Measuring the Complexity of Traces Using Shannon Entropy

Abdelwahab Hamou-Lhadj
Department of Electrical and Computer Engineering
Concordia University
1455 de Maisonneuve West
Montreal, Quebec H3G 1M8
abdelw@ece.concordia.ca

Abstract

Exploring the content of large execution traces can be a tedious task without efficient tool support. Building efficient trace analysis tools, however, requires a good understanding of the complexity embedded in traces. Trace complexity has traditionally been measured using the file size or the number of lines in the trace. In this paper, we argue that these metrics provide limited indication of the effort required to understand the content of a trace. We address this issue by introducing new trace complexity metrics based on the concept of entropy. Our metrics measure two important aspects of an execution trace: repeatability and variability. We present a case study where we apply the metrics to several execution traces. A discussion on how we can reduce the complexity of a trace based on these metrics is also presented.

Keywords: Dynamic analysis, trace complexity, Shannon entropy, program comprehension, software maintenance

1. Introduction

The analysis of execution traces can provide valuable insight into the way a software system behaves. Traces, however, can be quite complex due the excessive number of events invoked. Typical traces can easily be millions of lines long.

Existing trace simplification techniques such as the ones presented in [1, 13, 14, 15] have been surveyed and found to be limited in their ability to handle large and complex traces [4]. This is partly attributable to the fact that these techniques were designed without a clear understanding of the amount of work required to understand the content of a trace.

At first glance, one might imagine measuring the amount of information contained in a trace could be rather straightforward: A naïve approach might just be to report the file size or the number of lines in the trace. However, a myriad of subtleties arise, for example:

- The file size and number of lines depend on the schema used to represent the trace and the syntax used to convey the data specified by such a schema.
- Not all elements of a trace are equally important, or contribute equally to complexity. For example, a long series of identical method calls would be rather simpler to understand than a highly varied and non-repetitive sequence of the same length.
- The notion of complexity is itself rather vague, suggesting that we need to be able to measure different aspects that may contribute to complexity so we can later experiment with various approaches to complexity reduction.

The execution traces targeted in this paper are those based on routine calls (i.e., call trees). We use the term 'routine' to represent functions, procedures or methods of classes. Although the system analyzed in this paper is object-oriented, the metrics proposed here can also be applied to non-OO systems.

In this paper, we present two metrics, based on the concept of entropy, to measure the amount of information contained in an execution trace. The first metric is based on the observation that traces are lengthy because they carry way too many repetitions. Therefore, the aim of the metric is to measure the amount of repetitions found in a trace. We call this metric the measure of *repeatability*. The objective of the second metric is to measure the degree of complexity of highly varied and non-repetitive sequences of calls. Our approach is based on the idea that, given a subtree rooted at routine 'r', if we can always predict the next call made by 'r' then this subtree might be easier to understand than another subtree for which this prediction is difficult to make. We refer to this metric as the degree of *variability* of the trace.

This paper is organized as follows: In the next section, we present the metrics along with basic concepts of information theory. In Section 3, we apply the metrics to traces generated from an object-oriented system called Weka. In Section 4, we discuss trace filtering techniques. In Section 5, we present related work. Finally, we conclude our work in Section 6.

2. Shannon Entropy Applied to Traces

Information theory defines several ways of quantities that agree with what an information measure should be.

Shannon entropy is perhaps one most important of such measures, and quantifies the amount of information in a random variable. The basic concept of entropy in information theory has to do with how much randomness exist in a random event. The more uncertain we are about the content of the message, the more informative it is [2].

Given a discrete random variable X with values in S and probability mass function *p(x)*, the Shannon entropy, $H(X)$, of the variable X is defined by:

$$H(X) = -\sum_{x \in S} p(x) \log p(x)$$

Shannon entropy H(X) depends on the probability distribution of X rather than the actual values of X. The logarithm with base two is usually considered in the computation of H(X). In this case, H(X) represents the average number of bits per symbol that is needed to encode the values of X.

H(X) varies from zero to log(|S|); zero meaning that there is no uncertainty, whereas log(|S|) is reached when all elements of X have equal probabilities, in this case, the uncertainty is at its maximum [2].

