Informatics in Education, 2026, Vol. 0, No. 0, 1–38

DOI: https://doi.org/10.15388/infedu.2507.010

Assessing Graphical Loop Invariant Based Programming Performance in a CS1 Course

Géraldine BRIEVEN *, [0000-0003-1410-1470], Benoit DONNET [0000-0002-0651-3398]

Montefiore Institute, Université de Liège, Belgium
e-mail: gbrieven@uliege.be, benoit.donnet@uliege.be

Abstract. We investigate the pedagogical impact of Graphical Loop Invariant Based Programming (GLIBP) in an introductory programming course. This approach encourages students to visually model the objects and variables handled in the loop, *before* implementing it. To evaluate the efficiency of this GLI model, a four-condition A/B/C/D test was conducted across two problems, with students receiving varying levels of scaffolding (from no support to a fully constructed GLI). Analysis of students' code showed that a well-designed GLI reduced errors related to the loop guard and the update of variables. However, many students struggled to understand or represent a GLI. The fill-in-the-blank GLI version, in particular, often added cognitive load rather than reducing it. Three recommendations emerged: train students to interpret a provided GLI when writing code; second, teach students to sketch their own model by recognizing similarities to previously solved problems; finally, guide students with questions to ensure all necessary variables and relationships are properly identified.

Keywords: Graphical Loop Invariant Based Programming, CS1, A/B Testing, Diagrammatic Reasoning, Student Difficulties.

1. Introduction

Implementing loops remains among the most challenging aspects for novice programmers (Cherenkova *et al.* (2014); Scapin and Mirolo (2019)). As main difficulties, Morazán (2020) points out the abstract reasoning required to correctly mutate state variables and terminate the loop. In this paper, we explore Graphical Loop Invariant Based Programming (GLIBP), designed to construct loops in a structured and logical way. GLIBP encourages students to graphically describe variable states through a Graphical Loop Invariant (GLI), before writing any code. This diagrammatic reasoning is intended to shift students' focus from trial-and-error coding toward more thoughtful and systematic solution analysis.

GLIBP draws inspiration from formal methods, particularly invariants, as introduced by Dijkstra (1976) and others. Here, these concepts are made more accessible by using informal, visual representations, referred to as a GLI. A GLI illustrates the objects and the variables involved in a loop solution, as well as the relationships between them. The GLI

*Corresponding author.

must hold at each evaluation of the loop guard: before entering the loop (Initial State), when exiting the loop (Final State), and before and after each iteration of the loop body (In-loop State). During the execution of the loop body, intermediate results are refined and the GLI may temporarily become false. However, subsequent updates within the same iteration must restore the GLI while making progress toward loop termination. Once the GLI is successfully restored and progress is ensured, the loop body is complete.

In practice, Walker (2023) and our own observations suggest that GLIBP introduces a notable learning curve. Students often struggle to construct meaningful representations or interpret instructor-provided ones. It raises questions about its practical effectiveness in introductory programming (CS1) settings. To address them, we designed a controlled classroom experiment to evaluate the effectiveness and limitations of GLIBP. Our study is guided by three research questions (RQs):

RQ 1: Does the GLI help students in finding a solution to a loop-based problem?
RQ 2: Does GLIBP improve students' code correctness?
RQ 3: Are there errors students can avoid in their code thanks to GLIBP?

To answer them, we conducted a two-session experiment with first-year computer science (CS) students who had prior training in our GLIBP approach (see Section 4.2). Participants were divided into four groups (A/B/C/D) and tasked with solving two problems using a loop. Each group received a different level of scaffolding: Group A (the control group) did not get any GLI support; Group B worked with a fill-in-the-blank GLI (i.e., a GLI with boxes to complete with variables or text description); Group C created the GLI themselves; and Group D was given a completed GLI, designed by instructors. Students' code and GLI were analyzed using multiple metrics, including code status (e.g., correctness, completeness), types of semantic errors, and GLI quality (general and specific states). We also collected student perceptions via an anonymous survey.

The results show that providing students with a completed GLI (Group D) helped them avoid semantic mistakes, such as incorrectly initializing variables, misidentifying the loop guard, or using the same variable for multiple roles. However, students in Group B (who received a fill-in-the-blank GLI) often struggled to understand or complete the diagram. Their open-ended responses suggest that this format may impose extra cognitive burden rather than provide useful guidance. Students from Group C showed some benefit in constructing their own model, but many students lacked accuracy. Finally, Group A generally struggled more with code correctness and made a broader range of errors compared to the experimental groups. Our study suggests that, while GLIBP can enhance students' reasoning about loops, its instructional design must be carefully aligned with learners' cognitive readiness and the problem to solve. Section 8 discusses how GLI scaffolding could better support novices in developing robust loop logic and deeper conceptual understanding.

2. Related Work

2.1. *Teaching Loops in CS1*

Scapin and Mirolo (2019) recommend loop invariants as a methodological tool adaptable to less formal learning styles. Besides loop invariants, Gomes *et al.* (2019) explore

Pedagogical Approach	**Source**	**Structural View (variables and relations)**			**Operational View (code)**	
		Describing Variables' Roles	Determining Loop Guard	Initializing Variables	Updating Variables	Distinguishing In-Loop and Outside-Loop
Graphical Loop Invariant Based Programming	Brieven *et al.* (2023), This work (Figure 1)	Description of variable content (✓)	FINAL STATE representation (✓)	INITIAL STATE representation (✓)	Variables' update to reflect state mutation (✓)	Solution parts mapping to each zone in the code (✓)
Learning recursion before loops	Morazán (2020)	Notion of state variables (⌣)	Termination argument (✓)	Argument value passed to the first recursive call (✓)	Accumulator update in the recursive call (✓)	(✗)
Pattern-based instruction (including worked examples)	Fernández Alemán and Oufaska (2010); Iyer and Zilles (2021)	(✗)	Dedicated field (✓)			(✗)
Block-based environment (Scratch etc.)	Grover and Basu (2017); Cetin (2020)	(✗)	Dedicated block (✓)			Block shape (✓)
Visual representation through cards	Gomes *et al.* (2019)	(✗)	Dedicated card (✓)			Card shape (✓)
Visualization for simulation	Zhang *et al.* (2013)	(✗)	Visualizing loop exit at the end of the simulation (⌣)	Visualizing initial value(s) (⌣)	Tracking value changes along iterations (✓)	Visualizing what is repeated or not (✓)

Table 1
The five challenges when teaching loops, in relation to different pedagogical approaches in CS1. (✓) indicates the approach explicitly addresses the challenge; (✗) indicates it does not; (⌣) indicates implicit or partial support.

other pedagogical approaches in the literature, which are presented in Table 1. The table also lists the challenges students face in mastering loops. Five challenges were selected from the work of Grover and Basu (2017) and slightly reformulated based on their occurrence in the papers referenced in the "Source" column. In this way, we could connect them to the listed pedagogical approaches. Among the challenges, we distinguish between *describing variables' roles* and *updating variables*. On the one hand, students must precisely know the content of variables at a given iteration, and on the other hand, they must correctly update them to make progress and converge towards termination. This differentiation is emphasized by categorizing the five challenges into structural versus operational views, following Sfard (1991)'s dual perspective that shapes GLIBP, as described in Subsection 2.3.

All approaches in Table 1 decompose a loop into key building blocks (initialization, loop guard, and variable updates), which can be associated to the subgoal learning framework presented by Morrison *et al.* (2015). Visualization can support it by playing a structural role. It is employed in all the approaches except *Learning recursion before loop* and, to a lower extent, *Pattern-based instruction*. With respect to the five challenges, the added value of GLIBP is that it also requires students to clearly identify the role of each variable and describe the relationships with each other. By benefiting from this deeper structural understanding, we expect students to implement correct loops, with limited trial and error.

2.2. *(Graphical) Loop Invariant*

While extensive research (including Cormen *et al.* (2009); Broy *et al.* (2024)) exists on invariants for code verification, their application to program construction has received limited attention. In the ACM report from Kumar *et al.* (2024), invariants are categorized in "Algorithmic Correctness" and not as a construction aid. While Graphical Loop Invariant Based Programming (GLIBP) is related to problem solving, abstraction, and algorithmic and mathematical skills, previous systematic literature reviews about introductory programming (Rodrigues *et al.* (2022); Luxton-Reilly *et al.* (2018)) do not mention it. We chose to promote GLIBP in our course because of its strong connection to Formal Methods (FM). Dongol *et al.* (2024) argue that FM thinking is a mindset computer scientists should develop from their first day of study. Kamburjan and Grätz (2021); Zhumagambetov (2021); Broy *et al.* (2024) have also highlighted the importance of FM in education, given the increasing use of FM in industry and the persistent problem of buggy programs. Fowler *et al.* (2021) also documented computer science graduates' difficulties in using loop invariants effectively to understand and debug algorithms. GLIBP aims to address this gap by familiarizing students with structural thinking early in their education.

The foundational work on invariant-based programming was grounded in Hoare's logic (Hoare (1969)) and established by Dijkstra (1976), followed by significant contributions like Gries (1987) and Morgan (1990). They represent invariants as logical assertions.

A few years later, several educational approaches have emerged to teach invariant-based programming. Tam (1992) advocates introducing students to invariants early in programming courses and provides examples of code construction using informal invariants expressed in natural language. Astrachan (1991) suggests using graphical loop invariants (GLI) in introductory computer science courses (CS1/CS2), though specific states are not represented, contrary to our approach. To our best knowledge, Astrachan is the first researcher using diagrammatic reasoning (defined by Anderson *et al.* (2002)) to make loop invariant more accessible. More generally, visualization has been acknowledged as an appropriate tool in introductory programming for topics like loops (Cetin (2020)), role of variables (Al-Barakati and Al-Aama (2009)), sorting algorithms etc. (Luxton-Reilly *et al.* (2018)). Back (2009) introduces nested diagrams resembling state charts that simultaneously represent both the invariant and the code structure. However, their approach still relies on logical assertions for invariant expression. More recently, Mannila (2010) developed an invariant-based programming approach (IBP) that aligns more closely with our **GL**IBP methodology. Their approach targets novice programmers through visual program construction designed to minimize notational overhead. Like our approach, they represent invariant states visually. The key distinction is that we employ textual notations while their method requires students to write predicates within diagrams. Mannila (2010) also conducted empirical studies, as we do in this paper, on common student errors in code and invariant construction, including variable updates, guard loops, infinite loops, incomplete diagrams, and missing relationships in GLIs. Eriksson *et al.* (2018) propose a pictorial language specifically for array invariants, combining visual elements (drawn data structures with colored partitions representing universally quantified predicates) with formal

language components (predicates expressing partition meanings). At Seton Hall University, Morazán (2020) describes how students are taught to use loop invariants to code loops. However, according to their approach, students should first think recursively. Therefore we refer to it in Table 1 as a distinct approach.

