Automatic Grading of Programming Exercises using Property-Based Testing

Clara Benac Earle*
Universidad Politécnica de Madrid, Spain
cbenac@fi.upm.es

Lars-Åke Fredlund
Universidad Politécnica de Madrid, Spain
lfredlund@fi.upm.es

John Hughes
Chalmers University of Technology and Quviq AB, Göteborg, Sweden
rjmh@chalmers.se

ABSTRACT

We present a framework for automatic grading of programming exercises using property-based testing, a form of model-based black-box testing. Models are developed to assess both the functional behaviour of programs and their algorithmic complexity. From the functional correctness model a large number of test cases are derived automatically. Executing them on the body of exercises gives rise to a (partial) ranking of programs, so that a program A is ranked higher than program B if it fails a strict subset of the test cases failed by B. The model for algorithmic complexity is used to compute worst-case complexity bounds. The framework moreover considers code structural metrics, such as McCabe's cyclomatic complexity, giving rise to a composite program grade that includes both functional, non-functional, and code structural aspects. The framework is evaluated in a course teaching algorithms and data structures using Java.

Keywords

Automated assessment; Testing; Java

1. INTRODUCTION

We consider an approach to automate the grading of programming exercises based on testing. Test models for the exercises are developed using property-based black-box testing [5], and from testing student programs against the models a *ranking* of the programs is derived, based on the observation of which programs have fewer bugs than others. However, teachers typically consider far more factors when correcting programming exercises, e.g., whether the code is well written, whether the implementation of an algorithm has the right algorithmic complexity, etc.

In our approach, property-based testing is reused for computing the algorithmic complexity of a program (e.g., whether the execution time of a program for sorting an array is quadratic in the size of the array). However, such models do not specify the behaviour of the program, but rather guide the testing process into computing the best and worst case complexity. At present this approach is semi-automatic; complexity bounds are computed automatically, but the grading of a program with regards to complexity is the decision of a teacher.

Sect. 2 discusses related work, while Sect. 3 presents the techniques used for grading. Thus, Sect. 3.1 describes how testing is used to evaluate functional behaviour (i.e., find bugs), leading to a ranking of different implementations with regards to their bugs. Sect. 3.2 describes how programs are ranked with regards to algorithmic complexity.

Note that our approach to evaluating functional correctness and algorithmic complexity is largely programming language agnostic. In the article the analysis is applied to Java programs. However, with only small changes to the test models, programs written in other languages are analysable. Sect. 3.3 and Sect. 3.4 show how to combine these analyses with language specific code metrics, such as counting source code statements, and measuring the branching structure of a program (McCabe's cyclomatic complexity).

The resulting framework has been validated in an experiment at the School of Computer Engineering at the Technical University of Madrid, Spain. The setting was a course on algorithms and data structures attended by second year undergraduate students, taught using Java. The course has a heavy emphasis on using practical exercises to improve programming skills, including ten supervised sessions where students are given a simple programming task which they are supposed to complete during a two hour session. Although teachers provide assistance to the students during the programming sessions, the students are graded on how well they complete the programming exercises. On average 100 programs were handed in for each exercise, resulting in a total of around 1000 programming exercises to correct.

The last few years, a single associate professor has been responsible for grading exercises, and has spent, on average, 20 hours per exercise, which amounts to using 12 minutes grading a program. Although having a single teacher do the correction ensures consistent grading, there are several drawbacks too, e.g., students do not receive their exercise grades until three weeks after the hand-in date, and receive little feedback regarding problems with their programs.

*Work partially funded by European Comission FP7 project ICT-2011-317820 (*PROWESS*), Comunidad de Madrid grant S2013/ICE-2731 *(N-Greens Software)* and by Spanish MINECO Project TIN2012-39391-C04-03 (*StrongSoft*).

ITiCSE '16, July 09–13, 2016, Arequipa, Peru.

