Data Visualization using Graph Implementation on Partial Panama Papers Dataset in C++

Anh Nguyen
Gina-Cody School of Engineering & Computer Science
Concordia University
Montréal, Canada
anhhoang.nguyen@concordia.ca

Instructor: Prof. Dr. Yan Liu
Gina-Cody School of Engineering & Computer Science
Concordia University
Montréal, Canada

***Abstract*—This report describes the programming project C++ source code that utilizes the Object-Oriented Programming (OOP) paradigm to visualize the partial Panama Papers dataset. Using adjacency list and the depth-first search graph traversal algorithm, the visualization successfully represented connections between corporate entities in the dataset by jurisdiction of the tax haven the entity registered in and by the country of origin of the entity. On average, the program took 13.20 seconds on Windows platform and 3.50 seconds on Unix-like platforms to successfully parse 2000 dataset entries into nodes and edges. The executable program serves a proof of working application of several programming paradigms, methodologies, and techniques.**

***Index Terms*—object-oriented programming, graph theory, search algorithm, c++, Panama Papers, data science, data visualization**

I. Introduction

Tax has always been a influential and nonetheless controversial aspect of an economy, dating back to the Ancient Egypt civilization thousands of BC's ago. Now, tax plays an important role in maintaining a country's economy, developing its infrastructure, as well as funding for protecting its sovereignty. Individuals and corporations have to pay taxes to their host country or region. Such concept could be more simply described as "paying one's rent" for living or a type of "subscription model" per se in the mdoern-day software bartering environment. For big multi-national corporations, such taxes are levied against a large chunk of their annual gross income, so it is natural for them to find a way to get lower taxes, sometimes through loopholes that fall between the line of what are legal and what are illegal.

The Panama Papers [1] is a collective term used for referencing the leaked incriminating documents against big corporations who are creating shell companies in the Pacific, where asset taxes are lower, through a middle-man law firm called *Mossack Fonseca* based in the Republic of Panama, dubbed a "tax haven" by critics. Originally emails and transactions and ledgers between the corporations, the shells, and the law firm, the *International Consortium of Investigative Journalists* (ICIJ) sifted through nearly millions of documents and as a result, compiled a database of corporate entities involved in one of the largest tax evasion scheme.

With such an incredibly large database with equally large number of features (or attributes), one would need to make sense of and, in turn, infer connections by using an automated method. That is where this C++ program comes in to play. As the de facto, high-level programming language for high efficiency and implementation of the object-oriented paradigm, `C++` was used to parse the datasets in the *Panama Papers* database into graphs and then visualize them with adjacency lists.

II. Choose and Clean the Dataset

A. The Choosing Part

On the ICIJ's main article for these financial crimes investigation, there are 4 total databases: *Bahamas Leaks*, *Offshore Leaks*, *Panama Papers*, and *Paradise Papers*. This report focus on a particular dataset in the *Panama Papers* database with the name `panama_papers.nodes.entity.csv` since the parsing and visualizing that dataset are more easily implemented as a proof-of-concept.

B. The Cleaning Part

Though the `panama_papers.nodes.entity.csv` dataset was well-documented, well-annotated, and well-recorded with up to 17 features and more than 21,000 entries (data points), some of the features are either not useful for data visualization or proved to be extremely complicated for the scope of the project to implement. In addition, the original dataset was marred with the delimiter symbol (comma `,`) or double-quotes (`""`) which could confuse the standard `C++ <stringstream>` library; plus, the overwhelming length of the dataset would either inevitably crash this single-threaded program due to non-parallel computation limit or run out of memory allocated by the compiler by default.

To clean up the string data and cut down on the number of features and length, the *Python* `pandas` package was employed in a *Jupyter Notebook* environment to drop 17 features (columns) to just 6 and 21000 length to just 2000. The 6 chosen features were: *node_id, name, jurisdiction_code, jurisdiction, country_code, and country_name*. Then, a string search-and-destroy method was involved to remove commas and double-quotes from the string data. The resulting cleaned

partial dataset proved to be more manageable by the completed program in the end, as well as more legible for human readers.

