# A Data-Driven Analysis of Informatively Hard Concepts in Introductory Programming

R. Paul Wiegand
Institute for Simulation & Training
University of Central Florida
Orlando, FL, USA
wiegand@ist.ucf.edu

Anthony Bucci
119 Amory St.
Cambridge, MA, USA
anthony@bucci.onl

Amruth N. Kumar
Ramapo College of New Jersey
Mahwah, NJ, USA
amruth@ramapo.edu

Jennifer L. Albert
The Citadel
171 Moultrie Street
Charleston, SC, USA
jalbert@citadel.edu

Alessio Gaspar
University of South Florida
4202 E. Fowler Avenue
Tampa, FL, USA
alessio@usf.edu

## ABSTRACT

What are the concepts in introductory programming that are easy/hard for students? We propose to use Dimension Extraction algorithm (DECA) inspired by coevolution and co-optimization theory to answer this question. We propose and use the metrics of *informatively easy/hard* concepts to identify programming concepts that are solved correctly by the most "dominated student" versus solved incorrectly by the most "dominant student". As a proof of concept, we applied DECA to analyze the data collected by software tutors called problets used by introductory programming students in Spring 2014. We present the results, i.e., informatively easy/hard concepts on a dozen different topics covered in a typical introductory programming course. It is hoped that these results will inform programming instructors on the concepts they should (de)/emphasize in class. They will also contribute towards creating a concept inventory for introductory programming.

## CCS Concepts

**•Social and professional topics → Computer science education; •Applied computing → Education;**

## Keywords

introductory programming education; dimension extraction; multiobjective optimization, performance analysis; problets

## 1. INTRODUCTION

Attrition in the introductory programming course is legendary. Students find programming to be hard. What are the concepts in introductory programming that are hard/easy for students? Knowing this would help educators better target the hard concepts, provide focused support to students, and possibly improve retention in the course.

Typically, a study of hard programming concepts would be conducted in the context of creating a concept inventory. However, no formal concept inventory exists for programming concepts. Delphi process has been proposed as one mechanism for identifying hard concepts [6]. While this process relies on the expertise and consensus of educators, the results still need to be empirically validated.

In this paper, we propose a data-driven approach for identifying easy and hard programming concepts. In addition to easy and hard concepts as traditionally understood, we propose the notions of *informatively easy* and *informatively hard* concepts. As proof of concept, we apply our approach to data collected by software tutors for programming, called problets (problets.org) in Spring 2014. We present the results and discuss their significance.

The data-driven approach we propose is based on *dimension extraction coevolutionary algorithm* (DECA) and inspired by coevolution and co-optimization theory [2]. The algorithm identifies structural relationships amongst students and problems by constructing a geometry of problems and students that illustrates how student performance can be distinguished in fundamentally different ways. While the algorithm has been used in technical applications such as analyzing game playing strategies, it has seen very little application in educational domains such as identifying difficult programming concepts in computer science education.

*SIGCSE '16, March 02 - 05, 2016, Memphis, TN, USA*
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-3685-7/16/03...$15.00
DOI: http://dx.doi.org/10.1145/2839509.2844629

## 2. BACKGROUND

### 2.1 Hard Concepts in Programming

Defining and measuring the hardness of concepts is important for scaffolding learning according to Vygotsky's Zone of Proximal Development (ZPD) theory [15]. One of very few attempts to identify hard concepts in programming [6] recently identified a list of topics using Delphi process and ranked them according to importance and difficulty (see Table 1 for the subset used in this study). A second study [7] surveyed instructors of CS1 and CS2 on how much time they spent on various topics and correlated that importance with student performance. However, neither study delved into individual concepts within each topic.

Usually, researchers enumerate concepts and assess their level of hardness when creating a concept inventory on a subject. In Computer Science, concept inventories have been attempted on various subjects (e.g., digital logic, operating systems, algorithms) [13] but none has been completed on introductory programming. Two recent attempts have both been preliminary, one identifying topics [6] and another identifying some high-level concepts [14]. To create a concept inventory for introductory programming, assessment items must be created on a continuum of specific concepts within each topic that students do not understand. We address this issue of granularity [12] by proposing to identify low-level concepts on most of the topics typically covered in introductory programming, particularly those concepts whose hardness can be measured using data-driven analysis.

