Variability-Inducing Requirements for Programs: Increasing Solution Variability for Similarity Checking

Ashley Pang
Computer Science and Engineering
University of California, Riverside
Riverside, California, USA
apang024@ucr.edu

Frank Vahid
Computer Science and Engineering
University of California, Riverside
Riverside, California, USA
vahid@cs.ucr.edu
Also with zyBooks

ABSTRACT

Similarity checking is a common approach for detecting cheating in programming courses. A known limitation is high rates of similar pairs for programs lacking variability in possible solutions, especially for small programs. We experienced this issue in our CS1 course, where similarity checking in early weeks yielded many highly-similar pairs, many of which were not likely due to copying. Yet, we wish to catch copying students early, so that we can intervene and help those students avoid developing copying habits that may cause them trouble later. Our approach is to modify the program specifications to include variability-inducing requirements, namely places in the specifications where students make choices in their solutions, where different choices reduce the similarity scores. Those variability-inducing requirements are intentionally designed to avoid making the problem much harder for students. Examples of variability-inducing requirements include adding requirements to check for invalid input, or counting items. Such requirements have many different possible ways of implementing each. Essentially, variability-inducing requirements decrease the odds that two students would submit programs scored as highly-similar by a similarity checker, even for small programs. For 5 programs in our CS1 course, we added some variability-inducing requirements. Compared to an earlier term, the similarity checker's highly-similar-pairs rate dropped from 52% to 20% on average. Students' scores stayed the same from 98% to 96%, though time did increase from 18 min to 31 min on average. Adding such requirements helps instructors to do similarity detection and perform early interventions if desired.

CCS CONCEPTS

• Social and professional topics - Professional topics - Computing education - Computing education programs - Computer science education - CS1

KEYWORDS: CS1, similarity, variability, cheating, plagiarism

ITiCSE 2023, July 8–12, 2023, Turku, Finland

ACM ISBN 979-8-4007-0138-2/23/07. https://doi.org/10.1145/3587102.3588855

ACM Reference format:
Ashley Pang and Frank Vahid. 2023. Variability-Inducing Requirements for Programs: Increasing Solution Variability for Similarity Checking. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2023), July 8-12, 2023, Turku, Finland. ACM, New York, New York. USA.* 6 pages. https://doi.org/10.1145/3587102.3588855

1 INTRODUCTION

Similarity detection tools [1][2], [3], 4 help instructors detect cheating on programming assignments, aka labs. However, if a lab's solutions don't have much variability, then similarity due to copying is hard to distinguish from coincidental similarity, resulting in a long unuseful similarity list. Figure 1 provides an example list, where 490 solution pairs have high similarity scores above 9.0, using a MOSS-based [2] similarity checker that outputs values from 0-10.0. For labs with low-variability solutions, which often dominate early labs in a class, the long similarity lists may cause instructors to skip cheating detection for those labs. Unfortunately, skipping cheating detection can lead to students developing a habit of copying, which may get them in trouble later, such as doing poorly on exams or being caught cheating on later labs.

Pair	Student #1	Student #2	Similarity Score
1	A	B	10.0
2	A	C	10.0
...	...	...	...
490	...	...	9.1

Figure 1: Similarity detector output list, without added variability-inducing requirements, is quite lengthy.

Figure 2 provides an example scenario. In this example, assume a priori that Student B copied from Student A, with similar code highlighted, yielding the similarity score of 10.0 in Figure 1. However, assume a priori that Student C did not copy but coincidentally wrote a very similar solution, also yielding a 10.0 similarity score with A. Student D did not copy either, yielding a

5.2 similarity score with A. Labs with low-variability solutions have too many coincidentally-similar pairs like (A, C) that are hard for instructors to distinguish from pairs like (A, B). These "false positives" can create excessive work for instructors, who cannot determine whether students A, B, or C are copying from each other.

