Matthew Effects in Learning Computer Programming Concepts

Candido Cabo, Ph.D.
Department of Computer Systems
New York City College of Technology
City University of New York
ccabo@citytech.cuny.edu

Abstract - In this research to practice full paper we quantified whether student progress in learning computer programming concepts in a Java course is consistent with the Matthew effect, that is, if early success (or failure) in the acquisition of concepts/skills begets later success (or failure) in the acquisition of more concepts/skills. We found that 63% of students had difficulty understanding basic programs involving the assignment operator and a sequence of statements. The inability of students to understand assignment and sequencing proved to be a substantial obstacle to student progress in learning more advanced flow control and data structures concepts like selection, repetition loops and arrays. About 66% of students who succeeded in assignment/sequencing also succeeded in selection structures. On the other hand, about 34% of the students who did not succeed in assignment/sequencing succeeded in selection structures. About 77% of the students who succeeded in assignment/sequencing and selection also succeed in repetition. Students who did not succeed in both assignment/sequencing and selection had lower percentage of success in repetition: 39% when they succeeded only in selection; 61% when they succeeded only in assignment/sequencing; 38% when they failed in both assignment/sequencing and selection. The same trends were observed when analyzing performance in student understanding of arrays. In conclusion: 1) student performance in computer programming concepts taught early in the course affects performance in computer concepts taught later in the semester; 2) The ability of students to understand concepts involving a sequence of statements is a good early predictor of success/failure in understanding more advanced concepts like selection, repetition and arrays; 3) Matthew effects are at play in learning computer programming: early success (or failure) in understanding basic computer programming concepts begets later success (or failure) in understanding more advanced computer programming concepts.

Keywords – Matthew Effect, Teaching and Learning Computer Programming, Java

I. Introduction

The Matthew effect (or cumulative advantage effect) describes situations where an initial (dis)advantage on possessing any material or immaterial resource tends to lead to further accumulation of (dis)advantage on that resource later on (for example, overtime the rich tend to get richer and the poor tend to get poorer) [1]. In the field of education, the Matthew effect could be expressed as: early success (or failure) in the acquisition of knowledge and skills begets later success (or failure) in the acquisition of more knowledge and skills.

Computer programming requires an understanding of many interrelated concepts and the ability to apply those concepts to solve problems. In computer programming courses, simpler concepts like assignment and data types are presented first, and more complicated concepts like selection and repetition, which build on simpler ones presented earlier, are presented later. One of the challenges in teaching computer programming is to structure the material such that learning difficult concepts/skills builds on already learned simpler concepts/skills, so students can make adequate progress. Understanding the dynamics of student progress in learning computer concepts in the framework of the Matthew effect can lead to further insights into teaching and learning computer programming.

The goal of this study is to quantify progress in students' understanding of fundamental computer programming concepts like assignment, selection, repetition and arrays in a Java course. Understanding the dynamics of learning progress will make possible the identification of early predictors of student success/failure and suggest pedagogical strategies to improve student outcomes in learning computer programming.

II. Background and Prior Work

The Matthew effect was first described and named by Merton [2] while studying the reward system in science. Merton noticed that prestigious scientists get more credit for comparable achievements than lesser-known scientists. In his words: "…eminent scientists get disproportionately great credit for their contributions to science while relatively unknown scientists tend to get disproportionately little credit for comparable contributions [2]." So, an initial unequally in

"prestige" gets amplified over time to increase the "prestige" gap between famous and less famous scientists. The name of the effect comes from Matthew's Gospel (25:29): "For to everyone who has, more will be given, and he will have abundance; but from him who has not, even what he has will be taken away." Before Merton, de Solla Price [3] described a similar phenomenon that he called cumulative advantage while studying networks of citations of scientific papers. He found that the number of citations that a scientific paper will get is proportional to the number of citations that it already has [3]. With this mechanism of citation growth, an initial difference in number of citations will get disproportionately amplified over time. The Matthew effect seems to be pretty general and has been used to analyze phenomena in very different fields [1][4].

The concept of the Matthew effect has also been used to understand how students learn. Walberg and Tsai [5] found that young adults with advantageous family and educational experiences are more motivated and efficient in acquiring new information. Walberg et al. [6] speculated that early achievements and rewards may lead to more motivation to learn, which in turn may lead to more achievements and rewards, in an endless cycle. Stanovich [7] explored the relationship between reading ability and cognitive processes under the lens of the Matthew effect: "The very children who are reading well and who have good vocabularies will read more, learn more word meanings, and hence read even better. Children with inadequate vocabularies – who read slowly and without enjoyment – read less, and as a result have slower development of vocabulary knowledge, which inhibits further growth in reading ability." Stanovich [7] attributed the Matthew effect to a reciprocal causality (i.e., a positive feedback mechanism) between vocabulary and reading ability.

