Analyzing the Code Quality of Large-Scale Software Packages

Bosco Ndemeye
Computer and Information Science
University of Oregon
Eugene OR, USA
ndemeye@cs.uoregon.edu

***Abstract*—Many popular metrics used for the quantification of the quality or complexity of a codebase (e.g., McCabe's Cyclomatic Complexity) were developed in the 1970s or 1980s when source code sizes were significantly smaller than they are today, and before a number of modern programming language features were introduced in different languages. Thus, the many thresholds that were suggested by researchers for deciding whether a given function is lacking in a given quality dimension need to be updated. In the pursuit of this goal, we study a number of open-source high-performance codes, each of which has been in development for more than 15 years—a characteristic, which we take to imply good design and devise a method for identify project-specific thresholds. As such, first, we employ the LLVM/Clang compiler infrastructure and introduce a Clang AST tool to gather AST-based metrics, as well as an LLVM IR pass for collecting those based on a source code's static call graph. Then we perform statistical analysis to identify reference thresholds of 22 code quality and callgraph-related metrics at the function-level. Next, we show that the gathered function-level data can be used to identify instances of major refactoring during the evolution of a codebase, and lastly we demonstrate how critical functions in source code can be identified and selected for careful study.**

***Index Terms*—datasets, static analysis, metrics, data mining, llvm, clang**

I. Introduction

The quality of a library or an application's release (referred to throughout the rest of this paper as a *codebase*) dictates not only its trustworthiness among its users, but implies also the amount of additional work needed from the software developers in the next release of the project. Thus, quality (as well as complexity) needs to be quantified so that the usual tools of data science can be used to rank sections of the codebase and locate areas that need improvement. Such quantification is usually done either dynamically by selecting a number of example applications which are run in order to gather the metrics of interest, or the quantification is done statically by directly analysing the structure of the codebase after passing it through the syntactic and semantic stages of the compiler.

The Software Engineering (SE) research community has introduced several tools (see [1], and [2] for example) to do just such a quantification, and to provide actionable recommendations to developers of their client codebases. One approach used by many of these tools involves identifying multiple thresholds and ranges for a number of quality metrics. The tools then identify both those code segments with a score lower than the threshold as lacking in the quality dimension of interest, as well as those with a score higher than the upper range boundary. As a consequence, the tool separates the codebase into problematic and non-problematic territory for their clients; who can then focus their quality-improving-efforts only on those deficient parts.

This paper concerns itself with structural code quality (as opposed to security-related code quality for example). And the function-level quantification of this particular type of quality for many tools, involves the use of metrics proposed by McCabe [3], and/or Halstead [4]. Utilizing these metrics seeks to quantify and infer such aspects of the function's quality as ranging from its *difficulty* to the amount of *time* it should take an experienced developer to write it; all from the total number of *operands* and *operators* used in the function. We do not seek to question the validity of these metrics in this paper, but instead implement the metrics as specified. Moreover, we seek to relax the thresholds that tools, such as those mentioned above, rely on to identify good and bad design for our applications of interest—namely High Performance Computing (HPC) software packages. We do this because, despite the fact that all of the software packages we analyze have been around for over a decade (which should imply that, overall, they are well designed), many of the "normal" threshold-based tools would flag most of the methods in these applications as problematic.

The tool we propose relies on the LLVM/Clang compilation infrastructure to collect function level values for both traditional SE metrics (such as [4]) as well as many other graph-theory-based centrality metrics. The graph-theory-based metrics are extracted through an analysis of the source code's callgraph with the help of *networkit* [5]. Halstead metrics, on the other hand, are collected with the help of *Clang Tools* [6] by walking over an Abstract Syntax Tree[1] (AST) generated with the help of Clang's front-end.

With our developed tool, first, we performed a case study with three well-known HPC libraries, namely PETSC [7],Su-

[1]Recall that an Abstract Syntax Tree refers to the resulting structure obtained after a source file successfully passes a compiler's syntactic analysis.

perLU [8], and SLEPC [9] , and gathered 22 traditional [4] metrics as well as 6 centrality metrics—borrowed from network analysis, but which we believe can be interpreted in software engineering terms and lead to insightful inferences. Second, like [10] we performed a statistical analysis by leveraging the discriminative power of the Easyfit tool [10] to compute the threshold values of each metric. Because the values of our metrics are unbounded in nature, we considered only probability distributions to fit our metric data. Consequently, we recommend the reference values of each metric and label them as low, intermediate range and high.

To summarize, we sought to answer the following research questions:

1) **Since tools might need to be customised to incorporate project-specific needs, what is an empirical procedure for identifying project-specific thresholds for code quality metrics?**
2) **Using function-level data, how can we identify instances of major refactorings of a codebase?**
3) **Since the development of functions has been shown to obey power distribution laws, how can we identify the most critical functions in source code?**

As such, our contributions can be summarized as follows:

- A scalable and extensible tool for collecting code quality metrics.
- A procedure for identifying project-specific thresholds for code quality metrics.
- A process for identifying the most critical functions in a source code repository.
- A comparison of the efficacy of two methods for identifying the most critical functions in source codes.

II. Background

A. Graph Centrality Metrics

This section reviews concepts related to the type of data gathered by our tool, and is used to rank functions and procedures based on their relative levels of influence in a codebase.

Recall that mathematically, a static *callgraph* of an application is a directed graph $G = (V, E)$ where each element v of the set of vertices V is a function declared in the source code, whereas each element (f, g) of the ordered set of edges E, represents the fact that f will call g when the source code is compiled and run. Recall also that in the presence of *aliasing* (having multiple variable names refer to the same memory location in a program), precise computation of such a graph is undecidable. This is because computing such a callgraph—one in which there is an edge from a function f to a function g if and only if f calls g—depends on a solution to the static alias analysis problem, which is itself undecidable [11]. Consequently, modern tools for building static call graphs are *conservative*; never omitting an edge between functions f and g if the call may occur in the program, but over-approximating and including function calls that might never occur during the actual execution.

Recall however that, even though precise static call graph construction is undecidable, *dynamic call graphs*, for a given code-example pair, can be constructed without much difficulty. This is because these are constructed at runtime, when all the information which a program needs to run is known, and thus the triggered calling context tree can be recorded. Unfortunately, because the scope afforded by such a precise dynamic callgraph extends only to the size of the chain of function calls set into motion by the chosen example, the question, of the total number of examples needed for the generation of a callgraph which covers most of the codebase, is inevitably raised.

