ABSTRACT

Complexity metrics are useful to identify potentially problematic code. But there is little agreement regarding which complexity metrics should be used and exactly what should be measured, partly because many different factors influence code complexity and comprehension. Code regularity has recently been identified as another such factor, which may compensate for other complexity factors, especially in long functions with high cyclomatic complexity. Given that regularity consists of repeated code structures, it has been suggested to measure regularity by compressing the code with standard text compression tools. But such compression can be done in many ways. We compare five widely available compression tools (LZ77, gzip, LZMA, bzip2, and bicom) and four levels of preprocessing the code (using the code as is, or reducing it to a skeleton of keywords and possibly some formatting). The comparison is done in terms of how the different combinations discriminate between different functions, how they correlate with human perceptions of complexity, and how well they handle relatively short functions. The results show that different combinations of compression tool and code preprocessing lead to significantly different levels of discrimination and correlation with human perceptions, and in addition some combinations are extremely bad in handling many functions and should be avoided. Our recommendation is to use gzip or bicom on a code skeleton containing keywords and formatting.

Categories and Subject Descriptors

D.2.8 [Software Engineering]: Metrics—complexity measures

General Terms

Measurement, Experimentation, Human Factors

Keywords

Software complexity metrics, Code regularity, Compression

1. INTRODUCTION

Program comprehension is a vital preliminary step of software maintenance. The ability to comprehend a given program naturally depends on the programmer’s experience, his or her knowledge of the problem domain, and the complexity of the program itself. Our focus is on the measurement of program complexity, and in particular of one specific factor that has an influence on this complexity, namely the code’s regularity.

Measuring code complexity is difficult because there are so many different factors that have an effect on developers who are trying to comprehend, correct, or modify the code. As a result there is no single metric of complexity, and in fact, any given metric will fail to match human perceptions of complexity in some cases [3, 13, 11]. Myriad metrics are therefore used to measure distinct aspects of complexity: McCabe’s cyclomatic complexity and nesting measure control flow complexity [16, 7], Halstead’s metrics measure vocabulary and operator use [6], fan-in and fan-out measure data flow [8], and other metrics measure elements of style and formatting [9, 5]. Regularity was recently introduced as yet another code attribute that may affect comprehension [11, 12, 10, 20]. Specifically, it was demonstrated that developers faced with long regular functions perceive them as less complex than the conventional metrics (e.g. LoC and MCC) suggest, and also perform cognitive tasks better than when faced with shorter non-regular versions of the same functions. An example of such a regular function from the Linux kernel is shown in Figure 1.

In order to further investigate the importance and effects of code regularity we need an objective metric that can quantify the degree to which given code is regular. Intuitively, regularity means that the same structures in the code repeat themselves over and over again. It has therefore been suggested that regularity may be quantified by compressing the code, and noting the compression ratio. This indirect methodology is based on the mechanisms used in compression algorithms, where repeated segments are replaced with pointers to earlier instances in order to derive a shorter representation.

Still, this basic idea may be implemented in many different ways. First, there is the question of which compression scheme to use. In the following we compare common tools such as gzip and bzip2 and more exotic ones like LZMA and bicom. Then there is the question of possible preprocessing of the code, to better express the regularities in the control structure. We therefore compare compression of the
static int amd8111_priv_coalesce(struct net_device *dev) {
    struct amd8111_priv_elp *elp = netdev_priv(dev);
    struct amd8111_priv_twopackets *twopackets =
        elp->twopackets;
    int tx_rate;
    int tx_timeout;
    int tx_size;  
    tx_rate = coalescrtwo_packets -> coalescrtwo_packets;  
    tx_size = coalescrtwo_packets -> sizeof_twpackets;  
    if (tx_rate < 128) {
        coalescrtwo_packets = 256;
        coalescrtwo_packets_size = 256;
    } else if (tx_rate < 128) {
        coalescrtwo_packets = 512;
        coalescrtwo_packets_size = 512;
    } else if (tx_rate < 1024) {
        coalescrtwo_packets = 1024;
        coalescrtwo_packets_size = 1024;
    } else if (tx_rate > 1024) {
        coalescrtwo_packets = tx_rate;
        coalescrtwo_packets_size = 1024;
    }
    return 0;
}

Figure 1: Example of a regular function from the Linux kernel.

raw code with compression of a skeleton containing only the keywords, possibly with some of the formatting.

