

On the Use of Isomorphs to Enhance the Teaching and the Grading Methods in a Data Structures Course

Franc Brglez
NC State University
Raleigh NC, USA
brglez@ncsu.edu

Young J. Pyun
NC State University
Raleigh NC, USA
yjpyun@ncsu.edu

ABSTRACT

This paper presents an approach to automate the preparation, the delivery, and the grading of programming assignment as isomorphs. The approach has been put to a test in a third-year one-semester course on data structures and algorithms for computer science majors, with more than 80 students in the class. Among factors that influenced our approach include (1) the quest for a viable alternative to current methods that rely on the Internet service programs such as Moss in dealing with the perceptions of homework plagiarism, and (2) the premise that deterring plagiarism will always be more effective than detecting it. We report on implementation, features and first-hand experience with a software prototype AutoGradeR and its components, including automated generation of problem isomorphs and a well-defined experimental environment for the students, followed by compiling, running, evaluating, and grading the programming assignments submitted by each student.

1. INTRODUCTION

The idea of problem isomorphs to design and evaluate learning experiments goes back to 1969¹. The first example was number scrabble, an isomorph of tic-tac-toe, soon followed by between a dozen and two dozen isomorphs of the Tower of Hanoi puzzle, most of which were used in one or more trend-setting experiments [2, 3, 4]. Apparently, this is still an active area of research [5, 6, 7].

The isomorphs investigated by the first author of this paper have different context as well as formulation. Our introduction of well-defined instances of isomorphs in the same equivalence class has been motivated for their intrinsic ability to improve the reliability and the statistical significance of experimental performance evaluation of algorithms. In particular, we have demonstrated the merits of isomorphs in a number of very different problem domains, with one feature in common: all are instances of combinatorial problems that may in the worst case require exponential time to complete, .e.g. [8, 9, 10, 11, 12, 13].

¹According to the author himself, on page 382 of [1]: *I think I invented the idea of problem isomorphs about 1969, or a bit earlier Now it is only a small step (at least by hindsight) from the idea that a subject can find a problem easily by finding the right representation and the idea than an experimenter can make a problem harder or easier for a subject by representing it in one guise or another.*

What makes our approach readily implementable and scalable is the ability to generate each isomorph instance automatically from a given *reference instance* – using well-defined rules that are specific to the domain of the problem. Once we recognize the syntax and the structure of the reference problem instance in the context of its domain, we can define the rules to generate any number of instances either in the a class of isomorphs or a variety of other equivalence classes [14].

In this paper, we present an approach that extends the earlier work on the use of isomorphs (for reliable performance testing of algorithms) to preparing and grading a collection of *homework assignment isomorphs* in a third-year one-semester course on data structures and algorithms for computer science majors. The emphasis of this paper is on automating the preparation, the delivery, and the grading of programming assignment as isomorphs. The factors that influenced our approach include:

- Prerequisites that require each student to complete two one-semester courses in Java programming, and one-semester course in discrete mathematics that covers a number of chapters from [15].
- A third-edition textbook [16] with a comprehensive companion Java package of interfaces and classes that implement fundamental data structures and algorithms for immediate use in homework assignments [17].²
- A linux-like computational environment under afs, readily accessible to all students and instructors, on-campus and off-campus [18].
- A web-based course-management system [19], designed to improve and simplify online course development and delivery at N. C. State University, using a web browser as the means through which instructors administer their online content and deliver it to students. Moreover, the same interfaces also allows students to upload the completed assignments as pdf files or as source files that are re-compiled before grading. See Section 4 and Figure 1 for more details.

²The methodology introduced in this paper does not depend on specifics of the programming language. Homework assignments in any programming language (that includes a tested library of components) can be prepared, delivered, and graded as described in the paper.

- A relatively large class size: more than 90 students started the course, with more than 80 continuing to participate throughout the semester.
- The quest for a viable alternative to current approaches like the Internet service programs such as Moss [20] in dealing with the perceptions of homework plagiarism, and also the premise that deterring plagiarism will always be more effective than detecting it [21].
- The opportunity to apply the prior experience in performance evaluation of algorithms to first-time teaching a large group of students: with first author as the instructor and the second author as the teaching assistant.

The paper is organized as follows. Section 2 provides additional context and motivates the approach that is expanded in Section 3. Implementation highlights are covered in Section 4. We evaluate the effectiveness of the approach from different points of view in Section 5 and conclude with Section 6. Additional details about various approaches to generation of isomorphs in this paper are available in [22].

2. MOTIVATION

We introduce two simple illustrative examples: an actual homework assignment, and a quadratic equation. The latter illustrates the process of automated generation of problem isomorphs as (1) homework assignments, and (2) as components for comprehensive in-class tests.

A Homework Assignment. Consider the problem below as part of the homework assignment from the chapter on analysis of algorithms [16].

FindN0. The number of operations executed by algorithms A and B is

$$76 + 110 \cdot n \text{ and } 38 + 2 \cdot n^2 \text{ respectively.}$$

Determine n_0 such that A is better for $n \geq n_0$:
 $n_0 =$

Given the model where an identical homework is assigned to a large class of students, what assurance do we have of each submitted solution represents an authentic effort from each student, in particular in view of the prevailing statistics on student plagiarism [23]? The approach we pursue in this paper supports the premise from [21]:

Deterring plagiarism will always be more effective than detecting it once it occurs and significantly less timeconsuming than pursuing and punishing offenders.

