Software Analysis Using Code Metrics

Darren Blanckensee — 1147279

***Abstract*—**

I. Introduction

SOFTWARE is being developed at an ever increasing rate. It is important when writing programs to follow good programming practices and to hold the code one writes to a certain standard so as to ensure a high quality product. Often when software projects are large with many different classes and files with thousands of lines to keep track of it becomes increasingly difficult to maintain a standard as it becomes impossible for developers to read all the lines of code and make sure that the quality requirements are met. One way to address this issue is to make use of software metrics. Software metrics are standards of measurement that provide developers with quantifiable measurements of various characteristics of the code they have developed.

One of the most simple metrics often used as an example in software is the lines of code metric (LOC) which measures how many lines of code exist in a program. This metric alone does not however provide the developer with useful information relating to the quality of the software and whether or not good programming practices are being followed. As So and so once said Puthequotehere pleasedarren. Other more useful metrics exist and the use of these in analysing software projects is the purpose of this report. All code in this project (including the metrics tool used) is written in Python.

The Chosen Metrics section details which metrics have been selected to allow for sound analysis of a code base that provides developers with useful information that could be used to maintain a standard of coding. Along with the selected metrics, explanations of each metric will be provided so as to explain the importance of each of the chosen metrics and how they should be used. The Code Base Analysis section covers the analysis of the authors own code base along with three major releases of the open source software, Freevo Media Library.

II. Understanding Chosen Metrics

The metrics being used in this project are the following:

- Linoes Of Code
- Cyclomatic Complexity
- Maintainability Index
- Coverage
- Halstead Metrics
- Dependencies

The tools used to calculate these metrics are Radon, Pylint, Graphviz, Webdot and Snakefood. Each of these metrics used are explained in the subsections below.

A. Lines Of Code

The lines of code metric simply measures the number of lines in a block of code. This metric was calculated using Radon. This metric is not very useful as an indicator of whether or not the code being analysed follows good programming practices.

B. Cyclomatic Complexity

The cyclomatic complexity metric, calculated using the Radon tool, is used to determine the number of distinct paths a program can take during running. It gives the developer an idea of how many decisions are being made by their code. There are various constructs and conditional statements or decisions that have an effect on the cyclomatic complexity of any program. These and their additive effects on the cyclomatic complexity are shown in the table below.

TABLE I
STATEMENTS AND THEIR EFFECTS ON CYCLOMATIC COMPLEXITY

Statement	Effect On Cyclomatic Complexity
If	1
Elif	1
Else	0
For	1
While	1
Except	1
Finally	0
With	1
Assert	1
Comprehension	1
Boolean Operators	1

An example (not the author's code base) and its cyclomatic complexity are shown in figure 1 and the paragraph that follows respectively.

Listing 1. Example code to explain cyclomatic complexity metric.

```
def careForDog(dog, isHungry, isHome):
    if isHungry == True:
        if isHome == True:
            dog.Feed()
        else:
            dog.Locate()
            dog.Feed()
    else:
        if isHome == False:
            dog.Locate()
        else:
            # Do nothing
```

While this code is simple and may not follow the best programming practices it is sufficient to explain how the cyclomatic complexity metric works. There are four ways this code can run. The first being if the dog is hungry and it is home then the dog will be fed. The second being if the dog is hungry but is not home then the dog will be located and fed. The third way is if the dog is not hungry and is not home then the dog will be located. The last way is if the dog is not hungry but is home then nothing is to be done. Because there are four ways this program can be run the cyclomatic complexity of this code is 4.

Radon was used to calculate the cyclomatic complexity of the various code bases analysed in this project. Radon's cyclomatic complexity measurement outputs a letter between A and F along with a score larger than zero relating to the number of decisions. A relates to a section of code that has less than five decisions while F relates to more than 41 decisions. The scores and letters are produced for each class, method and function.

The cyclomatic complexity command on Radon returns these letters and scores for each class, method and function in a section of code. Code with many decisions means many different ways the code can run which leads to high risk that code may not behave as the developer expects or intends. It is ideal to have a cyclomatic complexity score of less than 10 or B [1].

C. Maintainability Index

The maintainability index, calculated using the Radon tool, is used to determine how easily a section of code can be maintained. This metric measures how easy it would be to change this code in future. The maintainability index as a metric was introduced is calculated using the equation below.

$$MI = max[0, 100 * \frac{171 - 5.2\ln V - 0.23G - 16.2\ln L + 50\sin(\sqrt{2.4C})}{171}] \quad (1)$$

Where V is the Halstead volume, G is cyclomatic complexity, L is the number of source lines of code and C is the percent of comment lines in the program converted to radians. This equation is Radon's variation from the original maintainability index formula. In [2] Coleman et al. using a slightly different equation state that a maintainability index of above 85 is ideal and relates to code that is easy to maintain and update, while code with a maintainability index of between 85 and 65 is fairly maintainable and a maintainability index of less than 65 relates to code that is very difficult to maintain. Translating these values generated from the formula used in [2] to values generated from the formula in equation 1 above (used by Radon); A value of 85 and above using Coleman's formula corresponds to a value of 20 and above, a value of between 85 and 65 using Coleman's formula corresponds to a values between 20 and 10 and a value of below 65 using Coleman's formula corresponds to a value below 10.

For programs that are long term and will have many releases this metric is very important as it helps developers ensure that their code is written in a way that allows it to be changed and improved easily. This will help developers in the future tasked with working on that code perhaps to add new features or fix bugs. In [3] various versions of the Linux kernel are analysed using the maintainability index and it can be seen that with each new release the maintainability index actually increases. This shows how having a good initial maintainability index helps future developers add on to and sustain existing code.

D. Coverage

The coverage metric, calculated using the Coverage tool, can be used to determine the percentage of the code that was actually executed when the code runs. This metric is useful in determining code that is no longer used or is not a key piece of the code. Sometimes in a program added features may incorporate the functionality of old features therefore meaning the old functions are no longer used and are therefore purposeless and should be removed. This however is not the main advantage of using this metric.

The main advantage of using this metric is its ability to determine which portions of the code are being tested. If the coverage command is run with the test code as the argument then the coverage returned will be what percentage of the code has run as a result of testing. This allows the developers to see if they are testing all of the code and if not to see which sections of the code are not being run by the tests.