## 2.2 Co-optimization and Coevolution

The challenge of defining hard and easy problems arises in the study of *co-optimization* problems and *coevolutionary* algorithms [11]. Co-optimization problems are distinguished from optimization problems by the presence of two or more types of entity (e.g., student or problem) that might vary and can interact with one another with a measurable outcome or score. In the present context we have computer science students and a variety of computer programming problems we might present to those students; any student can attempt–interact with–any of the problems, and they receive a score. Therefore, the real-world domain of students solving problems resembles a co-optimization problem, and we believe the tools and techniques developed in the study of those algorithms can be fruitfully applied to this domain.

One foundational line of work in co-optimization and coevolutionary algorithms concerns *dimension extraction* methods [3, 4, 2, 5]. Abstractly, this work hypothesizes that the information gleaned from interactions, for instance the scores of students solving problems, can be decomposed into a vector-space-like *coordinate system*. Within a coordinate system, there are potentially *multiple axes* or *dimensions*, consisting of a linearly ordered subset of the set of all entities. Entities further along a given axis are "no worse than" those preceding it. Across two different axes, entities are incomparable to one another, in the sense that an entity on one axis will be better in some ways, but worse in other ways, than an entity on another axis. This method has been used for a number of purposes, including automatically identifying key conceptual tactics in the game of Nim [5].

We will apply a minor variant of the dimension-extraction algorithm presented in [4] to analyze data about populations of students. A "student axis" will be a subset of the students, which might be thought of as a "learner type". Students further along the axis they are in tend to be better on the collection of problems they attempted than the students preceding them on the axis. This analysis will help us distinguish student performance, a key insight of this paper.

## 3. METHODS

## 3.1 Problets

The software tutors used in this study, problets (problets.org), cover topics typically studied in introductory programming courses. Each problet addresses one topic, but covers several concepts (see Table 1). The concepts address different skills: evaluating expressions, tracing programs (e.g., predicting the output), debugging programs and identifying the state of variables in a program. An advantage of using data collected by problets is that problets have been used by third-party educators in their introductory courses for over a decade and have been continually and extensively evaluated (e.g., [9, 10]).

Problets are adaptive, i.e., they tailor the sequence of problems to the learning needs of the student. Initially, each problet presents the student with one test problem per concept. Thereafter, it only presents practice problems on concepts on which the student incorrectly solved the test problem. Since the purpose of this study was to identify easy/hard concepts, we used only the data from the initial test administered by each problet — this data represented a snapshot of what the students knew/did not know before using the problet but after attending one or more lectures on the topic in class.

Table (1) lists the topics on which problets are available, and the number of concepts covered by each problet. It lists the number of students who solved all the test problems in each problet, and their schools in Spring 2014. Participants were students of introductory programming from high schools, community colleges and four-year colleges, who usually used the problets for after-class assignments after the topic had been covered in class.

| **Topic** | **Concepts** | **Schools** | **Students** |
|---|---|---|---|
| Arithmetic | 25 | 29 | 982 |
| Relational | 24 | 20 | 582 |
| Logical | 21 | 16 | 572 |
| Assignment | 19 | 15 | 599 |
| Selection | 12 | 21 | 774 |
| Switch | 12 | 12 | 221 |
| while | 9 | 18 | 523 |
| for | 10 | 18 | 700 |
| do-while | 15 | 13 | 175 |
| Advanced Loops | 13 | 8 | 161 |
| Functions/Bugs | 9 | 7 | 282 |
| Functions/Tracing | 10 | 8 | 449 |
| Array | 14 | 14 | 221 |
| Class | 18 | 4 | 125 |

**Table 1: Concepts per topic and number of schools/students who used each problet in Spring 2014**

## 3.2 Dimension Extraction

Dimension Extraction Coevolutionary Algorithm (DECA) [4] is a coevolutionary algorithm that uses a dimension extraction method, as outlined in Sect. 2.2, at its core. While the coevolutionary algorithm searches through *candidate solutions* and *tests*, for our purposes here we will be using its internal dimension-extraction algorithm to analyze students and problems.