Our goal in this present work is to find the best combination of compression scheme and preprocessing, within the framework of using compression to quantify regularity. Naturally, this does not go to say that there are no other ways to quantify regularity. However, finding the best parameters is enough to support continued work on code regularity, and is also important for future comparisons with completing approaches.

In order to identify the best combination, we use all of the available combinations to compress 18755 functions taken from seven systems in different domains. These functions have an MCC of 20 or more to exclude short and simple functions where regularity is not expected to play a part. Our results show that different combinations indeed lead to very different results, so it is important to select the compression methodology carefully. In particular, some of the combinations fail to effectively compress thousands of functions, because they (or some of their preprocessed versions) are too short. As being able to handle functions of modest length is important, these combinations should be avoided.

The remainder of this paper is structured as follows. In the next section we motivate our work and present its research questions. Our methodological approach, including a description of the compression schemes and preprocessing levels, is presented in section 3. We present the results and analyze them in section 4, also showing the correlation of regularity with perceived complexity and documentation of the code. Finally, we discuss the results and conclude in section 6.

2. MOTIVATION AND RESEARCH QUESTIONS

In view of the large number of metrics that have been defined for measuring code complexity, it is now accepted that there is no one metric or factor that fully reflects the complexity of source code [4, 17].

In previous work we have suggested regularity as an additional factor that affects code comprehension, especially in long functions, and provided experimental evidence for its significance [12, 10]. Specifically, we conducted several experiments where developers with different levels of experience were required to understand functions and to perform maintenance tasks on functions, where different subjects were actually working on different versions of the same function. Thus we could evaluate the dependence between performance and the style in which the function was coded. Additional experiments required subjects to evaluate and grade a set of functions. The produced rankings provide us with "ground truth" regarding how human developers perceive code complexity. Using this information we can now ensure that our metrics reflect human perception, a quality that is missing in many metrics that were proposed on theoretical grounds.

The preliminary operational definition of regularity we used in that study was based on compression. We applied the gzip tool to compress each of the 30 functions and used the compression ratio as a metric for regularity. This was actually done twice: first with the full function code and then using only the control structure while removing formatting, layout, and expressions.
Comparing the compression ratios with the human grading, we found a weak correlation between the grades and the compression ratios achieved on the whole function. We found a moderate correlation between the grades and the compression ratios of the control structure, and even better correlation when using the grades given based on the visual representation of the functions.

These results show that the methodology of calculating the compression has an effect on the results. Consequently, a systematic investigation of the methodology is needed.

Our ultimate goal is to define an objective metric for code regularity that reflects perceived complexity. Based on the framework of using compression ratios to quantify regularity, this may be itemized into the following research questions:

1. Does it matter what compression scheme is used?
2. What elements of the source code should be compressed?
3. What is the best combination of compression scheme and code preprocessing level that would reliably reflect code regularity?

To answer these questions, we need a way to evaluate the available compression schemes and combinations. We use the following three criteria:

1. Good discrimination. We expect functions with different levels of regularity to exhibit different compression ratios. A good compression algorithm that compresses all the functions to the same degree would be useless for us, even if the compression ratios are all very high. To check this we use thousands of functions from multiple sources, and look at the distributions of compression ratios produced by the different compression-preprocessing combinations.

2. The compression ratio should negatively correlate with perceived complexity: a higher compression ratio means higher regularity which should yield better comprehension. To verify this we use the same 30 functions we used in the previous work, and check the correlation of complexity scores we have with the compression ratios achieved by the different compression-preprocessing combinations.

3. Success on as many functions as possible. Some compression schemes fail to compress some functions, especially when only a minimal skeleton is used, probably because they are too short. We obviously prefer metrics that can work on any function.

3. METHODOLOGICAL APPROACH

3.1 Compression Schemes

As explained above our operational quantification of regularity is based on compression. However, there are many compression schemes available. The compression schemes that we examine in this work are three that are based on the Lempel-Ziv algorithm: LZ77, gzip, and LZMA; bzip2; and bicom. Table 1 summarizes these schemes and indicates the versions used.