As testing is an extremely important part of software development, making sure that the tests are exhaustive is imperative as this allows developers to confidently say that there code is performing exactly is intended (so long as tests have been designed correctly). The coverage metric returns values of coverage for each file that is run when the program is passed to the coverage command. It is also able to tell the developer which lines are missed during execution.

E. Halstead Metrics

There are eight Halstead metrics that are used. These are calculated using a number of properties determined using the source code. These properties are:

- n_1 = Number of distinct operators
- n_2 = Number of distinct operands
- N_1 = Total number of operators
- N_2 = Total number of operands
- $n = n_1 + n_2$
- $N = N_1 + N_2$

These properties are used to calculate the eight Halstead metrics that Radon focuses on, vocabulary, length, calculated program length, volume, difficulty, effort, time to program and number of bugs. This report focuses on the difficulty and effort metrics. To understand the measurements it is important to first understand the terms operators and operands. Operators are any symbols that represent an action that has an effect on whatever is being acted on. Operands are what operators have effects on. Examples of operators are '+', '-', 'or' and even ''=='. Examples of operand are any variables or any constants and magic numbers in the code.

1) Difficulty
The formula for Halstead's difficulty as given by the Radon tool is:

$$D = \frac{n_1}{2} * \frac{N_2}{n_2} \tag{2}$$

This metric while often used as a measure of how difficult it is to read and understand the code later it also gives developers an idea as to how error prone the code being measured is. Seeing that the formula has that ratio of total number of operands and number of distinct operands, shows that the difficulty is higher if the same operands are used multiple times in the code, this is a characteristic of what is often called Spaghetti Code [5].

Errors are often made when operations take place. A simple example that can go unnoticed is using the '=' operator in an if statement as opposed to '==' which will result in behaviour that was not intended by the developer. This metric is useful for determining whether or not good programming practices have been used. Where good programming practices are used the difficulty metric will be low for example the code in listing 1 has a difficulty level of 0.75.

2) Effort
The formula for Halstead's effort as given by the Radon tool is:

$$E = \frac{n_1 N_2 N \log_2 n}{2n_2} \tag{3}$$

This metric gives developers an idea as to the level of mental activity required to code an already established algorithm. For example knowing how to transpose a matrix how much mental activity is needed to program a function that does it. It is not immediately clear what this metric measures without knowing the unit of measurement. The unit of measurement of the effort metric as defined by Halstead is elementary mental discriminations [4].

Given that a discrimination is a recognition of the difference between one thing and another this helps explain that the unit of measurement relates to the number of decisions the developer has to make with regards to the statements and conditions that could be used to accomplish a goal and how many decisions are made to determine the correct way in which to do it. High effort relates to code that is complex to implement and will most likely also be complex to maintain and change in future. The effort level of the code in listing 1 is 15.6.

III. Dependencies

The dependencies metric used in this report is Snakefood which is a graphic metric. It reads the source code and checks what classes, files, libraries etc. that a piece of code depends on. It then draws a graph connecting code that has dependencies to the code that it depends on. This can be done in two ways, using the –follow option makes sure that the code not only checks the immediate dependencies of the code in question but also checks for dependency within those code bases. This can cause the graph to be extremely saturated. Running the command without the –follow instruction just shows the immediate dependencies. This however is not a full picture and does not provide a good enough idea to developers of how heavily their code depends on other code.

One can also exclude internal files from being included in the graph meaning that only code from external libraries will be included in the graph. This is often helpful when determining how difficult it would be to change from one library to another if a different library that performs better is found. It is is important to have the ability to change from one library to the other as libraries are constantly changing and being improved.

IV. Analysis

This section covers the analysis of four code bases. The first being the authors own code developed to perform matrix multiplication of large matrices using the mapReduce techniques in MrJob. The second, third and fourth are versions 1.7, 1.8 and 1.9 of Freevo the media center written in Python [6]. This section aims firstly to verify the accuracy of tools used to calculate the various metrics by using them on the authors own small code base. Secondly this section aims to critically analyse and interpret the metrics calculated using the three releases of Freevo that have been downloaded.

A. Metric Tool Result Verification Using Small Code Base

The code in listing 2 is the authors code base. The purpose of this code is to use mapReduce to implement matrix multiplication of two matrices given in matrix market format in files outA1.list and outB1.list. The code reads the files and uses the indices and values given in the files to generate key, value pairs which are then used to map the inputs. These key value pairs are then used to reduce the entries and do the multiplication. The output of this code is a file with the resultant matrix in matrix market format. The code base depends on the MRJob, os and time libraries. The metrics are addressed in the sections below.

1) Lines Of Code
Running Radon for lines of code gives an output of 110. When examining the listing in the appendix it can be seen that this value is correct, proving that the radon tool returns the correct result.

2) Cyclomatic Complexity
The results of running Radon for cyclomatic complexity gives this as an output:

- C 5:0 matrixMultiply - A (5)
- M 7:1 matrixMultiply.sortbyJ - A (1)
- M 10:1 matrixMultiply.mapper - A (5)
- M 28:1 matrixMultiply.reducer - B (9)
- Average complexity: A (5.0)

The sortbyJ and mapper methods are analysed to validate the cyclomatic complexity metric calculation done by the Radon tool. Within the sortbyJ method there are no decisions in this method and therefore there is only one way this method can go and that is why the cyclomatic complexity is 1.

In the mapper method there are a number of decisions, the first being line 13 where it checks if the line has 3 entries or not. That is already two ways the program can go (i.e. cyclomatic complexity = at least 2). Within that decision is another decision on line 15 which determines if the line is from the first file or the second. This creates another way that the program can go (cyclomatic complexity = 3). The next decision is associated with the for loop in line 18 where there is a check whether k is within the range or not thus creating another path for the program to go down if k is within the range (cyclomatic complexity = 4). Then the next decision occurs within the for loop inside the else, on line 23. Again and in the same way as above this for loop creates another path depending on the variable i and whether it is in the specified range or not (cyclomatic complexity = 5). This analysis proves that the correct value has been returned for the mapper method.

3) Maintainability Index

The results of running Radon for maintainability index gives this as an output:

- Algorithm1.py - A (58.95)

This is according to Radon a high maintainability index which means changing this code in future would not be difficult. To verify this result it is important to calculate all necessary variables used in equation 1 for MI above. V Halstead's volume which for this code is calculated to be 444.6 as explained in the Halstead metrics verification subsection below. G in this formula is the cyclomatic complexity calculated above to be 5. L is the number of source lines of code which is 81. C is the percentage of the code that is comments. There are 14 lines of comments which makes the percentage comments 12.7%. Using equation 1 the maintainability index when calculated is 55.366. It is assumed that the difference (6% difference) comes from rounding errors. This verifies that the maintainability index metric calculation takes place correctly.

