

Creating a Cognitive Metric of Programming Task Difficulty

Brian de Alwis
Dept of Computer Science
University of British Columbia
Vancouver, B.C., Canada
bsd@cs.ubc.ca

Gail C. Murphy
Dept of Computer Science
University of British Columbia
Vancouver, B.C., Canada
murphy@cs.ubc.ca

Shawn Minto
Dept of Computer Science
University of British Columbia
Vancouver, B.C., Canada
sminto@gmail.com

ABSTRACT

Conducting controlled experiments about programming activities often requires the use of multiple tasks of similar difficulty. In previously reported work about a controlled experiment investigating software exploration tools, we tried to select two change tasks of equivalent difficulty to be performed on a medium-sized code base. Despite careful effort in the selection and confirmation from our pilot subjects finding the two tasks to be of equivalent difficulty, the data from the experiment suggest the subjects found one of the tasks more difficult than the other.

In this paper, we report on early work to create a metric to estimate the cognitive difficulty for a software change task. Such a metric would help in comparing between studies of different tools, and in designing future studies. Our particular approach uses a graph-theoretic statistic to measure the complexity of the task solution by the connectedness of the solution elements. The metric predicts the perceived difficulty for the tasks of our experiment, but fails to predict the perceived difficulty for other tasks to a small program. We discuss these differences and suggest future approaches.

Categories and Subject Descriptors

D.2.8 [Software Engineering]: Metrics

General Terms

Algorithms, Measurement, Theory

1. INTRODUCTION

To assess the utility and compare proposed approaches to aid programming tasks, it can be useful to conduct controlled experiments. This form of experimentation requires comparable programming tasks. In a previously reported on human-based controlled experiment that we conducted to compare several specialized software exploration tools, we found it difficult to determine whether two tasks had comparable difficulty. After carefully selecting the tasks and confirming with pilot subjects that the tasks were comparable, we were surprised to find that the results from

the programmers using a specialized tool were generally either inconclusive, or even worse than than using the standardly available tools. In examining these results, we realized that what were seemingly similar tasks to us were not perceived as such by the participants, and that the tasks had a significant effect on the results. What was perceived to be the more difficult task by the subjects actually required and understanding fewer elements in the system and also required fewer changes.

In this paper, we describe some preliminary work on creating a metric of the cognitive difficulty of a task that might enable better comparisons of tasks prior to conducting an experiment. We represent the program source code as a directed graph, where types, methods, and fields correspond to nodes and static references correspond to edges between nodes. Our proposed metric uses a graph-theoretic statistic called the average clustering coefficient (avgCC) to measure the connectedness of those particular program elements that form the solution to the task. This metric uses a theory that to understand a program element generally requires some understanding of the element's context: how it is defined, how it is used, and what it uses. We assume that the graph characteristic approximates the cognitive characteristic. In using the metric, we find the metric successfully predicts the perceived difficulty for the tasks of our experiment. The metric fails to predict the perceived difficulty for other tasks to a small program.

Such a metric would provide some support in selecting a set of tasks to use in an experiment, and would also provide some support in comparing results across different studies.

This paper proceeds as follows. We review related work in Section 2. In Section 3, we describe the tasks used in our original experiment. In Section 4, we compare the tasks to understand how they differ, and what made the tasks difficult. In Section 5, we describe our preliminary task difficulty metric. In Section 6, we compare the predictive power of our metric on another codebase. We discuss the results in Section 7, and conclude in Section 8.

2. RELATED WORK

Other fields that use comparisons between tasks, such as cognitive psychology, avoid potential issues from using different tasks through the use of isomorphous tasks. Such tasks are created by taking an existing task and applying a series of structure-preserving transformations, such as renames, producing a new, yet isomorphic, task. The importance and meaning associated with terms used in program identifiers (e.g., the Gang-of-Four design patterns [5]) make this approach rather difficult to apply.

In reviewing the literature, we have found little directly related work in estimating the perceived difficulty of a programming task in a realistically sized system. The closest we have found is the study by Curtis *et al.* [2] that found a good correlation between

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CHASE'08, May 13, 2008, Leipzig, Germany.

