Missouri University of Science and Technology
Scholars' Mine

Computer Science Faculty Research & Creative Works
Computer Science

07 Jan 1981

Regular Expressions In A Program Complexity Metric

Kenneth I. Magel
Missouri University of Science and Technology

Follow this and additional works at: https://scholarsmine.mst.edu/comsci_facwork

Part of the Computer Sciences Commons

Recommended Citation

K. I. Magel, "Regular Expressions In A Program Complexity Metric," *ACM SIGPLAN Notices*, vol. 16, no. 7, pp. 61 - 65, Association for Computing Machinery (ACM), Jan 1981.
The definitive version is available at https://doi.org/10.1145/947864.947869

This Article - Journal is brought to you for free and open access by Scholars' Mine. It has been accepted for inclusion in Computer Science Faculty Research & Creative Works by an authorized administrator of Scholars' Mine. This work is protected by U. S. Copyright Law. Unauthorized use including reproduction for redistribution requires the permission of the copyright holder. For more information, please contact scholarsmine@mst.edu.

Regular Expressions in a Program Complexity Metric

Kenneth Magel
Computer Science Department
University of Missouri
Rolla, Missouri 65401

A large variety of metrics have been proposed to measure how complex a computer program is. Unfortunately, nearly all of the proposals imply that everybody knows what complexity is; we just need a simple way to assign a number to the complexity of each program. Zolnowski and Simmons even provide a six step procedure for developing complexity metrics [1]. In fact complexity reasonably can mean any of the following:

(1) how difficult it is to understand the program as indicated by the success people have in answering questions about the program [2];
(2) how difficult it is to debug and maintain the program [3]
(3) how difficult it is to explain the program to someone else;
(4) how difficult it is to change the program in specific ways;
(5) how much effort is involved in writing the program from a design specification;
(6) how many computer resources the program requires to execute.

Some of these definitions (e.g. (2) and (4)) seem to correlate well, but other do not.

Another assumption of most work on complexity metrics which appears unwarranted is that complexity is intrinsic to the program. The complexity of a program as defined by any of the first five definitions seems to depend on four factors:

(1) the person trying to do the tack;
(2) the available documentation of the program or its function;
(3) the environment in which the program is used;
(4) the program itself.

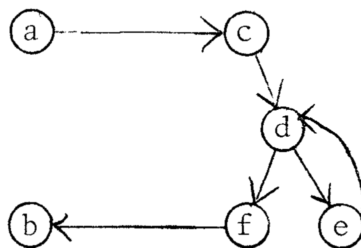
Complexity, like beauty, and pornography, is in the eye of the beholder.

This paper proposes another program complexity metric which suffers from both of the problems mentioned above. It does provide a different view of program control flow complexity than previous metrics, however.[1] Previous metrics have considered the program text and counted all or specific patterns of control constructs [3,4,5]. Regular expressions can be used to look at the complexity of possible execution sequences for a program. The complexity of possible execution sequences should be useful with definitions (2) and (4) above.

The program is represented by its flowgraph. Consecutive non-branching statements which are always executed together (basic blocks) are represented by one node. Branches are represented by arcs from one node to another. Each node is labeled with a name to be used in the regular expression. For example, the following Pascal program [6, page 165], is represented by the associated flowgraph:

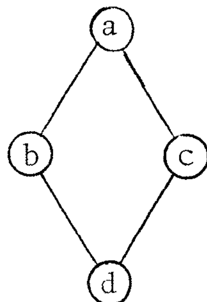
1 Program control flow complexity is the most commonly measured aspect of program complexity. Other important aspects are: program size, data structure complexity, pattern of data usage, and levels of nesting.

```
          program fig 4 (input):
            type p = ↑ letters;
                 letters = record
                             letter:char;
                             next:p
                         end
               var pp:p;
               procedure build;
                 var first, last:p;ltr:char;
                 begin
      c {            new(first);
                     last:=first;
      d {            while 7 eoln(input)do
                      begin
                         read(ltr);
      e                  last↑.letter;=ltr;
                         new(pp);
                         last↑.next :=pp;
                         last :=pp
                     end;
      f          dispose(pp);
                 last↑.next :-nil
                 end;
            begin
a {             build
b {         end.
```



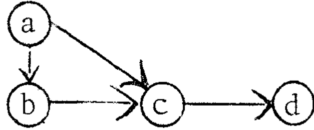
All possible execution sequences for this program are given by the regular expression acd(ed)*fb which means an execution of a followed by an execution of c, an execution of d, zero or more executions of the pair e followed by d, an execution of f and an execution of b.

