = 0; j < pb_n; j++) {
        if (possible_assignments[i][j]) {
          (*assignments).push_back(pair<int, int>(sub.get_value(i),
                                                  graph.get_value(j)));
        }
      }
    }
  }

  return assignments;
}

vector < vector<bool> > create_possible_assignments(Graph &sub, Graph &graph) {
  // possible_assignments[i][j] == true iff a possible assignment
  // exists from i in sub
  // to j in search graph
  vector < vector<bool> > possible_assignments (sub.vertices.size(),
    vector<bool>(graph.vertices.size(), false));

  // at first, every vertex in search graph with rank >= i
  // is a possible assignments
  // this will be refined later
  ssize_t i, j;
  ssize_t s_n = sub.vertices.size();
  ssize_t g_n = graph.vertices.size();
  for (i = 0; i < s_n; i++) {
    for (j = 0; j < g_n; j++) {
      if (graph.vertices[j].second >= sub.vertices[i].second) {
        possible_assignments[i][j] = true;
      }
    }
  }

  return possible_assignments;
}


bool refine_possible_assignments(Graph &sub, Graph &graph,
    vector < vector<bool> > &possible_assignments) {
  ssize_t i, j;
  ssize_t pa_n = possible_assignments.size();
  ssize_t pb_n = possible_assignments[0].size();
  bool changes_made = true;

  while (changes_made) {
    changes_made = false;
    for (i = 0; i < pa_n; i++) {
      // check if this row contains no 1s
      bool no_one = true;
      for (j = 0; j < pb_n; j++) {
        if (possible_assignments[i][j]) {
          no_one = false;
          // check if all of i's neighbors have a possible assignment
          // to a neighbor of j
          // iterate through all neighbors of i
          unordered_set<int> neighbors_i = sub.neighbors(sub.get_value(i));
          unordered_set<int>::iterator n_i = neighbors_i.begin();

          for (; n_i != neighbors_i.end(); n_i++) {
            // for each neighbor of i, iterate through all its possible assignments
            ssize_t k;
            bool has_corresponding_neighbor = false;
            for (k = 0; k < pb_n; k++) {
              if (possible_assignments[sub.get_index(*(n_i))][k]) {
                // for each possible assignment from i's neighbor to graph,
                // check if it is neighbor of j
                if (graph.has_edge(graph.get_value(j), graph.get_value(k))) {
                  // if so, then check the next neighbor
                  has_corresponding_neighbor = true;
                  break;
                }
              }
            }

            // if this neighbor has no corresponding neighbor,
            // then j is an invalid match.
            // move on to the next possible assignment
              if (!has_corresponding_neighbor) {
                possible_assignments[i][j] = 0;
                changes_made = true;
                break;
              }
          }
        }
      }

      // we never found a 1 in this row, so there's no point in continuing
      // there's at least one vertex in subgraph that cannot be mapped to any
```

```
      // vertex in the search graph, so we know that
      // possible_assignments cannot specify any isomorphism
      if (no_one) {
        return false;
      }
    }
  }

  // M was successfully refined without creating any rows with all 0s. return true.
  return true;
}
```

Listing 7: C++ algorithm to find isomorphism.

6.2.3 Main

```
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <vector>
#include <tuple>
#include <getopt.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include "graph.h"
#include "isomorphism.h"

using std::string;

const char* program_name;
```

```
string *handle_args(int argc, char **argv);
void usage (FILE* stream, int exit_code);
Graph import_data(const char *filename, const int debug);

int main(int argc, char **argv) {

  program_name = argv[0];
  string *args = handle_args(argc, argv);

  int interactive = atoi(args[0].c_str());
  int debug = atoi(args[1].c_str());

  char filename_default[] = "stdin";

  const char *graph_filename = (!interactive) ? args[2].c_str()
                                              : filename_default;
  const char *sub_filename = (!interactive) ? args[3].c_str()
                                              : filename_default;
  if (debug) {
    printf("inter = %d, debug = %d\n", interactive, debug);
    printf("Graph file %s, subgraph file %s\n", graph_filename, sub_filename);
  }



  Graph graph = import_data(graph_filename, debug);
  Graph subgraph = import_data(sub_filename, debug);

  int matches = 0;
  vector < pair<int,int> > *result = find_isomorphism(subgraph, graph);
  printf("Made it to after find_isomorphism");
  return 0;
```

```
}

Graph import_data(const char *filename, const int debug) {

  FILE *fd;
  if (strcmp(filename, "stdin") == 0) {
    fd = stdin;
  }

  else {
    fd = fopen(filename, "r");
    if (fd == NULL) {
      fprintf(stderr, "Could not open file %s for reading.\n", filename);
      perror("Error calling fopen()");
      exit(errno);
    }
  }

  char *line = NULL;
  size_t n = 0;

