Copyright
by
Erich Jurg Hauptli
2014

The Report Committee for Erich Jurg Hauptli
Certifies that this is the approved version of the following report:

# ProGENitor : An Application to Guide Your Career

APPROVED BY

SUPERVISING COMMITTEE:

Adnan Aziz, Supervisor

Peter Hofstee

# ProGENitor : An Application to Guide Your Career

**by**

**Erich Jurg Hauptli, B.S.**

**REPORT**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**Master of Science in Engineering**

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2014

This paper is dedicated to my wife Jessica and my son Lukas. Without their motivation, calming effect, and love I would never have completed this master's degree. Through them I derive all of my sense of purpose and meaning. Thank you for the years of continued support, as I know completing this degree has taken a huge amount of time and effort from our family.

# Acknowledgments

I would like to first and foremost thank Dr. Aziz for supervising my master's report. He has helped push me into areas I never though I'd venture and I truly value that growth. I'd also like to thank Dr. Hofstee for taking the time to be my reader; I hope he felt the time was well spent. Additionally, I'd like to thank Will O'Donnel for taking the effort out of LaTex. Finally, I'd like to thank my family and all of my many co-workers who have helped me along the way through the many years of my working towards my master's degree.

# ProGENitor : An Application to Guide Your Career

Erich Jurg Hauptli, M.S.E.
The University of Texas at Austin, 2014

Supervisor: Adnan Aziz

This report introduces ProGENitor; a system to empower individuals with career advice based on vast amounts of data. Specifically, it develops a machine learning algorithm that shows users how to efficiently reached specific career goals based upon the histories of other users. A reference implementation of this algorithm is presented, along with experimental results that show that it provides quality actionable intelligence to users.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Motivation

Navigating a career can be a difficult endeavor. Individuals and corporations not only have to keep up with a daily demands, but they also have to look to the future to advance, grow, and prepare for unseen demands. Knowing how to prepare directly impacts how successful an individual or corporation is in the future. Focusing on the wrong preparation and training wastes time, effort, and can impact motivation. This costs companies and individuals money, lost time, and missed opportunities. ProGENitor was designed to empower users with information about where they could take their careers and how to actually reach their career goals. It does this by processing vast amounts of data from a career database. This data is then turned into a career path graph that shows users how to reach a job and provides insights into the paths that are most likely to achieve the end goal. This vast amount of data exists today within social sites such as LinkedIN, but the data is not used in a fashion that empowers end users to make career decisions. ProGENitor fills this gap and can be used to assist individuals and companies in targeting efforts towards the most effective actions to achieving a career goal or advancement.

## 1.2 Project Overview

ProGENitor is built on the vision of providing end users with actionable data to make career decisions through the analysis of vast amounts of career data. It does this by consuming data from career databases, processing it using the algorithms presented within this paper and through an open source tool called Weka. The Weka tool set analyzes the data that is passed to it and draws complex associations between the data using predefined algorithms. The actual implementation of the algorithms used by ProGENitor will be demonstrated through a proof of concept project. A benchmarking study will be presented to further analyze the results. This paper will show that ProGENitor can present a user with a complete career path graph based off of a simple query. Additionally, it will show that the user will be able to dig deeper into the graph to obtain further insights into which actions within the graph have the most significant impact to reaching a career goal. Additionally, using the Weka tool, it will also be demonstrated that a combination of actions may also be required to achieve the user targeted career goal.

## 1.3 Sample Applications

ProGENitor is an application that can be setup to parse through a vast amount of data to provide insights into an individuals career. It can easily be integrated into large scale social platform such as LinkedIN or a smaller scale corporate database. This provides it with two core applications, which are discussed in detail in Sections 1.3.1 and 1.3.2.

### 1.3.1 Individual Career Planning

ProGENitor is an excellent tool to assist an individual in their own career planning. Implemented in a social networking framework, ProGENitor can use the vast amount of data in the site to provide individuals with guidance towards meeting a specific goal.

For example, consider Tom the engineer. Tom wishes to some day be the lead engineer on his own project. Using ProGENitor, he could pass a job title such as Chief Engineer into ProGENitor. He would then be presented with a graph showing how other users within the social network achieved this goal. The most common paths would be stand out in the graph and the significant details about each job or education vertex could be displayed upon request. Finally, through Weka analytics, certain combinations of decisions would also be presented if they had a significant impact in achieving Tom's goal of becoming a Chief Engineer. Tom could then use this information to make an educated decision about what actions might get him to his career goals fastest.

### 1.3.2 Skill Identification

A corporation could use ProGENitor to ensure that talent was always available to fill the jobs that were needed by the company. The company could add performance review data into the database and then start identifying traits about successful employees. This information could then be used to make hiring decisions. Additionally, managers could use the feedback from

ProGENitor to help guide their employees into gaining experience to aid them in moving into key roles.

For example, Carole, a manager at a large technology firm, has to counsel employees in their careers and fill an already existing job vacancy. ProGENitor would improve her hiring decisions by identifying what skills and traits make employees successful in her team and company. When she is preparing to help guide employees on their career paths, ProGENitor would help her provide concrete suggestions of actions the employee could take to achieve career goals. The ProGENitor results would benefit Carole's company by helping create better trained and more satisfied employees.

### 1.3.3 Financial Benefits

ProGENitor would financially benefit companies by increasing the total data they obtained from the user base. Data has become another form of currency in the digital age and there are many ways companies profit from it. If users see a direct advantage to adding to their profiles and updating data about themselves they will likely do so. This means companies and social networks will increase the amount of data collected and then be able to use it for other applications beyond ProGENitor. For instance, LinkedIN collects a vast amount of data about its users, but would likely benefit from having even more information, as they could then better target their services and advertisements towards their users.

## 1.4 Contributions

This paper provides four key contributions. First, it details the vision of ProGENitor, an application to empower users to make career decisions based on large data sets. Second, it provides the algorithms to generate the data used make these decisions. Third, it shows the results of these algorithms and discusses the presentation to the user. Lastly, the performance of the application is presented through a benchmarking study.

## 1.5 Report Organization

This report will be broken into five chapters, including this chapter. In Chapter 2, the project architecture will be further discussed, as well as the choices behind the technology stacks used. In Chapter 3, the algorithms and implementation will be stepped through in detail. Chapter 4 will present the results and present some software engineering metrics regarding the overall project. Finally, Chapter 5 will discuss the project as a whole, other related work, and future work.

# Chapter 2

# Architecture

## 2.1 High Level Architecture

ProGENitor is broken into several different pieces of code. To place a call to ProGENitor, the user must send a query and a query type when starting the application. This query is typically a job title that the user wishes to reach. The query is then passed to the database block which will fetch data from the database. The data in the database is loaded with synthetic data for testing purposes, but could easily be replaced with an actual database. The fetched data would then be passed to a block of code, that processes the data to produce a career path graph and then extracts the significant information about the graph points. Additional information is also extracted by a tool called Weka, which is a collection of machine learning algorithms for data mining tasks. Weka is used to provide more complex insights into the combinations of data points that may be relevant to reaching the queried goal. This data is then returned to the user in a data object that can be rendered by a user interface. See Figure 2.1 for a representation of this sequence.

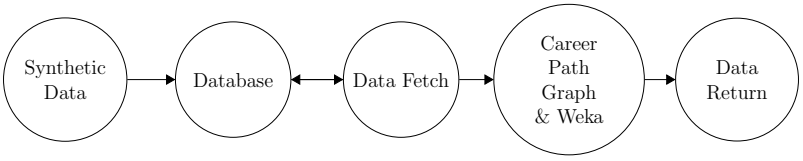



Figure 2.1: High Level Architecture

### 2.1.1 Overall Technology Stack

All of the code for ProGENitor is written in Java except for the synthetic data generation code which is written in Perl. Java was selected because the code needed to interface well with web applications and pages. Java is easily run through web interfaces and can easily pass data by passing JSON [3] Objects. Additionally, Java interfaces well with databases. As this project does a lot of database scraping this was very important.

## 2.2 Database Architecture

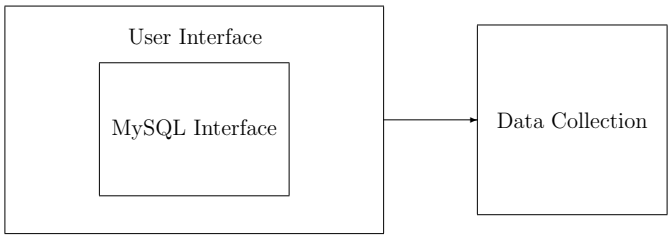



Figure 2.2: Database Block Diagram

As depicted above in Figure 2.2, the ProGENitor database code is designed around a MySQL database. The code is modular, so that any alternate database type can be inserted to allow for a quick transition to another

database architecture. Currently the database is broken into four different tables, but more can easily be added if necessary. These databases contain information on each individual user, user job history, user education history, and a listing of all of the headers for each database. The database SQL code allows for reading from, writing to, and creating the databases. The user wrapper creates a single point for the SQL interface that would need to be modified to support a change in the database architecture. Additionally, the wrapper provides some basic query commands to pull data from the database, based on a search field. It can also extract Meta data about the databases. The database is then pulled through a data collection package, which pulls data based on either a similar query field, vertex, or user id.

### 2.2.1 Database Technology Stack

The MySQL database was chosen as it is relatively easy to learn and manage. The goal was to quickly setup a database with minimal effort, so that the core of the project, career data analytics, could be focused on. In many instances, databases containing career information, such as LinkedIN have chosen to go the NoSQL route as these databases are better for unstructured data. For instance, LinkedIN uses a database called Sensei [15], which is a NoSQL based database. Many NoSQL databases are proprietary, where as the MySQL database follows a standard that the entire database community is familiar with. Thus, ProGENitor was built around a MySQL database, as many of the interface commands are similar for both NoSQL and MySQL.

Also MySQL is more widely known, standardized, and simpler to learn for someone new to databases.

## 2.3 Synthetic Data Architecture

To test and run ProGENitor, a database with data must be present. As there was not an existing one readily available, synthetic data needed to be generated and populated into a database. The SQL code allows for a file containing comma separated values to be loaded into the database. Thus, code to generate this file with meaningful data was required.

A Perl program was written to generate an uploadable data file. Each line in the data file first indicates the database the line should be loaded into and then the user id the data is associated with. Each individual user is assumed to have a distinct user id. The subsequent data in each row is built off a random selection from an array of data for each column. For example, when generating the university for a particular user's education vertex, a text file containing potential universities is loaded into an array and then the value is randomly selected from the array. This is done for each piece of data loaded into the database, such that the database looks like it contains real user data. The randomness can be controlled and particular elements can be weighted so they show up more frequently. Additionally, the paths and frequency users traverse through vertices can also be adjusted through variables and the content within the text files. This process will be stepped through in detail in Chapter 3.

### 2.3.1 Mock Data Technology Stack

Perl was the language chosen for the data generation for multiple reasons. The language excels at text manipulation. The regular expressions are excellent, as are the array processing capabilities. As Perl is a scripting language, it is not bound by the many rules surrounding an object orientated language. Thus, there was no need to keep track of variable types and the code required to do many things was much more concise. The language also very easily manipulates files. For all of these reasons, creating the data set to be loaded into the database could be done quickly and easily through Perl scripting.