4) Coverage

The results of running the coverage tool gives a coverage of 78%. This is due to the fact that the contents of the outA1.list file and outB1.list file are representative of matrices of size 100x100 and there is a portion of the code that only executes if there is a column or row vector involved in the multiplication. This code is between line 79 to line 98. The actual amount of source lines of code in this section is 17. The actual amount of source lines of code in the whole program is 81. The percentage that is run is therefore $\frac{81-17}{81} * 100 = 79\%$. This verifies that the coverage metric is calculated correctly.

5) Halstead

The results of the Halstead metric calculations of interest are shown below:

- n_1 = 5
- n_2 = 47
- N_1 = 26
- N_2 = 52
- Difficulty = 2.76
- Effort = 1229.83

When counting the number of operators (n_1) it can be seen that there are indeed 5 distinct operators namely '+', '=', '==', '!=' and '*'. Furthermore upon review it seems that the difficulty metric captures the difficulty of writing/understanding this code quite effectively while the effort metric seems to over exaggerate the amount of elementary mental discriminations necessary to code this algorithm.

6) Dependencies

This code is dependent on three libraries as can be seen in the first three lines of code. The MRJob library the os library and the time library. When snakefood is run to determine dependencies the following graph is the result.

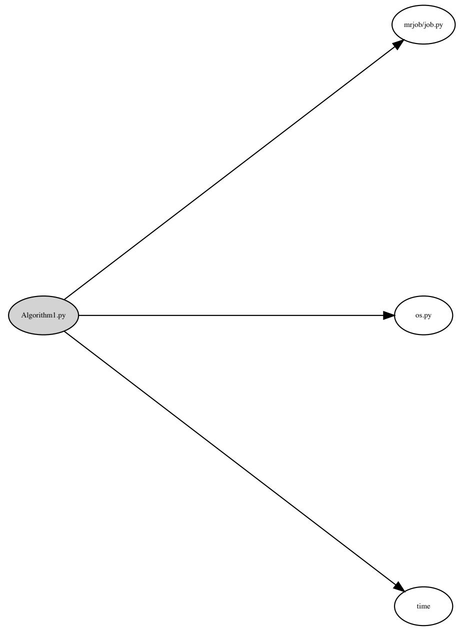

Fig. 1. Dependency Diagram.

This proves that the dependency metric behaves as it should.

B. Freevo Code Base Analysis

What follows is the analysis using the chosen metrics of three major releases of the Freevo Media Center application. For each metric a summary output is given and any interesting points are elaborated upon. Freevo is currently in a dormant state [6]. The reason for this is said to be because too many changes needed to be made to get the next version to work and because of errors made in the past. This analysis is done using the three latest versions available. These versions were chosen to determine whether or not the program could have been saved if the developers at Freevo had used these metrics to analyse their own code. This is discussed in the points of interest section.

1) Lines Of Code

Version 1.7 of Freevo contains a total of 76196 lines of code, version 1.8 of Freevo contains a total of 98718 lines of code and version 1.9 of Freevo contains a total of 121728 lines of code. A graph showing how the number of lines of code increases with each release is shown below.

2) Cyclomatic Complexity

a) Freevo 1.7

The average cyclomatic complexity of all the methods/classes and functions in all of the python files in the source code is 4.0219. This is, according to [1] a very good cyclomatic complexity. This average result however is extremely misleading as upon further inspection there are 73 methods/functions/classes with a cyclomatic complexity above 21 and 30 with a cyclomatic complexity of above 31 which is as Radon defines it alarmingly high and needs attention. There is one function that has a cyclomatic complexity of 102 which is unreasonably high and is most likely causing a number of bugs. This is the menu widget event handler method within the menu.py file.

b) Freevo 1.8

The average cyclomatic complexity of all the methods/classes and functions in all of the python files in the source code is 3.805. Which again is an extremely good score when it comes to cyclomatic complexity. It seems at a glance that the complexity has decreased which is counter-intuitive. However as before there are many classes/methods/functions with very high complexities. There are 94 classes/methods/functions with a cyclomatic complexity of higher than 21 and 39 classes/methods/functions with a cyclomatic complexity of higher than 31. The highest is once again the menu widget event handler method which has gone from a cyclomatic complexity of 102 to 116.

c) Freevo 1.9

The average cyclomatic complexity of all the methods/classes and functions in all of the python files in the source code is 3.630. Yet again it appears that the cyclomatic complexity overall has decreased which implies good coding and significant effort was put in to make sure the complexity was well managed however as is the case for the previous two releases this result is misleading. There are 121 classes/methods/functions with a cylomatic complexity of higher than 21 and there are 48 methods/classes/functions with a cyclomatic complexity of higher than 31. Once again both of these numbers have increased quite significantly. One noteworthy piece of information is that the menu widget event handler method has decreased significantly from 116 to 55.

A graph showing both the number of very high error prone unstable blocks (i.e. cyclomatic complexity of higher than 21 (D) and 31 (E)) over the three releases along with the misleading average cyclomatic complexity can be seen in figure 2 below.

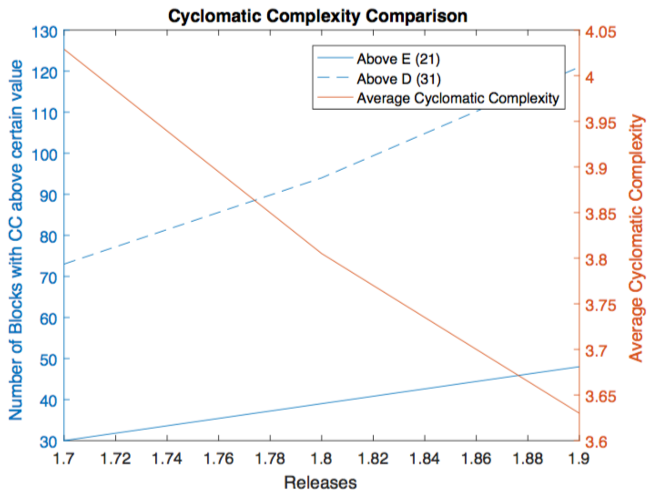

Fig. 2. Cyclomatic Complexity Graph.

