Automatically Computing Path Complexity of Programs*

Lucas Bang, Abdulbaki Aydin, and Tevfik Bultan
Department of Computer Science, University of California, Santa Barbara, CA 93106, USA
{bang,baki,bultan}@cs.ucsb.edu

ABSTRACT

Recent automated software testing techniques concentrate on achieving path coverage. We present a complexity measure 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. We define the *path complexity* of a program as a function that takes a depth bound as input and returns the number of paths in the control flow graph that are within that bound. We show how to automatically compute the path complexity function in closed form, and the *asymptotic path complexity* which identifies the dominant term in the path complexity function. Our results demonstrate that path complexity can be computed efficiently, and it is a better complexity measure for path coverage compared to cyclomatic complexity and NPATH complexity.

Categories and Subject Descriptors

D2.8 [**Metrics**]: Complexity Measures

General Terms

Algorithms, measurement.

Keywords

Path complexity, path coverage, automated testing.

*This material is based on research sponsored by NSF under grant CCF-1423623 and by DARPA under agreement number FA8750-15-2-0087. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government. Part of this research was conducted while Tevfik Bultan was visiting Koç University in İstanbul, Turkey, supported by a research fellowship from TÜBİTAK under the BİDEB 2221 program.

1. INTRODUCTION

Automated testing, i.e., automated generation of test cases, has become a very active area of research. Part of the surge in research activity in this area stems from the fact that, in the last two decades, automated verification research has mainly focused on bug finding techniques which resulted in merging of automated verification and automated testing research areas. Exhaustive exploration techniques such as model checking [15, 35], static analysis techniques such as extended static checking [13], and symbolic reasoning techniques such as symbolic execution [24] combined with advanced decision procedures [2, 8] have been effectively adopted to automated testing [16, 31, 5, 6, 7, 30].

This blurring of the lines between automated verification and testing resulted in new techniques that focus on *path coverage* rather than more traditional testing criteria such as *statement coverage*, *branch coverage*, and *MC/DC coverage*. Path coverage demands that all paths in the program are explored during testing. This is not always possible since some program paths may not be feasible. Moreover, in the presence of loops and unbounded recursion, the number of program paths may not be bounded. So, rather than targeting full path coverage, many modern automated software testing techniques try to cover paths up to a certain execution *depth* (where the execution depth can be increased iteratively with iterative deepening) [30].

In this paper, we show how to compute the *path complexity* of a program which provides an upper bound for the number of execution paths in a program with respect to the execution depth. As such, path complexity can be used to assess the difficulty of achieving path coverage with respect to increasing depth. 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 as we demonstrate in this paper.

Path complexity of a program with an acyclic control flow graph (i.e., no loops or recursion) is simply the number of paths in that control flow graph. In the presence of cyclic control flow graphs, path complexity of a program is a function that takes a single number n as input, which corresponds to the execution depth, and evaluation of the path complexity function gives the number of paths of the program within that depth. In this paper, we show how to compute the path complexity function using graph theoretic techniques. Note that path complexity function could be a complicated function with many terms and constants. We also define and show how to compute the *asymptotic path*

complexity which corresponds to the most significant term in the path complexity function. Using asymptotic path complexity, we can classify programs as programs with constant, polynomial, or exponential path complexity, and identify the degree of the polynomial for programs with polynomial path complexity and the base of the exponent for programs with exponential path complexity.

Based on the techniques we present in this paper, we implemented a tool for computing path complexity and asymptotic path complexity of Java methods. We first extract the control flow graph of each method from the Java bytecode and then compute the path complexity and the asymptotic path complexity on the control flow graph. Our current implementation is intra-procedural where method calls are represented as single edges in the control flow graph, and during path complexity analysis, the execution paths in the callee method's body are not investigated. Our approach can be extended to inter-procedural path complexity analysis by combining control flow graphs of individual methods with a procedure call graph. However, such an extension would generate an over-approximation since it would also count spurious paths where call and return sites do not match. A context sensitive extension to path complexity analysis that matches calls and returns would result in a more precise inter-procedural analysis and would be a valuable extension to the techniques we present in this paper.

Related Work.

There have been earlier efforts in assessing the difficulty of testing a program by analyzing the the control flow graph (CFG) of the program. For example, *cyclomatic complexity* determines the number of *linearly independent paths* in a given program by analyzing the CFG of the given program [27, 21]. One testing strategy, called *basis point testing*, is to exercise all such paths during testing. The upper bound for number of test cases required for basis point testing is given by the cyclomatic complexity. Moreover, it can be shown that cyclomatic complexity also provides an upper bound for the minimum number of test cases required for branch coverage and statement coverage. However, cyclomatic complexity does not consider the paths that are subsumed by the other paths (i.e., if all the nodes in a path is subsumed by the other paths, it is not considered).

Another complexity metric, called *NPATH complexity*, determines the number of paths in a program (which may be subsumed by each other) by restricting each loop execution to one or zero executions [29, 28]. NPATH complexity can increase exponentially with the length of the program and can result in very large numbers. 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.

In this paper, we are focusing on use of complexity metrics for assessing the difficulty of testing a program. There has been research on use of complexity metrics for assessing other properties of programs, such as maintenance costs [22, 14] and number of errors [23]. We have not investigated use of path complexity for such purposes but believe that these are interesting future research directions.

There has been recent work on automated worst-case complexity analysis [17, 18, 20, 19]. The main idea is to first instrument the loops in a program with auxiliary counters that keep track of how many times a loop is executed. Then, using invariant generation techniques, one infers invariants that relate the values of these counters to the inputs of the program. In effect, these discovered invariants provide the complexity bound for the program since they demonstrate how many times loops of a program are executed for a given input value. Note that this is significantly different than the path complexity we investigate in this paper. In worst-case complexity analysis the goal is to discover a relationship between the input size and the length of execution paths, whereas in path complexity analysis, the goal is to discover a relationship between the execution depth and the number of execution paths. Moreover, the path complexity analysis we present in this paper is a purely structural analysis which ignores branch conditions, whereas the worst-case complexity analysis discussed in [17, 18, 20, 19] requires sophisticated constraint analysis for invariant generation. Success of this approach depends on the quality of the invariant generation which is a difficult problem. For example, the abstract interpretation based approach described in [20] is semi-automated and requires user guidance during invariant discovery process. In contrast, the path complexity analysis we present in this paper is fully automated, and it is scalable.

The techniques we present in this paper for computing path complexity of a program are based on algebraic graph theory and analytic combinatorics techniques for counting the number of paths in graphs [3, 32, 12]. In particular, we use the transfer-matrix method based on generating functions [32, 12]. Recently, generating functions have also been used in model counting constraint solvers for determining the number of solutions to a given string constraint [26, 1].

There has been some earlier work on path complexity of programs [9]. However, the path complexity concept discussed in this earlier work defines the path complexity as a relation between the input size and the number of execution paths rather than a relation between the execution depth and the number of execution paths as we define in this paper. Furthermore, this earlier work does not present an automated way of computing the path complexity.

The remainder of the paper is organized as follows. In Section 2 we briefly discuss and contrast cyclomatic, NPATH, path and asymptotic path complexity. In Section 3 we present the techniques for automatically computing path complexity. In Section 4 we compare cyclomatic, NPATH, and asymptotic path complexity on several CFG patterns. We discuss our implementation and experiments in Section 5 and provide our conclusions in Section 6.