## 2.4 Career Path Architecture

Once the databases are established and loaded, the real work can begin. ProGENitor takes in a request query and then using the career path modules, graphs out the similar career paths taken by others who traversed through the query point. ProGENitor uses basic graph theory [11] where it treats all significant events as vertices and all of the transitions between these events as edges. It also provides details about what was special about these individuals, as they went through each vertex of their lives. Then, through Weka, the code draws out the complex events that had the largest impact towards the the users passing through the query point. One advantage to using graph theory to present the results is that most of the end customers for ProGENitor will be very familiar with this type of data. As many of these customers will be

social sites, which essentially operate off of graph theory as well [13], the end users should be well accustom to the results and potentially may already have a method to render the data.

As depicted in Figure 2.3, the code is broken into four pieces. One piece gathers all of the vertices and defines the important edges between the vertices. Another piece of the code looks at the many different edges and attempts to order the vertices in a manner that they can be graphed from left to right without a lot of confusing edge crossings. A third piece of code extracts the data about each vertex and identifies the significant pieces of information that separate the users that reached the queried goal from everyone else who passed through the vertex but did not reach the goal. Finally, the last piece of code, uses Weka to extract the complex action or actions that had the greatest significance in causing the users to pass through the query point.

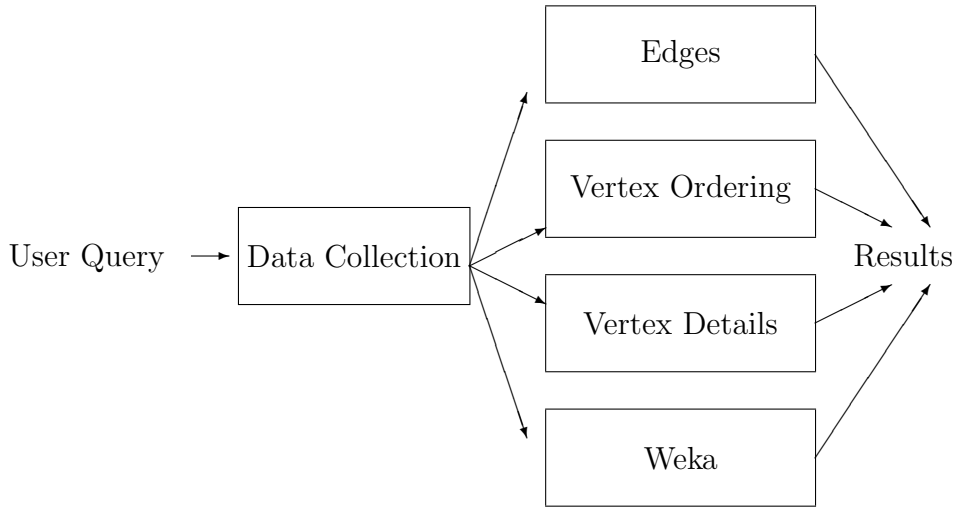



Figure 2.3: Career Path Block Diagram

### 2.4.1 Career Path Technology Stack

As detailed in Chapter 1, the code is written in Java. As the code for the graphing can only show users paths taken and important pieces of data along the way, Weka code was added to also derive insights based on combinations of data. Machine learning based on data can be very math intensive and complex. There are many different ways of looking at data sets. To simplify this, the Weka tool set was used in this project. The tool set has many different algorithms that can be easily implemented and applied. In the case of ProGENitor, Weka has only been applied to the education data; however, it could easily be expanded to analyze additional data, such as the data about jobs. Weka was chosen as it has a well designed Java API and is open source. One good alternative that could have been used in place of Weka is RapidMiner. Weka was chosen for the implementation as there is significant documentation surrounding both tool sets and RapidMiner's largest advantage, the graphical interface, is not applicable.

# Chapter 3

# Implementation

As outlined in the Chapter 2, ProGENitor is made up of several different sections. It contains a database framework to draw data for analysis. It has a tool to generate synthetic data for testing. The core tool contains algorithms to map out career paths and show important points along the path. It also uses Weka to draw some insights about the data that the mapping algorithms may fail to capture. ProGENitor would then return the results in an object such that a user interface could render the information to an end user.

## 3.1 Database

As ProGENitor needed a method to pull large amounts of data off the back end server database, a method had to be implemented that interfaced with the database. MySQL was chosen as it is open source, widely used, and fairly easy to quickly learn. Once the MySQL interface was established a wrapper was added such that another database method could be inserted without a significant work effort in the rest of the code. Then, through this wrapper, the many different function calls were implemented to collect the

data needed to generate the career path graph and derive any further insights through Weka.

### 3.1.1 Database Interface

To greatly simplify this work, a predefined library was added to the project. The library allows for easy access to a database for creating, reading from, writing to, and querying the database. In the case of ProGENitor, once the library was added, the code was very straight forward. Through a couple commands, the code established the database connection, ran the specified query or other command, and collected the returned data [6]. Using the library allowed the interface to quickly add in functions to create a database, collect query matches from the database, upload lines and files to the database, modify lines within the database, and even pull the entire database. With these functions in place, ProGENitor easily and quickly can access any defined database. In the case of ProGENitor, four tables were generated; one for user profile data, one for education data, one for job data, and finally one that contains the headers of the other tables. If, in the future, additional tables are needed, ProGENitor can easily add them. Additionally, as the SQL commands are standard commands, the interface can easily be replaced with another database interface or expanded upon by anyone familiar with a SQL language.

### 3.1.2 User Wrapper

As previously stated, it was desired that ProGENitor be setup such that it was easy to swap out the database interface with another interface. Thus, the user interface wrapper was written to call the various SQL commands. If in the future, the database needed to be changed, the work to do so would reside in adding the database interface and changing the user interface wrapper to point to the new database. The wrapper also adds in commands that make interfacing with the code a bit more clear. Commands such as add user, database setup, query matching users, and return headers all allow users working through other portions of code to understand what the function calls are actually doing and do not require the developer to necessarily understand the database interface commands. Then using these commands, ProGENitor can collect data that it passes along to be processed by the career path graphing and Weka packages.

### 3.1.3 Data Collection

With the wrapper and SQL interface in place, ProGENitor then implements a couple different calls to gather data to be analyzed. The first of these calls code that polls the data base for all users that match the query field value passed to the method. This method then returns the user IDs in a set for all of the users that matched the query. The next data collection method available does the same function, but instead returns all of the matching data in a list. The final data collection method available returns a list containing

all of the data associated with the set of IDs passed to the method. These methods are all very similar, but allow for easy data collection by the career path graphing and Weka methods.

## 3.2 Synthetic Data

As most end user's databases are not easily accessible and having control over the data in databases allows for better testing, generating synthetic data that is then loaded into a local database was chosen for ProGENitor development. ProGENitor is easily attached to any other databases, so this only speeds up the development process. To generate this data, a script was written that consumes various data files containing possible data values and then randomly selects from these values to populate the database. The number of users generated and other variables are also controllable within the script. Once the script completes it outputs a file containing all of the user data, which can then be uploaded to the database through the database architecture included with ProGENitor.

### 3.2.1 Data Files

To allow for easily updatable synthetic data, separate data files were implemented. This was done so that the values weren't embedded deep within the data generation code. There are two different types of data files. The first type simply contains a list of all possible values. These values are then simply loaded into an array by the data generation script. Then, the script

randomly selects from the array when it needs one of these values. The second type contains a listing of possible values dependant on a previous value. For example, in the line below, to get a Master's Degree in Circuits or Computer Systems, the user must first obtain a Bachelor's Degree in Electrical.

```
Electrical:Circuits,Computer Systems
```

Thus, the code will search the second file type for the line that meets the dependency. Once the line is found, it will load the possible values into an array and then randomly select from one of these values.

### 3.2.2 Random Selection

There are two places the code must randomly select data. The first is the data that is loaded for each vertex. This random selection simply places the data from one of the data files into an array and uses a standard random function to randomly select an array index value to pull the data from. This data is then applied to the individual user's vertex. When the vertex has all necessary data generated, it is loaded into the data file to be loaded into the database. In testing, to force a particular piece of data to occur more frequently, simply placing it multiple times into the data files will increase the frequency that it will show up in the user data.

The second place that code must randomly select data is when it is determining if a user enters a vertex or not on each pass. There are four

possible types of vertices that a user could enter during each loop. In each pass of the loop, they could potentially enter up to all of the vertices. These vertices are an undergraduate degree, a master's degree, a PhD, and a job. For each of these vertices, a chance value is assigned in the variables at the top of the script. Then essentially a ten sided dice is rolled at each vertex, this is done by loading values 1 through 10 into an array and applying the standard random function to the array. If the dice roll is greater than the predefined chance variable value, the vertex is entered and data is generated. When any vertex is entered, all educational chance variable values are incremented by 1. This means it is much less likely someone will get an advanced degree as their career progresses. Additionally, each educational degree level is currently limited to one degree and requires the previous level have been completed. All of these variables are adjustable in the code; so many different scenarios can be generated.

### 3.2.3 Code Flow

Once the data files and random data selection is understood, the code flow is relatively straight forward. The code steps through the data, section by section, generating data for each user and then stores it in a data file that can later be uploaded into the database. Figure 3.1 depicts this process.