There are a few examples in the literature that suggest that Matthew effects also play a role in learning computer programming. It has been reported that introductory computer programming courses have bimodal grade distributions with high rates of low grades, high rates of high grades, with low rates of mid-range grades [8][9][10]. This bimodal grade distribution is suggestive of a Matthew effect in learning computer programming: success (or failure) in the acquisition of programming concepts/skills begets later success (or failure) in the acquisition of more concepts/skills. To explain that bimodal distribution, Robins [11] proposed what he called the "learning edge momentum" effect. According to Robins [11], "success in acquiring one concept makes learning other closely linked concepts easier (whereas failure makes it harder)." Robins' description of the "learning edge momentum" effect [11] reads like a description of the Matthew effect. It should be noted that other reports provide alternative explanations regarding the presence or the mechanism of a bimodal grade distribution in introductory programming courses [12][13].

Another study quantified progress in the ability of first-year students to write viable programs using sequencing, selection and repetition in Python [14]. About 50% of students performed adequately in all three structures and 28% of students performed inadequately in all three. Success (failure) in writing programs using sequencing led to success (failure) in using selection, and success (failure) in using selection led to success (failure) in using repetition. These dynamics of student progress in writing computer programs are consistent with Matthew effects during the acquisition of successive computer programming skills.

III. Research Questions

The main goal of this study is to quantify student progress in learning fundamental Java programming concepts that are important to write viable computer programs. The specific research questions are:

RQ1: How student performance in computer programming concepts taught early in the course affects performance in computer concepts taught later in the semester?

RQ2: Is there an early predictor of success/failure in understanding computer programming concepts?

RQ3: Do Matthew effects determine student progress in learning programming concepts?

IV. Methods

4.1. Participants and setting

Our institution is one of the most diverse institutions of higher education in the northeast United States: 29% of our students are African American, 34% are Latino, 20% are Asian or Pacific Islanders, and 10% are Caucasian. The College's fall 2019 enrollment was 17,036.

Students enrolled in eight sections of a Java Fundamentals course between Fall 2014 and Spring 2019 were part of the study (n = 145). Students who officially dropped the class or stopped attending the class were excluded from the analysis. A number of students who participated in the study (n = 124) had taken a previous problem-solving course in which they were exposed to computer programming concepts, using different tools and environments like flowcharting, Alice and Python at our institution. The problem-solving course [14] is designed to introduce students to concepts of problem solving and computer programming languages. A major emphasis of the course is to teach students programming structures that control

the flow of execution such as sequencing, selection, and repetition loops. The remaining 21 students were students who transferred to our institution and took the Java Fundamentals course.

4.2. Student performance in understanding Java concepts

We measured student performance in understanding the syntax and semantics of Java programming in four different conceptual categories: assignment/sequencing, selection (if/else), repetition loops (for/while) and arrays. Like in other institutions teaching similar courses, students were presented with those concepts in that chronological order. Assessments were implemented as multiple choice and short answer questions that involved reading and writing small snippets of code. We used assignment/sequencing assessments to evaluate student understanding of the syntax of the assignment operator and data types. We also evaluated student understanding of assignment operator semantics by asking students to compute the values stored in variables after the execution of a sequence of statements, and to write code that swaps the values of two variables. We used selection assessments to evaluate students' ability to write and evaluate boolean expressions, to translate conditions expressed with words into boolean expressions, and to write if/else statements that use those conditions. We used repetition loop assessments to evaluate student understanding of for/while loop syntax. We also evaluated student understanding of repetition loop semantics by asking students to convert for loops to while loops and vice versa, to compute the output produced by given repetition loops and to write input validation loops. We used array assessments to evaluate students' ability to declare, populate and display arrays, and basic array processing. Each problem was graded on a 0-10 scale. Students who obtained a grade of >= 7 (equivalent to