In the following subsections, we show how Shannon entropy will be used to define the repeatability and variability metrics.

2.1. A Measure of Repeatability

Through this subsection, we will use the simple call tree illustrated in Figure 1. Let S be a set of distinct routines of the trace T of Figure 1, that is S = {A, B, C, D}. Let X be a random variable taking its values from S. We compute the probability distribution x of S by dividing the frequency of occurrences of x by the size of the trace. For example, p(A) = 1/9. In this case, the entropy of variable X using the above formula is H(X) = 1.66. That is, we need an average of 2 bits per symbol of X in order to represent the content of the trace T.

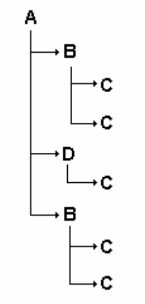

Figure 1. Example of a trace with H(X) = 1.66 and RI = 0.18

We further normalize H(X) so as to have a metric that ranges from 0 to 1. We achieve this by dividing H(X) by log(|S|), which represents the maximum amount of uncertainty possible.

Furthermore, we define the repeatability index, RI, of a trace as:

$$RI = 1 - H(X)/\log(|S|)$$

RI quantifies the amount of certainty with which we can predict the values of X. RI equals zero in two cases. The first case is when the trace (or the part of the trace under study) contains no repetitions (e.g. see Figure 2a). The second case is when all distinct routines invoked are repeated the same number of times. When applied to the entire trace, the second case does not happen in practice since we have at least the entry point of the program that occurs only once. If applied to parts of the trace, we need to fine tune RI so as to take into account the number of events invoked in this portion of the trace. In this paper, we only focus on measuring the repeatability index of the entire trace. Future work will focus on refining the concepts presented in this paper to take into account other types of metrics such as the size of the trace, etc. It is easy to see that RI converges to 1 if the trace consists of one repeated routine (Figure 2b).

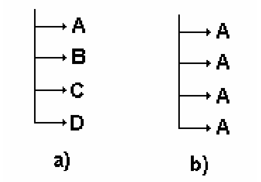

Figure 2. Two traces used to illustrate the repeatability index

It should be noticed that RI does not take into account the order in which the routines are invoked. In other words, any trace whose elements have the same probability distribution will have the same repeatability index. This converges with the definition of RI since it only indicates the amount of repeatability that exists in a trace no matter how or where repetitions occur.

2.2. A Measure of Variability

In this subsection, we are interested in measuring the degree of variability of the calls invoked in the trace. For example, consider the two traces T1 and T2 of Figure 3. These traces have the same number of calls and invoke identical routines. The repeatability index is also the same since both traces have the routines B and C repeated twice and A and D invoked only once. However, in T1, the routine B calls C whenever B occurs, whereas, in T2, the structure of B varies depending on whether C or D is invoked. This suggests that the content of T1 might be easier to understand than T2. For example, one can simply collapse the subtrees rooted at B in T1 in order to have a more compact trace than T2.

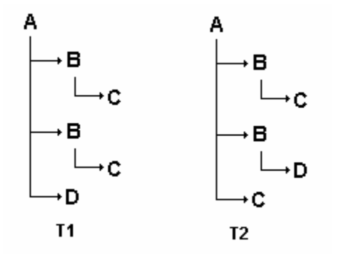

Figure 3. The trace T2 has higher variability than T1 which suggests that it is also more complex

Our approach for computing the variability of a trace is based on the concept of conditional entropy, which measures how much entropy a random variable Y has remaining if we have already learned completely the value of a second random variable X. The higher the conditional entropy the more one can predict the value of a variable by knowing the value of the other variable. When applied to traces T1 and T2 of Figure 3, we can see that when B occurs, in T1, the probability that C occurs is 100%, whereas, in T2, the probability that C appears after knowing that B is 50%.

