CS267 - HW#4

Burak Kadron

March 14, 2017

1 Introduction

Software verification and testing is focused on exploring all states of the program and covering all paths. Because of state space explosion, covering all paths usually take exponential time in the size of program. If we knew number of paths in a program, we would have used it as an estimation of how complex the program is. It would also be a measure about the time it would take to cover all paths.

Previous work on path counting focuses on intra-procedural analysis and doesn't analyze the effect of function calls, just the operations within the program. If functions are not complicated, this analysis is true but if called functions introduce an overhead, analysing the overhead will improve our analysis. On the contrary, inter-procedural analysis takes function calls into account for a better estimation of number of paths of a program. This measure also gives us an upper bound on asymptotic complexity of the program.

An example on difference between intra and inter-procedural analysis would be counting the number of paths on a recursive function that traverses a graph with preordering. The function visits the root and calls the function on left and right children of the root respectively. An intra-procedural analysis would conclude the program has a constant number of paths with any given depth. An inter-procedural analysis would take the recursive calls into account to count the number of paths for each root where root has children and would return a function of n.

In this work, we improve upon previous work of intra-procedural analysis by applying inter-procedural analysis to path counting problem.

2 Related Work

There is not much work on path counting and complexity but we will reference and go over the related work. The most important work we use is Chomsky

Schutzenberger enumeration theorem. It states that for a given grammar, we can represent the number of words in that grammar by a power series function. We can also convert the grammar to that power series representation by using some steps but we will mention that in our next section.

Bang et al's work is very important as well. In that work, the graph of the program is converted to a matrix and that matrix is processed to obtain the number of paths for a given program. The operations on the graph are not distinguished, therefore it is an intra-procedural analysis but it is an important contribution and a good baseline for improvement.

3 Interprocedural Path Complexity Analysis

In this section, we will go over our methodology of our analysis. To count the number of paths efficiently, we convert our program to a pushdown automata and use common techniques to count paths on pushdown automata.

Firstly, we use Soot, a Java bytecode analysis tool to convert the bytecode to a static single assignment intermediate representation called Jimple. We use that representation to create a control flow graph where branch conditions are represented as two paths branching from the condition node. Loops are represented as branching at the branch condition part of the loop with one edge continuing looping, the other exiting the loop. Secondly, we convert our graph to a context-free grammar which is another representation for pushdown automata. For transformation, we use several rules to convert the graph to the grammar. Our grammar is unary and contains a single character (k) in the alphabet. We convert the statements which are assignments or return statements and don't call another function to character k. Some basic functions like arithmetic functions (addition, multiplication, etc.) are just calling a bytecode execution so we assume those statements do not call any other function. We convert statements which invoke functions we are analysing to their corresponding rule name. Additionally, we convert the branching to or sign("|") in the rules.

After we obtain the grammar, we convert each grammar rule to an equation and the solution of the set of equations will be number of paths for n where n is length of the path according to Chomsky-Schutzenberger enumeration theorem. To convert from a rule to a function, we replace concatenation with multiplication, or with addition, characters with a variable (x) and rules to function names.

For example:

$A- > B \mid C$ becomes $A = B + C$

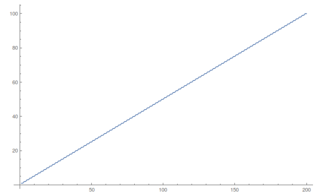
$B- > kBkB|k$ becomes $B = x^2B^2 + x$

$C- > BA$ becomes $C = B * A$

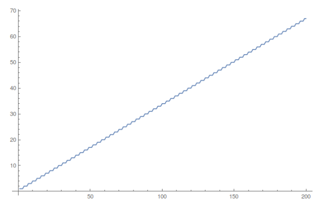
After conversion, we solve the systems of equations related the function we are analysing and the functions that function is calling using Mathematica. Then we convert our solution to a generating function by expanding the solution using Taylor series expansion. If our function is $g(x)$, the expansion converts it to $g(x) = g(0) + \frac{x * g'(0)}{1!} + \frac{x^2 * g''(0)}{2!} + \cdots + \frac{x^n * g^{(n)}(0)}{n!} + \ldots$ In this expansion, the coefficient corresponding to x^n is the number of paths of length n. By using this method, we can count the number of paths for any function. This count also gives an upper bound on the complexity of the program.

4 Experiments

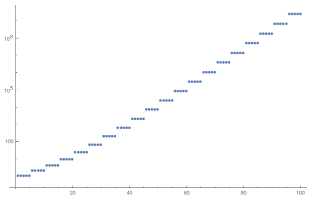
In this section we will show our analysis results for three examples, a recursive implementation of generating n'th Fibonacci number, a recursive implementation of generating n'th factorial of a number and a recursive implementation of determining whether a number is even.



