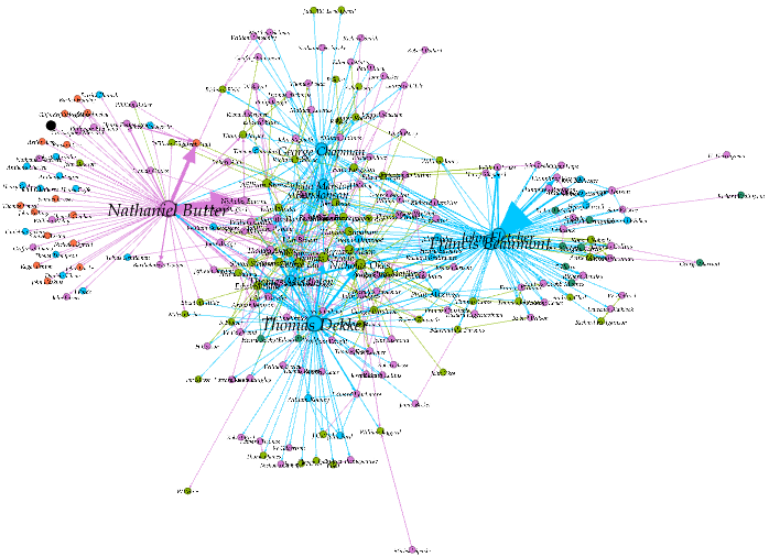
12/29/22

WHO AM I AND WHAT IS THIS?

Hello, my name is Sarah and I'm an undergraduate student who worked for Dr. Giugni this past semester. I intend to major in history and minor in computer science, but I used to be entirely terrified when it came to writing code. However, this past year I began taking computer science courses and realized that learning to code was an incredibly useful skill, even at the most basic of levels. Even though I intend to pursue the humanities, code can enable me to do things for research purposes that traditional means of conducting research cannot. The following code write-up is merely a small example of what basic coding can help us do when it comes to digital humanities. It comes from the project I worked on for Dr. Giugni this past semester in which we attempted to reconstruct part of the social network of the publishing and printing world of 17th century London.

SOCIAL NETWORK ANALYSIS & GEPHI

I used two programs to write this code that generates the network. One is Visual Studio Code, which is a standard run-of-the-mill source code editor. The other is Gephi, which is a program that takes spreadsheets of nodes (objects) and edges (connections between objects) and creates networks that look something like the image below (this is my own network). Gephi is convenient for analyzing social networks as it provides visual representations of the strength of relationships between people.



12/29/22

Gephi requires two spreadsheets, one of nodes, and another of edges to generate visual representations of social networks. Nodes, which in this instance are people (either publishers, printers, authors, booksellers, or preachers), are required to have a **Label** and a unique **ID.** The label is the full name of the person in question, and the ID I wanted to make for each person is meant to include the first three letters of their first name (if they were known), a period, and then the first four letters of their last name. Nodes in Gephi can also be given various characteristics, such as **Type.** I wanted to label each node accordingly with a type that indicates whether the person in question was a publisher, a printer, an author, a bookseller, or a preacher.

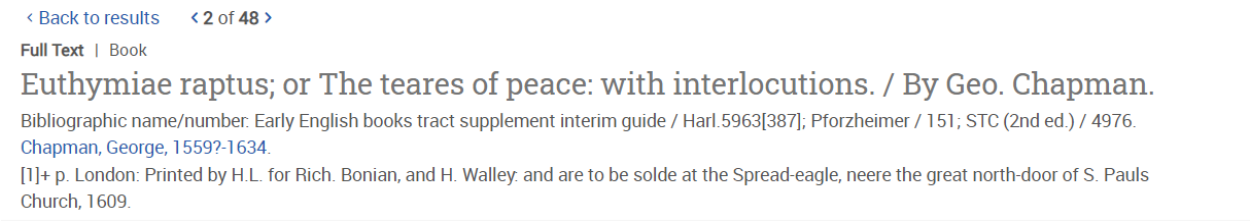
Edges can be thought of as links between two people, and their strength in this representation was based on the frequency of which two people collaborated. In Gephi, edges must have a **Source,** a **Target,** and a **Weight.** The source is the first person (in the form of ID corresponding to their node), and the target is the second person they are being linked to (in the form of ID corresponding to their node). Weight is a measure of the strength of their relationship, which was based on frequency in this model. The more two people collaborated, the greater the weight of the edge between them.