In a *problem analysis*, the students are the candidates and the problems are the tests. The result is a coordinate system of problems, where each dimension orders a subset of the problems. Problems further along a dimension are not passed by at least the same students than those lower on the dimension. Problems on different dimensions are those on which students performed incomparably. Intuitively, each dimension corresponds to a different concept that differentiated student performance and the problems that are highest on each dimension are the problems the most students did not solve.

Dually, in a *student analysis*, the problems are the candidates and the students are the tests. The result is a coordinate system of students this time, where each dimension orders a subset of students in terms of the number of problems they solved correctly. Students on different dimensions perform incomparably on the tests. In this sense, each dimension corresponds to a different pattern in which students performed on problems or intuitively speaking a "type" of learner. The students furthest along an axis solve the most problems, and in that sense are the best-performing representatives of each type.

Application of these analyses to the performance of students on problets potentially provides several sources of insight. First, the number of dimensions extracted from problem analysis gives an implicit measure of the number of distinct *concepts* that appear in the problem set; that is, the number of different ways that problems distinguish student performance. Likewise, the number of dimensions extracted from student analysis gives an implicit measure of the number of distinct *types of learner* that appear in the student set, or in other words the number of different ways students can be grouped in terms of differences in performance.

Additionally, such analyses allow us to highlight the importance of how informative different problems are. It is simplistic to assign a "difficulty" measure to problems based purely on how many students solve or do not solve a problem. For instance, if Garry Kasparov were to play a classroom of intermediate chess students, he would likely beat them all, and it would be impossible to determine which students were among the best in the class. Likewise, if someone who never played chess was used as the opponent, this person would likely lose to everyone, and we would be equally uninformed about the relative performance of students. Similarly, a problem that most students miss (or get right) may not be an informative problem because it might not provide distinctions to help educators know where students are having problems.

Instead, we want challenging problems that distinguish student performance as much as possible. As educators, we should be interested in constructing problems that lie in a *zone of proximal development* [15] — problems of varying levels of difficulty that are on the boundary of where students perform in fundamentally different ways and, consequently, where true performance can best be judged. To begin to address this shortfall, we introduce two new terms grounded in our dimension extraction analysis, *informatively hard* and *informatively easy.*

For each dimension extracted in a student analysis, i.e. type of learner, we identify the *most dominant student* as the one who solved the most problems among students on that dimension. We count each problem not passed by such students, for all of the most dominant students. Likewise, we identify the *most dominated student* as the one who solved the fewest problems among students on their dimension. We count each problem solved correctly by such students, for all of the most dominated students. Then, a problem is **informatively hard** if it is most often solved incorrectly by the most *dominant* students. Similarly, a problem is **informatively easy** if it is most often solved correctly by the most *dominated* students.

These two measures do not coincide with their traditional counterparts. For example, it is possible for a problem to be missed by everyone *but* the strongest students, for an otherwise challenging problem to be solved by the weakest students. When such things happen, it is likely that student behavior is confounding a linear expectation of performance, and is therefore interesting. Measures such as informatively easy and hard highlight and preserve the idea that different students understand different concepts in different ways. An informatively easy concept may be understood in the context of Zone of Proximal Development as the concept nearest to some other concept, whereas an informatively hard concept is farthest from any other concept.

By contrast, we define a problem as (traditionally) **hard** if it is solved incorrectly by most students and as **easy** if it is solved correctly by most students.

As we will see in the next section, rarely is an informatively easy concept also easy, or an informatively hard concept also hard, although our intuitive notions of monotonicity of hardness would dictate this. Recall that informatively hard concepts are determined by the most dominant students vis-a-vis the general population, and informatively easy problems are determined by the most dominated students. The more interesting, although rare, cases are when an informatively easy concept is hard and an informatively hard concept is easy. The former corresponds to when it is easy for the general population to make mistakes while solving a problem. The latter corresponds to when it is easy for the general population to guess the answer.

## 3.3 Consistency with the Data

We will begin to demonstrate the utility of our approach in the next section; however, an important preliminary question is whether or not our method for identifying informatively hard and informatively easy remains relatively consistent across a given data set. Our discussion below describes our analysis for identifying informatively hard problems; however, the same analysis was also conducted for informatively easy problems.