(a) Plot of even/odd function



(b) Plot of factorial function



(c) Log plot of fibonacci function

Figure 1: Plots of analysed functions where y axis denotes length of paths and x axis denotes number of paths with y or less length.

As we can see from Figure 2, even/odd and factorial functions have linearly increasing number of paths but Fibonacci function has exponentially increasing number of paths as we can see log of number of paths is increasing in a linear fashion. You can also see the zoomed version of Figure 2 in Figure 3.

5 Future Work

Even though we can achieve what we aimed for at the start of the class, we believe there is room for improvement. One problem that can be improved is overapproximation on number of paths. For any function that calls itself twice, our path complexity will be exponential. But there are programs where the complexity is much lower because of efficient algorithms. One example is merge sort. For a list of n elements, it will take $O(n \log n)$ time to sort the list. The number of paths are greater than $n \log n$ but they are not exponential in this case because program terminates much quicker. We believe using symbolic execution to prune unfeasible paths will improve the approximation on number of paths. We also believe our claim on correlation between number of paths and execution time on verification needs to be tested.

6 Conclusion

In conclusion, we have presented a method on inter-procedural path counting using automata theory and program analysis. We believe even without improvements, it gives us a good measure on number of paths for a given program while taking the function calls into account and with improvements, the results can be improved greatly.

7 Appendix

This section contains some code examples and graphs related to the paper.

Implementation for even/odd function:

```
public static boolean even(int n){
    if (n == 0) {
        return true;
    } else {
        return odd(n-1);
    }
}
public static boolean odd(int n){
    if (n==0){
        return false;
    } else
        return even(n-1);
}
```

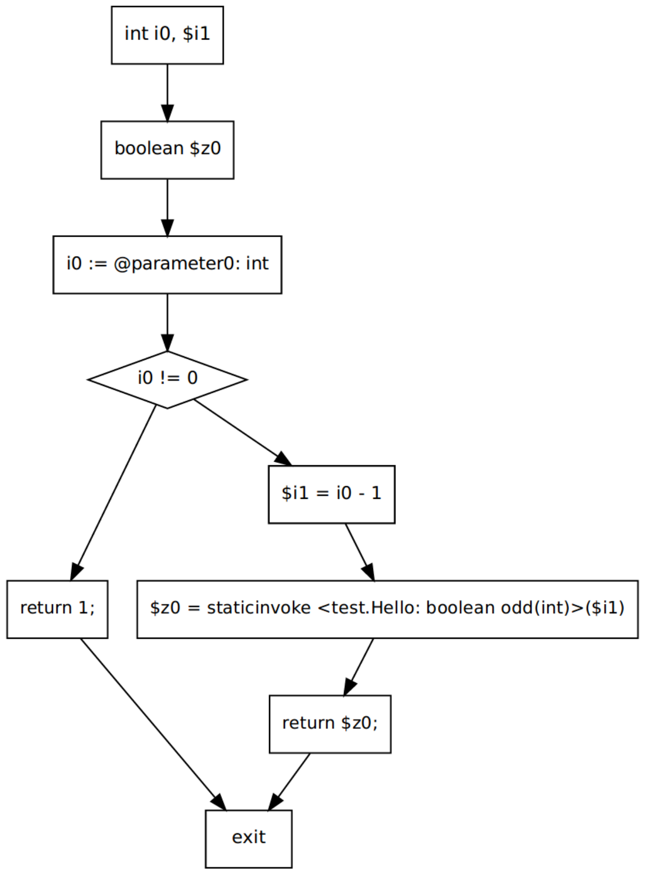


Figure 2: Control Flow Graph of even function

Implementation for Fibonacci function and corresponding graph:

```
public static int fibonacci(int n){
    if (n == 0 || n == 1){
        return 1;
    } else {
        return fibonacci(n-1) + fibonacci(n-2)
    }
}
```

