Use of Control Flow Graphs to Assist with Fault Localization

Zhuo (Cecilia) Chen and Chris Murphy

Computer Science Department, Bryn Mawr College

Abstract

Debugging, especially bug localization, is a time-consuming but necessary process for coding. In the study "Visualization of Test Information to Assist Fault Localization," Jones et al. proposed a method that displays lines of code in different colors based on the likelihood of containing bugs.

In this research, we aim to improve upon this approach by introducing two key modifications. Firstly, we display the code as a Control Flow Graph, providing users with a clear visualization of the code's path likely to contain bugs. Secondly, to evaluate the effectiveness of the approach, we conducted an experiment to determine the accuracy of code highlighting for bug localization, and sought to improve the accuracy.

This work presents a prototype of a Control Flow Graph that uses color to highlight parts of code that are likely to contain a bug. We also show that it is possible to improve the accuracy of the approach by considering edges coming from conditional statements in the graph.

Introduction

Debugging is an important aspect of software engineering, which includes finding and fixing the errors. Locating faults, or bugs, in software can be very costly in terms of both money and time. To locate bugs, developers need to try many different inputs, including enough failing test cases. Statements that are involved in many failing test cases may be those that have a high possibility of containing bugs.

To help developers narrow their search by selecting suspicious statements, Jones et al. proposed a method in "Visualization of Test Information to Assist Fault Localization" that displays lines of code in different colors based on the likelihood of the line containing the bug, which is calculated acoording to the formula:

$$1-\frac{\%passed(s)}{\%passed(s)+\%failed(s)}$$

In order to figure out if the visualization of the code will help with debugging or not, we propose two changes to the previous work: (1) displaying the code as a **Control Flow Graph**, so that the user can more clearly see the path of code likely to contain the bug; (2) evaluating how accurate the formula is, and trying to improve the accuracy.

Research Questions

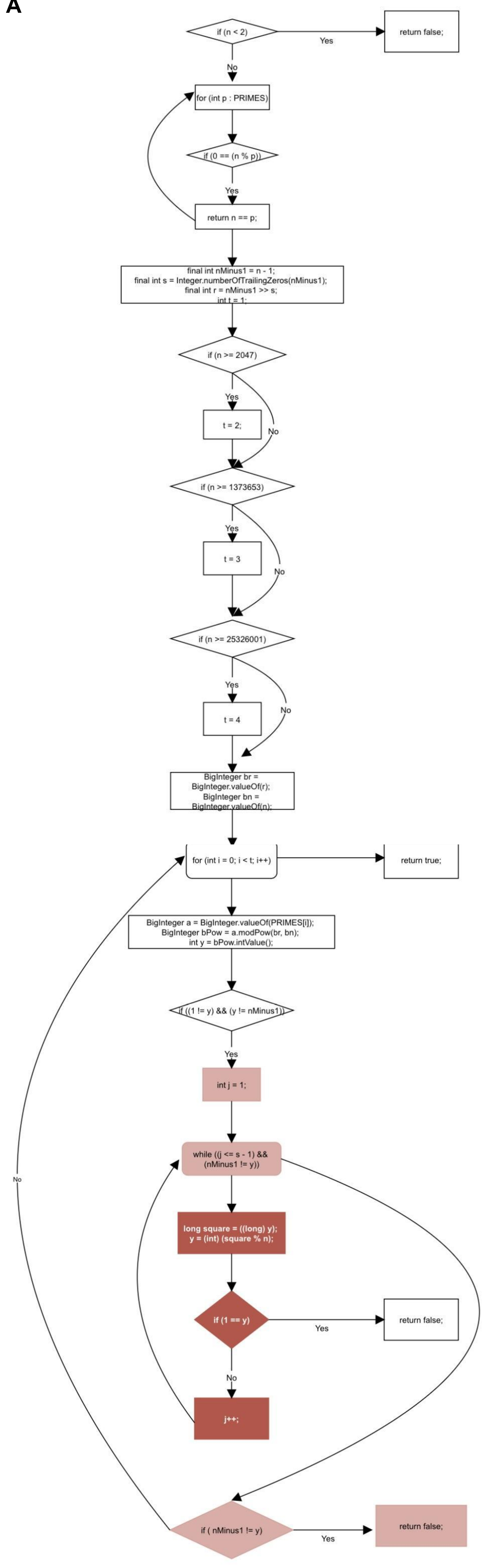
1. How can we visualize the code to emphasize the line that is likely to contain the bug?
2. How accurate is the formula of calculating the likelihood of a line containing a bug?
3. What can we do to improve the accuracy?