To address the question of consistency, we performed the following experiment. For each topic, we split the data so that students were assigned to one of two groups independently and uniformly at random. We extracted dimensions on both groups and used the results to identify informatively hard problems. Since there can be multiple problems that are equally informatively hard, when the two extractions produced an overlap in the set of identified problems, we considered that split to have been "*consistent*". This was repeated 50 times, where each splitting was independent.

Traditional proportions or goodness of fit tests are not possible in this case because the algorithm can return multiple results for a given split. Instead, we first computed per-trial probabilities combinatorially, noting that given that one split produces $k$ informatively hard problems while another produces $j$ informatively hard problems over $n$ such problems, the probability of random subsets overlapping is given by: $1 - \frac{(n-k)!\,(n-j)!}{(n-k-j)!\,n!}$.

For each topic, we averaged these probabilities across the 50 trials, which were then used to compute theoretical values for how many trials should have produced matches in the 50 trials, assuming a random process. After that, we used a traditional one-tailed difference of two proportions statistical test. The results were significant ($\alpha = 0.05$) for both informatively hard and informatively easy calculations, for all but two topics. The *Functions/Bugs* topic did not produce significant results, and the *while* topic results were significant for only informatively easy calculations. Overall, this suggests that the method is fairly consistent; however, when many potential informative cases are identified, consistency issues can arise. However, in such scenarios, the algorithm is not necessarily reporting random information. Since many of the problems are equally discriminating, the specific data set being analyzed simply becomes the biggest factor in identifying informative problems.

## 4. RESULTS: EASY AND HARD CONCEPTS

Students used problets in sufficient numbers on 13 topics in Spring 2014 for our analysis to be meaningful, as listed in table 1 (We combined the results of the two functions tutors). For each of these topics, we list below the informatively easy and informatively hard concepts as determined by the Dimension Extraction algorithm, along with easy and hard concepts as traditionally understood.

**Arithmetic Expressions:**
**Easy:** Evaluating expressions such as $8 - -5$, which involve subtracting a negative number; • evaluating fully parenthesized expressions, such as $(14/((6-3)+2))$, where the order of evaluation of operators is fully dictated by parentheses.
**Hard:** Evaluating expressions such as $12\%5 + 5\%12.0$, containing remainder operator along with real operands.
Fully parenthesized expressions can be evaluated without knowledge of precedence and associativity rules. Remainder operator and data types are both novel concepts first taught in programming classes.
**Informatively Easy:** Evaluating expressions involving divide-by-zero error: e.g., $9/3/0$.
**Informatively Hard:** Evaluating expressions featuring integer division: e.g., $5/3 + 3 * 5$
Integer division is also a novel concept, specific to programming. Dividing by zero is informatively easy since it is emphasized as a problem in math.

**Relational Expressions:**
**Easy:** Evaluating expressions where the two operands are the same: e.g., $5 >= 5$
**Hard:** Evaluating expressions featuring chained relational operators: e.g., $3 > 7 >= -1$
Chaining of relational operators results in errors in Java, and yields surprising results in C++. Both are a clear departure from math, and are novel to programming.
**Informatively Easy:** Evaluating expressions that involve both arithmetic and relational operators, both being at the lower level of precedence in their respective groups (e.g., + in arithmetic, != in relational), such as $3 - -5 \quad != \quad 3 + 5$
**Informatively Hard:** Evaluating expressions containing both arithmetic and relational operators, the two being at different levels of precedence at their respective groups, e.g., $-3 * -4 \quad != \quad 6 * 2$
Expressions involving both arithmetic and relational operators, but no parentheses, seem to be both informatively easy and hard at the same time. The difference could be that students already understand the concept of subtracting a negative number, but not so much multiplying two negative numbers.

**Assignment Expressions:**
**Easy:** Evaluating pre-fix assignments: e.g., `5 + ++var1`
**Hard:** Evaluating pre- and post-fix assignments: e.g., `var2 = ++var1` and `var2 = var1++`
Both pre- and post-fix assignment operators are inherently confusing to students. That prefix increment also features in the easy expression leads us to speculate that students guess the answer to this problem more often than not.
**Informatively Easy:** Evaluating coercion during assignment: e.g., `double var1 = 3 + 5`
**Informatively Hard:** Evaluating postfix decrement operator: e.g., `var2 = var1--`)
By the time students learn assignment expressions, it is likely they are already familiar with the concept of coercion.

