Interactive Visualizations Final Project Report

Introduction

We examined the possibility of whether software complexity followed the same exponential trends that show up in the evolution of computer hardware over time. The most common observational trend of computer hardware is Moore's Law which notes that the number of transistors in a computer processor tend to double approximately every 18 months. We hypothesized that evolution of code complexity parallels Moore's Law's exponential trend, with software complexity, too, doubling every 18 months. After testing several metrics for software complexity, we were unable to find the hypothesized trend in the data we collected, but we believe that this inquiry is not fully explored, and that there are still many metrics pertaining to software complexity that would serve well for future study.

Motivation & Audience

From a subjective standpoint, it seems that modern computers don't feel very much faster than their older counterparts despite measurable, and well-advertised, improvements in the hardware the computers used. This discrepancy lead to the inquiry into how software has changed over time and whether that advancement accounts for the unchanging user experience of computer speed for computers of different hardware capabilities.

If the complexity of software paralleled the hardware improvements, then the subjective experience of computer speeds wouldn't change over time and because hardware improvements followed Moore's Law, it was natural to compare the evolution of software complexity to Moore's Law. To better represent the results of the study, we broke the hypothesis down into two parts:

H1: The evolution of the software complexity follows an exponential trend with respect to time.

H2: The complexity of software will double every 18 months (1.5 years) like observed in hardware by Moore's Law.

Due to the varied nature of software and measurements of software, we decided that it would be best to test several different methods and software. To homogenize the data we would be working with, we decided to focus on computer operating systems as they are a consistent dataset with years of updates and upgrades to represent the evolution of software over time while also being a central part of the users' experience with a computer. The first source to be tested would be the Linux kernel on GitHub. This source was chosen due to the ubiquity of programs that use such as Ubuntu servers and the Android operating system, while also having a thorough set of temporal data accessible through GitHub. The second and third sources to be tested would be the published computer requirements by Windows and the disk image size of different Microsoft operating systems. These sources were chosen due to the popularity of the Windows line of operating systems as well as the assumption that the ubiquitous nature of these operating systems would make it easy to find data about the specifications of the different Windows operating systems.

Evolution of the Final Project



Figure 1: Graph of RAM specification data including missing data. Missing data was set to have a value of zero causing a line of bad data on the x axis.

The initial goal of the project was to get data from plenty of sources and test them against our hypothesis. The visualization itself did not go through much evolution as a scatterplot or line graph effectively represented the data we had quite well. Even the plans for the interactivity of the graphs was decided at the start of the project. However, as the data was collected, it became evident that the data sources had several looming problems such as the missing data in the published minimum specs, or the non-exponential trend for the Linux kernel line count. We did not originally plan to get non-exponential results, and the existence of that linear trend actually convinced us to split the hypothesis to better cover the different kinds of results we were getting.

Related Work



Figure 2: Transistor count of CPUs and GPUs from the Sun et.al paper on Moore's Law.

Moore's Law is an observation by Gordon Moore that every one or two years, the number of transistors in a typical dense integrated circuit doubles. A 2019 paper by Sun et.al. confirms the validity of

Moore's Law for modern CPUs and GPUs allowing us to use Moore's Law as a consistent metric for changing hardware complexity for testing software complexity against. Despite the rigorous study of Moore's Law and evolution of hardware heuristics, we were unable to find studies that tracked the evolution of software over time. Nevertheless, there are many about the general measurement on software complexity. Alsultanny, in his paper about software complexity, actually lists several different methods for determining software complexity of a program including the McCabe method where complexity is determined through analysis of the control paths of the program and Halstead's method where the complexity is determined by analysis of the different operators in the program. However, the main limitation of all listed measurements of software complexity is that they require the source code of the programs whose complexity is to be measured. Because only open source programs have consistently (and legally) accessible source code, these methods only work on the subset of open source programs that we were planning on testing.

The Linux kernel is public on GitHub, so getting its data was as simple as parsing a downloaded clone of the project. For the Windows specs, however, it appears that Microsoft doesn't store its operating system data in any centralized location, making official documents all but unusable as a source of data. In the end, we scraped data from a fan-made site called WinWorld that actually stored copies of the operating system disk images that we could analyze.

The Methods Measurement for Software Complexity

For analyzing the complexity of the Linux kernel, we did a line count of the program snapshot at different years in the Linux operating system. Some flaws with the line count heuristic come from the fact that different structures in the code can have drastically different performance in terms of computational and spacial complexity. The existence of comments which use no system resources as well as compound commands that store multiple commands on one line further muddy the water for the data.

The published minimum required hardware specs were taken as they were from the WinWorld site. On analysis of the collected data, two flaws became evident. The first was that many of the operating systems were missing values for their minimum specs. For the minimum CPU requirement, the missing data was so bad that it had to be thrown out. The second problem was that many of the published minimum requirements were heavily rounded, often to only one significant figure. It is also likely that the rounding in itself is an overestimate of the actual requirements of the operating system.

The size of the Windows disk file was taken as the average of the file sizes for the disk files listed in the respective operating system on the WinWorld site. Possible errors for this source come from how the disk files contain many types of data besides the code itself including art assets, audio files, and text files. In addition, several of the listed disk files on the WinWorld site included different language packs that could have skewed results.

Technical Details

To get the line counts of the Linux kernel from GitHub, Luke created a basic script in Ruby along with some bash script. The program looked at the Git repository's reference log (reflog) in chronological order and extracted the latest commit numbers within each year. Luke had to create a custom date format and grab the ones that met the definition. The program extrapolated the last commit of a repository in a year, and then from there, ran a specialized word count (wc) to return only the line counts of each file. In the case of the Linux kernel, this process took a few short minutes.