Despite limited adoption, emerging research led by Walker (1998) and Brieven *et al.* (2024) examines student attention patterns when working with GLIs and code. A study from Walker (2023) indicates that students typically focus on code-level solutions rather than using GLI for problem-solving, suggesting a gap between the intended pedagogical approach and actual student practice. Our study builds upon these primary observations and raises some recommendations.

2.3. *Theoretical Framework underlying GLIBP*

In addition to the diagrammatic reasoning theory, GLIBP is underpinned by Sfard (1991)'s operational and structural conceptions. Originally rooted in Mathematics, Mirolo *et al.* (2022) later extended these dual conceptions to Computer Science, using the concept of *iteration* as an illustrative example. GLIBP concretizes that example, as it connects:

- a (graphical) loop invariant (that can be assimilated to a *structural* conception, where variables and objects' state are described with respect to each other, thereby forming a programming construct)
- a code snippet (that can be assimilated to an *operational* conception, where variables and objects are altered across the repetition of instructions)

The link with such conceptions consistently shows the relation of GLIBP to formal methods (grounded in Mathematics).

3. Description of our Graphical Loop Invariant Based Programming approach

In this section, we describe the GLIBP approach taught in our CS1 course. It applies when students need to implement a loop and relies on an informal version of the invariant: the *Graphical Loop Invariant* (GLI). It is described in detail by Brieven *et al.* (2023). The GLI must depict the object(s) being iterated over (an array, a range of integers, etc.), the variables involved in the loop, their domain, and the relationships among them, maintained across all iterations. Fig. 1 shows that a GLI includes a general state, instantiated into initial and final states. These three diagrams form a structural view of the solution, similar to what is taught by Walker (2023). He also asks students to represent invariants to solve loop-based problems and then instantiate it into the initial, in-loop, and final states. Next, as illustrated in Fig. 1, students derive the variable initialization, the loop guard and the loop body. Additionally, they must provide a *loop variant* function, ensuring the loop terminates. In this experiment, we omitted this step, as it is more theoretical in nature and related to the exit condition that students are already specifying.

As a concrete example, Fig. 2 shows the GLI for the problem of computing the product of all integers in `[a, b]`. It models the iteration over all integers from `a` to `b`. A vertical red bar (called the *dividing line*) divides the integer line into two regions, representing the

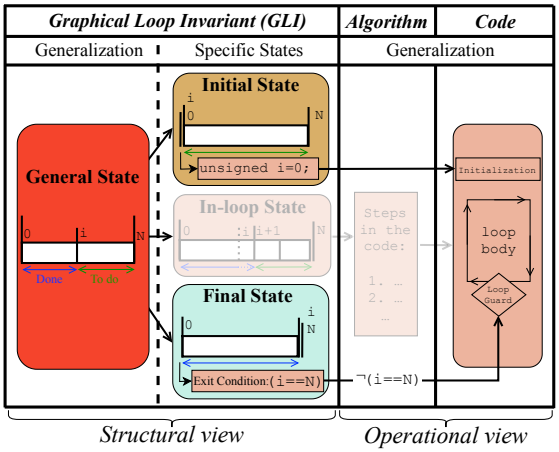


Figure 1. Solution parts of the Graphical Loop Invariant Based Programming approach. Neither the In-loop State nor the steps of the program are explicitly required from students, which is why they are blurred.

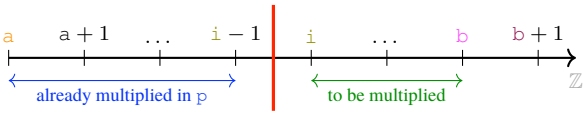


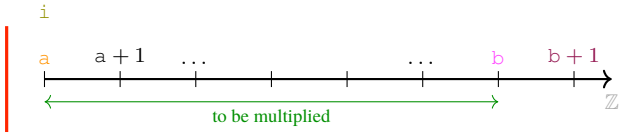
Figure 2. GLI for the product of all integers in [a, b].

solution state after a given number of iterations. The left blue region contains the integers that were already multiplied in a variable p (storing intermediate results). The right green region covers the integers that still have to be multiplied. The integer immediately to the right of the dividing line is labeled i. It serves as the iterator variable. At that step, the GLI contains the variables involved in the code (a, b, i, and p) and suggests their type (int in this example, as the graduated line is labeled with $\mathbb{Z}$). To derive the loop guard and variable initialization, we instantiate the general form of the GLI into specific states.

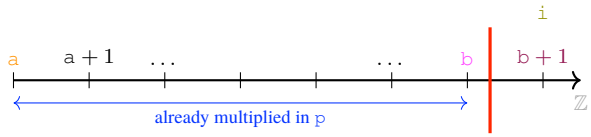
Fig. 3a illustrates the Initial State (i.e., before entering the loop). It is obtained by moving the dividing line to its first position, aligning variable i (attached to the line) to its minimum value (i.e., a) and causing the blue zone to disappear. This means no product has been computed yet, and the initial value of p should be the empty product, i.e., 1.

To determine the exit condition, we represent the Final State of the loop by shifting the dividing line to the right until the green zone has been fully covered. As shown in Fig. 3b, the loop's goal is reached when $i == b+1$ and p contains the product of all integers between a and b. The exit condition is therefore $i == b+1$, whose logical negation yields the loop guard $i \neq b+1$.

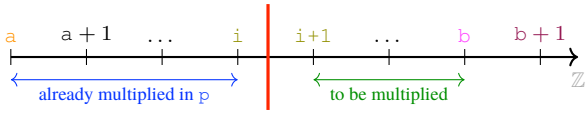
The loop body must contain the instructions that will allow the program to make progress towards the goal, knowing that the loop guard and the GLI are true. Since the blue zone covers the integers already multiplied in p (from a to $i-1$), we can expand it by multiplying p by the next integer standing to the right of the dividing line (i). The



(a) INITIAL STATE where the initial values are highlighted.



(b) FINAL STATE where the exit condition is highlighted.



(c) IN-LOOP STATE after having multiplied p by i.

Figure 3. Manipulating the GLI for deducing INITIAL STATE, IN-LOOP STATE, and FINAL STATE for computing the product of integers between a and b. The corresponding GLI (GENERAL STATE) is provided in Figure 2.

resulting situation is depicted in Fig. 3c. At this stage, to restore the GLI, i+1 should be assigned to i (i.e., i must be incremented).

4. Method

4.1. *Teaching Graphical Loop Invariant Based Programming in our CS1 Course*

Our course emphasizes GLIBP, using C as the programming language. The learning goals of GLIBP are to teach how to construct correct loops and prepare students for formal methods. All the course activities supporting this approach are illustrated in Fig. 4. They span roughly a third of the course schedule. Additionally, students practice GLIBP on our learning platform (Brieven *et al.* (2024)) through three homework assignments. On this platform, they complete fill-in-the-blank GLIs (see examples in Fig. 20 and Fig. 25). By shaping the expected solution this way, we can anticipate students' responses and provide personalized automated feedback (Brieven *et al.* (2025)). However, in the midterm and final exam, students are expected to follow the GLIBP approach without any scaffolding. Fig. 4 highlights these two summative assessments and shows that the exam served to measure the baseline knowledge of participants before the two experimental sessions. The two sessions were conducted during the CS2 course and introduced to students as an opportunity to refresh GLIBP, as it provides foundations for the upcoming project ("Program

Construction"), which relies on invariants. During the experimental sessions, instructors minimized their interventions to avoid influencing students' solutions.

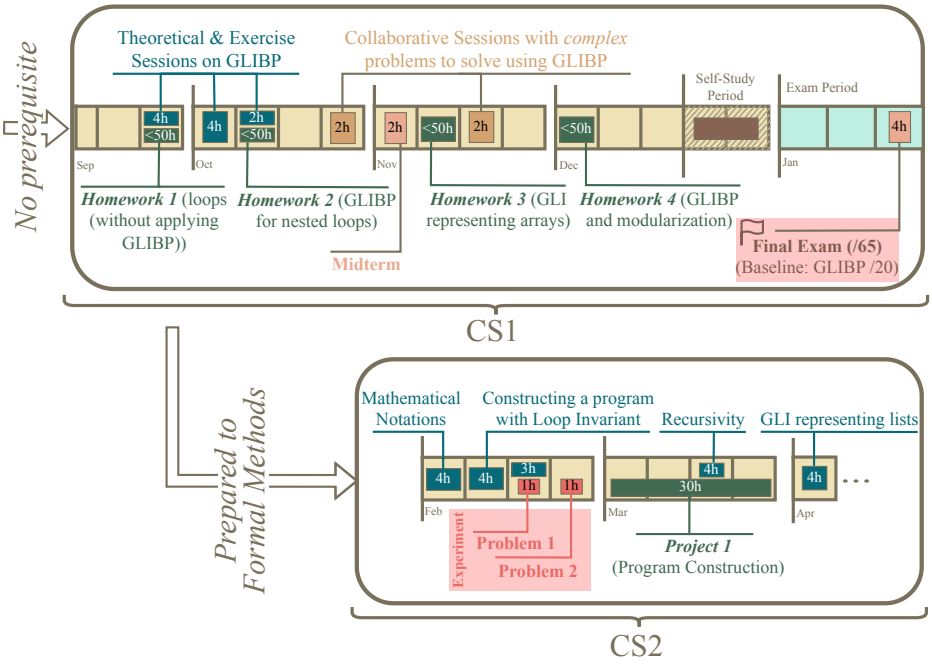


Figure 4. Timeline of the CS1 and CS2 courses. Only the content related to GLIBP is presented.

