ABSTRACT

As more curricula integrate computational thinking, designing assessments that are compatible with both student engagement and educator efficacy becomes more urgent. In this paper we discuss the potential role of complexity metrics as an aide to assessing student comprehension. In this first stage of the research, we examined the relationship between manually assigned scores and computer generated scores. While the sample size was too small for statistical results, the positive correlation suggests that this is a plausible research direction.

KEYWORDS

cs4hs, Scratch, student progress

1. INTRODUCTION

Code, one of the fundamental building blocks of the field of computer science, has an advantage over other written languages; it is well defined, clear, and not ambiguous. With attributes like these, one would expect that it would be possible to automate evaluation of student submissions, both for correctness and comprehension of the topics taught in class.

However, this has proven more difficult that one might anticipate. When presented with a problem, many students discover multiple ways of solving it; often using methods that the educator did not anticipate. The educator, after receiving this solution, must either take the time to understand it (which can be substantial), or penalize the student for creative thinking that differs from their normal thought process; surely not a path to the diverse array of cognitive styles we would like to cultivate.

One way to flexibly evaluate student work might be a code complexity metric. This paper will discuss existing methods of evaluating the complexity of code, as a first step towards the objective of creating a system that will grade a submission and come with a result that is similar to one that a human would assign the submission.

Unlike time or space complexity, which measures the change in processing time or resources in relation to input changes, software complexity is only concerned with the textual or syntactic properties of code prior to execution, usually with the goal of determining the impact of these properties on human understanding, testing or writing.
There have been many attempts over the years to measure software complexity; some of the well known ones by McCabe [McCabe 1976], Oviedo [Oviedo 1993], and Halstead [Halstead 1977]. Our goal in this paper is to develop a method which can eventually be used to evaluate the most common complexity measures; such that it reveals correlations between how real people perceive the complexity of a program in comparison to how various complexity metrics understand it. Here we report on a small-scale experiment with 5 people, asking them to rank 5 projects (which produce similar output) on how complex they appear to be. Later research will repeat this experiment on a larger, and more diverse, group of subjects.

In order to more emphasize the focus of this paper on the research's application to education, this paper will utilize the visual programming environment created for the 3Helix program at Rensselaer Polytechnic Institute, CSnap. This is similar to other well-known programs, such as Scratch, but has a greater emphasis on integrating culture into the computer science curriculum. We have included links at the end of this paper to images of these CSnap scripts.

1. ANALYSIS OF EXISTING METRICS

Rather than reinvent something that has been explored extensively in existing works, this paper focuses on analyzing existing metrics to determine which are most applicable to the goal of evaluating student works. The first step of doing this is to look at which metrics have been proven to be correct, and which incorrect.

Most of these metrics have had their accuracy questioned, and for the most part been proven to fail in one or more ways. A very well known paper that described the failure of these metrics is Evaluating Software Complexity Measures, by Elaine J. Weyuker [Weyuker 1988]. Weyuker’s work has been used as the core for many subsequent software complexity papers, including the recent publication by Hongwei Tao, Complexity measure based on program slicing and its validation [Tao & Chen 2014].

Although none of these metrics by themselves are perfect, we hope to find a combination of them that will work together to provide a comprehensive view of a program's complexity. In the next few sections, we will give a brief description of the metrics we are evaluating, and their known limitations.

Many of these metrics utilize control graphs; control graphs are merely a way of mapping the flow of a programs execution to a directed graph. Rather than reiterating the methods for creating these graphs, we recommend McCabes paper, which has a clear presentation of these methods.

Cyclomatic complexity

Thomas McCabe proposed a complexity measure, cyclomatic complexity [McCabe 1976]. In this section, I will briefly describe how it works, and discuss some responses to it (specifically the responses made by Elaine Weyuker in her paper [Weyuker 1988]).

Cyclomatic complexity requires the creation of a control-flow graph; this graph is a representation of the program in the abstract form of a directed graph. The start and end nodes represent the start and end of the program, respectively. We can identify loops in these graphs by looking for nodes that have an incoming edge from one of it's children nodes. We can identify branching and conditionals by looking for nodes which have more than one exit.