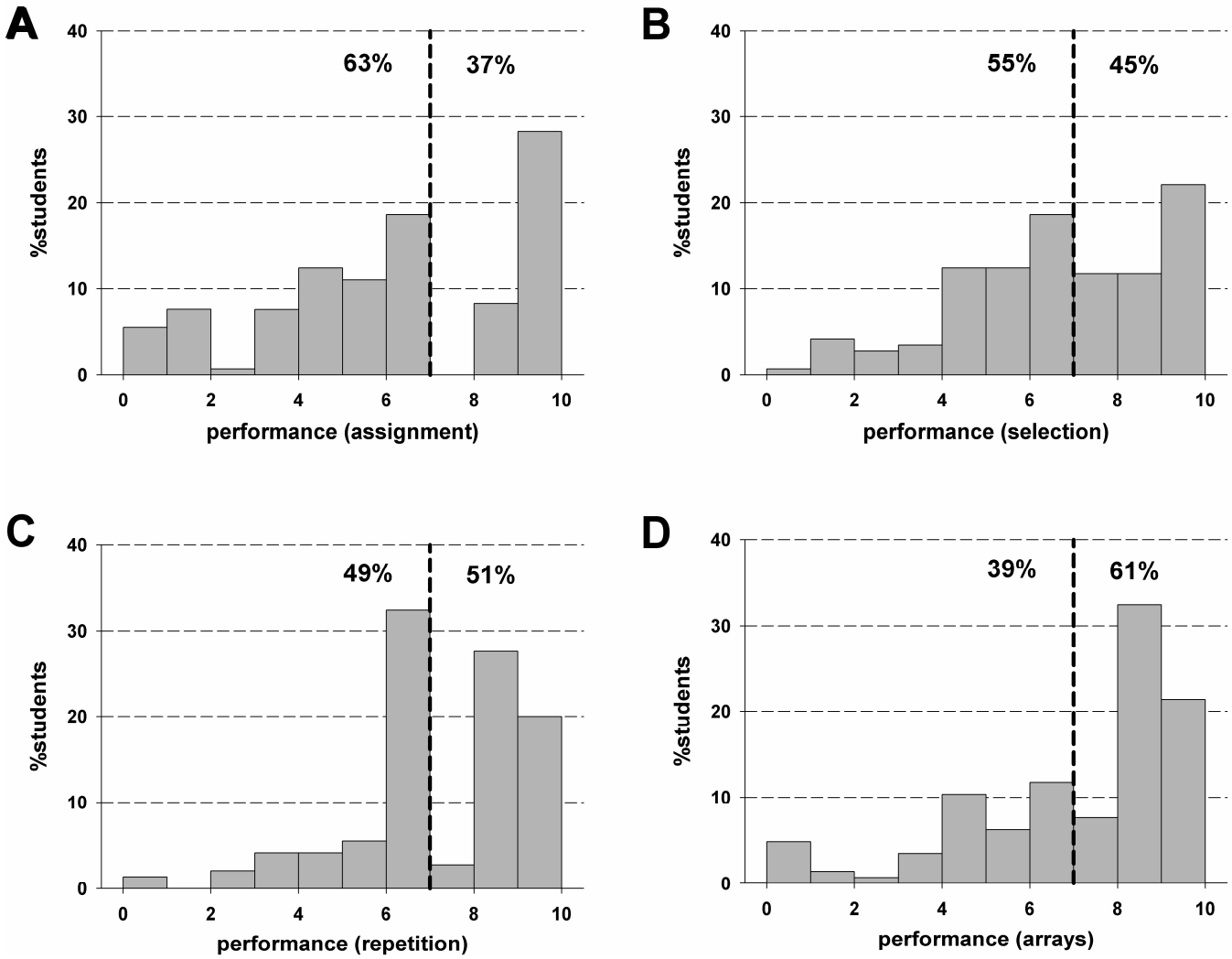

Figure 1. Histograms of student performance (0-10) in assignment/sequence (A), selection (B), repetition(C) and arrays (D). Vertical dashed line indicates adequate performance (70%). The numbers to the right (left) of the vertical dashed line indicate the percentage of students with adequate (inadequate) performance in the different conceptual categories.

70% or a C) in the average of problems in a given category were considered proficient in that category.

V. RESULTS

5.1. Summary of student performance in understanding Java concepts

The average student performance (scale 0-10) in understanding Java concepts in the four different categories was: 6.3+/-3.1 (assignment/sequence), 6.5+/-2.4 (selection), 7.3+/-2.1 (repetition) and 7.2+/-2.7 (arrays). Differences in average performance between the different categories were not statistically significant (analysis of variance, $p<0.05$). The average student performance hovers around the level of proficiency (7/10 or 70%). Average performance does not give a clear idea of how many students adequately understand the different programming concepts. The histograms in Figure 1 show in more detail the percent of students in each performance bin (0-10) for the four different conceptual categories investigated. The figure also shows the percentage of students with adequate (>=7, number to the right of the vertical dashed line) and inadequate performance (<7, number to the left of the vertical dashed line) in understanding the different Java concepts. For many of our students understanding assignment (63%), selection (55%) and repetition (49%) Java concepts is a challenge.

5.2. Learning progress binary tree

The main goal of our study is to quantify how success/failure in learning concepts taught early in the course affect success/failure in learning concepts taught later in the course. That information cannot be obtained from the performance summary in Figure 1 because that figure does not present performance associations between the different concepts.

We found that associations between student performance in different Java concepts and progress in student learning can be better visualized using a binary tree (Figure 2). In this section we describe how the tree was constructed and what the different levels and nodes in the tree represent. In the next section we explain the implications of the results presented in Figure 2 to teaching and learning Java concepts.

The level of the nodes in the tree indicates the order in which a given concept was presented to students: assignment (level 1), selection (level 2), repetition (level 3) and arrays (level 4). The first level of the tree (labeled ASSIGNMENT) consists of two nodes: the top node (n=53 or 37%) represents the number of students who had a performance >= 7 in assignment concepts (note incoming up arrow labeled as “pass”, meaning that those students were proficient in that concept); the bottom node (n=92 or 63%) represents the number of students who were not proficient (performance <7) in assignment concepts (note incoming down arrow labeled as “fail”).