Despite the contrast between static and dynamic callgraphs however, the underlying mathematical notion remains the same. A callgraph is a directed graph, and as such, any techniques discovered from studying such structures can be used on it and yield possibly interesting results. Consequently, we review a number of directed graph *centrality* metrics, which we extract from the static callgraphs generated by our tool, and use to rank an application's functions in accordance with their relative influence in the graph.

1) Average Shortest Distance: The distance $d(v_i, v_j)$ between any two vertices $v_i, v_j \in V$ is defined as the number of edges in a shortest path connecting them. As such, the average shortest distance of v_i is defined as

$$A(v_i) = \frac{d(v_i, v_j)}{|V| - 1}, i \neq j$$

Where $d(v_i, v_j) = 0$ if there is no path from v_i to v_j. A lower value of this metric indicates that the node v_i is relatively close to every other node in the graph. Especially, $A(v_i) = 1$ if and only if v_i is connected to every other node in the network.

2) Closeness Centrality: Related to *average shortest distance*, the formula for a node's *closeness* centrality value is defined so that the closer a node is to every other node, the higher its value for the metric:

$$C(v) = \sum_{w \in V} \frac{1}{d(v, w)}$$

Thus, if information needed to flow through a network, nodes with a higher *closeness* value would be able to relay messages using the least number of steps.

3) Betweenness Centrality: As suggested by its name, *betweenness* is a measure of the relative number of shortest paths that a node v appears on. Specifically,

$$g(v) = \sum_{s \neq v \neq t} \frac{\delta_{st}(v)}{\theta_{st}}$$

Where θ_{st} is the total number of shortest paths from s to t and $\delta_{st}(v)$ is the number of those paths that pass through v. For our purposes, a function with a higher *betweenness* value is relatively more likely to be called by an example application.

4) Eccentricity: The greatest distance $\epsilon(v)$ between a vertex v and any other vertex is called its *eccentricity* and can be calculated thus:

$$\epsilon(v) = \max_{u \in V} d(v, u).$$

The *radius* r of a graph is the minimum eccentricity, while its *diameter* d is the maximum eccentricity.

A central vertex can be thus be defined as one whose eccentricity equals the graph's radius.

5) FanIn & FanOut: Otherwise known as *in-degree* and *out-degree* respectively, for a directed graph $G = (V, E)$, *fan-in* of $v \in V$ is a count of the number of edges for which v is an endpoint, whereas *fan-out* counts the number of edges for which v is a starting point:

$$FanIn(v) = |S_v| \text{ where } S_v = \{(w, y) \in E \mid y = v\}$$

and

$$FanOut(v) = |S_v| \text{ where } S_v = \{(w, y) \in E \mid w = v\}$$

B. Code Quality Metrics

1) Cyclomatic Complexity: Developed by T.J McCabe this metric measures the complexity of the decision structure of a program. Specifically, for the implementation described in this paper, this is a count of the following modern control flow directives:

- *If* statements
- *For* loops
- *While* loops
- *?* : The conditional/ternary operator
- *break*
- *continue*
- *case*

2) Number of Operators: Given a function f this metric measures the number of *operators* local to the function. Specifically, this a count of all the **binary, unary,** and **ternary** operators; summed up with all the **function calls** made by the function.

3) Number of Operands: A function's total number of operands is calculated as the total number of its local variables in addition to the number of its arguments.

III. Proposed Methodology

In the following sections, first we provide a high-level overview of the proposed static analysis tool's workflow. Next, we expose a methodology for choosing project-specific quality thresholds for a given toolkit. Finally, we use PETSc as a use case, and demonstrate that the gathered data can be used to answer quality-related questions with efficiency.

A. Overview of proposed tool

This section describes our static analysis tool which relies on the Clang/LLVM infrastructure to collect code quality data, given an HPC application. As Figure 1 indicates, given an HPC application and an architecture on which the application is to be built, first a compilation database—usually a JSON file specifying compile commands for each source file configured to be built—is generated. This can either be done directly using *cmake* [12] by turning on the `-DCMAKE_EXPORT_COMPILE_COMMANDS`, or, for applications that do not support a *cmake*-build, by using the *bear* [13] tool.

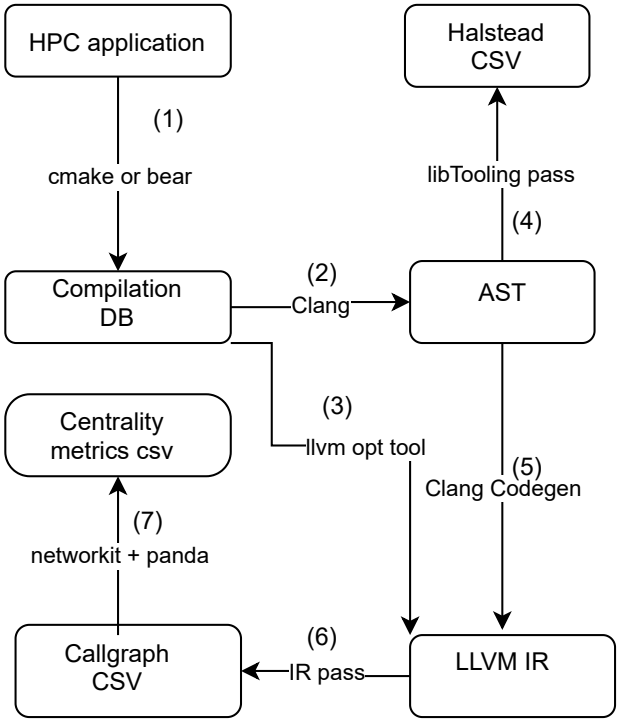


Fig. 1: Overview of the static analysis tool.

Once all files to be compiled, as well as their corresponding full compile commands are known and stored in the compilation database (the JSON file), this file is post processed by our tool[2]. This is done so that each source file's corresponding *Abstract Syntax Tree* (AST) as well as its *LLVM Intermediate Language* (LLVM IR) can be generated rather than the original object code—see arrows (2) and (3) in Figure 1.

As a result, the next phase is a pass that takes in a *clang-ast* per file, and uses *clang-LibTooling* to walker over it collecting Halstead metrics into a spreadsheet which is exported—arrow (4) of Figure 1. Similarly, the phase represented by arrow (6) of Figure 1 is an IR pass that is written to export the file's callgraph into a spreadsheet which is then fed into a *python* script. This script—as arrow (7) indicates— utilizes *pandas* [14] and *networkit* to collect the callgraph's centrality metrics into their own spreadsheet.