In the case of the homework assignment example, we propose to make individual assignments to each student as isomorphs of the reference problem above. We list examples of the first three isomorphs as assignment definitions and solutions for the equivalence class FindN0, without the companion text, on the three lines below:

$$\text{instance 0: } 76 + 110 \cdot n \text{ and } 38 + 2 \cdot n^2, n_0 = 56$$

$$\text{instance 1: } 135 + 63 \cdot n \text{ and } 68 + 3 \cdot n^2, n_0 = 23$$

$$\text{instance 2: } 42 + 101 \cdot n \text{ and } 21 + 3 \cdot n^2, n_0 = 34$$

We argue that the level of confidence about the authenticity of solutions based on n isomorphs is at least as high, if not significantly higher, than the confidence we may have

when n identical homeworks are assigned to and returned by n students.

The instances of isomorphs as presented for the equivalence class FindN0 are *syntactically equivalent* unlike the isomorphs that were originally defined in a different context [2, 3, 4]. This essential difference in the representation also accounts for very different experimental designs, although both representations have important but complementary roles in the experimental evaluations of the learning processes.

The solution of each isomorph instance in the equivalence class FindN0 requires solving a quadratic equation that has at least one real and positive root. This imposes constraints on the values of the coefficients in formulating each instance (e.g. for instance 2: {42, 101, 21, 3}). The next example illustrates the process of generating an equivalence class of isomorphs for the homework assignment FindN0 and for a series of comprehensive tests in a classroom setting.

Quadratic Equation Isomorphs. Consider a class of n students who are being taught not only an algebra topic such as how to derive roots of a second-order polynomial

$$ax^2 + bx + c = 0, \text{ i.e. } x_{1,2} = (1/2a)(-b \pm \sqrt{b^2 - 4ac})$$

but also how to use a hand-held calculator to solve this equation, i.e. addition, subtraction, division and finding the square root. A typical test, to evaluate the class as a whole on the effective use of the calculator (to solve a quadratic equation), may proceed as follows:

- Select a quadratic equation, say with coefficients $a = 5, b = -49, c = -631$, with solutions, after taking the ceiling of each root value, $\lceil x_1 \rceil = -183, \lceil x_2 \rceil = 429$.
- Administer a classroom test to n students with following instructions:

TestOne. Use the calculator and the formula given in class to find the solutions $x_{1,2}$ of the quadratic equation $ax^2 + bx + c = 0$ with coefficients

$$a = 5, b = -49, c = -631$$

and report the following:

$$\lceil x_1 \rceil =$$

$$\lceil x_2 \rceil =$$

$$\text{solution time} = \quad (\text{in seconds})$$

where the solution time is observed from the wall-clock in class. Return the test to the instructor immediately upon completion (who will also enter the total elapsed time for the test).

- Evaluate test results in two groups: *passed* (correct values reported for both roots) and *failed*; recording the distribution of the solution time for each group. If n is large and the number of tests in the group that reports correct values for both groups is large, say > 32 ,³ the distribution of solution time is most

³We frequently use, in this paper and other related work, the number 32 as the preferred number of independent trial experiments or the number of subjects in an equivalence class. Our preference stems from the observations by statis-

likely near-normal and we have a reliable estimate of the average time it takes for this group of students to solve a quadratic equation using a calculator. However, we have no reliable indicator about the average performance of any student with this test. Is the short solution time by chance? Is the failed score by chance?

In order to reliably evaluate the skill of each student with the calculator (to solve a quadratic equation), we propose a test that relies on an equivalence class \mathcal{A} of $m + 1$ isomorph instances. Instances for this class can be readily generated by adapting the generic rules formally introduced in the next section:

L1: Let coefficients a, b, c define the quadratic equation $ax^2 + bx + c = 0$, and hence the problem category.

L2: Define a reference problem instance 0 by assigning it the reference coefficients valued as $a_0 \stackrel{\text{def}}{=} \{a = 2, b = -10, c = -100\}$

L3: Define invariants for the reference problem instance. Since the objective is to reliably measure the skill of each student with the calculator, we select the number of digits for each coefficient as the invariants: a positive 1-digit integer for a , a negative 2-digit integer for b , and a negative 3-digit integer for c (which also guarantees a two-root real-valued solution of the quadratic equation).

L4: Define the feasible range for each of the selected invariants, e.g.

$$\text{range}(a_0) \stackrel{\text{def}}{=} \{2 \leq a \leq 9, -99 \leq b \leq -10, -999 \leq c \leq -100\}$$

L5: Randomly sample $\text{range}(a_0)$ for its three coefficients until the class size is $m + 1$. For example, the instance $a_i \in \mathcal{A}$ could well be

$$a_i \stackrel{\text{def}}{=} \{a = 5, b = -49, c = -631\}$$

Given the isomorphs from the equivalence class \mathcal{A} , we can devise an alternative to **TestOne** as follows:

TestTwo. Use the calculator and the formula given in class to find the solutions $x_{1,2}$ of the 32 quadratic equations, $ax^2 + bx + c = 0$, using the coefficients in the table:

| inst | a | b | c |
|------|----|-----|------|
| 0 | 2 | -10 | -100 |
| 1 | 5 | -49 | -631 |
| 2 | 3 | -71 | -239 |
| .. | .. | .. | |
| 32 | 6 | -39 | -874 |

Report the results as follows:

| inst | $[x_1]$ | $[x_2]$ | solution time |
|------|---------|---------|---------------|
| 0 | .. | .. | |
| 1 | .. | .. | |
| 2 | .. | .. | |
| .. | .. | .. | |
| 32 | .. | .. | |

ticians: for sample size > 30 the normal distribution and the t-distribution are considered sufficiently close for many cases of practical importance [24].

where the solution time (in seconds) is timed with a suitable stop watch assigned to each student (e.g. a check clock may be particularly well suited). Compared to **TestOne**, a well-executed **TestTwo** provides not only more data to assess the performance of the entire class but also the necessary data to assess the individual performance of each student. Of course, **TestTwo** takes more time to administer and may be justified only for mission-critical testing of student skills and their learning potential.