The second level of the tree (labeled SELECTION) consists of four nodes. The top two nodes at the SELECTION level represent how many of the students who were proficient in assignment (level 1, top node) were also proficient in selection (top node, n=35 or 24%), or not proficient in selection (second node from the top, n=18 or 13%). The lower two nodes at the SELECTION level represent how many of the students who were not proficient in assignment (level 1, bottom node) were proficient in selection (third node from the top, n=31 or 21%), or not proficient in selection (fourth node from the top, n=61 or 42%). The numbers next to the arrows indicate conditional probabilities. For example, the probability that a student that is proficient in assignment be also proficient in selection concepts is 66% (level 2, top node), and the probability that a student proficient in assignment not be proficient in selection concepts is 34% (level 2, second node from the top).

The same process was applied to create level 3 (REPETITION) and level 4 (ARRAYS) of the tree. For example, the fifth node from the top at level 4 (n=7, 5%) represents the number of students who were proficient in assignment AND not proficient in selection AND proficient in repetition AND proficient in arrays.

Each node at each level has only one incoming arrow: if the arrow is pointing upwards (“pass”) it means that the node represents a group of students proficient in the concept represented by the level; if the arrow is pointing downwards (“fail”) it means that the node represents a group of students not proficient in the concept represented by the level. Each node at each level has two outgoing arrows (except the leaf nodes in level 4): the arrow pointing upwards leads to a node that represents a group of students proficient in the concept represented by the next level; the arrow pointing downwards leads to a node that represents a group of students not proficient in the concept represented by the next level. At each level the sum of all students (*n* numbers inside the nodes) is the total number of students, and the sum of all percentages (numbers in parenthesis inside the nodes) is 1 (or 100%).

5.3. Matthew effects in learning Java concepts

In this section we describe the implications of the performance associations shown in Figure 2 for teaching and learning Java programming concepts. About 37% of students were proficient in assignment concepts early in the course and 63% were not (Figure 2, level 1). The majority of students who were proficient in assignment (66%) were also proficient in selection (Figure 2, level 2, top node). On the other hand, the majority of students who were not proficient in assignment