Copyright 2008 ACM 978-1-60558-039-5/08/05 ...\$5.00.

statement recall by programmers and standard software complexity measures on their modified program. However, this work used very small programs.

There have been a variety of metrics proposed for assessing the comprehensibility of whole programs. For example, Gold *et al.* [6] review and test a set of comprehension metrics based on spatial complexity. However, the tasks used in experiments rarely require understanding a whole program, and different tasks within a program will have different relative difficulties. Our approach differs in attempting to evaluate the difficulty for a particular task.

There are a number of techniques for assessing the relative difficulty of two tasks using an assessment of subjective mental workload. Using such techniques, a subject performs each task and is then be assessed to determine which of the tasks was considered to be more difficult. In our previous work we used the NASA TLX [7], a simple instrument using two questionnaires [4]. Other techniques include affective evaluations (e.g., [9, 3]) which use the subject's physiological responses to gauge relative mental effort. The use of such methods requires using a new programmer for each pairwise comparison who has had no previous exposure to the candidate tasks or candidate system. Finding new programmers makes such methods increasingly expensive and difficult when selecting from several tasks and several systems. We propose an approach that does not require a human.

3. EXPERIMENT AND TASKS

3.1 Experiment

Our within-subjects experiment compared a programmer's code exploration behaviour when using a specialized software exploration tool and using the standard exploration facilities in Eclipse's Java Development Tools (JDT).¹ A within-subjects design satisfied the requirements of one of the instruments used in our experiment to have a baseline for comparison (the TLX [7], used to assess subjective mental workload), and allowed compensating for strong individual differences seen in programmers (e.g., [11, 10]).

Each subject was asked to investigate and document a solution for two change tasks (see Section 3.2) to an open-source editor, jEdit 4.1-pre6.² This particular version of jEdit comprises approximately 65 KLOC³ and 679 classes. The order of the tasks was randomized between subjects. Each subject was instructed to create a plan for performing the tasks, identifying the program elements that either need to be changed or that need to be understood. The plan was to be appropriate for a senior student to rapidly turn into working code.

Each subject used the normal JDT facilities for the first task, which served as a baseline for comparison as required for the TLX. For the second task, each subject was randomly assigned to use one of the specialized exploration tools being compared. By stressing the creation of documentation to guide another programmer in making the necessary changes, we aimed to emphasize the searching to understand the behaviour of the application.

Full details on this study are available elsewhere [4].

3.2 Tasks

We had two tasks, which we refer to as AS and SR. One of the tasks, AS, involved adding a capability to jEdit's auto-saving functionality, and was chosen because it was identical to that used in

previous work [10]. The other task, SR, involved reloading settings when the settings files were edited within jEdit. We identified this task by examining the jEdit source code and identifying a small piece of functionality that was easily removed. The SR task seemed comparable to AS: both interacted with only small amounts of the overall system, and both used already-existing functionality and required little new code.

The experiment was piloted on three students. Each of these students felt the tasks were of equivalent difficulty.

Model solutions to the tasks were created by identifying the methods, fields, and types in the existing source that were either necessary to understand or that required modification. The model solutions did not include new methods or new types that might have been necessary. Types were only included when the solution required them to be subclassed, or when their entire workings were required to be understood in detail. The model solution for the AS task was identified from the solution from Robillard *et al.*'s original study [10]. As the SR task was created by removing existing code, we simply examined the code that was removed.

Because finding a starting point is difficult [12], we provided an initial *seed*, a single pertinent element to help start each task.

3.3 Results of Experiment

We expected programmers using the specialize exploration tools would do better than when using JDT alone. Our results instead showed a confounding effect from our choice of tasks, despite selecting tasks that we thought were similar in scale and that had been successfully piloted. Our results in particular suggested that programmers found the SR task to be more difficult than the AS task.

4. QUALITATIVE TASK ASSESSMENT

What made the SR task, which initially seemed to be of similar difficulty, to be perceived as more difficult? A qualitative *post hoc* examination of the solutions of these two tasks has revealed two differences that may contribute to the perceived difficulty: the types of changes required as part of the solution, and the background information about jEdit subsystems required to understand and accomplish the task.