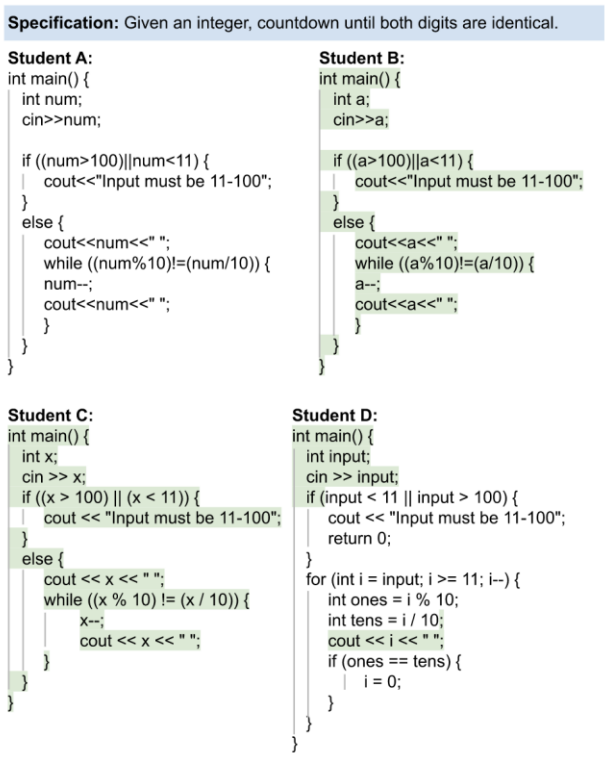

Figure 2: Code similarity for a lab, with no variability-inducing requirements introduced yet. The highlighted text is what the similarity checker considers similar compared to Student A; the checker outputs that Student B has 10.0 similarity, Student C has 10.0 similarity, and Student D has 5.2 similarity.

Many approaches to reducing the number of highly-similar students focus on improving the similarity detection tool itself, such as improving string-, graph-, or metric-based comparison [5], [6]. Some tools exclude small files to reduce false positives [7]. Some research [8] has examined how detection tools behave when students modify copied code. One approach aims to detect the original in a set of similar programs [9]. Some suggest looking beyond just code, to also consider comments and other features [10]. Many suggest going beyond similarity detection, such as requiring students to commit code and using machine learning to detect oddities [11], or similarly to allow resubmission for higher scores and detecting oddities in that history [12].

We focus on modifying the lab assignment itself to reduce coincidental similarity. Our approach introduces variability-inducing requirements to a lab: requirements that give students more implementation choices, yet don't make the program substantially harder. More solution flexibility yields fewer coincidentally-similar solutions, meaning the remaining similar pairs may be more likely to instances of copying.

This paper describes our efforts to introduce variability-inducing requirements in early week labs in a class. Our experiments showed that the number of highly-similar pairs of students decreased, while scores stayed about the same, though time spent did increase. Instructors can follow a similar procedure in their classes to enable more effective similarity checking on their labs too.

2 ADDING VARIABILITY-INDUCING REQUIREMENTS INTO OUR CS1 CLASS'S LABS

For years, we were frustrated by not being able to effectively use similarity checking in the early weeks of our introductory programming (CS1) class, due to excessively-long similarity lists. Our CS1 is offered every 10-week quarter at a large public state university. The class has 300-500 students (half computing majors, half in other science/engineering majors that require CS1), with two instructor-led 80-min lecture sessions and one teaching-assistant-led 110-minute lab session per week, in C++. The class had 329 students in Winter 2022 and 539 students in Fall 2022, which are the terms compared below. The class uses a zyBook [13], with weekly: before-lecture interactive readings having ~100 questions (Participation Activities or PAs), ~20 code reading or writing homework problems (Challenge Activities or CAs), and 5-8 weekly programming assignments (Lab Activities or LAs). All are auto-graded with auto-feedback, partial credit, and unlimited resubmissions. The course grade is typically 10% PAs, 10% CAs, 20% LAs, 5% class participation, and the remaining 50-60% from a midterm exam and final exam, taken in-person, half multiple-choice and half code-writing.