2. OVERVIEW

In this section we use three example methods (shown in Figure 1) from Java SDK to explain the following concepts and their differences: cyclomatic complexity introduced in [27], NPATH complexity introduced in [29], and the path complexity and asymptotic path complexity introduced in this paper. In order to present all these concepts in a uniform manner we use the control flow graph (CFG) representation, i.e., given a program, we first extract a CFG from the program and then we compute these complexity measures on the CFG. A CFG consists of a set of nodes that correspond to basic blocks of the program. Nodes of

```
private static void rangeCheck(int length, int fromIndex,
    int toIndex) {
  if (fromIndex > toIndex) {
    throw new IllegalArgumentException(
      "fromIndex(" + fromIndex + ") > toIndex(" + toIndex + ")");
  }
  if (fromIndex < 0) {
    throw new ArrayIndexOutOfBoundsException(fromIndex);
  }
  if (toIndex > length) {
    throw new ArrayIndexOutOfBoundsException(toIndex);
  }
}
```

(a) `java.util.Arrays.rangeCheck()`

```
public Matcher reset() {
  first = -1;
  last = 0;
  oldLast = -1;
  for(int i=0; i<groups.length; i++)
    groups[i] = -1;
  for(int i=0; i<locals.length; i++)
    locals[i] = -1;
  lastAppendPosition = 0;
  from = 0;
  to = getTextLength();
  return this;
}
```

(b) `java.util.regex.Matcher.reset()`

```
private static int binarySearch0(long[] a,
    int fromIndex, int toIndex, long key) {
  int low = fromIndex;
  int high = toIndex - 1;
  while (low <= high) {
    int mid = (low + high) >>> 1;
    long midVal = a[mid];
    if (midVal < key)
      low = mid + 1;
    else if (midVal > key)
      high = mid - 1;
    else
      return mid; // key found
  }
  return -(low + 1);  // key not found.
}
```

(c) `java.util.Arrays.binarySearch0()`

Figure 1: Three example methods from Java SDK.

the CFG are connected with edges that correspond to control flow of the program (i.e., if there is an edge from node 1 to node 2, then, in some program execution, basic block that corresponds to node 2 can be executed immediately after the basic block that corresponds to node 1). We assume that each CFG has a unique entry node denoting the program point where the execution starts, and a single exit node denoting the program point where the execution terminates (it is easy to convert CFGs that have multiple termination points to this form by adding an extra node). Figures 2(a), (b), and (c) show the CFGs for the methods shown in Figures 1(a), (b), and (c), respectively.

Cyclomatic Complexity.

Given a CFG, cyclomatic complexity computes the maximum number of linearly independent paths in the CFG from the entry node to the exit node [27]. A set of paths through a program are linearly independent if and only if each path in the set contains at least one edge that is not included in any other path in the set.

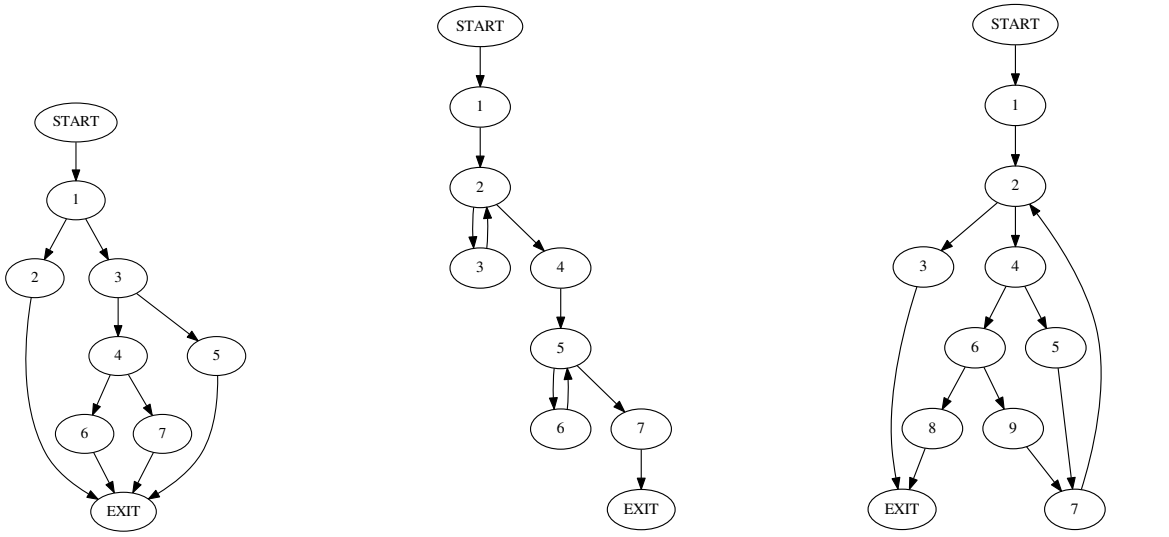
For example, the method shown in Figure 1(a), with the CFG shown in Figure 2(a), has at most 4 linearly independent paths: $\langle S, 1, 2, T\rangle$, $\langle S, 1, 3, 4, 6, T\rangle$, $\langle S, 1, 3, 4, 7, T\rangle$, $\langle S, 1, 3, 5, T\rangle$ where we identify each path by the sequence of nodes visited during the execution of the path, and we use S to denote the entry point where the execution starts and T to denote the exit point where the execution terminates. Since the maximum number of linearly independent paths for the method shown in Figure 1(a) is 4, the cyclomatic complexity of this method is 4.

On the other hand, the method shown in Figure 1(b), with the CFG shown in Figure 2(b), has at most 3 linearly independent paths: $\langle S, 1, 2, 4, 5, 7, T\rangle$, $\langle S, 1, 2, 3, 2, 4, 5, 7, T\rangle$, $\langle S, 1, 2, 3, 4, 5, 6, 5, 7, T\rangle$, hence its cyclomatic complexity is 3. Note that, we can replace the last path in the set above with the path $\langle S, 1, 2, 3, 2, 3, 4, 5, 6, 5, 7, T\rangle$ and the resulting set of three paths will still be linearly independent. However, we cannot have both $\langle S, 1, 2, 3, 4, 5, 6, 5, 7, T\rangle$ and $\langle S, 1, 2, 3, 2, 3, 4, 5, 6, 5, 7, T\rangle$ in the set of linearly independent paths since these two paths are not linearly independent.

Finally, the method shown in Figure 1(c), with CFG shown in Figure 2(c), has at most 4 linearly independent paths: $\langle S, 1, 2, 3, T\rangle$, $\langle S, 1, 2, 4, 6, 8, T\rangle$, $\langle S, 1, 2, 4, 6, 9, 7, 2, 3, T\rangle$, and $\langle S, 1, 2, 4, 5, 7, 2, 3, T\rangle$. We can find other sets of linearly independent paths for this method, but the cardinality of the set of linearly independent paths can never be more than 4. For this CFG, after 4 linearly independent paths, any other path we add will not include any new nodes and, therefore, will not be linearly independent.

Cyclomatic complexity can be computed easily from a CFG by first counting the number of edges (E), the number of nodes (N), and the number of connected components (P) in the CFG. Then, the cyclomatic complexity is given by the expression $E - N + 2P$. For example, for the CFG shown in Figure 2(a) we have: $E = 11$, $N = 9$, $P = 1$, and the cyclomatic complexity is $11 - 9 + 2 \times 1 = 4$. On the other hand, for the CFG shown in Figure 2(b) we have: $E = 10$, $N = 9$, $P = 1$, and the cyclomatic complexity is $10 - 9 + 2 \times 1 = 3$, and for the CFG shown in Figure 2(c) we have: $E = 13$, $N = 11$, $P = 1$, and the cyclomatic complexity is $13 - 11 + 2 \times 1 = 4$.