ISBN 978-1-4503-4231-5/16/07…$15.00

DOI: http://dx.doi.org/10.1145/2899415.2899443

In view of these difficulties, the course on algorithms and data structures presents a setting where introducing automated grading of exercises might lead to an improved course, and this paper reports on an evaluation of this potential. In parallel with the normal running of the algorithms and data structures course, where student programming exercises where graded manually as usual, we developed test models and used structural analysis tools to calculate an automatic grading of exercises. To contrast the different approaches to grading we estimated how *efficient* automatic correction is, by comparing the time spent on developing models for testing programs with the time spent for manual correction. Secondly, we try to estimate the *quality* of automatic correction, i.e., whether the calculated grades accurately reflect the differences in quality of the programs being graded. To answer the second research question, we assumed that the results of the manual grading was perfect, and calculated the average error introduced by using the automatic grading framework. Note that in this article we do not attempt to quantify other expected benefits of introducing automatic grading, i.e., quicker and better feedback to students, which are likely to be at least as important as the grading itself.

As the course focuses on algorithms and efficient implementations of data structures, rather than object-orientation, the evaluation criteria for programs do not include analyses for Java-specific concerns. However, such analyses can be combined with the main results from this work in the same manner as e.g. the McCabe cyclomatic complexity metric was taken into account into a final program grade.

Further details regarding the course, and the evaluation procedure, are described in Sect. 4. The final section 5 summarises the work, and discusses items for future work.

2. RELATED WORK

Continuous assessment during a programming course is key to ensure that students get enough practice as well as early feedback on the quality of their solutions, and many automated assessment tools have been developed. A comprehensive survey can be found in [8]. One notable recent system is ASys [9], which mainly focuses on correctness properties orthogonal and thus complementary to the ones considered in this article. ASys is mainly used to check that the students source code fulfills some syntactical properties desired by the teacher (e.g., compiles correctly, correct implementation of interfaces, etc.). In our framework, the focus is on behavioural properties of the code. Web-CAT [6] is a very interesting testing based grading system, but in contrast to our system, it grades students on how well *they* have tested their code.

An example of a widely used random testing tool for Java is Randoop [11]. Randoop can be used to generate a *suite* of independent failing tests, for one implementation. Our approach, in contrast, generates a suite of tests from *all* the implementations, where each test represents a different bug.

For estimating the algorithmic complexity of a program essentially two methods are used: static analysis of the program [1], and testing-based empirical computation of complexity [3]. As static analysis based tools for estimating complexity bounds are often quite limited in the types of programs for which it is possible to compute complexity bounds automatically, we opted for a testing-based approach.

3. GRADING

This section describes the various techniques used to grade programs with respect to functional correctness (bugs), algorithmic complexity, and structural complexity.

3.1 Functional Correctness

To grade programs with respect to how functionally correct they are, the property-based testing tool QuviQ Erlang QuickCheck [2] (henceforth abbreviated QuickCheck) is used. QuickCheck is part of a by now rather large family of property-based testing tools for different programming languages, inspired by the original Haskell tool [4].

The basic functionality of QuickCheck is simple: when supplied with a data term that encodes a boolean property, which may contain universally quantified variables, QuickCheck generates a random instantiation of the variables, and checks that the resulting boolean property is true. This procedure is by default repeated at most 100 times. If for some instantiation the property returns false, or a runtime exception occurs, an error has been found and testing terminates. Although QuickCheck uses random test case generation instead of coverage directed test generation, this does not mean that the the tool is inferior (see [7] for a thorough comparison on the different methods of test case generation).

QuickCheck moreover provides a state machine based library for testing API's with side effects. The goal of the library is to enable a tester to easily generate randomly sequences of sensible calls to the API, and to decide if the execution of such a call sequence was successful (i.e., the API returned the expected results) or not. Technically the tester, with help from the library, builds a state machine that serves as a model of the behaviour of the API, *and* is used to generate the sequences of calls used to test the API.

A state machine is invariably used as the basic test model to detect problems of functional correctness, even if the API implemented by the program is stateless. The reason is that we want to be able to detect errors caused by the implementation of stateful solutions for pure functional API's (e.g., using static class attributes inappropriately).

Ranking Algorithm.

Perhaps the most obvious method to grade programs depending on how functionally correct they are is to count the number of bugs they have, i.e., a program A is worse than a program B if it has more bugs than program B.

Unfortunately using random black-box testing it is quite difficult to identify multiple program bugs. Repeated testing using the QuickCheck state machine library of a buggy program will yield a set of test cases, such that each test case is a sequence of API calls which when executed by the program under test causes the program to behave incorrectly. The difficulty is in determining when two such (not trivially identical) test cases really exhibit the same underlying bug.