Generalizations. The approach to generation of isomorphs can be generalized to any category where the problem description has well-defined syntax. The specific rules L1–L5 have a generic counterpart introduced in the next section. Moreover, we also show that the problem invariants and the corresponding feasible sampling range depend on our interpretation of the problem category and the objective of the class assignment, the class test, or the experiment.

3. APPROACH

One of the course objectives was to prepare 10 homework assignment sets that cover the last ten chapters (3–12) from the textbook [16], with increasing emphasis on the use of the complementary Java package [17]. This package contains a collection of Java interfaces and classes that implement fundamental data structures and algorithms, such as sequences, trees, priority queues, search trees, hash tables, sorting algorithms, graphs and their traversals and applications. Starting with a mix of non-programming and programming assignments for the first three sets, the emphasis shifted to strictly programming assignments for the remainder of the course⁴. The experimental environment that was created and given to students as part of each programming assignment is described in Section 4.

The emphasis in this section is to hint at the diversity of isomorphs that were introduced with each assignment set; full details are provided in [22]. The pattern that emerges is summarized with a sequence of few simple generic rules. When specialized to a specific problem, this sequence supports the generation of a well-defined equivalence class of $m + 1$ isomorphs, each with the same syntax:

- R1: Define syntax that describes the domain of the problem;
- R2: Define the reference instance of the problem, using this syntax;
- R3: Define the invariants with respect to the reference instance;
- R4: Define the feasible sampling range within which the invariants must be maintained;
- R5: Define the sampling process in terms of the reference instance and the feasible sampling range; continue sampling until the class size is $m + 1$.

We now outline a subset of reference and isomorph instances, using the actual identifier labels of each assignment set in this course. Since students in the course were

⁴The testing of underlying concepts and theory was committed to six 20-minute quizzes that followed completion of chapters, two mid-term exams, and a final exam. For an overview of the grading process, see Sections 4, 5.

the first to review the early draft of this paper (after their final exam), preserving the original labels of the homeworks is important. For complete list of the assignments, see [22].

Ch06HW: This assignment uses binary tree methods to implement a Java program that parses a file in a 'nested-parentheses-format', e.g. the reference instance

```
B ( G ( U ( C K ) T ) Y ( O ( V ) A ) )
```

The objective is to output values for a set of predefined variables associated with this problem. The output below shows the variable names and the values computed by the program, given the reference instance as defined above.

```
TotalNumberOfNodes = 10
SortedNodeNames = [A, B, C, G, K, O, T, U, V, Y]
SortedExternalNodes = [A, C, K, T, V]
SortedInternalNodes = [B, G, O, U, Y]
RootNodeOnly = B
PreorderTraversal = [B, G, U, C, K, T, Y, O, V, A]
PostorderTraversal = [C, K, U, T, G, V, O, A, Y, B]
InorderTraversal = [C, U, K, G, T, B, V, O, Y, A]
TreeHeight = 3
```

An isomorph of the reference instance is any description where the names of the vertices have been either randomly permuted or renamed, e.g.

```
A ( B ( C ( G K ) T ) Y ( O ( V ) U ) )
```

Ch08HW: This assignment implements two *performance-counting versions*⁵ of an *Unordered Dictionary* in a Java main program: one version uses a list-based dictionary (LBD), and the other uses a hash-table dictionary (HTD). Each version counts and reports the values associated with four variables: 'costFind', 'costFindAll', 'costInsert', 'costRemove'. The Java main program is executed 33 times: once on the reference instance, followed by 32 executions on 32 distinct isomorphs. In order to capture the average-case asymptotic performance of the two algorithm implementations, the process is repeated for a number of sizes of the reference instance: 8, 16, 32, 64, 128, 256, 512. The reference instance of size n consists of two files, an n -line 'dictionary file' and an 'operations file' with the number of lines equal to $2\lfloor(n-1)/2\rfloor + 4$.

In an example shown next, the n lines in the dictionary file contain integer key and text pairs. The first $2\lfloor(n-1)/2\rfloor$ lines in the operations file show the pairs of the remove/insert operations that randomly shuffle the contents of the initial dictionary such that the size of the dictionary

⁵Starting with the homework set Ch07HW (outlined in the Appendix of [22]), all assignments that follow include performance-counting versions of methods and algorithms, using the definitions introduced in the textbook. For example, in Ch07HW the version based on the sorted list data structure (SLPQ), 'costOfInsert' counts each forward advance move to the appropriate position as '1', whereas in the version based on the heap data structure (HeapPQ), 'costOfInsert' counts each swap operation as '1'. Similar considerations apply for Ch08HW, where we count values for variables 'costFind', 'costFindAll', 'costInsert', and 'costRemove' for two implementations: list-based dictionary and hash-table dictionary. For example, if all entries are hashed into different index and there are no duplicates, the total cost will be 2, one for each of the above operations; if different keys hash into same index (a collision) it costs more than one operation to locate the correct index, i.e. there may be cases where the total cost will be more than 2.

remains constant after completion of each pair of operations. The last four lines in the operations file show the four operations 'find', 'findAll', 'insert', and 'remove' that are executed last. In the performance evaluation of each instance, we only record the cost of the last four operations.