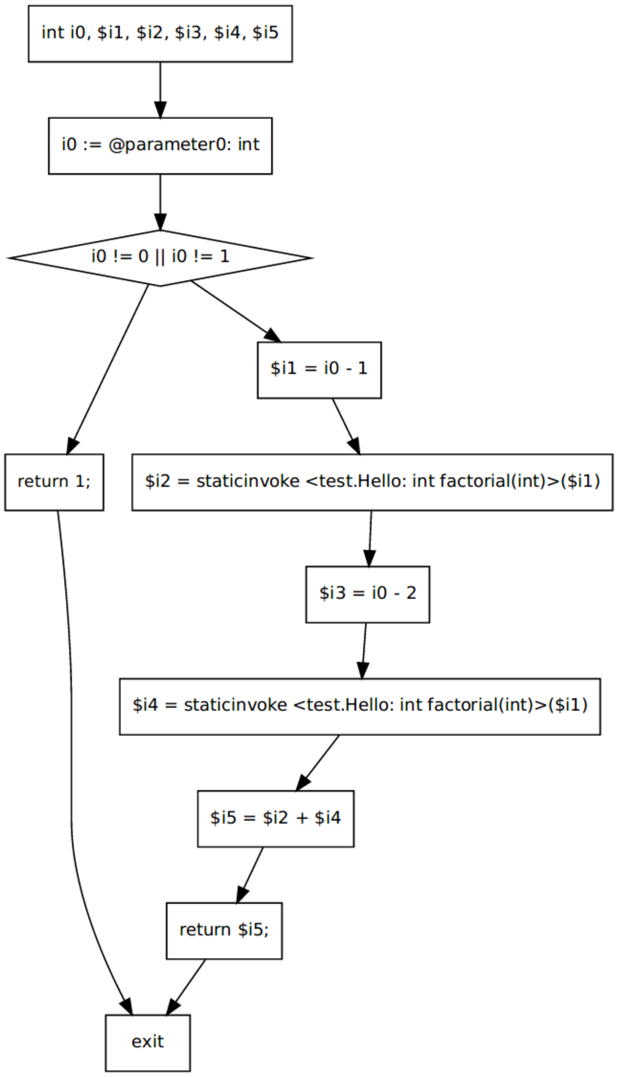
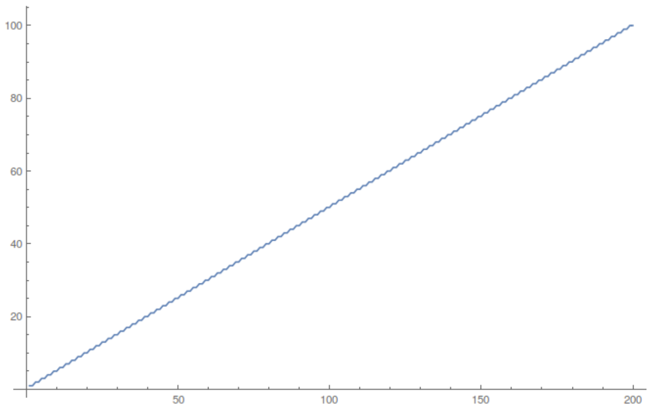
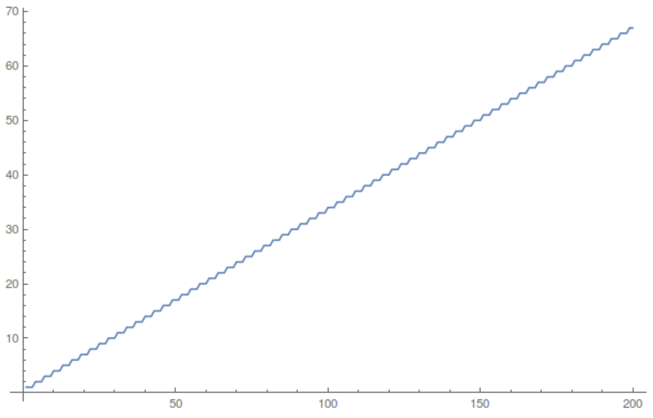

int i0, $i1, $i2, $i3, $i4, $i5
i0 := @parameter0: int
i0 != 0 || i0 != 1
$i1 = i0 - 1
return 1;
$i2 = staticinvoke <test.Hello: int factorial(int)>($i1)
$i3 = i0 - 2
$i4 = staticinvoke <test.Hello: int factorial(int)>($i1)
$i5 = $i2 + $i4
return $i5;
exit

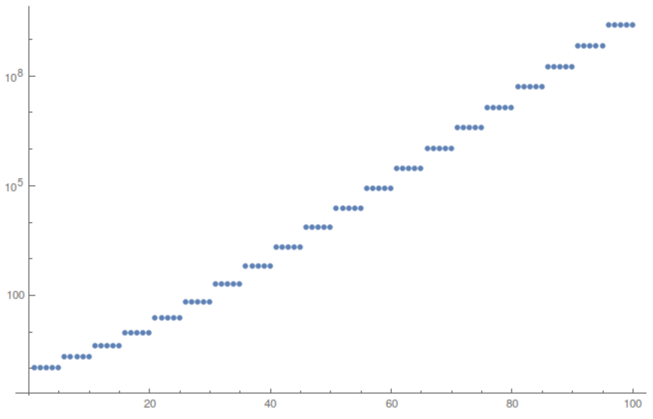
Figure 3: Control Flow Graph of Fibonacci function



(a) Plot of even/odd function



(b) Plot of factorial function



(c) Log plot of fibonacci function

Figure 4: Zoomed version of Figure 2