**Logical Expressions:**
**Easy:** Evaluating fully parenthesized expressions, e.g., `(true && (true && (true || false)))` that do not involve short-circuit evaluation or need for precedence rules; and • expressions evaluated left to right with easy evaluation semantics: e.g., `true && false && true`.
**Hard:** Evaluating C++ expressions wherein numerical values are used as boolean operands: e.g., $3 * 0 \&\& 3 + 0$
Again, these support the hypothesis that difficulty is linked to novel concepts, specific to programming.
**Informatively Easy:** The same as the fully parenthesized easy expression described before.
Informatively Hard: The same as the non-parenthesized easy expression described before — even though this is easy, the best students still make mistakes when evaluating it.

**Selection Statements:**
**Easy:** Identifying the output of nested `if` statements.
**Hard:** Identifying the output of an `if-else` statement with a condition that evaluates to true.
The hard problem was designed to be the simplest on `if-else` statements. Lack of familiarity with the user interface may explain why students solved it incorrectly in larger numbers — selection being the first tutor and this problem being the first problem in the tutor. It is surprising that students found nested `if` statements to be easy, suggesting that the bifurcation of control flow is what is complicated about selection statements, not conditional execution.
**Informatively Easy:** Predicting the output of multiple back-to-back `if` statements.
**Informatively Hard:** Tracing the output of an `if-else` statement whose condition evaluates to false; • an `if` statement whose condition evaluates to true; • nested `if` statements.
Once again, user interface may be to blame for why students found the first two concepts informatively hard.

**`switch` Statement:**
**Easy:** Tracing the output of a `switch` statement whose condition matches no case; • a `switch` statement in which the body of a `case` statement is empty.
**Hard:** Identifying the output of a `switch` statement with a missing `break` statement.
Both easy concepts entail code not generating any output, and may be traced with only a cursory glance.
**Informatively Easy:** Specifying the output of nested `switch` statements; • identifying as syntax error when the `case` value is not an integer expression.
**Informatively Hard:** Predicting the output of multiple back-to-back `switch` statements.
Both informatively easy concepts come as a surprise, since

the correct answer to neither is easily guessable, and cursory glance is not sufficient to arrive at the correct answer. This is an example where data-driven analysis contradicts intuition.

**`while` Loop:**

**Easy:** Predicting the output of a loop that iterates only once; • a loop that never iterates because its condition evaluates to false on the first try.

**Hard:** Identifying the output of multiple back-to-back loops, wherein the behavior of the second loop depends on the execution of the first loop.

The two easy concepts require students to trace not so much the repeated execution of the loop as the independent one-time evaluation of the condition and the loop body.

**Informatively Easy:** Identifying the output of nested loops, wherein execution of the inside loop is independent of that of the outside loop.

**Informatively Hard:** Predicting the output of • nested loops, with the inner loop's behavior dependent on that of the outer loop; • loop that never iterates because its condition fails immediately; • loop whose condition variable is updated before it is printed inside the loop; • loop with sentinel value changing on each iteration; • code that appears and is executed after the loop exits.

Informatively hard concepts involve behavior that deviates from that of a simple loop. The informatively easy concept of independent nested loops might suggest that nesting is not so much a problem as unexpected behavior of code.

**`for` Loop:**

**Easy:** Predicting the output of a zero-iteration loop, which results in no output for the code.

**Hard:** Tracing the output of two back-to-back loops, where second loop's counter depends on the first loop.

**Informatively Easy:** Identifying the output of a loop which iterates exactly once.

**Informatively Hard:** Identifying the output of nested dependent loops, wherein the inner loop's counter depends on the outer loop's execution.

All the above findings concur with those on `while` loop.

**`do-while` Loop:**

**Easy:** Specifying the output of a loop that iterates only once.

**Hard:** Identifying the output of a C++ loop whose condition is an assignment statement.

The empirically hard problem has been a long-known source of error for novice programmers [8].