We examined our past CS1 offering from Winter 2022. Many labs in Weeks 1-5 had similarity lists so long that we could not cheat check those labs. Week 1 and 2 labs are relatively easy, covering input/output, variables, assignments, and math functions. So we focused on Weeks 3, 4, and 5, which covered Branches (3), While Loops (4), and For Loops / Strings (5). Those topics tend to be more challenging than in Weeks 1 and 2, and thus copying becomes more likely. Table 1 summarizes the 5 labs we chose. Labs 1 and 2 are from Week 3, Lab 3 from Week 4, and Labs 4 and 5 from Week 5.

Table 1: The 5 selected labs, and the added variability-inducing requirements.

Lab summary	Added requirements
Lab 1: Largest number: Output the largest number given three integers	Output the instances of the largest number
Lab 2: Leap year: Given year, write a function returning whether leap year	Output whether the year is also a century year (evenly divisible by 400)
Lab 3: Countdown until matching digits: Given an integer, countdown until both digits are identical	Output the distance from the start and end number
Lab 4: Count input length: Output number of characters excluding periods, exclamation marks, or question marks	Output the number of end-of-sentence punctuation characters found
Lab 5: Output inclusive/exclusive range: Given two numbers, output every number in the range	Based on one more input, include or exclude the high / low bounds in the range

The table also summarizes the variability-inducing requirements that we added to each lab, in our Fall 2022 CS1 offering.

For example, Lab 3's original requirements asked students to read an input number 11-100, and countdown until digits match, as in input 46 yielding 46 45 44. Some student solutions were shown earlier in Figure 2. In our roughly 300-student class, that lab had 490 pairs of students with a similarity score greater than 9.0, which is generally the threshold above which copying students may appear. Not only is that number of pairs too many for us to examine, but we usually could not determine cheating by looking at pairs because the similarity could have been coincidental. To induce variability in the solution, we added a requirement that the program also output the distance from the start to end numbers. This simple added requirement has various implementations. Figure 3 shows, via underlining, how Students A and C (from Figure 2) chose two different implementations of the new requirement.

Student A used an in-line arithmetic operation to output the distance between start and end numbers, whereas Student C initialized a counter and incremented the counter in the while loop. Those two solutions drop the similarity score from 10.0 to 8.8. Note that more possible solution approaches exist, such as incrementing using "count += 1" or "count = count + 1" or increasing the counter before decrementing in the while loop. Also, students could choose to calculate the distance in an intermediate variable before outputting.

Ideally, the additional variability-inducing requirements should not make the lab substantially more difficult. For Lab 3 above, students already had labs using arithmetic operations in cout statements, and already learned the concept of counters and how to increment by 1.

Specification: Given an integer, countdown until both digits are identical. Additionally, output the distance between start and end.

Student A:

```
int main() {
  int num;
  int startNum;
  cin >> num;
  startNum = num;

  if ((num>100)||num<11) {
    cout << "Input must be 11-100";
  }
  else {
    cout << num << " ";
    while ((num%10)!=(num/10)) {
      num--;
      cout<<num<<" ";
    }
    cout<<" "<<startNum-num;
  }
}
```

Student C:

```
int main() {
  int x;
  int count = 0;
  cin >> x;
  if ((x > 100) || (x < 11)) {
    cout << "Input must be 11-100";
  }
  else {
    cout << x << " ";
    while ((x % 10) != (x / 10)) {
      x--;
      count++;
      cout << x << " ";
    }
    cout << " " << count;
  }
}
```

Figure 3: Code solutions for Lab 3, with a variability-inducing requirement introduced. Student C, who did not copy from A, chose a different solution approach, receiving a similarity score of 8.8, dropping C's similarity with A below 9.0.

But now the program has many different solutions, and the odds of coincidentally-similar solutions is reduced. As such, the added requirement greatly reduces the size of the similarity list; for example, Figure 4 shows that the similarity list from Figure 1 was reduced from 490 pairs down to 103 pairs. Copying students would be easier to detect in that smaller list.