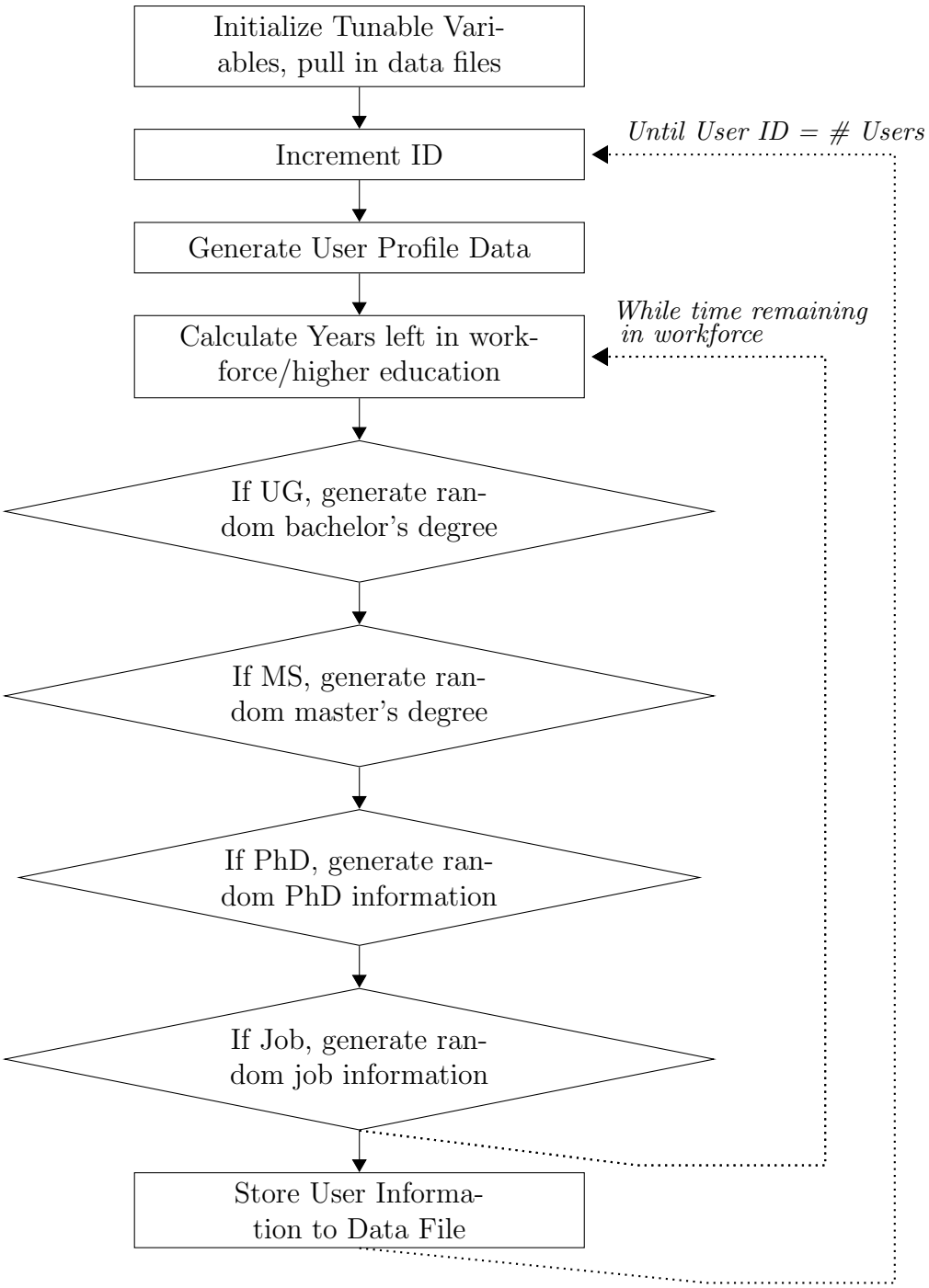



Figure 3.1: High Level Data Generation

## 3.3 Career Paths

The goal of the career paths module is to generate data, such that a web user interface could generate a graph of the career paths taken to reach the specified career goal. The vertex edge transitions along with the edge transition frequencies are returned to the user interface as objects. Additionally, the vertex ordering and information about each individual vertex is also returned in an object. This information can then be used to generate a graph depicting various ways of achieving a career goal.

An example of one of these graphs depicted in Figure 3.2, which shows various interconnected vertices that eventually arrive at the goal vertex. The vertices are arranged such that the user most likely travels from left to right, but the occasional infrequently traveled transition may flow in the reverse direction. In this example, the frequency that the edge is traveled is depicted through line thickness. This is done so that a visual representation is available to show approximately how many of the total returned users transition from one vertex to another. A dotted line would be the least frequently traveled edge, then a dashed line, a thin line, and finally the most frequently traveled edge would be the thick line.

Each vertex would then be able to display the individual vertex information upon user request; either through clicking on the vertex or through some other user action. Note, that ProGENitor does not limit the method in which the user interface is displayed; it simply passes back statistical information about the vertices, the transitions between each vertex, and the

individual data about each vertex. It is up to the web user interface developer to determine how the end product is rendered.

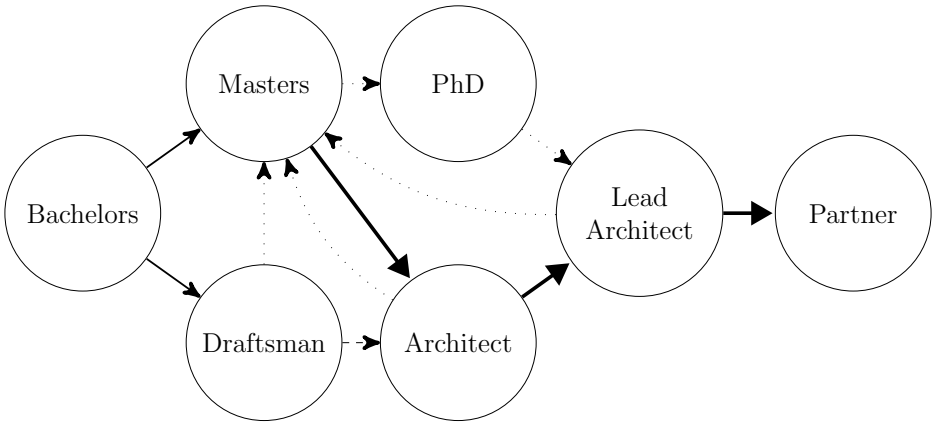



Figure 3.2: Career Path Graph

### 3.3.1 Graph Edges

The graph edges portion of code finds all of the vertices that users pass through and the order at which they pass through them. It then tallies the number of times all of the users pass along each transition path to allow for the career path graph to depict not only the point to point connections, but also how frequently that edge is traveled. The high level process to generate this graph interconnection data is depicted in Figure 3.3.

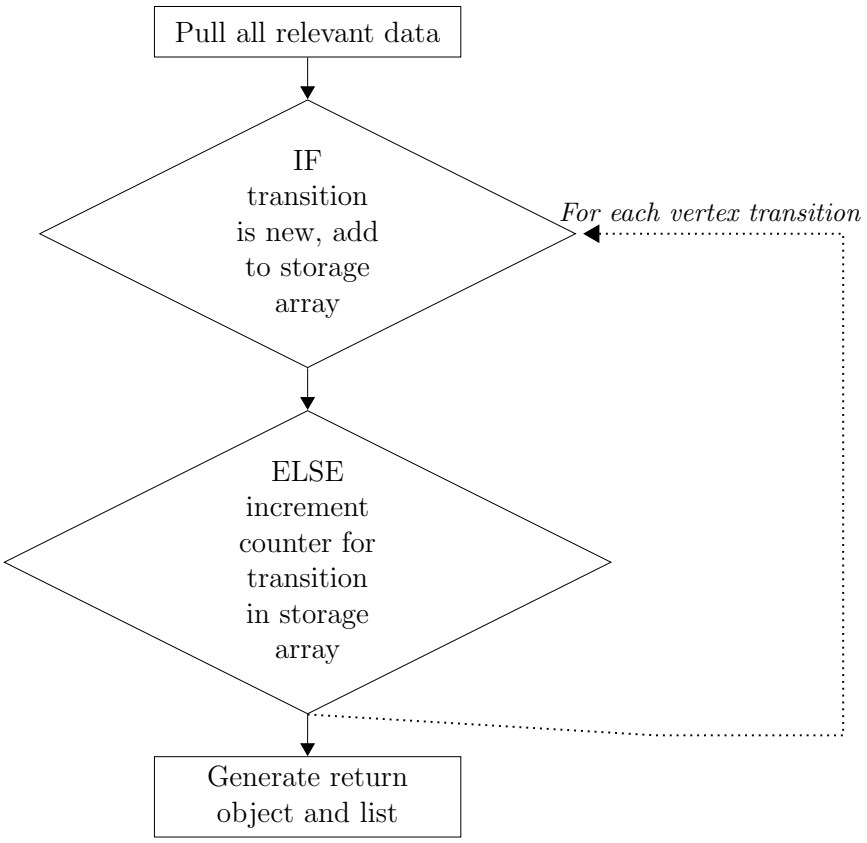



Figure 3.3: High Level Graph Edge Generation

The process flow in defining and counting these edges is listed in detail below:

**Graph Edge Generation:**

1. For each ID passed to edge generation module:

   (a) Pull job data and add it to the vertices list.

   (b) Pull education data and add it to the vertices list.

   (c) Set Min equal to Max Integer and Max equal to MIN Integer.

(d) For each element of the vertices list:

i. If date of data for element of vertices is less than Min and more than Max, store the data and set Min equal to date of data.

ii. After all elements of vertices list considered, add stored data to user list and set Max equal to Min.

(e) For each element of user list:

i. If A is NULL, set A equal to user element vertex name.

ii. Else set B equal to A and set A equal to user element vertex name.

iii. If edges list is empty, add B,A,1 to edges list.

iv. Else check if B,A exists in the edges list:

A. If it exists, increment the counter of the row.

B. If it does not exist, add B,A,1 to the edges list.

2. Push edges list containing all graph transitions and transition counts to an object containing an array.

3. Return both the list and the object.

### 3.3.2 Vertex Ordering

The vertex ordering portion of code sorts the vertices such that the major transitions flow in order from start to finish. It does this so that the flow of transitions can be graphed in a manner that is not overly confusing.

Figure 3.4 shows the high level process that the code follows to generate the vertex groupings. These groupings can then be fed to the end user interface to order the vertices in a fashion that shows the typical flow of careers that reach the destination goal.

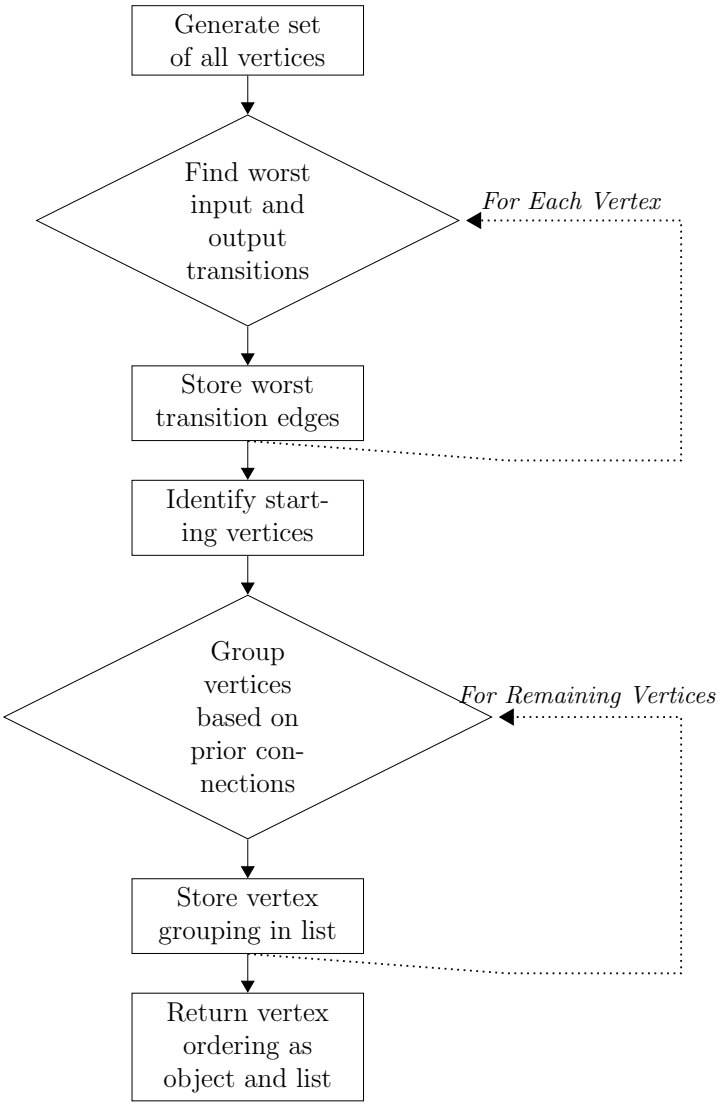



Figure 3.4: High Level Vertex Order Generation

The process flow in defining the vertex ordering for the vertices is listed in detail below:

**Vertex Ordering Generation:**

1. Generate set of all vertices
2. For each vertex in set:
    (a) Initialize transitional weight to 0.
    (b) For each element of the graph edge list:
        i. Check if vertex matches the input vertex.
        ii. Check if the number of transitions to the vertex is greater than the transitional weight.
        iii. If both checks are true; set the transitional weight to the current list line's number of transitions.
        iv. Also, if both checks are true; store this list line.
    (c) After the worst input transition is found for the vertex, store it in the heavy edges set.
    (d) Repeat this entire step for the output vertices.
3. For each heavy edge element, search the graph edge list for input vertices that are also destination vertices.
    (a) Any vertices not found are set as start vertices.

