Discovering, Autogenerating, and Evaluating Distractors for Python Parsons Problems in CS1

David H. Smith IV
University of Illinois
Urbana, IL, USA
dhsmith2@illinois.edu

Craig Zilles
University of Illinois
Urbana, IL, USA
zilles@illinois.edu

ABSTRACT

In this paper, we make three contributions related to the selection and use of distractors (lines of code reflecting common errors or misconceptions) in Parsons problems. First, we demonstrate a process by which templates for creating distractors can be selected through the analysis of student submissions to short answer questions. Second, we describe the creation of a tool that uses these templates to automatically generate distractors for novel problems. Third, we perform a preliminary analysis of how the presence of distractors impacts performance, problem solving efficiency, and item discrimination when used in summative assessments. Our results suggest that distractors should not be used in summative assessments because they significantly increase the problem's completion time without a significant increase in problem discrimination.

CCS CONCEPTS

• **Social and professional topics → Computing education**.

KEYWORDS

Parsons Problems, CS1, tools, distractors, item discrimination

ACM Reference Format:
David H. Smith IV and Craig Zilles. 2023. Discovering, Autogenerating, and Evaluating Distractors for Python Parsons Problems in CS1. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2023), March 15–18, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3545945.3569801

1 INTRODUCTION

Parsons problems were first proposed by Parsons and Haden [28] as a method of facilitating the development of fundamental semantic and syntactic concepts in introductory programming students. These problems consist of individual blocks of code that must be arranged in a specific order in order to construct a valid solution. Code blocks that either contain errors or are not used in the final solution, commonly referred to as distractors, were introduced in the original and continue to be included in many studies involving Parsons problems [6].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGCSE 2023, March 15–18, 2023, Toronto, ON, Canada
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9431-4/23/03…$15.00
https://doi.org/10.1145/3545945.3569801

One of the commonly cited principles for using distractors in Parsons problems is that those distractors should reflect common programming errors and misconceptions [6, 28]. Much like with multiple-choice questions, distractor blocks are added to Parsons problems with the intention of distracting students with plausible, but incorrect alternatives. When included on homework assignments the purpose of distractors is to correct common errors and misconceptions students may have when writing programs without having them deal with the full cognitive load of writing a program from scratch [10, 17]. When included on exams, their purpose shifts to providing an additional dimension by which to discriminate based on student knowledge. As such, the selection of distractors that accurately reflect common errors is important when pursuing either of these goals.

The design and impact of distractors with respect to multiple-choice questions has long been a topic of consideration given the widespread usage of this question format. As summarized by Gierl et al. [14], designing distractors is difficult for three main reasons. First, it requires an experienced individual to write a large number of plausible but incorrect options. Next, if distractors are too obviously incorrect it can limit the amount of thought students must put into answering the question. This, in turn, limits the learning potential of the question [22]. Finally, the quality of distractors impacts the quality of feedback an instructor can receive from the question on the misconceptions their class may hold.

Though experts may attempt to construct distractors based on their perceptions of the issues students face when writing a given line of code, expert-blind spots may make this an unreliable method for generating distractors [27]. As such, one of the common methods by which distractors are created is by collecting common errors or misconceptions from related short response questions [4, 15]. Though these are commonly collected from students responses on test and homework questions, there are other methods by which these responses can be collected at scale. For example, Scheponik et al. [30] found that using open-ended questions posted on Amazon's Mechanical Turk was a useful method by which to collect distractors for multiple-choice questions.

Towards extending the practice of leveraging errors made on short answer responses, this paper presents a process of analyzing such questions and automating the construction of distractors sets for Parsons problems. Section 3 describes the process by which common errors were analyzed from an introductory Python course. Section 4 describes the system used to automatically generate distractor sets from a solution's source code. Finally, Section 6 provides preliminary comparison of the difficulty and item discrimination of problems with distractors compared to those without on a large introductory Python course's final exam.

2 BACKGROUND

2.1 Origins and Utility of Parsons Problems

The development of "Parsons Programming Puzzles" by Parsons and Haden [28] was initially aimed at allowing for a more engaging form of the type of practice drills that are common in other introductory science and engineering courses. The purpose of those drilling exercises, and by extension Parsons problems, is to facilitate the development of fundamental programming skills. These include the abilitiy to correctly identify correct syntax and create logical constructs. As such, these programming puzzles were developed with the following design goals in mind:

(1) Permitting common syntactic and logical errors.
(2) Addressing misconceptions through immediate feedback.
(3) Modeling good code.
(4) Constraining the logic used to solve a problem.