Finally, the two spreadsheets—one from step (4) and the other from step (7)—are merged together to form the overall dataset (see section III-B for a description of the structure of the resulting dataset).

1) Tool Constraints:

- As the numbered arrows of Figure 1 indicate, our tool requires a few external dependencies. First, in addition to the requirement that its source code be written in C or C++ (*step 1*), the codebase should detail the process of building it with *cmake*. Next, the system needs to have

[2]Notice that we never use the linker. We assume that our results will remain the same pre- as well as post- linking and therefore choose to avoid the additional time overhead.

Software	Version
llvm-config	13.0.0git
opt	LLVM version 13.0.0
clang	clang version 13.0.0
python	3.6.9
pandas	1.1.5
networkx	2.5.1
neworkit	9.0

TABLE I: Tool Dependency Versions

a version of *clang* with *libTooling* support (*steps 2 and 4*). A copy of LLVM needs to be installed on the system as well (*step 3, 5, and 6*), and finally, *python* with the *networkit, networkx,* and *pandas* packages needs to be installed.

- The *callgraph* of *step 6* is generated on a *local* basis—meaning that a callgraph is generated per *LLVM IR module*, pre-linking.
- Because the usual callgraphs generated by *clang* have one node into which all indirect calls point, we were forced to write our own custom callgraph pass. The custom callgraph pass we wrote is able to handle many *indirect calls*, and each of the identified such calls is counted towards the *FanOut* value of its parent method. In order to do this, first, we compile source programs so that results of the LLVM project's implementation of *type-based alias analysis* [15] are still attached to IR instructions. Then we rely on the fact that a pointer to the code of any function that can be called indirectly, already has to be loaded, and as a consequence, keep a stack of the load instructions in the LLVM IR at the beginning of every function. When a load instruction is encountered in the body of the function, it is pushed onto the stack; a process which goes on until an indirect call is encountered. When an indirect call is encountered, we start popping instructions off the stack and comparing the type of the popped instruction, with the type of the indirect call. If the two types match, we stop and construct a name for the indirect call's pointer based on results of *type based alias analysis* attached to the load instruction. Otherwise, we keep popping off the stack. When this heuristic fails, a "fail" node is added to the callgraph instead.
- See Table I for the comprehensive list of software versions used in the tool as of this writing.

B. Dataset Structure

As previously mentioned, the proposed tool was used to extract data from 3 commonly used HPC libraries. Table IV records some demographic information about the libraries, while Table III lays out the dataset structure.

C. HPC Toolkits

The tool traverses modern HPC software packages and outputs a number of code quality and call graph metrics.

TABLE II: Overview of Halstead and graph theory-based metrics.

Metric	Description
mu1	The number of unique operators
mu2	The number of unique operands
N1	Total occurrences of operators
N2	Total occurrences of operands
N	Length = N1 + N2
mu	Vocabulary = mu1 + mu2
mu1'	Potential operator count
mu2'	Potential operand count = number of function arguments
V	Volume $= N * \log_2(mu) \rightarrow$ the number of mental comparisons needed to write a program of length N
V^*	Volume on minimal implementation $= (2 + mu2') * \log_2(2 + mu2')$
L	Program length $= \frac{V^*}{N}$
D	Difficulty $= \frac{1}{L}$
I	Intelligence $= L' * V^*$
E	Effort to write program $= \frac{V}{L}$
T	Time to write program $= \frac{E}{18}$
CC	Cyclomatic complexity
FanIn	Total number of callers
FanOut	Total number of callees
BC	Betweenness centrality metric representing a node's influence over information flow in a graph
Closeness	Centrality metric that measures a node's capability for efficiently spreading information
ASPath	Average shortest path

TABLE III: Overview of Dataset Structure

Variable Name	Description
V1	Name of HPC Library
V2	Name of function
V3-V18	Used for 16 Halstead code quality metrics
V19-V24	Used for graph theory related metrics

We consider three such toolkits namely PETSC, SLEPC, and SuperLU. PETSc (Portable, Extensible Toolkit for Scientific Computation) is a widely used HPC software toolkit which enables the large-scale parallel solution of partial differential equations in applications using C, C++, Fortran, and Python. SLEPc is a tookit built on top of PETSc for parallel computation of eigenvectors and eigenvalues of sparse matrices. SLEPc developers can use basic constructs of PETSc such as its data structures, error checking and automatic profiling. Finally, SuperLU is used to compute the solution of large, nonsymmetric, and sparse systems of linear equations. All three tookits feature parallel implementations based on MPI, OpenMP, and CUDA. See Table IV for the descriptive statistics of these tookits.

TABLE IV: Descriptive statistics of HPC toolkits.

HPC Pkg Name	Num. of Methods	URL
PETSc	718,497	https://petsc.org/
SLEPc	210,626	https://slepc.upv.es/
SuperLU	71,552	https://github.com/xiaoyeli/superlu_dist

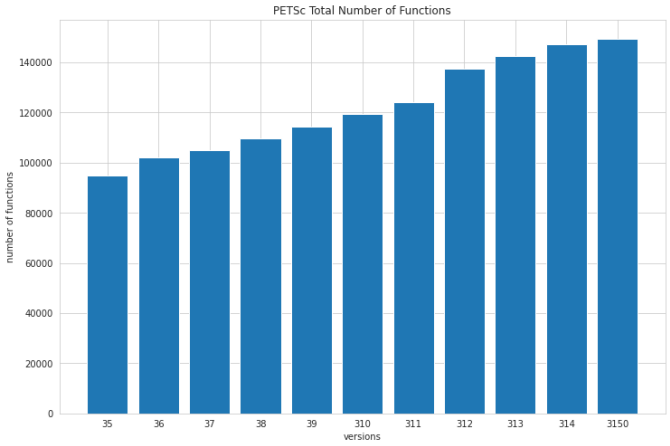


Fig. 2: Total Collected Functions per Version of PETSc.

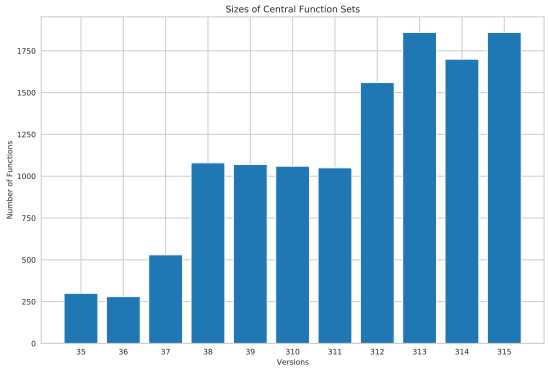


Fig. 3: Sizes of Central Functions in PETSc

IV. A PETSc Use Case : Analyzing the Most Critical Functions

In this section, we describe how data collected by the tool of section III-A was used to identify and generate quality history plots for the most critical functions of PETSc. As Figure 2 indicates, we were able to collect data for 11 versions of PETSc, starting with PETSc-3.5 and ending with PETSc-3.15.0.