(b) Repeat this step for output vertices that are also starting vertices. Any vertices not found are set as ending vertices.

4. Add all the starting vertices to vertex 0 and add them to the vertex store set.

5. Add all vertices that are not starting vertices to the remaining vertices set.

6. Increment the group number to 1.

7. Until the remaining vertices set is empty, loop through the following steps.

   (a) For each vertex in vertex store, store all destination vertices in a set that vertex in vertex store transitions to.

   (b) For each destination vertex stored in the previous step, find all possible next destination vertices and check if they are contained within the set generated in the previous step.

      i. If one is contained within the previously generated set, remove the vertex from the set.

   (c) Add remaining vertices to next vertex grouping. Also remove remaining vertices from remaining vertices set.

   (d) Add the vertex group to the vertex return list.

   (e) Increment the group number.

(f) Replace the vertices in the starting vertices set with the vertices that were just added to a group.

8. Generate an object containing an array of the vertex groupings from the vertex return list.

9. Return both the object and the vertex return list.

### 3.3.3 Vertex Details

Presenting all of the potential information would overwhelm any user interface, so instead many of the details are buried within each vertex and can be queried by the end user, by selecting the vertex of interest. As each vertex contains additional details such as the place of employment or education, time spent at the school or job, or any other vertex relevant pieces of information; the data must be either gathered upon user request or each vertex must be pulled concurrently, as to not slow down the overall graph generation. Once the request is made, the data about the individuals who reached the goal vertex and the data about all of the users who did not, but still passed through a particular vertex are pulled. This is done because both the users who reached the goal and the users who did not need to be considered to determine what relevant pieces of data contributed to a user reaching the end goal. This data is then broken down into a statistic for both cases and compared against each other to determine if something occurred more frequently for the users who reached the goal vertex versus those who had not. This way any significant differences could be raised to the end user's attention as potentially important

steps to reaching the final goal. The high level process to generating this data is depicted in Figure 3.5.

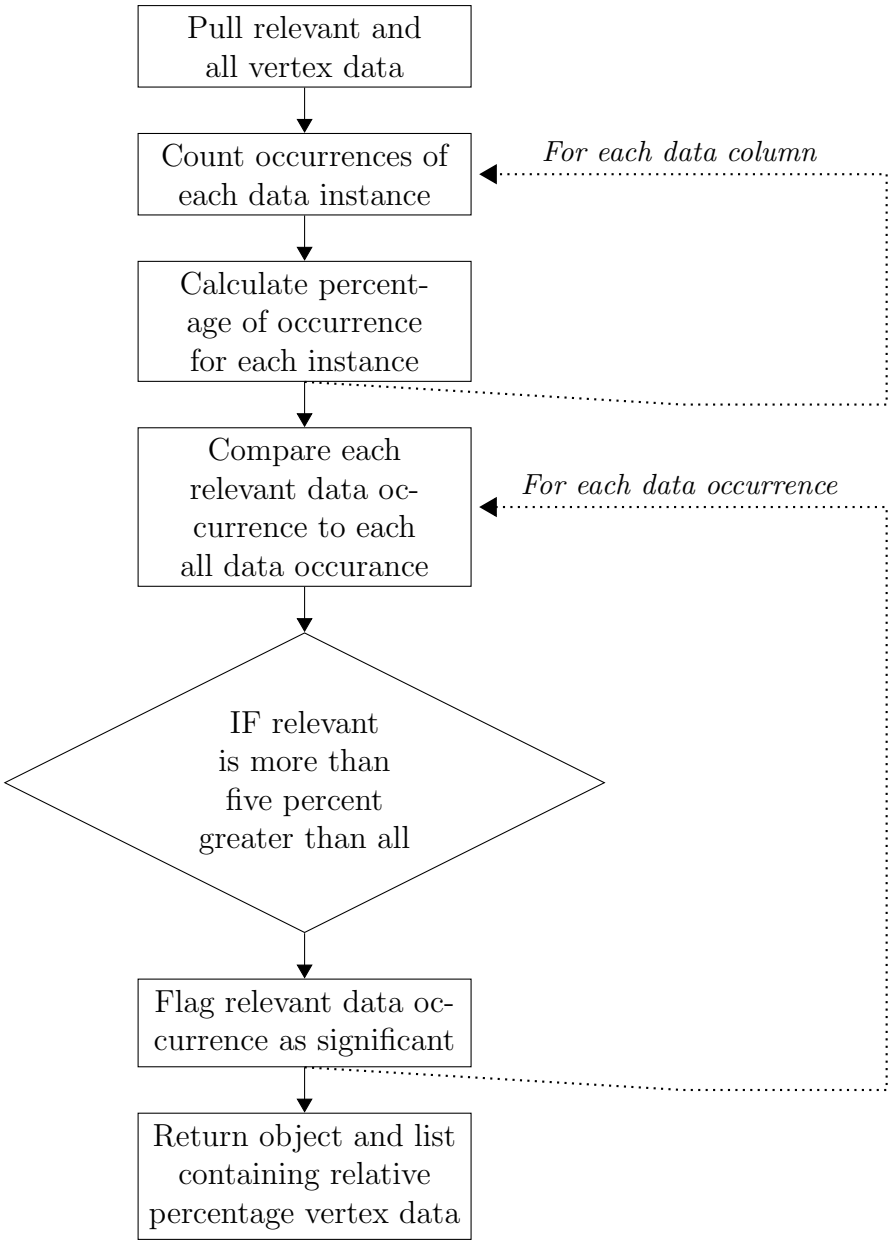



Figure 3.5: High Level Vertex Detail Generation

The process flow in defining the details and significant details for each vertex is listed in detail below:

**Vertex Detail Generation:**

1. Pull in profile list, tag each element as a profile, and then add the element to the complete list.
2. Repeat this for the jobs list and the education list.
3. Check each element of the complete list.
    (a) If the element contains the vertex that details are being pulled on, add the element to the relevant list.
4. Pull the headers associated with the vertex that details are being pulled from.
5. Pull all the data in the database for that vertex and store in the all vertex data list.
6. For each element of the complete list:
    (a) Split the element into columns and step through each column.
        i. Check if the column element is a start or end year and instead calculate the years spent at the vertex.
            A. If the end year is set to current, find the current year and then calculate the total years spent at the vertex.

ii. Add the column value to a set to obtain all possible values for the column.

iii. Step through the column counting each value instance to obtain a count for each different value.

iv. Calculate the percentage for each value in the column by diving the count by the total number of elements.

v. Push these values into the relevant list.

7. Repeat for each element of the all vertex data list

8. Compare the percentages for each element of the relevant list to the percentages from the all vertex data list.

   (a) Flag the column value for any instance where the relevant value's percentage exceeds the percentage for all the data by 5%.

   (b) Return this value as relevant so that it can be identified to the user as significant to the vertex.

9. Return the relevant list and an object containing an array of the same data.

## 3.4 Weka

One of the most popular ways of drawing insights from data through machine learning is by using a predefined library. This is because the library takes much of the technical effort out of the development. All of the math behind the machine learning is hidden behind the library and often there are nice user interfaces or APIs associated with the library. Typically, there are many different methods that can be called to comb through the data to extract insights and relationships about the data. In the case of ProGENitor the Weka library was chosen as it has an excellent API and access to many different methods. Choosing the method to extract information from the data requires some knowledge about the data itself. In this case, clustering was chosen as the data is mostly non-numerical data and the goal is to define some grouping that leads to the end goal. To extract the data, first ProGENitor must generate a data file to feed into Weka. Then Weka has to evaluate it with the chosen classification, which in ProGENitor's case is clustering.

### 3.4.1 Weka Data File Creation

Weka uses the .arff file format to feed data into the Weka tool set. The arff file contains two major sections. These sections are the header section and the data section [14]. The header contains the name of the relation, a list of attributes, and their types. The data section contains the data that will be

used for machine learning. A sample .arff file would look like the following:

```
@relation education


@attribute degree {PhD,Bachelors,Masters}
@attribute specialization {Electrical,Circuits,Analog,Computer
       Architecture,Digital}
@attribute goal {true,false}


@data
Bachelors,Electrical,false
Masters,Circuits,false
Bachelors,Electrical,true
Masters,Circuits,true
PhD,MSU,Digital,true
```

ProGENitor currently generates the .arff file containing just the educational vertices. One of the keys to getting quality insights out of Weka is controlling the data being fed into the tools. In this case, only the educational data is fed into the tool. This process could easily be replicated for additional insights. ProGENitor contains a method that follows the procedure detailed in Figure 3.6 to generate the data file that is later used by Weka.

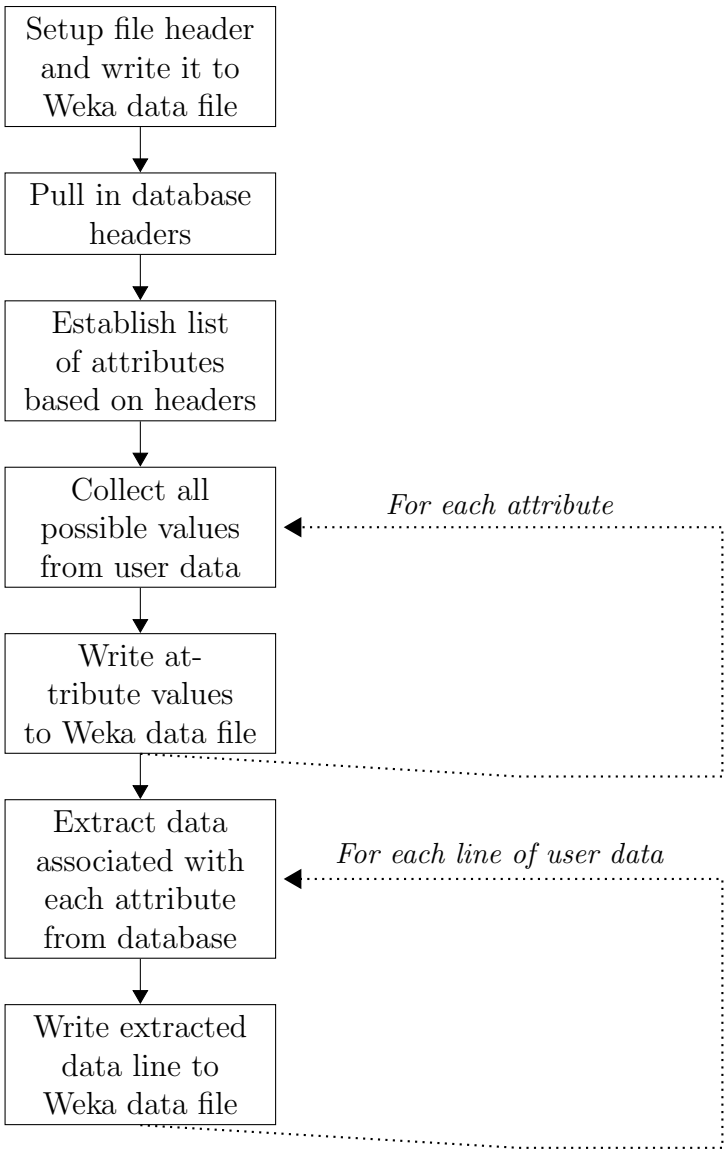



Figure 3.6: Weka Data File Generation

### 3.4.2 Clustering

One major advantage to using the Weka library is it takes complex code and makes it relatively simple. As seen in Figure 3.7, the process that is followed to analyze the data in the Weka data file is very simple and straight forward. Once the Weka library is imported into the project, the code is very quick to implement, as good documentation is available for the API [7]. The complex portion of work is then ensuring that the appropriate classification is applied and the data is parsed in a useful fashion.

