**CodeCut**

**Project Report**

**May 11, 2018**

**Investigators:**

**Cole Bennington - Team Leader - TU -** [**cbenni2@students.towson.edu**](mailto:cbenni2@students.towson.edu)

**Tucker Howard - Researcher - TU -** [**thowar9@students.towson.edu**](mailto:thowar9@students.towson.edu)

**Richard Awojoodu - Researcher - TU -** [**rawojo1@students.towson.edu**](mailto:rawojo1@students.towson.edu)

**Noah Day - Researcher - TU -** [**nday1@students.towson.edu**](mailto:nday1@students.towson.edu)

**Abdulai Dibasy - Researcher - TU -** [**adibas1@students.towson.edu**](mailto:adibas1@students.towson.edu)

**Field of Research:**

**The general area of the project is reverse engineering code through binary analysis.**

**Keywords:**

* **Reverse Engineering**
* **Call Graph Algorithm**
* **CNM**
  + **Modularity(Clauset Newman Moore)**
* **Clustering**
* **Binary analysis**

The intention of this project is to implement and analyze the CNM algorithm’s performance and the validity of its results compared to previous results obtained by other INSuRE teams. Provided time allows us to do so, we plan to implement the NCUT and DBScan algorithms and compare all three as a secondary objective.

Since we started the project, we have been able to make tremendous progress. We first try to get an understanding of the problem and how to approach it. We then implemented the CNM algorithm with the snap library and run it through our dataset. While running the algorithm we were able to determine the graph structure which node and edges are link to each other. However, we can’t run the algorithm through large dataset.

Nowadays, cybersecurity is still a major concern in information technology field. It is related to every person’s daily life, every company’s business, and every country’s national security. Reverse engineering and binary analysis are widely used cybersecurity experts in analyzing malware and security vulnerabilities. Cybersecurity experts use many algorithms in binary analysis for analyzing function calls from machine code, such as CNM (Modularity) algorithm. In this project, we propose a way to analyze the accuracy of the CNM (Modularity) algorithm tested against the predetermined clusters of a programs call tree by testing new datasets and comparing our results with previous results.

**CodeCut**

**Executive Summary**

**May 11, 2018**

**Investigators:**

**Cole Bennington - Team Leader - TU -** [**cbenni2@students.towson.edu**](mailto:cbenni2@students.towson.edu)

**Tucker Howard - Researcher - TU -** [**thowar9@students.towson.edu**](mailto:thowar9@students.towson.edu)

**Richard Awojoodu - Researcher - TU -** [**rawojo1@students.towson.edu**](mailto:rawojo1@students.towson.edu)

**Noah Day - Researcher - TU -** [**nday1@students.towson.edu**](mailto:nday1@students.towson.edu)

**Abdulai Dibasy - Researcher - TU -** [**adibas1@students.towson.edu**](mailto:adibas1@students.towson.edu)

**Field of Research:**

**The general area of the project is reverse engineering code through binary analysis.**

**Keywords:**

* **Reverse Engineering**
* **Call Graph Algorithm**
* **CNM**
  + **Modularity(Clauset Newman Moore)**
* **Clustering**
* **Binary analysis**
* **Snap**

This proposal will be done by our team at Towson University supervised by Dr. Lin Deng with work done by a group of student researchers, Tucker Howard, Richard Awojoodu, Noah Day, and Abdulai Dibasy led by Cole Bennington. We propose a way to analyze the accuracy of the CNM (Modularity) algorithm tested against the predetermined clusters of a programs call tree by testing new datasets and comparing our results with previous results.