Since it is rather tedious to track down how many times two people worked with each other through a slew of spreadsheets by hand that correspond to EEBO entries (which some of my fellow undergraduate researchers did on this project), I wrote two modules in Java that could efficiently clean data and get it ready for Gephi to make networks with, calculating both IDs for nodes and correctly creating edges with calculations of the number of times two people collaborated with each other. My code also kept track of what persons were publishers, authors, or printers etc. so that my nodes for Gephi could be correctly labeled with their respective Types.

I: DATA FORMATTING FOR THE CODE TO PROCESS

The information for this project came from Early English Books Online (EEBO). EEBO entries look like this:

12/29/22


‹ Back to results ‹ 2 of 48 ›
Full Text | Book
Euthymiae raptus; or The teares of peace: with interlocutions. / By Geo. Chapman.
Bibliographic name/number: Early English books tract supplement interim guide / Harl.5963[387]; Pforzheimer / 151; STC (2nd ed.) / 4976.
Chapman, George, 1559?-1634.
[1]+ p. London: Printed by H.L. for Rich. Bonian, and H. Walley: and are to be solde at the Spread-eagle, neere the great north-door of S. Pauls Church, 1609.

This is not helpful for trying to determine who knows who and put it into Gephi, especially since Gephi is incredibly particular about its specifications. However, representing relationships between people like the spreadsheet format below is perfect for processing in my code.

Francis Beaumont:Author	Dorman Newman:Publisher
John Fletcher:Author	Thomas Collins:Publisher
Francis Beaumont:Author	Thomas Collins:Publisher
John Fletcher:Author	Thomas Collins:Publisher
Francis Beaumont:Author	Thomas Collins:Publisher
John Fletcher:Author	Dorman Newman:Publisher
Francis Beaumont:Author	Dorman Newman:Publisher
Dorman Newman:Publish	George Marriott:Bookseller
Thomas Collins:Publisher	George Marriott:Bookseller
Anne Maxwell:Printer	John Fletcher:Author
Anne Maxwell:Printer	Francis Beaumont:Author
Anne Maxwell:Printer	Robert Roberts:Printer
Robert Roberts:Printer	Francis Beaumont:Author
Robert Roberts:Printer	John Fletcher:Author
Anne Maxwell:Printer	Dorman Newman:Publisher
Anne Maxwell:Printer	Thomas Collins:Publisher
Robert Roberts:Printer	Dorman Newman:Publisher
Robert Roberts:Printer	Thomas Collins:Publisher

Each line corresponds to a single collaborative relationship. Francis Beaumont is the author, Dorman Newman is the publisher for whatever work this information was pulled from when we look at the top line. Assembling this information in this exact format for every work by each author, publisher, and or printer you are looking at the network for is crucial to being able to use the code I wrote to create a network using Gephi. My code takes this raw data and spits back relationships with strengths

corresponding to the number of times two people collaborated. Its nodes spreadsheet output looks like the image below. This can be copied and pasted into Google Sheets using a comma delimiter to create the spreadsheet Gephi requires.

```
Label (Name), ID, Status
----------------------
Adam Islip,Ada.Isli,Printer
Andrew Clark,And.Clar,Printer
Andrew Cooke,And.Cook,Publisher
Andrew Penneycuicke,And.Penn,Publisher
Anne Maxwell,Ann.Maxw,Printer
Anne Moseley,Ann.Mose,Publisher
Anthony Magino,Ant.Magi,Author
Anthony Sherley,Ant.Sher,Author
Arthur Johnson,Art.John,Publisher
```