To correctly solve the AS task required understanding the conditions controlling the autosave functionality, identifying the locations that check these conditions and perform the autosave, and finally how to find the right hook in the user interface to enable the functionality. Having obtained this information, the actual change is relatively straight-forward, modifying how and when those conditions are set. This task required understanding or modifying 15 different program elements.

To correctly solve the SR task required understanding an internal message bus framework, which serves to decouple the producers of a message from the consumers of a message, and using it to act upon *editor-changed* messages. Understanding the bus required understanding the type hierarchy of the bus messages, and then identifying the messages to monitor. This task required understanding or modifying 11 different program elements.

From considering only the number of elements required for each solution, AS would appear to be the more difficult task. However, in asking the programmers to explain their proposed solutions, it appeared that SR was found to be more difficult. Although SR required understanding fewer elements than AS, something made understanding SR's elements more difficult. SR's message bus was a particular source of difficulty. We suspect two reasons. First, the message bus was a new body of code and was unrelated to any other code seen previously in jEdit. Second, the decoupling from

¹www.eclipse.org/jdt; verified 2008-01-25.

²www.jedit.org; verified 2008-01-25.

³Thousands of lines of code, only counting non-blank lines of code.

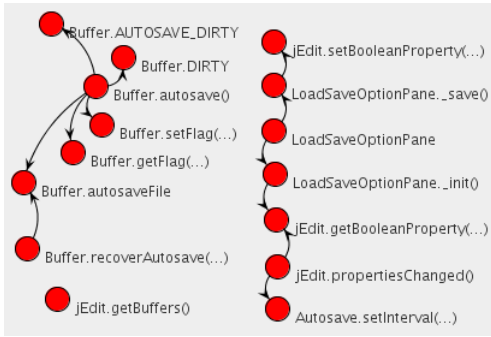


Figure 1: The relationships amongst the solution program elements for the AS task.

using the bus complicates tracing through the call chains once they involve the message bus.

5. A PRELIMINARY METRIC

In an effort to predict the likely perceived difficulty of a task, we linked our qualitative assessment to a graph-based analysis that uses the number of program elements necessary to be understood as part of the solution. The theory behind this metric is that to understand a program element generally requires some understanding of the element's context: how it is defined, how it is used in the system, and what it uses in the system.

To compute our metric, our analysis first transforms the jEdit source code into a program graph; we describe this process in Section 5.1. The analysis then computes a graph-based statistic by examining the neighbourhoods of the nodes corresponding to the solution program elements. We considered two approaches which we describe in Sections 5.2 and 5.3.

5.1 The Program Graph Transformation

We form a directed graph from the program in which nodes correspond to types, methods, and fields, and edges correspond to relations. We add edges for: declarations of field, method, and inner types; extends and implements; returns-type; has-argument of type; references field; calls method; throws exception; catches exception; and creates object instance. We collapse multiple relations between nodes into a single edge, but this is a rare occurrence: less than 1% of edges in the jEdit program graph have multiple relations. These edges were between a method and a type, and involved some combination of returning the type, creating an instance of the type, catching or throwing an exception type, or having an argument of the type.

In creating our program graph, we specifically excluded types from the Java run-time (e.g., java.*, javax.*, com.sun.*) and Java primitive types: since the programmers in our experiment were experts, we assumed that these types were already known. As jEdit is a Swing application, we did include java.awt and javax.swing.

Figures 1 and 2 show subgraphs for the model solutions. As described in Section 3.2, the nodes correspond to the program elements that are necessary to understand to craft a solution for the task. These graphs include any edges that exist between the elements in the solution: these edges correspond to program relations that exist between the program elements, and indicate program elements that could be found either from direct examination of the source code or using a simple query.

The two solution subgraphs are not connected, meaning that some of the solution program elements are not directly reachable from

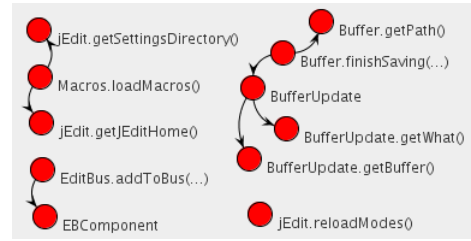


Figure 2: The relationships amongst the solution program elements for the SR task.