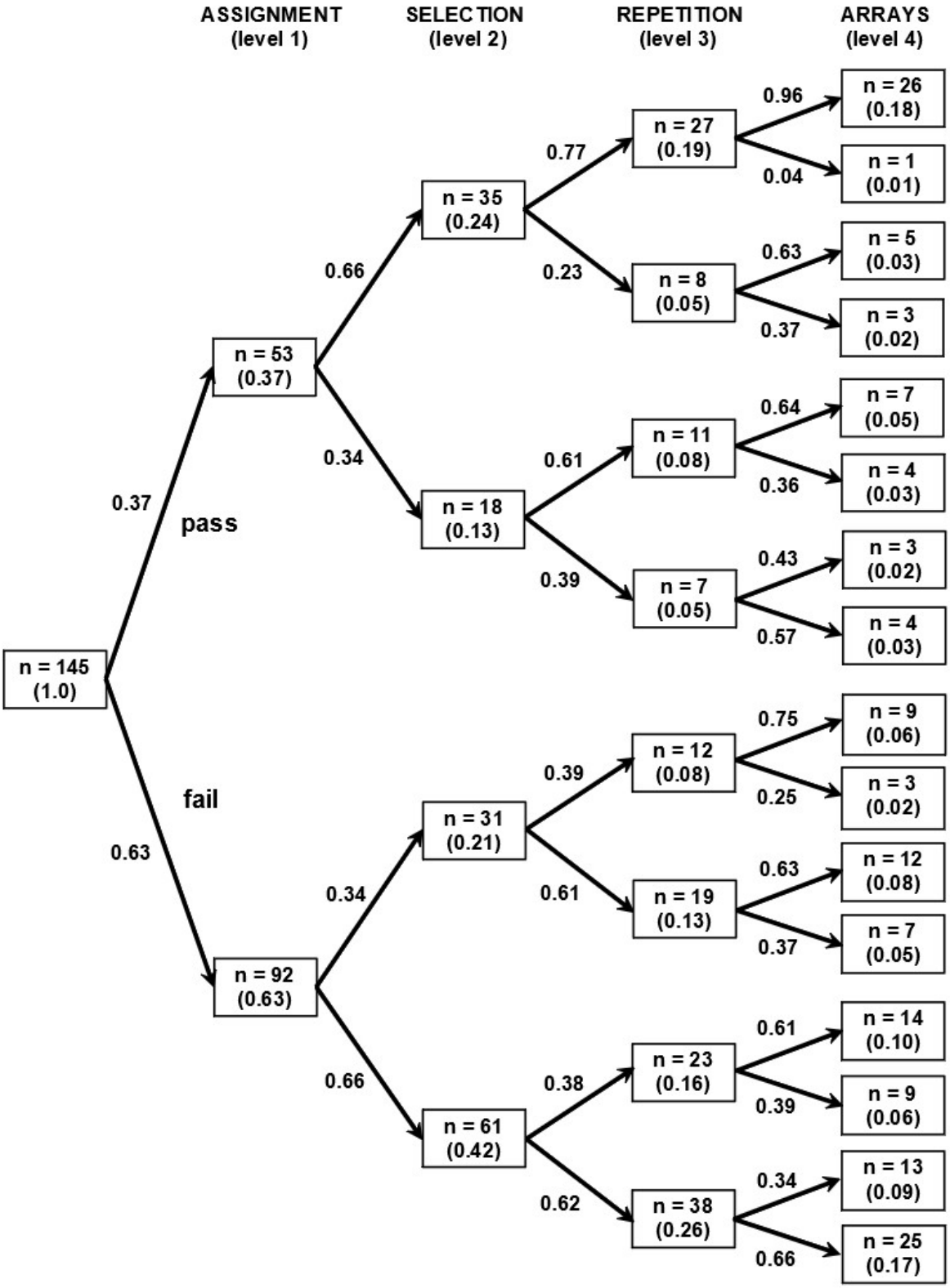

Figure 2. Learning Progress Binary Tree consisting of four levels: Assignment (level 1), Selection (level 2), Repetition (level 3) and Arrays (level 4). The tree shows students proficient (pass) or not proficient (fail) in each of those four conceptual categories. See section 5.2 in the text for explanation.

(66%) were not proficient in selection either (Figure 2, level 2, bottom node). After assessments in assignment and selection (level 2 in Figure 2) the majority of students are in the top node (24%, adequate performance in both assignment and selection concepts) or in the bottom node (42%, poor performance in both assignment and selection concepts). This is consistent with a cumulative advantage effect (or Matthew effect) because it shows that good performance in assignment concepts leads to good performance in selection concepts, and that bad performance in assignment leads to bad performance in selection. It should be noted that 13% of students did well in assignment but not in selection, and that 21% of students did not do well in assignment but did well in selection.

As we proceed to level 3 in the tree (REPETITION), cumulative advantage effects in student learning of Java concepts persist. The majority of students who were proficient in assignment and selection concepts (77%) were also proficient in repetition (Figure 2, level 3, top node). On the other hand, the majority of students who were not proficient in both assignment and selection (62%) were not proficient in repetition (Figure 2, level 3, bottom node). Students who did not succeed in both assignment and selection had lower percentage of success in repetition: 39% when they succeeded only in selection (level 3, fifth node from the top); 61% when they succeeded only in assignment (level 3, third node from the top); 38% when they failed in both assignment and selection (level 3, seventh node from the top). As, it occurred in level 2, after assessments in assignment, selection and repetition (level 3 in Figure 2) the majority of students are in the top node (19%, adequate performance in assignment, selection and repetition concepts) or in the bottom node (26%, poor performance in assignment, selection and repetition concepts).

The same trends were observed when the performance in student understanding of arrays was associated with their performance in assignment, selection and repetition (Figure 2, level 4, ARRAYS). The majority of students who were proficient in assignment, selection and repetition (96%) were also proficient in arrays (Figure 2, level 4, top node). On the other hand, the majority of students who were not proficient in assignment, selection and repetition (66%) were not proficient in arrays (Figure 2, level 4, bottom node).

Note that for every level in the tree, the nodes with the larger number of students are the top node (students who are proficient in all conceptual categories) or at the bottom node (students who are not proficient in any of the conceptual categories). This is characteristic of processes exhibiting cumulative advantage (or cumulative disadvantage) effects.

Overall, the results in Figure 2 are consistent with Matthew effects in learning Java concepts. Success (or failure) in the acquisition of earlier Java concepts begets success (or failure) in the acquisition of later Java concepts.

5.4. Student performance in assignment assessments

One of the main results presented in Figure 2 is that the students' performance in assignment assessments determines their later performance in selection, repetition and array assessments. About 49% (=26/53) of students who are proficient in assignment will be also proficient in selection, repetition and arrays. On the other hand, about 27% (=25/92) of students who are not proficient in assignment will not be proficient in selection, repetition and arrays either. So, the question is: What determines the initial disparity in performance in assignment concepts?

Figure 3 shows the relationship between student grades in an earlier pre-requisite problem-solving course (see Methods, section 4.1) and their performance in assignment assessments in the Java course. The grade in the pre-requisite problem-solving course, which is an estimation of the degree of preparedness in basic understanding of computer programming concepts, had an effect on how well students did in assignment assessments: ~44% of A grade students, ~31% of B grade students, and ~20% of C grade students were proficient in assignment concepts in the Java course.