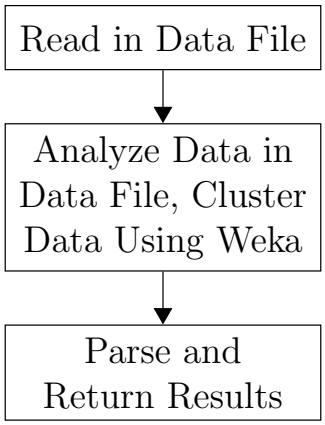



Figure 3.7: High Level Data Clustering

The Weka data analysis can take many different forms as there are many different classifications that can be applied. In the case of ProGENitor, EM (expectation maximization) clustering was chosen as it automatically determines the number of clusters required through cross validation. The algorithm that EM follows is shown in Figure 3.8 [2]. EM differs from other clustering algorithms in that it uses probability of cluster membership instead

of a distance method used by other clustering methods such as k-mean clustering [9]. EM starts with one cluster, then cross validates the data and applies the probability of cluster membership classification. It then calculates the log-likelihood for the set and if it increases, creates a new cluster and starts over. It repeats this process until the log likelihood no longer increases. The left over clusters will then be returned as the results.

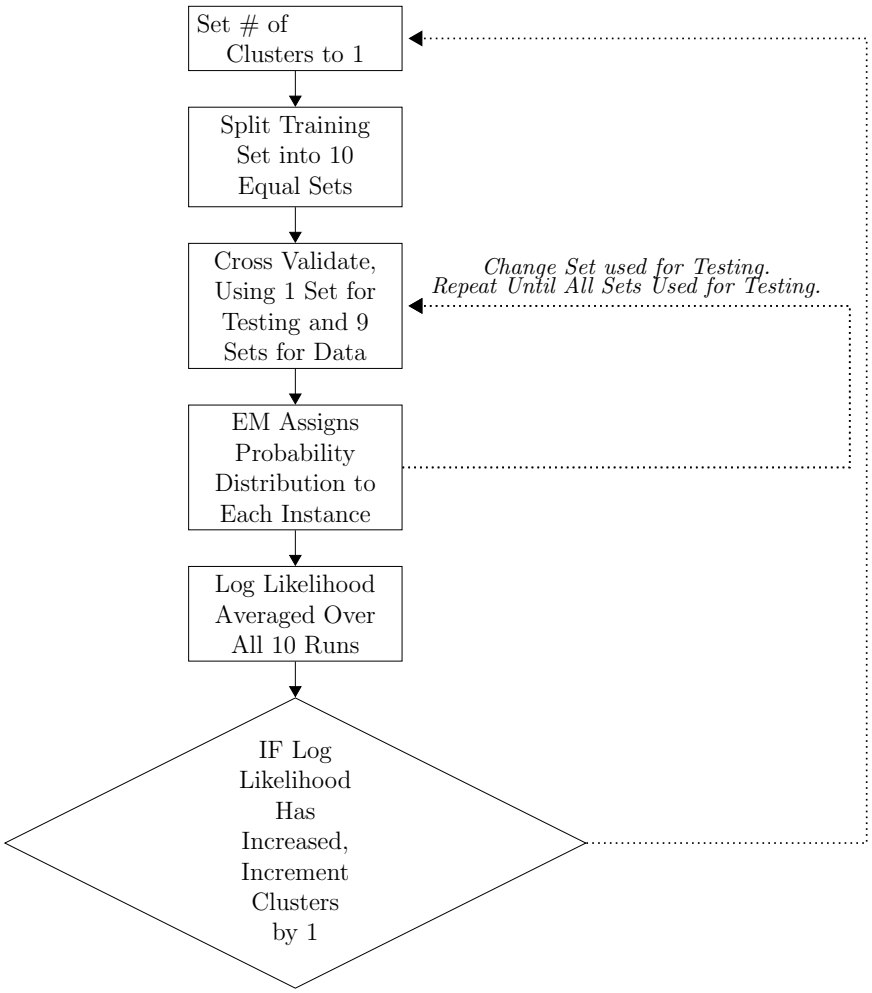



Figure 3.8: EM Clustering Algorithm

## 3.5 Fuzzy Matching

One of the challenges with processing the data of a database such as LinkedIN is that the data is free form. Although ProGENitor makes no attempt of matching similar jobs or other data points, it does attempt to account for minor spelling differences. Thus, if the users misspell a word or use a slightly different spelling, the similarities will still be captured. This fuzzy matching is done by using the Levenshtein distance algorithm [12] outlined in Table 3.1.

| Step | Description |
|---|---|
| 1 | Set n to be the length of s.<br>Set m to be the length of t.<br>If n = 0, return m and exit.<br>If m = 0, return n and exit.<br>Construct a matrix containing m+1 rows and n+1 columns. |
| 2 | Initialize the first row, s[0], to column number (starting with 0) .<br>Initialize the first column, t[0], to row number (starting with 0). |
| 3 | Examine each character of s (i from 1 to n). |
| 4 | Examine each character of t (j from 1 to m). |
| 5 | If s[i] equals t[j], the cost is 0.<br>If s[i] doesn't equal t[j], the cost is 1. |
| 6 | Set cell d[i,j] of the matrix equal to the minimum of:<br>a. The cell immediately above plus 1: d[i-1,j] + 1.<br>b. The cell immediately to the left plus 1: d[i,j-1] + 1.<br>c. The cell diagonally above and to the left plus the cost: d[i-1,j-1] + cost. |
| 7 | After the iteration steps (3, 4, 5, 6) are complete, the distance is found in cell d[n,m]. |

Table 3.1: Levenshtein Distance Algorithm

Once the algorithm calculates the difference between two words, it then checks to see if the difference is within the acceptable range. Currently this range is set to less than or equal to two. If the difference is acceptable, ProGENitor will consider the two words identical for matching purposes.

# Chapter 4

# Results

Upon running ProGENitor, the user will be returned a large object containing all of the data that is extracted from the database queries. This object will contain an object for the career path results and the Weka results. Each career path object will also contain three objects. These objects contain the graph vertex ordering data, the graph edge data, and the details on each vertex. Currently, all the data for each vertex is returned. In the future, to improve performance, switching the vertex data extraction to be upon request will speed up the overall career path graph results.

## 4.1 Career Path Results

Examples of the objects returned by the career path graph portion of the ProGENitor code are shown below. Each example does not contain a complete set of data as that would be too much to show in this report.

### 4.1.1 Vertex Edge Results

In the vertex edge object, an array of vertex edges is returned. Each element of the array contains a starting vertex and an ending vertex for the

transition. Additionally, the array element also contains a transition frequency. The transition frequency indicates how often the transition occurs. To prevent exposure of user data, these transitions are scaled to a value ranging from 0 and 10.


```
{"Vertex Connections":
{"vertex A":"Bachelors","vertex B":"Masters","transition frequency":6},
{"vertex A":"Masters","vertex B":"Circuit Designer","transition frequency":7},
{"vertex A":"Circuit Designer","vertex B":"Block Owner","transition frequency":8},
{"vertex A":"Block Owner","vertex B":"Design Owner","transition frequency":9},
...
{"vertex A":"Coder","vertex B":"Function Lead","transition frequency":0},
{"vertex A":"Function Lead","vertex B":"Masters","transition frequency":0}}
```


### 4.1.2 Vertex Ordering Results

In the vertex ordering object, an array containing the order which the vertices should be display is returned. Each element of the array contains a vertex name and the order number it should be displayed. Thus, the vertices with an order of 1 should be the first vertices displayed in the career path graph, then moving sequentially up, each vertex in the group should be displayed until the final vertex group is displayed. This will allow the graph to flow with minimum edges flowing in the reverse direction.

```
{"Vertex Ordering":
       {"vertex name":"Timing","order":"2"},
       {"vertex name":"Signal Integrity","order":"2"},
       {"vertex name":"Platform Chief Engineer","order":"7"},
       {"vertex name":"PhD","order":"5"},
       {"vertex name":"Entry Coder","order":"1"},
       ...
       {"vertex name":"Block Owner","order":"6"},
       {"vertex name":"Chiplet Designer","order":"1"}]}
```


### 4.1.3 Vertex Details Results

In the vertex detail object, an array containing all of the various vertices will be returned. Each vertex will be a nested object containing an array of data points. Examples of these data points are titles, companies, time spent at the vertex, and any other points of interest within the database. Each of these data points will be an object that also contains a nested array. This array will then contain data about each data point, broken down into the percentage of users who matched a specific piece of information for that data point. For example, shown below is a data point for the companies that users worked for when they worked at a Timing job. To protect user data, this is not shown as number of users, but as the percentage of users who spent time working for one company versus all users who spent time working at that particular

job. To avoid returning millions of entries, a threshold is set such that only statistically relevant data is returned and everything else is lumped into an "other" group. This "other" group would then contain the total user data that did not meet the threshold. Finally, an object containing any significant data points is also returned.

ProGENitor compares the users who reached the goal against all users who passed through the vertex to determine what was statistically different from the users who reached the target goal. These differences are listed in the significant data object. In the example below, the significant data point is that 100% of the users who passed through this vertex and reached the goal vertex worked for Verizon. This is significant because only 11% of the the users who spent time in this vertex worked for Verizon. Thus, working for Verizon in the Timing job is an important step to reaching the goal vertex. Significance is flagged whenever the users who reached the goal, had a data point occur 5% more than then everyone who passed through the vertex. This value could easily be modified by the company deploying ProGENitor if a greater difference was required for significance.

```
{"Vertices Data":
       {"Vertex Name":"Timing","Vertex Data":
       {"Data Breakout":
              {"name":"Other","value":"0.9174312%"}
              {"name":"Timing_all","value":"100.0%"},
       "Data Point Name":"title"},
       {"Data Breakout":
              {"name":"Verizon","value":"100.0%"},
              {"name":"Verizon_all","value":"11.33721%"},
              {"name":"Cisco Systems_all","value":"12.790698%"},
              {"name":"Boeing_all","value":"7.5581393%"},
              {"name":"Hewlett-Packard_all","value":"8.139535%"},
              {"name":"IBM_all","value":"7.2674417%"},
              {"name":"General Motors_all","value":"8.72093%"},
              {"name":"General Electric_all","value":"7.2674417%"},
              {"name":"Microsoft_all","value":"8.72093%"},
              {"name":"Intel_all","value":"8.72093%"},
              {"name":"Lockheed Martin_all","value":"11.046512%"},
              {"name":"AT&T_all","value":"8.430233%"},
       "Data Point Name":"company"},
       ...
       {"Significant":Verizon}}
```

### 4.1.4 Example Career Path 1

With the ProGENitor tool, several examples of functionality can easily be demonstrated. In the first example, consider a user that is interested in what it takes to become a partner in an architecture firm. The user would submit the query on partner and the career graph shown in Figure 4.1 would be returned.