Cyclomatic complexity can be used for estimating the cost of testing since it provides an upper bound for the minimum number of test cases required for branch coverage. Hence, as the cyclomatic complexity of a program increases, the cost of achieving branch coverage for that program is likely to increase. However, cyclomatic complexity is not very useful for assessing difficulty of achieving path coverage. For example, the methods shown in Figure 1(b) and (c) contain loops, hence, achieving path coverage during testing for these methods will be difficult. However, cyclomatic complexity of the method shown in Figure 1(b) is lower than the cyclomatic complexity of the method shown in Figure 1(a), and the cyclomatic complexity of the method shown in Figure 1(a) is same as the cyclomatic complexity of the method shown in Figure 1(b). Hence, if we are targeting path coverage, cyclomatic complexity is not a useful measure. Even without a coverage criteria in mind, if we accept the fact that testing programs with loops is harder than testing programs without loops, cyclomatic complexity is not a useful metric for determining difficulty of testing since it does not distinguish between the back and forward edges in CFGs



(a) CFG for `java.util.Arrays.rangeCheck()`

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

(c) CFG for `java.util.Arrays.binarySearch0()`

Figure 2: Control Flow Graphs (CFGs) for the the three methods in Figure 1.

and, therefore, it is ambivalent to existence of loops in the CFG.

Another reason for the inadequacy of cyclomatic complexity as a metric for difficulty of achieving path coverage is the fact that cyclomatic complexity grows at most linearly with the number of edges in the CFG. However, the number of paths in a CFG can grow exponentially even without loops. For example, consider the CFG of a program that consists of K if-then-else statements executed back to back. The cyclomatic complexity of such a CFG structure would grow linearly with respect to K, whereas the number of paths in the CFG would grow exponentially, i.e., would be proportional to 2^K.

NPATH complexity.

NPATH complexity overcomes this short-coming of cyclomatic complexity by counting all acyclic paths in the CFG without requiring them to be linearly independent [29]. For example, the NPATH complexity of the method shown in Figure 1(a), with the CFG shown in Figure 2(a), is 4, which simply is the number of paths in the CFG. Since NPATH complexity focuses on counting the number of paths in the CFG, it is more suitable for assessing the difficulty of achieving path coverage than cyclomatic complexity. However, for CFGs with loops, NPATH complexity also makes a simplifying assumption which makes it unsuitable for assessing the difficulty of achieving path coverage. In the presence of loops, NPATH complexity only counts the paths that execute the loop body once or zero times. For example, there are two loops in the method shown in Figure 1(b) with the CFG shown in Figure 2(b). NPATH complexity counts the path that executes each loop zero times ($\langle S, 1, 2, 4, 5, 7, T\rangle$), it counts the two paths that execute one loop one times and the other loop zero times ($\langle S, 1, 2, 3, 2, 4, 5, 7, T\rangle$, $\langle S, 1, 2, 3, 2, 4, 5, 6, 5, 7, T\rangle$), and it counts the path that executes each loop one time ($\langle S, 1, 2, 3, 2, 4, 5, 6, 5, 7, T\rangle$), resulting in 4 paths. So the NPATH complexity of the method shown in Figure 1(c) is 4.

The method shown in Figure 1(c), with the CFG shown in Figure 2(c), has a single loop that contains conditional statements in the loop body (unlike the loops in the CFG shown in Figure 2(b) which do not contain conditional statements in the loop body). For the CFG shown in Figure 2(c), the NPATH complexity takes into account the path that executes the loop body zero times ($\langle S, 1, 2, 3, T\rangle$) the path that executes the loop body once and takes the first then-branch ($\langle S, 1, 2, 4, 5, 7, 2, 3, T\rangle$), the loop that executes the loop body once and takes the first else-branch and the second then-branch ($\langle S, 1, 2, 4, 6, 9, 7, 2, 3, T\rangle$), and, the loop that executes the loop body once and takes the both else-branches ($\langle S, 1, 2, 4, 6, 8, T\rangle$). So, the NPATH complexity of the method shown in Figure 1(c) is also 4.

NPATH complexity was originally defined through operations that are applied directly to structured programming code [29]. However, we observe that NPATH complexity can be restated in terms of the CFG in the following way: NPATH is the number of paths in the CFG that use any edge at most once. Thus, for any node u, the number of paths from u to the exit node is given by the recursive definition

$$NPATH(u, G) = \sum_{e_{uv} \in G} NPATH(v, G - e_{uv}) \qquad (1)$$

where e_{uv} is an edge from u to v and $G - e_{uv}$ is the graph obtained by deleting e_{uv} from G. Thus, we can compute $NPATH(START, G)$ using dynamic programming techniques and applying the initial condition $NPATH(EXIT, \cdot) = 1$.

Since NPATH complexity only considers execution of each loop body once or zero times, it is not able to distinguish

Table 1: Results of different complexity measures.

Method	Cyclomatic Complexity	NPATH Complexity	Path Complexity	Asymptotic Complexity
`java.util.Arrays.rangeCheck()` Figure 1 (a)	4	4	4	$\Theta(1)$
`java.util.regex.Matcher.reset()` Figure 1 (b)	3	4	$0.12 \times n^2 + 1.25 \times n + 3$	$\Theta(n^2)$
`java.util.Arrays.binarySearch0()` Figure 1 (c)	4	4	$6.86 \times 1.17^n + 0.22 \times 1.1^n + 0.13 \times 0.84^n + 2$	$\Theta(1.17^n)$

between loops that create a large number of paths from the loops that do not create a large number of paths. Moreover, since NPATH complexity produces a single constant number, it is not possible the distinguish CFGs that have a constant number of paths from the ones that have an unbounded number of paths (i.e., the ones with loops). For example, the NPATH complexities of the methods in Figure 1(a), (b) and (c) are all the same although two of them contain loops and one of them does not contain a loop.

Path Complexity.

Path complexity is equal to the NPATH complexity for the programs that do not contain loops or recursion, i.e., for the programs with a constant number of paths, path complexity just returns the number of paths. However, for the programs with loops or recursion, path complexity is not a constant number but a symbolic expression on a single variable n which denotes the execution depth. The expression computed by the path complexity is an upper bound for the number of paths in the program up to the depth n. I.e., substituting a constant value for n in the function returned by the path complexity and then evaluating the function results in a constant number that corresponds to an upper bound for the number of paths in the program within that execution depth.

For example, for the method shown in Figure 1(a), the path complexity is 4. Since this method does not contain any loops, its path complexity is equal to its NPATH complexity.

However, for the method shown in Figure 1(b), the path complexity is $0.12 \times n^2 + 1.25 \times n + 3$. Here, we observe that the number of paths in this method increases polynomially with respect to the depth of execution.

On the other hand, for the method shown in Figure 1(c), the path complexity is $6.86 \times 1.17^n + 0.22 \times 1.1^n + 0.13 \times 0.84^n + 2$. I.e., for this method, the number of paths increases exponentially with respect to the depth of execution.

Asymptotic Path Complexity.

The path complexity expressions could easily get extremely complicated. In order to produce a more readable and understandable complexity measure, we also define the asymptotic path complexity which simply extracts the dominant term from the path complexity expression. So, the asymptotic path complexity of the method shown in Figure 1(a) is $\Theta(1)$, denoting that its number of paths are constant, the asymptotic path complexity of the method shown in Figure 1(b) is $\Theta(n^2)$, denoting a quadratic growth in the number of paths with increasing execution depth, and the asymptotic path complexity of the method shown in Figure 1(c) is $\Theta(1.17^n)$, denoting an exponential growth in the number of paths with increasing execution depth.

The summary of different complexity measures for the methods shown in Figure 1 is shown in Table 1. We claim that path complexity and asymptotic path complexity reflect the cost of achieving path coverage in testing better than cyclomatic complexity and NPATH complexity.