This graph indicates that while average cyclomatic complexity may be a tempting metric to use it does not give a good idea about the actual complexity of the program until you investigate each class, method and function individually.

3) Maintainability Index

The maintainability index metric calculated using the Radon tool gives a maintainability index value for each python file. The average of these values was then calculated. Furthermore the names of the files with a maintainability index of less than 20 were written to a text file to compare for each release how the files that have low maintainability indexes developed throughout the releases.

a) Freevo 1.7

The average maintainability index for version 1.7 was 61 which as Radon states is very high and so should mean that the code is easily maintained. Upon further analysis when counting the number of files with a maintainability index of less than 20 (Medium to low maintainability index relating to code that is hard to maintain) it is seen that there are eight files in this category. With a total of 270 files this means the percentage that are difficult to maintain is 2.96% which is reasonable.

b) Freevo 1.8

The average maintainability index for version 1.8 was 60 which shows that there has been a very small decrease in the maintainability index however as Radon states this is still high means that the code is easily maintained. Once again however, upon further analysis when counting the number of files with a maintainability index of less than 20 i.e hard to maintain it is revealed that there are 13 files in this category. With a total of 329 files this means the percentage of files that are difficult to maintain is 3.95% which is once again reasonable however it is important to note that this has risen by 1% from version 1.7 to version 1.8 indicating that files that are originally hard to maintain only become harder to maintain as time goes by and changes to these files are made.

c) Freevo 1.9

The average maintainability index for version 1.9 was 59. The overall maintainability index continues to decrease with

TABLE II
TABLE SHOWING HALSTEAD METRICS FOR THE DIFFERENT RELEASES OF FREEVO

Freevo Version	1.7	1.8	1.9
n1	7.536	7.619	7.932
n2	69.74	76.23	81.43
N1	47.02	50.60	54.19
N2	90.41	97.37	104.63
Vocabulary	77.28	83.85	89.36
Length	137.44	147.97	158.82
Calculated Length	510.12	567.35	617.52
Volume	981.68	1076.28	1176.96
Difficulty	4.831	4.85	5.089
Effort	8195.5	8949.78	10356.41
Time	455.30	497.21	575.35
Bugs	0.3272	0.3587	0.3923

no indication that it will ever turn which suggests that this program is not sustainable unless significant work and effort is put in to redesign the problematic files. For this release the number of files with a maintainability index of less than 20 is 18. With a total of 385 files this means the percentage of files that are difficult to maintain is 4.67%. This percentage has once again risen from the previous version.

4) Coverage

a) Freevo 1.7

b) Freevo 1.8

c) Freevo 1.8

5) Halstead Metrics

The Halstead metrics were calculated using Radon. Each files Halstead metrics were calculated individually and then averages were taken. The table below shows the Halstead metrics for each release.

The table shows that with each successive release all the metrics increased. The metrics of interest in this report are the difficulty and effort metrics. The difficulty of the authors code base was 2.76. The average difficulty of all files in version 1.7 is 1.75x the difficulty of the authors code. The average difficulty of all files in version 1.8 is 1.76x the difficulty of the authors code. The average difficulty of all files in version 1.9 is 1.84x the difficulty of the authors code. This suggests that the code is notably difficult to read and understand and also that it is prone to errors. The difficulty increases with each release. This is to be expected as with a high initial difficulty it only becomes harder to interpret and more error prone when more code is added. This level of difficulty also suggests that there could be spaghetti code in a number of these files.

The error metric is above 8000 for all three releases. This is high and indicates that this code required many elementary mental discriminations and will require many more elementary mental discriminations if any additional functionality is to be implemented as can be seen by the significant increase in effort over the three releases. There were 754 more elementary mental discriminations in release 1.8 than 1.7 and there were 1407 more elementary mental discriminations in release 1.9 than there were in 1.8. This increase should and does result in the increase in difficulty shown in the table.

The increase in Halstead metrics while expected is in some cases like effort and volume quite high which leads to the conclusion that this code is and will continue to be difficult to understand and maintain.

6) Dependencies

The dependencies were calculated using the Pylint tool. The Pylint tool prints out the names of the files/libraries that each python file depends on. Code was written that then counts the number of dependencies for each file. A table showing the number of dependencies for each release is shown below.

TABLE III
TABLE SHOWING HOW NUMBER OF DEPENDENCIES CHANGED OVER TIME

Freevo Version	1.7	1.8	1.9
No. of Dependencies	201	227	312

This table shows that as the program advanced the number of dependencies increased quite significantly. In the appendix graphs of the dependencies for each release are shown which show just how complicated the relationships between the files are.

V. POINTS OF INTEREST

This section deals with a number of key points that stand out and could have been used to help the developers at Freevo address issues that eventually led to the current dormant state of Freevo. The first point of interest is that there are a large number of blocks of code with cyclomatic complexities higher than 31 which means that a large percentage of the code is alarmingly complex. Having such complex code means that the chance of having errors is larger and noticing this early on perhaps in version 1.7 would have meant that the developers could have put effort into finding the errors and reducing the cyclomatic complexity.

The second point of interest is the number of classes with a maintainability index of less than 20 which relates to code that is notably hard to maintain. It makes sense that code that is initially hard to maintain remains hard to maintain and becomes more and more difficult to maintain as time goes by. Files that had high initial maintainability indexes saw a climb in maintainability index with each successive release. If this was noticed by the developers they would have been able to identify the problematic files and address them in a way that made them more maintainable.

The third point of interest is the effort metric calculated as part of the Halstead metrics. The average value for effort over each successive release was extremely high which means that anyone who tries to view and understand and change this code in future would have to make many elementary mental discriminations. This means that future development of the code is very difficult especially if its new developers coming in that are not familiar with the code.

These are all the reasons that have led to Freevo being dormant. Its high cyclomatic complexity means potential for many bugs that would be very difficult to find and fix. Its number of classes with low maintainability index means that the code gets harder and harder to maintain and develop as time goes by and new releases are made eventually leading to code that is no longer maintainable and therefore ends up in a dormant state as has happened with Freevo. Having high effort metrics also means that anybody who would have been interested in continuing the Freevo program would experience extreme difficulty in trying to understand the code and add to it. If these metrics were measured at all steps along the way from the beginning of Freevo then it would have been easier to make sure that the code was held to certain standards that would have increased its lifespan and made it a better program all around.