A choice construction (e.g., if-then-else) would require the alteration (+) operator in the regular expression. Execution sequences for



are represented by a(b+c)d meaning an execution of a followed by an execution of either b or c and then an execution of d.

Confusing program segments require longer regular expressions. For example,



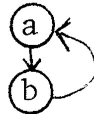
has possible execution sequences given by a((bc)* +c(bc)*)d.

The simplest way to use regular expressions to derive a complexity metric involves counting the number of symbols (operands, operators and parentheses) in the minimally parenthesized regular expression. The first example, acd(ed)*fb, would have a metric of 10(7 operands plus one use of the Kleene closure (*) operator and 2 parentheses). The second example, a(b+c)d, would have a metric of 7. The third example, a((bc)*+c(bc)*)d would have a metric of 16.

Complexity metrics usually are studied by selecting a small sample of arbitrary programs and applying the metric to each of those. Instead, we will examine a more systematic sample of programs, namely all the flowgraphs with one or fewer binary branches. A binary branch is the last statement in a basic block which has two possible successors. Subroutine or function calls will be ignored. There are three flowgraphs with no binary branches:

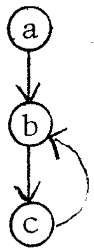
(1)

which has regualr expression, a, and a metric of 1,



(2)

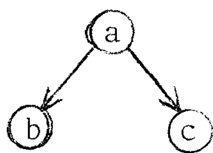
which has a regular expression of ab(ab)* and a metric of 7, and



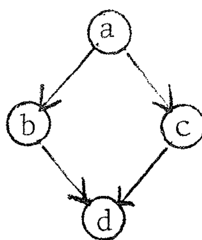
(3)

which has a regular expression of abc(bc)* and a metric of 8. In the latter two cases, we are assuming we do not need to have the program are terminate.

There are eleven flowgraphs with exactly one binary branch.

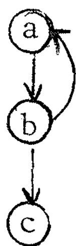


(4)



(5)

(6)

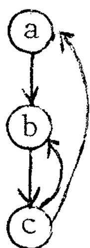


(7)

(8)

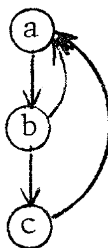
(9)

(10)

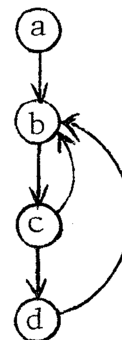


(11)

(12)



(13)



(14)

The regular expression and resulting metric for each of these flowgraphs is given in the table. "n" stands for no node in (6) and does not contribute to the metric.

Flowgraph	Regular Expression	Metric
(4)	a(b+c)	6
(5)	a(b+c)d	7
(6)	a(b+n)c	6
(7)	ab(ab)*c	8
(8)	a bc(bc)*d	9
(9)	ab(ab)*c(bc)*	13
(10)	abc(bc)*d(cd)*	14
(11)	abc(bc)*(abc(bc)*)*	19
(12)	abcd(cd)*(bcd(cd)*)*	20
(13)	ab(ab)*c((ab(ab)*c)*	20
(14)	abc(bc)*d((bc(bc)*d)*	21

Table 1: One Binary Branch

Flowgraphs (9), (10), (11), (12), (13), and (14) represent programs which never terminate. The regular expression metric considers the two nonnested loops in (9) or (10) less complex than the two nested loops in (11), (12), (13), or (14). The forward branch in (6) is considered less complex than the backward branch in (7). (4) and (5) are examples of the if then else construction. That construction is considered simpler than the do until construction of (7) and (8).

[1] Zolnowski, Jean and Dick Simmons, "Measuring Program Complesity in a COBOL Environment", Conference Proceedings, National Computer Conference Volume 49, 1980, pages 757-766.

[2] Shneiderman, Ben. Software Psychology: Human Factors in Computer and Information Systems, Winthrop Publishing Co., Cambridge, Mass 1980.

[3] McCabe, Thomas J. "A Complexity Measure", IEEE Transactions on Software Engineering, December, 1976, pages 308-320.

[4] Chen, Edward T., "Program Complexity and Programmer Productivity", IEEE Transactions on Software Engineering, May, 1978, pages 187-194.

[5] Woodward, M.R. et al., "A Measure of Control Flow Complexity in Program Text", IEEE Transactions as Software Engineering, January 1979, pages 45-50.

[6] Wasserman, Anthony. "Testing and Verification Aspects of Pascal-like Languages", Computer Languages, Vol. 4, no. 314, 1979, pages 155-169.