Text compression usually works on complete files: an input file is compressed to create an output file. To work on functions, we create temporary files that include only the function of interest.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LZ77</td>
<td>N/A</td>
<td>The basic Lempel-Ziv dictionary-based compression algorithm as implemented by Marcus Geelnard.</td>
</tr>
<tr>
<td>gzip</td>
<td>1.4</td>
<td>A variant of the Lempel-Ziv algorithm as included in the GNU project.</td>
</tr>
<tr>
<td>bzip2</td>
<td>1.0.5</td>
<td>Compression algorithm using the Burrows-Wheeler block sorting transformation and Huffman coding.</td>
</tr>
<tr>
<td>bicom</td>
<td>1.01</td>
<td>Bijective compression based on prediction by partial matching.</td>
</tr>
</tbody>
</table>

The most basic compression scheme we use is the original Lempel-Ziv algorithm LZ77 [23]. This is a dictionary-based compression scheme, where repeated occurrences of a string are replaced with a pointer to the original occurrence. The key point is that the pointer consumes less space than the string itself assuming the matched string is long enough. Literals that are not matched are output verbatim. The dictionary need not be stored, as it can be reconstructed during the decompression.

A very popular version of the Lempel-Ziv algorithm is implemented in the gzip tool, which is part of the common GNU software distribution. It combines LZ77 with Huffman coding. The output file format includes some static overhead (magic number, version number, timestamp, original file name, and CRC check) so in some cases, especially with small input files, the output may be larger than the input.

Another compression scheme based on the Lempel-Ziv algorithm that we use is LZMA. This is based on LZ77 followed by a range encoder. The dictionary size is huge relative to previous implementations, with special support for repeatedly used match distances. The encoding is done using context-based prediction.

Another compression scheme we use is bzip2. This uses, at its core, the Burrows-Wheeler block sorting transformation, which treats blocks of input to create sequences of repetitions of the same symbol. This is then put through run-length encoding and Huffman coding. Similar to the gzip tool, bzip2 always performs the compression even if the compressed file is larger than the input.

The last compression scheme we use is bicom. This is a bijective compressor from the PPM family. Bijective means that it can always operate both ways: any file can be both compressed and decompressed. In other words, it does not produce any specified file format. PPM means prediction by partial matching. This is an adaptive statistical compression scheme, where the last n symbols are used to predict what will come next. Arithmetic coding is used to represent the output. This compressor is efficient even for very short input.
Table 2: Keywords in the C language and their letter-code mappings.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>if</td>
<td>A</td>
</tr>
<tr>
<td>else</td>
<td>B</td>
</tr>
<tr>
<td>while</td>
<td>C</td>
</tr>
<tr>
<td>for</td>
<td>D</td>
</tr>
<tr>
<td>switch</td>
<td>E</td>
</tr>
<tr>
<td>case</td>
<td>F</td>
</tr>
<tr>
<td>do</td>
<td>G</td>
</tr>
<tr>
<td>?</td>
<td>H</td>
</tr>
</tbody>
</table>

sequences. It was developed for a Windows platform, but we easily compiled and used it on a Linux system.

3.2 Code Preprocessing Levels

Regularity in code may occur in different forms, such as repeated block structures, formatting, identifier names, and operators usage. We believe that regularity in structure, which is dominated by the control-flow constructs, has a large effect on the overall understanding of the code. When using compression to quantify regularity, the question is then what parts of the code should be compressed to best reflect the regularity and provide a good correlation with humans' opinions.

We define four levels of code preprocessing. The first level is the raw code, where we take the source code as is (including comments and blank lines) and compress it. The other extreme is the unformatted control flow skeleton, where we remove all expressions, layout, and comments. Thus we are left with just the sequence of control flow constructs (keywords) and braces (to preserve the nesting), with no linebreaks. In between are two levels where we retain the formatting (linebreaks and indentation), in order to better reflect the block structure. The difference between them is that one contains only the formatting, while the other also indicates the existence of individual statements (by retaining each statement's semicolon).

A potential problem with preserving keywords from a given programming language is that keywords have different lengths, and there may be common substrings that cause keywords to overlap. These characteristics may be expected to affect the compression without reflecting any regularity. For example, multiple repetitions of switch may be compressed more than a similar sequence of the shorter if. To prevent such bias in the compression process we replace all the occurrences of each keyword with a single letter. For example, the keyword if is replaced by the letter code A. Table 2 shows the mapping between keywords and letter codes. (It should be noted that such a replacement was not used in our previous study [12].) The results of applying the different transformations are exemplified in Figure 2.

3.3 Data Collection

We have 5 compression schemes combined with 4 code preprocessing levels yielding 20 different combinations. We examine these combinations on 18755 functions from different systems taken from different domains. The different systems, their domains, and the number of functions extracted (filtered) from each system are summarized in Table 3.