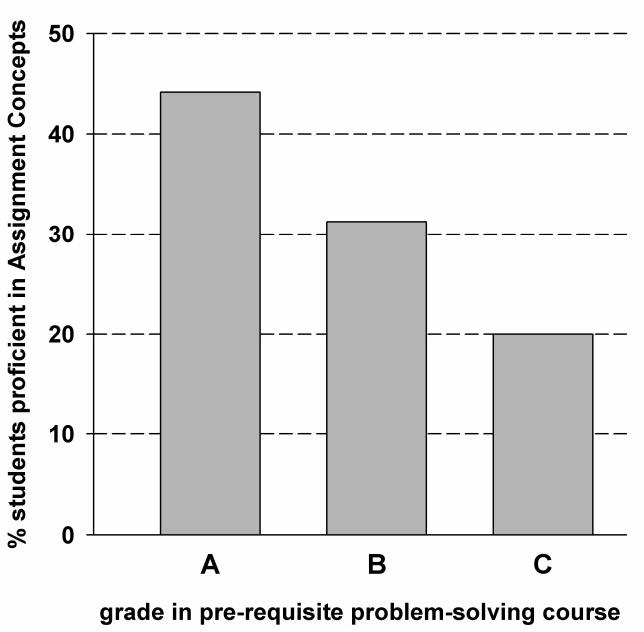

Figure 3. Relationship between student grades in an earlier pre-requisite problem-solving course and their performance in assignment assessments in the Java course.

Therefore, student performance in the pre-requisite problem-solving course affects their performance in assessments at the beginning of the semester in the Java course (and beyond as shown in Figure 2). This means that the Matthew effects at play in the Java course amplifies initial disparities on computer programming concepts preparedness. Interestingly, Matthew effects also affect student progress in the problem-solving course [14].

VI. DISCUSSION

6.1. Student progress in learning computer programming concepts (RQ1 and RQ2)

Figure 2 shows that student performance in understanding a given concept affects performance in concepts taught later in the semester as students make progress learning assignment, selection, repetition and arrays. For example, understanding assignment concepts facilitates the understanding of selection concepts, and understanding assignment and selection concepts facilitates the understanding of repetition concepts. The same dynamic applies when students do not understand a concept. Not understanding assignment concepts hinders the understanding of selection concepts, and not understanding assignment and/or selection concepts hinders the understanding of repetition concepts.

Not all students performed well in all concepts or poorly in all concepts (i.e., not all students are at the top and bottom nodes of each level in the tree in Figure 2). For example, Figure 2 shows that ~8% (4/53, level 4, 8th node from the top) of students who performed adequately in assignment concepts performed inadequately in selection, repetition and arrays. It is possible that some students who performed well in concepts taught early in the semester became disengaged later as the course progressed. Figure 2 also shows that ~10% (9/92, level 4, 9th node from the top) of students who performed inadequately in assignment concepts performed adequately in selection, repetition and arrays. It is possible that some students who had a poor performance in assignment concepts eventually caught up and understood those concepts as they were used and reinforced later the semester while learning other concepts.

Our results, showing that about 63% of students have difficulty with assignment and sequence statements (Figure 2), are consistent with earlier reports. Dehnadi [15] showed that ~50% of students failed to understand the meaning of assignment and sequence in the first three weeks of a Java course. Common misconceptions about assignment and understanding a sequence of statements include: 1) confusing the syntax of the assignment operator with equality in mathematical expressions (for example, some students erroneously think that *a=a+a;* produces a syntax error, and that *a+b=c;* does not); 2) not understanding how the mechanics of the assignment operator work (for example, some students erroneously think that *a=b;* followed by *b=a;* swaps the values of *a* and *b*); 3) ignoring data types (for example, some students erroneously think that the name of a variable determines its type, and that a variable named *word* and declared as *boolean word;* can contain text); 4) confusing text and numeric literals (2 and "2"). It is possible that those misconceptions result from the difficulty of students in creating mental models of how programs work [16][17]. Therefore, it is important that instructors develop strategies to help student create adequate mental models of programming. Moreover, it would be unproductive for students to proceed to learn selection, repetition and array concepts until they have an adequate understanding of assignment and a sequence of statements.

Common misconceptions about selection structures, which are consistent with earlier reports [18][19], include: 1) inability to write and evaluate boolean expressions; 2) inability to translate a condition formulated with words into a boolean expression; 3) syntax errors writing if/else statements (not understanding if/else blocks with and without curly braces and incorrect placement of a semicolon after the condition of an if statement). Since repetition loops use boolean expressions, it is not surprising that 59% (39/66) of students who performed well in selection also performed well in repetition (Figure 2). On the other hand, 57% (45/79) of students who performed poorly in selection also performed poorly in repetition (Figure 2). It should be noted that performance in repetition is affected not only by performance in selection concepts but also by performance in assignment concepts: 77% (compare to 59%) of students who performed well in both selection and assignment concepts also performed well in repetition. In contrast, only 39% (compare to 59%) of students who performed well in selection but not in assignment performed well in repetition.