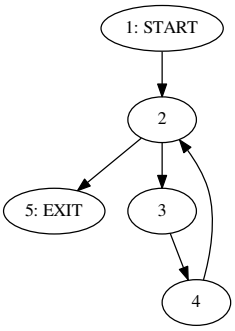
3. COMPUTING PATH COMPLEXITY

In order to compute the path complexity of a method within a program, we first extract its control flow graph (CFG). A CFG G is a tuple $\langle N, E, S, T \rangle$ where N is the set of nodes denoting basic blocks of the program, $E \subseteq N \times N$ is the set of edges denoting the control flow among basic blocks, $S \in N$ is a unique node denoting the basic block where the execution starts, and $T \in N$ is a unique node denoting the basic block where the execution terminates (programs with multiple termination points can be represented with CFGs in this format by creating an extra termination node). Below, we assume that S is always the node labeled with the smallest number (i.e., 1) and T is always the node labeled with the highest number.

Once we have determined the control flow graph, we make use of techniques from algebraic graph theory and analytic combinatorics to count the number of execution paths [3, 32, 12]. Given a CFG G, using standard graph theory notation, a path π is a sequence of edges in G, $e_1, e_2, \ldots e_n$, and the length of a path $L(\pi)$ is the number of edges in the path. Given a bound, n, on the execution depth, we produce a closed-form solution function, $path(n)$, that computes the number of paths, π, with $L(\pi) \leq n$. From $path(n)$, the asymptotic behavior can be determined and used to categorize the path complexity of the method from which the CFG was derived.

In order to aid the reader, we demonstrate the technique on a small example. Consider the CFG shown in Figure 3. Define $count(n)$ to be the number of paths of exactly length n and $path(n)$ to be the number of paths with length less than or equal to n. That is, $path(n) = \sum_{i=1}^{n} count(i)$. For our small example graph, one can verify that $count(n)$ is 1 if $n = 3k + 2$ for some non-negative integer k and 0 otherwise; there is one path of length 2, one path of length 5, and so on. The values of $count(n)$ and $path(n)$ are listed in the table shown in Figure 3 for values of n up to 9.

We define the *path count sequence* of a graph G to be an infinite sequence $a_0, a_1, \ldots, a_i, \ldots$ where each $a_i = path(i)$. In our running example, the path count sequence is $\{0, 0, 1, 1, 1, 2, 2, 2, 3, \ldots\}$. Further investigation would indicate that $path(n) = \lfloor \frac{n+1}{3} \rfloor$ where $\lfloor x \rfloor$ is the integer floor function. Asymptotically speaking, we can see that the number of paths grows linearly, approximately as $\frac{n}{3}$, with the depth bound n. We will demonstrate how such asymptotic analysis can be automated using the theory of integer sequence generating functions.



n	$count(n)$	$path(n)$
0	0	0
1	0	0
2	1	1
3	0	1
4	0	1
5	1	2
6	0	2
7	0	2
8	1	3
9	0	3
⋮	⋮	⋮

Figure 3: An example CFG and corresponding path counts.

Generating Function for an Integer Sequence: For an integer sequence $\{a_i\}$ that represents the bounded path counts of a CFG, we can encode values of $path(i)$ as the coefficients of a polynomial: an ordinary generating function (GF).

The *ordinary generating function* [32, 12] of the sequence a_0,a_1, …, a_i, … is the infinite polynomial

$$g(z) = \sum_{i \geq 0} a_i z^i. \tag{2}$$

Although $g(z)$ is an infinite polynomial, it can be interpreted as the Taylor series of a finite rational expression [10]. I.e., we can also write $g(z) = p(z)/q(z)$, where $p(z)$ and $q(z)$ are finite degree polynomials. If $g(z)$ is given as such a finite rational expression, each a_i can be computed from the Taylor series expansion of $g(z)$:

$$a_i = \frac{g^{(i)}(0)}{i!}, \tag{3}$$

where $g^{(i)}(z)$ is the i^{th} derivative of $g(z)$. We write $[z^i]g(z)$ for the i^{th} Taylor series coefficient of $g(z)$. This is what makes a generating function useful; it is a compact finite representation of an infinite series.

Returning to our running example, we can write the generating function for $path(n)$ both as a rational function and as an infinite Taylor series polynomial. The reader can verify the following equivalence by computing the coefficients via equation (3):

$$\begin{aligned} g(z) &= \frac{z^2}{1 - z + z^3 - z^4} \\ &= z^2 + z^3 + z^4 + 2z^5 + 2z^6 + 2z^7 + 3z^8 + \ldots \end{aligned} \tag{4}$$

Given the generating function for our CFG written as a finite rational expression, we can take the n^{th} Taylor series coefficient to determine the number of paths with length bounded by n. For example, to compute the number of paths with a length bound of 10, we can compute $path(10) = [z^{10}]g(z) = 3$.

In fact, we can use the generating function to derive a closed-form function of n that computes the desired coefficient. In the following, describe how to determine the generating function as a finite rational expression for a CFG, extract a closed-form function for counting paths, and how to perform asymptotic analysis of that function.

Generating Function for a CFG: Given a CGF G and length n, we can compute the generating function $g(z)$ such that the n^{th} Taylor series coefficient of $g(z)$ is equal to $path(n)$. From a CFG G, we construct the adjacency matrix (also called the "transfer-matrix" [32, 12]) T, where T_{ij} is 1 if there is an edge from v_i to v_j and 0 otherwise. In addition, $T_{|N|,|N|} = 1$, where $|N|$ is the number of nodes in G. Then the generating function for $path(n)$ is

$$g(z) = (-1)^{|N|+1} \frac{\det(I - zT : |N|, 1)}{\det(I - zT)}, \tag{5}$$

where $(M : i, j)$ denotes the matrix obtained by removing the i^{th} row and j^{th} column from M, I is the identity matrix, and $\det(M)$ is the matrix determinant.

The transition matrix T, and the terms $(I - zT)$ and $(I - zT : n, 1)$ for our running example are:

$$T = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}, I - zT = \begin{bmatrix} 1 & -z & 0 & 0 & 0 \\ 0 & 1 & -z & 0 & -z \\ 0 & 0 & 1 & -z & 0 \\ 0 & -z & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 - z \end{bmatrix},$$

$$(I - zT : 5, 1) = \begin{bmatrix} -z & 0 & 0 & 0 \\ 1 & -z & 0 & -z \\ 0 & 1 & -z & 0 \\ -z & 0 & 1 & 0 \end{bmatrix},$$

and applying equation (5) by taking the appropriate determinants results in the generating function

$$g(z) = \frac{z^2}{1 - z + z^3 - z^4}. \tag{6}$$

This is precisely the same GF that counts paths in our example CFG given in equation (4).

A Closed-form Solution: From a rational generating function $g(z) = p(z)/q(z)$ we can derive a closed-form function for $path(n)$ that is a sum of products of simple polynomial and exponential terms. Given a generating function, the form of $path(n)$ is determined by the roots of the denominator $q(z)$ [11, 12].

Suppose $q(z)$ is a polynomial of degree d. By the Fundamental Theorem of Algebra [25], $q(z)$ has exactly d roots over the complex numbers, accounting for multiplicity. Ignoring multiplicity, suppose there are D distinct roots. Let r_i be the i^{th} distinct root of $q(z)$ and m_i be the multiplicity of r_i. Then $path(n)$ is given by

$$path(n) = \sum_{i=1}^{D} \sum_{j=0}^{m_i - 1} c_{i,j} n^j \left(\frac{1}{r_i}\right)^n, \tag{7}$$

where c_i are coefficients that are determined by the first d 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 d equations and d unknowns. This system can be solved for the coefficients $c_{i,j}$ via elementary linear algebra.