**Informatively Easy:** Tracing the output of a loop whose condition is a disjunctive logical expression.

**Informatively Hard:** Predicting the output of a loop whose condition is a conjunctive logical expression.

Presumably, disjunctive expressions such as `status == 's' || status == 'f')` are easier to evaluate than conjunctive expressions such as `limit >= 0 && limit <= 100`.

**Advanced loop concepts:**

**Easy:** Identifying that modifying the value against which the condition variable is compared results in an infinite loop..

**Hard:** Predicting the output of a loop containing `continue` statement.

Surprisingly, identifying an infinite loop turned out to be easy, possibly because of the emphasis placed on it in class.

**Informatively Easy:** Predicting the output of a loop containing `continue` statement.

**Informatively Hard:** Identifying the bug when a `break` statement appears outside any loop or `switch` statement.

The informatively easy concept is also hard. This suggests that it is easy to make mistakes when tracing loops with `continue` statement, even when one understands the underlying concepts. The informatively hard concept corresponds to a pathological use of the language.

**Functions:**

**Easy:** Tracing the output when a function is called as part of an expression; • identifying as a bug when the caller tries to use the value returned by a function with `void` return type.

**Hard:** Predicting the output when a variable is passed by value to a function; Identifying as a bug when • return statement is missing in the definition of a non-`void` function; • data type of the value in the return statement is incompatible with the return type of the function.

In accord with our previous findings, easy concepts correspond to scenarios where functions are treated like math functions, and hard concepts deal with novel programming-specific constructs. Please note that parameter passing by reference was not included in problets.

**Informatively Easy:** Tracing the behavior of a function that has multiple conditional return statements; • Identifying bugs due to type incompatibility between formal and actual parameters; • Identifying bugs due to type incompatibility between returned value and return type of a function.

**Informatively Hard:** *Not* identifying as a bug when two variables with the same name appear in two different functions; • Identifying the C++ bug wherein a function is called before it is defined or prototyped.

Informatively hard concepts deal with advanced concepts of scope and extent . Informatively easy concepts should both be familiar to students by the time functions are introduced.

**Arrays:**

**Easy:** Specifying the contents of a fully initialized array; • predicting the behavior of referencing a random element.

**Hard:** Identifying the contents of an array declared with incomplete initialization.

The hard concept is among the pathological cases often skipped in lectures, but must be known to students nevertheless. Problets are designed to cover these concepts.

**Informatively Easy:** Predicting the behavior when an element of an array is referenced before it is initialized.

**Informatively Hard:** Identifying type mismatch when an array is passed as parameter to a function.

Although students are familiar with parameter passing and type mismatch by now, applying these concepts to arrays turns out to still challenge them.

**Access in classes:**

**Easy:** Identifying the behavior when a `public` object member is accessed by a member function of the class.

**Hard:** Tracing the behavior of the default constructor when an object is created.

Constructors are hard because they are the first example of call-back functions that students study. (While `main` is also a call-back function, it is usually not introduced as one.)

**Informatively Easy:** Identifying mismatching numbers of actual / formal parameters when calling member functions.

**Informatively Hard:** Tracing the behavior of constructors, with or without parameters, when an object is created.

While students have had additional opportunities to learn about parameter-passing, call-back constructors are novel.

## 5. DISCUSSION

Informatively easy or hard concepts overlapped with traditional easy or hard concepts only occasionally. This supports our claim that the use of DECA to identify them have the potential to bring new insights to educators.

Our results show that the novelty of a concept significantly influences whether the concept is easy or hard, e.g., integer division and remainder operator are novel and therefore, hard. This is consistent with previous research [7]. As a corollary, what is informatively easy or hard changes throughout a semester, as exposure to and familiarity with specific concepts increases, e.g., data type compatibility. Another recurring theme is that students found it easier when they could relate programming concepts to math concepts they already knew. So, instructors may want to highlight the relationship between concepts in programming and math.

We found that some concepts that are intuitively thought to be hard turned out to be easy (e.g., independent nested `while` loops) and vice versa (e.g., conjunctive logical condition in a `do-while` loop). Clearly, there is a need for data-driven analysis to confirm or refute intuitive notions in Computer Science education and inform educators.