Common mistakes in the use of arrays relate to: 1) array indexing; 2) iteration through an array with repetition loops; 3) treating strings as arrays. Since array processing relies on repetition loops, it is not surprising that 77% (56/73) of students who performed well with repetition loops also performed well with arrays (Figure 2). On the other hand, 54% (39/72) of students who performed poorly with repetition also performed poorly with arrays (Figure 2). Performance in arrays is affected not only by performance in repetition but also by performance in earlier concepts. For example, 96% (compare to 77%) of students who performed well in repetition and selection and assignment concepts performed well in arrays. In contrast, 61% (compare to 77%) of students who performed well in repetition, but poorly in selection and assignment concepts, performed well on arrays (Figure 2).

One approach to improve student success in Java programming courses is to find early predictors that could identify students at risk. However, finding effective early predictors of student performance has remained elusive [9] [11]. Our results show that student performance in assignment and sequence concepts is critical because it may determine their performance in subsequent concepts like selection, repetition and arrays (Figure 2). Therefore, performance in assignment concepts can be used as an early predictor of students' future performance. As a consequence, improving student understanding of assignment and sequence concepts should lead to a better understanding and performance in future concepts. Performance in assignment and sequence concepts can also be used to identify potential high-performing and low-performing students to create pair-programming teams. Pair-programming has been shown to be an effective pedagogical strategy in introductory computer programming courses [20][21].

6.2. Matthew effects in learning computer programming concepts (RQ3)

Figure 2 shows that the dynamics of student progress in learning computer programming concepts are consistent with Matthew effects. For every level in the tree, the nodes with the larger number of students are the top node (students who are proficient in all conceptual categories) or at the bottom node (students who are not proficient in any of the conceptual categories). This is characteristic of processes exhibiting cumulative advantage (or cumulative disadvantage) effects. The fact that the understanding of programming concepts taught later in the course relies on the understanding of concepts taught earlier may explain Matthew effects in learning computer programming. Earlier research on Matthew effects on education suggests that reciprocal causality (i.e., positivity feedback) could be one of the mechanisms [6][7]. Early achievements and rewards may lead to more motivation to learn, which in turn may lead to more achievements and rewards, in an endless cycle [6]. This relationship between motivation and learning success could also be a mechanism at play in learning computer programming. Further studies are necessary to find the reciprocal causality networks (like the relationship between vocabulary and reading ability in [7]) that cause Matthew effects in learning computer programming.

Some students are not at the top or bottom node for each layer and do adequately in some of the conceptual categories. For example, for level 4 in Figure 2, ~18% of students do well in all concepts, ~17% do poorly in all concepts, and ~65% of students do well in one, two or three of the four categories. If progress were determined by purely Matthew effects, at each level, students would be located in just the top and bottom nodes, with no students in the nodes in between. This shows that the Matthew effect can be modulated: some students are able to succeed after an initial failure, and some students may fail after initial success. Moreover, Matthew effects may be different for different pedagogies and tools used to teach programming. For example, when students learn programming with a flowchart interpreter, Matthew effects are less pronounced than when students learn programming with Python [14].

Other strategies that could emphasize the benefits and deemphasize the hindrances of Matthew effects could be: 1) to present concepts to students in an effective scaffolded sequence, so students can make adequate progress in their ability to understand computer concepts [22]; 2) to perform continuous reviews and reinforcement of earlier material and merge earlier concepts with new ones; 3) help students develop metacognitive skills so they understand which current concepts they need to master before trying to understand new ones to prevent disengagement and increase self-efficacy [23]; 4) to develop personalized teaching and learning strategies [24].

VII. Future Work

Students' challenges when learning programming are multifactorial [24][25]. A major long-term goal is to investigate how those multiple challenges interact with each other creating reciprocal causality networks that result in Matthew effects. The pedagogical strategy would be to emphasize reciprocal causality networks that beget success from early success, and to deemphasize (or compensate for) reciprocal causality networks that beget failure from earlier failure.

VIII. Limitations

Further studies are necessary to determine if our results hold in a different context, and apply to larger and/or different student populations [26].

IX. Conclusions

1) Student performance in computer programming concepts taught early in the course affects performance in computer concepts taught later in the semester.

2) The ability of students to understand concepts involving a sequence of statements is a good early predictor of success/failure in understanding more advanced control structures like selection, repetition and arrays.

3) Matthew effects are at play in learning computer programming: early success (or failure) in understanding basic computer programming concepts begets later success (or failure) in understanding more advanced computer programming concepts.

REFERENCES

[1] Rigney D. 2013 The Matthew effect: how advantage begets further advantage. Columbia, UK: Columbia University Press.