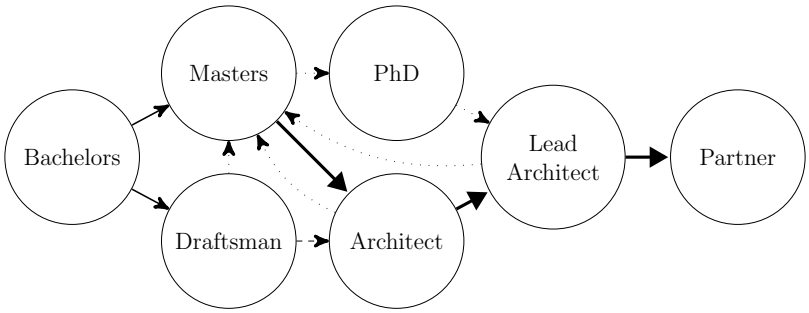
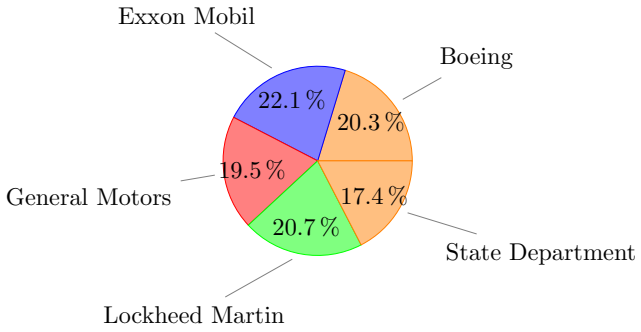



Figure 4.1: Career Path Graph

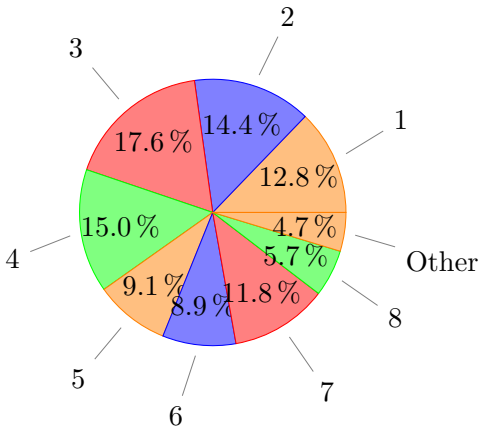
This graph quickly shows the user that a bachelor's degree is required. Next the users can see that a master's degree could help them immediately move into an architect role versus starting out as a draftsman. In either case, both options can eventually lead to the desired partner position, with no major indicator which one yielded a higher likelihood of achieving the goal. It also shows that it is rare for someone to return for a master's degree once they've entered the workforce and doing so later in your career can actually set the users back, if they've moved up past the architect position. Finally, very few people who reached the partner status also obtained a PhD. Although this is an uncommon path to reaching partner, it is an option that could be pursued.

Next, the user could select one of the vertices to pull up additional information about that vertex. The three pie charts below in Figure 4.2 show the information that would be returned if the user were to select the lead architect vertex. These charts show that there were five key employers for all of the users who reached partner. They also show that most of the partners were lead architects for less than five years, and it became increasingly rare to reach partner after this time. The data also shows that no particular city had lead architects getting promoted to partner more frequently. Thus, any lead architects looking at this data would know to reach partner they need to be focused on doing so within the five year window or they can expect their chances of doing so to diminish over time. Also, they should know that where and who they work for is not important, as long as they work for one of the five companies shown.

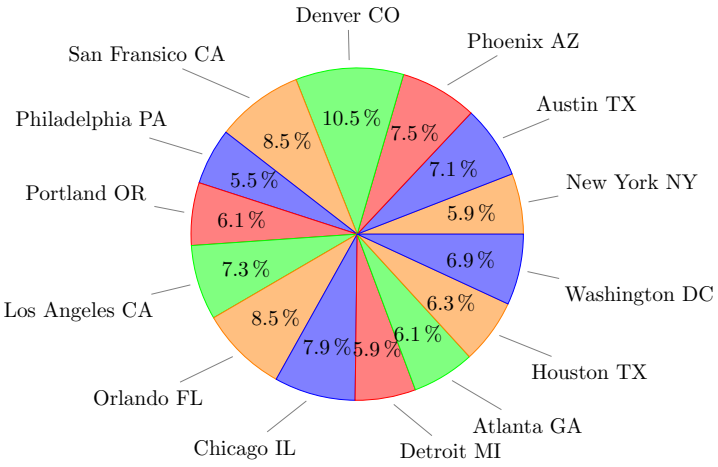
Alternatively, the user could click on the master's degree vertex, to see more information about these users. In doing so, the data generated immediately shows everyone who got a Master's degree did so in a single year with a specialization in Infrastructure and the only variation being in the school attended. In this case there were eight different schools attended, but none were attended at a more significant frequency than the rest. Thus, the user could immediately know if they wished to reach partner and do so by obtaining their master's degree, they need to get an Infrastructure degree within a year by attending one of these eight schools.




(a) Company




(b) Years Spent at Job




(c) Job Location

Figure 4.2: Lead Architect

The proper functionality of the algorithms can be demonstrated trough several modifications to the data fed into the synthetic data generation script. With the following modifications, it will be shown that the results ProGENitor produce align with the expected changes. First, a second specialization is added to Civil Engineering but it is occurs a third of the time. This is done by adding the following line to the Masters text file.

```
Civil:Infrastructure,Energy,Infrastructure
```

Next, an additional vertex, junior partner is added prior to partner. This is done by modify the titles text file. To make this change, the simple replacement of one line with two new lines was needed.

Remove: `Lead Architect:Partner`

Add: `Lead Architect:Junior Partner`

Add: `Junior Partner:Partner`

Finally, the likelihood of someone obtaining a master's degree was reduced by incrementing a probability variable by 2 in the data generation script.

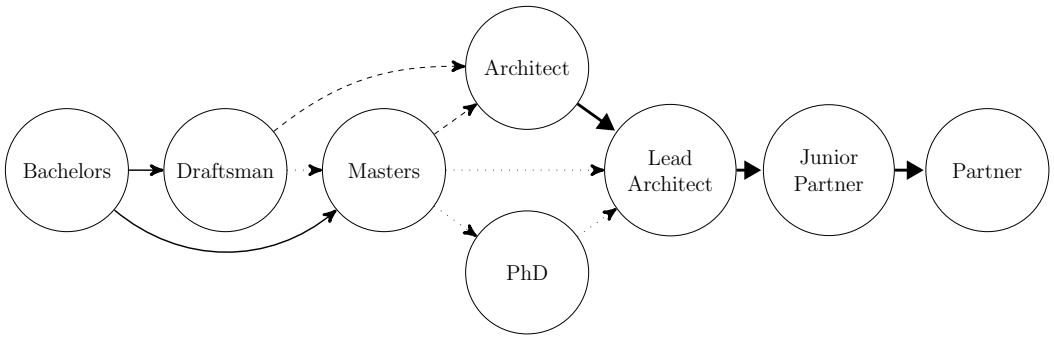



Figure 4.3: Modified Career Path Graph

With these changes in place, the new career path graph for achieving the partner position shows the expected changes. The new vertex step of junior partner is present and it also shows the reduction of users obtain a master's degree. In the case of users transitioning from a bachelor's to master's degree, it is not clear from the edges, but by looking at the data in the returned object, the frequency did decrease by 10%. In the detailed data for the master's degree vertex, the Infrastructure degree makes up approximately 2/3rds of the total degrees. This is shown in the object return for the masters vertex with the following text:

```
Data Breakout"
                {"name":"Infrastructure","value":"61.50794%"},
                {"name":"Energy","value":"38.49206%"}
```

Additionally, both Infrastructure and Energy would be returned as significant pieces of data, as they occurred much less frequently for the users who traveled through this master's degree vertex and did not reach the partner vertex.

### 4.1.5 Example Career Path 2

In the second example, consider another user who is interested in reaching the system chief engineering role. They would input this query into the tool and Figure 4.4 would be generated. From this graph, the user would quickly be able to see that obtaining an advanced degree was unnecessary to become a system chief engineer. They could then delve deeper into each vertex if they wanted to learn more about users who did the various jobs that also became system chief engineers.

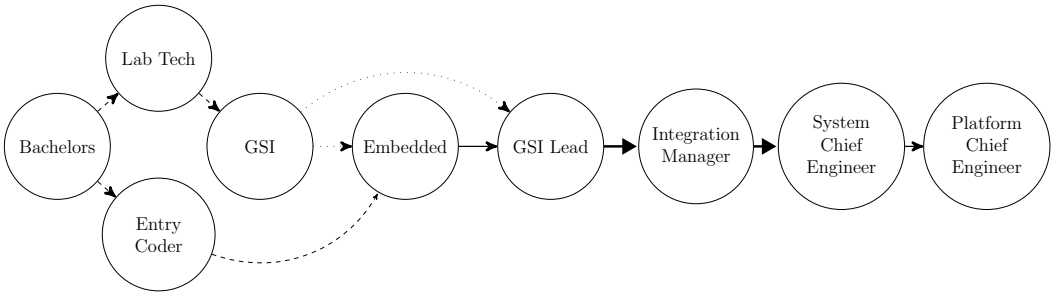



Figure 4.4: System Chief Engineer Graph

One thing that might also spark an interest in the user is the fact that any jobs beyond the queried job that the matched users also completed would be shown as well. In this case there was one of these such vertices, the platform chief engineer. From the edges it is clear that not all system chief engineers reached this job. If the user were interested then instead in the platform chief engineering role, they could re-run the query. They would then be presented with the career graph shown in Figure 4.5. This is obviously a much more complex graph, but it still yields the same capability of quickly showing users

complex career paths to a particular goal. In this case it shows that there are essentially three paths to this job. The first path was detailed in the initial query, the second path is through a design career path, and the third path is through an advanced degree. What is most notable about these paths is if the advanced degree path is taken, the initial jobs the users take don't have much impact, as long as it is within the career realm. The other notable thing is the most common path taken to getting to the platform chief engineering job is through the design path.


Figure 4.5: Platform Chief Engineer Graph

### 4.1.6 Career Path Performance

All of the work on this project has been done on a personal laptop with an 8 core i7 2.70GHz processor, a 500GB 7200 RPM 32MB Cache SATA 6.0Gb/s hard drive, and 16GB of DDR3 Memory. As ProGENitor would be run on a server instead of a personal laptop, it can be expected that the performance for all workloads would be improved. Still, the overall application run time would be impacted by both the number of users within the database and the total access times to the database itself. As the database was on a local drive, the access times were much less in these run times than they could be with a remote database.