The combination of permitting common errors through the inclusion of distractors and providing immediate feedback is used as a method by which common errors can be addressed in a scaffolded environment. Parsons problems have since gained traction given their positive reception among students and instructors alike [9, 11].

Work done by the BRACElet project has suggested that skills used in the solving of Parsons problems occupy a middle position in their proposed programming skills hierarchy; specifically, between the more advanced skill of code writing and more rudimentary skills like code tracing [23, 32, 36]. However, a later study suggests that the community hasn't yet performed the experiments necesary to justify how Parsons problems should be sequenced with other kinds of questions [12]. Despite the limitations of the proposed skill hierarchy, the role of Parsons problems in facilitating progression through it are still of pedagogical interest due to the benefits found by prior work.

Several studies have indicated that Parsons problems are both more efficient and require less cognitive load while still achieving the same learning benefits compared to practicing with code writing and code fixing exercises. Ericson et al. [10] compared Parsons problems that require students to correctly indent blocks, code fixing, and code writing problems on the basis of efficiency, effectiveness, and cognitive load. Efficiency was operationalized as the amount of time needed to complete the problem, effectiveness was the increase in performance between a pre and post test after completing the set of treatment problems, and cognitive load was measured using the CS Cognitive Load Component Survey [26].

2.2 Variations of Parsons Problems

Since their inception, many variations of Parsons problems have been developed and evaluated. These investigations have primarily focused on increasing efficiency and reducing cognitive load while maximizing, or at least maintaining, the learning gains that Parsons problems have been demonstrated to achieve [6].

There are two main ways in which distractors are included in Parsons problems. The original method simply randomly placed distractors in with correct blocks and left it to the students to distinguish which they should use, a process that would come to be known as jumbled distractors. Prior work has found that including jumbled distractors leads to a decrease in efficiency and completion while increasing cognitive load [13, 16] and increasing the number of distractors made problems more difficult [9]. A recent alternative to jumbled distractors involves visually grouping the distractor(s) with the correct block of code they are associated with which regains some efficiency making the need to select between multiple options more explicit [5].

In addition to distractors, Parsons problems can be designed to be insensitive to indentation for languages like C and Java, commonly referred to as one-dimensional, or sensitive to it for languages like Python, also referred to as two-dimensional [19]. Prior work indicates that two-dimensional problems are more difficult than their one dimensional counterparts [18].

Weinman et al. [34] has investigated the usage of "Faded Parsons Problems". These differ from traditional Parsons problems in that they do not consist of static blocks of code. Rather, each block contains some skeleton code with entry boxes for certain values and symbols left to be filled in by the student.

Finally, recent work has investigated the utility of adaptive Parsons problems, where, as students submit incorrect responses, distractors are eliminated and correct blocks are combined. The purpose of these problems is to progressively morph the problem to fit the students current ability level [7, 8]. The goal of these problems is to keep the student completing them in the zone of proximal development [33], where students are challenged but still able to progress rather than stagnating and becoming frustrated.