Initially, our scripts extracted all functions of all systems. However, there is an intrinsic problem in C source code that is caused by the C preprocessor (CPP) conditional compilation directives. This interweaving causes problems in particular due to unbalanced braces. To avoid this we dropped each source file that has such problems.

Functions that passed the first step were filtered by their cyclomatic complexity value. We took functions with MCC 20 or higher to ensure a minimum size of the function's structure, as regularity is especially meaningful for long functions and compression may fail on very small functions. We use the pmccabe [1] tool to calculate the MCC values of the different functions.

After these two filtering steps we had 18755 different functions. However, for many of these functions some of the different compression schemes still yielded weird size reduction percentages. For example, there were cases with negative reduction percentages and others with values that passed 100%. Removing all these problematic cases led to a reduced set of 7782 functions.

Looking at the 10973 "bad" functions we found that almost all of them have MCC lower than 40, which means that they are relatively small functions. As compression schemes perform better on large inputs, this might explain the negative

<table>
<thead>
<tr>
<th>Name</th>
<th>Version</th>
<th>Domain</th>
<th># functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>WRK-v1.2</td>
<td>Op. syst.</td>
<td>420</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>9</td>
<td>Op. syst.</td>
<td>1413</td>
</tr>
<tr>
<td>OpenSolaris</td>
<td>8</td>
<td>Op. syst.</td>
<td>864</td>
</tr>
<tr>
<td>Linux</td>
<td>2.6.37.5</td>
<td>Op. syst.</td>
<td>2819</td>
</tr>
<tr>
<td>Firefox</td>
<td>9</td>
<td>Browser</td>
<td>681</td>
</tr>
<tr>
<td>GCC</td>
<td>4.8.0</td>
<td>Compiler</td>
<td>1391</td>
</tr>
<tr>
<td>OpenSSL</td>
<td>1.0.0k</td>
<td>Library</td>
<td>194</td>
</tr>
</tbody>
</table>

Figure 2: Example of the four levels of preprocessing the code.

Table 3: The systems from which functions were taken.
values they received. To support this conjecture we looked at all combinations to see where the negative values come from, and found them all in the most extreme preprocessing level (keywords and braces only) compressed by the bzip2 scheme and in some cases also the LZMA scheme. In this level each function is in its shortest form, as we remove all its content including formatting and layout and preserve only control flow keywords which are then replaced with a single letter. Thus the functions may be reduced to a few dozen characters. The problems with bzip2 and LZMA do not mean that such behavior does not occur in other schemes, but it is not as prevalent. Table 7 shows the different combinations and the number of functions out of the total 18755 that each combination failed to compress. Apparently the other schemes handle small inputs better. We show later that, for schemes that do not fail often, using the full 18775 functions or the reduced set of 7782 functions does not lead to significant changes in the results.

4. RESULTS AND ANALYSIS

4.1 Discrimination

As described above we have 20 candidate combinations of compression scheme and preprocessing level. We applied these to 7782 functions taken from 7 different systems that belong to 4 domains. Each function was preprocessed at 4 different levels, and each of the results was then compressed by 5 different compression schemes.

Figure 3 shows the distribution of the results using the complementary cumulative distribution function (survival function) of the obtained compression ratios. This distribution function shows the probability to observe a sample that is bigger than a given value.

According to this figure both the compression scheme and the preprocessing level have a significant effect on the achieved compression. The different schemes and the different preprocessing levels lead to different distributions. This means that selecting the best compression scheme and preprocessing level is indeed important. Arbitrarily selecting a popular compression scheme with some or no preprocessing is inappropriate.

When comparing compression schemes, the figure shows that some compression schemes consistently compress better than others, relatively independently of the preprocessing level. For example, the gzip, and bicom schemes compress very well at all preprocessing levels, so all their distributions are concentrated between moderate and high compression ratios. There is even a slight advantage for the bicom scheme (compresses better than gzip).

The bzip2 and LZMA schemes exhibit similar behavior for three of the preprocessing levels. But with the keywords only preprocessing level (level 2) the distributions also include
correlation between perceived complexity and regularity, where regularity was measured by compression using the `gzip` of a keywords plus braces representation of the code. There was no significant difference between the representations.

We can now compute the correlations between the rankings and all 20 combinations of compression and preprocessing, to see which combination best matches the rankings of the human programmers. The results are presented in Table 5 and Figure 4, for both modes of presenting the functions (visual and listing).