To estimate career path graph performance, 10 cases were run, as shown below in Table 4.1. These 10 cases generate a range of matched users, total users, and number of vertices returned. By doing this, a rough estimate as to how long a query to ProGENitor might take can be ascertained. As seen in Table 4.1, an average query would take about 6.9 seconds, but might take much longer depending on the number of users in the database and the number that match the query.

| Case | Matched Users | Total Users | Data Collection | Edge Generation | Order Generation | Total |
|---|---|---|---|---|---|---|
| Platform Chief | 109 | 5000 | 708.9ms | 341.4ms | 74.4ms | 1.12s |
| Civil Degree | 2684 | 5000 | 11.0s | 12.2s | 8.1ms | 23.2s |
| Architect | 2330 | 5000 | 9.53s | 9.67s | 7.3ms | 19.2s |
| Circuit Designer | 675 | 5000 | 2.97s | 1.2s | 76.9ms | 4.3s |
| Worked For IBM | 260 | 5000 | 1.31s | 457.5ms | 85.6ms | 1.85s |
| Fission Degree | 260 | 5000 | 1.31s | 407.6ms | 6.3ms | 1.73s |
| Analog Degree | 24 | 5000 | 361.2ms | 66.8ms | 43.1ms | 471.3ms |
| Embedded | 55 | 5000 | 466.8ms | 269.5ms | 94.7ms | 831.1ms |
| Floor-planning | 49 | 5000 | 441.5ms | 106.5ms | 103.3ms | 651.5ms |
| Circuit Designer | 1401 | 10000 | 11.4s | 4.2s | 84.7ms | 15.7s |
| Minimum | 24 | 5000 | 361.2ms | 66.8ms | 6.3ms | 471.3ms |
| Maximum | 2684 | 10000 | 11.4s | 12.2s | 103.3ms | 23.2s |
| Average | 785 | 550 | 3.95s | 2.9s | 58.4ms | 6.9s |

Table 4.1: Career Path Generation Time

As seen in Table 4.1, more than half of the time that ProGENitor runs is spent in querying the database and pulling in the data to be processed. Then about 40% of the time is spent generating the vertex edges. Finally, determining the order in which to display the vertices runs in about 1 to 2% of the overall runtime. Thus, to improve or maintain performance most of the focus needs to be on the database pull. This is not an uncommon problem and

many people spend careers working on this problem. ProGENitor assumes that whoever deploys the tool would either have a smaller database or a database expert who could help refine the database accesses.

| Case | Matched Users | Total Users | Total Vertices | All Vertices | Average Vertex |
|---|---|---|---|---|---|
| Platform Chief | 109 | 5000 | 23 | 4.7s | 204.4ms |
| Civil Degree | 2684 | 5000 | 7 | 10.9s | 1.55s |
| Architect | 2330 | 5000 | 6 | 9.4s | 1.57s |
| Circuit Designer | 675 | 5000 | 34 | 9.48s | 278.7ms |
| Worked For IBM | 260 | 5000 | 40 | 6.8s | 170.1ms |
| Fission Degree | 260 | 5000 | 6 | 1.72s | 653.8ms |
| Analog Degree | 24 | 5000 | 12 | 3.94s | 328.6ms |
| Embedded | 55 | 5000 | 30 | 5.1s | 170.1ms |
| Floor-planning | 49 | 5000 | 18 | 4.6s | 155.2ms |
| Circuit Designer | 1401 | 10000 | 34 | 18.0s | 529.1ms |
| Minimum | 24 | 5000 | 6 | 1.72s | 155.2ms |
| Maximum | 2684 | 10000 | 40 | 18.0s | 1.57s |
| Average | 785 | 550 | 21 | 7.5s | 561.0ms |

Table 4.2: Vertex Detail Generation Time

In Table 4.2, the vertex detail generation performance is shown for the same 10 cases run previously. The table shows that if all the data was returned serially it could potentially add 18 seconds to the overall run time. This

would be too slow and unnecessary for the end user. By making each vertex call separate, the average return time on the vertex information would be about half a second prior to rendering the data. This is broken out separately because the assumption is that when ProGENitor runs it would either make this data available upon user request for each vertex or it would have to run each vertex call concurrently, such that all calls were completed prior to the career graph function call completing. This would have to be done to protect the user experience, as the total ProGENitor application call has to complete as quickly as possible. The method chosen would depend upon how much hardware would be deployed to support ProGENitor, as threads would need to be available to process the concurrent function calls. In either case, a method is available to allow for the vertex data retrieval without impacting overall system performance.

## 4.2 Weka Results

Weka analysis is run upon user request for additional insights. This is done due to the fact that it takes a significant amount of time to complete the analysis. Section 4.2.2 details the overall performance and explains why the Weka tool is not run automatically when the career path graph is generated. When running Weka, the results are not initially returned in an object. Weka returns the data in the following way shown below. Then ProGENitor parses the data and place the extracted insights into an object that can be returned to the end user as part of the overall object.

```
EM ==

Number of clusters selected by cross validation:  8
                                                Cluster
  Attribute        0        1        2        3        4        5        6        7
                (0.23)   (0.28)   (0.06)     (0)   (0.13)   (0.18)   (0.07)   (0.05)
degree
  PhD             1.0   1.0029     1.04        1      1.0      1.0     1.03    478.9
  Bachelors       1.0     2684      1.0        1   1.0015     1688      630      1.0
  Masters        2164      1.0      589        1     1252      1.0      1.0      1.0
  total          2166     2686      591        3     1254     1690      632    480.9
school
  Duke            273      458      1.0        1      1.0      1.0      1.0       24
  Stanford        1.0      1.0    164.3        1    163.7      231    239.9       48
  USC             277      426      1.0        1      1.0      1.0      1.0       38
  Berkeley        267      445      1.0        1      1.0      1.0      1.0       36
  ...
  Texas           1.0      1.0      1.3        1    202.7      442     1.05       23
  MSU           270.2    231.9      1.2        1    186.5      203      1.1       53
  MIT             254    155.7    219.4        1    144.6      157      154       77
  CalTech         1.0      1.0    206.4        1    166.6    206.9    238.1       40
  total          2175     2694      599       11      262     1698      640      489
specialization
  Fusion          1.2        1      211        1      1.5      1.0      1.0      1.0
  RF              1.0      1.0      1.0        1      1.0      1.0      1.0       20
  Magnetics       1.0      1.0      1.0        1      1.0      1.0      1.0       22
  Circuits        1.9      1.0     17.9        1      658      1.0      1.0      1.0
  Analog          1.0      1.0      1.0        1      1.0      1.0      1.0       25
  ...
  Digital         1.0      1.0      1.0        1      1.0      1.0      1.0       26
  total         2184.6   2704.4    609.2       21     1272   1708.2    650.4    498.9
goal
  true            1.0      1.0      1.0        1       91      110      1.1       14
  false         2164.6   2684.4      589        1   1162.2   1579.3    630.4    465.9
  total         2165.6   2685.4    590.2        2     1253   1689.2    631.4    479.9
=== Clustering stats for training data ===

Clustered Instances
  0   2163   ( 23%)
  1   2684   ( 28%)
  2    471   ( 5%)
  4   1369   ( 14%)
  5   1716   ( 18%)
  6    600   ( 6%)
  7    478   ( 5%)
Log likelihood:  -4.016
```

### 4.2.1 Explanation of Weka Results

In the example above, 212 user instances reached the end goal the query was searching for. This can be determined by adding up all the data in the goal equals yes row and subtracting the number of columns. Weka uses a minimum value of 1 for each element in the columns, thus the total instances of a value within a Weka data file would be the sum of the row minus the number of columns. When it states that 212 instances reached the end goal, this does not mean there were 212 users who reached the end goal, but instead there were 212 user educational instances. The Weka data file treats each educational instance as a new input, thus, when all is said and done for this particular data file there are 9481 educational instances with the database. This makes sense if you add up all the rows for bachelor degrees, master degrees, and PhDs. This math will also result in 9481 instances, assuming you subtract one for each value. One other odd piece about the data is the fact that the numbers are not integers. In the attempt to generate the various clusters, Weka assigns a probability to each educational instance that it belongs in a cluster. This means the math will get complex and not always place a value perfectly in only one cluster. This causes the values to come close to integers, but some times instances don't neatly fit within one cluster.

Once how the data is populated in the results and is understood; it can be used to draw some educated conclusions from the various clusters. As the interest is in the users who reached the end goal is the focus of the work, any clusters that are equal to 1 for a goal of yes can immediately be ignored.

In this case, that leaves three clusters. Looking through cluster column 4 for instance, shows a higher number of users who obtained a Master's Degree in Circuits and attended Stanford, Texas, MSU, MIT, or Caltech for this degree.

Cluster column 5 shows the same information, only instead the users obtained a bachelor's degree. The interesting thing of note between cluster columns 4 and 5 is the slight drop in users who reached the goal who obtained the master's degree. The drop is not significant which implies getting the master's degree is still very important for a user who wishes to reach the end goal. One other thing to note, in the above group of data, important information is not displayed as the total master's degrees does not match the ones displayed. This data was simply shortened for the report, but would normally be displayed in the Weka results. The same is true for the other clusters.

In the third cluster, column 7, only a few users show up as reaching the goal. This cluster shows students who obtained a PhD. The school from which they obtained the degree did not stand out in the cluster, but the degrees obtained did. The core degrees highlighted were RF, Magnetics, Analog, and Digital. It is worth noting however, that the results don't give us the ability to determine which one of these degrees is important as all 4 instances have more users than users who reached the goal while obtaining a PhD. In any case, due to the significant drop in users who obtained a PhD, it is clear that a PhD is helpful but not required in reaching the goal vertex.

### 4.2.2 Weka Performance

For the Weka runs, the same 10 previous runs for the career path performance testing were also used to estimate the Weka performance. In Table 4.3, a couple things can be observed. First, generating the Weka data file takes an insignificant time compared to the time it takes Weka to analyze the data. Second, the time it takes Weka to analyze the data is far too long to be suitable for an interactive request.

| Case | Matched Users | Weka Data Generation | Weka Analysis | Total |
|---|---|---|---|---|
| Platform Chief | 109 | 159.3ms | 280.5s | 280.6s |
| Civil Degree | 2684 | 155.7ms | 739.0s | 739.1s |
| Architect | 2330 | 210.5ms | 97.7s | 97.9s |
| Circuit Designer | 675 | 154.2ms | 441.1s | 441.3s |
| Worked For IBM | 260 | 154.1ms | 617.0s | 617.1s |
| Fission Degree | 260 | 186.1ms | 383.7s | 383.9s |
| Analog Degree | 24 | 187.7ms | 206.8s | 207.0s |
| Embedded | 55 | 155.2ms | 277.7s | 277.8s |
| Floor-planning | 49 | 158.0ms | 251.6s | 251.7s |
| Circuit Designer | 1401 | 280.0ms | 1211.1s | 1211.4s |
| Minimum | 24 | 154.1ms | 97.7s | 97.9s |
| Maximum | 2684 | 280.0ms | 1211.1s | 1211.4s |
| Average | 785 | 164.7ms | 450.6s | 450.7s |

Table 4.3: Weka Insight Generation Time

The data in Table 4.3 shows that the Weka request would have to be an option that a user specifically requests in addition to what ProGENitor typically runs. Wka would be one of the most likely pieces of ProGENitor to be sped up by running on a server because it is strictly computational and not limited by database accesses. That being said, the average run currently takes about seven and a half minutes, which would be far too long to ever be deployed to an end user. Thus, the server would have to significantly speed up the run over the development laptop used in this project to ever considering deploying Weka within the ProGENitor tool.