III. Methodology and Design

After the dataset was processed for the program, various programming methods could then be employed to write the source code.

A. Exploratory Data Analysis

The better write a suitable program, a better understanding of the dataset was needed. Therefore, some exploratory data analysis was done using, again, the `pandas` package in the aforementioned development environment. It was already known that there were 6 features to work on and 2 out of 6 were just the full name of the country codes. Table I below shows the features, their corresponding data types, and how many unique values each attribute (node's ID excluded) had.

TABLE I
METADATA OF PANAMA PAPERS NODES DATASET

Feature	Data Type	Unique Values
node_id	`int64`	not applied
name	`string`	1996
jurisdiction	`string`	6
country_codes	`string`	52

It was observed that for 2000 entries, there are 1996 unique name values because 3 entries had the same name and same jurisdiction, but different in country of origin. With 6 unique values for jurisdiction and 52 country of origin, a preliminary image of how the data was visualized could be formed. It was decided that 2 entries would be connected with an edge by their country code, which the adjacency list would also rely on for applying DFS algorithm to print out branches. Then, the jurisdiction would be printed alongside the nodes.

B. OOP and Abstraction of Data Entries

Leveraging the Object-Oriented Paradigm of C++, the dataset could be parsed into (undirected) graph objects, nodes objects, and edge objects. The choice of using undirected graph was based upon the fact that these connections by countries were only scalar without a way to represent a directed relationship between entities. In the future, it could be improved by adding how much tax was evaded through shell companies and a weighted directed graph could be obtained; however, as of the writing of this report, implementing such a complex structure was infeasible without additional time, knowledge, and processing power. Here is description of how the dataset was parsed into objects and their respective attributes and the accompanying Unified Modeling Language (UML) diagram could be found in the appendix A V.

- `Container` is an object that stores many `Graph` objects.
- `GraphBase` is an abstract class that can be implemented into concrete class `Graph` object.
- `Graph` is an object that represents a dataset tentatively in the form of an undirected graph. It stores a node list and edge list as `std::vector` of pointers to `Node` and `Edge` objects.
- `Node` is an object that represents an entry in the dataset. Each has a unique ID along with the 5 aforementioned features as attributes.
- `Edge` is an object that represents connection between 2 `Node` objects by their country of origin. Each has a binary node list as `std::vector` of pointers to said `Node` objects.

C. Non-trivial Methods

There are several non-trivial methods implemented across the utils header and 3 concrete object classes (Graph, Node, and Edge) to streamline the data parsing. Here are two examples of non-trivial member functions:

1) `Edge::Edge(Node& n1, Node& n2)` is the overloaded constructor for an Edge object which, in turn, calls `addNodePair(n1, n2)` method to add 2 Node objects to itself.
2) `Graph::display()` is the centerpiece of the data visualization process. Rather than passing in an argument, it starts an interactive subroutine where user inputs an index range. After receiving and checking the validity of the range, it uses the DFS graph traversal algorithm to create a subtree for each index in the range with the root node as the Node object at index. After each subtree is created, the subroutine outputs the result as an adjacency list with other Node objects that share a common attribute to the root, in this case: country code. Each subtree is pivoted on the jurisdiction feature of the root Node.

D. Programming Techniques

The programming techniques used in the program were: inheritance, polymorphism, operator overloading, and exception handling.

1) **Inheritance** was used in defining the abstract `GraphBase` and deriving the concrete `Graph` for the actual implementation. As said before, this allows more different types of derived graph-like classes with specialized features and functionalities. Figure 1