Fig. 4 also shows that the GLIBP approach requires no prerequisites, although students with stronger abstraction and pattern recognition skills may find GLIBP more accessible. The advantages for instructors adopting this approach include: (i) making easier the teaching of loop invariants later in the curriculum (involved in program verification) and (ii) establishing a unified conceptual framework for loops. However, potential challenges include: (i) the additional instructor workload to get familiar with the approach and (ii) the need to sustain student motivation, as students often see the payoff from this work only after several weeks or months, like expressed by Walker (2023). Brieven *et al.* (2023) presents how GLIBP can be taught in CS1.

4.2. *Measuring the Impact of GLIBP on Students' Code*

To assess the impact of GLIBP on programming performance, we conducted a two-session classroom experiment. In each session, students were split into four groups and asked to solve a loop-based programming problem within one hour.

4.2.1. *Two Problem Statements*

To select the problem statements for this experiment, we defined the following criteria:

1. The solution should rely on **one loop**.
2. The solution can be found in **one hour**.
3. The solution can be implemented in **20 lines** of code.
4. The solution should involve **objects that are familiar to students** (e.g., arrays etc.).
5. The solution should depend on **different zones** in the object(s).

The final two criteria are introduced to emphasize the relevance of the GLIBP approach. The GLI should assist in identifying which variables correspond to which zones, how the variables are updated, and how control flow can determine which zone should be processed at each iteration. Based on these criteria, we selected the two following problems:

1. *Compressing an array* (detailed in Appendix A.1). Students should detect consecutive same values in a given array and express them in another array as the length of the sequence followed by the redundant value in this sequence. And a single instance is just copied in the array holding compression.
2. *Printing a calendar in the terminal* (detailed in Appendix A.2). Based on the number of days in the month and the day of the week the month starts, students must display a monthly calendar in the terminal. This problem is inspired by Sooriamurthi (2009).

The relevance of these problems with respect to the criteria is validated in Appendix B.

4.2.2. *Groups of Students*

Participants were divided into four groups, each with a different level of scaffolding:

- *Group A (control)*: No scaffolding was provided. Students can solve the problem using any method, including GLIBP, without being explicitly encouraged to do so.
- *Group B (experimental)*: A fill-in-the-blank GLI (see Fig. 20, for instance) was provided. Students were asked to complete it, derive the Initial and Final States, and code the solution, like they are used to in the formative assessments (i.e., homework).
- *Group C (experimental)*: Students were asked to construct the GLI from scratch, and instantiate it into the Initial and Final States to code the solution, as asked in the summative assessments (i.e., midterm and final exam).
- *Group D (experimental)*: A completed GLI (see Fig. 17, Fig. 18 and Fig. 19, for instance) is provided. Students are asked to extract relevant information (variable initialization and exit condition of the loop) and write the code.

This design allowed us to measure the influence of GLI-related scaffolding both in comprehension (Group D), guided construction (Group B), and independent construction (Group C), relative to an unscaffolded baseline (Group A). The GLIs given to students were designed by one instructor and reviewed by the professor of the course as well as two undergraduate teaching assistants.

As in the summative assessments, students were required to write their code on a sheet of paper (as well as their GLI). The goal is to assess students' ability to correctly code, without any influence from test results. Moreover, this choice guaranteed that they were not using any external tools (LLMs, Copilot, etc.) to develop their solution.

To ensure comparable group composition, students were ranked according to their performance on the exam question involving GLIBP. They were then assigned to the four

groups in a round-robin manner, such that successive students in the ranking were allocated to different groups, ensuring an approximately balanced distribution of prior performance across groups. Groups were rotated between sessions to minimize bias due to individual characteristics (e.g., Group A in Session 1 became Group B in Session 2, and so on). This is illustrated in Fig. 5. Since the experiment was not mandatory, a small number of students (from 2 to 4) per group were not present in the second session. And all the students taking part in the second session also took part in the first session.

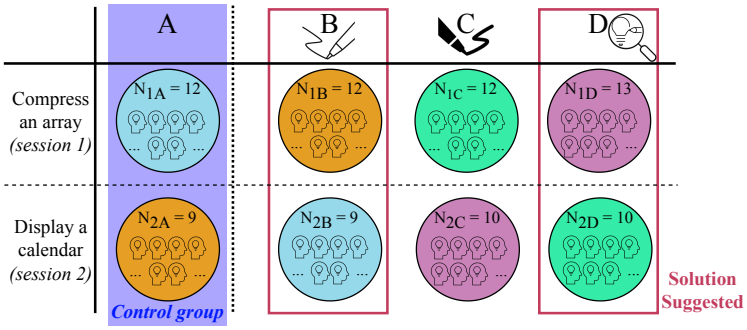


Figure 5. Experimental sessions (with one specific problem per session) and groups. The number refers to the problem (or session) while the letter refers to the group.

To capture the actual baseline knowledge of participants in each group, Appendix C presents the distribution of students' performance in the final exam that students took three weeks earlier, in each group. It only considers the grades of the GLIBP-related question.

4.3. *Data Collection*

In addition to collecting students' code, GLI and draft, we also asked students to fill in a survey. It was anonymous, students were only asked to specify the group they belonged to. Table 2 presents the survey questions.

Exact Survey Questions	Format
"To what extent did the problem sound complicated for you?"	5-point Likert anchors
"To what extent was the GLI useful to solve the problem?"	5-point Likert anchors
"Was there anything that blocked you in solving the problem?"	Open-ended question

Table 2
Exact questions asked in the survey.

The questions were developed by the teaching assistant. They were then reviewed by the course professor. Given that the survey serves an exploratory purpose to start answering RQ1, we opted for a brief instrument with one question per construct.

The metrics characterizing students' code and GLI are summarized in Fig. 6 and described in the two next sections.

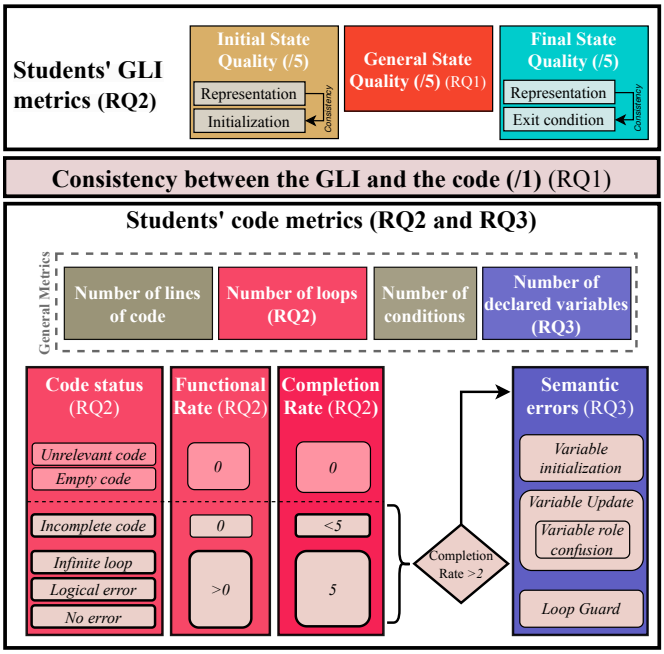


Figure 6. The metrics characterizing students' code and GLI, as well as the consistency between the two.

4.4. *Metrics related to Students' Code*

Students' code was carefully and manually characterized, based on these features:

- **Number of lines.** This count was performed considering one instruction per line.
- **Number of loops, conditions in the loop(s), and declared variables.** For the count of conditions, an occurrence of `"if ...else"` was considered as two conditions.
- **Code status.** The possible statuses were tuned over the code snippets being reviewed. Initially, four categories were defined: *Empty code* (no code is provided), *run-time error* (no output is produced as program execution is halted due to some failed operation, e.g., infinite loop), *Logical error* (test fails as the output is somehow incorrect) and *No error* (the code passes all unit tests, designed by the instructor). Over the analysis of students' code, we broke down *run-time errors* into three subcategories: *Infinite loop* (when the loop does not terminate), *Unrelevant code* (when the code does not respond to the problem statement - some students were typically redeclaring and/or initializing the input in a `main()` rather than implementing the core of the function), *Incomplete code* (when the code misses essential instructions, e.g., loop guard, or `printf()` calls (while the problem is about displaying something)). Since students coded on paper, minor syntax errors (e.g., missing semicolons) were corrected without penalty, as the assessment targeted semantics and code structure rather than syntax.
- **Functional rate.** For code that could be executed, a functional rate ranging from 0 to 5 was computed, based on the number of unit tests that passed.

- **Completion rate.** Five key instruction blocks were each counted as one unit in the completion rate (ranging from 0 to 5): variable initialization, loop guard, iterative variable update, intermediate result update[2], and cases in the loop (in both problem statement, three conditional branches were needed).
- **Semantic errors.** Semantic errors were identified only for code snippets that were "complete enough", as shown in Fig. 6. Otherwise, incomplete code could appear deceptively correct because fewer errors arise, whereas students were not even exposed to some errors. We defined the following categories of errors: *Variable Initialization*, *Loop Guard* and *Variable updates* (including *Variable role confusion*[3]). They match with four of the five challenges presented in Table 1. We did not consider *Distinguishing In-Loop and Outside-Loop* because no student made any mistake on that.

These features and possible values were predefined and tuned across a first analysis made by one teaching assistant with four years of experience with the course. Initially, only the functional rate and semantic errors were tracked to address RQ2 and RQ3. However, during the first analysis, a large proportion of students' code appeared insignificant with respect to these two metrics, often due to incompleteness. This led us to first categorize students' code more broadly, according to a code status. Their distribution is discussed as a first step toward answering RQ2. Then, in order to still extract semantic errors from incomplete students' code, a completion rate was defined, and the code snippets considered *complete enough* were analyzed to address RQ3. Students' code was reviewed multiple times, by the same teaching assistant. To validate the manual coding, two solutions per group per session were randomly selected and ultimately reviewed by the course professor, yielding 16 code snippets and their associated diagrams for analysis. Full agreement (100%) was obtained across all metrics (summarized in Fig. 6), which substantiates the consistency of the values obtained after multiple reviews. However, relying on a single coder remains a limitation.

4.5. *Assessing the GLI Quality*

Besides the code, the GLI (general and specific states) was also rated, by the same teaching assistant, based on a score ranging from -1 to 5. -1 means that students were not providing any representation. The GLI (General State) was rated based on the five criteria highlighted in Fig. 7. These criteria are also used to grade exams, meaning they had already been refined for accuracy and clarity over several years of use. Each one accounts for one point in the score. On the right, the figure applies the criteria to the GLI developed for the second problem.

