Python Based Software for Calculating Cyclomatic Complexity

Ms. Pooja Malhotra*
Asst. Professor, Dept. of IT, KJSCE, Mumbai
Mr. Karan Shah
B.E., Student, Dept. of IT, KJSCE, Mumbai

Ms. Jasmi Rathod
B.E., Student, Dept. of IT, KJSCE, Mumbai
Mr. Mehulkaran Mehta
B.E., Student, Dept. of IT, KJSCE, Mumbai

Abstract
Complexity is always considered as an undesired property in software since it is a vital reason of diminishing software quality. Software complexity, deals with how difficult a program is to comprehend and work with. Software metrics are developed and used by the various software organizations for evaluating and guaranteeing software code value, process, and maintenance. In this paper we have tried to address this issue using Cyclomatic complexity [1]. These metrics are a type of control flow metrics. We have developed software in Python which can calculate Cyclomatic complexity for programs written in python. This is an attempt to introduce automation in software metrics design in order to decide its complexity and in the end the quality of the software.

Keywords: Software complexity, Cyclomatic Complexity, Python

1. Introduction

Software complexity is one branch of software metrics that is focused on direct measurement of software qualities, as disparate to ancillary software trials such as project innovative status and testified system fiascos. There are hundreds of software complexity trials, vacillating from the meek, such as basis lines of code, to the esoteric, such as the number of variable definition/usage associations.

Cyclomatic complexity (or conditional complexity) is software metric (measurement) [2]. It was established by Thomas J. McCabe, Sr. in 1976 and is used to indicate the complexity of a program [3]. It directly deals with the number of linearly independent paths through a program's source code. The cyclomatic complexity measure, which measures the volume of decision lucidity in a source code function, does meet the open reengineering norm.

2. Cyclomatic Complexity

Cyclomatic complexity measures the amount of decision logic in single software module. It is based entirely on the structure of software’s control flow graph [4]. It is computed using the control flow graph of the program: the nodes of the graph correspond to indivisible groups of commands of a program, and an engaged edge connects two nodes if the second command might be executed immediately after the first understanding. Cyclomatic complexity may also be pragmatic to discrete functions, components, methods or classes within a package or a program.

3. Control Flow Graph

Control flow graphs describe the logic structure of software components. A component resembles to a single function or subroutine in archetypal languages, has a solitary access and exodus plug, and is able to be used as a design element via a call/return contrivance [5]. Each flow graph consists of nodes and edges (boundaries). The nodes represent computational declarations or lexes, and the edges represent allocation of control among nodes. For example,

---

*Corresponding author.
The above control flow graph represents a meek program where it initiates performing from the starting point, then arrives at loop (set of three nodes closely beneath the starting point) \(^6\). Ongoing the loop, there is an uncertain avowal (set below the loop), and lastly the program exoduses at the ending point. For this graph, \(E = 9\), \(N = 8\) and \(P (paths) = 1\), so the cyclomatic complexity of the program is 3. The cyclomatic complexity of a section of source code is the count of the number of linearly liberated paths through the source code.

Mathematically, the cyclomatic complexity of a structured program is defined with reference to a directed graph containing the basic blocks of the program, with an edge among two basic blocks if control may pass from the first to the second (the control flow graph of the program). The complexity is then termed as \(^7\):

\[
M = E - N + 2P
\]

where
- \(M\) = cyclomatic complexity
- \(E\) = edges
- \(N\) = nodes
- \(P\) = connected modules

4. Implementation

Cyclomatic Complexity is implemented in Python using Radon. Radon is a Python tool that computes various metrics from the source code. It has a set of functions and classes that you can call from within your program to analyze files.

The cc command in radon analyzes Python source files and computes Cyclomatic Complexity. File or directories exclusion is supported through glob outlines. Every positional argument is interpreted as a path. The program then paces over its children and analyzes Python files. Every block will be classified from A (finest complexity score) to F (vilest one). The command used for displaying the complexity along with its rank is \(–s\) \(^8\).

<table>
<thead>
<tr>
<th>Rank</th>
<th>Complexity Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-5</td>
<td>A (low risk - simple block)</td>
</tr>
<tr>
<td>6-10</td>
<td>B (low risk - well-structured and stable block)</td>
</tr>
<tr>
<td>11-20</td>
<td>C (moderate risk - slightly complex block)</td>
</tr>
<tr>
<td>21-30</td>
<td>D (more than moderate risk - more complex block)</td>
</tr>
<tr>
<td>31-40</td>
<td>E (high risk - intricate block, alarming)</td>
</tr>
<tr>
<td>41+</td>
<td>F (very high risk - error-prone, unbalanced block)</td>
</tr>
</tbody>
</table>

Figure a: Flowchart to demonstrate the working of the proposed system

Figure b: Cyclomatic Complexity Rank table

The above table represents the various ranks and associated score displayed when cyclomatic complexity is calculated in python. The score from A to F, where A stands for the finest and best score and F the most complex and vilest one.

The principle to convert the score into an index is subsequently shown:

\[
\text{rank} = \lceil\text{score} \times 10\rceil - H(5 - \text{score})
\]

where, \(H(s)\) stands for the Heaviside Pace Function. The rank is then allied to a letter \((0 = A, 5 = F)\).
The example of implementation of Cyclomatic Complexity for Python is shown below:

```python
import random
from collections import namedtuple

def get_primes(start, stop):
    if start == stop:
        return []
    primes = [2]
    for n in range(3, stop + 1, 2):
        for p in primes:
            if n % p == 0:
                break
        else:
            primes.append(n)
    while primes and primes[0] < start:
        del primes[0]
    return primes

def are_relatively_prime(a, b):
    for n in range(2, min(a, b) + 1):
        if a % n == b % n == 0:
            return False
    return True

def make_key_pair(length):
    if length < 4:
        raise ValueError('cannot generate a key of length less than 4 (got {!r})'.format(length))
    n_min = 1 << (length - 1)
    n_max = (1 << length) - 1
    start = 1 << (length // 2 - 1)
    stop = 1 << (length // 2 + 1)
    primes = get_primes(start, stop)
    while primes:
        p = random.choice(primes)
        primes.remove(p)
        q_candidates = [q for q in primes if n_min <= p * q <= n_max]
        if q_candidates:
            q = random.choice(q_candidates)
    return p, q

Figure c: Python program for DES

Figure d: Computed Cyclomatic Complexity of the above Program

The output of the cyclomatic program above is given by blocks which are nothing but functions or classes or methods [9]. The complexity is first shown according to the block then the function and classes present in it. The Cyclomatic Complexity is given individually to the
functions classes and method used within that block in order to know how complex the program is been built.

5. Conclusion
Cyclomatic Complexity helps the developers and testers to determine independent path’s complexity for its execution. As the number of decision point (rank) increases the complexity of the code increases. With the increase in the complexity the probability of errors and time required for it also increases.

So by reducing the Cyclomatic complexity the programmer would not only reduce its complexity and errors but also its time of execution. Though it doesn’t check or give the goodness of the program but helps in maintaining the control flow in it.

6. References