Pair	Student #1	Student #2	Similarity Score
1	A	B	10.0
2	B	X	10.0
...			
103	...	...	9.1

Figure 4: Similarity detector output list, with variability-inducing requirements added, is much shorter.

As another example, Lab 1 originally asked the student to output the max of three input numbers. We added the requirement that the student also output the number of times that the largest number appeared in those three numbers. That additional requirement logically is itself easier than the original problem, but can be done in different ways as seen in Figure 5. For example, a student could, at the program's end, use a counter to count how many times the largest value matched one of the inputs. Or they could count as the max was being determined. There are several other ways.

For most labs, we only introduced one new variability-inducing requirement, but Lab 5 also altered an existing requirement. Previously, the lab asked for a range in "increments of 10." However, to increase the number of possible solutions, we altered the range to increment by 1, allowing for increment operators such as "i++" and "++i" in addition to "i += 1" and "i = i + 1".

Specification: Output max of three numbers and output the number of instances of the largest number.

Student A:

```
int main() {
    int a, b, c, max;
    int count = 0;
    cin >> a >> b >> c;
    if (a >= b && a >= c){
        max = a;
    }
    else if (b >= a && b >= c){
        max = b;
    }
    else {
        max = c;
    }
    if (max == a) {
        count++;
    }
    if (max == b) {
        count++;
    }
    if (max == c) {
        count++;
    }
    cout << max << " "<< count;
}
```

Student C:

```
int main() {
    int a, b, c, max;
    int count = 1;
    cin >> a >> b >> c;
    if (a >= b && a >= c){
        max = a;
        if (a == b) { count++; }
        if (a == c) { count++; }
    }
    else if (b >= a && b >= c){
        max = b;
        if (b == a) { count++; }
        if (b == c) { count++; }
    }
    else {
        max = c;
        if (c == a) { count++; }
        if (c == b) { count++; }
    }
    cout << max << " "<< count;
}
```

Figure 5: Code solutions for Lab 1. Student A uses a counter at the end of the program, but student C increments the counter while checking for max. C has a similarity score of 8.1 vs A.

As with other labs, we added a new variability-inducing requirement in Lab 5 wherein a third input, which could be 0 or 1, would indicate whether the range would be inclusive or exclusive of its low/high bounds. Again, many solution approaches exist. Figure 6 shows two such approaches; Student A introduced new variables for the bounds, and set those variables according to the third input, whereas Student C modified the for loop's initialization by adding the fourth input, and modified the for loop's ending condition by subtracting the fourth input. More choices exist as well. The inclusive/exclusive approach adds a bit of difficulty but not much.

Specification: Output the given range. Include or exclude the low/high bounds based on a third input.

Student A:

```
int main() {
    int x, y, z;
    cin >> x >> y >> z;

    if (z == 1) {
        x += 1;
        y -= 1;
    }

    if (x <= y) {
        for(int i = x; i <= y; ++i) {
            cout << i << " ";
        }
    }
    else {
        cout << "Error: " << x << ">" << y;
    }
}
```

Student C:

```
int main() {
    int m, n, l;
    cin >> m >> n >> l;

    if (m <= n) {
        for(int i = m + l; i <=n - l; ++i) {
            cout << i << " ";
        }
    }
    else {
        cout << "Error: " << m << ">" << l;
    }
}
```

Figure 6: Code solutions for Lab 5. Student C has a similarity with A score of 8.5.

Labs 1, 2, 3, and 5 all used variations of an approach that adds simple requirements to the existing ones, where the added requirement has multiple implementation choices. Lab 4 was somewhat unique, in that we generalized an existing requirement. Previously, it asked to check for "periods, exclamation marks, and question marks", leading to nearly all students creating an expression with the checks in that same order: if (c == '.' || c == '!' || c == '?'). We generalized by merely asking for "end-of-sentence punctuation characters", such that students tended to use different orders. Figure 7 shows two examples provided in the lab specifications to clarify what "end-of-sentence punction characters" should be included. To avoid implying an order, the first example used an exclamation point followed by a period, while the second example used a question mark, then a period, then an exclamation point.