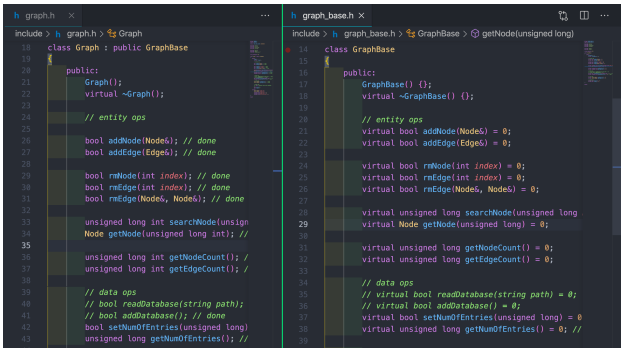

Fig. 1. Inheritance of Graph from abstract GraphBase

2) **Polymorphism** was used in conjunction with inheritance to overload methods of the defined classes. One example was overloading the `rmEdge` method, one was used for removing edge by its index in the list, while other was used for remove the Edge object along with its child Node objects pointing to the node list. Figure 2 shows the implementation code.

```
bool Graph::rmEdge(int index)
{
    edge_list.erase(edge_list.begin() + index);
    edge_count--;
    return 1;
}

bool Graph::rmEdge(Node& n1, Node& n2)
{
    int counter = 0;
    for (auto edge : edge_list) {
        if (*find(edge->getNodePair().begin(), edge->getNodePair().end(), &n1)
            && *find(edge->getNodePair().begin(), edge->getNodePair().end(), &n2)) {
            rmEdge(counter);
        }
        counter++;
    }
    return 1;
}
```

Fig. 2. Implementation of the Graph::rmEdge method overloading

3) **Operator Overloading** was used as a method overloading == of the `Edge` object to check whether or not the new `Edge` being added had already existed. It The method returns a Boolean value by comparing the ID of the child `Node` objects. Figure 3 shows the implementation.

```
bool operator==(const Edge& lhs, const Edge& rhs)
{
    bool flag;
    string a,b,c,d;
    a = lhs.getNodePair()[0]->getName();
    b = lhs.getNodePair()[1]->getName();
    c = rhs.getNodePair()[0]->getName();
    d = rhs.getNodePair()[1]->getName();
    flag = ((a == c && b == d) || (a == d && b == c) || ((a == b) && (c == d)));

    return flag;
}
```

Fig. 3. Implementation of == operator overloading in Edge

4) **Exception Handling** was used generically wherever user input required during the program runtime (inputting the range for subtree generation) and wherever a system-level command was invoked (system call to `pause` or `read` for legibility).

IV. Interfacing, Processing, and Parsing

For the program to interface with the host platform, find the dataset, process it, and parse it into the graph implementation, a robust utility header `utils.h` was conceived. By doing such, the program package would be more modular and the functions would more reusable on different implementations of the abstract `GraphBase` class. The following figure 4 describes the life cycle of the `utils` interface. Hereforth, "utility", "utils", and other morphics of utility refer to the user-defined `utils` header, not the `std::utility`.

The `utils` interface starts out by determining the host platform on which the program is being executed. Then, the interface queries the full system path to the data directory

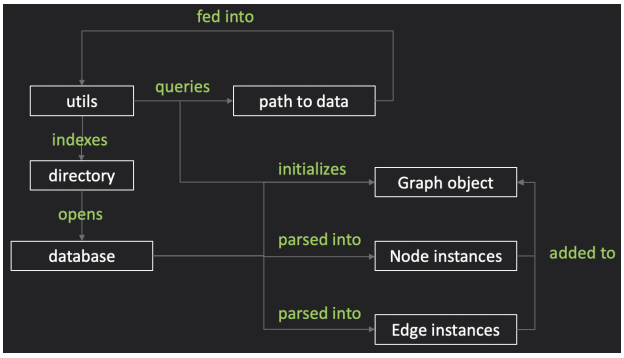

Fig. 4. The Utility Functions Life Cycle During Program Runtime

within the program package. That path string is then fed back into the interface for indexing any at all datasets in `.csv` format within the directory. Found datasets are initialized into a temporary `ifstream` and a new `Graph` object. The interface then parses, by line, entries in the currently operated dataset into `Node` and `Edge` objects. The connections between nodes (entries) are checked and made into edges on-the-go with every new node added. After iterating through one entry, the interface calls the graph's mutators to update node and edge list, as well as counter, private attributes. Then the interface loops until End of File and moves on to next dataset, if any. Figure 5 shows an example of the "data2graph" pipeline of the program.