**Table of Contents**

[**Introduction**](#_m885puqfplt4) **3**

[**Literature Review**](#_9xrxpnirc7mg) **4**

[**Methods and Procedures**](#_23uao1h7a40j) **6**

[Characterization of Research Methods](#_ds0evox98wbw) 6

[Procedures/plan overview](#_c0snefpvkl6u) 6

[Schedule](#_2rzg3br4xv0d) 7

[Deliverables](#_cy7zdqz1nic) 7

[Limitations and Delimitations](#_wa24ep5j2de4) 8

[**Findings**](#_n6pabc4dpm0t) **9**

[**Issues**](#_crddxicoyx0r) **9**

[**Conclusions and Recommendations**](#_3j2r8avzxo61) **9**

[**References**](#_8ea8h8lqp8tk) **10**

[**Team**](#_nzu18he2jysm) **11**

[Bio sketches](#_j9l5ajbzdmjg) 11

[Tasking](#_v05xvjs9fv26) 12

[**Final Deliverables**](#_r58p2t6jqe4n) **13**

[Source Code](#_b1m8xydjyezg) 13

[Libraries](#_bquaoo2jss5p) 14

[Data results](#_kre61d89bcc8) 15

[Artifacts](#_cgiubudghw2t) 17

# Introduction

In today's society, cybersecurity is a major concern for everyone. Information technology is related to every person’s daily life, every company’s business, and every country’s national security. Around the world, all kinds of systems are prone to be affected by malware that can be used for theft, market manipulation, political gain, or even infrastructure sabotage. Especially, with the rapid development of the Internet of Things (IoT), almost every electronic device, appliance, and equipment is controlled by embedded software systems. It would be extremely detrimental for malware or security vulnerabilities to exist in the embedded control systems of electronic devices around us.

The problem that we are trying to solve through this project is to determine whether the CNM algorithm is an effective way to parse code to be reverse engineered. Our purpose in this project is to improve the standard of reverse engineering tools to advance this field of study.  
  
We are furthering the field of binary analysis which is an integral part of reverse engineering. Reverse engineering binary code can be very important for the cybersecurity field and widely used by cybersecurity experts in analyzing malware and security vulnerabilities. In the typical scenario for security analysts, source code is not accessible by people other than its original developers. If a malicious programmer developed malware, the only method for security experts and law enforcement to detect the malware is reverse engineering and binary analysis. Under a rapidly expanding arsenal of malware, cybersecurity experts and analysts are buried in code with functions that aren’t immediately apparent.   
  
Working to improve the efficiency and effectiveness of tools which are able to help in this process can increase our ability to defend against attackers. The CodeCut project is significant for cybersecurity experts when analyzing a piece of software for bugs, vulnerabilities, or malware because it can drastically save them time and possibly provide more insight. Our work will help extract meaning and see hidden functionality in executables, which is usually all there is to work from when reverse engineering. Cybersecurity experts use many algorithms in binary analysis for analyzing function calls from machine code, such as the CNM (Modularity) algorithm. This proposed project will primarily focus on analyzing the CNM algorithm.This problem is especially challenging because binary code is not readily readable to the human eye, thus we will have to use datasets representing functions and their calls to other functions. From there we can analyze the community clusters and refine them to be compared to the ground truth.

## 

# Literature Review

The team has reviewed a variety of scholarly papers which all outline different algorithms and their research and usage. These papers detail how the algorithms break down and their use in other aspects of technology, such as reverse engineering malware, and using these algorithms to heighten security within software by finding vulnerabilities. We chose these articles based on their relevance to the algorithm we are researching.

We are incrementally furthering the field of binary analysis, because it is an integral part of reverse engineering, which is important for software security. Around the world, systems are affected by malware that can be used for theft, market manipulation, political gain, or even infrastructure sabotage. Under a rapidly expanding arsenal of malware, reverse engineers and analysts are buried in code whose functions aren’t immediately apparent. Working to improve the efficiency and effectiveness of tools which are able to assist can increase our ability to defend against attackers. Codecut is significant because often when analyzing a piece of software for bugs, vulnerabilities, or malware, reverse engineers must work from binary code. Our work will help to extract meaning and see hidden functionality in executables, which is usually all there is to work from when reverse engineering. This problem is especially challenging because binary code is not readily readable to the human eye. The following papers were reviewed for background info and technical knowledge.

**CodeCut by Daniel Lee** As part of INSuRE, research was conducted on reverse engineering that forms the basis of the group’s plan for implementing and analyzing the CNM algorithm using the purity metric. Lee researched graph visualization as well as implementations of the NMCut and DBScan algorithms. He tested both algorithms on an older dataset and determined that NMC showed promise and that there were some inconsistencies in DBScan.   
  
**Finding Community Structure in Very Large Networks**by Aaron Clauset, M. E. J. Newman, and Cristopher Moore  
 In this paper, the CNM authors introduced some optimizations to their algorithm which allowed it to be feasibly run on larger networks than before. The algorithm, which groups nodes together based on modularity, was tested on Amazon’s co-purchasing network, which at the time had nearly 2.5 million edges. Though this algorithm is not intended for reverse engineering, we will implement it and test it on the improved dataset. The CNM authors were able to achieve a running time complexity of O(md log n) where m represents the number of edges, n represents the number of vertices, and “d is the depth of the ‘dendrogram’ describing the network’s community structure (Clauset, et al.).”  
  