According to these results, using the raw code (preprocessing level 1) has very low correlation across all schemes. The other three preprocessing levels achieve reasonable correla-

Table 4 shows the results for the different combinations, both for the common set of 7782 functions and for the larger set of 18044 functions (out of a total of 18755) that are handled successfully by `LZ77`, `gzip`, and `bicom`.

According to this table combinations at level 2 (keywords and braces only, with no formatting) exhibit the best discrimination, sometimes by a wide margin. One explanation for this is that because it is the shortest representation of the code, compression ratios are necessarily lower, and every little difference in length or regularity has an effect.

Levels 1 (raw) or 3 (keywords with formatting) vie for second place. With `gzip` and `bicom` raw code provides a bit more discrimination, whereas with `LZMA` and `bzip2` the formatted skeleton appears a bit better. With `LZ77` they are essentially the same. Level 4 (including also semicolons for statements) is nearly always the least discriminative.

The results are not changed when looking at different sets of functions. Obviously, in order to achieve a fair comparison, all the combinations should be evaluated on the same set of functions. However, some of the combinations turn out to mishandle a large fraction of the original 18755 functions (this is discussed further below). The problem is that maybe limiting the evaluation to the subset of 7782 functions that all combinations can handle may distort the results regarding the better schemes, which can actually handle many more functions. We therefore also checked the distributions for a much wider set of functions, which are well handled by only three compression schemes. The results for this set are also presented in Table 4. They are consistent with those discussed above for the smaller set of functions.

### 4.2 Correlation of Regularity with Human Perception

In this section we examine the different combinations of compression schemes and preprocessing levels against other factors that are supposed to be related to regularity: perceived complexity and documentation in the source code.

#### 4.2.1 Regularity and Perceived Complexity

In one of the experiments conducted in our previous work 30 diverse functions with high MCC (McCabe's cyclomatic complexity) values were presented to 15 experienced programmers [12]. The experiment was conducted in two phases, on different days. In one phase each subject was presented with listings of the functions and in the other phase he was presented with code structure diagrams (CSD, a visual representation of the code [11]). Which representation came first was randomized across subjects. The subjects were asked to assign a perceived complexity score to each function in each representation. The result was a moderate negative correlation between perceived complexity and regularity, where regularity was measured by compression using `gzip` of a keywords plus braces representation of the code. There was no significant difference between the representations.

We can now compute the correlations between the rankings and all 20 combinations of compression and preprocessing, to see which combination best matches the rankings of the human programmers. The results are presented in Table 5 and Figure 4, for both modes of presenting the functions (visual and listing).

According to these results, using the raw code (preprocessing level 1) has very low correlation across all schemes. The other three preprocessing levels achieve reasonable correla-

Table 4: Discrimination ability of the different combinations, as measured by the difference in compression ratios at the 15th and 85th percentiles of the distributions of Figure 3.

<table>
<thead>
<tr>
<th>Combination</th>
<th>7782 functions Width per func.</th>
<th>18044 functions Width per func.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lz77_1</code></td>
<td>20.6 0.0037</td>
<td>21.0 0.0016</td>
</tr>
<tr>
<td><code>lz77_2</code></td>
<td>26.9 0.0049</td>
<td>29.8 0.0023</td>
</tr>
<tr>
<td><code>lz77_3</code></td>
<td>20.9 0.0038</td>
<td>24.6 0.0019</td>
</tr>
<tr>
<td><code>lz77_4</code></td>
<td>18.1 0.0033</td>
<td>21.2 0.0016</td>
</tr>
<tr>
<td><code>gzip_1</code></td>
<td>13.0 0.0024</td>
<td>13.5 0.0010</td>
</tr>
<tr>
<td><code>gzip_2</code></td>
<td>18.4 0.0033</td>
<td>23.1 0.0018</td>
</tr>
<tr>
<td><code>gzip_3</code></td>
<td>11.6 0.0021</td>
<td>14.6 0.0011</td>
</tr>
<tr>
<td><code>gzip_4</code></td>
<td>9.0 0.0016</td>
<td>11.2 0.0008</td>
</tr>
<tr>
<td><code>lzma_1</code></td>
<td>13.6 0.0025</td>
<td></td>
</tr>
<tr>
<td><code>lzma_2</code></td>
<td>30.1 0.0055</td>
<td></td>
</tr>
<tr>
<td><code>lzma_3</code></td>
<td>16.7 0.0030</td>
<td></td>
</tr>
<tr>
<td><code>lzma_4</code></td>
<td>11.8 0.0021</td>
<td></td>
</tr>
<tr>
<td><code>bzip2_1</code></td>
<td>13.7 0.0025</td>
<td></td>
</tr>
<tr>
<td><code>bzip2_2</code></td>
<td>38.8 0.0071</td>
<td></td>
</tr>
<tr>
<td><code>bzip2_3</code></td>
<td>19.6 0.0036</td>
<td></td>
</tr>
<tr>
<td><code>bzip2_4</code></td>
<td>13.1 0.0024</td>
<td></td>
</tr>
<tr>
<td><code>bicom_1</code></td>
<td>11.0 0.0020</td>
<td>11.0 0.0009</td>
</tr>
<tr>
<td><code>bicom_2</code></td>
<td>13.0 0.0023</td>
<td>18.0 0.0013</td>
</tr>
<tr>
<td><code>bicom_3</code></td>
<td>9.0 0.0016</td>
<td>12.0 0.0009</td>
</tr>
<tr>
<td><code>bicom_4</code></td>
<td>7.0 0.0013</td>
<td>9.0 0.0007</td>
</tr>
</tbody>
</table>