  Graph g;

  if (debug) {
    printf("Reading file %s...\n", filename);
  }

  char *tempa, *tempb, *tempweight;
  char tempweight_default[] = "0";
  // getline returns -1 on failure to read a line (including EOF)
  while (getline(&line, &n, fd) != -1) {
    tempa = strtok(line, " ");
    // strtok returns NULL if the token (space in this case) was not found in the str
    // otherwise return the string up to the token (exclusive)
    if (tempa == NULL) {
      fprintf(stderr, "Error parsing file %s. Adjacency list must be of form \
            'node1 node2 [edge_weight]', but line was %s\n", filename, line);
      exit(-1);
    }

    tempb = strtok(line, " ");
    if (tempb == NULL) {
      fprintf(stderr, "Error parsing file %s. \
          Adjacency list must be of form 'node1 node2 [edge_weight]',\
          but line was %s\n", filename, line);
      exit(-1);
    }

    tempweight = strtok(line, " ");
    if (tempweight == NULL) {
      tempweight = tempweight_default;
    }

    int atempa = atoi(tempa);
    int atempb = atoi(tempb);
    int atempweight = atoi(tempweight);

    // this will also add the vertices if they don't exist
    // add_vertex is called from within add_edge if no vertex
    // with the given value exists
    g.add_edge(atempa, atempb, atempweight);
  }

  if (fd != stdin) {
    int stat = fclose(fd);
    if (stat != 0) {
      perror("Error closing file");
      // we never wrote to the file
      // so we probably don't even need to check for errors in the first place
      // but just kill the program if there's an error closing it
      exit(errno);
    }
  }

  free(line);

  return g;
}

string *handle_args(int argc, char **argv) {
  // for getopt
  const char* const short_options = "hid";
  const struct option long_options[] = {
    { "help",  0, NULL, 'h' },
    { "interactive", 0, NULL, 'i'},
    { "debug", 0, NULL, 'd'},
    { NULL, 0, NULL, 0 }
  };

  // we only need to save interactive and debug for now
  string *returnargs = (string *) malloc(2*sizeof(string));
  int interactive = 0;
  // TODO change to 0 for final deployment
  int debug = 1;

  int next_option;
```

```
do {
  next_option = getopt_long(argc, argv, short_options,
                            long_options, NULL);
  switch(next_option) {
    case ’h’:
      usage(stdout, 0);
      // calling usage quits the program anyway but whatever
      // we’ll put a break here anyway
      break;

    case ’i’:
      interactive = 1;
      break;

    case ’d’:
      debug = 1;
      break;

    case ’?’: // invalid option
      usage(stderr, 1);
      break;

    case -1: // no more options
      break;

    default: // something went very wrong if we got here
      free(returnargs);
      abort();
  }
}
while (next_option != -1);
```

```
  // now that we've gone through all the options, OPIND points to first
  // nonoption argument. which is hopefully the first filename. unless
  // interactive was specified, in which case yell at the user.

  if (!interactive) {
    // interactive was 0.
    // free the memory it took up and reallocate enough for the two args
    // and the two filenames
    free(returnargs);
    returnargs = (string *) malloc(4*sizeof(string));
    returnargs[2] = argv[optind];
    returnargs[3] = argv[optind+1];

  }

  if (debug) {
    printf("Interactive set to %d. Debug is %d.\n", interactive, debug);
    if (!interactive) {
      printf("Using file %s for graph and %s for subgraph.\n",
          argv[optind],argv[optind+1]);
    }
  }

  // put what we want to return in their places.
  returnargs[0] = std::to_string(interactive);
  returnargs[1] = std::to_string(debug);

  return returnargs;
}

void usage (FILE* stream, int exit_code) {
  fprintf(stream, "usage: %s [FILE] [OPTION]\n", program_name);
  fprintf(stream,
    "Options and arguments:\n"
    "FILE:               Space-separated file containing graph adjacency list. \
              If edge weights are not specified, all weights will be assumed 0.\n"
    "-h, --help:         Print this help message and exit.\n"
    "-i, --interactive:  Get space-separated graph adjacency list from stdin.\n"
    "-d, --debug:        Print debugging information.\n");

  exit(exit_code);
}
```

Listing 8: C++ main function to process arguments and run the ismorphism algorithm.

References

[1] Vaughn Cato. *How to partially compare two graphs*. Ed. by Spiros Eliopoulos. Nov. 24, 2012. URL: https://stackoverflow.com/a/13537776 (visited on 11/26/2017).

[2] 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).

[3] *TimeComplexity*. URL: https://wiki.python.org/moin/TimeComplexity (visited on 11/26/2017).

[4] 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.