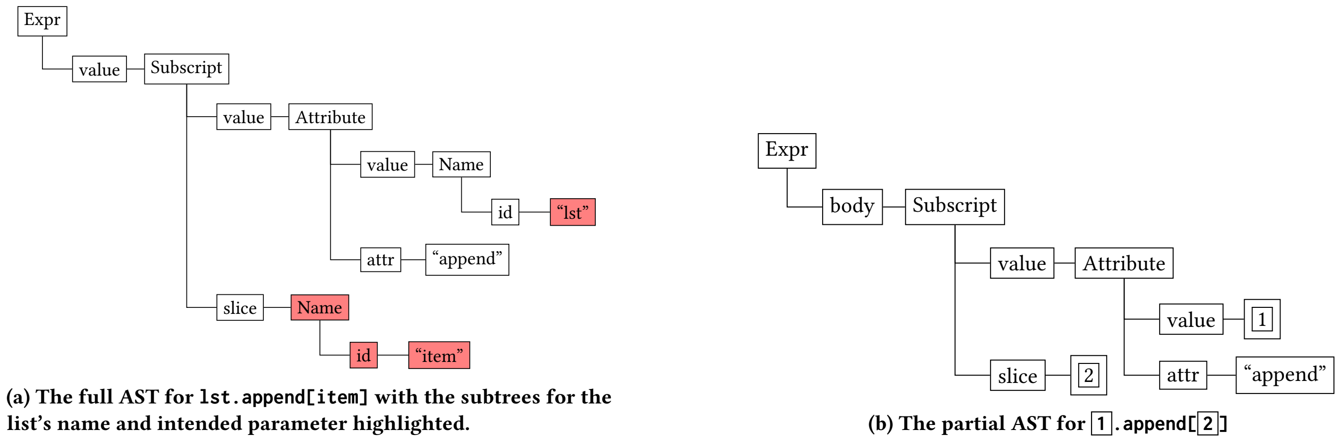
2.3 Role of Distractors

Despite the initial purpose of Parsons problems as a type of drilling exercise, they have also been used as a tool for examination [21, 23]. Denny et al. [5] investigated the correlations between Parsons, code writing, and tracing problems on an assessment and found a particularly high correlation between students scores on Parsons and code writing questions, suggesting they measure similar skills. Additionally, they suggest that the errors made in a Parsons problem provide a more explicit indication of the concepts with which students might be struggling. However, to the best of our knowledge, no studies have investigated the effect of distractors on item discrimination.

The concept of "desirable difficulties" has long been a consideration when constructing test-items and learning activities [2, 3]. In the context of multiple-choice questions on exams, the selection of distractors can impact the retrieval processes necessary to solve the problem [22]. The retrieval processes that occur during assessments have been associated with increasing retention of the information on which the student is being tested in what has come to be known as the "testing-effect" [20, 29]. As such, the selection of effective distractors is not only a concern in constructing questions that effectively discriminate between high and low performing students, but they may also play a role in maximizing a given item's learning potential.

3 CONSTRUCTING A DISTRACTOR TEMPLATE SET

The construction of our sets of distactor templates begins with a set of problems we refer to as "statement questions". These questions require students write a single line of code that accomplishes



(a) The full AST for `lst.append[item]` with the subtrees for the list's name and intended parameter highlighted.

(b) The partial AST for $\boxed{1}$`.append[`$\boxed{2}$`]`

Figure 1: A full AST and the partial AST that is capable of matching it

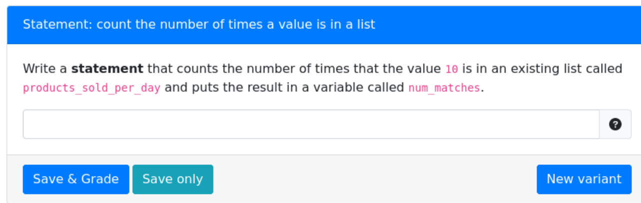


Figure 2: One of the statement questions analyzed for which submissions were analyzed to construct distractor templates.

some task such as appending to an existing list (Figure 2). These problems are much simpler to analyze than typical code writing solutions as they restrict possible misconceptions and errors to the one particular concept covered by a given question. In total, 42 of these questions have been deployed thus far in the course from which we draw our set of submissions on both homeworks and exams with the questions covering the majority of fundamental statements needed to write larger pieces of code in the Python course. For the purposes of illustrating the tool and contextualizing the pilot study detailed in Sections 5 and 6, we will focus on the subset of statement problems constructed for list statements.

The set of submissions for each question from homework and exams from four regular semesters and two summer semesters ($N_{students}$ = 2853) were parsed into Abstract Syntax Trees (ASTs). These allow us to represent the structure of code while abstracting away the specifics of the syntax used (e.g., spacing between commas). Not all submissions were parsable into ASTs due to severe syntax errors (e.g., unclosed parentheses, non-matching quotes). Like Harms et al. [16], we omit these submissions and analyze only the parsable submissions which account for 78.9%-97.9% (mean 90.1%) of the submissions to each statement question, since doing so is unlikely to cause us to miss important distractors. This provides us with more than 5,000 submissions for most questions.

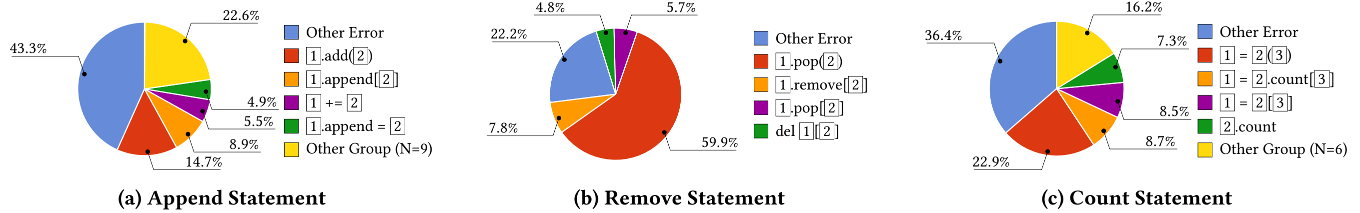
In order to construct sets of templates that can be used to generate distractors for individual lines of code, we must first identify sets of common errors that are associated with each statement. This process begins by manually examining submissions to statement questions, identifying errors that appear to be occuring frequently, and defining a partial AST that matches all submissions in which that error occured. For example, if we identify that students commonly mistake parentheses with brackets when appending to a list, we would construct an AST for a statement that characterizes that error, such as `lst.append[item]` (Figure 1a). Those subtrees that are arbitrary are removed and replaced with wildcards (Figure 1b). In doing so, a partial AST is constructed that characterizes the submissions that match the error $\boxed{1}$`.append[`$\boxed{2}$`]`, where $\boxed{1}$ and $\boxed{2}$ are arbitrary placeholders for variables and parameters. With the partial AST constructed, we can then use the structural pattern matching supported natively by Python 3.10 to count all submissions that match with this partial AST.