In the following,**we make a comparison between two methods used for identifying the most critical functions in each of the studied PETSc versions**. In the first method, first we calculated the *radius* of the callgraph associated with each version's source code, then we selected those functions whose eccentricity equals the radius and treated those as the most central. Next, we removed functions whose *betweenness* was not at least zero, thereby keeping only those functions which appear on at least one independent path of the callgraph. Figure 3 displays the sizes of the resulting sets over time. As can be seen, the more project grew, so did the number of its most critical functions.

The second method uses *PageRank* [16] to sort functions according to their influence on other functions in the callgraph. Figure 4 displays the overlap between functions returned by the eccentricity-based ranking and the top n pageranked

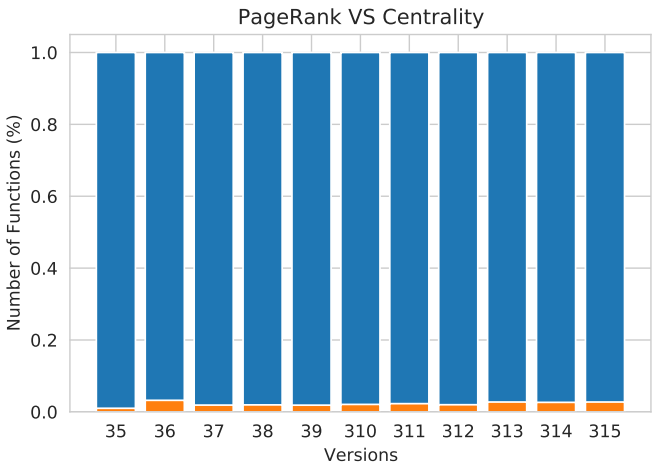


Fig. 4: Centrality VS PageRank in PETSc

functions, where n equals the size of the set returned by the eccentricity based ranking described above. As can be seen, we found little to no overlap between the two sets of functions throughout.

Next, we sought to analyze **how critical the resulting functions really were**. In Figures 6 and 7, we plot the percentage *average shortest path, cyclomatic complexity, fan in,* as well as the total *betweenness* values, accounted for by the resulting most critical functions using the *eccentricity-based* ranking and *pagerank* respectively.

We find that, functions whose eccentricity equals the radius of the callgraph in PETSc, account for around 35% of *average shortest paths* on average. This means that in any random experiment where every function in a selection of 100 PETSc functions is called, around 35 functions in the selection will trigger a central function on average. Therefore, for applications that use PETSc, an error in any one of those central functions will manifest itself relatively quickly.

Perhaps surprisingly however, the influence of these functions (those found via eccentricity-based ranking), seems to dissipate when *betweenness* (or *cyclomatic complexity*) rather than *average shortest path* is considered for an indicator of frequency of use. As figure 6 indicates, their *cyclomatic complexity* influence is almost non-existent and their influence on the overall *betweenness* values looks negligible.

When one remembers the size of the central functions computed via the radius of the callgraph, and considers only that many among the top results of pagerank however, the central functions returned account for non-negligible percentages of not only *fan-in* and *average shortest path*, but also *betweenness* and *cyclomatic complexity*.

Thus, it seems as though the *PageRank* algorithm finds a much more critical set of functions than the eccentricity-based search algorithm. However, due to the negligible overlap between the central functions found by the two methods, and because the eccentricity-based ranking also returns a very critical set of functions in terms of *average shortest path*

and *fan in*, we find that using both methods rather than just *PageRank* might yields better analysis results.

After investigating how critical the found functions really were, we sought to demonstrate how the data returned by our tool can be used to plot the histories of every function in the source code in terms of various centrality and quality metrics, starting with those most critical.

Figure 9 displays the history of the top 10 most central functions in the oldest of PETSc (PETSc-3.5) found via eccentricity-based ranking, while Figure 8 displays the history of the top page-ranked functions in the latest version of PETSc (PETSc-3.15). In both figures, the history in terms of the function's *average shortest path* is displayed on the left, while the history in terms of *betweenness* is displayed on the right.

From the figure depicting the history of the top 10 functions in the oldest version of PETSc, we can infer for example, that PETSc version 3.8 retired the `PetscStrcmpNoError` function. In addition, we are able to see that, even though the total use of `PetscCheckPointer` on independent paths of the callgraph of PETSc was significantly diminished (its *betweenness* went down) in version 3.10, its total use in terms of *average shortest path* went up significantly, suggesting that the total number of unique functions in which valid pointer checks were made, significantly went up.

Moreover, much like Figure 9 allowed us to visualize and identify the moment when a given critical function was put out of use, from Figure 8 inferences about versions in which a number of functions were introduced can be made: `KSPCGGetObjFcn` in PETSc-3.7, `TaoBoundSolution`, `TaoBNKRecomputePred`, `TaoBoundSolution`, `TaoBNKPerformLineSearch` in PETSc-3.9.

Furthermore, we can see that `PETScError` dominates *betweenness* calculations, and should probably be removed before any further inferences resulting from analyzing the metric could be made.

Finally, Figure 10 depicts the history of the top 10 page-ranked functions in terms of each function's quality—measured in terms of *cyclomatic complexity* (left), as well as the function's *total number of operations* (right). From the figures, functions such as `TaoBNKTakeCGSteps`, `KSPCGGetObjFn` and `TaoCreate_BNK` can be identified as functions whose active development seems to have steadied because of their almost constant *cylomatic complexity* and *total number of operations* since version PETSc-3.13. This is in opposition to the remainder of the found critical functions as development for these seems to be active and on-going.

As such, using techniques illustrated in this section, the callgraph of sizeable application such as PETSc can be used to identify the most critical functions in the application. Their perceived relative influence can be investigated, and the history both of their use and their quality in terms of various metrics can be visualized and thereby any functions that domain scientists might deem to require more attention, can be found.