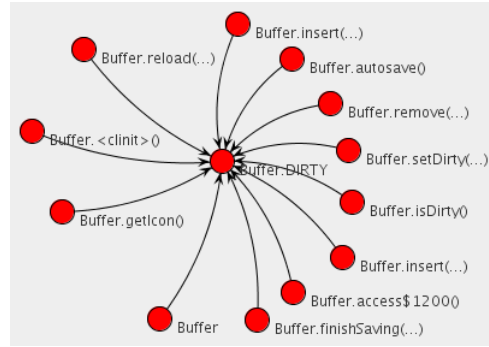


Figure 3: The local context of the field Buffer.DIRTY.

others. Disconnectedness means that the programmers would not have been able to determine all the relevant program elements without exploring other areas of the code. It is rare that a program element can be understood in isolation, and usually requires understanding some part of its local context. Figure 3, for example, shows the local context of the static field Buffer.DIRTY. This field has twelve relations: one to its declaring class, and eleven to the different references of this field. By including the local contexts of the solution elements, the solution subgraphs become connected, and we use these subgraphs as the basis for our analyses.

5.2 Approach 1: Edge Counting

Our first approach to quantifying the task solution difficulty simply counted the number of elements in the local context of the solution elements. The AS task had 229 elements, and SR 184, which does not match the perceived difficulties from the programmers results. But in comparing a field access to a class inheritance, it is clear that the latter will generally require more work to understand, and a strictly unweighted count does not capture the difficulties for different types of links.

5.3 Approach 2: Clustering Coefficients

Our second analysis approach used an intuition that the difficulty of understanding the local context of a program element is related to the amount of sharing between the program elements forming the local context. Situations where elements in the local context use other elements also in the local context should require less knowledge of the programmer.

We attempted to approximate the degree of sharing in the local context by calculating the average clustering coefficient (avgCC) of the solution elements for a task. The clustering coefficient serves as a measure of the inter-connectedness of a node's neighbourhood, and may be informally considered as the fraction of a vertex's neighbours that are also neighbours of each other. Suppose a node i has k_i neighbours, and there are n_i edges between the neighbours.

Table 1: Characteristics of the solution graphs for jEdit and the two experiment tasks, and Ko *et al.*'s tasks; an asterisk (*) indicates what was perceived to be the most difficult task.

Source	Vertices	Edges	AvgCC	Local Context
jEdit	42 545	173 262	0.39	–
AS	15	12	0.15	229
SR*	11	7	0.08	184
Paint	34 339	135 025	0.39	
C3	2	0	0.17	221
R2	4	2	0.02	372
C2	2	1	0.03	383
R1*	5	2	0.07	147
C1*	6	2	0.17	144

Then the clustering coefficient C_i of node i is defined as follows [1]:

$$C_i = \begin{cases} \frac{n_i}{k_i(k_i-1)} & k_i > 1 \\ 0 & k_i \leq 1 \end{cases}$$

In terms of understanding a program element, a higher value for C_i indicates more sharing between the program elements forming the local context, and thus there are fewer program elements to investigate to understand the original program element. By measuring the avgCC of a solution subgraph, we assess the degree of sharing in the local context of the solution subgraph. Solution subgraphs with a higher avgCC should share more parts, and thus lessen the burden of investigating and learning the interactions.

Table 1 reports the graph metrics for the jEdit graphs. The jEdit program graph as a whole has an avgCC of 0.39. Although the solution to SR would appear to be simpler than that for AS if simply comparing the absolute numbers of elements, their avgCCs are distinctly different. The avgCCs indicate that the local context for SR has fewer elements that are jointly shared. We interpret this as programmers must investigate more elements to identify the correct solution.

6. VALIDATION

To validate this possible metric, we compared the avgCC values to those computed from the five Paint tool tasks used by Ko *et al.* [8]. The Paint tool is a small Java application based on Swing. It is significantly smaller in size than jEdit, being comprised of only 9 classes and 59 methods.