low compression ratios. Interestingly, the `LZ77` scheme has more diverse distributions: one is relatively high, two distribute over moderate up to high values, and one concentrates at rather low values.

When using the results to compare preprocessing levels, we observe that the raw code preprocessing level (level 1) compresses relatively highly across the different schemes. One explanation is that the code is the longest at this level when compared with others, and therefore has more potential for compression. Moreover, the input content at this level is real source code and English text, which are what most compression schemes are optimized to handle (as stated explicitly in the `gzip` manuals for example).

High compression ratios are obviously a desirable trait for compression schemes. But in the context of using compression to measure regularity, uniformly high compression ratios may be counterproductive. Instead, what we want is a good discrimination between input functions that have different degrees of regularity. (In the next section we add to this the requirement that this discrimination also corresponds to complexity as perceived by human developers.)

To assess the discrimination provided by the different combinations of compression and preprocessing, we focus on the central 70% of each distribution. This is the steepest part of the graph, excluding the bottom 15% which are always considerably lower and those above the 85th percentile which are always considerably higher. A large difference between the 15th and 85th percentiles of the compression ratio distribution indicate that good discrimination is possible. A small difference runs the risk that small changes in the code may lead to large and inappropriate changes in the placement in the distribution. In addition, we also divide this span of compression ratios by the number of functions, to see the average difference per function.
Figure 4: Correlation between perceived complexity and compression ratios. Top row shows results when using code listings and second row when using CSD visualizations.

Table 5: Correlations between perceived complexity and compression ratio for different combinations of compression schemes and preprocessing levels.

<table>
<thead>
<tr>
<th>Combination</th>
<th>Code</th>
<th>CSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>lz77_1</td>
<td>-0.179</td>
<td>-0.393</td>
</tr>
<tr>
<td>lz77_2</td>
<td>-0.421</td>
<td>-0.449</td>
</tr>
<tr>
<td>lz77_3</td>
<td>-0.452</td>
<td>-0.564</td>
</tr>
<tr>
<td>lz77_4</td>
<td>-0.495</td>
<td>-0.579</td>
</tr>
<tr>
<td>gzip_1</td>
<td>-0.152</td>
<td>-0.358</td>
</tr>
<tr>
<td>gzip_2</td>
<td>-0.445</td>
<td>-0.567</td>
</tr>
<tr>
<td>gzip_3</td>
<td>-0.520</td>
<td>-0.617</td>
</tr>
<tr>
<td>gzip_4</td>
<td>-0.501</td>
<td>-0.561</td>
</tr>
<tr>
<td>lzma_1</td>
<td>0.150</td>
<td>0.358</td>
</tr>
<tr>
<td>lzma_2</td>
<td>0.381</td>
<td>0.436</td>
</tr>
<tr>
<td>lzma_3</td>
<td>0.446</td>
<td>0.548</td>
</tr>
<tr>
<td>lzma_4</td>
<td>0.466</td>
<td>0.536</td>
</tr>
<tr>
<td>bzip2_1</td>
<td>0.136</td>
<td>0.337</td>
</tr>
<tr>
<td>bzip2_2</td>
<td>0.217</td>
<td>0.193</td>
</tr>
<tr>
<td>bzip2_3</td>
<td>0.323</td>
<td>0.431</td>
</tr>
<tr>
<td>bzip2_4</td>
<td>0.411</td>
<td>0.489</td>
</tr>
<tr>
<td>bicom_1</td>
<td>0.152</td>
<td>0.362</td>
</tr>
<tr>
<td>bicom_2</td>
<td>0.340</td>
<td>0.535</td>
</tr>
<tr>
<td>bicom_3</td>
<td>0.502</td>
<td>0.608</td>
</tr>
<tr>
<td>bicom_4</td>
<td>0.493</td>
<td>0.556</td>
</tr>
</tbody>
</table>