The scoring of the quality of the Initial and Final States is illustrated in Fig. 8.

In addition to the quality of the three GLI representations, a boolean metric was defined to specifically reflect their whole consistency with respect to the code (not just the variable initialization and loop guard). If an instruction was not consistent with one of the three GLI representations, the boolean metric was turned to false.

[2] Writing values into the result array (Problem 1) or dates into the calendar (Problem 2).

[3] Any code falling in the class *Variable role confusion* was also marked in the broader *Variable updates* category.

Criteria	Point
Object representation and name	/1
Bounds	/1
Indexes and zones	/1
Expressions holding current result	/1
Text description of what has been performed so far	/1

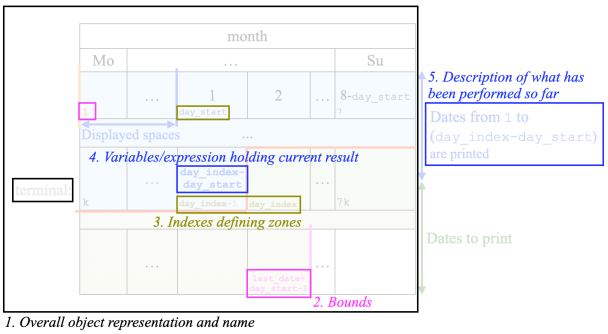


Figure 7. The five criteria to rate the GLI (General State) (applicable to Groups B and C). The right part of the figure applies the criteria to the second problem.

	Criteria for the initial (resp. final) state of the GLI	Point (Groups B and C)	Point (Group D)
Diagram State	Position of the Dividing Lines	/1	Not applicable (provided)
Diagram State	Position of the variables	/1,5	
Code	Correct initial value (resp. exit condition)	/1,5	/2,5
	Consistency between the two	/1	/2,5

Figure 8. The four criteria to rate the Initial and Final States (applicable to Groups B, C and D).

5. Results

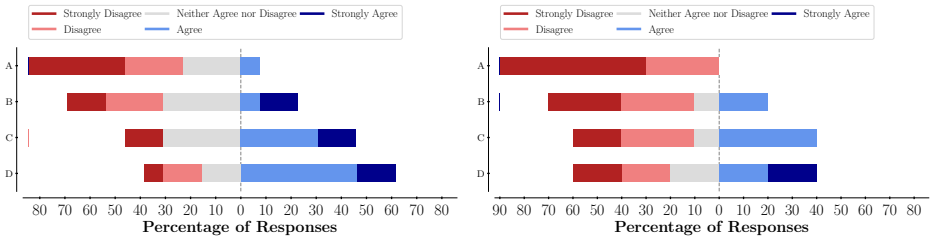
Equipped with the metrics defined above, we first study the utility of the GLI, based on its quality score and student perceptions (RQ1). Then, we investigate the relationship between diagrams' quality and code correctness (RQ2). And finally, we identify error types in student code that GLIBP may help prevent (RQ3). Section 6 discusses the findings.

5.1. *RQ1: Does the GLI help students find a solution to a loop-based problem?*

To answer this question, we first analyze student perceptions through Fig. 9. It illustrates that, in Group A, only one student spontaneously applied GLIBP, across both problems[4]. This validates Group A as an appropriate control group. Notably, 69% of students in this group preferred to either code immediately or rely on examples.

In the experimental groups, both Fig. 9 and Table 3 show that students provided with a correct GLI (Group D) used it the most. However, six students from this group reported being unable to use it effectively. Half of them felt blocked because of some specific elements of the given GLI. For example, for the second problem (see Fig. 22), two students

[4]This student was excluded from the control group to avoid bias in the subsequent research questions.



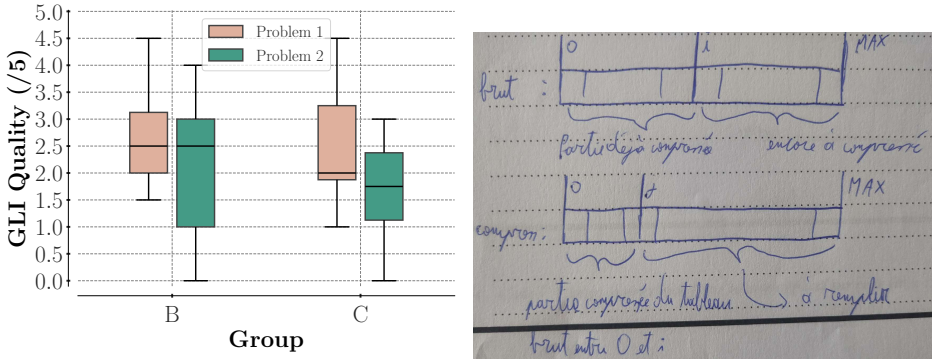
(a) Problem 1 (Compressing an array). (b) Problem 2 (Drawing a calendar).

Figure 9. The claim: *"To what extent was the GLI useful to solve the problem?"*

	Group B	Group C	Group D
Problem 1	40%	60%	85%
Problem 2	50%	62%	67%

Table 3
Percentage of students from the experimental groups who are consistent between their GLI (general and specific states) and their code.

reported they could not understand the purpose of the `k` variable. Both Fig. 9 and Table 3 also illustrate that few students from Group B were actually using the GLI to code their solution. In Problem 1, approximately one-third of students found the fill-in-the-blank GLI confusing or hard to understand.



(a) GLI (General State) quality score (see Fig. 7 for details on the five criteria to compute that score). (b) Example of GLI (General State) drawn by a student (from Group C) to solve Problem 1.

Figure 10. Quality of the GLI (General State).

To go further, Fig. 10a compares the quality of the GLI (General State) between Groups B and C. The quality appears limited (with a median equal to 2.5 out of 5 in Group B, and 2 in Group C). Scores are a bit lower in Group C as students were exposed

to more errors/inaccuracies. For example, to compress an array (Problem 1), 7 out of 12 students represented only the input array, omitting the array storing the result. Moreover, among these 12 students, 7 depicted only two zones within the input array (as illustrated in Fig. 10b), whereas four zones were expected (see Fig. 17). To draw a calendar (Problem 2), three students (out of 10) were just representing a number line. While not incorrect, the solution omits two key aspects of the problem: printing of the initial spaces and detecting line breaks based on the current day's index. These results are discussed in Subsection 6.1.

5.2. *RQ2: Is GLIBP associated with students' code correctness?*

Having established that students find the GLI useful to varying degrees, we now examine whether a well-constructed GLI leads to correct code.

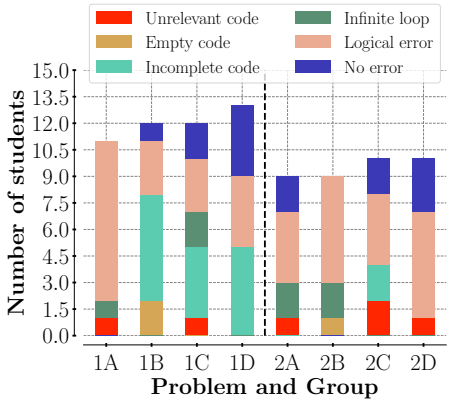


Figure 11. Code statuses in each group and session (see Sec. 4.3 for more details about each status).

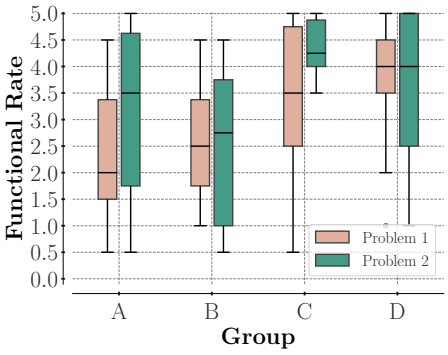


Figure 12. Functional rates of students' codes (that are complete).

First, Fig. 11 shows the distribution of code statuses for each problem and group. Group D outperformed the other groups, with seven students producing error-free code across both problems. *Infinite loop* occurred at least once per session in Group A, while remaining absent in Group D. This contrast suggests an association between GLIBP and greater code correctness. Finally, we can see that, in Problem 1, many students in the experimental groups did not complete their code (with a median completion score of 3 out of 5). Incompleteness was mostly due to some conditional branches missing in the loop.

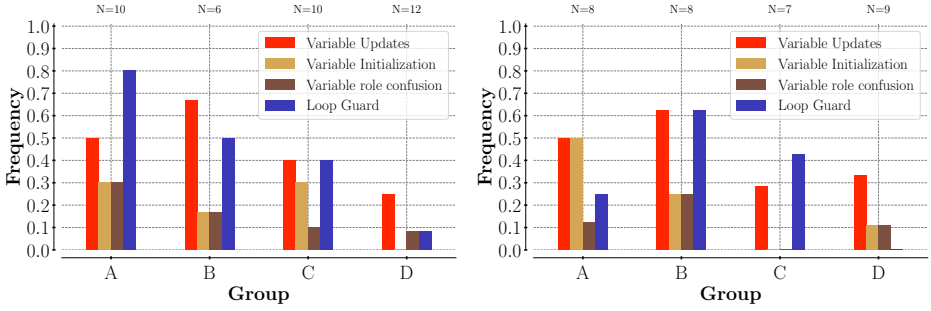
To investigate further, Appendix D explores the relationship between GLI's quality (general and specific states) and code status and functional rate. In general, the quality of the Initial, In-loop and Final States of the GLI showed moderate, statistically relevant correlations with both code status and functional rate across both problems ($r \in [0.30, 0.58]$, $p \in [0.001, 0.036]$), suggesting GLIBP is a good companion for code correctness. There is one exception for Problem 1, where the GLI (General State) quality showed no linear correlation with either code status ($r = 0.05$, $p = 0.401$) or

functional rate ($r = 0.1$, $p = 0.315$). This can be attributed to the zero functional rates carried by incomplete code snippets, combined with stronger coders' tendency to reason directly over the code, bypassing the GLI. Subsection 6.2 discusses it.

Finally, Fig. 12 compares functional rates across groups for complete codes only. Group D achieved the highest median in Problem 1, while Group C achieved the highest median in Problem 2. In Problem 1, medians increased progressively from Group A to D, suggesting benefits from GLI usage. However, this analysis excludes students with incomplete code, potentially overrepresenting stronger coders in the experimental groups. In Problem 2, Groups B and D showed similar medians (though with higher variance), while Group A performed better, and Group C showed moderate improvement.