This process was continued until the remaining set of submissions were too heterogeneous to find groups of similar errors. After completing this process we are able to capture a significant portion of each of the errors students encountered when attempting common list statements, typically with a relatively small set of groups (range 4-22, mean=10). We remove those submissions that had an error unrelated to the function call (e.g., incorrect name, wrong value) and display the portion of incorrect submissions accounted for by the sets of distractor templates for `list.append()`, `list.count()`, and `list.remove()`, that were used for the pilot study in Figure 3.

4 AUTOMATICALLY GENERATING DISTRACTORS

Using the distractor sets and the AST matching process introduced in the previous section, this section presents a process by which an individual line of source code can be automatically matched and transformed into its associated set of distractor templates.

The process begins by constructing partial ASTs for the correct form of each of the statements for which sets of distractor templates were constructed. For example, if we want to support creating distractors for the append statement we would construct a partial AST for $\boxed{1}$`.append(`$\boxed{2}$`)`. As shown in Figure 4, if a line of code is then entered that matches this partial AST (e.g., `numlst.append(item)`)



(a) Append Statement **(b) Remove Statement** **(c) Count Statement**

Figure 3: The percentage of parsable but incorrect submissions accounted for by each of the most common errors. "Other Error" are errors that were not prevelant and therefore ignored. "'Other Group" is a submission that is accounted for but less prevelant. For "Other Group", the number of groups collapsed into that category is shown along side the label.

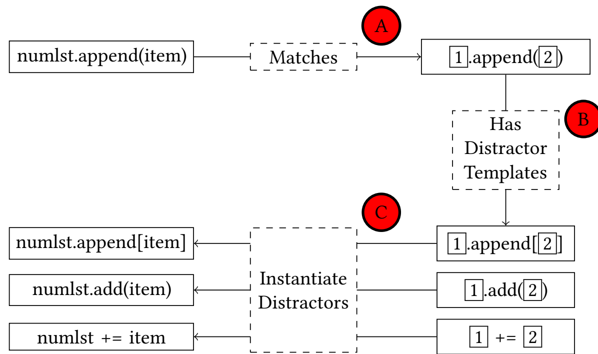


Figure 4: Transforming `numlst.append(item)` into a set of distractors.

that match can (A) be identified and (B) the set of distractor templates associated with the matched statement can be retrieved. The final step (C) involves extracting and unparsing the subtrees from the original line of code's AST that are occupied by wildcards in the partial AST. These subtrees represent the values that will be placed into their respective positions in the distractor templates in order to generate the final set of distractors.

5 METHODS

Three solutions (shown in Figure 5) were constructed using programming patterns that students were already familiar with from the course. For each of these problems, two Parsons problems were constructed, one without distractors and the other with four distractors for the circled line. Those four distractors were selected from the error categories uncovered by the analysis detailed in Section 3, and Parsons problems with those distractors were generated using the tool described in Section 4 [31]. Additionally, distractors were jumbled in among the correct blocks rather than visually pairing them.

These problems were included on the final exam for a large introductory Python course (N=494) at a large, public university in the United States using PrairieLearn [35] to administer the exams and a Computer Based Testing Facility for proctoring [37]. The course's exams were computer-based and allowed students to submit answers for grading in real time, allowing for partial credit on partially correct submissions. Students were randomly assigned one of the six Parsons problems through exam versioning.

```
def filter_numeric(lst):
  nums = []
  for item in lst:
    if type(item) in [int, float]:
      nums.append(item)
  return nums
```

(a) P1: Append

```
def count_longest(str_lst):
  longest_str = ''
  for string in str_lst:
    if len(string) > len(longest_str):
      longest_str = string
  count = str_list.count(longest_str)
  return count
```

(b) P2: Count

```
def remove_longest_string(str_lst):
  longest_str = ''
  for string in str_lst:
    if len(string) > len(longest_str):
      longest_str = string
  str_list.remove(longest_str)
```

(c) P3: Remove

Figure 5: The solutions from which Parsons problems were generated. The statements encircled in red are those from which distractors were generated.