The computed metrics for each task are also reported in Table 1, sorted by perceived task difficulty as assessed by one of the authors (from easiest to hardest). We have marked the two tasks described as the most difficult tasks by Ko *et al.* From these results, the avgCC metric does not correctly predict the task difficulty of the Paint tasks.

7. DISCUSSION AND FUTURE WORK

It is disappointing that our metric does not predict the task difficulty for Paint. This may be attributable to notable differences between jEdit and Paint. Although both are Swing applications, jEdit is significantly bigger and includes a greater amount of separate functionality. Paint is only 9 classes as compared to the 679 in jEdit. An alternative characterization is that Paint is a small application that is based on the Swing framework, whereas jEdit is an application that uses Swing as part of its functionality.

Our avgCC-based metric could be improved in two ways. First, the avgCC only assesses the degree of sharing *within* the local con-

text for a program element, and does not account for the degree of sharing *between* the local contexts of the solution elements. Second, the avgCC is unweighted and treats links from a field access as being equivalent to an inheritance relationship. Given that understanding the message type hierarchy was complex for the programmers, perhaps some weight should be assigned to different types of edges, or using the protection status (e.g., public/private/protected). Our graph construction process also collapses multiple relations between two elements as a single edge; this is a rare occurrence in practice, and arises only for methods with an argument and return-type of the same type.

Our task difficulty metric does not take into account other more ambient information available in the program. For example, programmers will often place the definitions of related elements to be adjacent in a file. However, spatial proximity seems to have a tenuous relation to comprehension [6].

8. CONCLUSIONS

We have constructed a graph-based metric to attempt to predict the perceived difficulty of a change task to a software system. Our metric attempts to quantify the number of program elements necessary to be understood as part of the solution. Our metric successfully predicts the perceived task difficulty for two tasks used in a controlled experiment on a medium-sized software system. However the metric does not correctly predict the relative task difficulty of a set of tasks from a different system.

9. REFERENCES

- [1] D. Chakrabarti and C. Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM Comput. Surv.*, 38(1), June 2006.
- [2] B. Curtis, S. B. Sheppard, P. Milliman, M. A. Borst, and T. Love. Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. *IEEE Trans. Softw. Eng.*, SE-5(2):96–104, Mar. 1979.
- [3] M. Czerwinski, E. Horvitz, and E. Cutrell. Subjective duration assessment: An implicit probe for software usability. In *Proc. Joint IHM–HCI Conference*, volume 2, pages 167–170, 2001.
- [4] B. de Alwis, G. C. Murphy, and M. P. Robillard. A comparative study of three program exploration tools. In *Proc. Int. Conf. Program Compr. (ICPC)*, pages 103–112, 2007.
- [5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, 1994.
- [6] N. E. Gold, A. M. Mohan, and P. J. Layzell. Spatial complexity metrics: an investigation of utility. *IEEE Trans. Softw. Eng.*, 31(3):203–212, Mar. 2005.
- [7] S. G. Hart and L. E. Staveland. Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. In P. A. Hancock and N. Meshkati, editors, *Human Mental Workload*, volume 52 of *Advances in Psychology*, pages 139–183. North-Holland, 1988.
- [8] A. J. Ko, H. H. Aung, and B. A. Myers. Eliciting design requirements for maintenance-oriented IDEs: A detailed study of corrective and perfective maintenance tasks. In *Proc. Int. Conf. Softw. Eng. (ICSE)*, pages 126–135, 2005.
- [9] R. W. Picard and S. B. Daily. Evaluating affective interactions: Alternatives to asking what users feel. In *CHI Workshop on Evaluating Affective Interfaces: Innovative Approaches*, 2005.
- [10] M. P. Robillard, W. Coelho, and G. C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Trans. Softw. Eng.*, 30(12):889–903, 2004.
- [11] M. B. Rosson. Human factors in programming and software development. *ACM Comput. Surv.*, 28(1):193–195, 1996.
- [12] C. Tjortjis and P. Layzell. Expert maintainers' strategies and needs when understanding software: A case study approach. In *Proc. Asia-Pacific Softw. Eng. Conf. (APSEC)*, pages 281–287, 2001.