**SIGMA: A Semantic Integrated Graph Matching Approach for identifying reused functions in binary code**by Saed Alrabaee, Paria Shirani, Lingyu Wang, and Mourad Debbabi  
 In this paper, the authors identified a need for a platform which is able to recognize and identify reused functions from binary code. They recognized that software, especially malware, tends to borrow functions or sections of code from known repositories. Their platform allows these reused code segments to be recognized in binary, which allows for a head start in the binary analysis. It was helpful to the group to read about a different approach to reverse engineering, and one that could be potentially synergistic with our own.   
  
**BAP: Binary Analysis Platform**  
 This platform exemplifies yet another approach to reverse engineering. The authors detailed how BAP works, and a few potential uses, as well as including its limitations. Though BAP provides an often effective method for analyzing binary, it often requires human intervention to reconcile false positives.   
  
**Reversing Software Engineering** This article is a bit dated, but it served to provide a solid foundation about the basics of reverse engineering to the team. The main improvement to the approach used by the authors is that procedural and nonprocedural processes no longer require a different algorithm for cluster mapping. The foundation provided by this paper will help the team implement our own algorithm.   
  
**Reverse Engineering the Internet**  
 The researchers of this paper provide insight into how the internet is designed and operated. Though this does not directly align with the team’s project, many of the concepts outlined overlap. The researchers annotated a map of the internet with properties that described activity at each node. Using these annotations, they were able to make insights into what is happening in different areas.

## 

# 

# Methods and Procedures

## Characterization of Research Methods

The research we conducted can be categorized as explanatory and constructive. Explanatory research is done to identify and define a problem with no work being done other than learning and finding information. It is done to help researchers and their audience understand the issue at hand whereas constructive research actually tests possible solutions and proposes new solutions to further the progress on the topic.

When we first started researching this topic, we had a lot of explanatory research to do because we knew little about binary code, reserve engineering, and potential tools that can be used for this research. We had to delve into previous works done on the topic and read a lot about how binary code is broken down, what reverse engineering can be used for, and what tools are currently being used to do such tasks. This provided us with great insight when we finally started the constructive part of our research because we knew how the problem was being approached and what to do when we run into an issue.

After learning what we felt was necessary, we decided to put our efforts into testing an existing solution, the CNM algorithm. With the information we learned from our explanatory research, we were able to implement the algorithm using python and test the datasets on it. This left us with some community clusters of the datasets functions and a modularity score. From here we had to determine how we would deal with overlapping of functions in the clusters and how we would then compare the clusters we obtained with the ground truth. We used this method because we felt the only way to get some reasonable progress on this topic in its current state is to propose new solutions and try to implement them. From here on we would expect research involving more empirical methods to test the feasibility of this solution.

## Procedures/plan overview

The major tasks of this research plan were to learn about reverse engineering and the CNM algorithm, learn python and implement the algorithm, and get meaningful output. To do so we did a lot of online research, read a lot of papers, consulted with our technical director, and managed a list of tasks to be completed. The research plan goes as follows:

1. Understand datasets
   1. learn about call graphs
2. Research reverse engineering
   1. read scholarly articles on reverse engineering
   2. learn about different tools for working with call graphs
   3. choose an algorithm
3. Research CNM algorithm
   1. learn what the algorithm does and how it can be implemented
4. Learn python
5. Implement algorithm
   1. setup snap.py library
   2. use existing CNM algorithm in library
   3. modify algorithm to sort community structures and handle overlapping
6. Test datasets
7. Compare results to ground truth
   1. output results in .map file
   2. compare with ground truth .map file