Students were given a maximum of 6 points for their first attempt, 3 for their second, and 1 for their third. However, partial credit was given during each attempt. The score for a given attempt was calculated by taking the total number of correctly placed blocks (B_c), subtracting the number of distractors selected (B_d), and dividing the difference by the total number of blocks in the correct solution (B_t). An indentation penalty was calculated by dividing the number of correctly indented blocks (I_c) by the number of blocks in the correct solutions (B_t) and finding the product of the two ratios (Equation 1).

$$S_{\%} = \frac{max(B_c - B_d, 0)}{B_t} * \frac{I_c}{B_t} \quad (1)$$

For a given attempt, the number of points rewarded was calculated by multiplying the maximum number of points for the attempt

	With Distractors			Without Distractors				
	N	Mean	SD	N	Mean	SD	U	p
P1: Append	84	5.23	1.49	83	5.59	1.14	3952.0	>0.05
P2: Count	84	3.55	2.13	82	4.25	1.93	4105.0	<0.05*
P3: Remove	73	4.05	1.99	83	4.29	1.88	3175.5	>0.05

Table 1: Score comparisions between versions of problems with and without distractors (out of 6).

	With Distractors			Without Distractors				
	N	Mean	SD	N	Mean	SD	U	p
P1: Append	84	123.05	77.33	83	89.96	79.85	2057.5	<0.001***
P2: Count	84	271.38	174.10	82	217.75	194.66	2548.5	<0.01**
P3: Remove	73	179.57	106.00	83	156.37	156.28	2046.0	<0.001***

Table 2: Duration comparisions between versions of problems with and without distractors (in seconds).

	With Distractors			Without Distractors				
	N	Mean	SD	N	Mean	SD	U	p
P1: Append	76	122.84	76.07	78	89.02	81.04	1619.0	<0.001***
P2: Count	41	233.00	134.62	48	176.97	135.17	685.0	<0.05*
P3: Remove	46	164.34	101.94	46	114.48	80.35	567.0	<0.001***

Table 3: Duration comparisions for student who gave a correct submission between versions of problems with and without distractors (in seconds).

by the difference between the score for the current attempt and the previous best attempt.

Students were given feedback on each attempt using the system introduced by Ihantola and Karavirta [19]. Blocks up to the first error are highlighted in green. The first error is highlighted in red if it is the wrong block or in yellow if it merely has incorrect indentation. The remaining blocks remain grey.

6 RESULTS

6.1 Score and Duration Comparison

Both score and question duration statistics were collected from the assessment platform. A Shapiro-Wilk test indicated our score and duration data was not normally distributed. As such, we used Mann-Whitney U tests, a non-parametric alternative to an independent, two-sample t-test.

For the score results, all of the questions that included distractors had lower average scores, but only one of three pairs had a statistically significant difference (Table 1). With respect to duration, students spent more time on problems that included distractors with all three pairs showing a statistically significant difference (Table 2) with a moderate to low effect size (P1: $d = 0.42$, P2: $d = 0.29$, P3: $d = 0.17$). This finding holds even if we restrict our attention to students who ultimately reached a correct answer (Table 3) and the effect size becomes consistently moderate (P1: $d = 0.43$, P2: $d = 0.42$, P3: $d = 0.54$). In summary, when distractors were included, it increased the amount of time students spent on the problem, decreased efficiency for those who did solve the problem, but did not necessarily have a significant impact on score.

	With Distractors	Without Distractors
P1: Append	0.6448	0.5825
P2: Count	0.8108	0.7415
P3: Remove	0.8066	0.8867

Table 4: Item Discrimination - Pearson's R coefficients between each Parsons problem score and exam score[1]

6.2 Item Discrimination Comparison

In keeping with the standards and practices of classical test theory, we calculate discrimination as the Pearson's correlation between test scores and item scores [1]. [1] The correlation coefficients are presented in Table 4. In general, items with discrimination between 0.4 and 0.7 are considered to be excellent [24, 25]. Both the versions with and without distractors meet this criteria. P1 and P2 showed a marginal increase in discrimination with the inclusion of distractors. However, P3 showed a decrease in discrimination by including distractors.