This metric was included in the analysis performed by Weyuker in her paper [Weyuker 1988]. Ultimately, she concluded that McCabes metric failed to address 4 of her 9 properties. Notably, it fails to:

- Have finite sets for each “score” it assigns to a program; that is, it could assign the same score to an infinite number of programs
- Account for the interaction of variables when two programs are combined
- Reflect the complexity introduced by the order of statements in a program

Data flow complexity

Oviedo proposed a metric which relies on 2 key components; data flow (DF) and control flow (CF) [Oviedo 1993]. He concludes with the equation $C = \alpha F_C + \beta F_D$ (where $\alpha = \beta = 1$, $C$ is the complexity).
This metric is effectively an extension of the Cyclomatic complexity; it utilizes control flow graphs in a very similar fashion to McCabe's metric, but also accounts for the use of variables in the code. This addition is well reflected in Weyuker's analysis, as it only fails for 2 of her 9 properties. Namely, it fails to:

- Have finite sets for each “score” it assigns to a program; that is, it could assign the same score to an infinite number of programs
- Prevent the concatenation of programs from decreasing their complexity

Lines of Code (LOC)

We will also consider lines of code (also referred to as statement count). This metric is almost universally accepted as not indicative of the complexity of a program, and will provide a sane reference point to verify that we are at least above this baseline.

Lines of code is very dependent on the language in which we are evaluating programs. Since we are using CSnap as a small-scale staging run, we will consider a “line of code” to be a command block (as defined by Scratch, Snap! and CSnap). These blocks include things such as “if”, “go to x coordinate”, etc.

Normalized Compression Distance

Proposed by Rudi Cilibrasi and Paul Vitanyi, Clustering by compression [Cilibrasi & Vitányi 2005] puts forth a very straightforward way of calculating complexity changes between two pieces of code, without knowledge of even the structure of the content. They envisioned this method being used to cluster data for machine learning, however, we believe it may also be a good way to measure change between two programs; if the distance between a start point and an end point are large, there was a great deal of change. If the distance is not significant, there was not a significant amount of change. They proposed using a standard compression method (such as gzip or PPMZ) to approximate the Kolmogorov complexity of a program.

This content-agnostic method has several advantages over the above stated methods; namely, the ease of implementation. However, it would require a baseline to compare to, to determine whether or not the student has made significant change. In cases where the student modifies a scaffold provided by the teacher, this method may be viable.

2. EXPERIMENT SETUP

In order to test our complexity measure, we will first apply it to the domain of education and student progress. The first step of entering this domain is to interface with the primary stakeholders: teachers and educators that work in the schools. Before getting input from teachers, a survey needs to be developed and tested. It will consist of a few scripts, which have varying degrees of complexity. The teachers will ultimately look at these scripts, and provide an absolute and relative complexity ranking that we can use to help calibrate our metric to present them with a very solid example to base their more qualitative feedback on. To ensure the viability of the evaluation before presenting to the teachers, we tested the survey on a group of NSF graduate fellows. In the next section, our initial results are documented.

3. EXPERIMENT RESULTS

We presented the subjects with a survey documented in illustration 11; it consisted of 5 questions, asking both questions about their identity and the sample projects.

And finally, we used a program to calculate the metrics of several of scripts the subjects were presented with.

Table 1. Calculated Results of Sample Scripts.
The scripts presented to the subjects can be seen in figures 1, 3, 5, 7, and 9.

### Survey Results

<table>
<thead>
<tr>
<th>Script</th>
<th>Cyclomatic</th>
<th>Data flow</th>
<th>NCD</th>
<th>LOC</th>
<th>Survey (avg rel.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha</td>
<td>1</td>
<td>2</td>
<td>0.054</td>
<td>23</td>
<td>1.74</td>
</tr>
<tr>
<td>Beta</td>
<td>2</td>
<td>6</td>
<td>0.031</td>
<td>14</td>
<td>3.25</td>
</tr>
<tr>
<td>Bravo</td>
<td>1</td>
<td>2</td>
<td>0.039</td>
<td>11</td>
<td>2.00</td>
</tr>
<tr>
<td>Charlie</td>
<td>6</td>
<td>27</td>
<td>0.056</td>
<td>25</td>
<td>4.50</td>
</tr>
<tr>
<td>Delta</td>
<td>7</td>
<td>23</td>
<td>0.060</td>
<td>19</td>
<td>3.50</td>
</tr>
</tbody>
</table>