4.2.2 Regularity and Comments

Regularity is characterized by repeated code; similar code segments may consecutively occur within a function. It seems reasonable to assume that programmers document such regular functions less than other non-regular ones. The rationale is that once the first instance of some repeated code segment is documented the programmer fairly believes that following instances of that pattern are understandable by implication, so he would provide less comments for these instances or even not provide comments at all.

Thus we conjecture that the more the function is regular the less likely it is to be documented. To examine this idea we measure the comments of each function of this study and check if there is any correlation between this measure and the regularity as quantified by the 20 different combinations of compression scheme and preprocessing.
Many more functions have MCC below 20 and were not included in our sample to begin with. (The cyclomatic complexity is a relevant threshold criterion as we look at the control structure of each function.)

Functions with relatively low MCC values lead to small input files that might cause the compression algorithms to create compressed files that are larger than the original ones. In particular, one should remember that most of the compression schemes have some headers that enlarge the output files without reflecting real compression.

The problem is critical in levels 3, 4, and especially 2, as in these levels much of the functions’ contents are removed and the resulting input files are very small. This greatly reduces the effectiveness of the compression schemes in identifying and quantifying regular code as they fail to compress these files by shortening them.

We have already seen that more than half of the functions checked failed to be compressed in level 2 of bzip2, and more than a quarter failed in level 2 of LZMA. It is important to mention that other schemes and levels fail also, but not as massively as bzip2 and LZMA. Table 7 shows the numbers of the “bad” functions under the different combinations. Note that in this work we considered only functions with MCC above 20. We expect that functions in the range between 10 and 20 would cause many more failures for the different compression schemes.

While gzip fails for a relatively small number of functions, bicom stands out for its ability to compress small files — it did not fail for a single function, regardless of preprocessing level. We believe that these differences and the inability of the current compression schemes to deal with small files indicate that there is room for considering other compression schemes, especially ones that are good at small files and maybe even irreversible (lossy) compression schemes. This is permissible in our context because we are not concerned with storing the information, just with measuring the regularity.

At the same time, we note that the shorter the function, the smaller the scope it has for regularity. Moreover, short
functions are typically considered easier to understand, so the question of regularity is less pressing for short functions.

5. RELATED WORK

To the best of our knowledge we are the first to study regularity in the context of code comprehension. In [11] we examined more than 1000 versions of the Linux kernel where we identified functions with very high cyclomatic complexity values. We found that these functions are not really as complex as their MCC complexity metric suggests. In particular, many turned out to be well structured and very regular. We then suggested the use of compression (using gzip) as an operational metric to enable the quantification of regularity. A survey we conducted provided empirical evidence for correlation between the measured regularity scores and perceived complexity by developers. In [12] we extended this work to encompass more systems and more domains, finding that regular functions also occur in systems and domains other than Linux.

Based on these results, we set out to verify the conjecture that regularity is one of the factors that allows developers to handle long high-MCC functions successfully. In a controlled experiment we compared the performance of subjects in terms of time and correctness when working on different implementations of the same specification, where one of the implementations adopted a regular style [10]. We found that the subjects working with the regular version achieved better results than others. The tasks that were used to assess comprehension in this experiment were feature adding, bug fixing, and functionality description.

Similar observations and results were reported in [20]. They introduced the idea of Control Flow Pattern (CFP) and Compressed Control Flow Pattern (CCFP). They used CCFPs to eliminate some repetitive structure from flow graphs. They concluded that method with high cyclomatic complexity have very low entropy and are easy to understand.

Sasaki et al. also ascribed the large cyclomatic values that some modules exhibit to the presence of repeated structures such as consecutive if-else structures [19]. They claimed that it would not be so difficult to understand such source code. They proposed to preprocess the code to make complexity measurement more efficient.

Regularity has been noticed before, but not quantified. Chaudhary et al. conducted an experiment to study the effect of control and execution structures on program comprehension [2]. One result that contradicted their intuitive expectation was the positive correlation between the subjects’ score and the control structure complexity. They attributed this result to the existence of syntactic and semantic regularities in the code. They claimed that these regularities reduced the efforts in the learning process and yielded a higher score.