5.3. *RQ3: Are there errors students can avoid in their code thanks to GLIBP?*

Results from RQ1 and RQ2 suggest that students from Group D mostly found the GLI useful, and their code correctness supports this perception, with $7/23$ achieving error-free solutions. This raises the question of which specific errors GLIBP helps prevent.



(a) Problem 1 (Compressing an array).

(b) Problem 2 (Drawing a calendar).

Figure 13. Semantic errors found in code considered *complete enough* (completion rate > 2).

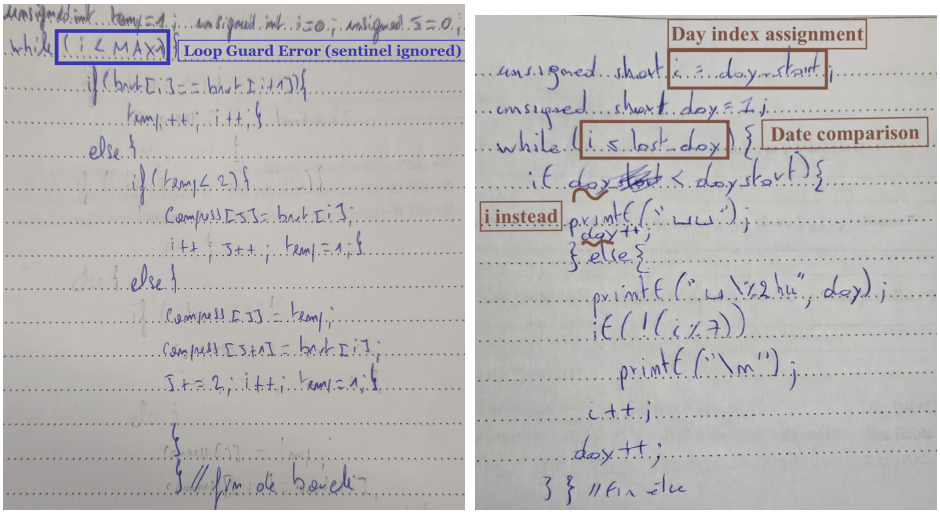
For both problems, Fig. 13 shows that fewer errors occurred when students received the completed GLI. It provides strong evidence that GLIBP helps students code correctly, without trial and error (since they wrote code on paper).

Conversely, Groups A and B encountered more errors. This demonstrates that GLIBP can guide proper loop implementation, but a fill-in-the-blank GLI may not be as helpful as intended. In Problem 1, only six Group B students provided sufficiently complete solutions (completion rate $> 2/5$). Half struggled to define the loop guard and update variables. Students with a wrong loop guard also filled in the GLI incorrectly (getting a score in $[0.5, 2]$) and had poor Final State scores ($[2.5, 3]$). The most common mistake related to the Final State was exiting the loop when `i==MAX` instead of when reaching the sentinel value -1. Fig. 14a shows an example of this error from a student in Group C. Many students blindly imitated previous examples, traversing the whole array. In Problem

2, students from Groups B and C making loop guard mistakes also provided a poor GLI (with a score varying from 0 to 2.5) and corresponding FINAL STATE ($[-1, 2.5]$).

Next, at least half of the students in Groups A and B *updated variables incorrectly*. This error type was often coupled with incorrect *variable initialization*, occurring most frequently in Group A across both problems. Some Groups C and D students were also not able to correctly *update their variables* in the loop body. Subsection 6.3 discusses it.

For *variable role confusion* (e.g., using the same index for different array locations), Problem 1 showed highest occurrence in Group A and decreasing from Group B to D. This pattern may indicate that the GLI supported some students in understanding variable roles. However, the Problem 2 results do not corroborate this, with at least one occurrence in all groups, except Group C. Fig. 14b illustrates a code written by a Group B student that demonstrates confusion regarding the role of variable `i`. The variable is inconsistently treated as both a date (in the loop guard) and a day index (in the initialization)[5].



(a) Incorrect loop guard (Compressing an array). (b) Variable role confusion (Drawing a calendar).

Figure 14. Examples of students' code.

6. Discussion

This section synthesizes the findings from the three research questions. For each, we interpret the observed results, situate them with respect to prior work, and draw bounded implications for instructors considering GLIBP.

[5]Fig. 21 clarifies this distinction with `day_start` being a day index and `last_date` a date.

6.1. *RQ1: Some impediments to the GLI use*

Subsection 5.1 showed that a non-negligible proportion of participants across the three experimental groups did not leverage the GLI as expected.

In Group C, the GLI drawn by students often lacked accuracy, which may explain why only half of these students were actually using it to code. Incomplete invariant diagrams were also among the most common errors observed by Mannila (2010).

Regarding the two other experimental groups, we noticed that students could not easily complete a fill-in-the-blank GLI (Group B). Their open-ended survey responses suggest that they faced two main difficulties, that can be connected to the Cognitive Load Theory (CLT), described by Plass *et al.* (2010). The first challenge was the fill-in-the-blank GLI comprehension (extraneous load). The second was the solution elaboration (germane load). Both of these came in addition to understanding the problem itself (intrinsic load). Although we did not find any prior work connecting the CLT to Sfard (1991)'s operational-structural framework matching GLIBP, we find it relevant to relate them to each other. The fill-in-the-blank GLI attempts to facilitate students' structural view. However, for Group B, the incomplete scaffold may have created an additional difficulty in connecting the incomplete diagram to their natural (operational[6]) reasoning process. On the other hand, Group C, building GLI from scratch, found more useful to adopt a structural view (see Fig. 9) and navigate more organically from operational to structural understanding (see Table 3). This is even more pronounced for Group D that could rely on a complete structural representation, reducing the germane load.

An alternative explanation for Group B's difficulties is problem-specific rather than structural: the `k` variable in Problem 2 appeared confusing across groups, which may have amplified the fill-in-the-blank GLI's difficulty, independently from its fill-in-the-blank format. Some students from Group D were also struggling to understand it. Instructors should treat this as a signal that GLI design should also rely on observed student strategies.

6.2. *RQ2: GLIBP appearing neither necessary nor sufficient for code correctness*

Building on findings from RQ1 and summarizing results for RQ2, we can point out that some students could not complete their code. It might result from the cognitive load imposed by the GLI, as discussed previously. Section 8 gives recommendations to address this difficulty. We also observed that better code (both status and functional rate) is often associated with higher-quality GLI (in both general and specific states). Moreover, a given GLI helps students write error-free codes. The GLI likely protects students from certain errors. This hypothesis is examined in RQ3. The constructive role of the GLI is also corroborated by the progressive improvement of median functional rates from Group A to D in Problem 1. It is less pronounced in Problem 2 likely due to the confusing design around the day index variable, as already noticed in RQ1. In response to that, an alternative version is presented in Section 7.

[6]Corney *et al.* (2012) demonstrates that students prefer thinking from a code tracing perspective (being an instance of operational reasoning) over explaining code at a relational level.

Finally, we noticed that some students can produce working code despite poor GLI quality. When inspecting their GLI more closely, we identified the following issues:

- *Incorrect Analogical Reasoning*: One student represented a GLI related to another similar problem previously shown in the course. Despite this flawed analogical reasoning (consistent with the findings from Saxena *et al.* (2021); Kao *et al.* (2022)), the student was still able to maintain correct variable relationships and loop guard in their code implementation. This problem-solving behavior echoes the pattern-based instruction approach mentioned in Section 2. Its effectiveness in teaching loops was demonstrated by Fernández Alemán and Oufaska (2010). We may promote it to strengthen GLIBP.
- *Incomplete GLI but correct implementation*: The second student produced an inaccurate GLI but demonstrated a well-organized conditional logic in their code, though using an excessive number of variables (5 instead of 3).
- *Repeating each action in a dedicated loop*: Despite being asked to fit their solution in a single loop, some students used more than one loop to solve the problems, bypassing the GLI by reducing the difficulty of their loops. Grover and Basu (2017) also observed that some students feel more comfortable doing so.

6.3. *RQ3: Dimensions of GLIBP associated with fewer semantic errors*

The error analysis provides the most concrete evidence of GLIBP's positive impact during the experimental sessions. Across both problems, Group D exhibited consistently fewer semantic errors. Loop guard errors and variable role confusion were less frequent in this group, suggesting that the GLI may support students in visualizing sentinel values and clarifying each variable's role. This aligns with Al-Barakati and Al-Aama (2009)'s study, demonstrating that visualizing variable roles significantly improves code quality in CS1 settings.

On the other hand, variable update errors persisted across all groups, including Group D. Morazán (2020) also identified variable management among the primary difficulties beginners face. In Group D, for the second problem, it occurred because we designed a GLI with limited variables, forcing students to understand the relationship between variables. Some students benefited from it while others became confused. Other groups defined additional variables for clarity. When we next present this problem with a corresponding GLI, we may define a specific variable for the current date rather than expressing it through the iterative variable and the starting date. However, more variables increase error exposure since they require correct maintenance. Variable update errors were most common in Groups A and B, which illustrates this pattern. They used more variables than needed ($\mu_{1_A} = 3.7$, $\mu_{1_B} = 5.5$ vs. 3 needed; $\mu_{2_A} = 2.4$ vs. 1 expected). The correlation between variable count and variable update errors in Problem 2, for example, was $r = 0.58, p = 9.17e - 04$, suggesting that the use of additional variables may increase the likelihood of variable update errors.

7. Threats to Validity

The results of this study are subject to certain biases that we could not avoid: (i) the composition of the student groups, (ii) the size of each group, (iii) the selection of problem statements, and (iv) the single coder to score students' code and diagrams.

Ideally, a more rigorous experimental design would have included a control group composed of students who did *not* learn Graphical Loop Invariant Based Programming. However, identifying such a comparable control group was too challenging. Considering students from other courses would have introduced confounding factors (programming language, course structure, parallel coursework), making it difficult to isolate the specific effects of GLIBP. Another limitation is that the participants in our experiment were still beginners. Many of them struggle to grasp the purpose of GLIBP and do not engage with it. "Dijkstra described this as mental resistance among students", as Mannila (2010) recalls. Students often prefer a trial-and-error coding approach that yields quick results, as also illustrated by Reed and Sinclair (2004). Consequently, not all participants fully embraced our GLIBP approach, leading students to rely on underdeveloped GLIs.