Fig. 5. An Example for the Data2Graph Pipeline of the Program

Although the naivety of this parsing approach with nested loops makes it easy to implement, this approach, however, is far from optimized with both space and time complexity being exponentially proportional to the number of entries. As of the writing of this report, a better solution has yet to be conceived.

V. Testing and Results

The project source code was successfully compiled and tested on Windows, Linux, and macOS platforms. In addition, a Dockerfile configured for the project is provided, as well as an online development environment (Repl.it).

Please refer to the following screen recording of the runtime demonstration: Google Drive.

References

[1] I. C. of Investigative Journalists, "The panama papers: Exposing the rogue offshore finance industry," Available at https://www.icij.org/investigations/panama-papers/ (2021/04/21).

APPENDIX A. UML DIAGRAM AND DESCRIPTIONS

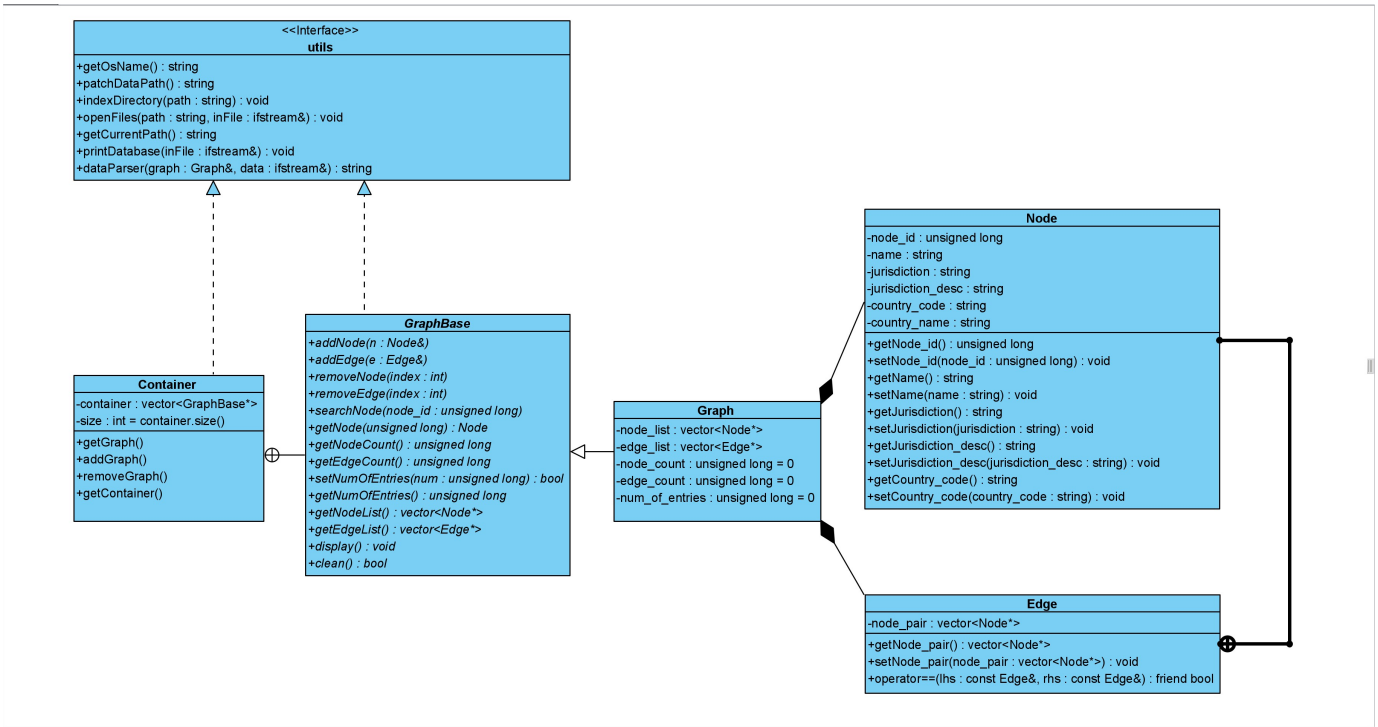

Fig. 6. The Unified Modeling Language (UML) Diagram for the Project