In our example, $q(z) = 1 - z + z^3 - z^4$ and has four roots. The root 1 is repeated and so there are three distinct roots: $r_1 = r_2 = 1$ with multiplicity $m_1 = m_2 = 2$, $r_3 =$

$-\frac{1}{2} - \frac{i\sqrt{3}}{2}$ with multiplicity $m_3 = 1$, and $r_4 = -\frac{1}{2} - \frac{i\sqrt{3}}{2}$ with multiplicity $m_4 = 1$. In this particular case, it turns out that $\frac{1}{r_3} = \overline{r}_3$ and $\frac{1}{r_4} = \overline{r}_4$, where $\overline{r}$ is the complex conjugate of r. So our desired function becomes

$$\begin{aligned} path(n) &= c_{1,0} + c_{1,1}n \\ &+ c_{2,0}\left(-\frac{1}{2} - \frac{i\sqrt{3}}{2}\right)^n \\ &+ c_{3,0}\left(-\frac{1}{2} + \frac{i\sqrt{3}}{2}\right)^n \end{aligned} \tag{8}$$

The first four terms of the Taylor series expansion of $g(z)$ are $a_0 = a_1 = 0$ and $a_2 = a_3 = 1$. By evaluating $path(n)$ at $n = 0, 1, 2, 3$ we can solve for the coefficients $c_{i,j}$ to find that $c_{1,0} = \frac{4}{3}$, $c_{1,1} = \frac{1}{3}$, $c_{2,0} = \frac{1}{6}\left(\frac{i}{\sqrt{3}} - 1\right)$, $c_{3,0} = \frac{1}{6}\left(\frac{i}{\sqrt{3}} + 1\right)$. Thus, we have

$$\begin{aligned} path(n) &= \frac{4}{3} + \frac{1}{3}n \\ &+ \frac{1}{6}\left(\frac{i}{\sqrt{3}} - 1\right)\left(-\frac{1}{2} - \frac{i\sqrt{3}}{2}\right)^n \\ &- \frac{1}{6}\left(\frac{i}{\sqrt{3}} + 1\right)\left(-\frac{1}{2} + \frac{i\sqrt{3}}{2}\right)^n \end{aligned} \tag{9}$$

While it appears that this function may take on fractional and complex values, for every value of n, any imaginary terms conveniently cancel out and fractions combine to result in integers. In fact, one may verify that the resulting function $path(n)$ is equivalent to $\lfloor \frac{n+1}{3} \rfloor$. Furthermore, although $path(n)$ has a seemingly complicated form, it can be derived automatically from the transfer-matrix method and the resulting generating function.

Upper Bounds: In order to perform asymptotic analysis of $path(n)$ we first bound the complex terms. Because the complex roots of $q(z)$ always appear in conjugate pairs, we can bound and combine them in pairs. For any complex number w, $w^n + \overline{w}^n$ is a real number. Furthermore $-2|w|^n \leq w^n + \overline{w}^n \leq 2|w|^n$, where $|w|$ denotes the complex norm of w. Thus, we can replace each exponentiated complex root in $path(n)$ with its norm in order to get a sound upper bound. We denote the upper bound on $path(n)$ by $upper(n)$.

In our example, $|r_3| = |r_4| = 1$. Performing the substitution and simplifying, we get

$$\begin{aligned} path(n) &\leq \frac{4}{3} + \frac{1}{3}n + \frac{1}{6}\left(\frac{i}{\sqrt{3}} - 1\right)\cdot 1 - \frac{1}{6}\left(\frac{i}{\sqrt{3}} + 1\right)\cdot 1 \\ &= \frac{n+1}{3} = upper(n). \end{aligned} \tag{10}$$

Notice that this is the same result given in our initial informal analysis of the example CFG, without the floor function. We plot the exact solution given in equation (9) and the upper bound given in equation (10) together in Figure 4.

Asymptotic Analysis: We extract the asymptotic complexity from the upper bound on $path(n)$ using standard asymptotic analysis, where $f(n) = \Theta(g(n))$ if and only if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$ [34]. The asymptotic path complexity of our running example can be described by $upper(n) = \Theta(n)$.

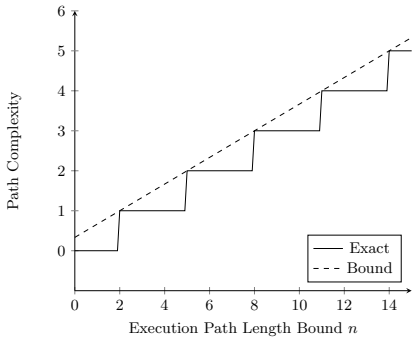


Figure 4: Comparison of exact solution and upper bound.

We have been able to determine that the path complexity of our example CFG grows linearly with the execution depth. This same method outlined here works on more complex CFGs as well, allowing 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.

Numeric Approximation: In some instances, it is necessary to perform numeric approximation for the roots of $q(z)$. This is due to the fact that there is no explicit formula for solving polynomials with degree larger than 4. By the Abel–Ruffini Theorem [33] a general solution does not exist for exactly expressing roots in terms of the elementary algebraic operations of addition, subtraction, multiplication, division, exponentiation, taking roots, and so on. Note that this does not mean that all polynomials of degree 5 or larger cannot be solved exactly, but that there is no all-purpose method for producing exact solutions.

On the other hand, the roots of any polynomial can be approximated to within any desired finite precision using standard numeric root-finding techniques [4].

Thus, when the degree of $q(z)$ is greater than 4 we numerically approximate each root r_i by $(r_i \pm \epsilon)$. In our experiments ϵ is such that we maintain at least 15 digits of precision.

Algorithm Overview: Now that we have described the mathematical theory behind path complexity, we give a high level description of the path complexity analysis in Algorithm 1.

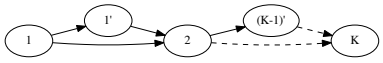
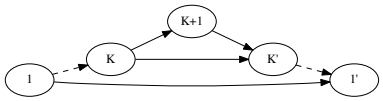
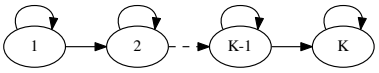
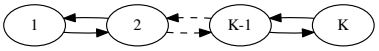
Algorithm 1 COMPUTING PATH COMPLEXITY OF CFG G

1: Determine adjacency matrix T of the CFG G.
2: Compute generating function via Eq. 5 as follows:

$$g(z) = \frac{p(z)}{q(z)} = \frac{(-1)^{n+1}\det(I - zT{:}n,1)}{\det(I - zT)}$$

3: Compute roots of $q(z)$ to determine $path(n)$ via Eq. 7.
4: Solve for each $c_{i,j}$ in $path(n)$ using the first $|N|$ Taylor series coefficients of $g(z)$, via Eq. 3, which gives $path(n)$.
5: Determine upper bound on $path(n)$ by replacing each complex root w with $|w|$ and simplifying.
6: Determine asymptotic behavior of upper bound on $path(n)$.

Table 2: Complexity measure comparison.

Pattern	Control Flow Graph	Cyclomatic Complexity	NPATH Complexity	Asymptotic Path Complexity
K If-Else in sequence		$K+1$	2^K	2^K
K If-Else nested		$K+1$	$K+1$	$K+1$
K Loop in sequence		$K+1$	2^K	$\Theta(n^K)$
K Loop nested		$K+1$	$K+1$	$\Theta(b^n)$

4. COMPARISON WITH OTHER COMPLEXITY MEASURES

In Table 2 we give the cyclomatic, NPATH, and asymptotic path complexity expressions for four common programming patterns. Given an integer constant K, we examine the following: a sequence of K conditional statements, fully nested conditional statements to a depth of K, a sequence of K loops, and fully nested loops to a depth of K.