## Schedule

|  |  |  |  |
| --- | --- | --- | --- |
| **W** | **Dates** | **Research** | **Deliverables** |
| **4** | **February 19-23** | **1. Proposal**  **2. Understanding the project**  **3. CNM Algorithm research** | **Final proposal due Friday;** |
| **5** | **February 25 -March 2** | **1. CNM algorithm research**  **2. Dataset research** |  |
| **6** | **March 5-9** | **1. Finalize Scrum**  **2. Practice progress report presentations**  **3. CNM algorithm implementation research** | **Progress report draft due** |
| **7** | **March 12-16** | **1. Progress report presentation** | **Progress report due Friday**  **Progress report presentation** |
| **8** | **March 19-23** | **Spring break** |  |
| **9** | **March 26-30** | **1. Planning** | **Dashboard 1 due** |
| **10** | **April 2-6** | **1. Determine which languages it is already implemented in**  **2. Choose a language** | **Dashboard 2 due** |
| **11** | **April 9-13** | **1. Learn python** | **Dashboard 3 due** |
| **12** | **April 16-20** | **1. Implement snap.py library** | **Dashboard 4 due** |
| **13** | **April 23-27** | **1. Implement CNM algorithm** | **Dashboard 5 due** |
| **14** | **April 30-May 4** | **1. Modify algorithm to handle overlapping and sort the dataset**  **2. Make research paper** | **Practice final presentation**  **Dashboard 6 due** |
| **15** | **May 7-11** | **1. Final presentation** | **Final presentation**  **Final report** |

## Deliverables

The items we will be delivering as a group will be:

*Source Code* - For implementing our version of the algorithm

Contents: zip file of our project that we implemented and a link to the snap.py library

*Research Paper* - For learning about our work from the plan to the outcome

Contents: formal research report

*Datasets* - For testing our version of the algorithm

Contents: graphviz files for mall graph and .map file for ground truth

## Limitations and Delimitations

There were quite a few limitations throughout the span of this project. The first of which had to do with the dataset format. We had no choice over how it was given to us or what it was representing or how big it was. We had no idea what .map files were or how to make one which ended up being a big limitation coming to the final step of our project. The next major limitation we ran into was the choice of language to implement the algorithm in. It had currently only been implemented so many times. It was originally written in C++ and has been modified and remade in other languages but we only could find it in python. Both languages which we were not very familiar with at all. Another limitation we came across was the size of the dataset we used for input. The algorithm could only take input of about 1500 calls before it stopped working. We assume the issues was something to do with the snap library and a restriction of some sort. The biggest restriction we had was with time because we weren’t able to get meaningful results without comparing the output to the ground truth and because of the lack of time we had, we weren’t able to do this.

As for delimitations, or the boundaries of our research, there are also many we can mention. First we chose to work only with the CNM algorithm because we knew we wouldn’t be able to get around to working with all of the options we had available. We also chose to work specifically with python as it was recommended by our technical director and seemed like an easy way to understand the program and modify it. We decided to focus our literature review on the CNM research paper done by the authors of the algorithm because this would give us better understanding of the algorithm. We decided to stop the research after we eliminated overlap and sorted the clusters of the CNM’s original output because we had no time to develop a .map output and then compare that to the ground truth.

# Findings

Throughout the duration of the research, we are able to verify that the CNM algorithm is likely suitable for solving this problem. The output is generally correct, and performs mostly as expected. In its current state, our program is able to accept a call graph of function calls as an input, identify and output communities, and calculate the modularity of the graph.

The algorithm is likely suitable because it is able to group nodes in a large graph into communities based on their interconnectedness. The input for the algorithm is a call graph which represents function calls. Based on this graph, it distributes appropriate nodes into communities which are densely interconnected. As output, each community is printed, along with the nodes that it is comprised of. Additionally, the modularity of the graph is outputted, along with a map file containing the sorted communities. Some additional work is required on the map file in order to compare output with the ground truth file, which is generated at source code compile time. The ground truth represents the original object boundaries of the compiled source code.

# Issues

A number of issues were encountered during the course of this research. The first of which was our python familiarity, since none of the team had spent much time with it. Though python is considered a relatively user friendly language to learn, we did have to spend some time familiarizing ourselves with the functionality of the language before we were able to become productive with it. Another problem is that there really is no established metric for analyzing the correctness for this problem. Output formatting was another issue we ran into. To use the comparison script our technical director wrote, our output would have to be in a map file instead of just grouped communities. Additionally, we ran into some issues with the size of our data set. Currently, the program encounters an vector exception when run on a data set larger than roughly 1500 nodes. Also, there isn’t much documentation available for the Snap library that we used, so some time was spent learning how to use it. We feel that if we had more time to work on this problem, we could have worked through a number of these issues. A group working from the foundation we laid should not have much trouble completing the program.