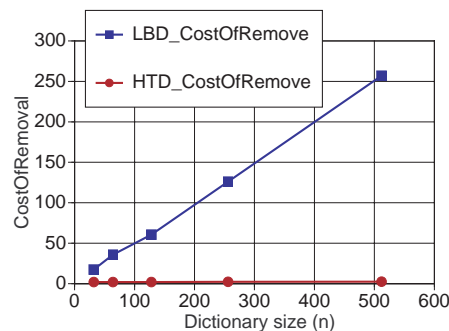
| dictionary 0008.dict | operations 0008.ops |
|----------------------|---------------------------|
| 782 " text 1 " | remove 1312 |
| 1312 " text 2 " | insert 682 " text 1new " |
| 3575 " text 3 " | remove 1702 |
| 977 " text 4 " | insert 3726 " text 2new " |
| 2991 " text 5 " | remove 977 |
| 4073 " text 6 " | insert 389 " text 3new " |
| 538 " text 7 " | find 3726 |
| 1702 " text 8 " | findAll 4073 |
| | insert 682 " text 3new " |
| | remove 3575 |

All key values in the reference instance are randomly selected integers in the range (0, 5000) and depend on the value of the initial seed. A simple algorithm ensures that all entries removed from the dictionary are actually the ones that have been entered earlier. Each isomorph instance is represented with another file pair, syntactically the same as the reference instance above, except for the new choice of randomly assigned key values. Each student is assigned a different initial seed to invoke the pre-assigned experimental environment script which runs the entire experiment for a given size of the reference, captures all intermediate results and concludes with a statistical summary such the one shown below:

| | RefVal | median | mean | stdev |
|----------------|--------|--------|------|-------|
| LBDcostFind | 64.0 | 222 | 209 | 144 |
| HTDcostFind | 2.00 | 2.00 | 2.34 | 0.79 |
| LBDcostFindAll | 512 | 512 | 512 | 0.00 |
| HTDcostFindAll | 3.00 | 2.00 | 2.31 | 0.64 |
| LBDcostInsert | 1.00 | 1.00 | 1.00 | 0.00 |
| HTDcostInsert | 2.00 | 2.00 | 2.34 | 0.70 |
| LBDcostRemove | 193 | 260 | 257 | 161 |
| HTDcostRemove | 2.00 | 2.00 | 2.53 | 1.16 |

**DictionarySize = 512, NumTrials = 32, InitSeed = 1970

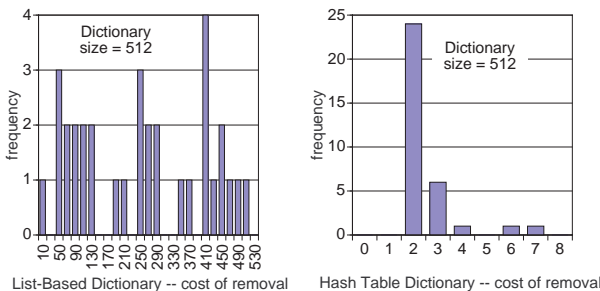
The mean values reported for variables 'LBDcostRemove' and 'HTDcostRemove' of the two algorithm implementations above also contribute the two points in the asymptotic performance graph below (for dictionary size = 512):



Different seeds will induce different reference instances and also different isomorphs, hence the statistical summary will be different from the one above. In order to compare whether

two implementations are equivalent when using different seeds for the same size of the reference instance, we must perform a t-test on the mean values reported by each experiment [24].

Given the statistical summary, asymptotic graphs such as shown above can be generated and analyzed for other variables in the table. Data on which the statistics are based is also available for analysis as histograms. For example, note the contrasts in the performance and the variability of 'LBDcostRemove' and 'HTDcostRemove':



See Section 5 for a more detailed analysis of student submissions of this homework. For additional isomorphs and results of evaluations based on performancecounting versions of different algorithm implementations for the remaining homework assignments, see [22].

4. IMPLEMENTATION

We implemented the approach as described in Section 3 within the computational environment under afs [18], leveraging the web-based interface of WolfWare [19]. Figure 1 shows the relevant portion of the WolfWare schema and the organization of directories and files in the context of this course (CSC316-001). A feature of our implementation is the delivery of Java assignments and the way we process the student submissions that are compiled, executed on isomorphs, and graded with an automated script, making use of the the t-test statistics [24] with respect to the mean values of solutions generated by the 'reference Java assignment'. Each student retrieves the reference Java assignment in the directory `www/wrap/hw/hid/defined/`, here with a basename of 'Prob' as three linked components:

- `Prob_Main_Ref.java`: a main Java reference program source file that defines (1) the 'performance-counting variables' (as defined in Section 3), (2) the formats in which these variables are written to a file, (3) the interfaces to methods that invoke and compute these 'performance-counting variables';
- `Prob_Pack_Ref.class`: a Java program package that implements the methods expected by the main Java reference program, delivered as a class file only;
- `AutoRunSt_hid`: problem-specific script that invokes a series of experimental runs using isomorphs either with the program `Prob_Main_Ref.java` or, in the next phase, by the program `Prob_Main.My.java` implemented by the student.

Thus, after studying the performance of the reference program on a variety of isomorph classes and sizes, the student needs only be concerned with the implementation of data

structures and the methods in the package `Prob_Pack.My.java` that will deliver comparable performance, proceeding as follows:

- Copy `Prob_Main_Ref.java` as `Prob_Main.My.java` and make global substitution of postfixes `*_Ref` with `*_My` throughout the code;
- Implement and compile the package `Prob_Pack.My.java` using the interfaces previously defined for `Prob_Pack_Ref`, but now referenced with the postfix `*_My`.
- Run `AutoRunSt_hid` again, but now to test `Prob_Main.My` which references methods in `Prob_Pack.My`. Consider a comparable range of isomorphs and compare results to ones obtained earlier with `Prob_Main_Ref`.