The edges spreadsheet output it generates looks like the next image.

```
Source, Target, Weight of Relationship (Frequency):
---------------------------------------------------
Ada.Isli,Wil.Holm,1
And.Penn,J.Bel,2
Ann.Maxw,Dor.Newm,2
Ann.Maxw,Fra.Beau,1
Ann.Maxw,Joh.Flet,1
Ann.Maxw,Lan.Curt,1
Ann.Maxw,Rob.Robe,2
Ann.Maxw,Tho.Coll,2
Ann.Mose,Hum.Mose,1
Ann.Mose,Hum.Robi,1
```

This can also be copied and pasted into Google Sheets with a comma delimiter to create the edges spreadsheet Gephi requires. Altogether, the code generates unique IDs for each node in the Gephi graph and it correctly counts how many times two individuals collaborated. It also keeps track of the status (Type) of each node. But how does it do it?

12/29/22

DataCleaner

DataCleaner is the first module I wrote in Java to process the raw spreadsheet. It has 3 methods.

The first method readtoList acts as a file reader of the raw spreadsheet and converts it into a list of every name mentioned in the raw spreadsheet. This step is crucial for taking the data out of the spreadsheet and turning it into something my code can process.

```
public ArrayList<String> readtoList(String csvFile) {

    ArrayList<String> allNames = new ArrayList<String>();

    try {
        File file = new File(csvFile);
        FileReader fr = new FileReader(file);
        BufferedReader br = new BufferedReader(fr);

        String line = "";

        String[] tempArr;
        while ((line = br.readLine()) != null) {
            tempArr = line.split(",");
            for (String tempStr : tempArr) {
                allNames.add(tempStr);
            }
```

The line `ArrayList<String> allNames = new ArrayList<String>();` **creates the list into which every single name in the spreadsheet is going to be inputted. The FileReader and BufferedReader lines create objects that Java needs in order to read my file. The file I am inputting into this code is a CSV (Comma Separated Values) version of the raw spreadsheet I created.**

`while ((line - br.readLine() != null)` **simply states that as long as the spreadsheet has more lines to be read/processed, my code will keep processing the spreadsheet.**

`tempArr = line.split(",")` takes each line of the spreadsheet and splits it wherever it finds a comma. It creates a temporary array, hence the name tempArr. Since the spreadsheet is in CSV format, this means that this will separate each line of the spreadsheet into two since each line has Name1:Status, Name2:Status.

```
for (String tempStr: tempArr) {
allNames.add(tempStr);
}
```

Is a loop of code that essentially says for every entry inside a line in the spreadsheet, add that name to the list we created at the very beginning called allNames.

```
                System.out.println();
            }
            br.close();

        } catch (IOException ex) {
            ex.printStackTrace();
        }

    return allNames;
    }
```

The rest of this code is pretty much unimportant in understanding why the code works. All this does is close the file and make sure that if there's a handling error in the uploading of the file to the code our program has a way to keep running. The one important thing is that it returns allNames, meaning that this method gives us a list of names in the spreadsheet with their corresponding statuses.

allNames would look something like this:

`[George Chapman:Author, Richard Bonian:Publisher, H Walley:Publisher]` … and so on and so forth.

The second method **getNamePairs** acts as a file reader/scanner of the CSV version of the raw spreadsheet and converts it into an ArrayList of every single collaboration

between people with each item in this ArrayList being formatted as Name1&Name2. This is necessary for the way buildRelationships runs in NodesandEdges (see next section) to correctly build the edges needed for Gephi.

```
public ArrayList<String> getNamePairs(String csvFile) {
        ArrayList<String> namePairs = new ArrayList<String>();

        try {
            File file = new File(csvFile);
            FileReader fr = new FileReader(file);
            BufferedReader br = new BufferedReader(fr);

            String line = "";
```