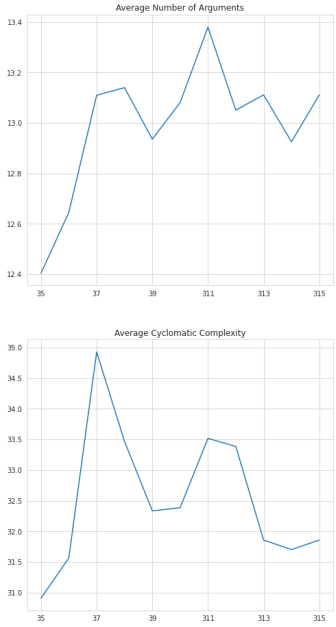


Fig. 5: PETSc Number of Arguments and CC Trends

A. Threshold Identification

As section I mentions, each of the structural code quality metrics needs a threshold that can be used to sort sections of a codebase into problematic versus non-problematic categories. For *cyclomatic complexity*, the default suggested by the recent standard on *Information Technology - Software Quality Measurement - Automated Source Code Quality Measures* (ISO/IEC 5055:2021 [17]), is 20, while for the number of function arguments, the number is 7. However, these numbers may need to vary for different specialized software, especially those whose specific problem domain might dictate unusual numbers. For example, from Figure 5 it can be seen that the average *cyclomatic complexity* value for any version of PETSc (starting from 3.5 to 3.15) is always above the recommended, and similary for the average number of arguments per function. This unusual behavior leads us to conclude that any code quality tool, should provide users with means to customize thresholds based on their application specific knowledge. This lead us to investigate a method by which desired subjective metrics can be identified through statistical analysis. This section describes the proposed method.

The process has three steps. First, the tool from section III-A is used to gather data for the metrics of interest. Second, a type of probability distribution from which data collected per metric, could be considered to be a sample, is identified. Finally, metric thresholds are recommended based on the fit from the resulting distribution.

For the probability distribution of best fit, we used the Easyfit tool [18]. Easyfit has a large collection of potential probability distributions. However, due to the unbounded nature of the values of our metrics, we only consider the Beta, Gamma, Logistic, Cauchy, Weibull, and Exponential

distributions for potential fits.

Recall that a probability distribution is described through either its PDF (Probability Density Function) or its CDF (Cumulative Distribution Function). The PDF is often represented as f(x) and describes the probability that a random variable equals to the value x. Similarly, the CDF is usually represented as F(x) and describes the probability that a random variable is less than or equal to the value of x.

For each metric, the Easyfit tool fits our data with each probability distribution and ranks the fits through the Kolmogorov Smirnov test—a non parametric test which computes the distance between empirical and cumulative distribution functions of sample and reference distributions respectively [19].

Once the probability distribution for a given metric is known, the shape of its PDF is manually used to identify the appropriate reference threshold values for the metric. Throughout, we take the low value to refer to the start of the curve that fits the most data, while the high value refers to the value from which the skewness of the curve starts. The values inside this range, we take to be the feasible values. Tables V through VII contain the results of our analysis, complete with the distribution of best fit, as well as threshold values for each metric; for all three HPC toolkits considered in our study (see section III-C).

TABLE V: Threshold values for PETSc.

Metric	**Prob. Dist.**	**Thresholds Reference Values**		
		Low	Range	High
mu1	Weibull	≤ 9	10-59	≥ 60
mu2	Gamma	≤ 10	11-219	≥ 220
N1	Exponential	≤ 200	201-630	≥ 631
N2	Weibull	≤ 250	251-1200	≥ 1201
N	Weibull	≤ 600	601-1900	≥ 1901
mu	Weibull	≤ 50	51-198	≥ 199
mu1'	Cauchy	≤ 0.30	0.31-0.68	≥ 0.69
mu2'	Beta	≤ 5	6-16	≥ 17
V	Weibull	≤ 4000	4001-14000	≥ 14001
V*	Beta	≤ 28	29-122	≥ 123
L	Weibull	≤ 0.10	0.11-0.40	≥ 0.41
D	Weibull	≤ 55	56-190	≥ 191
I	Weibull	≤ 7	8-32	≥ 33
E	Weibull	≤ 40000	40001-200000	≥ 200001
T	Weibull	≤ 25000	25001-10000	≥ 10001
CC	Weibull	≤ 400	401-800	≥ 801
FanIn	Weibull	≤ 500	501-1500	≥ 1501
FanOut	Weibull	≤ 10	11-70	≥ 71
BC	Weibull	≤ 16000	16001-64000	≥ 64001
Closeness	Beta	≤ 0.002	0.003-0.006	≥ 0.007
ASPath	Gamma	≤ 4	5-800	≥ 801

The metric thresholds identified can then be used to pick out code smells by looking at each method's score and whether it falls below, in or above the reference threshold range. For example, in the case of PETSc (Table V), we can recognize that 400 starts the feasible range of CC values—which most likely refer to functions that make appropriate use of multiple and nested if statement and loops. Similarly, any function whose CC value is greater than 800 (High value of CC) probably needs a developer's attention and should probably be split into multiple functions.

TABLE VI: Threshold values for SLEPc.

Metric	**Prob. Dist.**	**Thresholds Reference Values**		
		Low	Range	High
mu1	Gamma	≤ 8	9-25	≥ 26
mu2	Exponential	≤ 50	51-180	≥ 181
N1	Weibull	≤ 1600	1601-4000	≥ 4001
N2	Weibull	≤ 2500	2501-9000	≥ 9001
N	Weibull	≤ 4000	4001-12000	≥ 12001
mu	Exponential	≤ 90	91-180	≥ 181
mu1'	Gamma	≤ 0.30	0.4-0.6	≥ 0.7
mu2'	Beta	≤ 4	5-22	≥ 23
V	Weibull	≤ 20000	20001-60000	≥ 60001
V*	Gamma	≤ 8	9-28	≥ 29
L	Weibull	≤ 0.10	0.2-0.4	≥ 0.5
D	Weibull	≤ 400	401-1500	≥ 1501
I	Weibull	≤ 2	3-7	≥ 8
E	Weibull	≤ 40000	40001-200000	≥ 200001
T	Weibull	≤ 25000	25001-10000	≥ 10001
CC	Weibull	300	301-1200	1201
FanIn	Weibull	≤ 50	51-260	≥ 261
FanOut	Weibull	≤ 7	8-35	≥ 36
BC	Weibull	≤ 100	101-400	≥ 401
Closeness	Gamma	≤ 0.002	0.003-0.008	≥ 0.009
ASPath	Gamma	≤ 15	16-450	≥ 451

TABLE VII: Threshold values for SuperLU.