The ranking procedure [5] which is used in this article does not attempt to count the number of bugs a program has, but rather decides that a program A is worse than program B, if, when a large number of test cases have been run, the set of test cases which program B fails is a strict subset of the set of test cases which program A fails. That is, $\mathit{failed}(B) \subset \mathit{failed}(A)$ where $\mathit{failed}(P)$ is the set of test cases failed by program P.

An open source tool (available at https://github.com/fredlund/Ranker.git) implements the ranking procedure as a

partition refinement algorithm, where initially all programs are in the same set. A partition is split when an "interesting" test case is found, i.e., a test case for which the set of failing programs splits an earlier set. To find a new "interesting" test case QuickCheck generates a random new test case, and runs it against all implementations. If the test case does not cause any new split, the process is repeated, until 100 random test cases have been generated without causing a split.

An example of an analysis result is depicted in Fig. 1. Nodes correspond to sets of students whose programs behave identically. Edges are labelled by test cases, such that all students in the nodes located below the edge (transitively) fails the test cases, and all students above the edge executed the tests successfully. For example, the top node contains 63 students; test cases t3 and t6 causes the students 901 and 187 to fail, and moreover student 187 also fails test case t0 which student 901 does not fail.

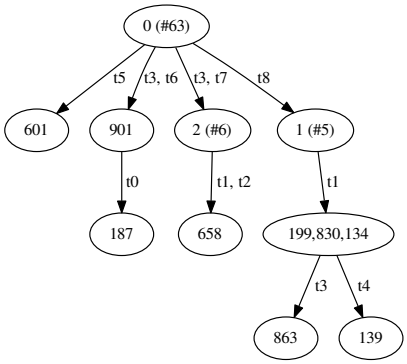

Figure 1: A ranking tree

Using the Ranking for Grading.

We can use such a ranking tree for grading by observing that it defines a partial order: implementations on a higher level (e.g., all the top implementations in Fig.1), which are connected to implementations on a lower level, either directly (e.g., 601) or transitively (e.g., 187), behave strictly better than the lower-level implementations. Implementations that are not hierarchically linked (e.g., 901 and 601) are in a sense not comparable, since there are test cases (t3, t5, t6) that one implementation fails, and the other executes successfully, and vice versa. However, we can convert the partial order into a total one using various heuristics. The heuristic used in the framework uses the maximum path length to an implementation in the ranking tree as the ranking measure for that implementation.

A Derived Test Suite.

The set of test cases labelling the ranking tree forms a test suite, which is used to inform students of the bugs in their programs.

3.2 Algorithmic Complexity

To compute the algorithmic complexity of a program the "Complexity" tool (open source, available at https://github.com/prowessproject/complexity) is used. The tool empirically calculates the best case and worst case algorithmic complexity of a program, e.g., that the complexity of a method to sort an array in Java has the worst case complexity $O(n\ log\ n)$ in terms of the size of the array. The calculation of such complexity measures is accomplished using property-based testing.

To calculate the parameters for a Java method call which yield the worst (or best) performance, the Complexity tool follows a search procedure, guided by the test model. Beginning with a parameter of a small size, parameters of a somewhat larger size are randomly generated guided by the test model, and the method is executed. Next the tool selects a subset of these parameters which are deemed promising (e.g., having worse execution times than calls with other parameters), and repeats the above procedure, generating larger and larger parameters.

When the search procedure terminates, the parameter yielding the worst (or best) execution time for a given parameter size is selected as a data point. The Complexity tool then tries to fit different curves to the resulting data points, and the best fitting curve (e.g., logarithmic or linear) is reported as the result of the complexity estimation.

Developing Performance models.

To use the Complexity tool it is concretely required to (i) define a "complexity model", i.e., a recipe for how to extend a test case of size N to an (interesting) test case of size $N + M$ (where M is usually 1), and (ii) provide a means for running the test case and to measure its execution time (or some other relevant metric). Note that developing such complexity models is not a trivial task, as usually there are too many possible tests to run given the available time for estimating the algorithm complexity, and thus the model must carefully prioritise some test cases which are thought to contribute most to best-case or worst-case behaviour.

Obtaining performance results.