VI. Conclusion

References

[1] Thomas J. McCabe, ”A Complexity Measure”. IEEE Transactions on Software Engineering, Vol. SE-2, No.4, December 1976.

[2] Don Coleman, Dan Ash, Bruce Lowther, Paul Oman, ”Using Metrics to Evaluate Software System Maintainability”. Computing Practices, August 1994.

[3] Lawrence Gray Thomas, ”An Analysis of Software Quality and Maintainability Metrics with an Application to a Longitudinal Sudy of the Linux Kernel”. Dissertation Submitted to the Faculty of the Graduate School of Vanderbilt University in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science, August, 2008.

[4] Rafa E. Al Qutaish, Alain Abran, ”An Analysis of the Design and Definitions of Halsteads Metrics”. Proceedings of the 15th International Workshop on Software Measurement, September 12-14, 2005, Montreal, Canada. pp. 337-352.

[5] Marwen Abbes, Foutse Khomh, Yann-Gael Gueheneuc, Giuliano Antoniol, ”An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, On Program Comprehension”. 15th European Conference on Software Maintenance and Reengineering, 2011.

[6] Dischi, ”Freevo Website”.
http://www.freevo.org. Last Accessed 18 April 2018.

APPENDIX

1) Dependencies

Shown in the figures 3-5 below are dependency graphs generated by Snakefood that show how the different files that make up each release of Freevo depend on each other and on external libraries. These graphs are extremely saturated and therefore a high level analysis approach is taken. These graphs are also included in the Appendix on a larger scale.

a) Freevo 1.7

The dependency graph for Freevo 1.7 is shown in figure 3.

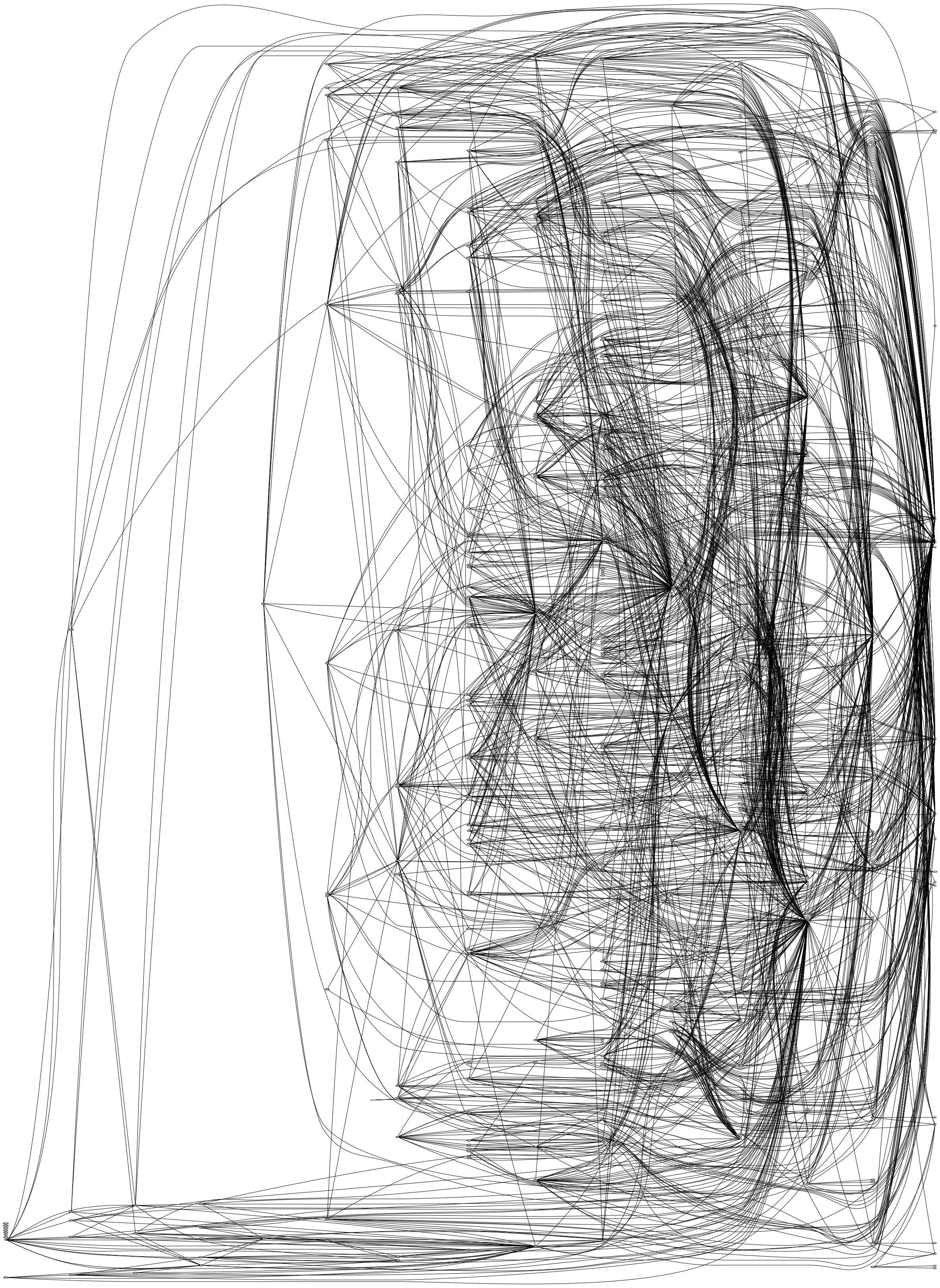

Fig. 3. Dependency Graph.

b) Freevo 1.8

The dependency graph for Freevo 1.8 is shown in figure 4.