Metric	**Prob. Dist.**	**Thresholds Reference Values**		
		Low	Range	High
mu1	Gamma	≤ 8	9-36	≥ 37
mu2	Weibull	≤ 50	51-180	≥ 181
N1	Weibull	≤ 2500	2501-10000	≥ 10001
N2	Weibull	≤ 2500	2501-17500	≥ 17501
N	Weibull	≤ 4000	4001-12000	≥ 12001
mu	Weibull	≤ 50	51-180	≥ 181
mu1'	Gamma	≤ 0.30	0.4-0.6	≥ 0.7
mu2'	Weibull	≤ 4	5-17	≥ 18
V	Weibull	≤ 40000	40001-240000	≥ 240001
V*	Gamma	≤ 20	21-60	≥ 61
L	Weibull	≤ 0.10	0.2-0.4	≥ 0.5
D	Weibull	≤ 100	101-400	≥ 401
I	Weibull	≤ 2	3-10	≥ 11
E	Weibull	≤ 25000	25001-100000	≥ 100001
T	Weibull	≤ 1000	1001-5000	≥ 5001
CC	Weibull	50	31-1000	1001
FanIn	Gamma	≤ 8	9-19	≥ 20
FanOut	Gamma	≤ 3	4-12	≥ 13
BC	Weibull	≤ 50	51-240	≥ 241
Closeness	Weibull	≤ 0.002	0.003-0.008	≥ 0.009
ASPath	Gamma	≤ 100	101-300	≥ 301

In conclusion, the main consequences of our threhold-finding experiments can be summarized as follows.

- Overall, we find that we cannot recommend a single probability distribution to identify the thresholds of all metrics.
- The above point is true both within a project for different metrics, and across projects even for the same metric. However, because each application's domain is different and might heavily influence the code's design, we do not find this to be a cause for concern. On the contrary, these project-specific distributions allow us to set project-specific thresholds allowing us to reliably detect undesirable trends within a project quickly.
- The Weibull distribution if most effective to fit most

of our metrics data of all the other probabilities we considered in our study.

V. Conclusion

We have introduced a Clang/LLVM-based static analysis tool to collect software quality related data from large applications. We sketched out its design that relies on compilation databases exported by *cmake* to replace the compiler used in a project with *clang* and use a custom written *libTooling* tool and an *LLVM IR* custom callgraph pass to collect AST related data, and export the project's static callgraph respectively. We described how *pandas* and *networkit* are then used to collect centrality data from the callgraph.

As a use-case, we demonstrated how we were able to use the data collected by our tool to identify the most critical functions of multiplef versions of PETSc, using two different methods—an eccentricity based ranking of functions, and *PageRanked*. We compared results returned by both methods, and plotted each of the resulting functions across time, which allowed to make judgements about when different influential functions were introduced in the project or when they were put out of use which might point to shifts in the project's areas of focus.

Finally, we presented a method for identifying thresholds for each metrics, thereby simplifying the process of spotting coding smells from such applications. We demonstrated how we used the proposed tool to extract data from three well-know HPC libraries, namely PETSc, SLEPc, and SuperLU, and exposed how we used the EasyFit tool to choose the best distribution used to fit our metrics data. Finally we presented reference thresholds—to be used for coding smell identification—that resulted from a manual inspection of the shape of the distributions returned by EasyFit.

VI. Related Work

The current work presented a tool based on Clang/LLVM and explained how the tool was used to collect a number of function-level code quality metrics, and in addition, used the collected data to rank functions in source code according to their relative influences, and evaluated the resulting functions in terms of their actual influence on the rest of the code-base.

Honglei et al [20] reviews a number of software metrics and complexity metrics, including the Halstead, and McCabe's metrics used in this work.

Vytovtov et al [21] collected a number of Halstead metrics using an LLVM-based tool and sought to classify source code sections through the K-means algorithm.

Sora et al [22] relied on PageRank in a recommender system they designed for identifying key classes in software systems by first modeling static dependencies in the application of interest as a directed graph.

Zhang et al [23] used a decomposition algorithm based on the k-core—another graph centrality metric—of a graph, to analyze the static structure of large-scale software systems.

Paloma et al [24] propose a different method for identifying relative thresholds for source code metrics by identifying a value p and a value k for every metric M, such that $p\%$ of the classes in the system have metric value $M \leq k$.

VII. Future Work

As it stands, the tool introduced collects code quality metric data at the function-level. In the future, we plan to include metrics at different other levels of granularity including the class-level, the module-level, data about requirements and tests, etc. Additionally, even though we demonstrated that some of the centrality data collected point to interesting aspects of the source code, we still need to investigate whether the structural code quality metrics data collected have any relationship to other aspects of the development process. An investigation of such a nature is planned for the future. Furthermore, due to the unusually large metric thresholds that were found for some of the metrics, the manual threshold approximation used to after a probability distribution has been identified and plotted is to be revised, automated, and its performance on unseen data analyzed.

Finally, we plan to investigate the relationship between the data collected by this static analysis tool, and a corresponding dynamic analysis tool to uncover the effect of modern dynamic programming language features like polymorphism and overloading.