Due to the difficulties of accurately measuring the execution time of method calls in Java (due to caches, just-in time compilation, etc.) we instead measure the number of Java virtual machine instructions executed for a method call, using a debugging build of a normal Java runtime. Such a measure will be invariant over different executions, but the performance predicted may not reflect real-world performance. Nevertheless, for comparing programs in a course on algorithms and data structures, the measure is surely relevant.

Best-case or worst-case.

However, even given repeatable experimental results, we still have no direct way to compare two implementations. What should be compared, best-cases, worst-cases, or both? And how relevant are the figures, really? A truly bad implementation that raises an exception for every combination of arguments will have a very good best- and worst-case complexity, for the simple reason that nothing is computed. We could compute the best and worst behaviour for correct computations only (using the functional correctness model too), but it might well be that the parameter combinations that would then be omitted are exactly the ones with the worst-case behaviour. So we decided not to do this, and to use the worst-case performance figures only, to avoid penalising implementations with bugs twice; whether this is a good decision is left for a future evaluation.

Comparing implementations.

Next, how exactly do we compare such performance figures? We could compare the computed algorithmic complexity, such that a program A is better than a program B if A has a "better" complexity (e.g., $O(n\ log\ n)$ instead of $O(n^2)$), or if they are the same, if the constant factors for A are better than the ones for B. However, one can argue against such a ranking too. If the constant factors for the $O(n\ log\ n)$ program are large, and the ones for the $O(n^2)$ program are small, it is conceivable that the supposedly inferior program has a better performance than the superior program, *for the test parameters used in the experiment*. Moreover, strictly speaking we cannot really say anything about the performance of the programs on bigger examples, as a program may actually contain different algorithms for handling parameters of different sizes. Another option is to compare the performance of two programs based on the size of the areas defined by the measured data points.

However, humans are quite good at interpreting performance data in diagrams. To explore this, we designed a simple tool for manual ranking, permitting to display and interact with plots depicting simultaneously the performance of all measured programs, utilising the gnuplot tool (http://www.gnuplot.info/). Upon invoking the plotting tool the upper figure in Fig. 2 is shown, which displays the worst-case performance graphs for the 83 programs analyzed. Already it is possible to draw conclusions about the relative performance of the implementations. Essentially there are three "performance segments"; the top one having what looks like quadratic complexity, the middle one probably also quadratic, and the lower one looks linear. Using the tool we can select the upper segment, and move it into its own window shown in the lower figure in Fig. 2.

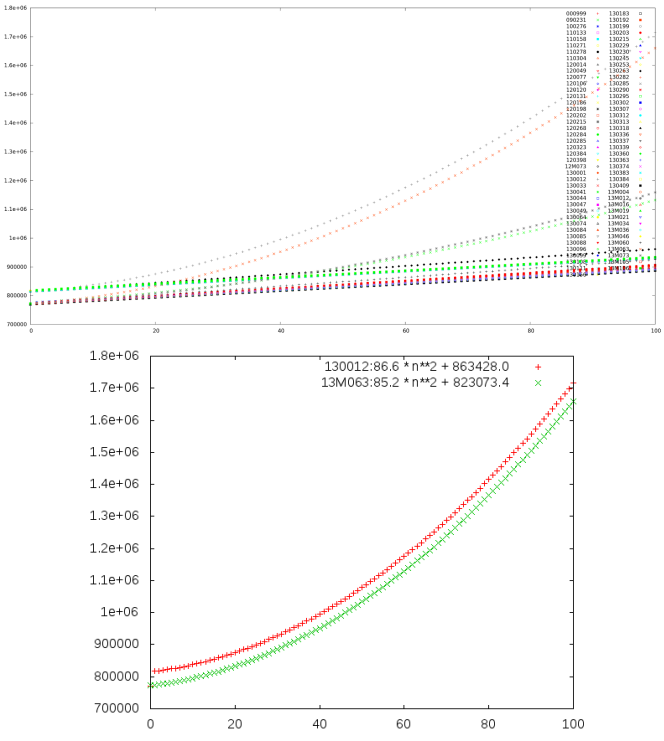

Figure 2: Grading Complexity

We can see that as expected both programs indeed have quadratic complexity (we used an option in the tool to display the complexity measures; this would just have cluttered the first plot). The procedure of partitioning the implementations into different windows is repeated, until we are satisfied. The remaining task is to assign a numeric grade to the different windows, corresponding to a subjective valuation of the performance of the implementations in a window.