Regularity has also been considered in other areas. Lipson has defined structural regularity as the compressibility of the description of the structure [14]. In his work, many different forms of regularity were described: repetitions, near-repetitions, symmetries, and self-similarities. In addition to the regularity definition, a metric for quantifying the amount of regularity was suggested. It was defined by the inverse of the description length or Kolmogorov complexity.

Recently, Zhao et al. have shown that regularity leads to spontaneous attention [22]. This may be part of the explanation of why regular code is easier to understand.

There are also works that have used the term “regularity” with different meanings. For example, Lozano et al. use regularity in the context of naming conventions, complementary methods, and interface definitions [15]. Zhang suggested a revised version of Halstead’s length equation. He based it on the fact that the distribution of lexical tokens in the studied systems follow Zipf’s law [21]. Similar results, regarding the distribution of lexical tokens, were presented by [18].

6. DISCUSSION AND CONCLUSIONS

We have already shown in a previous work that regularity is yet another factor that may have a substantial effect on code comprehension. We also suggested to measure it using compression. In this study we have performed a methodological investigation of this idea, and considered 20 different combinations of compression scheme and code preprocessing level. We used 5 compression schemes, namely LZ77, LZMA, gzip, bzip2, and bicom. We used 4 levels of preprocessing which are based on control-flow structure, formatting, and statement awareness. The effectiveness of the 20 combinations was evaluated by how well they discriminate between functions, how well their compression ratios correlate with perceived complexity, and how well they handle small functions.

The results show that bzip2 and LZMA are problematic even with not-so-small functions, so they are less useful and should not be used. bicom is best on small files, but has somewhat lower discrimination than gzip. gzip is also very good, except with the most extreme preprocessing.

The gzip scheme achieved the highest correlations with perceived complexity so it best reflects effect on humans. bicom’s performance was very close to that of gzip. Similar results were also achieved by LZ77, with an advantage of being more discriminative.

As for preprocessing levels, level 1 (using the raw code) leads to very low correlations, so this should not be considered and preprocessing should definitely be used. Several interactions occur between the correlations and other attributes. The most extreme preprocessing, level 2, led to the best discrimination, but had somewhat lower correlations than levels 3 and 4. Preprocessing level 4 gives the highest correlations for LZ77, LZMA, and bzip2, but level 3 was better for gzip and bicom.

Our conclusion is that gzip or bicom combined with preprocessing level 3 (retain keywords, braces, and formatting, but not statements) are the best combination. bicom may be better at handling small functions, and has the advantage of not adding a header that distorts the compression ratio (due to its bijective nature). But gzip is more widely available. Luckily, the combination we used in previous work turns out to be near optimal, and therefore the results are valid.

These results indicate that compression is a promising way to measure regularity, but it is important to choose an adequate scheme. Not all schemes correlate with perceived complexity and not all of them have the same discrimination ability. Furthermore, the whole code of the functions is not representative and a preprocessing step on the code should be performed prior to compressing.

Our work suffers from several threats to validity. We preprocessed the code in various levels by removing different things and retaining others. It might be that some of the removed stuff has an effect on regularity and we missed that. For example, Green et al. present a set of coding guidelines
that are partially based on formatting. They suggest considering a program as a table by using vertical alignments, and considering the use of white space to show structure. These guidelines represent factors that are part of regularity. Our work considers indentation and structure but not all these factors.

Another threat is that most compression schemes add headers to the compressed output, which distorts the compression ratio. This can be avoided by careful parsing of the output files to better reflect the true representation of the compressed data.

A third threat is that we evaluated the effectiveness of different combinations based on their correlation with a previous study in which complexity grades were given to 30 functions. Comparisons with larger sets of functions, and with multiple methods of assessing their complexity, would increase confidence in the results and allow for more general conclusions.

For future work, an interesting issue is measuring regularity of small functions. Indeed, we have shown a scheme that is capable of compressing small functions, but it has somewhat lower discrimination ability. Moreover, in this work we examined only functions with cyclomatic complexity of 20 or more, but a significant fraction of functions is below that.

Another avenue is to examine other families of compression schemes which were not examined in this study. For example, lossy compression scheme may be adequate as humans do not really read repeated code line by line and allow themselves to skip predictable parts.

Acknowledgments

this research was supported by the ISRAEL SCIENCE FOUNDATION (grant no. 407/13).

7. REFERENCES