All of this code is pretty much the same as the last method we looked at. It creates a new list for the pairs of names we want called `namePairs` and it creates the objects necessary for my code to read our file.

```
            String[] tempArr;
            while ((line = br.readLine()) != null) {
                tempArr = line.split(",");
                String name1Status = tempArr[0];
                String[] tempname1split = name1Status.split(":");
                String name1 = tempname1split[0];
                String name2Status = tempArr[1];
                String[] tempname2split = name2Status.split(":");
                String name2 = tempname2split[0];
                String namePair = name1 + "&" + name2;
                namePairs.add(namePair);
            }
            br.close();

        } catch (IOException ex) {
            ex.printStackTrace();
        }
    return namePairs;
    }
```

The rest of this method says that as long as there are more entries in the spreadsheet, we need to keep reading the file. The rest of the code specifies that it wants the list to return in the format of Name1&Name2 for each line of the spreadsheet being read.

The list `namePairs` looks something like:

`[George Chapman&Richard Bonian, Richard Bonian&H Walley]` … so on and so forth.

The third method is the **main** method. It calls all the functions from NodesandEdges (see next sections) to print the output of the Node spreadsheet for Gephi and the output for the Edge spreadsheet for Gephi with correct IDs and relationship strengths. Since this main method calls all the methods built in my other Java module, I will return to discuss this method after discussing the other module, NodesandEdges.

NodesandEdges

NodesandEdges is the second of the Java modules I wrote to process this new spreadsheet I had created. It contains 5 methods that create the nodes and edges necessary for a Gephi network.

The first method, **buildIDs** processes the spreadsheet and returns a dictionary/map where the keys are the names of people (labels for Gephi) and their corresponding values are the unique IDs needed for Gephi nodes. A dictionary (and or map depending on the programming language) is a data structure in which keys store corresponding values. This is vital for getting the labels and IDs needed for the node spreadsheet Gephi requires.

```
public  Map<String, String> buildIDs(ArrayList<String> allNames) {
    //Get all unique names and put them into a set

    Map<String, String> idDict = new TreeMap<String, String>();
    Set<String> uniqueNames = new HashSet<String>();
```

12/29/22

Since we want our code to keep track of what labels (names of people) correspond to which IDs, we make a map called `idDict`. However, since the raw spreadsheet has many repeat names (how else would we calculate the frequency of collaboration?) We will also use a data structure called a set that stores unique items only and does not store duplicates.

```
for (int i = 0; i < allNames.size(); i++) {
    String nameStatus = allNames.get(i);
    String[] nameandstatus = nameStatus.split(":");
    String name = nameandstatus[0];
    uniqueNames.add(name);
}
```

This for loop adds each name to the set.

```
ArrayList<String> namesList = new ArrayList<String>();

//Get according indexes so you can assign IDs
for (String s: uniqueNames) {
    namesList.add(s);
}

for (int i = 0; i < namesList.size(); i++) {
    String name = namesList.get(i);
    String[] firstLast = name.split(" ");
    String tempfirst = firstLast[0];
    String templast = firstLast[1];

    if (tempfirst.length() == 1 && templast.length() == 1) {
        String first = tempfirst;
        String last = templast;
        String id = first + "." + last;
        idDict.put(namesList.get(i), id);
    }
```

```
        else if (tempfirst.length() >= 3 && templast.length() >= 4) {
            String first = tempfirst.substring(0, 3);
            String last = templast.substring(0, 4);
            String id = first + "." + last;
            idDict.put(namesList.get(i), id);
         }

         else  {
            String first = tempfirst;
            String last = templast.substring(0, 3);
            String id = first + "." + last;
            idDict.put(namesList.get(i), id);
         }
      }
      return idDict;
   }
```

While this code seems somewhat long and confusing, all these if and else statements simply create a new unique ID for each name. Each ID includes the first 3 letters of each first name (if there is more than a first initial), a period, and then the first 4 letters of the last name. `idDict` **returns something that looks like the following in structure.**