We conclude this section by describing the logical sequence of six phases of the assignment preparation (by the instructor and the teaching assistant), of student submission process, and the grading process. Only the first and the third phase describe a manual process, the remaining phases describe the automated processes as components of the scripting package `AutoGrader`.

(1) Assignment Definition:

The major part of the problem definition consists of defining its 'performance-counting variables' (as defined in Section 3), the formats in which these variables are written to a file, and the interfaces to methods that invoke and compute these 'performance-counting variables', followed by the implementation of the main Java program `*Main_Ref.java` and its package `*Pack_Ref.java` that implements the methods (and that are not explicitly disclosed to the students). These files, and the homework id-specific configuration file are archived under `private/reference/hid`. This configuration file contains most of the information necessary for the script `AutoAssign` that is invoked in the next phase.

(2) AutoAssign:

| | |
|------|---|
| inp: | variable <code>hid</code> (homework ID) |
| | file from <code>admin/roll</code> (with student IDs, or sids) |
| | file <code>hid.config</code> (configuration file for <code>hid</code>) |
| out: | shared files under <code>www/wrap/hw/defined/</code> , e.g. <code>AutoRunSt_hid</code> , <code>*Main_Ref.java</code> , <code>*Pack_Ref.class</code> |
| | individual files under <code>assigned/hid/sid1/</code> , e.g. <code>readme</code> , <code>data</code> |
| | individual files under <code>assigned/hid/sid2/</code> |
| | individual files under <code>assigned/hid/.../</code> |
| | |
| | instructor-only file under <code>private/reference/hid/</code> configured as <code>AutoRun_hid</code> |

The scripts `AutoRunSt_hid` and `AutoRun_hid` are data-specific configurations of template scripts `AutoRunSt` and `AutoRun`. The script `AutoRunSt_hid` closely relates to the Case 1 of the script that is described for both cases later in this section.

(3) Student Submission:

Upon completion of the assignment (and before the posted deadline), the three files each student is always expected to submit for grading include (under `submitted/hid/`):

`Prob_Main.My.java`, `Prob_Pack.My.java`, and `*Results.*_My.trace` (as generated by the supplied script `AutoRunSt_hid`).

(4) AutoCompile:

| | |
|------|---|
| inp: | student source files under submitted/hid/sid1/ student source files under submitted/hid/sid2/ student source files under submitted/hid/.../ |
| out: | compiled source files under graded/hid/sid1/ compiled source files under graded/hid/sid2/ compiled source files under graded/hid/.../ |

Any submission that does not compile is flagged with an error file which prevents it be invoked for the subsequent run test and receives a score of 0.

(5a) AutoRun_hid (Case 1):

| | |
|------|--|
| inp: | variable ReferenceSize variable ReferenceMorphClass variable InitialSeed variable NumberOfTrials |
| out: | reference results trace file under reference/hid/ e.g. ProbId.Results.*.Ref.trace, where * implies integer labels based on input variable values |

The script for Case 1 is almost identical to the one received by each student as AutoRunSt_hid.

(5b) AutoRun_hid (Case 2):

| | |
|------|---|
| inp: | variable ReferenceSize variable ReferenceMorphClass variable InitialSeed variable NumberOfTrials variable -submitted |
| out: | student results trace file under graded/hid/sid1/ student results trace file under graded/hid/sid2/ student results trace file under graded/hid/.../ ... |

In Case 2, rather than computing new isomorphs for validating each student assignment, the isomorph instances are saved in individual files that are read as needed.

(6) AutoGrade:

| | |
|------|--|
| inp: | file reference/hid/*_Results.*.Ref.trace file graded/hid/sid2/*_Results.*.My.trace file graded/hid/sid2/*_Results.*.My.trace file graded/hid/.../*_Results.*.My.trace |
| out: | file graded/hid/sid1/grade.txt file graded/hid/sid2/grade.txt file graded/hid/.../grade.txt ... |

If the score in the file grade.txt is less than the maximum possible (100 points), the grading script will add an explicit scoring report. An example of such report is shown below:

```
points subtracted from this assignment are due to
score deduction of 10 points
  SLPQ_TotalCost=2358 not-in-range [854.0 1260.0]
score deduction of 20 points
  HeapPQ_CostOfInsert=580 not-in-range [112.6 151.3]
score deduction of 20 points
  HeapPQ_CostOfRemoveMin=1936 not-in-range [277.2 298.7]
score deduction of 20 points
  SLPQ_CostOfInsert=2294 not-in-range [790.0 1196.0]
score deduction of 10 points
  HeapPQ_TotalCost=2516 not-in-range [399.0 440.9]
```

5. EVALUATION

An evaluation to assess the effectiveness the proposed approach without bias can only be written by others after teaching the course a few times. What we offer at the conclusion of the first semester are four brief summaries: (1) a dilemma about resolving the issue of code plagiarizing using the existing Internet service program Moss [32]; (2) an analysis of run-time exceptions induced by a class of isomorphs on a sample homework assignment, some of which may or may not imply code plagiarizing; (3) histograms of homework average grades versus the final weighted average grades of 82 registered students; (4) tabulated responses to a questionnaire completed anonymously by the students after posting the final grades.

Code evaluations with MOSS. Before testing samples of submitted student assignments with MOSS, we created a reference Java program and several versions of programs that intentionally plagiarized the reference program with different levels of deception (so we thought). Surprisingly, MOSS was very good at finding the deceptions, issuing an average similarity score of 95%. We proceeded with actual Java code from the Ch08HW assignment: a pair of submissions that had isolated run-time exceptions during grading with isomorphs (described later) would give a similarity score of 88%, three submissions that all scored 100% under our isomorph-based testing (in this and most other assignments) would report an average similarity score of 50%. These results illustrate the dilemma: after devoting considerable time and effort to run and analyze MOSS results, we still may not be sure how much 'evidence' we would need to confront the student(s) with a case of plagiarism, in particular in our situation, where we give out the main program and mandate common interfaces to all methods for the package that is to be delivered by the students.