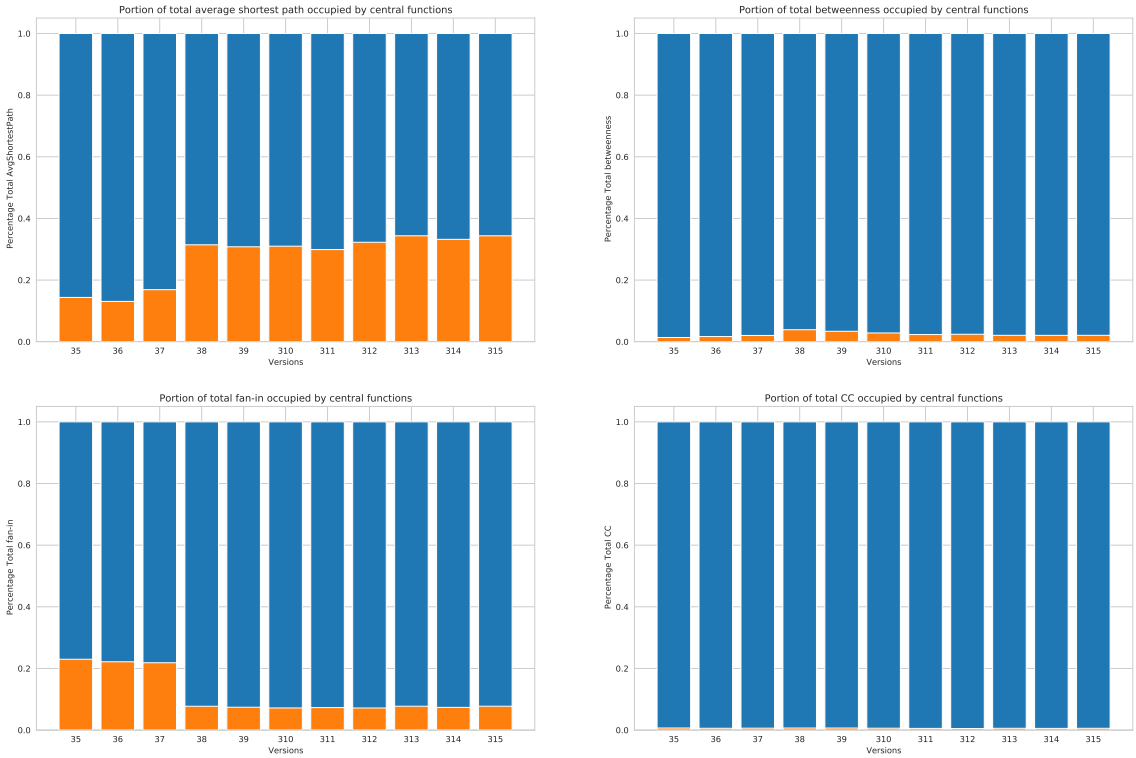


Fig. 6: How Critical Are Central Functions in PETSc?

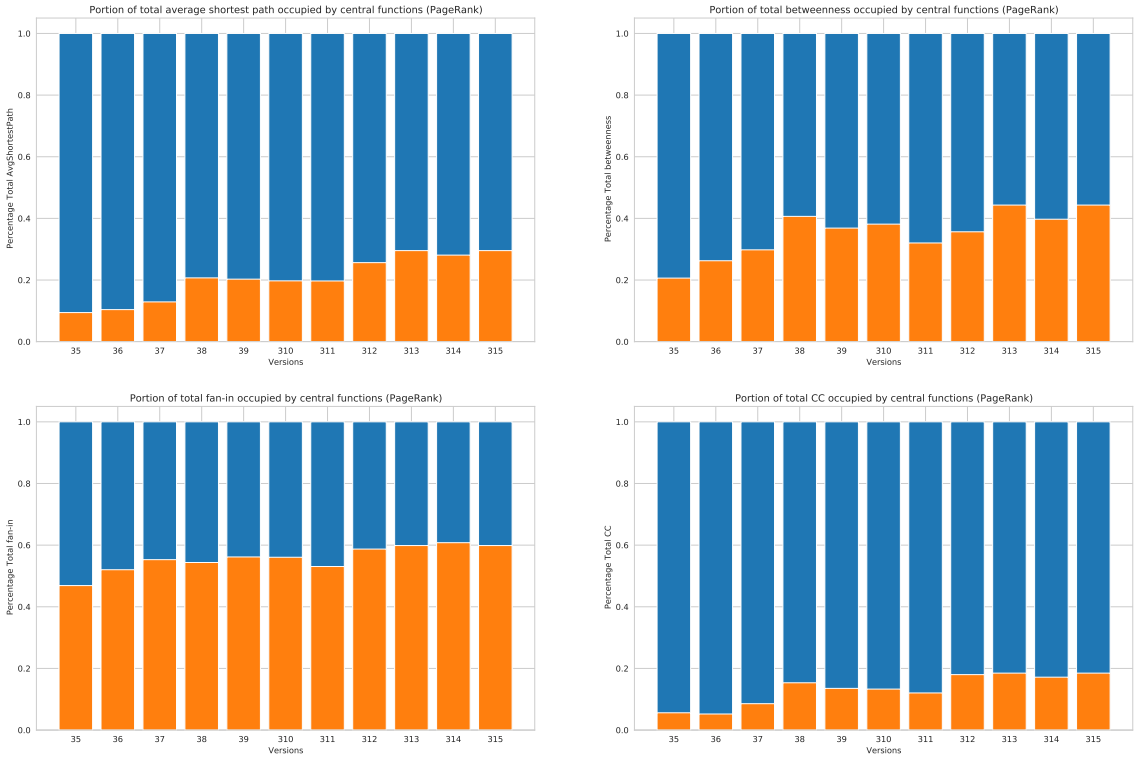


Fig. 7: How Critical Are PageRank Based Central Functions in PETSc?

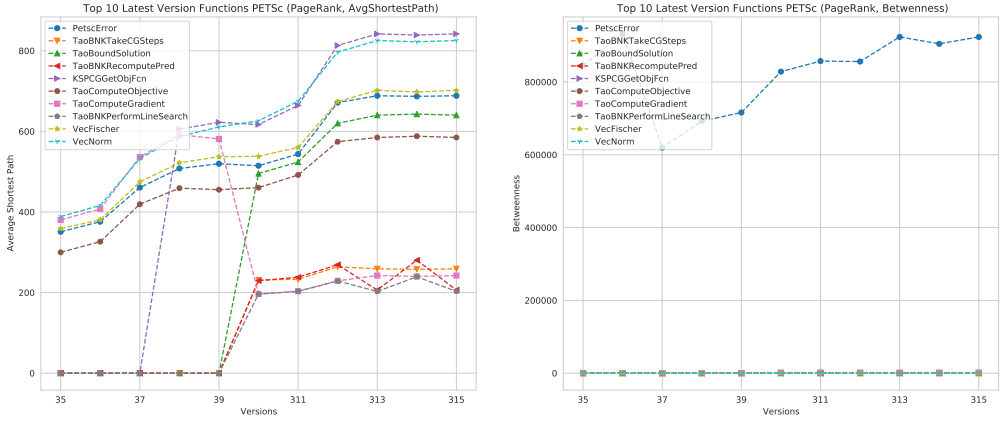


Fig. 8: History of the Top 10 Functions in the Latest Version of PETSc.