Key	Value
George Chapman	Geo.Chap
Richard Bonian	Ric.Boni

The second method statusDictBuilder processes the spreadsheet and returns a dictionary/map where the keys are the names of people (labels for Gephi) and their corresponding values are the status of the person as either a publisher, printer, author, preacher, or bookseller. This method is helpful for keeping track of the different Types of nodes present so that Gephi can use this information to color nodes according to their status.

```
public Map<String, String> statusDictBuilder(List<String> allNames){
```

```
        Map<String, String> statusDict = new TreeMap<String, String>();

        for (int i = 0; i < allNames.size(); i++) {
            String nameStatus = allNames.get(i);
            String[] nameandstatus = nameStatus.split(":");
            String name = nameandstatus[0];
            String status = nameandstatus[1];
            statusDict.put(name, status);
        }

        return statusDict;
    }
```

This for loop uses the list allNames we built to create a dictionary that is structured as follows.

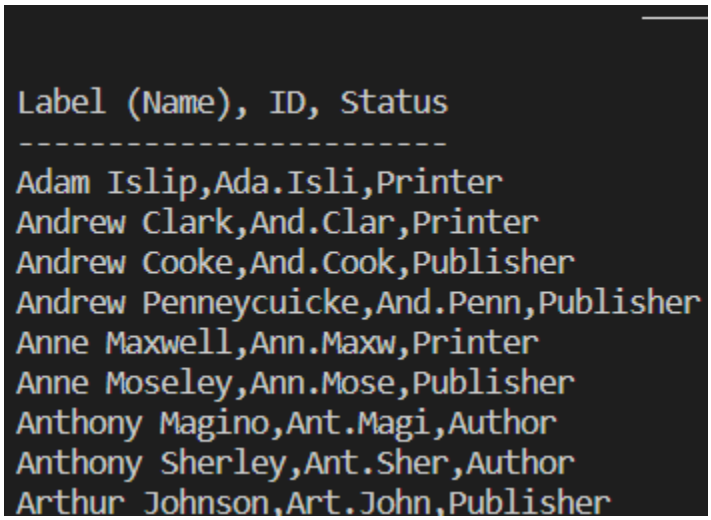
Key	Value
George Chapman	Author
Richard Bonian	Publisher

The third method printIDs generates the output of a node spreadsheet needed for Gephi with the data it prints organized with each line as the Label, ID, Type of an individual. The Type category corresponds to the status of someone as a publisher, author, printer, preacher, or bookseller. printIDs does this by using the existing dictionaries that were created by buildIDs and statusDictBuilder.

```
 public void printIDs(Map<String, String> idDict, Map<String, String>
statusDict) {
        System.out.println("Label (Name), ID, Status");
        System.out.println("------------------------");
        for (String s: idDict.keySet()) {
            System.out.println(s + "," + idDict.get(s) + "," +
statusDict.get(s));
        }
    }
```

12/29/22

This method simply prints out the dictionaries we built to the console and results in the following image:



The fourth method buildRelationships processes the spreadsheet and returns a dictionary/map where the keys are the names of people and the corresponding value to each key is a list of the people they collaborated with. This is necessary in order to create the edges that Gephi will want in a separate spreadsheet.

```
 public Map<String, List<String>> buildRelationships(ArrayList<String>
namePairs) {

        Map<String, List<String>> relationshipDict = new TreeMap<String,
List<String>>();

        for (int i = 0; i < namePairs.size(); i++) {
            String[] names = namePairs.get(i).split("&");

            if (! relationshipDict.containsKey(names[0])) {
                relationshipDict.put(names[0], new ArrayList<String>());
            }

            relationshipDict.get(names[0]).add(names[1]);

        }
```

```
        return relationshipDict;
    }
```

This method uses the list namePairs that we built to create that very dictionary. In order to avoid double counting relationships, we only add one person to the values list of one key since the edge, or the link between the two people, flows both ways.