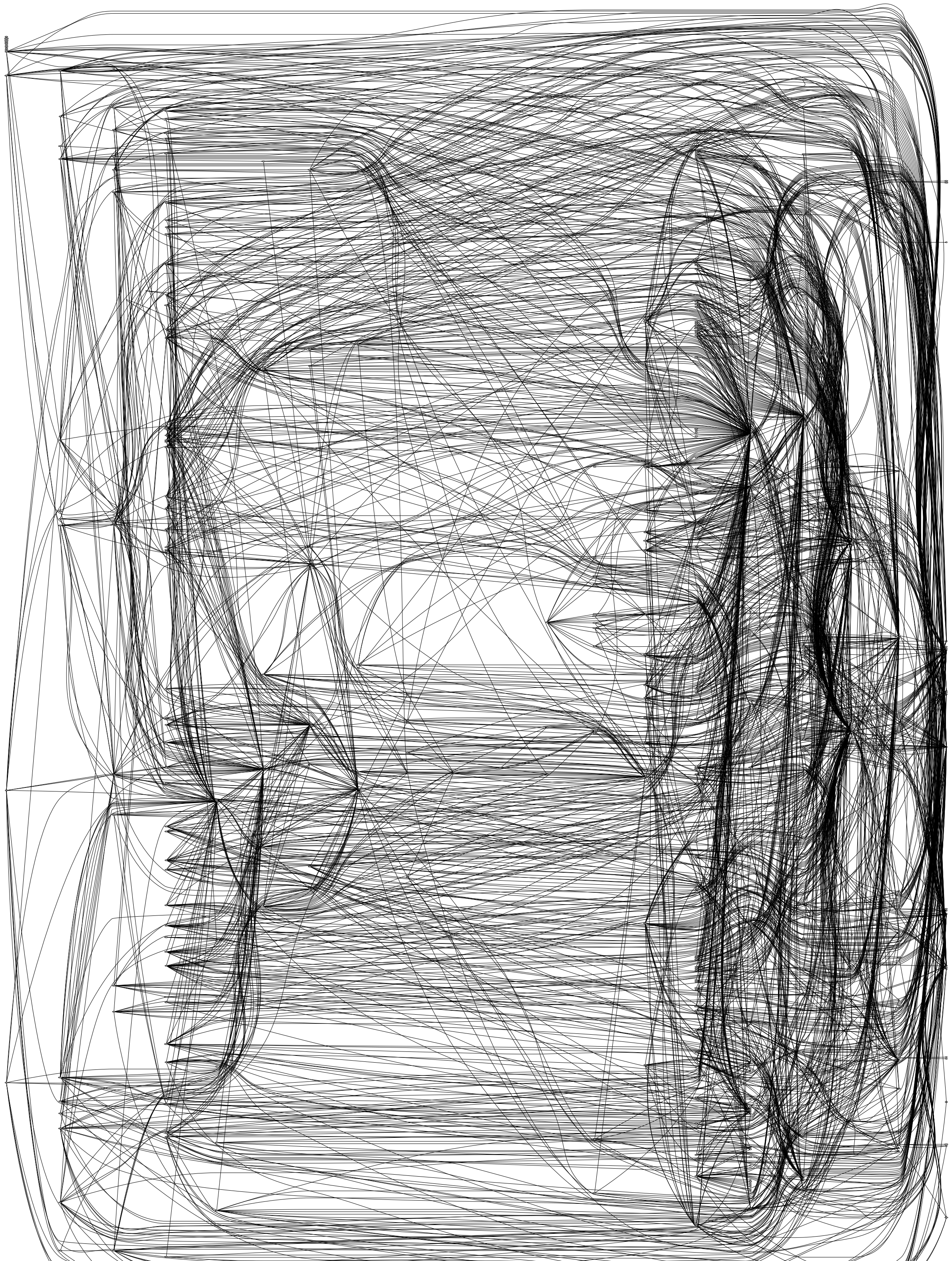

Fig. 4. Dependency Graph.

c) Freevo 1.9

The dependency graph for Freevo 1.9 is shown in figure 5.

d) Freevo 1.7 Without Internal

The dependency graph for Freevo 1.7 is shown in figure 3.