# Conclusions and Recommendations

We are confident in the suitability of the CNM algorithm for solving this kind of problem, and we are also confident that we are approaching this problem in the right way. However, there is still some work to be done on this problem. The first of which is formatting the output of the algorithm into a map file so it may be compared to the ground truth. Comparison can be done, as well as working to establish a metric for quantifying the correctness of the output. Finally, CNM should be tested against a wide variety of datasets to fully explore the capabilities of the algorithm.

# References

Alrabaee, S., Shirani, P., & Mourad, D. (2015, Mar. 6). SIGMA: A Semantic Integrated Graph Matching Approach for Identifying Reused Functions in Binary Code. *Digital Investigation, 12*(Supplement 1), pp. S61-S71. Retrieved March 2, 2018 from https://www.sciencedirect.com/science/article/pii/S1742287615000122#!

Barr, R., & Golebiowski, L. (n.d.). Call Tree Analysis.

Brumley, D., Jager, I., Avgerinos, T., & Schwartz, E. (n.d.). *BAP: A Binary Analysis Platform.*

Clauset, A., Newman, M. E., & Moore, C. (2004). *Finding Community Structure in Very Large Networks.* Ithaca: Cornell University.

Djidjev, H. N., & Onus, M. (n.d.). *Scalable and Accurate Algorithm for Graph Clustering.* Los Alamos: Los Alamos National Labratory. From https://www.cc.gatech.edu/dimacs10/papers/[06]-clustering.pdf

Fitzgerald, B., & Stol, K.-J. (2017, January). Continuous Software Engineering: A Roadmap and Agenda. *Journal of Systems and Software, 123*, Pages 176-189. From https://www.sciencedirect.com/science/article/pii/S0164121215001430?via%3Dihub

Graham, R. (2018). *Errata Security*. From Errata Security: http://blog.erratasec.com/

Hanson, A., Hahn, D., & Torkelson, L. (2017). *CodeCut.* Madison: Dakota State University.

Henderson, A., Yan, L. K., Hu, X., Prakash, A., Yin, H., & McCamant, S. (2017, Feb. 1). DECAF: A Platform-Neutral Whole-System Dynamic Binary Analysis Platform. *IEEE Transactions on Software Engineering, 43*(2), pp. 164-184. doi:10.1109/TSE.2016.2589242

Lee, D. (2017). *CodeCut.* Houston: University of Houston.

Leskovec, J. (2018, May 10). *Stanford Network Analysis Project*. From snap.stanford.edu/snappy/

Li, S. (2004). *“A Survey on Tools for Binary Code Analysis”.*

Memon, A. M., Banerjee, I., & Nagarajan, A. (2003). *GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing.* University of Maryland, Department of Compter Science. Retrieved Feb. 26, 2018 from http://www.cs.umd.edu/~atif/pubs/MemonWCRE2003-abstract.html

Nohl, K., Evans, D., Starbug, & Plötz, H. (n.d.). *Reverse-Engineering a Cryptographic RFID Tag.* Usenix. Retrieved Feb. 26, 2018 from http://static.usenix.org/legacy/events/sec08/tech/full\_papers/nohl/nohl\_html/

Park, Y., & Bader, J. S. (2017). *Fast and Reliable Inference Algorithm for Hierarchical Stochastic Block Models.* Johns Hopkins University, Department of Biomedical Engineering. Baltimore: Johns Hopkins University. From https://arxiv.org/abs/1711.05150

Peterson, D. G. (2008, 07 08). *Why Do Binary Analysis When You Have Source*. From Digital Bond: http://www.digitalbond.com/blog/2008/07/08/why-do-binary-analysis-when-you-have-source/

Prywes, N. S., Ge, X., Lee, I., & Song, M. (1988). *Reverse Software Engineering.* Philadelphia: University of Pensylvania. From https://repository.upenn.edu/cis\_reports/661/

Spring, N., Wetherall, D., & Anderson, T. (n.d.). *Reverse Engineering the Internet.* University of Washington, Department of Computer Science and Engineering. Wilmington: University of Washington. Retrieved February 27, 2018 from http://www.sds.lcs.mit.edu/HotNets-II/papers/reverse.pdf

WhiteHat Security Glossary. (2018). *Binary Code Analysis*. From WhiteHat Security: <https://www.whitehatsec.com/glossary/content/binary-code-analysis>

# Team

## Bio sketches

**Cole Bennington**, an undergraduate student at Towson University studying computer science, transferred to Towson from Penn State University where he studied computer engineering during his first two years of undergraduate studies. He has a desire to be involved and help other people which is how he got started with being a computer science tutor for the university. His studies currently focus on software development, quality assurance/testing, and algorithm design/analysis.