More formally, suppose X and Y are two random variables that take their values from sets P and R and $p(y|x)$ is the conditional probability of 'y' of R given a value 'x' of P. The conditional entropy H(Y|X), which is also the variability index (VI) of the trace, is then defined as [2]:

$$VI = H(Y \mid X) = -\sum_{x \in P} p(x) \sum_{y \in R} p(y \mid x) \log p(y \mid x)$$

Our approach for computing conditional entropy encompasses several steps. First, we define P as a set of distinct routines of a trace that make calls (at least one) to other routines (i.e. routines with outgoing edges) and R as a set of routines that are called by at least one routine (i.e. the ones with incoming edges). For example, when applied to the trace of Figure 1. P = {A, B, D} and R = {B, C, D}.

Table 1. Matrix corresponding to tree of Figure 1

	B	C	D
A	1	0	1
B	0	1	0

Second, we transform the call tree into a matrix where the rows represent the routines of P and the columns represent R's routines (see Table 1). For each row *x* and column *y*, we set Matrix[x, y] to 1 if there is a call made from the routine *x* to the routine *y*. Note that we have deliberately chosen to ignore the frequency of calls. In other word, the matrix in Table 1 represents the dynamic call graph generated from the trace T1. The rationale behind using the dynamic call graph is that software engineers will most likely want to ignore the number of repetitions when browsing parts of the trace. And in this case, two subtrees that contain identical routines but invoked with different frequencies should be considered similar and have, therefore, the same degree of variability.

The next step is to normalize the content of the matrix such as the entries of each row sum up to one as shown in Table 2. Hence, for a routine *x*, the corresponding vector of the normalized matrix holds the conditional probability p(Y|X = x).

Table 2. Normalized matrix

	B	C	D
A	1/2	0	1/2
B	0	1/1	0
D	0	1/1	0

The conditional entropy equals zero if we can always predict the value of Y by knowing X as illustrated in the following example:

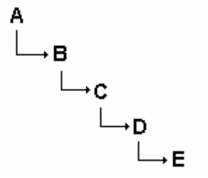

Figure 4. The conditional entropy of this trace is zero

3. Case Study

In this section, we present a case study where we apply the proposed metrics to traces generated from the execution of a Java-based system called Weka (ver. 3.0) [7, 8]. Weka supports a collection of machine learning algorithms for data mining tasks. It contains tools for data pre-processing, classification, regression, clustering, and generation of association rules.

Weka has 10 packages, 147 classes, 1642 public methods, and 95 KLOC. We did not bother to count the number of private methods since they are often ignored when generating traces. The reason is that they are considered mere utility components and do not add much value to the content of a trace from the comprehension perspective.

3.1. Generating Traces

We used our own instrumentation tool based on the BIT framework [6] to insert probes at the entry and exit points of each system's non-private methods. Constructors are treated in the same way as regular methods. Traces are generated as the system runs, and are saved in text files. Although all the target systems come with a GUI version, we can invoke their features using the command line. We favoured the command line approach over the GUI to avoid encumbering the traces with GUI components. A trace file contains the following information:

- Thread name
- Full class name (e.g. weka.core.Instance)
- Method name and
- A nesting level that maintains the order of calls

We noticed the Weka tool uses only one thread, so we ignored the thread information.

We generated several traces from the execution of the target system. The idea was to run the systems invoking different features in order to cover a large portion of the code. This will also allow us to better interpret the results. Table 3 describes the features that have been traced.

Table 3. The features used to generate traces

Trace	Description
T1	Cobweb Clustering algorithm
T2	EM Clustering algorithm
T3	IBk Classification algorithm
T4	Decision Table Classification algorithm
T5	J48 (C4.5) Classification algorithm
T6	Naïve Bayes Classification algorithm
T7	ZeroR Classification algorithm
T8	Apriori Association algorithm

3.2. Applying the Metrics

Table 4 shows the results of applying the repeatability and the variability indices, RI and VI=H(X|Y), to Weka traces. We also added two other metrics, S and M that measure, respectively, the total number of calls (i.e. the size of the trace), and the number of distinct methods invoked in the trace.