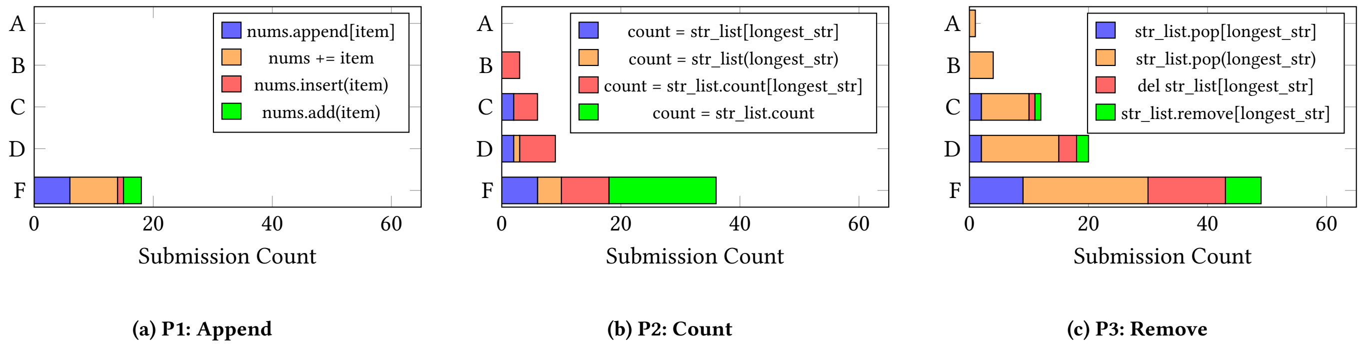
P3's lower discrimination from distractors appears to be the result of students of a wide range of performance selecting `list.pop()` when `list.remove()` is required. This can be seen in Figure 6c where we plot the number of submissions that included the distractors grouped by exam grade. In contrast, for P1 and P2 the majority of distractors which were included in a student's submission are in the exams of students who failed the assessment. However, for P3, we see that the `pop` distractor was frequently used in submissions in the B-D range. Given students were only allowed three attempts with some partial credit per submission, it is likely this distractor is at least, in part, responsible for the decrease in discrimination.

In interpreting these results, it is worth considering the context of the course and its curriculum. The `append` function was covered early on in the course and used throughout many programming and tracing problems in lecture, homework, and exams. Comparatively, the `count` and `remove` were used far less. This lack of practice may be partly responsible for many students (across grade group's) including distractors associated with the `count` and `remove` functions but only students who scored poorly on the assessment selecting distractors associated with the `append` function.

7 DISCUSSION

Overall, the results of this pilot study draw into question the utility of including jumbled distractors in Parsons problems on summative assessments. If the intention of the summative assessment in question is to reliably discriminate between high and low performing students the results presented suggest that the inclusion of jumbled distractors does little to help in this endeavor. With only small increases in the item discrimination (and a decrease in one case), but a large, statistically significant increases in time taken on those items that included distractors, it appears that not including distractor on Parsons problems could enable the inclusion of additional problems. However, as discussed in Section 6.2, the limited increase in item discrimination may be an artifact the high level of

[1] In calculating the test score, we omitted a subset of true-false and multiple-choice questions related to general computing knowledge as presented in the course's textbook. The rationale behind this decisions is that those questions as they are currently implemented on the test have a high degree of variance due to the large bank of randomly assigned questions associated with each question generator.



(a) P1: Append **(b) P2: Count** **(c) P3: Remove**

Figure 6: Occurences of distractors in submissions grouped by the overall exam grade they appeared on.

familiarity students had with those list statements at the time of taking the final. Future work may consider how including jumbled distractors in assessments and homework closer to when the topic is learned influences the impact of jumbled distractors on efficiency, performance, and item discrimination.

The findings of the pilot study showing that inclusion of jumbled distractors in Parsons problems increases the time needed to solve the problem is consistent with prior work [16]. Given previous work has shown that including visually paired distractors can help reduce the time needed to solve Parsons problems, future work should consider the impact of visually paired distractors on efficiency and performance when distractors based on common errors are used. Should prior findings be replicated this may be a method of improving problem solving efficiency and allow the focus to be placed on investigating the impact of distractors on performance and item discrimination.

Looking beyond the impact of distractors on item discrimination, distractors can potentially be used to identify errors and misconceptions that are common across students of all performance levels in the class. The ability to use distractors to make determinations such as these is one of the commonly cited motivations for using them in multiple-choice questions, as such information can be used to restructure lessons and curriculum [14]. Though this study considers the use of distractors on exams, it is likely similar information could be gleaned by including Parsons with distractors on formative assessments.