Thomas created a Python program with Selenium to scrape the operating system specifications off of the WinWorld site. The program took about an hour to scrape every operating system listing on the site. The obtained data was parsed and serialized into a byte-stream file using the Python Pickle library.

The D3 JavaScript library was used to generate the scatter plots and the line graph due to its flexibility and ease of use in making interactive visualizations. The resulting charts allowed the user to mouse over the points in the graph to see a tooltip containing detailed information about the data-point. Thomas, in order to make uniform graphs, made a short Python script to create the html files for the two scatter plots.



Results

Figure 3: Line counts of the Linux kernel resulted in a roughly linear trend.

The results of the data for the Linux kernel line count resulted in a slow linear trend in line count over time. Because the line count was assumed to be proportional to the complexity of the software, the result is a linear growth in the complexity of the code for the Linux kernel over time. Being non-exponential, this result fails H1. Also, as the apparent complexity failed to double even once over the 14 year period, the data also fails H2.



Figure 4: The left graph uses the published minimum RAM specification to represent software complexity; the right graph uses the average size of the disk image of the operating system to represent

software complexity; the black line is an exponential trend line for the data where b is the amount of time it takes for the graph to double.

The minimum RAM specification data had a few large outliers that seemed to skew the graph. Like the line count, the minimum RAM specification data was assumed to be proportional to the overall software complexity of the respective operating system. The coefficient of determination (the R^2 value) for the RAM specification data was 0.19 showing that the exponential trend does not account for much of the data, failing H1. The doubling time for the software complexity came out to be 3.5 years which fails H2.

The disk size data was the most comprehensive of the data we tested. Again, the disk size was assumed to be proportional to the software complexity of the operating system. Overall, $R^2 = 0.55$ for the disk size data representing a relatively strong exponential trend passing H1. However, the doubling time for the disk size data is 3.4 years which fails H2.

Only the disk size data exhibits a clear exponential trend and none of the three sources of data show the data doubling every 18 months to parallel Moore's law.

Audience Feedback

After presenting the project, we received a decent amount of feedback on what we did. Overall, the responses we got were quite mixed with the uniqueness of the idea and the thoroughness of the data receiving the most praise, and the method for measuring complexity and the visualization getting the most criticism. In addition, there were some questions about the cohesiveness of the different data sets we collected.

The most important criticism, from the standpoint of the credibility of the project as a whole, is the method we used for measuring the complexity of the different software. As mentioned in the Methods for Measurement section, the heuristics we used for measuring software complexity contained several major flaws. The reason we used these methods was mostly due to time limitations. Our main focus was on the acquisition of data, so we agreed to use simpler heuristics to measure complexity in order to spend more time on the data collection. We do sincerely hope that future work on the subject of evolving software complexity take into account different ways of measuring the complexity of programs.

The second major criticism came from the lack of complexity in the visualizations made for this project. Part of the reason for these simple visualizations came from the uncomplicated nature of the data itself. The lack of error sources and the basic statistic versus time really make the scatter plots and line graph used the most reasonable visualizations for the data we collected. The addition of interactivity was done to better satisfy the requirement of having an interesting visualization for the project. Given more planning at the start of the project, we might have been able to figure out some additional features to add to the graphs to make a more interesting final visualization.

The final concern about the cohesiveness of the project as a whole probably comes from the separate nature of how we compiled the work. Because each of the sources of data was simple enough that it could be scraped and parsed by a single person, we decided to split the data up based on the sources. During the planning stages of the project, we did not expect the data to be varied enough to justify an explanation as to how the data was related. In hindsight, a meta-analysis of the tests we did would have greatly helped bring the data sources we acquired together.

Conclusions and Future Work

None of the three datasets had software that shared the same increase of complexity as Moore's law. The lack of positive results suggests one of three possibilities: that the methods we used for measuring software complexity were unable to accurately measure software complexity, that the data we collected is not representative of the software complexity that we were looking for, or that software complexity does not follow Moore's law and the subjective experience described under the inspiration for the project is unfounded.

We somewhat doubt the third explanation as it would make sense for software to make full use of the available hardware for the period. For future research, we would like to see how different methods for measuring software complexity compare to the methods we used. One possibility is to run the software through a virtual machine and keep track of CPU cycles and memory used. We would also like to see how different pieces of software evolve in complexity over time and determine whether the results we got are unique to operating systems. In addition, as suggested in some of the feedback, it would be useful to see the effects of bloatware, or programs that run in the background that might also effect the perceived speed of a computer to a user.

Division of Work

The project was split based on the source of the data. Thomas handled the scraping of operating system specification data off of the WinWorld site. Luke handled the acquisition of the line count data from the Linux kernel GitHub project. Both students wrote their own code to scrape and parse the data from their respective source, and both students independently made their graphs in D3.

References

- Sun, Yifan, et al. (2019). "Summarizing CPU and GPU Design Trends with Product Data." Northeastern University, arxiv.org/abs/1911.11313.
- Alsultanny, Yas. (2011). Methodologies of Software Complexity Evaluation for Software written by Object Oriented Languages.
- > Torvalds. "Linux Kernel Git Repository." *GitHub*, 3 Apr. 2020, github.com/torvalds/linux
- ▶ Loeliger, Jon, and Matthew J. McCullough. *Version Control with Git:* O'Reilly, 2012.
- > "WinWorld Library." *WinWorld*, winworldpc.com/library/operating-systems.