Table 4. Results of applying the metrics to Weka traces

Trace	S	M	H(X)	RI	VI = H(X|Y)
T1	193165	75	3.05	0.51	0.63
T2	66645	64	3.16	0.53	0.50
T8	156814	72	3.8	0.62	0.71
T3	39049	114	4.37	0.64	0.63
T4	43681	97	4.27	0.65	0.68
T6	37095	114	4.71	0.69	0.61
T7	12395	93	4.51	0.69	0.58
T5	97413	181	5.22	0.70	0.72

Trace T1 is the largest of all traces in terms of the number of calls invoked although the number of distinct methods contained is relatively small, 75 methods. The repeatability index, RI = 0.51, suggests that the trace has a large number of repetitions. We analyzed the frequency of calls trace T1 in order to understand which parts of the trace are repeated the most. Figure 5 shows the frequency distribution of the routines invoked in T1. The frequency of 66 routines (out of 75) does not exceed 1% of the total number of calls in the trace, while the frequency of the remaining 9 routines varies from 1% to 20%. These routines are shown in Table 5. Based on the Weka documentation, we have concluded that these highly repetitive methods are mere utility components – Routines that help implement the core functionality of the system. Knowing this can help tool builders develop filtering techniques that are based on the removal of utility components, which can be used by software engineers, browsing traces, to automatically identify the most important content of a trace.

Table 5. The routines that are highly repeated in T1

weka.clusterers.Cobweb$CTree.UIndividual
weka.clusterers.Cobweb$CTree.sigma
weka.core.Instance.numAttributes
weka.core.Attribute.numValues
weka.core.FastVector.size
weka.core.Instance.attribute
weka.core.Attribute.isNominal
weka.core.Instances.attribute
weka.core.FastVector.elementAt

The trace T2 has a repeatability index (RI = 0.53) that is similar to T1. However, its variability index, VI = 0.50, indicates that the structure of the routines in T2 is less complex than the ones in T1. This might be due to the fact that the number of routines invoked in T2 is less than T1 (64 compared to 75).

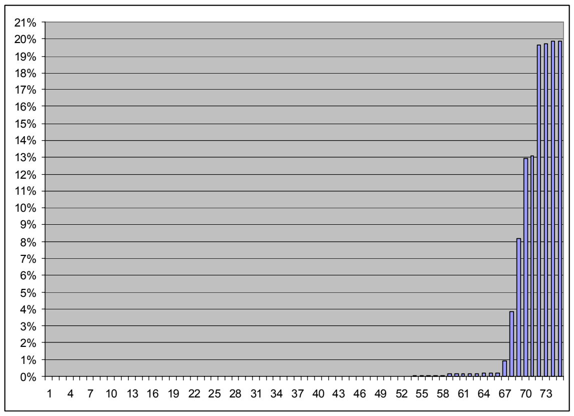

Figure 5. Frequency analysis of Trace T1

T3 and T6 have the same number of distinct methods and similar size. The repeatability and variability indices are also close, which suggests that these traces have similar complexity. A deeper look at the content of both traces reveals that these traces share a considerable number of distinct routines (almost 90% of the routines invoked in T3 are also invoked in T6). These traces represent two classification algorithms (IBk and Naïve Bayes) supported by Weka. The traces show that the implementation of these features share a large portion of the code.

The trace T5 seems to be the most complex of all traces with the highest repeatability and variability indices (RI =

0.70, VI = 0.72). Figure 6 shows the frequency distribution of the routines invoked in T6. We notice that except one routine whose calls represent more than 12% of the total calls, the frequency of any other routine does not exceed 8% of the total size of the trace. The reduction of complexity based on the removal of utilities, as suggested earlier, might not be sufficient when applied to this trace, since it will approximately result in a trace with only about 12% less calls. This suggests that this trace might necessitate the application of more advanced trace reduction techniques (see next Section for ways for reducing complexity).

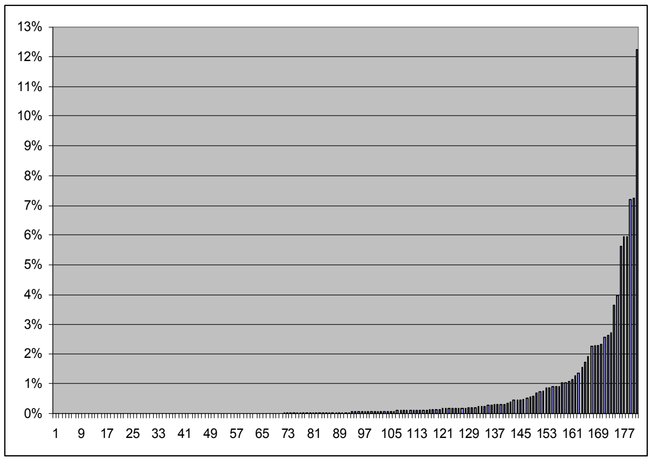

Figure 6. Frequency analysis of Trace T6

4. Reducing Complexity

In this section, we discuss techniques for reducing the complexity of traces based on repeatability and variability indices.