8 LIMITATIONS

The primary limitation of the process for discovering distractor templates and generating distractors from them, is that we are limited to individual statements. As such, the process and tool described does not currently support generating distractors relating to conditional statements and loops.

With respect to the pilot study, the selection of problems is reletively small and were presented to students at the end of the semester, a point at which they were more familiar with each of the statements under consideration. It may be the case that including distractors closer to when topics are being learned increases the item discrimination of the problem to a greater degree and does so more consistently.

A final limitation that exists in the pilot study lies within an error that was made when the tool was being constructed. Rather than generating the distractor `str_lst.count` for P3 the distractor `longest_str_count = str_list.count`, which was a far less common error, was generated instead. However, given the errors are similar it is likely the impact of using one less common error as the basis for generating a distractor had minimal impact on efficiency, score, or item discrimination regardless of the direction of that impact.

9 CONCLUSION

In this paper we present a methodology for collecting distractor templates and automating the construction of Parsons problems with these templates. In doing so, we were able to account for a significant number errors in student submissions on commmon list functions and use them to automatically generate distractors for Parsons problems that use those statements. The findings presented in the pilot study indicate that the inclusion of distractors decreased problem solving efficiency but did not have a significant impact on score in two of the three problems presented. Including distractors caused a minimal increase in item discrimination and, in one case, caused the item to become less discriminating. Future work should further consider the impact of distractors on item statistics throughout a semster in order to identify if and when they are most effectively employed. Future work should also consider the role that including distractors in Parsons problems have on learning in formative assessments. Such findings would provide guidance on where and when automatically generated distractors can most effectively be deployed.

REFERENCES

[1] Mary J Allen and Wendy M Yen. 2001. *Introduction to measurement theory.* Waveland Press.
[2] Elizabeth L Bjork, Robert A Bjork, et al. 2011. Making things hard on yourself, but in a good way: Creating desirable difficulties to enhance learning. *Psychology and the real world: Essays illustrating fundamental contributions to society* 2, 59-68 (2011).
[3] Elizabeth Ligon Bjork, Jeri L Little, and Benjamin C Storm. 2014. Multiple-choice testing as a desirable difficulty in the classroom. *Journal of Applied Research in Memory and Cognition* 3, 3 (2014), 165–170.
[4] Derek C Briggs, Alicia C Alonzo, Cheryl Schwab, and Mark Wilson. 2006. Diagnostic assessment with ordered multiple-choice items. *Educational Assessment* 11, 1 (2006), 33–63.
[5] Paul Denny, Andrew Luxton-Reilly, and Beth Simon. 2008. Evaluating a new exam question: Parsons problems. In *Proceedings of the fourth international workshop*

on computing education research. 113–124.

[6] Yuemeng Du, Andrew Luxton-Reilly, and Paul Denny. 2020. A review of research on parsons problems. In *Proceedings of the Twenty-Second Australasian Computing Education Conference*. 195–202.

[7] Barbara Ericson, Austin McCall, and Kathryn Cunningham. 2019. Investigating the affect and effect of adaptive parsons problems. In *Proceedings of the 19th Koli Calling International Conference on Computing Education Research*. 1–10.

[8] Barbara J Ericson. 2016. Dynamically adaptive parsons problems. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*. 269–270.

[9] Barbara J Ericson, Mark J Guzdial, and Briana B Morrison. 2015. Analysis of interactive features designed to enhance learning in an ebook. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*. 169–178.

[10] Barbara J Ericson, Lauren E Margulieux, and Jochen Rick. 2017. Solving parsons problems versus fixing and writing code. In *Proceedings of the 17th koli calling international conference on computing education research*. 20–29.

[11] Barbara J Ericson, Kantwon Rogers, Miranda Parker, Briana Morrison, and Mark Guzdial. 2016. Identifying design principles for CS teacher Ebooks through design-based research. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*. 191–200.

[12] Max Fowler, David H Smith IV, Mohammed Hassan, Seth Poulsen, Matthew West, and Craig Zilles. 2022. Reevaluating the relationship between explaining, tracing, and writing skills in CS1 in a replication study. *Computer Science Education* (2022), 1–29.

[13] Stuart Garner. 2007. An Exploration of How a Technology-Facilitated Part-Complete Solution Method Supports the Learning of Computer Programming. *Issues in Informing Science & Information Technology* 4 (2007).

[14] Mark J Gierl, Okan Bulut, Qi Guo, and Xinxin Zhang. 2017. Developing, analyzing, and using distractors for multiple-choice tests in education: A comprehensive review. *Review of Educational Research* 87, 6 (2017), 1082–1116.