Methods

To evaluate the accuracy of the formula to represent the likelihood of a line of code containing a bug, we use mutation analysis to generate 78 different bugs for isPrime() and 206 different bugs for pow() method. We run 10,000 test cases for each bug then use the formula for every line of the code for every buggy version to determine the ranking of the line containing the bug.

Results

A


if (n < 2)
Yes
return false;
No
for (int p : PRIMES)
if (0 == (n % p))
Yes
return n == p;
final int nMinus1 = n - 1;
final int s = Integer.numberOfTrailingZeros(nMinus1);
final int r = nMinus1 >> s;
int t = 1;
if (n >= 2047)
Yes
t = 2;
No
if (n >= 1373653)
Yes
t = 3
No
if (n >= 25326001)
Yes
t = 4
No
BigInteger br = BigInteger.valueOf(r);
BigInteger bn = BigInteger.valueOf(n);
for (int i = 0; i < t; i++)
return true;
BigInteger a = BigInteger.valueOf(PRIMES[i]);
BigInteger bPow = a.modPow(br, bn);
int y = bPow.intValue();
if ((1 != y) && (y != nMinus1))
Yes
int j = 1;
while ((j <= s - 1) && (nMinus1 != y))
long square = ((long) y);
y = (int) (square % n);
if (1 == y)
Yes
return false;
No
j++;
No
if (nMinus1 != y)
Yes
return false;

B

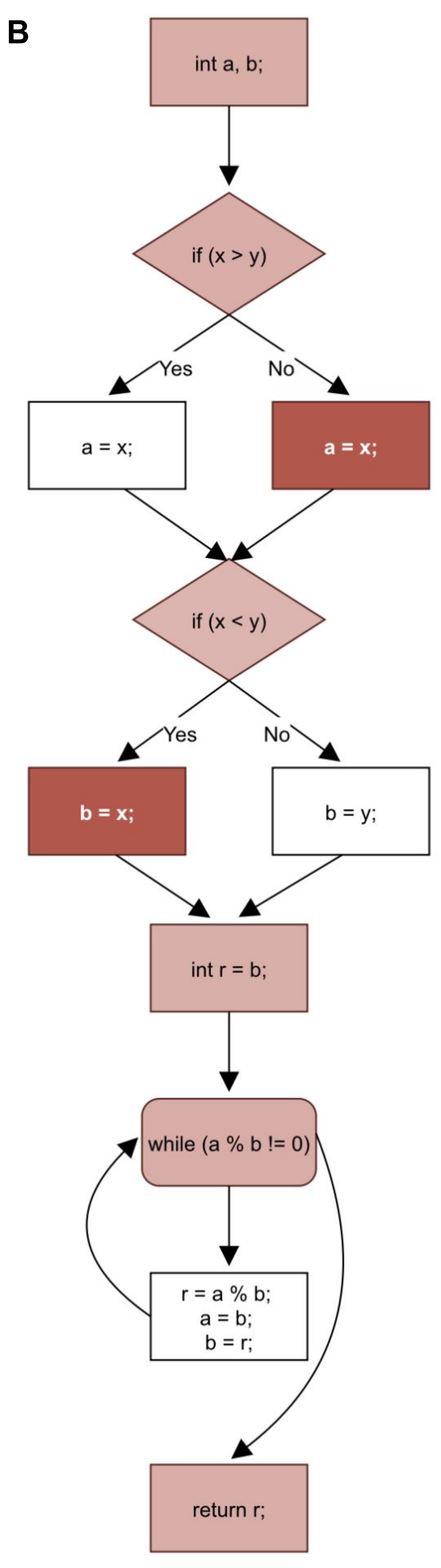
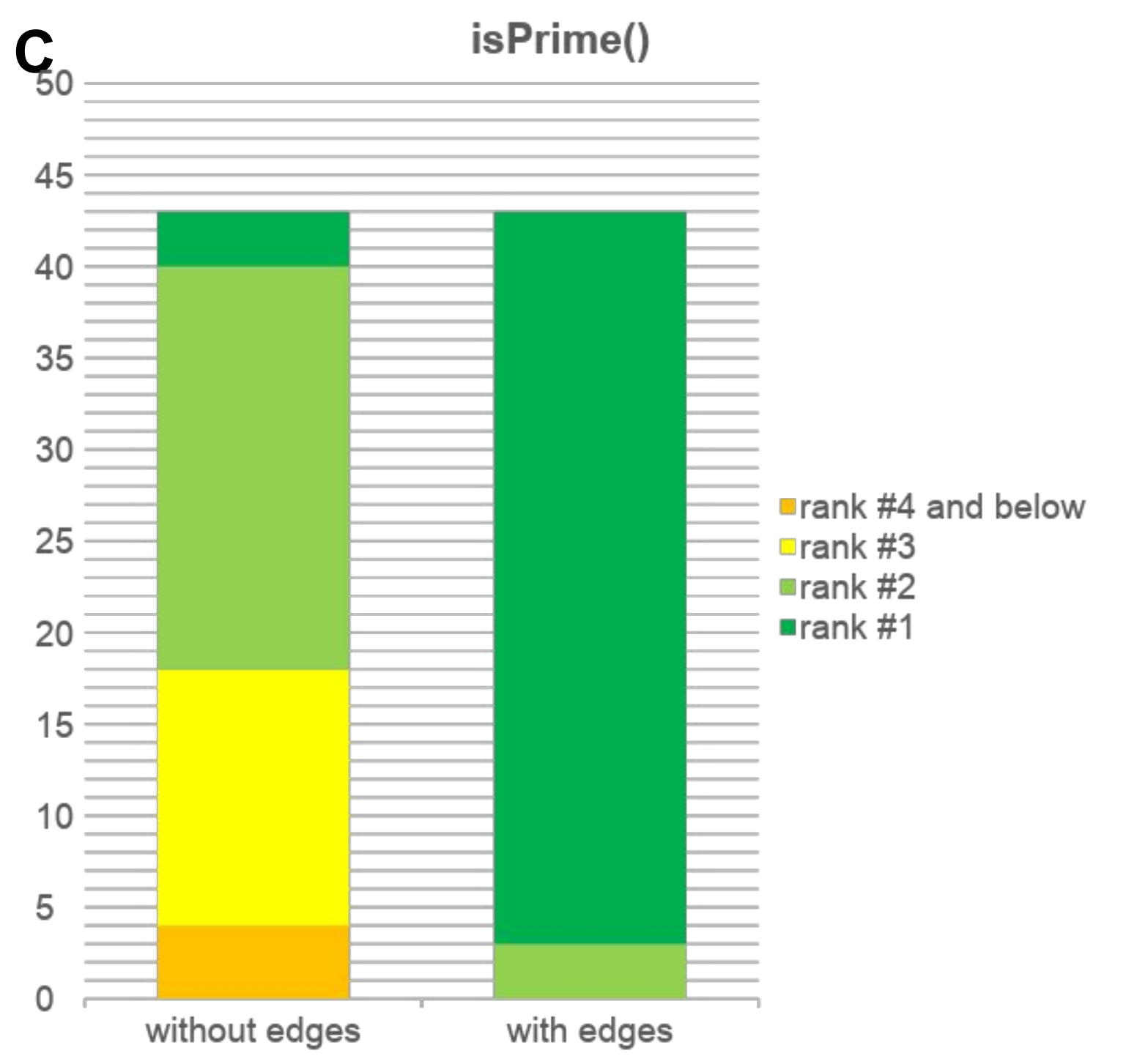