Group size was another limiting factor, especially in the second session. The 100 enrolled students split across 4 groups yielded a maximum of 25 per group. Furthermore, because participation in the experiment was voluntary, some students chose not to attend.

Besides this, the selected problem statements and their corresponding GLI also influenced the results. To mitigate this bias, the experiment included two different problems. Regarding the second problem in particular, in Groups B and D, we realized that expecting a single variable (`day_index`) to maintain the solution was confusing to students. Some of them could not identify the relationship between the index of the day and the date. In Groups A and C, most students defined two variables to avoid the associated cognitive effort. In response to this observation, if we propose that problem again, we may revise the associated GLI to limit students' confusion. An alternative version is illustrated in Fig. 15, with two variables (`nb_spaces` and `curr_date`) to support the solution.

Finally, the analysis of students' code and diagrams was performed by a single teaching assistant, due to staffing constraints. To mitigate that limitation, students' solutions were analyzed from three to four times and, to validate it, a subset of student diagrams and code was extracted and rated by the professor of the course to be compared with the teaching assistant's coding.

Despite careful experimental design, our findings remain context-dependent. However, they offer instructors meaningful insights into how novice programmers adopt GLIBP.

8. Conclusion

This paper presents preliminary findings regarding the potential role of Graphical Loop Invariant Based Programming (GLIBP) in supporting CS1 students while constructing loops more systematically and potentially with fewer semantic errors. This approach requires students to model their solution through a Graphical Loop Invariant (GLI) before

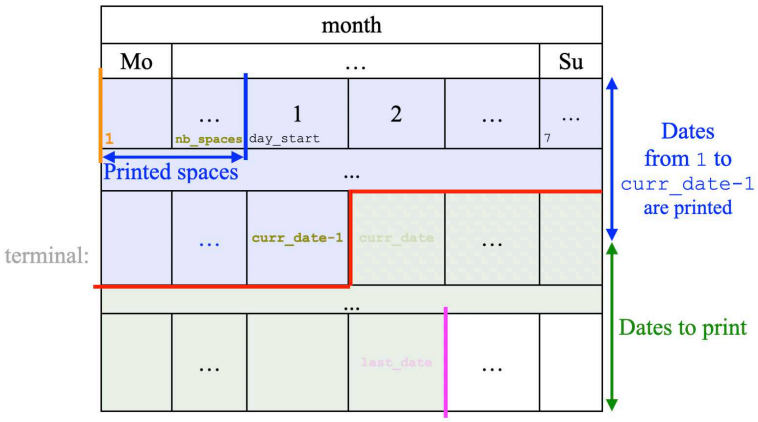


Figure 15. Alternative GLI for the second problem. The first version is presented in Fig. 22.

coding. It is intended to onboard students with FM thinking by developing their abilities to reason about their program. It can be assimilated to the first level of FM thinking (out of three) introduced by Dongol *et al.* (2024). It aims to capture "What's True Here" through natural language and/or informal diagrams. Developing these skills is crucial for a computer scientist (Broy *et al.* (2024)).

To conduct this study, we designed a controlled classroom experiment involving multiple forms of GLI exposure: (i) No GLI required (control group), (ii) fill-in-the-blank GLI, (iii) GLI to draw from scratch, and (iv) correct GLI provided. Within this classroom setting, we observed that the access to a correct GLI was associated with fewer semantic mistakes and with a better understanding of relationships between variables and exit condition. However, this benefit depends on the access to a correct GLI, which is a significant challenge. Many students struggled to represent a GLI and consequently do not perceive its benefits to construct the code. To overcome this, we introduced a few years ago the fill-in-the-blank GLI version, where the GLI is sketched for students. Unfortunately, our observations suggest that interpreting and completing these partial diagrams likely adds cognitive load and reduces the time available for coding, rather than actually supporting students. In the survey, a large proportion of students spontaneously raised that comprehending the fill-in-the-blank GLI was hard, suggesting that we should train them more on this isolated task. As instructors, we must be conscious of the *expert blind spot* effect and avoid overwhelming students with overly complex (fill-in-the-blank) GLI. We should design and refine the GLI we sketch (or provide) based on actual student interpretation, not solely on our assumptions that may not align with novice understanding.

Based on these preliminary observations, to address the difficulties some students face in constructing GLI, we recommend first training them to implement code from provided GLI diagrams. This approach aims to maintain active coding engagement while illustrating how GLI can be used as a problem-solving support tool. This experience may motivate students to subsequently adopt the full methodology. This may also help students become more familiar with common solution patterns and learn to adapt them to new problems.

This process involves pattern recognition skills, referenced in Table 1. To go further and improve students' ability to *accurately* fine-tune their GLI, we suggest that instructors could guide them through key questions: identifying the variables needed to solve the problem (as recommended by Tam (1992)), determining the domain of these variables, and understanding the relationships between them. This recommendation is consistent with the problem-solving strategies identified by Castro and Fisler (2020): reproducing similar solutions and following systematic plans. Finally, to further motivate students, it is crucial to select problems that are solvable, yet sufficiently challenging to require diagrammatic reasoning without causing students to become overwhelmed by implementation details.

As future work, replicating this study in other classrooms would be necessary to generalize these preliminary results. We also advocate for longitudinal studies assessing how early exposure to GLI shapes programming habits in more advanced coursework. Should these findings be confirmed at a larger scale, they could encourage other instructors to consider integrating GLIBP into their CS courses, despite not being familiar with formal methods. By establishing program correctness foundations early through invariant-based reasoning, students may be more engaged with formal methods in advanced courses.

Ethical considerations

This study has been carried out in accordance with the ethical principles and good practices of the APA and the World Medical Association's ethical principles for medical research involving human subjects (Declaration of Helsinki, adopted June 1964, with subsequent amendments).

To conduct the study, the researchers consulted the local ethics committee to determine whether formal ethics approval was required. The committee indicated that a formal ethics review was not necessary for this study because no directly identifiable research data were collected or retained. As illustrated in Fig. 16, the submitted diagrams, code, and questionnaire responses were collected and stored only under group identifiers and could not be traced back to individual students. Student identification numbers were temporarily collected in a separate spreadsheet solely for the purpose of retrieving prerequisite-course exam grades used to compare baseline knowledge across groups. These identifiers were never associated with students' submissions or questionnaire responses and were deleted after the grade retrieval process was completed.

Consent to participate

Prior to enrolment in the study, all students were informed verbally that their survey answers, diagrams, and code submissions would be used for research purposes.

Participation in the study was voluntary, and students could opt out without any academic consequences or impact on course grades or evaluation. All participants were adults

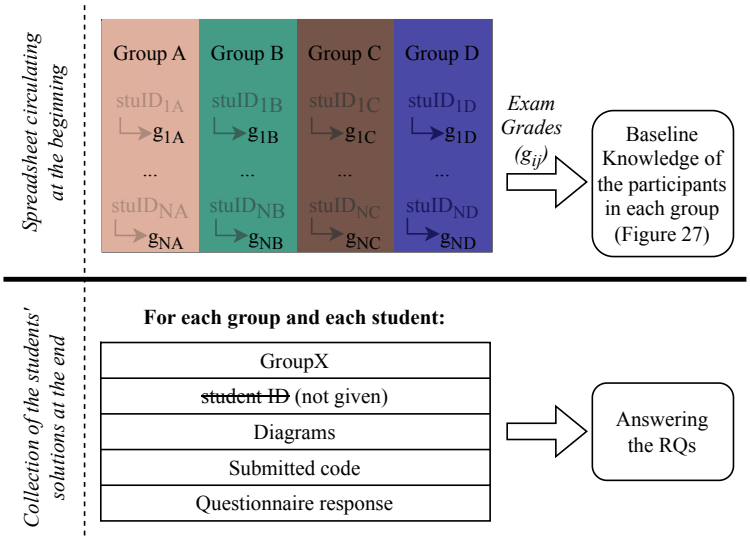


Figure 16. Collected data for each session.

(18 years or older) and provided informed oral consent. Three students (out of 52[7]) chose not to participate.

Data anonymity and availability

In each of the two sessions, as shown in Fig. 16, we circulated a spreadsheet in which students from each group were asked to indicate their student ID. The purpose of collecting this information was solely to obtain the prior exam grades associated with each group, allowing us to compare groups' baseline knowledge with one another.

The spreadsheet containing student identifiers was stored separately from the research data and was never linked to diagrams, code submissions, or questionnaire responses. After grade retrieval, student identifiers were deleted, and only the corresponding exam grades were retained in anonymized form. Next, each student's submissions (diagrams, code, and survey responses) were collected and associated only with a group number. They were not mapped to individual student IDs. All research data were stored on secure institutional servers of the University of Liège. Access to the data was restricted to the two researchers involved in the study. The anonymized research data will be retained for two years following publication. The datasets generated and analyzed during the current study are not publicly available due to privacy and ethical restrictions. However, fully anonymized student submissions may be made available from the corresponding author upon reasonable request.

[7]This number corresponds to the students who attended the regular exercise sessions but did not participate in the experimental sessions.

Declaration of conflicting interest

The authors declare no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

References

Al-Barakati, N.M., Al-Aama, A.Y. (2009). The Effect of Visualizing Roles of Variables on Student Performance in an Introductory Programming Course. In: *Proc. ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE)*. https://doi.org/10.1145/1562877.1562949.

Anderson, M., Meyer, B., Olivier, P. (Eds.) (2002). *Diagrammatic Representation and Reasoning*. Springer London. https://doi.org/10.1007/978-1-4471-0109-3.

Astrachan, O. (1991). Pictures as Invariants. In: *Proc. ACM Technical Symposium on Computer Science Education (SIGCSE)*. https://doi.org/10.1145/107004.107026.

Back, R.-J. (2009). Invariant Based Programming: Basic Approach and Teaching Experiences. *Formal Aspects of Computing*, 21(3), 227–244. https://doi.org/10.1007/s00165-008-0070-y.

Brieven, G., Malcev, L., Donnet, B. (2024). Practicing Abstraction Skills Through Diagrammatic Reasoning Over CAFÉ 2.0. In: *Proc. IEEE Global Engineering Education Conference (EDUCON)*. IEEE, pp. 1–10. https://doi.org/10.1109/EDUCON60312.2024.10578665.