We expect the easy, hard, informatively easy, and informatively hard concepts enumerated in the previous section to contribute to a concept inventory for introductory programming. Since these concepts are supported by data-driven analysis, their inclusion in the concept inventory will better reflect the needs of students.

Our results apply to C++, Java and C#, although some concepts identified as easy/hard are specific to C++. The concepts we considered relate to code analysis (program tracing, debugging and expression evaluation), but not synthesis (code design, writing) or evaluation (e.g., refactoring) skills in Bloom's taxonomy [1].

A confounding factor of this study is that the set of concepts identified for each topic were meant to be representative but not exhaustive or exclusive. Still, the approach we used for data analysis would be applicable to an exhaustive set of concepts, should one be collected; and the results of this study, while not a concept inventory in its entirety, will contribute to one when it is created.

In future, we will also examine the consistency of the dominance relationships produced by the algorithm. Additionally, we will use DECA to identify concepts and redundancies in the problem data. Further, we will examine how the structure uncovered by these methods might relate to "learning trajectories". Overall, this tool has great potential to help instructors design informative problem sets that will better elucidate student performance differences.

## 6. ACKNOWLEDGMENTS

This material is based in part upon work supported by the National Science Foundation under awards #1504634, #1502564, and #1504634.

## 7. REFERENCES

[1] L.W. Anderson, D.R. Krathwohl, P.W. Airasian, K.A. Cruikshank, R.E Mayer, P.R Pintrich, J. Raths, and M.C. Wittrock. *A taxonomy for learning, teaching, and assessing: A revision of BloomâĂŹs Taxonomy of Educational Objectives.* New York: Longman, 2001.

[2] A. Bucci. *Emergent Geometric Organization and Informative Dimensions in Coevolutionary Algorithms.* PhD thesis, Brandeis University, Boston, MA, 2007.

[3] A. Bucci, J.B. Pollack, and E.D. de Jong. Automated extraction of problem structure. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, GECCO '04, pages 501–512, 2004.

[4] E.D. de Jong and A. Bucci. DECA: Dimension extracting coevolutionary algorithm. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, GECCO '06, pages 313–320, New York, NY, USA, 2006. ACM.

[5] E.D. de Jong and A. Bucci. Objective set compression: Test-based problems and multi-objective optimization. In *Multi-Objective Problem Solving from Nature: From Concepts to Applications.* Springer, 2008.

[6] K. Goldman, P. Gross, C. Heeren, G. Herman, L. Kaczmarczyk, M.C. Loui, and C. Zilles. Identifying important and difficult concepts in introductory computing courses using a delphi process. *ACM SIGCSE Bulletin*, 40(1):256–260, 2008.

[7] M. Hertz and S.M. Ford. Investigating factors of student learning in introductory courses. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 195–200, New York, NY, USA, 2013. ACM.

[8] A. Koenig. *C Traps and Pitfalls.* Addison-Wesley Professional, 1989.

[9] A.N. Kumar. The effectiveness of visualization for learning expression evaluation. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 362–367, New York, NY, USA, 2015. ACM.

[10] A.N. Kumar. Solving code-tracing problems and its effect on code-writing skills pertaining to program semantics. In *Proceedings of the 20th Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE 2015, Vilnius, Lithuania, July 6-8, 2015*, pages 314–319, 2015.

[11] E. Popovici, A. Bucci, R.P. Wiegand, and E.D. de Jong. Coevolutionary principles. In *Handbook of Natural Computing*, pages 987–1033. Springer, 2012.

[12] L. Porter, C. Taylor, and K.C. Webb. Leveraging open source principles for flexible concept inventory development. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*, pages 243–248. ACM, 2014.

[13] C. Taylor, D. Zingaro, L. Porter, K.C. Webb, C.B. Lee, and M. Clancy. Computer science concept inventories: past and future. *Computer Science Education*, 24(4):253–276, 2014.

[14] A.E. Tew and M. Guzdial. Developing a validated assessment of fundamental cs1 concepts. In *Proceedings of the 41st ACM technical symposium on Computer science education*, pages 97–101. ACM, 2010.

[15] L.S. Vygotski. *The Collected Works of LS Vygotsky.* Springer, 1987.