The easiest way to reduce the amount of repetitions in a trace is to collapse contiguous repetitions of calls into one call and keep track of the number of repetitions if necessary. To understand the behavior of an algorithm by analyzing its traces, it is often just as effective to study the trace after most of the repetitions are collapsed. And, in that case, studying a trace of execution over a very large data set would end up being similar to studying a trace of execution over a moderately sized data set. In previous work [3], we showed that this technique reduces the size of traces to between 5% and 46% of the original size. However, the resulting traces continue to have thousands of calls, which makes this basic type of collapsing necessary but not sufficient.

A more interesting alternative is to filter the trace by removing the components that are mere implementation details. For example, the methods of the classes Instance, Attribute, FastVector, and Instances are the ones causing the excessive number of repetitions of Weka traces. These components are clearly utilities and need to be filtered out in order to exhibit the important content of the traces.

However, there is no evidence that the routines with high frequency are always utilities although it might appear to be the case. There is a need to study the relationship between the frequency of occurrence of a routine and the concept of utilities. In addition, there might be many other utilities that do not manifest themselves through their frequency of occurrence. A good example for this is the class Utils, which is invoked in most of the traces presented in the previous section. This class is clearly a utility class despite the fact that its methods are not as frequently invoked as other components.

One approach for reducing the variability of a trace is to consider similar subtrees, which are not necessarily identical, as parts of the same pattern. For example, consider a portion of a trace of routine calls, T1: A(B(CCCCCD)(B(DCC)), where A(B) denotes "A calls B". This trace can be transformed into T2: A(B(CD)) if the contiguous repetitions of "C" and the order of calls from "C" to "D" are ignored when comparing the two subtrees rooted at "B". At a high level, the information contained in T2 might be sufficient for the programmer's purposes.

There exist a variety of matching criteria in the literature (e.g. see [1]) that software engineers can use to explore large traces. However, the sheer size of typical traces makes this exploration process a daunting task, further complicated by the fact that some criteria require, in advance, the setting of specific parameters. In addition, the order in which they are applied can have a significant impact on the resulting trace. What is needed is a set of algorithms that will combine several criteria and automatically suggest appropriate settings for the rapid exploration of the trace content. The algorithms should be designed by taking into consideration the nature of the trace being studied (e.g. trace of routine calls, inter-process messages, etc.), as well as the current goals and experience of the maintainer. They will vary depending on the criteria used, the order in which they are applied, and input parameters specific to each criterion.

5. Related Work

We are not aware of any work that uses entropy to measure the information content of a trace.

There are several metrics that aim at measuring the complexity of traces, among which the most interesting one is presented by Larus in [9] and Reiss et al. in [10]. The authors converted the call tree into a directed acyclic graph by representing repetitive subtrees only once. This transformation allows to factor out repetitions and provides a good indicator of the repeatability aspect of the trace. The drawback is that it requires the transformation of the call tree into a graph. We intend to compare out techniques to the ones presented by these authors.

In [3], we presented several metrics that measure various properties of a trace of routine calls such as the number of distinct components (e.g. routines, classes, packages, etc) invoked in a trace, the number of calls of a trace remaining after all contiguous repetitions are collapsed, etc. These metrics can be used in combination with the ones presented in this paper to gain a deep understanding of the complexity of traces.

In [11], the authors present an extensive list of metrics to measure various aspects of run-time information generated from Java programs ranging from measuring the mere size of traces to assessing the degree to which polymorphism is used in Java program. Although these metrics are interesting, they do not directly quantify the amount of information found in a trace of routine calls. However, they can help design techniques for complexity reduction. For example, one can hide the details of polymorphic methods if the need is to display the content of the trace at a high-level. Knowing the number of polymorphic calls invoked in a trace can help predict the gain resulting from applying this complexity reduction technique.