Brieven, G., Malcev, L., Donnet, B. (2025). How to Automate Feedback on Diagrammatic Reasoning with a Relevant Degree of Freedom? In: *Proc. IEEE Global Engineering Education Conference (EDUCON)*. IEEE, pp. 1–10. https://doi.org/10.1109/EDUCON62633.2025.11016495.

Brieven, G., Liénardy, S., Malcev, L., Donnet, B. (2023). Graphical Loop Invariant Based Programming. In: *Proc. Formal Methods Teaching Workshop (FMTea)*. Springer, pp. 17–33. https://doi.org/10.1007/978-3-031-27534-0_2.

Broy, M., Brucker, A.D., Fantechi, A., Gleirscher, M., Havelund, K., Kuppe, M.A., Mendes, A., Platzer, A., Ringert, J.O., Sullivan, A. (2024). Does Every Computer Scientist Need to Know Formal Methods? *Formal Aspects of Computing*, 37(1), 1–17. https://doi.org/10.1145/3670795.

Castro, F.E.V., Fisler, K. (2020). Qualitative Analyses of Movements Between Task-level and Code-level Thinking of Novice Programmers. In: *Proc. ACM Technical Symposium on Computer Science Education (SIGCSE)*. https://doi.org/10.1145/3328778.3366847.

Cetin, I. (2020). Teaching Loops Concept through Visualization Construction. *Informatics in Education*, 19(4), 589–609. https://doi.org/10.15388/infedu.2020.26.

Cherenkova, Y., Zingaro, D., Petersen, A. (2014). Identifying challenging CS1 concepts in a large problem dataset. In: *Proc. ACM Technical Symposium on Computer Science Education (SIGCSE)*. https://doi.org/10.1145/2538862.2538966.

Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C. (2009). *Introduction to Algorithms*. MIT Press.

Corney, M., Teague, D., Ahadi, A., Lister, R. (2012). Some Empirical Results for Neo-Piagetian Reasoning in Novice Programmers and the Relationship to Code Explanation Questions. In: *Proc. ACM Australasian Computing Education Conference (ACE)*.

Dijkstra, E.W. (1976). *A Discipline of Programming*. Prentice-Hall, Inc.

Dongol, B., Dubois, C., Hallerstede, S., Hehner, E., Morgan, C., Müller, P., Ribeiro, L., Silva, A., Smith, G., de Vink, E. (2024). On Formal Methods Thinking in Computer Science Education. *Formal Aspects of Computing*, 37(1), 1–23. https://doi.org/10.1145/3670419.

Eriksson, J., Parsa, M., Back, R.-J. (2018). A Precise Pictorial Language for Array Invariants. In: *Proc. International Conference on Integrated Formal Methods (IFM)*. https://doi.org/10.1007/978-3-319-98938-9_9.

Fernández Alemán, J.L., Oufaska, Y. (2010). SAMtool, a Tool for Deducing and Implementing Loop Patterns. In: *Proc. ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE)*. https://doi.org/10.1145/1822090.1822111.

Fowler, M., Kraemer, E., Sitaraman, M., Hollingsworth, J.E. (2021). Tool-Aided Loop Invariant Development: Insights into Student Conceptions and Difficulties. In: *Proc. ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE)*. https://doi.org/10.1145/3430665.3456351.

Gomes, A., Ke, W., Lam, C.-T., Teixeira, A.R., Correia, F.B., Marcelino, M.J., Mendes, A.J. (2019). Understanding Loops: A Visual Methodology. In: *Proc. IEEE International Conference on Engineering, Technology and Education (TALE)*. https://doi.org/10.1109/TALE48000.2019.9225951.

Gries, D. (1987). *The Science of Programming*. Springer.

Grover, S., Basu, S. (2017). Measuring Student Learning in Introductory Block-Based Programming: Examining Misconceptions of Loops, Variables, and Boolean Logic. In: *Proc. ACM Technical Symposium on Computer Science Education (SIGCSE)*. https://doi.org/10.1145/3017680.3017723.

Hoare, C.A.R. (1969). An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10), 576–580. https://doi.org/10.1145/363235.363259.

Iyer, V., Zilles, C. (2021). Pattern Census: A Characterization of Pattern Usage in Early Programming Courses. In: *Proc. ACM Technical Symposium on Computer Science Education (SIGCSE)*. SIGCSE '21. Association for Computing Machinery, New York, NY, USA, pp. 45–51. https://doi.org/10.1145/3408877.3432442.

Kamburjan, E., Grätz, L. (2021). Increasing Engagement with Interactive Visualization: Formal Methods as Serious Games. In: *Proc. Formal Methods Teaching Workshop (FMTea)*. https://doi.org/10.1007/978-3-030-91550-6_4.

Kao, Y., Matlen, B., Weintrop, D. (2022). From One Language to the Next: Applications of Analogical Transfer for Programming Education. *ACM Transactions on Computing Education (TOCE)*, 22(4), 1–21. https://doi.org/10.1145/3487051.

Kumar, A.N., Raj, R.K., Aly, S.G., Anderson, M.D., Becker, B.A., Blumenthal, R.L., Eaton, E., Epstein, S.L., Goldweber, M., Jalote, P., Lea, D., Oudshoorn, M., Pias, M., Reiser, S., Servin, C., Simha, R., Winters, T., Xiang, Q. (2024). *Computer Science Curricula 2023*. Association for Computing Machinery. https://doi.org/10.1145/3664191.

Luxton-Reilly, A., Albluwi, I., Becker, B.A., Giannakos, M., Kumar, A.N., Ott, L., Paterson, J., Scott, M.J., Sheard, J., Szabo, C. (2018). Introductory Programming: a Systematic Literature Review. In: *Proc. ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE)*. https://doi.org/10.1145/3293881.3295779.

Mannila, L. (2010). Invariant Based Programming in Education – An Analysis of Student Difficulties. *Informatics in Education*, 9(1), 115–132. https://doi.org/10.15388/infedu.2010.07.

Mirolo, C., Izu, C., Lonati, V., Scapin, E. (2022). Abstraction in Computer Science Education: An Overview. *Informatics in Education*, 20(4), 615–639. https://doi.org/10.15388/infedu.2021.27.

Morazán, M. (2020). How to Design While Loops. *Electronic Proceedings in Theoretical Computer Science*, 321(13), 1–18. https://doi.org/10.4204/EPTCS.321.1.

Morgan, C. (1990). *Programming from Specifications*. Prentice-Hall.

Morrison, B.B., Margulieux, L.E., Guzdial, M. (2015). Subgoals, Context, and Worked Examples in Learning Computing Problem Solving. In: *Proc. ACM International Computing Education Research (ICER)*. https://doi.org/10.1145/2787622.2787733.

Plass, J.L., Moreno, R., Brünken, R. (2010). *Cognitive load theory*. Cambridge University Press.

Reed, J., Sinclair, J. (2004). Motivating Study of Formal Methods in the Classroom. In: *Proc. Teaching Formal Methods*. https://doi.org/10.1007/978-3-540-30472-2_3.

Rodrigues, G., Monteiro, A.F., Osório, A. (2022). Introductory Programming in Higher Education: A Systematic Literature Review. In: *Proc. International Computer Programming Education Conference (ICPEC)*. https://doi.org/10.4230/OASIcs.ICPEC.2022.4.

Saxena, P., Singh, S.K., Gupta, G. (2021). Analogy-based Instruction for Effective Teaching of Abstract Concepts in Computer Science. In: *Proc. International Conference on Higher Education Advances (HEAd)*. https://doi.org/10.4995/HEAd21.2021.13115.

Scapin, E., Mirolo, C. (2019). An Exploration of Teachers' Perspective About the Learning of Iteration-Control Constructs. In: Pozdniakov, S.N., Dagienė, V. (Eds.), *Proc. Informatics in Schools. New Ideas in School Informatics*. https://doi.org/10.1007/978-3-030-33759-9_2.

Sfard, A. (1991). On the Dual Nature of Mathematical Conceptions: Reflections on Processes and Objects as Different Sides of the Same Coin. *Educational Studies in Mathematics*, 22(1), 1–36. https://doi.org/10.1007/BF00302715.

Sooriamurthi, R. (2009). Introducing abstraction and decomposition to novice programmers. *SIGCSE Bull.*, 41(3), 196–200. https://doi.org/10.1145/1595496.1562939.

Tam, W.C. (1992). Teaching Loop Invariants to Beginners by Examples. In: *Proc. ACM Technical Symposium on Computer Science Education (SIGCSE)*. https://doi.org/10.1145/135250.134530.

Walker, H. (1998). Modules to introduce assertions and loop invariants informally within CS1: Experiences and observations. *ACM SIGCSE Bulletin*, 30, 31–35. https://doi.org/10.1145/292422.292437.

Walker, H.M. (2023). Curricular Syncopations: Where/How Do Loop Invariants Fit? *ACM Inroads*, 14(1), 6–13. https://doi.org/10.1145/3583089.

Zhang, J., Atay, M., Caldwell, E.R., Jones, E.J. (2013). Visualizing Loops Using a Game-Like Instructional Module. In: *Proc. IEEE International Conference on Advanced Learning Technologies*. https://doi.org/10.1109/ICALT.2013.137.

Zhumagambetov, R. (2021). Teaching Formal Methods in Academia: A Systematic Literature Review. In: *Proc. Formal Methods – Fun for Everybody*. https://doi.org/10.1007/978-3-030-71374-4_12.

A. Problems addressed in the two experimental sessions

A.1. *Problem 1: Compressing an array, based on consecutive identical occurrences*

A.1.1. *Problem statement given to students*

You are expected to write a program that compresses a sequence of 0 and 1. That initial sequence is stored in an array containing maximum `MAX` elements, always ending up with value -1 (marking the end of the data of interest). For instance, for `MAX` $= 20$:

	0																		MAX−1
brut:	0	0	1	1	1	1	0	1	1	1	0	0	1	0	0	0	0	−1	?	?

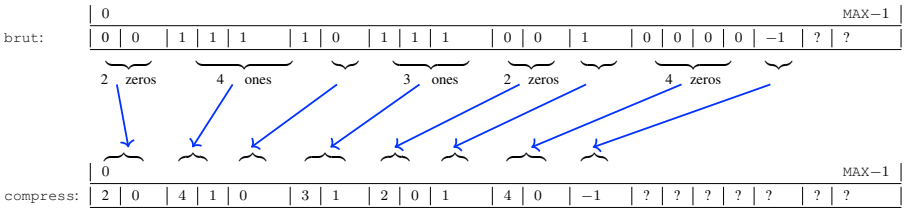
Your program stores the initial and compressed data in two arrays of size `MAX`, respectively called `brut` and `compress`. Pay attention to the fact that the final value (i.e., -1) isn't always stored at the last position of the array. A sequence whose length is less than `MAX` can be stored in an array or size `MAX` (as illustrated in the example above).