## 4.3 Engineering Metrics

When looking at preparing to write the code for this project it is good to look at about how much time will be required, how much code is needed to be written, and what challenges will be faced. This has been broken down by each major piece of the code below.

### 4.3.1 Databases

Writing the code for creating and pulling from the databases took about 30 commits. The code work took about 2 weeks or approximately 4% of a man year. The most difficult part of this code was simply learning and using the SQL database calls. In total the code was approximately 1100 lines of code.

### 4.3.2 Generating Synthetic Data

Writing the code for generating the synthetic data took about 13 commits. The code work took about 1 weeks or approximately 2% of a man year. The most difficult portion of this code was randomizing the data. In total the code was about 375 lines of code.

### 4.3.3 Career Path Graph

Writing the code for the career path graphing took about 11 commits. The code work took about 5 weeks or about 10% of a man year. This code has several areas that were particular challenging. One piece that was challenging, in the vertex edges code, was pulling only the worst case vertex transitions from the database. Another challenging part of the code, in the vertex ordering section, was ensuring that a vertex wasn't placed in a group if the prior vertex wasn't already in a previous group. Finally, in the vertex details code, pulling the significant data from the total pieces of data was also challenging. In total the code was about 1050 lines of code.

### 4.3.4 Weka Insights

Writing the code for the Weka insights took only 3 commits. The code work was quick due to the ease of implementing the API. It took less than 1 week to implement or 1% of a man year. The code was not difficult to write as the documentation gave clear examples on how to run Weka. The most challenging part was learning the Weka API and then choosing the analysis

method. In total the code was about 150 lines of code.

### 4.3.5 Total Code

Combining all this code, there was approximately 60 commits and about 2700 lines of code. All this coding took about 17% of a man year or about 2 solid months of coding. In reality, the project was worked on only part time and stretched out to about three and a half months. The most important decision in this project was focusing on the graph analysis of the returned data and on ensuring that the results and conclusions drawn from the results were valid. Extracting valid conclusions is extremely important, which is difficult to do because you have to look at both the users who reached a goal and those who did not.

### 4.3.6 Version Control

This project used GIT to manage the version control of the code. This helped greatly with managing the many aspects of the code, maintaining a change list, and reverting code back to functional states when something went wrong. If ProGENitor became a multi-person project, GIT would become increasingly more important as branching and merging would become very important. Finally, with a customer or multi-customer deployment, release trees would need to be implemented to avoid releasing development code or customer directed code to all customers.

# Chapter 5

# Conclusion

## 5.1 Summary

ProGENitor delivers upon the initial vision of this project by consuming large data sets to produce a career path graph that shows a user how other users have reached the queried goal. Additionally, it shows the user specific relevant actions that could be taken to reach that goal in the most efficient manner possible. As shown through this paper, the tool can take a very complex data set and provide quick insights into an individual's career aspirations and provide actionable next steps through detailed vertex, graph, and Weka analysis to help the user reach their goals. Using this tool can help an individual focus their career efforts and see the most efficient path to reaching their target.

## 5.2 Future Work

Although ProGENitor is already to a point that it could quickly be deployed, there are still a lot of improvements that could be made. For instance, creating a web user interface to visually demonstrate ProGENitor's capabilities would go a long way to helping sell it to future customers. Another

important work item would be to test and validate the software with several different database types such as a NoSQL database and several live databases. As ProGENitor has only been tested with synthetic data thus far, deploying it on an actual database could present some challenges that should be worked through. For example, LinkedIn's Sensei database interface should have similar queries, but there may be some differences. Additionally, the data returned may also take a bit of work to ensure it is in the proper format. Focusing on code parallelization and on optimizing database pulls could greatly improve performance. Further improvement could be gained by pulling the vertex detailed data only upon user request versus the current implementation, which pulls all vertex data at once.

Currently, ProGENitor only looks at the education data for Weka. Additional insights could be gathered by generating more .arff files to be fed into Weka. This could be expanded to also look at the job data or other aspects of the user data depending on the area the user was interested in. To make this feasible within a reasonable time window, Weka pulls would need to be attached to an advanced insight request by the user. Weka performance is far too slow to have it be part of the initial career mapping query.

The quality of the results could be improved by growing the user data to include information beyond education and employment. The profiles could be grown to include data about personality, work style, publications, or any other number of useful pieces of information. Finally, focusing on security and robustness by adding in some testing would also be valuable for a deployable

product. Again, ProGENitor could be deployed now with some minimal effort, but to deploy a quality well performing product additional work should be implemented.

## 5.3 Related Work

A lot of career planning focuses around an individual forming a mind map or taking quizzes to determine what they would like to do. Next, individuals are told to talk to people they know and look at existing jobs on job boards. Although these are valuable things that job seekers should do, it is not really a proactive method to ensure that the individual develops the skills they need to reach a desired job or career. ProGENitor takes a different approach in guiding users. By pulling all of the data in a career database, such as LinkedIn, ProGENitor generates insights into the most efficient path to reaching an individual's career goals. This is a much more accurate method than asking a friend or neighbor or searching job boards. Several companies are starting to take similar approaches to ProGENitor, but at the moment this is not a common method.

One example of a company doing something similar to ProGENitor is TalentGuard [1]. TalentGuard has a product that maps out career paths in a similar fashion to ProGENitor. First, the user feeds in a starting and end point. Then the tool generates the data in between these points. This requires the company to create the career paths and enter the data into the tool. It does not generate this data from existing users as ProGENitor does, thus it

requires some significant work on the company to deploy the tool.

Another example of a company doing something similar to ProGENitor is Mozilla. They wrote a program called Discover [4], which uses their existing OpenBadges [5] to generate career paths. Currently this product is just a prototype, but offers an alternate approach with a very polished and fun user interface. The tool expands upon what is looked at to include interests, experiences, education, and personality traits. It allows the user to view other career paths, so that they can model a career path based on another individual's careers. Unlike ProGENitor, it does not provide an aggregate of all of the users. This means the user either has to do a lot of comparison shopping, or they could end up mimicking someone who took an inefficient of rarely traveled path to the end goal. Additionally, it requires the user to use the open badges, which limits the application to an online community that uses the open badges tool.

A third example of a company that did something similar to ProGENitor was LinkedIn. They had a service called Career Explorer [10] which allowed students to explore different career paths. It gave the students the ability to visualize career paths, identify people in their networks who could assist with a career path, and provided other data from companies on the career path. One weakness of the tool is it did not draw from the whole breadth of the LinkedIn database. LinkedIn removed the tool when they went public as as they felt at the time their resources could be better invested elsewhere. [8] ProGENitor expands beyond students, draws from the full database to provide all profes-

sionals with advice, and should be more profitable due to a larger customer base.

All in all, though some tools do exist, there are not any that draw from the vast amount of data collected today as ProGENitor does. Additionally, though some of these tools offer some similar features to ProGENitor, all have significant disadvantages to what ProGENitor can offer. ProGENitor offers a scalable service that currently no other company has available.

## 5.4 Conclusions

If you are interested in doing a project similar to ProGENitor you should focus on the data that you wish to present to the end user. Focusing on relevant insights is key as you want the end user to draw valid conclusions. This is difficult as you have draw from data about both the users who achieved the career goal and those who did not. As the group that failed to reach a goal is vast, you must think about how to limit the data pulled to provide a result to the user within a reasonable time frame. In ProGENitor, the data set for the users who failed to reach the goal vertex was limited to all the users who entered a particular vertex of interest, thus conclusions could only be made about activity within that vertex and about frequency of edges traveled by those users who reached an end goal. This leads to one area of improvement that could be made to the ProGENitor results. It would have been useful to also know about the edges traveled for users who did not reach the goal, as it would allow the end user to draw conclusions about career tracks that might

be more successful but less traversed.

Once you have the algorithms defined, the coding is relatively straight basic. As a novice programmer, I was able to complete the project in about 3 months with no incoming knowledge about databases or Weka and a basic knowledge of Java programming. Using synthetic data vastly sped up the whole process and it gave me more control over testing my solution. If you wish to go further on this project, enhancing performance should be the main focus as that would allow for more insights to be batched and could also provide for a better user experience. The core areas for these performance enhancements would come from parallelization of the data analysis, optimization of the data fetches, and running the workloads on better computing hardware. A second key area of focus should be on adding a user interface. This would be required for the tool to really be used by an end user, as without it, the data would be too cumbersome for the user to consume.

All in all, ProGENitor delivered on the vision of providing users actionable data built off of large career data sets. Using the tool presents users with insights into how others have achieved a particular goal and what some of the key factors to achieving that goal were. Using this information the user could focus their efforts on the most important factors to achieving their goal and do so in the most efficient manner possible.

# Bibliography


[1] Career Pathing Features. http://www.talentguard.com/content/career-pathing-features#cppathbuilder. [Online; accessed 29-October-2014].

[2] Class EM. http://weka.sourceforge.net/doc.dev/weka/clusterers/EM.html. [Online; accessed 7-November-2014].

[3] Introducing JSON. http://www.json.org/. [Online; accessed 21-November-2014].

[4] Mozilla Discover. http://discover.openbadges.org/. [Online; accessed 29-October-2014].

[5] Mozilla OpenBadges. http://openbadges.org/. [Online; accessed 29-October-2014].

[6] SQLite Java Tutorial. http://www.tutorialspoint.com/sqlite/sqlite_java.htm. [Online; accessed 13-October-2014].

[7] Use WEKA in your Java code. http://weka.wikispaces.com/Use+WEKA+in+your+Java+code. [Online; accessed 11-October-2014].

[8] LinkedIn Career Explorer - No Longer Supported. http://help.linkedin.com/app/answers/detail/a_id/4640/~/linkedin-career-explorer---no-longer-supported, 2012. [Online; accessed 21-November-2014].

[9] Ethem Alpaydin. *Introduction to Machine Learning Second Edition.* The MIT Press, Cambridge, USA, 2010.

[10] A Diana. LinkedIn Launches Student Career Mapping Tool. `http://www.informationweek.com/applications/linkedin-launches-student-career-mapping-tool/d/d-id/1093177?`, 2010. [Online; accessed 29-October-2014].

[11] Reinhard Diestel. *Graph Theory.* Springer-Verlag, New York, USA, 2000.

[12] M Gilleland. Levenshtein Distance, in Three Flavors. `http://people.cs.pitt.edu/~kirk/cs1501/Pruhs/Spring2006/assignments/editdistance/Levenshtein%20Distance.htm`. [Online; accessed 11-October-2014].

[13] Jure Leskovec, Anand Rajaraman, and Jeffrey D. Ullman. *Mining of Massive Datasets.* Cambridge University Press, New York, USA, 2014.

[14] G Paynter. Attribute-Relation File Format (ARFF). `http://www.cs.waikato.ac.nz/ml/weka/arff.html`, 2008. [Online; accessed 11-October-2014].

[15] J Want. Introducing SenseiDB 1.0: an open-source, distributed, realtime, semi-structured database. `https://engineering.linkedin.com/open-source/introducing-senseidb-10-open-source-distributed-realtime-semi-structured-database`, 2012. [Online; accessed 14-September-2014].