Ex: If the input is "Listen, Sam! Calm down. Please.", the output is:

```
28
3
```

Ex: If the input is "What time is it? Time to get a watch. O.K., bye now!", the output is:

```
43
5
```

Figure 7: Two example sentences for Lab 4.

3 RESULTS

Figure 8 provides similarity results of our adding variability-inducing requirements to labs, comparing the original 5 labs from Winter 2022 vs. those labs in the Fall 2022 term with the new requirements added. The number of pairs above 9.0 similarity, with 9.0 chosen from our past cheating investigation experience, dropped from an average of 687 pairs per lab to 135 pairs per lab, for an 80% reduction (p = 0.0056, using a two-sample equal-variance one-tailed t test).

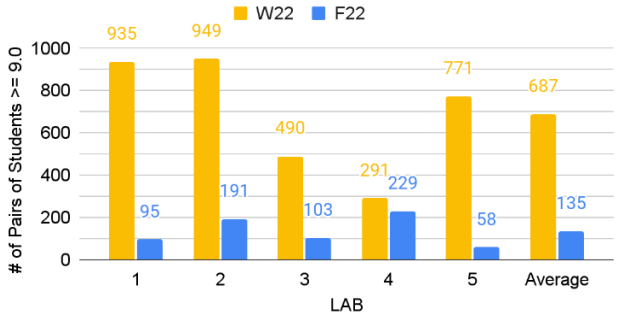

Figure 8: # of pairs of students with similarity score >= 9.0.

We have also begun doing cheat checking not just by seeking out the highest code pairs, but by seeking the students who have high similarity (above 9.0) with classmates. This is especially useful to focus our limited time on students who seem to be copying on many labs. Thus, Figure 9 shows the % of students on each lab who have at least one above-9.0 similarity with any other student. Whereas originally 65% of all Winter 2022 students were involved in a high-similarity pair, after adding the variability-inducing requirements, only 22% were found in Fall 2022 -- a 66% reduction

(p = 0.0005). Note: The figure only considers students who actually submitted the lab, as that is most proper in determining the %, though that value is close to the same as considering all students since most students did all five labs in both terms.

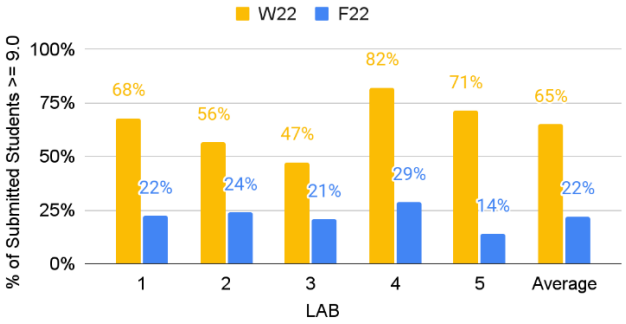

Figure 9: % of submitted students with similarity score >= 9.0.

To determine whether adding variability-inducing requirements made the labs harder for students, we examined the average lab scores and time spent. Figure 10 shows students receiving on average score of 98% in Winter 2022 and 96% in Fall 2022, with that small difference not being statistically significant (p = 0.174, using a two-sample unequal-variance one-tailed t test).

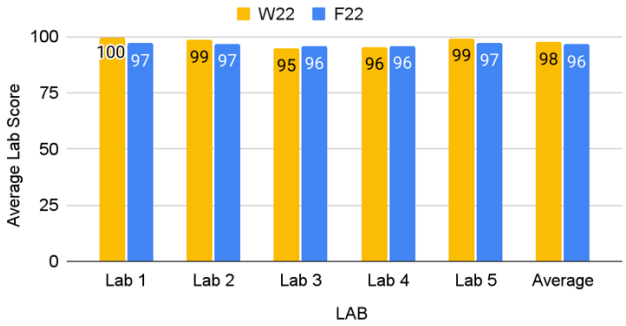

Figure 10: Average lab scores (%).

