Automatically Computing Program Path Complexity

Lucas Bang, Abdulbaki Aydin, and Tevfik Bultan

I. Introduction

Recent automated software testing techniques focus on achieving path coverage. We present a program complexity measure, *path complexity*, that provides an upper bound for the number of paths in a program, and hence, can be used for assessing the difficulty of achieving path coverage for a given method during automated testing.

We define the *path complexity* of a program as a function $path(n)$ that takes a depth bound n as input: $path(n)$ returns the number of paths π in the program's control flow graph (CFG) from start to exit with length $L_\pi \leq n$, where L_π is the number of edges in π. Consider the CFG given in Fig. 1(b). It is not possible to have a path with $L_\pi \leq 5$ so $path(n) = 0$ for $0 \leq n \leq 5$. By inspection we can see that there are 10 different paths with $L_\pi \leq 12$, so $path(12) = 10$. It turns out that $path(n) = O(n^2)$ due to the two independent loops.

In this paper, we show how to automatically compute a sound upper bound for $path(n)$ in closed form, and the *asymptotic path complexity* which identifies the dominant term in the path complexity function. Path complexity focuses only on the control flow structure and cannot determine the difficulty of finding input values that can exercise a certain branch condition. However, due to this abstraction, path complexity can be computed efficiently. We describe the details of computing our path complexity metric in the next section.

II. Computing Path Complexity

We make use of techniques from algebraic graph theory and analytic combinatorics to count the number of execution paths of a CFG [2], [6], [3]. Given a CGF G with nodes N and edges E, and a length n, we can compute the generating function $g(z)$ such that the n^{th} Taylor series coefficient of $g(z)$, denoted $[z^n]g(z)$, is equal to $path(n)$:

$$g(z) = \frac{\det(I - zT : |N|, 1)}{(-1)^{|N|+1}\det(I - zT)}. \quad (1)$$

where T is the *augmented transfer-matrix* (an adjacency matrix with $T_{|N|,|N|} = 1$), $(M : i, j)$ denotes the matrix obtained by removing row i and column j from M, and I is the identity matrix.

From $g(z) = p(z)/q(z)$ we can derive a closed-form function $bnd(n)$ as a sum of products of simple polynomial and exponential terms such that $path(n) = O(bnd(n))$. The form of $bnd(n)$ is determined by

$$bnd(n) = \sum_{i=1}^{D} \sum_{j=0}^{m_i - 1} c_{i,j} n^j \left(\frac{1}{|r_i|}\right)^n, \quad (2)$$

where $q(z)$ had D distinct roots, r_i is the i^{th} root of $q(z)$, m_i is the multiplicity of r_i, and c_i are coefficients that are determined by the first $|N|$ terms of the Taylor series expansion of $g(z)$. Since $path(n) = [z^n]g(z)$ for all n, we can define a system of $|N|$ equations and $|N|$ unknowns. This system can be solved for the coefficients $c_{i,j}$ via linear algebra. This gives a closed form function for $bnd(n)$.

We extract the dominant term $f(n)$ from $bnd(n)$ using standard asymptotic analysis, where $bnd(n) = \Theta(f(n))$ if and only if $\lim_{n\to\infty} \frac{bnd(n)}{f(n)} = 1$. This allows one to determine if the path complexity of a program is asymptotic to a constant, n, n^2, n^3, and so on, or b^n for some exponential base b. Table I shows the asymptotic path complexity for the CFG in Fig. 1(b) to be quadratic as expected.

All of these steps can be carried out automatically, and we discuss our implementation in Section IV.

III. Comparison with other complexity measures.

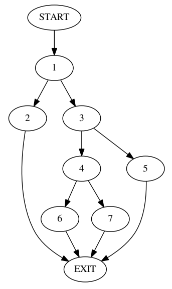
We demonstrate that path complexity is a better complexity measure for path coverage compared to two well-known program complexity measures. We compare our methods to

1) *Cyclomatic complexity* [4], the number of linearly independent paths in G, computed as $|E| - |N| + 2$, and
2) *NPATH complexity* [5], the number of acyclic paths in a program, computed dynamically using the recurrence $nPath(u, G) = \sum_{e_{uv} \in G} nPath(v, G - e_{uv})$, where e_{uv} is the edge from node u to node v in G.

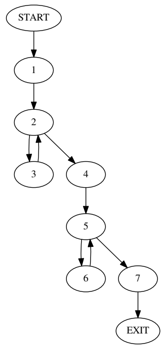
In Fig. 1 we show the CFGs for three sample methods selected from the Java 7 SDK: (a) an array out of bounds checking function, (b) a search resetting function for regular expression matching, and (c) binary search for a sorted array. One should expect the three methods to be increasingly costly in terms of performing automated testing; CFG (a) is simplest with only nested conditional branching, CFG (b) has non-interleaved loops, and (c) is most complex with nested conditional branching within a loop. In Table I we give the cyclomatic, NPATH, and path complexities for these functions.