**Tucker Howard**, an undergraduate student at Towson University. He earned an associate’s degree in Computer Science from Howard Community College in 2016. His studies focus on web development, artificial intelligence, and algorithm analysis.

**Richard Awojoodu**, an undergraduate student at Towson University, has studied the concepts of object-oriented programming and the SDLC through numerous projects in his four year time frame at the university. Richard’s desire is to work as a Application or Web developer.

**Noah Day**, an undergraduate student at Towson University, where he is a Computer Science major with a concentration in software engineering. Noah’s focus’ include web development, as well as some Android development. Noah is also a member of the Upsilon Pi Epsilon computing honors society.

**Abdulai Dibasy**, an undergraduate student at Towson University with a track in software engineering. Abdulai graduated with an associate in computer science from Prince George’s Community College then transfer to Towson University. During Abdulai’s time in Towson, he has worked on various projects both in web and mobile developments.

## 

## Tasking

|  |  |
| --- | --- |
| Team Member | Tasks |
| Cole Bennington | 1. Research on binary analysis/tools 2. Researched existing implementations 3. Made suggestions 4. Work contributed to paper and presentations 5. Team leader |
| Tucker Howard | 1. Researched CNM algorithm 2. Researched Snap library 3. Hosted and led team meetings 4. Facilitated discussion with TD 5. Testing and debugging 6. Assisted in writing the source code 7. Contributed to paper and presentations |
| Noah Day | 1. Researched different implementations of CNM algorithm 2. Researched the Snap.py library as well as implementation methods 3. Learned python language 4. Maintained the repository for housing the code 5. Wrote the algorithm and code 6. Debugging and testing 7. Contributed to paper and presentations |
| Richie Awojoodu | 1. Researched different implementations of CNM algorithm 2. Researched the Snap.py library as well as implementation methods 3. Learned python language 4. Contributed to paper and presentations |
| Abdulai Dibasy | 1. Learned python 2. Contributed on researching presentations 3. Did some research on the topic 4. solved issue with setting up snap 5. Contributed on the proposal and final paper 6. Did some literature reviews |

# 

# Final Deliverables

## Found at <https://github.com/Bookfan97/CodeCut_Spring2018_Towson.git>

## Source Code

## import snap

count=0

graphList=[**"DataFiles/px4.gv"**,**"DataFiles/gnuchess.gv"**,**"DataFiles/px4\_subsample\_100lines.txt"**,**"DataFiles/px4\_subsample\_762lines.txt"**, **"DataFiles/test\_length.txt"**]

UGraph= snap.PUNGraph.New()

w= 100;

communities=[[0 **for** x **in** range(w)] **for** y **in** range(w)]

clusters=[]

comm=[]

map=[]

allnodes=[]

cmtyindex=0

nodeindex=0

c=[]

community=[0 **for** x **in** range(50)]

v=0

**for** index **in** range(0, len(graphList)):

**print**(**"%(n)s:%(s)s"** % {**'n'**: index, **'s'**: graphList[index]})

filename=input(**"Which graph file would you like to test?"**)

**print**(str(graphList[filename]))

**with** open(graphList[filename], **"r"**) **as** f:

**for** line **in** f:

**if** line.strip() == **"}"**:

**break**

NodeIDL, NodeIDR= line.split(**"->"**)

NodeIDR=NodeIDR.split(**"["**)[0]

NodeIDR.strip()

string1= NodeIDR.split(**"x"**,1)[1]

string2= NodeIDL.strip(**"s\_0x"**)

a= int(string1, 16)

b= int(string2, 16)

**if** UGraph.IsNode(a) == False:

UGraph.AddNode(a)

allnodes.append(a)

**if** UGraph.IsNode(b) == False:

UGraph.AddNode(b)

allnodes.append(b)

**if** UGraph.IsEdge(a,b)==False:

UGraph.AddEdge(a, b)

count=count+1

CmtyV = snap.TCnComV()

modularity = snap.CommunityCNM(UGraph, CmtyV)

**for** Cmty **in** CmtyV:

string = **""**

**print** (**"Community: "**)

**for** NI **in** Cmty:

**print** (hex(NI))

comm.append(NI)

comm.sort()

clusters.append(comm)

comm=[]

v = v + 1;