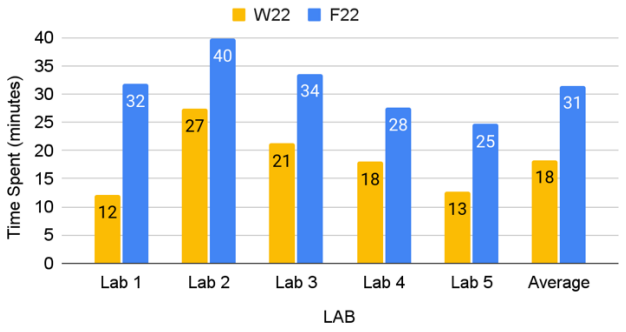

Figure 11: Average time spent (minutes).

Figure 11 shows students spending an average of 18 minutes in Winter 2022 vs 31 minutes in Fall 2022 (p = 0.0045, using a two-sample equal-variance one-tailed t test). Thus, lab scores stayed relatively the same, but time spent increased after introducing variability-inducing requirements. This makes sense, because the new labs involved some more work to implement the variability-inducing requirements, but that work wasn't substantially more complex, yielding increased time but the same scores.

4 HOW TO ADD VARIABILITY-INDUCING REQUIREMENTS

Based on our experiences, we developed some general guidelines for adding variability-inducing requirements into labs that otherwise may yield numerous coincidentally-similar pairs. The main technique we used to add the new requirements centered around adding requirements simpler than the main ones for that lab, such as:

- Adding a check for invalid input (divide by zero). Such checks can go in many different places.
- Asking for output to be formatted more cleanly, such as outputting a comma separated list but saying that the last number should not have a comma. Such formatting can be done in various ways using output statements in different places.
- Asking for complementary computations, such as asking not just for all numbers in a string but also for all non-numbers in a string. Many choices exist on how to get the complement, such as at the same time, or done separately afterwards.
- Adding conditions in a for loop that only apply to the first or last item, which can be done via different initializations, by branches in the loop, and more.
- Adding a counter or counting component, which can be done in many ways, either throughout the main solution, or near the end of the solution.

Another technique we discovered was to generalize how a problem was stated, so that students wouldn't all create the same ordering of items. The punctuation example above (Lab 4) was one example. Another was in how we wrote large equations; we can reformat them such that they look less like a program equation, so that there are many ways to convert the equation into a program.

Another approach, which we did not use, is to teach multiple styles. For example, an instructor can tell students that they can use i++ or ++i in for loops, and then intentionally switch between the two styles while teaching. Or, an instructor could teach that variables can optionally be declared on the same line, as in "int x, y". This can create even more variation in students' solutions. We did not use this approach because we like to keep things simpler for the students initially, but we notice students tend to use different styles from our class (often when copying from online solutions), and copying students are more easily detected when they use the same style yet that style has variations across the class.

5 DISCUSSION / THREATS TO VALIDITY

In both terms, we showed students the power of the similarity checker, with our goal being to deter cheating. But this could help some students learn how to beat a similarity checking. Of course,

those students might not appear on the high-similarity list, but this is an issue to consider in any work involving a similarity checker.

Fall 2022 had two instructors teaching the various sections, while Winter 2022 had only one of those instructors teaching all sections. We don't believe this influenced results, as the two instructors coordinated closely and basically taught the same class that term (with identical labs, exams, schedules, policies, etc.), and because all students were treated as one large class that term for purposes of similarity checking, but we mention the fact for completeness.

Fall 2022 experimented with a new late policy, allowing students to submit after a target date with a 1% per day penalty, up to 7 days late. This could have some impact on cheating by reducing pressure around deadlines, but likely not nearly as large of an impact as seen in the data presented above.