One can see that both cyclomatic and NPATH complexity give only constant numbers as complexity measures. Cyclomatic complexity is 4 for CFGs (a) and (c) and 3 for CFG (b) while NPATH complexity is 4 for all three methods. That is, cyclomatic complexity is insufficient for distinguishing the methods and NPATH cannot distinguish between the three methods at all. Neither cyclomatic nor NPATH complexity are adequate measures for assessing difficulty of path coverage of a program since cyclomatic complexity does not consider paths that are subsumed by other paths and the NPATH complexity does not consider multiple executions of loops.

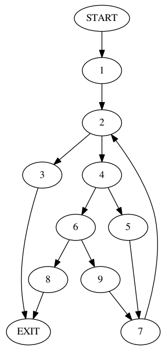
However, path complexity does capture the difference by providing three different complexity measures for the three methods. We see that CFG (a) has constant path complexity,



(a) CFG for `Arrays.rangeCheck`



(b) CFG for `regex.Matcher.reset`



(c) CFG for `Arrays.binSearch`

Fig. 1: CFGs for three methods from Java 7 SDK's `java.utils` package.

TABLE I: Results of different complexity measures from JAVA 7's `java.utils`.

Method	Cyclomatic Complexity	NPATH Complexity	Path Complexity	Asymptotic Path Complexity
Fig. 1(a): `Arrays.rangeCheck`	4	4	4	$O(1)$
Fig. 1(b): `regex.Matcher.reset`	3	4	$0.12 \times n^2 + 1.25 \times n + 3$	$O(n^2)$
Fig. 1(c): `Arrays.binSearch`	4	4	$6.86 \times 1.17^n + 0.22 \times 1.1^n + 0.13 \times 0.84^n + 2$	$O(1.17^n)$

CFG (b) has quadratic path complexity, and CFG (c) has exponential path complexity.

IV. Implementation and Experiments

We implemented the analysis of Section II in a tool called PAth Complexity analyzer (PAC). PAC accepts a Java class file or a jar of class files as input and reports path complexity, asymptotic path complexity, cyclomatic complexity, and NPATH complexity for each method in the provided classes. An online demo of PAC and source code for PAC are available from our website.[1]

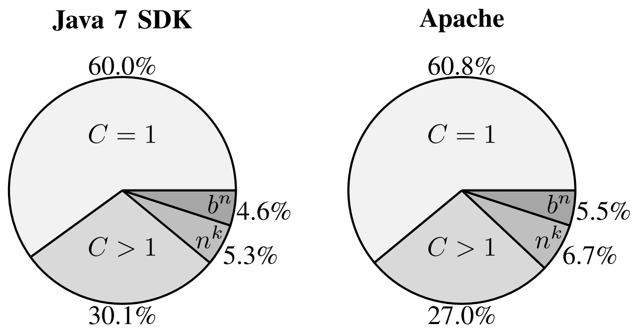


Fig. 2: Distribution of methods for four complexity classes: 1) single path [$C = 1$], 2) constant number of paths greater than one [$C > 1$], 3) polynomial [n^k, $k \geq 1$], and 4) exponential [b^n, $n > 1$].

We tested PAC by running it on the Java 7 SDK and the Apache Commons Libraries. In Fig. 2 we give the distributions of path complexities for the analyzed methods. We find that for the Java 7 SDK, 60% of the methods have only a single path (do not contain branching), 30.1% have a constant number of paths larger that 1 (contain only branching), 5.3% have polynomial path complexity (contain non-interleaved loops without nested branching), and 4.6% have exponential complexity (contain interleaved loops or loops with nested conditional branching). We observe similar results for the Apache Commons Libraries. More detailed experimental results are provided in [1]. All experimental results are available from our website.[1]

[1]`https://vlab.cs.ucsb.edu/PAC/`

V. Conclusion

We introduced a new metric for automated software testing, called path complexity, and showed how to compute it automatically. We demonstrated that asymptotic path complexity provides a better measure of the difficulty of achieving path coverage compared to cyclomatic and NPATH complexity. In addition, we built a publicly available tool, PAC, based on our techniques, and conducted an experimental evaluation of our metric on two large, popular Java libraries.

References

[1] L. Bang, A. Aydin, and T. Bultan. Automatically computing path complexity of programs. ESEC/SIGSOFT FSE: 61-72, 2015.

[2] N. Biggs. Algebraic Graph Theory. *Cambridge Mathematical Library*. Cambridge University Press, 1993.

[3] P. Flajolet and R. Sedgewick. *Analytic Combinatorics*. Cambridge University Press, New York, NY, USA, 1 edition, 2009.

[4] T. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308320, 1976.

[5] B. Nejmeh. NPATH: A measure of execution path complexity and its applications. *Commun. ACM*, 31(2):188200, 1988.

[6] R. Stanley. *Enumerative Combinatorics: Volume 1*. Cambridge University Press, New York, NY, USA, 2nd edition, 2011.