Fig. 1(A-B). The Control Flow Graph is colored based on the ranking of each line. It indicates the lines that are top 1 and top 2 by changing the color of the block, the text color, and the opacity, and bolding so that the user is more likely to notice the lines that are top-ranked.

A) The method isPrime(): the bug is "`square = ((long) y);`" it should be "`square = ((long) y)*y;`"

B) The method euclid(): the bug is "`a = x;`", it should be "`a = y;`"

C



D

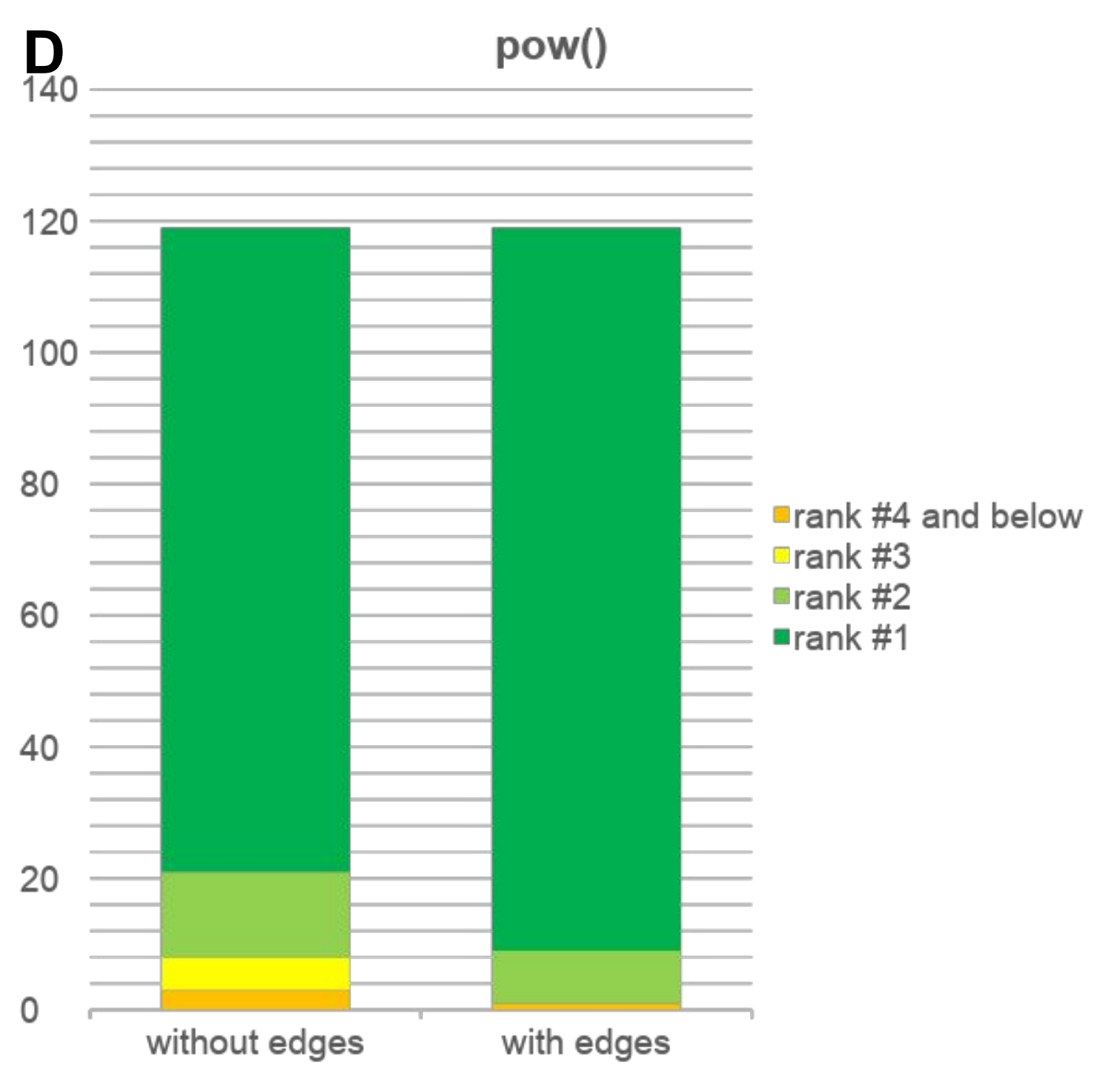


Fig. 2(C-D). In order to improve the accuracy, we applied the formula to edges in the CFG. The diagram illustrates the improvement introduced by incorporating edge considerations for each method, showing the cumulative rankings of the lines actually containing the bug as top-ranked, ranked #2, etc. both with and without considering edges. C) isPrime(): the percentage of buggy versions that are ranked as #1 and #2 has changed from 56% to 100% D) pow(): the percentage of buggy versions that are ranked as #1 has changed from 82% to 92.4%

Discussion

A **Control Flow Graph** can clearly visualize the code and represent the likelihood of each line for containing the bug. The line that has highest probability of containing the bug is more likely to hold the user's attention.

Without considering the edges, bugs in conditional statements might not always be top-ranked according to the formula. The **accuracy** of the formula improved after considering the edges coming from conditional statements for both isPrime() and pow().

However, the **limitation** of this method is that if the code has no conditional statements, then as long as there is a bug, all lines of the code will have the result of 100%, which is not helpful. Moreover, if there are too many parallel lines of code under a conditional statement, it might be too hard for the user to find the bugs efficiently.

Conclusions

We have shown that it is possible to use a Control Flow Graph to visualize code using Jones et al.'s formula to highlight the location of a bug. Though the formula is accurate in many cases, we have shown that it is possible to improve the accuracy by considering edges coming from conditional statements in the Control Flow Graph.

References

Jones, J. A., Harrold, M. J., & Stasko, J. (2002, May). Visualization of test information to assist fault localization. In Proceedings of the 24th international conference on Software engineering (pp. 467-477).

Acknowledgments

I gratefully acknoledge the opportuntity provided by SSR program, the invaluable and inspirational support of Chris Murphy, and the irreplaceable emotional support from my 12-year-old dog Xiaomo, who had to wait much longer to go outside this summer.