[2] Merton RK. (1968) The Matthew effect in science. Science 159, 53–63. (doi:10.1126/science.159.3810.56)

[3] de Solla Price DJ. (1965) Networks of scientific papers. Science 149, 510–515. (doi:10.1126/science.149.3683.510)

[4] Perc M. 2014 The Matthew effect in empirical data. J. R. Soc. Interface 11:20140378. http://dx.doi.org/10.1098/rsif.2014.0378

[5] Walberg, H. J., &Tsai, S. (1983). Matthew effects in education. *American Educational Research Journal, 20, 359–373.*

[6] Walberg, H. J., Strykowski, B. E, Roval, E., & Hung, S. S. (1984). Exceptional performance. *Review of Educational Research, 54*, 87–112.

[7] Stanovich KE. (2008) Matthew effects in reading: some consequences of individual differences in the acquisition of literacy. J. Educ. 189, 23–55.

[8] Hudak, M.A. & Anderson D.E. (1990). Formal operations and learning style predict success in statistics and computer science courses. *Teaching of Psychology, 17(4),* 231–234.

[9] Bornat, R., Dehnadi, S. & Simon (2008). Mental models, consistency and programming aptitude. Proceedings of the Tenth Australasian Computing Education Conference (ACE 2008), 53–62.

[10] Corney, M. (2009). Designing for engagement: Building IT systems. *ALTC First Year Experience Curriculum Design Symposium 2009.* Queensland, Australia: QUT Department of Teaching and Learning Support Services, 19–21.

[11] Robins, A. (2010). Learning edge momentum: a new account of outcomes in CS1. Computer Science Education, 20(1), 37–71.

[12] R. Lister. (2010) Computing education research geek genes and bimodal grades. ACM Inroads, 1(3):16-17, 2010.

[13] Elizabeth Patitsas, Jesse Berlin, Michelle Craig, and Steve Easterbrook. 2016. Evidence That Computer Science Grades Are Not Bimodal. In *Proceedings of the 2016 ACM Conference on International Computing Education Research* (*ICER '16*). Association for Computing Machinery, New York, NY, USA, 113–121. doi:10.1145/2960310.2960312

[14] Cabo, C. (2019) "Student Progress in Learning Computer Programming: Insights from Association Analysis," *2019 IEEE Frontiers in Education Conference (FIE)*, Covington, KY, USA, 2019, pp. 1-8, doi: 10.1109/FIE43999.2019.9028691.

[15] Dehnadi, S. A Cognitive Study of Learning to Program in Introductory Programming Courses. PhD thesis, Middlesex University, 2009.

[16] Johnson-Laird, P. N. (1983). *Mental models*. Cambridge: Cambridge University Press.

[17] Spohrer, J.C., Soloway, E (1986). Novice mistakes: are the folk wisdoms correct? Commun. ACM, 29(7):624-632, 1986.

[18] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. 2003. Identifying and correcting Java programming errors for introductory computer science students. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education* (*SIGCSE '03*). Association for Computing Machinery, New York, NY, USA, 153–156. DOI:https://doi.org/10.1145/611892.611956

[19] Neil C. C. Brown and Amjad Altadmri. 2017. Novice Java Programming Mistakes: Large-Scale Data vs. Educator Beliefs. *ACM Trans. Comput. Educ.* 17, 2, Article 7 (June 2017), 21 pages. DOI:https://doi.org/10.1145/2994154

[20] Charlie McDowell, Linda Werner, Heather Bullock, and Julian Fernald. (2002). The effects of pair-programming on performance in an introductory programming course. In *Proceedings of the 33rd SIGCSE technical symposium on Computer science education* (*SIGCSE '02*). Association for Computing Machinery, New York, NY, USA, 38–42. DOI:https://doi.org/10.1145/563340.563353

[21] Nachiappan Nagappan, Laurie Williams, Miriam Ferzli, Eric Wiebe, Kai Yang, Carol Miller, and Suzanne Balik. (2003). Improving the CS1 experience with pair programming. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education* (*SIGCSE '03*). Association for Computing Machinery, New York, NY, USA, 359–362. DOI:https://doi.org/10.1145/611892.612006

[22] Bruner, J.S. *Toward a Theory of Instruction*. Harvard University Press, Cambridge, MA, 1966.

[23] Bandura, A. (1977). Self-efficacy: Toward a unifying theory of behavioral change. *Psychological Review, 84*(2), 191–215. https://doi.org/10.1037/0033-295X.84.2.191

[24] Gomes, A., Mendes, A.J. (2007) Learning to program – difficulties and solutions. International Conference on Engineering Education (ICEE 2007), September 3-7, 2007, Coimbra, Portugal.

[25] Robins A, Rountree J, and Rountree N. (2003). Learning and Teaching Programming: A Review and Discussion. Computer Science Education 13: 137-172.

[26] Fincher, S., Lister, R., Clear, T., Robins, A., Tenenberg, J. & Petre, M. (2005). Multi-institutional, multi-national studies in CSEd Research: some design considerations and trade-offs. Proceedings of the First international Workshop on Computing Education Research (ICER '05), 111–121.