For these four patterns, cyclomatic complexity gives identical results, and it simply grows linearly with the number of conditional or loop structures. Consequently, cyclomatic complexity alone cannot distinguish the differences in these four patterns and is not a good measure of assessing the difficulty of achieving path coverage in testing these programs.

NPATH complexity does a slightly better job, but still only reports a constant number that depends directly on the size of the graph, not on the execution depth bound. NPATH complexity cannot distinguish between K conditional statements in sequence and K looping statements in sequence, both of which yield a constant value of 2^K for a given K. Likewise, NPATH cannot distinguish between K nested conditional statements and K nested looping statements, both of which yield a constant value of $K+1$ for a given K. Because NPATH complexity counts paths in which edges cannot be crossed more than once, it does not account for all possible combinations of loop repetitions, and, therefore, is also not a good measure of assessing difficulty of achieving path coverage.

Among the three metrics, asymptotic path complexity is the most useful in distinguishing between the complexity of the four programming patterns. It produces four different types of results. For K conditional statements executed in sequence it tells us that there are a constant number of possible executions and that the constant number is exponential in the number of conditions. For nested conditions it tells us that the number of paths is constant and that the number of paths is linear in the number of conditions. For K loops in sequence asymptotic path complexity tells us that, as the execution depth bound n increases, the number of paths grows polynomially as $\Theta(n^K)$. Finally, for K nested loop structures, we find that as the depth bound n increases, the number of possible executions grows exponentially, as $\Theta(b^n)$ where b is a constant number that depends on K.

5. IMPLEMENTATION & EXPERIMENTS

In order to evaluate the path complexity algorithm, we performed an experimental comparison with the two complexity measurement method described in Section 2: Cyclomatic complexity and NPATH complexity.

We implemented Algorithm 1 in a tool called PAth Complexity analyzer (PAC). PAC automatically computes the path complexity and asymptotic path complexity of Java methods. PAC accepts a Java class file or a jar of class files as input and reports path complexity and asymptotic path complexity for all methods of input Java class(es). PAC has two main steps: (1) Control-flow graph (CFG) extraction, (2) Path complexity calculation. PAC is available for download; see the Appendix for details.

We used the ASM[1] Java bytecode manipulation library to generate CFGs using an intra-procedural analysis for each method. PAC first computes the basic blocks of a method. A basic block is a sequence of code statements where there is only one entry point and one exit point. In other words, none of the instructions are the target of a jump instruction except for the first and none of the instructions are a jump/halt instruction except for the last. PAC constructs CFGs using basic blocks which results in compact CFGs.

We implemented Algorithm 1 based on the techniques described in Section 3 using MATHEMATICA[2]. Implementing the algorithm in the MATHEMATICA language allowed us to leverage the built-in routines for symbolic manipulation of matrices, polynomials, and complex expressions, as well as numeric root finding methods. We also implemented the computation of cyclomatic complexity and a version of NPATH complexity based on the CFG as described in Section 2.

To do a comparative evaluation we experimented with the Java 7 SDK (Java benchmark) and the 34 Apache Commons[3] libraries (Apache benchmark). We extracted 132,768 CFGs from the Java benchmark and 44,426 CFGs from the Apache benchmark. We ran all the experiments on an Intel I5 machine with 2.5GHz X 4 processors and 32 GB of memory running Ubuntu 14.04.

[1] `http://asm.ow2.org/`
[2] `http://www.wolfram.com/mathematica/`
[3] `http://commons.apache.org/`

Table 3: Comparison of asymptotic complexity classification with cyclomatic and NPATH complexity values.

Path Comp. Classifications	Asym. Comp. # methods	Cyclomatic Complexity # methods				NPATH Complexity # methods			
		$1-5$	$6-10$	$11-100$	>100	$1-5$	$6-10$	$11-100$	>100
$C>1$	51959	43347 (83.4%)	6082 (11.7%)	2516 (4.8%)	14 (0.0%)	41001 (78.9%)	5541 (10.7%)	4441 (8.5%)	976 (1.9%)
$\Theta(n)$	9120	7321 (80.3%)	1296 (14.2%)	503 (5.5%)	0 (0.0%)	6721 (73.7%)	1298 (14.2%)	917 (10.1%)	184 (2.0%)
$\Theta(n^2)$	693	308 (44.4%)	245 (35.4%)	140 (20.2%)	0 (0.0%)	181 (26.1%)	183 (26.4%)	252 (36.4%)	77 (11.1%)
$\Theta(n^3)$	137	56 (40.9%)	44 (32.1%)	37 (27.0%)	0 (0.0%)	0 (0.0%)	39 (28.5%)	69 (50.4%)	29 (21.2%)
$\Theta(n^e), e \geq 4$	55	0 (0.0%)	23 (41.8%)	32 (58.2%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	13 (23.6%)	42 (76.4%)
$\Theta(b^n), 1 < b < 2$	7776	2491 (32.0%)	2759 (35.5%)	2514 (32.3%)	12 (0.2%)	1954 (25.1%)	1700 (21.9%)	2674 (34.4%)	1448 (18.6%)
$\Theta(b^n), 2 \leq b < 3$	735	392 (53.3%)	237 (32.2%)	102 (13.9%)	4 (0.5%)	296 (40.3%)	171 (23.3%)	210 (28.6%)	58 (7.9%)
$\Theta(b^n), 3 \leq b < 4$	49	24 (49.0%)	11 (22.4%)	8 (16.3%)	6 (12.2%)	21 (42.9%)	8 (16.3%)	10 (20.4%)	10 (20.4%)
$\Theta(b^n), b \geq 4$	30	0 (0.0%)	16 (53.3%)	13 (43.3%)	1 (3.3%)	0 (0.0%)	13 (43.3%)	7 (23.3%)	10 (33.3%)

Experimental Results.

We ran PAC for the Java SDK 7 and the Apache commons libraries. The total execution time for the Java benchmark is 2 hours 34 minutes 49 seconds with an average of 0.07 seconds per class file. The total execution time for the Apache benchmark is 53 minutes and 10 seconds with an average of 0.07 seconds per class file.

We compared PAC's asymptotic path complexity with cyclomatic complexity and NPATH complexity. We defined four complexity classes: 1) constant with value 1 ($C = 1$), 2) constant with a value that is greater than 1 ($C > 1$), 3) polynomial(n^k, with $k \geq 1$), and 4) exponential (b^n, with $n > 1$). Figure 5 shows the percentage of number of methods for each complexity class based on the PAC analysis results. The Java and Apache benchmarks have similar distributions. More than half of the methods ($\sim 60\%$) have only 1 execution path for both benchmarks. I.e., there are no jump instructions in those methods except for the method exit. For the Java benchmark, 30.1% of the methods have a constant complexity value greater than one, and 9.3% of them have polynomial or exponential complexity. Similarly, 27% of the Apache benchmark methods have constant complexity greater than one and 12.12% of them have polynomial or exponential complexity. We will omit the discussion of the methods that have a single execution path as all complexity measurements report the constant value 1. For the rest of the discussion, we further define fine-grained complexity classes based on the degrees of the polynomials and bases of the exponentials (first column in Table 3) and give the number of methods at each fine-grained class that are classified by PAC (second column).