We only looked at labs in Weeks 3, 4, and 5. Ideally, this approach would be used in Week 2 as well, because we do end up finding some students cheating in Weeks 2 and even in Week 1, usually after catching them in a later week like Week 6, and then looking back at their earlier weeks.

Reducing high similarity pairs is a key goal of this work. But, a secondary benefit of the added requirements is that, even for pairs rated as highly similar by a similarity detection tool, the added requirements introduce some variability that an instructor might notice even if the similarity detection tool deems the code the same. This can help an instructor decide whether programs were copied or are coincidentally similar.

Ideally, we want to analytically quantify the impact that a variability-inducing requirement has on a set of possible solutions. Future work will be looking at actual solutions to a lab before and after adding variability-inducing requirements and determining the probability of each possible solution appearing. This work would help with the development of a variability score that suggests what score is recommended to minimize the possibility that two students had the exact same code by chance.

6 CONCLUSION

Similarity detection remains a central technique for detecting cheating in programming classes. Having found similarity detection weak for certain labs due to excessively long similarity lists, especially in early weeks of a course and smaller programs, we intentionally introduced variability-inducing requirements into our lab requirements for certain labs. Doing so reduced the list sizes by 80%, by reducing the likelihood of coincidental similarity. These smaller lists can then be checked for copying by instructors. We described techniques that instructors can use to introduce such requirements into their labs; more surely exist as well. Our goal ultimately is to catch copying students early enough that, instead of giving them an F for cheating across many weeks, we can apply a smaller penalty and correct their behavior so that they can ultimately succeed in the course.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 2111323.

REFERENCES

[1] Novak, M., Joy, M., & Kermek, D. (2019). Source-code similarity detection and detection tools used in academia: a systematic review. ACM Transactions on Computing Education (TOCE), 19(3), 1-37.

[2] Schleimer, S., Wilkerson, D.S. and Aiken, A., 2003, June. Winnowing: local algorithms for document fingerprinting. In Proceedings of the 2003 ACM SIGMOD international conference on Management of data (pp. 76-85).

[3] Software plagiarism detector, https://jplag.ipd.kit.edu/.

[4] Prechelt, L., Malpohl, G. and Philippsen, M. Finding plagiarisms among a set of programs with JPlag. Journal UCS, 8(11), 2002.

[5] Bellon, S., Koschke, R., Antoniol, G., Krinke, J., & Merlo, E. (2007). Comparison and evaluation of clone detection tools. IEEE Transactions on software engineering, 33(9), 577-591.

[6] Ducasse, S., Rieger M., and Demeyer S., "A Language Independent Approach for Detecting Duplicated Code," Proc. Int'l Conf. Software Maintenance (ICSM '99), 1999.

[7] Hage, J., Rademaker, P. and Van Vugt, N., 2010. A comparison of plagiarism detection tools. Utrecht University. Utrecht, The Netherlands, 28(1).

[8] Ragkhitwetsagul, C., Krinke, J. and Clark, D., 2018. A comparison of code similarity analysers. Empirical Software Engineering, 23(4), pp.2464-2519.

[9] Saoban, C. and Rimcharoen, S., 2019, July. Identifying an original copy of the source codes in programming assignments. In 2019 16th International Joint Conference on Computer Science and Software Engineering (JCSSE) (pp. 271-276). IEEE.

[10] Inoue, U. and Wada, S., 2012, May. Detecting plagiarisms in elementary programming courses. In 2012 9th International Conference on Fuzzy Systems and Knowledge Discovery (pp. 2308-2312). IEEE.

[11] Ljubovic, V. and Pajic, E., 2020. Plagiarism detection in computer programming using feature extraction from ultra-fine-grained repositories. IEEE Access, 8, pp.96505-96514.

[12] Tahaei, N. and Noelle, D.C., 2018, August. Automated plagiarism detection for computer programming exercises based on patterns of resubmission. In Proceedings of the 2018 ACM Conference on International Computing Education Research (pp. 178-186).

[13] zyBooks, http//www.zybooks.com, 2023.