**print** (**"The modularity of the network is %f"** % modularity)

allnodes.sort()

**for** x **in** xrange(1,len(clusters)):

index=0

cluster=clusters[x]

moduleStart = clusters[x][index]

indexplusone=index=1

**if** indexplusone>len(cluster)-1:

**break**

tempModuleEnd = clusters[x][indexplusone]

nextFunction1 = clusters[x][index]

allindex=allnodes.index(moduleStart)

nextFunction2 = allnodes[allindex]

**while** (nextFunction1 == nextFunction2):

index=index+1

**if** index>len(cluster)-1:

**break**

allindex=allindex+1

**if** allindex>len(allnodes)-1:

**break**

nextFunction1 = cluster[index]

nextFunction2 = allnodes[allindex]

**if**(nextFunction1 == nextFunction2):

tempModuleEnd = nextFunction1

finalindex=abs(moduleStart-tempModuleEnd)

map.append(hex(finalindex))

map.sort()

**print**(**"Map: "**+str(map))

file=graphList[filename].split(**".txt"**)

filestring=str(file[0])

mapname=filestring+**".map"**

### 

## Libraries

Snap.py was used to implement the CNM algorithm. This library, created by Stanford University, was developed in order to have a library of functions to be able to calculate community structures within code. Originally designed to create and output a random graph, the code was modified to allow for a graph to be read from a file to populate a graph.

## Data results

The program itself, at its’ current state, outputs several different features that was implemented within the code. The code allows for the user to select from several different files within the program, and will run the program using that file. When the program is run, it provides the different communities, including the nodes that create those communities. The program will then provide the modularity of the program, and will also provide the sorted map after the entire program is run.

This is an output example, using a subset of the px4.gv dataset, which consists of 100 lines of inputs.

C:\Users\shelt\PycharmProjects\CodeCut\venv\Scripts\python.exe C:/Users/shelt/Documents/GitHub/CodeCut\_Spring2018\_Towson/CNM\_working/CNM.py

0:DataFiles/px4.gv

1:DataFiles/gnuchess.gv

2:DataFiles/px4\_subsample\_100lines.txt

3:DataFiles/px4\_subsample\_762lines.txt

4:DataFiles/test\_length.txt

Which graph file would you like to test?2

DataFiles/px4\_subsample\_100lines.txt

Community:

0x402470

0x416340

Community:

0x402544

0x427100

0x427220

Community:

0x402d30

0x402db0

Community:

0x402d70

0x402dd0

Community:

0x402e20

0x402560

0x402eb0

0x402f10

0x402f30

0x403d00

0x404270

0x405820

0x405840

Community:

0x402fe0

0x402ff0

0x403000

0x403010

0x403020

0x403030

0x403040

Community:

0x4032b0

0x403370

0x403420

0x4042f0

0x404390

0x405280

0x40f430

0x405550

0x4055d0

0x405830

0x4031d0

0x403200

0x403230

0x403260

0x403290

0x4036a0

0x4036e0

0x4042b0

0x40fae0

Community:

0x404ff0

0x403f60

0x40f610

0x406740

0x40bfa0

0x407890

0x4084a0

0x409290

0x40bb20

0x409ab0

0x40ac40

0x404a20

0x404bc0

Community:

0x404c80

0x404d30

0x404d50

0x404f80

0x404fb0

0x404fd0

0x403b60

0x4064a0

0x4045f0

0x40b600

0x403a90

0x404af0

Community:

0x405780

0x40abd0

0x40e3e0

0x405850

0x406210

0x410400

0x4063d0

Community:

0x403a10

0x40a720

0x409c20

0x40aaa0

0x409d30

0x409e80

0x40a080

0x40e270

The modularity of the network is 0.661950

Map: ['0x10', '0x24bbc', '0x30', '0x60', '0x6210', '0x80', '0x8c0', '0xac0', '0xd0', '0xd0']

## Artifacts

The directory also contains a folder with a list of all the data files that are contained within the array at the beginning of the code. These files are: DataFiles/px4.gv, DataFiles/gnuchess.gv, DataFiles/px4\_subsample\_100lines.txt, DataFiles/px4\_subsample\_762lines.txt, and DataFiles/test\_length.txt. These contain both the subsets that our team used to test the code, the smaller subsets of the px4 dataset, in addition to the original data files that were given to us by our technical director.