The experience from grading the student exercises using the plotting tool is that grading with respect to algorithmic complexity is rapid, and generally without any controversy with regards to how to separate programs into different segments (for all six sets of exercises). Given the performance data for around 100 programs, the task of grading could normally be completed in less than five minutes. The difficulty in this approach seems to lie in how to obtain the performance data, rather than how to interpret it, and here the use of the Complexity tool was very helpful.

3.3 Measuring Source Code Complexity

This section considers code metrics to measure the structural complexity of code. We consider two such metrics: non-commenting source statements (NCSS), and McCabe's cyclomatic complexity metric [10]. Both metrics were computed by the JavaNCSS utility (http://www.kclee.de/clemens/java/javancss/). Informally, NCSS counts the number of ";" and "{" characters in a Java program.

It might be surprising that we do *not* measure a number of other common code metrics, e.g. the number of comments in the code, or checking for common Java anti-patterns. The reason is that the course used to evaluate the framework (see Sect. 4) did not penalise students for a lack of comments – earlier programming courses that students must attend *do* focus on such concerns, but in this course the focus is mainly on penalising *behavioural* defects of the program, rather than structural ones. The only exception is that teachers *do* penalise students who write overly complex or long programs to solve a task, leading to a focus on metrics such as the NCSS and the McCabe ones. For a different course, integrating other code metrics into grading is an easy task, following the methodology described in this article.

The grade of a program with regards to the NCSS metric, then, is computed, somewhat arbitrarily, as the absolute value v of dividing its NCSS metric with the average NCSS metric for the 7 implementations with the lowest NCSS figures (and with no detected functional bugs). If the resulting value v is less than 1.75, the NCSS grade is considered perfect (0), and otherwise its grade is penalised according to the computed value. The McCabe cyclomatic complexity metric is computed analogously.

3.4 Calculating a Composite Program Grade

The combined automatic grade g_{auto} for a program i is:

$$g_{auto} = 10 - p_{fc} * fc - p_{ncss} * ncss - p_{mc} * mc - p_{ac} * ac$$

where 10 is the perfect grade, and $fc, ncss, mc, ac$ are the "penalties" resulting from grading with respect to functional correctness, NCSS (source code complexity), McCabe Cyclomatic complexity, and algorithmic complexity. The constants p_x are weights, which should be tailored to account for the relative importance of the different grades.

4. EVALUATION

Our framework was evaluated in a course on algorithms and data structures using Java, taken by second-year undergraduate students in a four-year software engineering programme. Students participate in ten two hour laboratory sessions where they solve, in groups of two, a simple programming assignment.

Students submit solutions using a web-based system which checks the program against a fixed test suite, to enforce a minimum level of functionality on programs to be considered for grading. The system reports the test results to the student, and accepts the program if it passed the tests.

To evaluate the composite grading scheme, we focused on the first six assignments: counting the number of consecutive elements in an array (nG), removing the first occurrence of a sublist from a list (rm), merge two sorted lists using three variants: with loops (ml), obligatory use of recursion (mr), and use of iterators (mi), and finally sorting a list using the insertion sort algorithm (is).

4.1 Development and Grading Time

The time and effort to develop functional correctness models for these six assignments are summarised below:

	nG	rm	ml	is	mr	mi
model loc.	151	227	251	230	251	251
new loc.	151	124	61	43	0	0
% change	100%	54%	24%	19%	0%	0%
time (h)	4	3	1	1	0	0

The "model loc." row counts the number of lines in the functional correctness model, and the row "new loc." counts the number of new lines compared to the most similar earlier model[1], and "% change" is the change percentage. Note that the measure of change is a rather coarse over-approximation of the difference between the models, as even a very tiny change in a line counts as model change. Finally time (h) counts in hours the time needed for developing the correctness model; note that the merge models (ml, mr and mi) are identical. For completeness we also report on the results of the functional correctness analysis:

	nG	rm	ml	is	mr	mi
# programs	103	91	102	96	93	83
comp. time (s)	129	1055	662	259	503	885
# tests	4	13	14	7	14	21