We can clearly see a consensus that sample program charlie (figure 7) is the most complex project, and sample program alpha (figure 1) the least complex. There is some disagreement as to how complex delta (figure 9) was; 3 of the respondents listed it as fairly complex (and in most cases, much more complex than the less complex project), except for respondent 2. However, this respondent commented that the reason for the lower ranking was that it didn’t rely on “hidden magic” in the draw triangle block. This raises an interesting discussion around our program, which did not delve into the blocks when calculating complexity. It did this for 2 reasons:

1. Not all blocks are written in CSnap, so the program need to understand Javascript.
2. It is reasonable to expect that you won’t have access to all code all the time; the use of libraries complicates and obfuscates much code in real life.

By not evaluating hidden blocks, we can focus in on the code the user sees, which we believe to be more important in terms of measuring user contribution (but might lead to inaccurate data-flow results).

Comparing to NCD (normalized compression distance) is a bit tricky, as we need to see how similar they are. There is no accurate way to do this based on the survey, and would require a much larger testing group as it would be much more subjective.

Based on this small sample size, no correlations would be statistically significant. However, it does help to clarify the challenges, and demonstrate how a larger experiment may be conducted. A metric that can provide an objective basis to examine differences in subjective assessments of software might be useful in addressing issues such as gender equity. For example, Eskowitz et al [Etzkowitz,Kemelgor,Neuschatz & Uzzi 1992] note that female faculty in science tend to have fewer publications, but that the few have higher citation rates, and that this creates a gender bias when tenure committees are looking more at quantity than quality. They recommend that academics devise a more universal metric that would validate both styles. A similar gender difference in work styles is described by Turkel and Papert [Turkle & Papert 1992], who suggest that a male/female difference can be found in comparing “top-down” vs “bottom-up” code development styles: they suggest that this creates a challenge for assessment which tends to have a subjective
bias towards top-down (just as evaluators in Eskowitz et al found a bias towards quantity over quality). Thus automated analysis of code complexity could be a tool for helping evaluators avoid subjective bias.

With that in mind, there was an intriguing correlation: delta was the only project that did not use code that was “hidden” in blocks, and the two female participants ranked delta lower than the males had ranked it. Again the numbers here are too small for statistical significance, but as we move this system to larger numbers, we will continue to look for such patterns.

4. CONCLUSION

Based on the short survey we conducted, this is a viable method for examining correlations between human graders and several of the computerized metrics.

Another issue not directly broached by these metrics is the use of different programming paradigms; for example, can we verify that a student used recursion? Or looping? One could imagine using the control flow graphs to find these features with ease, and they contribute to the complexity, but we will need to enhance our program to search for them.

REFERENCES


Illustration 1: https://raw.githubusercontent.com/chuck211991/icedutech_media/master/media/alpha.png
Illustration 2: https://raw.githubusercontent.com/chuck211991/icedutech_media/master/media/alpha_0.png
Illustration 3: https://raw.githubusercontent.com/chuck211991/icedutech_media/master/media/beta.png
Illustration 4: https://raw.githubusercontent.com/chuck211991/icedutech_media/master/media/beta_0.png
Illustration 5: https://raw.githubusercontent.com/chuck211991/icedutech_media/master/media/bravo.png
Illustration 6: https://raw.githubusercontent.com/chuck211991/icedutech_media/master/media/bravo_0.png
Illustration 7: https://raw.githubusercontent.com/chuck211991/icedutech_media/master/media/charlie.png
Illustration 8: https://raw.githubusercontent.com/chuck211991/icedutech_media/master/media/charlie_0.png
Illustration 9: https://raw.githubusercontent.com/chuck211991/icedutech_media/master/media/delta.png
Illustration 10: https://raw.githubusercontent.com/chuck211991/icedutech_media/master/media/delta_0.png
Illustration 11: https://raw.githubusercontent.com/chuck211991/icedutech_media/master/survey.txt

The authors would like to acknowledge NSF grant DGE-0947980 in support of this work.