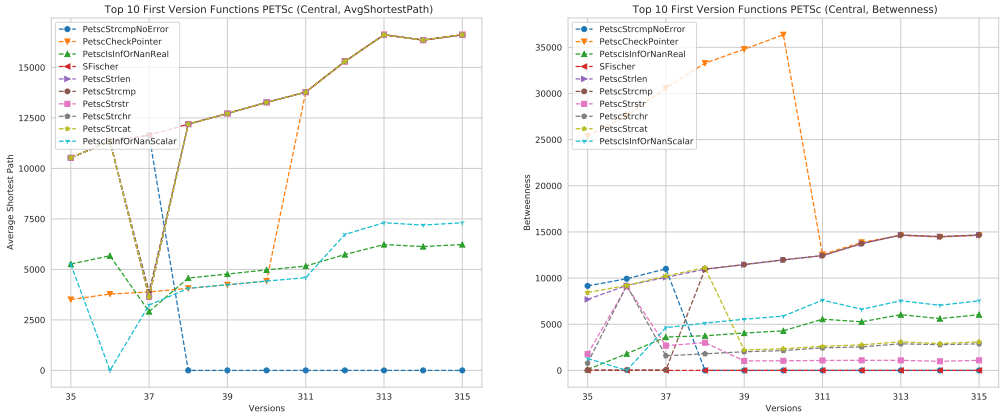
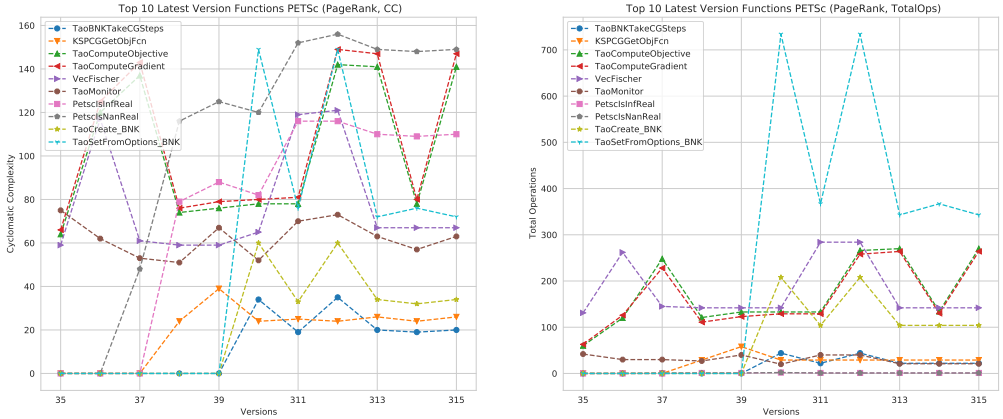


Fig. 9: History of the Top 10 Functions in the Oldest Version of PETSc.

Fig. 10: Quality History of Top PageRanked Functions in the Latest Version of PETSc.



REFERENCES

[1] "Cppdepend - code quality metrics," https://www.cppdepend.com/metricsFeature, (Accessed on 10/09/2021).

[2] "Sourcemonitor," https://www.campwoodsw.com/sourcemonitor.html, (Accessed on 10/09/2021).

[3] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[4] M. H. Halstead, "Elements of software science," USA, 1977.

[5] C. L. Staudt, A. Sazonovs, and H. Meyerhenke, "Networkit: A tool suite for large-scale complex network analysis," *Network Science*, vol. 4, no. 4, pp. 508–530, 2016.

[6] T. M. Kelley, "Code analysis and refactoring with clang tools, version 0.1, version 00," 12 2016. [Online]. Available: https://www.osti.gov//servlets/purl/1337570

[7] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, D. Karpeyev, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, "PETSc users manual," Argonne National Laboratory, Tech. Rep. ANL-95/11 - Revision 3.15, 2021. [Online]. Available: https://www.mcs.anl.gov/petsc

[8] SuperLU, "Superlu," Available at https://github.com/xiaoyeli/superlu (1997/04/02).

[9] J. E. Roman, C. Campos, L. Dalcin, E. Romero, and A. Tomas, "SLEPc: Scalable library for eigenvalue problem computations," Available at https://github.com/firedrakeproject/slepc. Year=2017.

[10] K. A. Ferreira, M. A. Bigonha, R. S. Bigonha, L. F. Mendes, and H. C. Almeida, "Identifying thresholds for object-oriented software metrics," *Journal of Systems and Software*, vol. 85, no. 2, pp. 244–257, 2012, special issue with selected papers from the 23rd Brazilian Symposium on Software Engineering. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121211001385

[11] W. Landi, "Undecidability of static analysis," *ACM Letters on Programming Languages and Systems (LOPLAS)*, vol. 1, no. 4, pp. 323–337, 1992.

[12] "Documentation — cmake," https://cmake.org/documentation/, (Accessed on 10/10/2021).

[13] "rizsotto/bear: Bear is a tool that generates a compilation database for clang tooling." https://github.com/rizsotto/Bear, (Accessed on 10/10/2021).

[14] W. McKinney *et al.*, "pandas: a foundational python library for data analysis and statistics," *Python for high performance and scientific computing*, vol. 14, no. 9, pp. 1–9, 2011.

[15] A. Diwan, K. S. McKinley, and J. E. B. Moss, "Type-based alias analysis," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, ser. PLDI '98. New York, NY, USA: Association for Computing Machinery, 1998, p. 106–117. [Online]. Available: https://doi.org/10.1145/277650.277670

[16] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer networks and ISDN systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[17] "Iso - iso/iec 5055:2021 - information technology — software measurement — software quality measurement — automated source code quality measures," https://www.iso.org/standard/80623.html, (Accessed on 11/29/2021).

[18] Mathwave, "Easyfit," http://www.mathwave.com/products/easyfit.html, 2010.

[19] F. J. Massey Jr, "The kolmogorov-smirnov test for goodness of fit," *Journal of the American statistical Association*, vol. 46, no. 253, pp. 68–78, 1951.

[20] T. Honglei, S. Wei, and Z. Yanan, "The research on software metrics and software complexity metrics," in *2009 International Forum on Computer Science-Technology and Applications*, vol. 1, 2009, pp. 131–136.

[21] P. Vytovtov and E. Markov, "Source code quality classification based on software metrics," in *2017 20th Conference of Open Innovations Association (FRUCT)*, 2017, pp. 505–511.

[22] I. Şora, "A pagerank based recommender system for identifying key classes in software systems," 05 2015, pp. 495–500.

[23] H. Zhang, H. Zhao, W. Cai, J. Liu, and W. Zhou, "Using the k-core decomposition to analyze the static structure of large-scale software systems," *The Journal of Supercomputing*, vol. 53, no. 2, pp. 352–369, 2010.

[24] P. Oliveira, M. T. Valente, and F. P. Lima, "Extracting relative thresholds for source code metrics," in *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, 2014, pp. 254–263.