Runtime code evaluations with isomorphs. Each student has been assigned a script and an initial random number to generate the trace file of results (and a statistical summary) with the Java main program and the Java package developed by the student – all of which are submitted for grading. The trace file contains a number of run-time environment authentication checks, including the time stamps and location of files being submitted, and a 'signed statement' of conformance with student academic integrity guidelines. The same Java files are re-compiled and expected to run not only with the seed value issued to student but also with any other isomorphs induced by a different seed. Here is an example of a relatively simple coding error not detected by the seed used by the student:

```
*** -> ERROR while executing DictionaryMain_My
commandLine=
java DictionaryMain_My 1970 dictionary_00127.dict
operations_00127.ops
Exception in thread "main" java.lang.NullPointerException
at HashTableDictionary_My.findAll
    (HashTableDictionary_My.java:139)
at DictionaryMain_My.main(DictionaryMain_My.java:70)
*** <- Sun Apr 24 20:46:03 EDT 2005
```

The student made an error in the code that could (but not always) lead to a NullPointerException error due to referencing an array element that may not exist. Another ex-

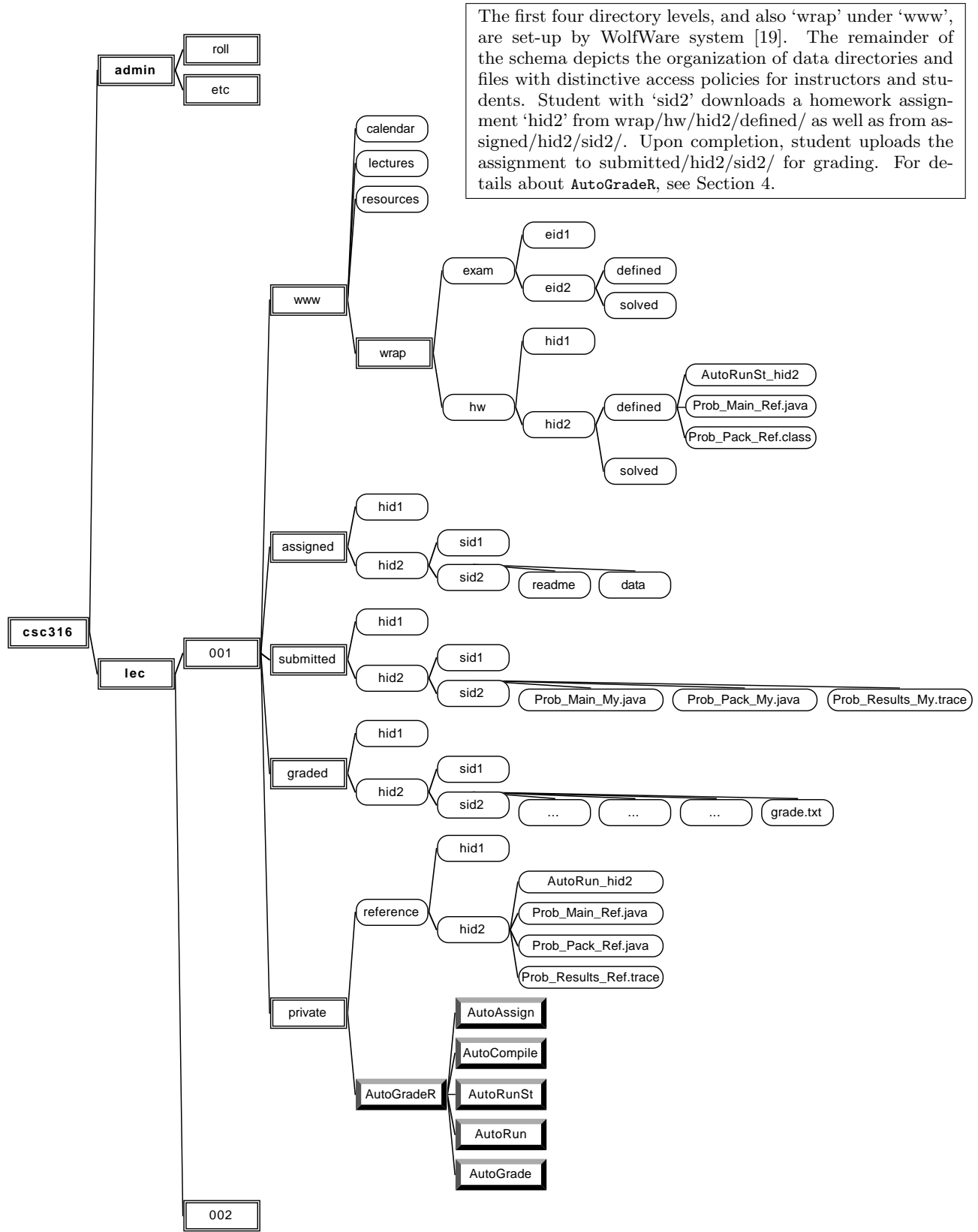


Figure 1: WolfWare schema and its extensions to work areas for instructors and students.

ample not only reveals the coding errors but also similarity between the code of 'student1' and 'student2'. Both assignments would fail the automated script for a small subset of the same isomorph instances. Upon manual inspection, we found the two programs very similar - with both having the same bug which could lead to a NullPointerException error due to the incorrect implementation of the same methods. The offending code fragment is below:

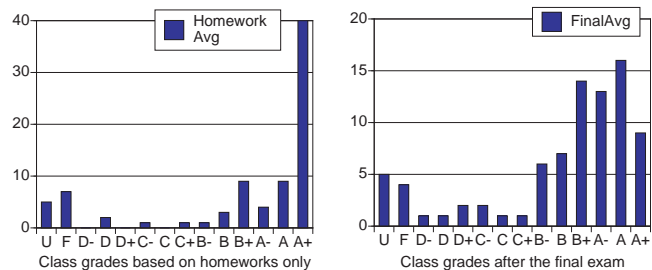
```
public Entry find(Object k) throws InvalidKeyException {
    int e = findEntry(k);
    if (e>0){
        Entry entry = A[e].find(k);
        costFind += A[e].costFind();
        return entry;
    }
    else return null;
}
public Entry remove(Entry e) throws InvalidKeyException {
    Object h = e.key();
    int i = findEntry(h);
    costRemove = costFind;
    if(i>0){
        n--;
        Entry e1 = A[i].remove(e);
        costRemove += A[i].costRemove();
        return e1;
    }
    else return null;
}
```

An operation to remove an element calls 'remove(find(key))' which first finds the key and returns the object to 'remove' which in turn removes the object. The 'findEntry(key)' returns the correct index of the key in the hash table that range from 0 to the size of the hash table. However, according to the code above, 'find(key)' returns null even though 'findEntry(key)' returns 0, which is a valid value and should not be ignored. Since 'find(key)' returns null for a valid element, 'e.key()' in 'remove(e)' flags a NullPointerException error. The same error existed in the source code of both students which in turn caused the same errors on the same data set.

Grade Histograms. The two histograms below depict the homework average grades and the final weighted average grades of 82 registered students. The letter grades A, B, C, D refer to the left side of the intervals valued at 90%, 80%, 70%, and 60% respectively. The letter U stands for 'unassigned grade'. These grade distribution as exactly as archived in the Wolfware system, and are accessible to students (individual grades only), instructors, and administrators.

The distribution of the average homework grades is far from normal, it reflects the style and the expected completion of each homework as described in Section 3. Starting with the fourth homework set, each set presents instances of the solutions which had to be effectively reproduced by writing new code for the assignment. In principle, students could take as much time as needed (subject to a generous submission deadline) to 'get it right'. As shown by the histogram, about 50% of students in the class did just that, hence the peak in the A+ grade category. The final weighted average grade was computed by weighting the 10 homeworks at 50%, 6 quizzes at 10exams at 10% each, and the final exam at 20% the histogram, the grade distribution appears to have

two near-normal clusters, one large one centered near A- and a much smaller one centered near D+. This distribution also tracks to a large extent the level of participation in the class and the class attendance. An interesting issue, raised by some of the students in the anonymous questionnaire (below), was the relative shortage of 'reference solutions' presented during the lectures that could serve as representative problem isomorphs for the questions on quizzes and exams.



Student Questionnaire. The table below summarizes 24 responses to a questionnaire completed anonymously by the students after posting the final grades. Students were also given a web-link to the pdf file of the near-finished draft of the technical report from which this paper is based [22]. The questions were asked exactly as shown in the table, and were prefaced with the following instructions:

When answering questions related to the role of the reference program, consider that (1) it provided you with the range of valid solutions for the problem set and (2) that it allowed you to anticipate the grade for your assignment in advance. Also, when answering the last question, please rank the methodology of the course and not the teacher (you may do this in the 'free format' that follows).

| | score** |
|---|--------------|
| usefulness of net.datastructures (0 = distracting, 4 = very useful) | 3.0/3.25/0.8 |
| overall usefulness of reference program (0 = distracting, 4 = very useful) | 4.0/3.42/0.8 |
| reference program as a learning tool (0 = rather not use, 4 = very useful) | 3.0/2.83/0.9 |
| assignment workload (0 = too much, 4 = just right) | 3.0/2.42/1.1 |
| grading of assignments (0 = unfair, 4 = fair) | 3.5/3.21/0.9 |
| isomorphs as problem sets (0 = unfair, 4 = fair) | 3.0/2.96/0.8 |
| course recommendation to other students (0 = not-at-all, 4 = highly) | 3.0/2.92/1.1 |

** median/mean/standard deviation, based on anonymous e-mail responses from 24 students

The surprising observation from this survey not seen from these statistics: some of the student who scored most of the categories relatively high, also evaluated the 'assignment workload' significantly towards the score of 'too much'.

6. CONCLUSIONS

The notion and representation of isomorphs as introduced in this paper is fundamentally different from the one suggested in [1, 2, 3, 4]. Each isomorph instance in the same equivalence class has the same syntax and the generation of a large number of instances can be automated. The applications to teaching and learning experiences and experiments may not be limited to those described in this paper.

We have no illusion that plagiarizing of homework assignments be eliminated, however we argue that the introduction of isomorphs can certainly make the effort harder, and thereby reduce the frequency of occurrence. Moreover, if plagiarizing does take place, the effort required is much more than a simple cut-and-paste of known solutions, and some learning does take place with solving each of the new isomorph instances.

The methodology proposed in this paper is not limited to teaching the course on data structures. While a number of general principles such as the five steps postulated in Section 3 will always apply, the context and the objective of each course will require some customization. For us, at least one more pass at teaching the course is needed before we can release a version of the `AutoGradeR` environment that can be configured for specific requirements by others.

Acknowledgments. A number of decisions about the structure of this course were discussed and voted on with active participation of the students in the class. This included the upgraded role of homeworks to programming assignments with isomorphs, the frequency of the quizzes (more was better), and the weights that balanced the role of homeworks (50%), quizzes (10%), two midterms (20%), and the final exam (20%).