The fifth method **printSourceTargetWeight** uses the prior dictionary built by buildRelationships to print the output of a Gephi edge spreadsheet. It prints line by line as the ID of one person, the ID of their collaborator, and the number of times they collaborated.

```
   //Will print a column of sources, targets, and the appropriate weights
given the relationships inputted
    public void printSourceTargetWeight(Map<String, List<String>>
relationshipDict, Map<String, String> idDict) {

        Set<String> sourceTargetWeightStrings = new TreeSet<String>();

        for(String s: relationshipDict.keySet()) {

            for (int i = 0; i < relationshipDict.get(s).size(); i++) {
                String name1 = s;
                String name2 = relationshipDict.get(s).get(i);
                int weight =
Collections.frequency(relationshipDict.get(name1), name2);

                String tempSourceTargetWeightString = idDict.get(name1) +
"&" + idDict.get(name2) + "&" + weight;

sourceTargetWeightStrings.add(tempSourceTargetWeightString);
            }
        }
        System.out.println("Source, Target, Weight of Relationship
(Frequency):");
```

```
System.out.println("-----------------------------------------------------");
;

        for (String s: sourceTargetWeightStrings) {
            String[] sourceTargetWeight = s.split("&");
            System.out.println(sourceTargetWeight[0] + "," +
sourceTargetWeight[1] +  "," + sourceTargetWeight[2]);
        }
    }
}
```

The result of this code is the following image:

```
Source, Target, Weight of Relationship (Frequency):
-----------------------------------------------------
Ada.Isli,Wil.Holm,1
And.Penn,J.Bel,2
Ann.Maxw,Dor.Newm,2
Ann.Maxw,Fra.Beau,1
Ann.Maxw,Joh.Flet,1
Ann.Maxw,Lan.Curt,1
Ann.Maxw,Rob.Robe,2
Ann.Maxw,Tho.Coll,2
Ann.Mose,Hum.Mose,1
Ann.Mose,Hum.Robi,1
```

And with that, we have everything we need for our Gephi spreadsheets and we do not have to manually count out the number of times two people collaborated. Instead, the code processes the raw data for us by calling all of these methods inside of DataCleaner's main method, which looks like this.

```
    public static void main(String[] args) {
        // csv file to read
        String csvFile = "C:/Users/Owner/Downloads/Master Spreadsheet-
Networks - RAW-TOTAL (For Code) (1).csv";
        DataCleaner trial = new DataCleaner();
        ArrayList<String> resultofAllNames = trial.readtoList(csvFile);
        NodesandEdges builderofIDs = new NodesandEdges();
```

```
        Map<String, String> idDict =
builderofIDs.buildIDs(resultofAllNames);
        Map<String, String> statusDict =
builderofIDs.statusDictBuilder(resultofAllNames);

        //Prints names, ID tags, and status(author, publisher, printer,
bookseller) delimited with comma (no ugly dictionary)
        builderofIDs.printIDs(idDict, statusDict);
        System.out.println();
        System.out.println("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
        System.out.println();



        NodesandEdges tryRelationships = new NodesandEdges();
        DataCleaner namePairsTester = new DataCleaner();
        ArrayList<String> namePairs =
namePairsTester.getNamePairs(csvFile);
        Map<String, List<String>> relationshipDict =
tryRelationships.buildRelationships(namePairs);

        NodesandEdges getSourceTargetWeight = new NodesandEdges();
        //Add something to mark each list as being Source, Target, and
Weight(based off of collaboration frequency)
        getSourceTargetWeight.printSourceTargetWeight(relationshipDict,
idDict);



    }
}
```

This code would work to generate network spreadsheets built in the raw format of the spreadsheet I used so long as the file path `"C:/Users/Owner/Downloads/Master Spreadsheet- Networks - RAW-TOTAL (For Code) (1).csv"` **was changed to be file path of the desired spreadsheet to be processed.**