[15] Ibrahim Abou Halloun and David Hestenes. 1985. The initial knowledge state of college physics students. *American journal of Physics* 53, 11 (1985), 1043–1055.

[16] Kyle James Harms, Jason Chen, and Caitlin L Kelleher. 2016. Distractors in Parsons problems decrease learning efficiency for young novice programmers. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*. 241–250.

[17] Carl C Haynes and Barbara J Ericson. 2021. Problem-Solving Efficiency and Cognitive Load for Adaptive Parsons Problems vs. Writing the Equivalent Code. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–15.

[18] Petri Ihantola and Ville Karavirta. 2010. Open source widget for parson's puzzles. In *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education*. 302–302.

[19] Petri Ihantola and Ville Karavirta. 2011. Two-dimensional parson's puzzles: The concept, tools, and first observations. *Journal of Information Technology Education* 10, 2 (2011), 119–132.

[20] Chizuko Izawa. 1966. Reinforcement-test sequences in paired-associate learning. *Psychological Reports* 18, 3 (1966), 879–919.

[21] Raymond Lister, Tony Clear, Dennis J Bouvier, Paul Carter, Anna Eckerdal, Jana Jacková, Mike Lopez, Robert McCartney, Phil Robbins, Otto Seppälä, et al. 2010. Naturally occurring data as research instrument: analyzing examination responses to study the novice programmer. *ACM SIGCSE Bulletin* 41, 4 (2010), 156–173.

[22] Jeri L Little and Elizabeth Ligon Bjork. 2015. Optimizing multiple-choice tests as tools for learning. *Memory & Cognition* 43, 1 (2015), 14–26.

[23] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the fourth international workshop on computing education research*. 101–112.

[24] Susan Matlock-Hetzel. 1997. Basic Concepts in Item and Test Analysis. (1997).

[25] Thayer W McGahee and Julia Ball. 2009. How to read and really use an item analysis. *Nurse educator* 34, 4 (2009), 166–171.

[26] Briana B Morrison, Brian Dorn, and Mark Guzdial. 2014. Measuring cognitive load in introductory CS: adaptation of an instrument. In *Proceedings of the tenth annual conference on International computing education research*. 131–138.

[27] Mitchell J Nathan, Kenneth R Koedinger, Martha W Alibali, et al. 2001. Expert blind spot: When content knowledge eclipses pedagogical content knowledge. In *Proceedings of the third international conference on cognitive science*, Vol. 644648.

[28] Dale Parsons and Patricia Haden. 2006. Parson's programming puzzles: a fun and effective learning tool for first programming courses. In *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*. 157–163.

[29] Christopher A Rowland. 2014. The effect of testing versus restudy on retention: a meta-analytic review of the testing effect. *Psychological bulletin* 140, 6 (2014), 1432.

[30] Travis Scheponik, Enis Golaszewski, Geoffrey Herman, Spencer Offenberger, Linda Oliva, Peter AH Peterson, and Alan T Sherman. 2019. Investigating crowdsourcing to generate distractors for multiple-choice assessments. In *National Cyber Summit*. Springer, 185–201.

[31] D. H. Smith, IV. 2022. *DistractingParsonsGen - A Parsons Problem + Distractor Generator*. https://github.com/CoffeePoweredComputers/DistractingParsonsGen/

[32] Anne Venables, Grace Tan, and Raymond Lister. 2009. A closer look at tracing, explaining and code writing skills in the novice programmer. In *Proceedings of the fifth international workshop on Computing education research workshop*. 117–128.

[33] Lev Semenovich Vygotsky and Michael Cole. 1978. *Mind in society: Development of higher psychological processes*. Harvard university press.

[34] Nathaniel Weinman, Armando Fox, and Marti A Hearst. 2021. Improving instruction of programming patterns with faded parsons problems. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–4.

[35] Matthew West, Geoffrey L Herman, and Craig Zilles. 2015. Prairielearn: Mastery-based online problem solving with adaptive scoring and recommendations driven by machine learning. In *2015 ASEE Annual Conference & Exposition*. 26–1238.

[36] J Whalley, Tony Clear, and RF Lister. 2007. The many ways of the BRACElet project. *Bulletin of Applied Computing and Information Technology* (2007).

[37] Craig B Zilles, Matthew West, Geoffrey L Herman, and Timothy Bretl. 2019. Every University Should Have a Computer-Based Testing Facility.. In *CSEDU (1)*. 414–420.