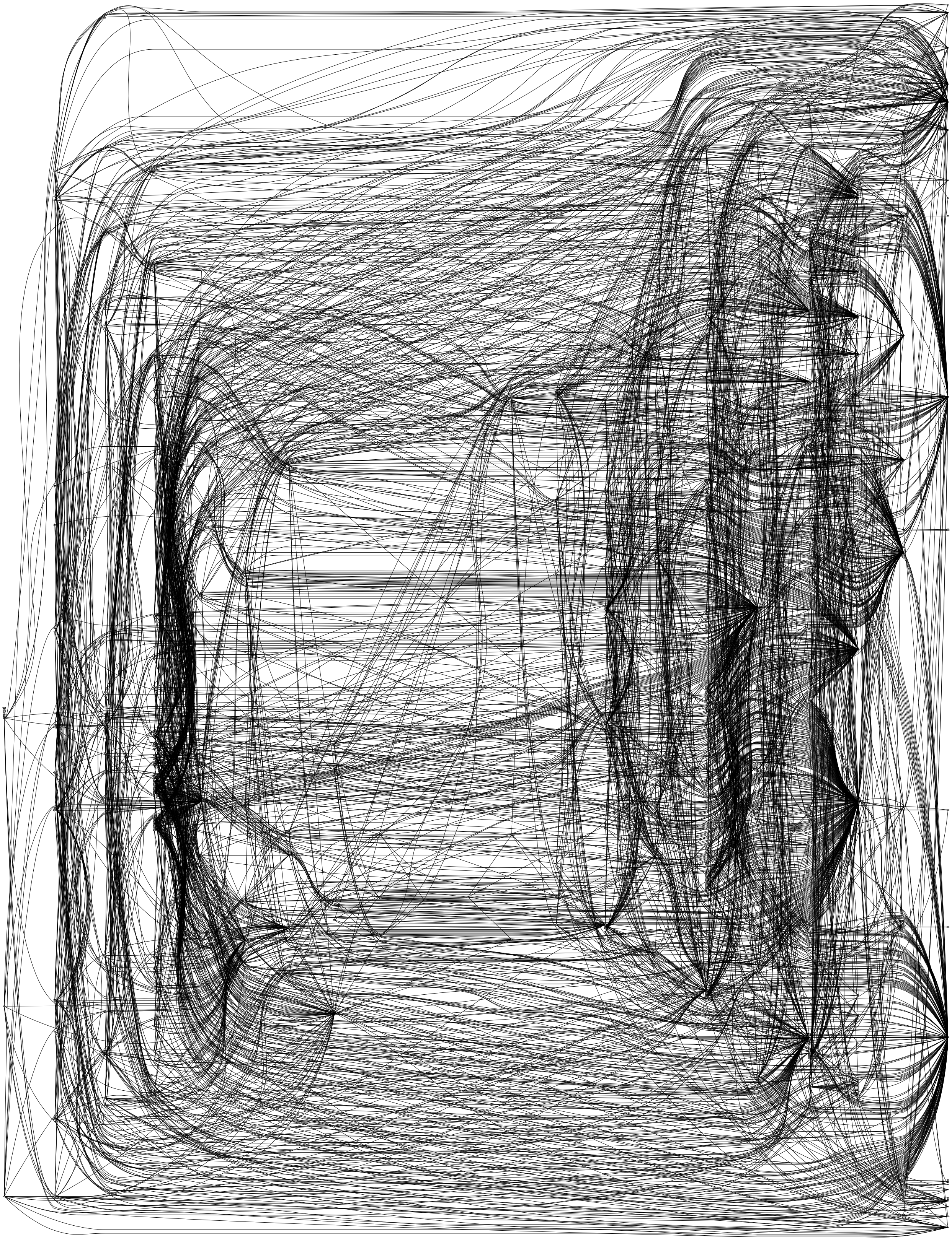

Fig. 5. Dependency Graph.

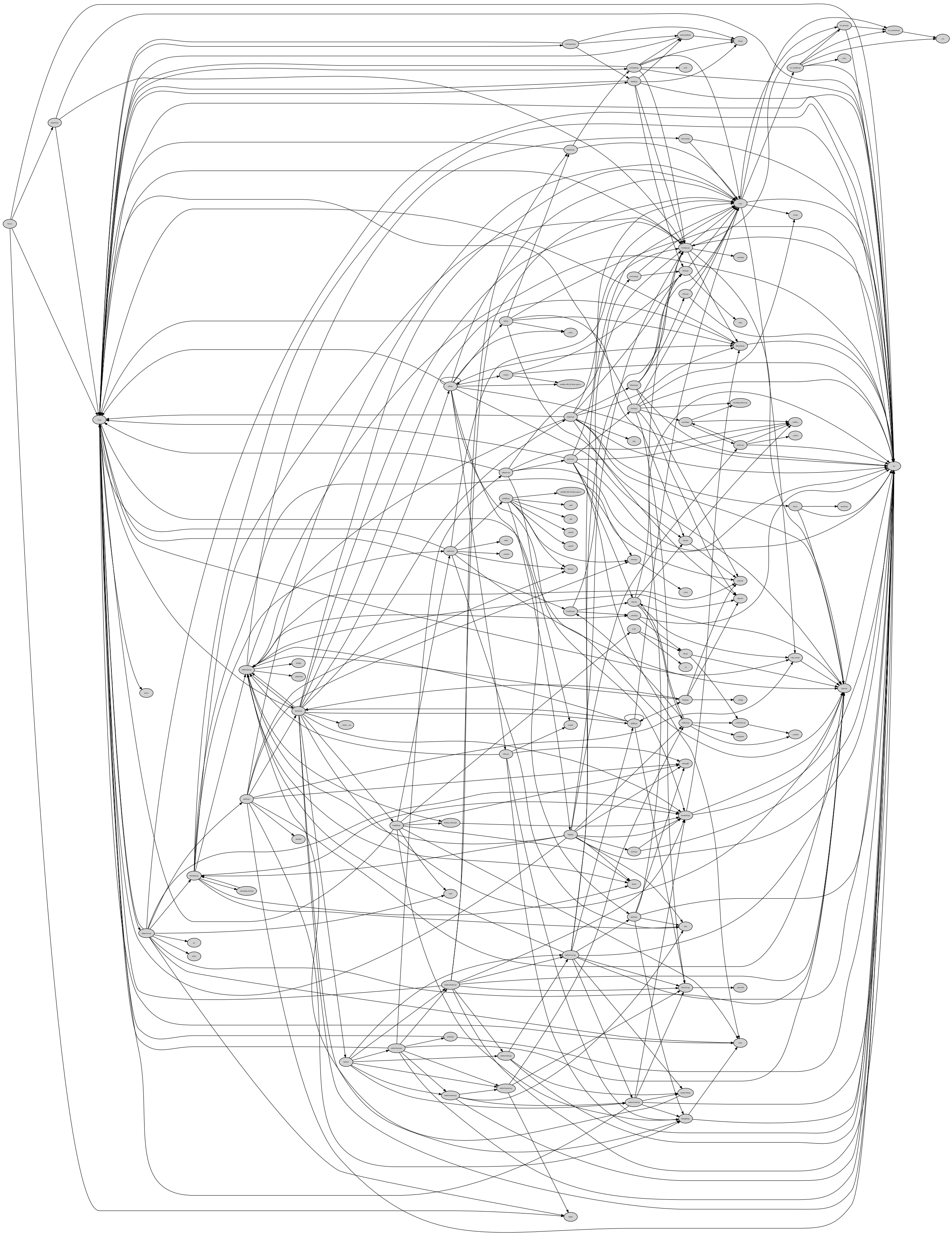

Fig. 6. Dependency Graph.

e) Freevo 1.8 Without Internal

The dependency graph for Freevo 1.8 is shown in figure 4.

Fig. 7. Dependency Graph.

f) Freevo 1.9 Without Internal

The dependency graph for Freevo 1.9 is shown in figure 5.

Listing 2. Author's code base to validate metric generation tools.

```
from mrjob.job import MRJob
import os
import time

class matrixMultiply(MRJob):

        def sortbyJ(listItem):
                return listItem[0]

        def mapper(self, _, line):
                #info1 = matrixM.readline()
                #info2 = matrixN.readline()
                if len(line.split()) == 3:
                        nameFile = os.environ['map_input_file']
                        if nameFile == "outA1.list":
                                iM,jM,valueM = line.split()
                                #print '---',iM,jM,valueM,jN,kN,valueN,'---'
                                for k in range(0,int(columnsN)):
                                        yield (int(iM),int(k)),('M',int(jM),int(valueM))
                                        #print iM, k, 'M', jM, valueM
                        else:
                                jN,kN,valueN = line.split()
                                for i in range(0,int(rowsM)):
                                        yield (int(i),int(kN)),('N',int(jN),int(valueN))
                                        #print i, kN, 'N', jN, valueN


        def reducer(self, key, values):
                oldlistM = []
                oldlistN = []

                for i in values:
                        if i[0]=='M':
                                oldlistM.append([i[1],i[2]])
                        else:
                                oldlistN.append([i[1],i[2]])
                #print oldlistM
                #print oldlistN
                listM = sorted(oldlistM, key=lambda x:x[1])
                listN = sorted(oldlistN, key=lambda x:x[1])
                #print listM
                #print listN
                P = []
                k=0;
                #for i in range((len(listM)/(int(columnsM))-1)*int(columnsM),len(listM)):
                for i in listM:
                        for l in range(0,len(listN)):
                                if listN[l][0]==i[0]:
                                        P.append(i[1]*listN[l][1])
                                        k=k+1
                                else:
```