There is also a large body of work in the area of profiles that focus on measuring cumulative data in order to assess/improve the performance of the system under study. For example, the frequency analysis metrics discussed earlier can be considered as one of these metrics (as shown by Ball in [12]). These metrics can be used in combination with the ones presented in this paper to detect parts of the trace that repetitive and hence design techniques for reducing the amount of repetitiveness.

6. Conclusion and Future Directions

In this paper, we presented two metrics based on the concept of entropy that can be used to assess the complexity of traces. One of our metric RI measures the amount of repetitions contained in a trace. The other metric, referred to as VI, measures the degree of variability of the sequences of calls. The higher is the variability, the more complex the trace.

We applied the metric to traces generated from an object-oriented system called Weka. RI showed that traces contain a very large number of repetitions. Using frequency analysis, we were able to uncover the most repetitive components.

We also discussed ways for reducing the complexity of traces including the use of pattern detection algorithms, the detection of utilities, etc.

Future directions should focus on conducting further experiments with these metrics on larger traces. The long-term objective is to design efficient trace simplification algorithms and tools.

References

[1] W. De Pauw, D. Lorenz, J. Vlissides, M. Wegman, "Execution Patterns in Object-Oriented Visualization", *In Proc. of the 4th USENIX Conference on Object-Oriented Technologies and Systems*, pp. 219-234, 1998

[2] T. M. Cover and J. A. Thomas. Elements of Information Theory. Wiley & Sons, New York, NY, USA, 1991

[3] A. Hamou-Lhadj, and T. Lethbridge, "Measuring Various Properties of Execution Traces to Help Build Better Trace Analysis Tools", *In Proceedings of the 10th International Conference on Engineering of Complex Computer Systems*, IEEE Computer Society, pages 559–568, 2005.

[4] A. Hamou-Lhadj and T. Lethbridge, "A Survey of Trace Exploration Tools and Techniques", *In Proceedings of the 14th IBM Conference of the Centre for Advanced Studies on Collaborative Research*, IBM Press, pages 42-55, 2004.

[5] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge, "Recovering Behavioral Design Models from Execution Traces", In *Proceedings of the 9th European Conference on Software Maintenance and Reengineering,* IEEE Computer Society, pages 112-121, 2005

[6] H. B. Lee, B. G. Zorn, "BIT: A tool for Instrumenting Java Bytecodes". *USENIX Sympo-sium on Internet Technologies and Systems*, 1997, pp. 73-82.

[7] WEKA: http://www.cs.waikato.ac.nz/ml/weka/

[8] I. H. Witten, E. Frank. Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations, Morgan Kaufmann, 1999.

[9] S. P. Reiss, M. Renieris, **"**Encoding program executions", *In Proc. of the 23rd international conference on Software engineering*, Toronto, Canada, pp. 221-230**.**

[10] J. R. Larus, "Whole program paths", *In Proc. of the ACM SIGPLAN '99 conference on Programming language design and implementation*, Atlanta, United States, ACM Press, 1999, pp. 259-269.

[11] B. Dufour, K. Driesen, L. Hendren and C. Verbrugge, "Dynamic Metrics for Java", OOPSLA 2003.

[12] T. Ball, "The Concept of Dynamic Analysis", *In Proceedings of the 7th European Software Engineering Conference,* Springer-Verlag, pages 216-234, 1999.

[13] T. Systä, "Dynamic Reverse Engineering of Java Software", *In Proc. of the ECOOP Workshop on Experiences in Object-Oriented Reengineering*, Lisbon, 1999, pp. 174-175.

[14] D. Jerding, and S. Rugaber, "Using Visualization for Architecture Localization and Extraction", *In Proc. of the 4th Working Conference on Reverse Engineering*, Amsterdam, Netherlands, October 1997, pp. 219-234.

[15] T. Richner, and S. Ducasse, "Using Dynamic Information for the Iterative Recovery of Collaborations and Roles", *In Proc. of the 18th ICSM*, Montréal, Canada, 2002, pp. 34-43.