The "# programs" row indicates the number of programs considered, while the "comp. time" row reports the number of seconds it took to run the ranking algorithm implementation, on a workstation with a six core Intel Xeon E5-1650 CPU (at 3.20GHz) with 32GB RAM installed. The time to run the algorithm depends heavily on a number of factors, e.g., the execution speed of the programs being compared[2], and the test harness used. For the harness the options are to either always create a new Java runtime for each combination of test case and program, or try to reuse an already existing Java runtime. Non-reuse is obviously the safest option, but leads to ranking times at least on an order of a magnitude worse than the figures reported here, and with substantial memory requirements. Full reuse is obviously unsafe, as students frequently use static attributes. Instead, the Javassist (http://jboss-javassist.github.io/javassist/) byte code manipulation library is used to create a new class for each test run against an implementation, while reusing the runtime itself. This is safe, as long as programs do not modify the state of other objects and classes.

The time required to develop an algorithmic complexity model was 4 hours on average. Note that a good algorithmic complexity model is more costly to develop than a functional correctness model. The maximum time permitted for the algorithm complexity analysis was set to 3 hours per sets of exercises. However, this figure is somewhat arbitrary, as investing more time can yield a more precise worst-case complexity bound. Computing the structural complexity metrics for a program was essentially instantaneous.

Finally, recall that on average manual grading took around 20 hours per exercise set, whereas the first (and by far the most time consuming) exercise required around 4 hours to develop the correctness model, and around 6 additional hours for the complexity model. In addition, the automatic grading tools had to be run. The functional correctness of a small percentage of programs (on average 2%) could not be determined automatically, so these had to be corrected manually. A conservative estimate is that the automatic grading took around 6 teacher hours on average, leading to a total figure of around 16 hours, which compares favourably with the average manual grading time of 20 hours. Both these estimates depend on the resources available: the teacher responsible for manual grading had at least 3–4 years of prior experience, and was very efficient at the task. The automatic grading was done by a teacher assisting the students during the laboratory exercises, and with substantial prior experience in developing models for functional correctness.

The row "# tests" reports the size of the test suite derived from functional correctness analysis. These are the test cases that were useful for distinguishing differently behaving programs, i.e., test cases which at least one program fails, and another program correctly executes. As an example, consider the method $nGroup(numOfOccs, elem, list)$ in the first exercise nG, which counts the number of times $elem$ occurs $numOfOccs$ times consecutively in $list$. The following four test cases were generated by the ranking algorithm: $nGroup(2, 0, null)$, $nGroup(1, 1, [1, 0, 1])$, $nGroup(2, 2, [2, 0, 2])$ and $nGroup(1, 0, null)$. The first and last test cases both check whether a program handles a null argument correctly. However, it is surprising that both test cases were necessary for computing a ranking. Manual inspection revealed that there was indeed one program that behaved correctly for $numOfOccs \equiv 2$ and $list \equiv null$, and failed for $numOfOccs \equiv 1$. This is one example where random testing arguably improves on black-box boundary condition testing, as the letter test case would probably never have been generated using boundary condition testing.

4.2 Adequacy of Automatic Grading

To judge how faithful the automatic ranking is to the manual ranking, the mean squared error (MSE) and the root mean squared error ($RMSE$), for n students, are:

$$MSE = \frac{1}{n} * \sum_{i=1}^{n} (g_{manual_i} - g_{auto_i})^2$$
$$RMSE = \sqrt{MSE}$$

[1] computed using the UNIX command
```
diff --suppress-common-lines --speed-large-files
     -y best_old_model new_model | wc -l
```

[2] Very slow, or non-terminating implementations, are automatically removed from consideration by the ranking algorithm implementation

where g_{manual_i} is the manual grade for student i, and g_{auto_i} is the grade computed by the framework.

To find out how useful automatic ranking can be we *minimise* the root mean squared error *RMSE*, finding *optimal* weights p_x. To minimise the error, the weights were varied from 0 to 3, with an increment of 0.25. The weights can be calculated more exactly. However, for a practical application of the automatic grading framework, we considered it beneficial having well-understood weights. The optimal weights for the six exercises, and the resulting errors, are:

	nG	rm	ml	is	mr	mi
p_{fc}	1.0	2.0	2.5	0.75	2.75	2.25
p_{ncss}	0	1.75	0	0.75	1.5	0.5
p_{mc}	0,75	0	0	0	0	0.25
p_{ac}	0.5	0.5	1.0	2.0	0	0.5
MSE	0.41	0.83	1.1	0.61	0.75	0.68
RMSE	0.64	0.91	1.0	0.78	0.86	0.83