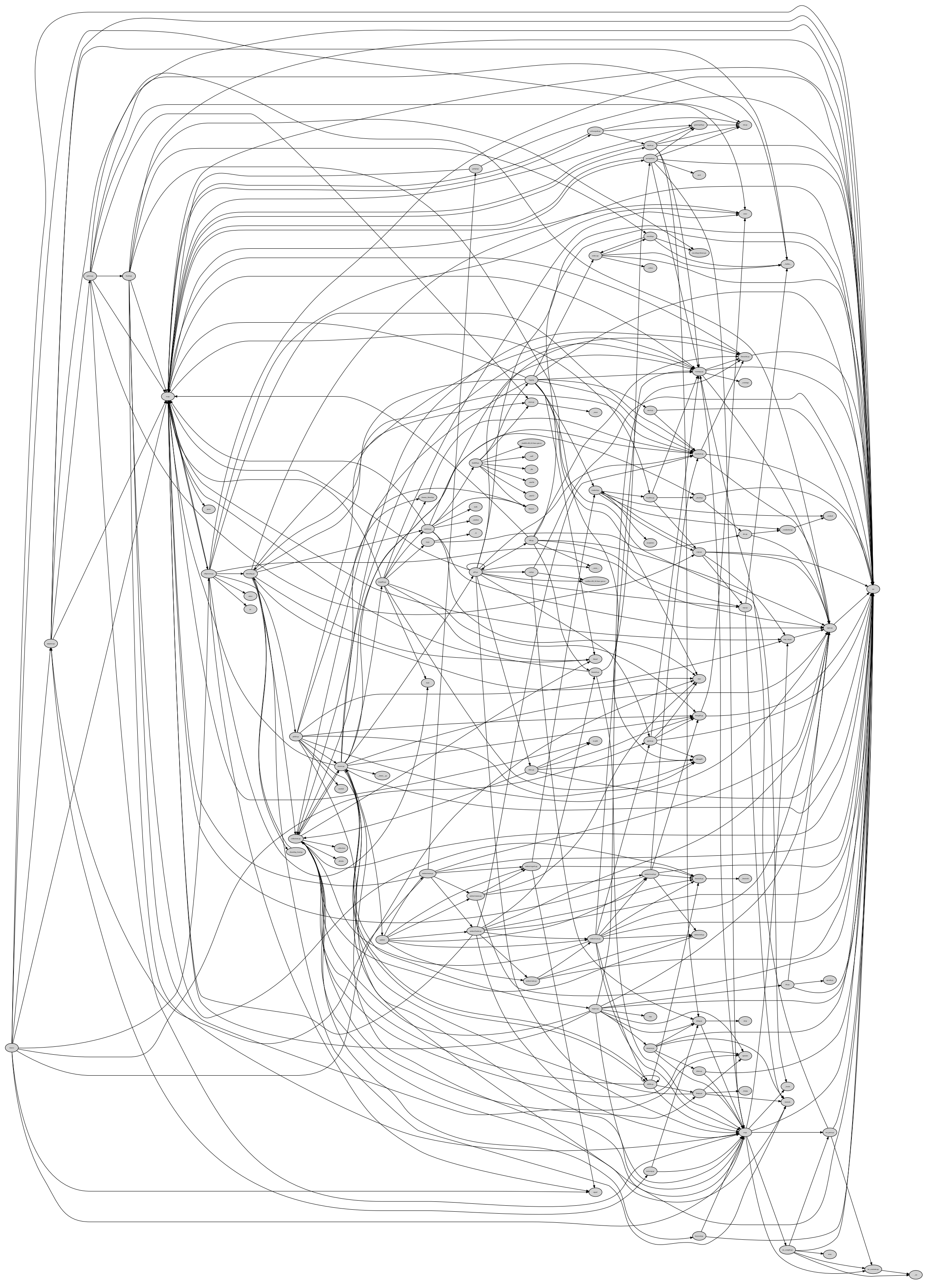

Fig. 8. Dependency Graph.

```
                                        P.append(0)

                        #key.sort()
                        sumofP=sum(P)
                        if sumofP != 0:
                                yield key,sumofP
                                f=""
                                g=""
                                keystr=str(key).split(',')
                                for i in range(1,len(keystr[0])):
                                        f+=keystr[0][i]
                                for i in range(0,len(keystr[1])-1):
                                        g+=keystr[1][i]
                                stringOut = f+g+' '+str(sumofP)+"\n"
                                outputFile.write(stringOut)


                        #print key, sum(P)



if __name__ == '__main__':
        matrixM = open("outA1.list","r")
        rowsM,columnsM = matrixM.readline().split()
        matrixN = open("outB1.list","r")
        rowsN,columnsN = matrixN.readline().split()
        print rowsN," ", columnsN
        if int(columnsN) == 1:
                #print "----"
                tempFile = open('Temp.txt','w')
                tempFile.write(rowsN+' '+columnsN+"\n")
                count=0
                line = matrixN.readline()

                while line != "":
                        i,val = line.split()
                        tempFile.write(str(count)+' '+'0 '+val+"\n")
                        count = count+1
                        line = matrixN.readline()

                tempFile.close()
                reWrite = open('outB1.list','w')
                reWrite.truncate(0)
                tempFile = open('Temp.txt','r')
                for iterator in range (0,int(rowsN)+1):
                        reWrite.write(tempFile.readline())
                reWrite.close()

        #print "columnsN=", columnsN
        outputFile = open('Output.txt','w')
        outputFile.write(str(rowsM)+' '+str(columnsN)+"\n")
        starttime = time.time()
        matrixMultiply.run()
        endtime = time.time()
        duration = endtime-starttime
        print "Time: ", duration
        outputFile.close()
        matrixM.close()
```

```
    matrixN.close()
```