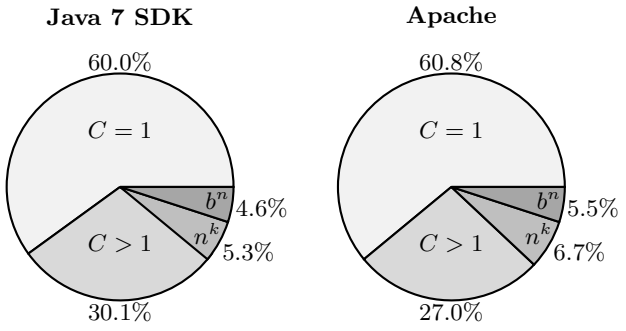


Figure 5: Distribution of the methods to four asymptotic complexity classes: 1) single path ($C = 1$), 2) constant number of paths greater than one ($C > 1$), 3) polynomial (n^k, $k \geq 1$), and 4) exponential (b^n, $n > 1$).

Table 3 shows the distributions of the cyclomatic and NPATH complexity results over the fine-grained complexity classes. We defined 4 bins for the values of cyclomatic and NPATH complexity as shown in the table header. The first row represents the methods that have finitely many execution paths (CFG contains only forward-edges). For this case, reporting the actual number of execution paths is more meaningful for path coverage rather than simply reporting that it is a constant number. For this reason, PAC also reports the actual number of execution paths in this complexity class. PAC and NPATH complexity give the same result, and cyclomatic complexity gives the same or a smaller number. NPATH complexity percentages for the bins $[11-100]$, $[>100]$ are larger than cyclomatic complexity percentages for the same bins. This is due to the fact that there can be exponential growth in the number of execution paths with the increase of forward-edges in a CFG, whereas cyclomatic complexity only grows linearly with the number of forward-edges. NPATH complexity is able to catch such exponential cases as shown in the first row of Table 2. By looking at only that row, we can say that cyclomatic complexity is not a good metric for assessing the difficulty of achieving path coverage.

The percentages of the corresponding bins in the ($C > 1$) row and $\Theta(n)$ row are close to each other both in cyclomatic and NPATH complexity. Hence, we can say that neither cyclomatic complexity nor NPATH complexity is able to differentiate between the methods with a constant number of execution paths and the methods with unbounded number of execution paths. As the degree of the polynomial increases, the percentages in the bin $[1-5]$ decreases. Similarly, as the base of the exponential increases, the percentages in the bin $[1-5]$ decreases. This is because of the fact that, as the complexity grows, we would expect more edges in the CFG. Cyclomatic complexity always grows linearly with the number of edges and NPATH complexity grows linearly or exponentially with the number of edges. We can see that the NPATH complexity shows a stronger increase in the percentages of higher range bins as the degree of polynomials increases compared to the increase in the base of the exponentials. This is due to that fact that, NPATH complexity grows exponentially when the complexity is polynomial and linearly when the complexity is exponential. Hence, NPATH complexity is not a good measure for assessing the difficulty of achieving path coverage.

One other important point is that, cyclomatic complexity and NPATH complexity only give a constant number as a measurement of program complexity. The number itself does not give sufficient information on difficulty of path coverage. There is always an overlap between different com-

plexity classes in terms of number of methods in the same range. For example, cyclomatic complexity reports values in the bin $[6-10]$ for 9 different complexity classes and NPATH reports values in the bin $[6-10]$ for 8 different complexity classes. As a concrete example, PAC reports $\Theta(n^3)$ for the method:

```
com.sun.org.apache.xerces.internal.impl.xs.XML
SchemaValidator$ValueStoreCache.initValueStoresFor
```

and $\Theta(3^n)$ for the method:

```
sun.java2d.loops.GraphicsPrimitive.satisfies
```

The corresponding cyclomatic complexity and NPATH complexity reported for both methods are 8 which does not carry enough information on actual difficulty of achieving path coverage for these methods.

We conducted the same comparison for all values of cyclomatic and NPATH complexity in $C > 1$, $\Theta(n)$, $\Theta(n^2)$, $\Theta(b^n), 1 < b < 2$, and $\Theta(b^n), 2 \leq b < 3$ complexity classes. Figure 6 shows the frequencies of NPATH complexity and cyclomatic complexity up to complexity measures of 100 in order to visualize the distribution. Frequencies in Figure 6a and Figure 6c for NPATH complexity follows similar trends for the complexity classes $C > 1$ and $\Theta(n)$. We can see a similar trend between any two frequency graphs of NPATH complexity. The frequencies of cyclomatic complexity values are also following the same trend as shown in Figures 6b, 6d, 6f, 6h, 6j. Figure 6 validates that cyclomatic complexity and NPATH complexity are not successful in differentiating between methods in different complexity classes, and hence, they are not good for assessing the difficulty of achieving path coverage.

Overall, our experimental results validate that path complexity is a better measure for assessing difficulty of achieving path coverage compared to to cyclomatic complexity and NPATH complexity. Our results also show that path complexity can be computed efficiently.

6. CONCLUSIONS

We present path complexity and asymptotic path complexity measures that provide an upper bound for the number of execution paths in a program. In order to compute the path complexity, we first extract the control flow graph, and then use graph theoretic techniques to generate a path complexity function that gives an upper bound for the number of execution paths within a given execution depth. Our path complexity analyzer can be used for assessing the difficulty of achieving path coverage with respect to increasing execution depth in the context of automated testing. Our experiments on popular Java libraries demonstrate the effectiveness and efficiency of the proposed approach. In the future, we plan to extend our path complexity analyzer with a context-sensitive inter-procedural analysis.

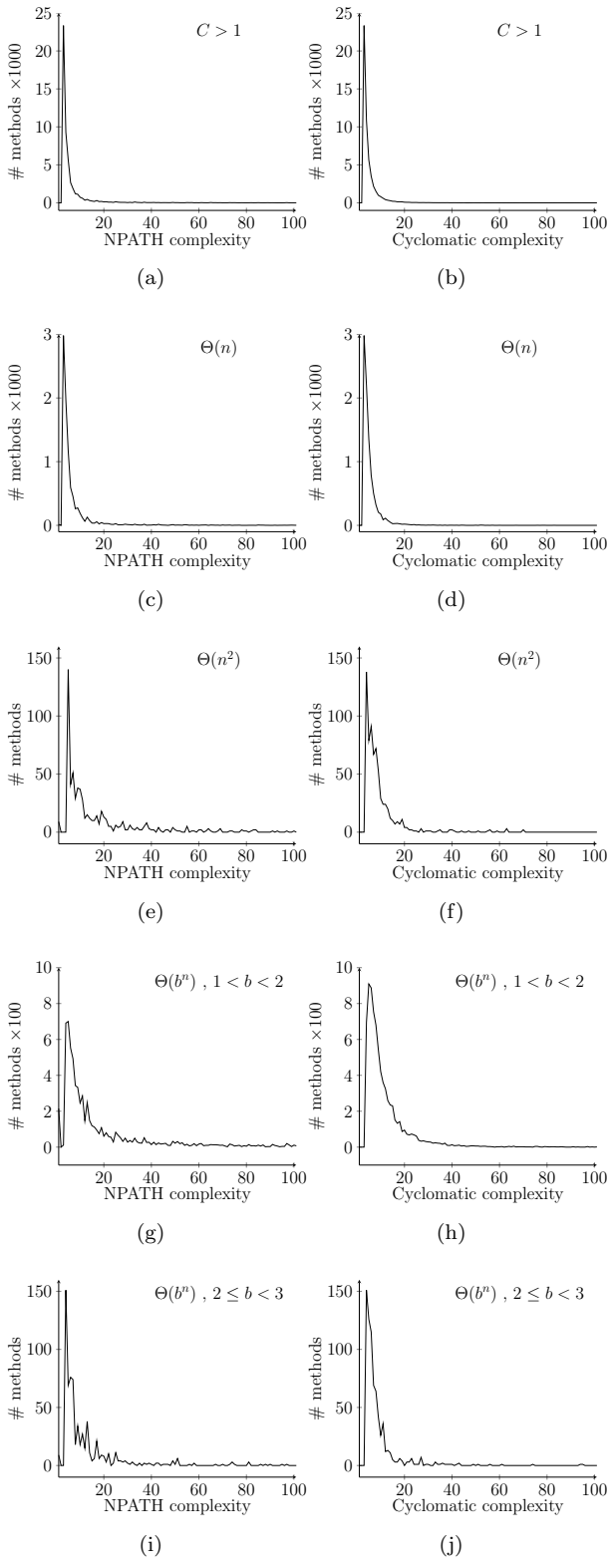


Figure 6: Distribution of NPATH and cyclomatic complexity values for the methods in different asymptotic complexity classes.