7. REFERENCES

- [1] H. A. Simon. *Models of my life*. MIT Press, 1996.
- [2] J. R. Hayes and H. A. Simon. Understanding written problem instructions. In L. W. Gregg, editor, *Knowledge and Cognition*. Erlbaum, Hillsdale, NJ, 1974. Reprinted in MOT1, Chap. 7.1.
- [3] H. A. Simon and J. R. Hayes. The understanding process: problem isomorphs. *Cognitive Psychology*, 8:165–190, 1976. Reprinted in MOT1, Chap. 7.2.
- [4] J. R. Hayes. Psychological differences among problem isomorphs. In N. J. Castellan and D. B. Pisoni and G. R. Potts, editor, *Cognitive Theory, Vol. 2*. Erlbaum, Hillsdale, NJ, 1974. Reprinted in MOT1, Chap. 7.3.
- [5] K. Kotovsky, J. R. Hayes, and H. Simon. Why are some problems hard? evidence from tower of hanoi. *Cognitive Psychology*, 17:248–294, 1985.
- [6] G. Gunzelmann and J. R. Anderson. Why are some problems easy? new insights into the tower of hanoi. In *Proceedings of the Twenty-Second Annual Conference of the Cognitive Science Society*, 2000. Mahwah, NJ: Lawrence Erlbaum Associates.
- [7] G. Gunzelmann and J. R. Anderson. An ACT-R model of the evolution of strategy use and problem difficulty. In *Proceedings of the Fourth International Conference on Cognitive Modeling*, pages 109–114, 2001. Mahwah, NJ: Lawrence Erlbaum Associates.
- [8] X. Y. Li, M. F. M. Stallmann, and F. Brglez. Effective bounding techniques for solving unate and binate covering problems. In *Proceedings of the 42th Design Automation Conference*, June 2005.
- [9] F. Brglez and. On SAT instance classes and a method for reliable performance experiments with SAT solvers. *Ann. Math. Artif. Intell.*, 43(1):1–34, 2005.
- [10] F. Brglez, M. Stallmann, X. Y. Li, and B. Militzer. Evolutionary and Alternative Algorithms: Reliable Cost Predictions for Finding Optimal Solutions to the LABS Problem. In R. Duro and M. Grana, editors, *Information Sciences, Special Issue on on Frontiers in Evolutionary Algorithms*. Elsevier Science Publishers, 2005.
- [11] X. Y. Li, M. F. Stallmann, and F. Brglez. QingTing: A Local Search SAT Solver Using an Effective Switching Strategy and an Efficient Unit Propagation. In Enrico Giunchiglia, editor, *Lecture Notes in Artificial Intelligence (LNAI), Special Issue on Satisfiability Testing*. Springer-Verlag, 2003.
- [12] M. Stallmann, F. Brglez, and D. Ghosh. Heuristics, Experimental Subjects, and Treatment Evaluation in Bigraph Crossing Minimization. *Journal on Experimental Algorithmics*, 2001.
- [13] J. E. Harlow and F. Brglez. Design of Experiments and Evaluation of BDD Ordering Heuristics. In R. Drechsler and D. Sieling, editors, *Software Tools for Technology Transfer (STTT), Special Issue on BDDs*. Springer-Verlag, 2001.
- [14] D. Ghosh. *Generation of Tightly Controlled Equivalence Classes for Experimental Design of Heuristics for Graph-Based NP-hard Problems*. PhD thesis, Electrical and Computer Engineering, North Carolina State University, Raleigh, N.C., May 2000.
- [15] K. H. Rosen. *Discrete Mathematics and Its Applications, Fifth Edition*. Mc Graw Hill Companies, Inc., 2003.
- [16] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java, 3rd Edition*. John Wiley & Sons, Inc., 2004.
- [17] M. T. Goodrich and R. Tamassia. Home of net.datastructures, 2005. A textbook companion, net.datastructures is a Java package containing a collection of Java interfaces and classes. Accessible from <http://net3.datastructures.net/>.
- [18] Home of Eos Computing Environment, 2005. See <http://www.eos.ncsu.edu/>.
- [19] Home of WolfWare Web-based Course Management System, 2005. A system for automatically creating and configuring course lockers at N. C. State University, with web tools for locker administration and delivery of content. See <http://wolfware.ncsu.edu/guide/guide.html>.
- [20] A. Aiken. Home of Moss: A system for detecting software plagiarism, 2005. Moss (for a Measure Of Software Similarity) is an automatic system for determining the similarity of C, C++, Java, Pascal, Ada, ML, Lisp, or Scheme programs. A submission script is accessible from <http://www.cs.berkeley.edu/~aiken/moss.html>.
- [21] Detering, detecting and dealing with student plagiarism, 2005. From the Joint Information Systems Committee (JISC) home page. For details, see <http://www.jisc.ac.uk/uploaded documents/JISCBP-Plagiarism-v1-final.pdf>.
- [22] F. Brglez and Y. J. Pyun. On the Use of Isomorphs to Enhance the Teaching and the Grading Methods in a Data Structures Course, 2005. A Technical Report, from <http://jupiter3.csc.ncsu.edu/~brglez/www/courses/-CSC316-001-2005-Spring/wrap/>.
- [23] M. Freewood, R. MacDonald, and P. Ashworth. Why simply policing plagiarism is not the answer. In Proceedings of the 10th Improving Student Learning Symposium, Oxford Centre for Staff and Learning Development, 2002.
- [24] R. L. Mason, R. F. Gunst, and J. L. Hess. *Statistical Design and Analysis of Experiments, With Applications to Engineering and Science, Second Edition*. Wiley-Interscience, Inc., 2003.