The average difference in the manually assigned grades, compared to the automatically assigned grades, ranges from 0.64 (nG) to 1.0 (ml); for a range of grades 0-10. Note that adding additional code metrics, apart from just functional correctness, is beneficial, as the "best" solutions invariably take into account more factors than just functional correctness. On the other hand, functional correctness is vital; there is no exercise which has weight 0 for functional correctness. Moreover a popular criticism against the McCabe cyclomatic complexity metric is that it essentially just measures the number of lines of code, or, as here, the NCSS metric. The results mostly confirm this, as there is just a single exercise (mi) where both p_{ncss} and p_{mc} are nonzero.

The weights used for different exercises vary substantially. In part this is because the manual grading was not consistent during the course, as for instance more emphasis was put on structural complexity code metrics in later parts of the course. Moreover, we can see that the weight for functional correctness is comparatively low (1.0 and 0.75) for the exercises where the students made comparatively few different mistakes (as evidenced by the table showing the number of test cases in the derived test suite), hence probably requiring the grading teacher to emphasise other grading criteria such as algorithmic and structural complexity.

The set of weights which give the lowest errors for all exercises is $p_{fc} = 2.25$, $p_{ac} = 0.5$, $p_{ncss} = 0.5$, $p_{mc} = 0.25$, with *MSE* 0.89 and *RMSE* 0.94. The maximum average error *MSE* is 1.3 for exercise ml; for other exercises the error using the fixed weights is below 1.

5. CONCLUSIONS AND FUTURE WORK

Is the automatic grading framework presented here good enough to replace an experienced teacher's manual grading? The results reported in this article suggest that they are. However, grades should only be a small part of the feedback students receive; more important is to communicate to students *why* their programs are unsatisfactory. In this respect, our framework makes a number of valuable contributions: feedback can be quicker (days instead of weeks), and feedback will include unit tests that the program fails. Detailed feedback can be given on performance, with calculated complexity bounds, and, perhaps more importantly, including a graph relating the performance of the students' program to other implementations of the same API, quickly highlighting bad performance.

Designing test models for functional correctness and algorithmic complexity is, unfortunately, not a common skill. Nevertheless, writing such models, for simple API's, is not so hard. Moreover, unlike manual grading, test model artifacts for automatic grading are highly *reusable*. In our experience exercises usually change little from year to year, and the resulting model changes are often trivial. Moreover, universities could collaborate to develop shared model libraries to support automatic grading for courses on the same topic.

What remains to be done? One missing grading factor vital for a course on algorithms and data structures is memory usage. It should be possible to calculate and illustrate memory usage using the Complexity tool. Unfortunately, at this time we were unable to find a good (open source) solution for calculating the exact memory usage of a Java program.

6. REFERENCES

[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: design and implementation of a cost and termination analyzer for Java bytecode. In *FMCO 2007, Amsterdam, LNCS 5382*, 2007.

[2] T. Arts, J. Hughes, J. Johansson, and U. T. Wiger. Testing telecoms software with QuviQ QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, pages 2–10, Portland, Oregon, USA, 2006.

[3] J. Burnim, S. Juvekar, and K. Sen. WISE: Automated test generation for worst-case complexity. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09. IEEE, 2009.

[4] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. *SIGPLAN Not.*, 35(9):268–279, Sept. 2000.

[5] K. Claessen, J. Hughes, M. Palka, N. Smallbone, and H. Svensson. Ranking programs using black box testing. In *Proceedings of the 5th Workshop on Automation of Software Test*, 2010.

[6] S. H. Edwards. Improving student performance by evaluating how well students test their own programs. *J. Educ. Resour. Comput.*, 3(3), Sept. 2003.

[7] G. Gay, M. Staats, M. W. Whalen, and M. P. E. Heimdahl. The risks of coverage-directed test case generation. *IEEE Trans. Software Eng.*, 2015.

[8] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, 2010.

[9] D. Insa and J. Silva. Semi-automatic assessment of unrestrained Java code: A library, a DSL, and a workbench to assess exams and exercises. In *Proc. of the 2015 ACM Conference on Innovation and Technology in Computer Science Education.*

[10] T. J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320, July 1976.

[11] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07. IEEE, 2007.