7. REPLICATION PACKAGE

The source code for PAC as well as a web interface are available from the University of California Santa Barbra Verification Lab (UCSB VLab) website:

`http://www.cs.ucsb.edu/~vlab/PAC/`.

Source Code and Data. Our tool, PAC, and experimental results have been successfully evaluated by the Replication Packages Evaluation Committee and found to meet expectations. The replication package for PAC can be downloaded as a compressed archive. Detailed instructions and scripts are provided for running our tools. Please note that our path complexity routines require a current installation of MATHEMATICA. The package consists of five major components.

1. **CFG Extractor.** This is an intra-procedural control flow graph extraction tool based on the ASM Java bytecode manipulation library. Our command-line-based application takes a directory of Java class files (or a single Java class file) as input. It then computes an intra-procedural control flow graph for every method. The output is a directory of CFGs exported to Graphviz `dot` format.

2. **Complexity Analyzer.** Cyclomatic, NPATH, and path complexities are computed with a set of MATHEMATICA routines. A command-line-based script is provided that takes a directory of control flow graphs (or a single control flow graph) as input. The input CFG's must be in Graphviz dot format. Each CFG is imported into a MATHEMATICA session and analyzed. The results are output into a user-defined file of comma separated values–see the following description of experimental data.

3. **Java Libraries.** The Java 7 SDK and Apache Commons libraries are provided as benchmark sets. The CFG Extractor described above can be run on these libraries in order to generate the control flow graphs as input to the Complexity Analyzer.

4. **Control Flow Graphs.** All of the control flow graphs generated from the Java libraries are provided.

5. **Experimental Data.** All experimental results for Java 7 SDK and Apache libraries, as summarized in Section 5, are also included. The results include the cyclomatic complexity, NPATH complexity, path complexity, asymptotic complexity, and complexity class (constant, polynomial, or exponential) for every method in the benchmark libraries.

Web Interface. A web interface for PAC is also available from the UCSB Verification Lab web page. The user may upload a Java class file, multiple class files, a folder of class files, or a `.jar` file. For every method in the submitted files, a table is produced that describes the method's cyclomatic complexity, NPATH complexity, path complexity, asymptotic path complexity, and complexity class. The complexity class is reported as either constant, polynomial, or exponential.

8. REFERENCES

[1] Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. Automata-based model counting for string constraints. In *Proc. 27th Int. Conf. Computer Aided Verification (CAV)*, 2015.

[2] Clark W. Barrett, David L. Dill, and Aaron Stump. A framework for cooperating decision procedures. In *Proc. 17th Int. Conf. Automated Deduction (CADE)*, pages 79–98, 2000.

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

[4] R. Burden and J. Faires. *Numerical Analysis*. Cengage Learning, 2010.

[5] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. 8th Symp. Operating Systems Design and Implementation (OSDI)*, pages 209–224, 2008.

[6] Christoph Csallner and Yannis Smaragdakis. Check 'n' crash: combining static checking and testing. In *Proc. 27th Int. Conf. Software Engineering (ICSE)*, pages 422–431, 2005.

[7] Christoph Csallner, Yannis Smaragdakis, and Tao Xie. Dsd-crasher: A hybrid analysis tool for bug finding. *ACM Trans. Softw. Eng. Methodol.*, 17(2), 2008.

[8] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proc. 14th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[9] Nishant Doshi. Approximation for the path complexity of binary search tree. *CoRR*, abs/1404.4692, 2014.

[10] A. Erdelyi. *Asymptotic Expansions*. Dover, 1956.

[11] Philippe Flajolet and Andrew M. Odlyzko. Singularity analysis of generating functions. *SIAM J. Discrete Math.*, 3(2):216–240, 1990.

[12] Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, New York, NY, USA, 1 edition, 2009.

[13] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proc. 2002 Conf. Programming Language Design and Implementation (PLDI)*, pages 234–245, 2002.

[14] Geoffrey K. Gill and Chris F. Kemerer. Cyclomatic complexity density and software maintenance productivity. *IEEE Trans. Software Eng.*, 17(12):1284–1288, 1991.

[15] Patrice Godefroid. Model checking for programming languages using verisoft. In *Proc. 24th Symp. Principles of Programming Languages (POPL)*, pages 174–186, 1997.

[16] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223, 2005.

[17] Bhargav S. Gulavani and Sumit Gulwani. A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In *Proc. 20th Int. Conf. Computer Aided Verification (CAV)*, pages 370–384, 2008.

[18] Sumit Gulwani, Sagar Jain, and Eric Koskinen. Control-flow refinement and progress invariants for bound analysis. *SIGPLAN Not.*, 44(6):375–385, June 2009.

[19] Sumit Gulwani, Bill McCloskey, and Ashish Tiwari. Lifting abstract interpreters to quantified logical domains. In *Proc. 35th Symp. Principles of Programming Languages (POPL)*, pages 235–246, 2008.

[20] Sumit Gulwani and Florian Zuleger. The reachability-bound problem. *SIGPLAN Not.*, 45(6):292–304, June 2010.

[21] Brian Henderson-Sellers, Yagna Raj Pant, and June M. Vemer. Cyclomatic complexity: theme and variations. *Australasian J. of Inf. Systems*, 1(1), 1993.

[22] Dennis G. Kafura and Geereddy R. Reddy. The use of software complexity metrics in software maintenance. *IEEE Trans. Software Eng.*, 13(3):335–343, 1987.

[23] Taghi M. Khoshgoftaar and John C. Munson. Predicting software development errors using software complexity metrics. *IEEE Journal on Selected Areas in Communications*, 8(2):253–261, 1990.

[24] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[25] S.G. Krantz. *The Handbook of Complex Variables.* Birkhäuser, 1999.

[26] Loi Luu, Shweta Shinde, Prateek Saxena, and Brian Demsky. A model counter for constraints over unbounded strings. In *Proc. 2014 Conf. Programming Language Design and Implementation (PLDI)*, page 57, 2014.

[27] Thomas J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308–320, 1976.

[28] Beth McColl and James C. McKim. Evaluating and extending npath as a software complexity measure. *Journal of Systems and Software*, 17(3):275–279, 1992.

[29] Brian A. Nejmeh. NPATH: A measure of execution path complexity and its applications. *Commun. ACM*, 31(2):188–200, 1988.

[30] Corina S. Pasareanu, Willem Visser, David H. Bushnell, Jaco Geldenhuys, Peter C. Mehlitz, and Neha Rungta. Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Autom. Softw. Eng.*, 20(3):391–425, 2013.

[31] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *Proc. 10th European Software Engineering Conf. and 13th Int. Symp. Foundations of Software Engineering (ESEC/FSE)*, pages 263–272, 2005.

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

[33] I. Stewart. *Galois Theory, Third Edition.* Chapman Hall/CRC Mathematics Series. Taylor & Francis, 2003.

[34] J. Stewart. *Calculus.* Cengage Learning, 2007.

[35] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.