The compression algorithm applies a very simple principle: it detects all the sequences of identical numbers (for instance, three "1" following each other). When a number occurs only once, it is preserved in `compress`. However, if it occurs n sequential times (with $n \geqslant 2$), it is stored under the format « n followed by the repeated element ». For example, the array `compress` corresponding to the array `brut` presented above is:

	0																		MAX−1
compress:	2	0	4	1	0	3	1	2	0	1	4	0	−1	?	?	?	?	?	?	?

Indeed, in the array `compress`, "0" occurs twice, then "1" appears four times, after "0" occurs once etc., which suggests the following compression process:

The skeleton of your code is the following:




```
1 void compression(int *brut, int *compress, unsigned int MAX){
2     // To fill in by you
3 }
```

A.1.2. *GLI with its specific states (provided to Group D)*

A possible GLI to solve this problem is presented in Fig. 17. It illustrates that the solution relies on two arrays: one (`brut`) that is read-only, and another (`compress`) that stores the resulting compressed sequence. In `brut`, four distinct zones are defined. The first zone contains elements that have already been compressed. This implies that the last element of this zone (`brut[i-cpt]`) differs from the first element of the second zone. The third zone includes the remaining elements of the sequence yet to be processed. The fourth zone contains elements that do not need to be visited to solve the problem. In `compress`, two zones are labeled as "Free space". This distinction is made to differentiate between the area that will eventually hold the rest of the compressed sequence and the area that will remain unused. The only configuration where all the array is needed is when `brut` contains an alternating sequence of `MAX`-1 values (ending with -1).

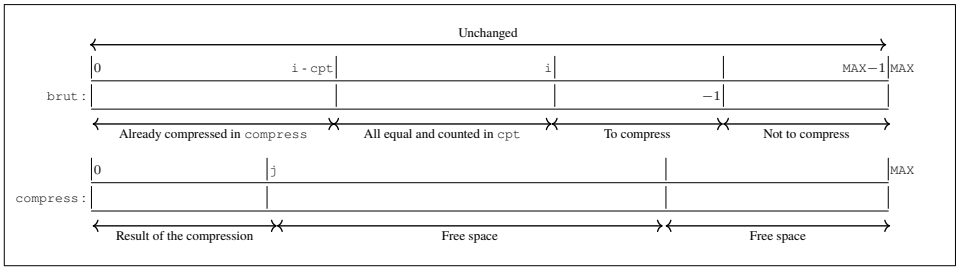


Figure 17. The GLI for solving the first problem (Compress an array).

From this GLI (General State), we derived the Initial and Final States, given in Fig. 18 and Fig. 19.

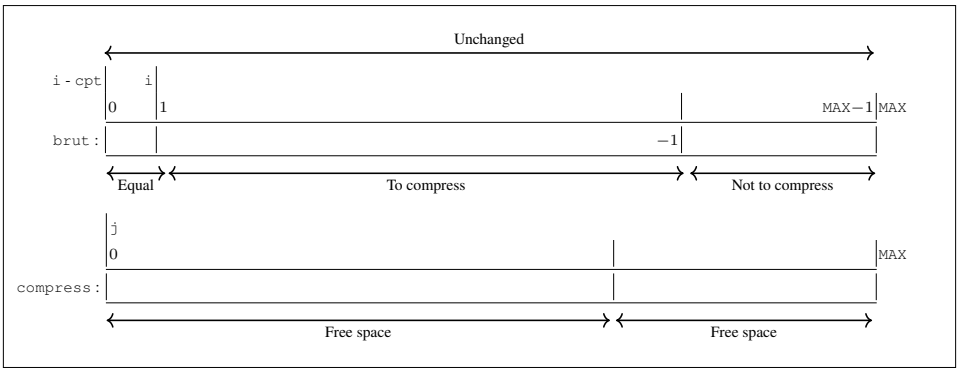


Figure 18. The Initial State of the GLI for solving the first problem (Compress an array).

The variables' declaration and initialization translated from Fig. 18 is given below. Fig. 18 clearly highlights that `i` and `j` are initially equal to 0 (`i` corresponds to the index of the first element in `brut`). Knowing that `i=0` and `i−cpt` aligns with −1, we deduce that `cpt` should be equal to 1.

```
unsigned i=0;
unsigned cpt=1;
unsigned j=0;
```

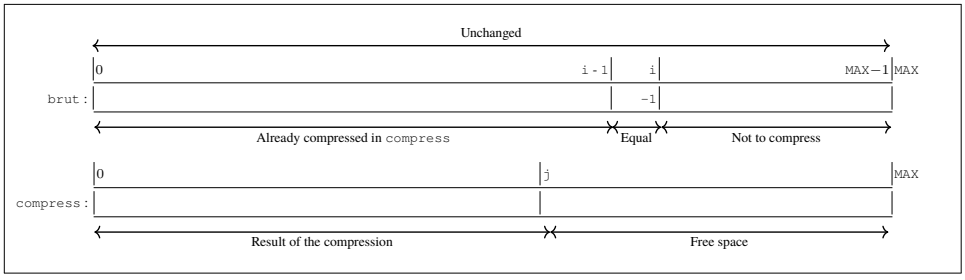


Figure 19. The Final State of the GLI for solving the first problem (Compress an array).

The exit condition derived from Fig. 19 is: `brut[i]==` −1. In this case, `cpt=` 1 and all the values preceding −1 have been compressed into `compress`, in accordance with the GLI, thus fulfilling the function's objective. The only remaining instruction upon exiting the loop is to assign −1 to `compress[j]` to terminate the compressed sequence.

A.1.3. *Fill-in-the-blank GLI (provided to Group B)*

From Fig. 17, we derived a fill-in-the-blank GLI. It is illustrated in Fig. 20. Red boxes host expressions while green boxes expect a symbol from a predefined list. Some boxes may be left empty. The list of symbols students can drag and drop is given in Table 4.

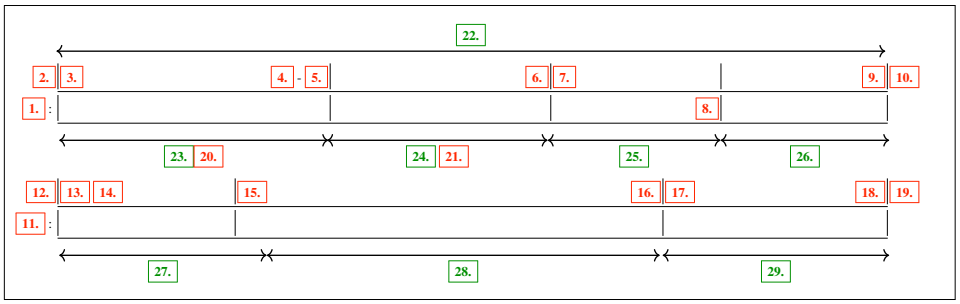


Figure 20. The fill-in-the-blank GLI for solving the first problem (Compress an array).

1. Already browsed	10. Still to compress	19. Already displayed
2. Free cells	11. Modified	20. Still to compute
3. Sum of	12. Already compressed in	21. Already computed in
4. Multiplied by	13. To browse	22. All equal and counted in
5. Already encoded in	14. Unchanged	23. Cells left empty
6. Still to display	15. $>$	24. Result of the compression
7. Result of the computation	16. Sum computed in	25. Even
8. Still to browse	17. Do not compress	26. $=$
9. Do not display	18. Already done in	27. Still to free

Table 4
GLI: Possible choices for solid green-outlined boxes (for problem 1).

A.1.4. *Code snippet solving the problem*

Students were expected to produce the following code (or an equivalent version):

```
1   unsigned int i = 0, j = 0;
2   unsigned int cpt = 1;
3
4   while(i < MAX && brut[i] != -1){
5     if(brut[i]==brut[i+1])
6       cpt++;
7     else{
8       if(cpt==1){
9         compress[j] = brut[i];
10        j++;
11      }else{
12        compress[j] = cpt;
13        compress[j+1] = brut[i];
14        j += 2;
15      }
16      cpt = 1;
17    }
18    i++;
19  }//end while()
20  compress[j] = -1; //end of sequence
```

A.2. *Problem 2: Drawing a calendar*

A.2.1. *Problem statement given to students*

We expect you to write a program to display a monthly calendar in the terminal. That calendar looks like an array made up of boxes. The first header line should contain the month and the year and the second one should list the seven days of the week. Next, the lines below display the corresponding dates. If a box is not included in the month, you should display an empty box. To draw your calendar, two inputs are given. The first one is the index of the first day of the month (called `day_start`). That index varies from 1 to 7 (1 referring to Monday and 7 referring to Sunday). The second one is the last day of the month (called `lastDate`). It varies from 28 to 31.

```
     February 2025
Mo Tu We Th Fr Sa Su
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28
```

Output 1. Example of result for (`month` ="February 2025", `day_start` = 6 and `last_date` = 28)

```
      April 2024
Mo Tu We Th Fr Sa Su
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30
```

Output 2. Example of result for (`month` ="April 2024", `day_start` = 1 and `last_date` = 30)

```
    September 2002
Mo Tu We Th Fr Sa Su
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30
```

Output 3. Example of result for (`month` ="September 2002", `day_start` = 7 and `last_date` = 30)

Figure 21. Example of results, based on given inputs.

Fig. 21 illustrates some examples and the skeleton of your code is the following:

```
void display_calendar(char *month, unsigned last_date, unsigned day_start)
{
    // ---- The 2 first lines ----
    printf("\t%s\n", month);
    printf("␣Mo␣Tu␣We␣Th␣Fr␣Sa␣Su\n");

    // ---- The dates ----
    // To fill in by you
}
```

To display a number and force it to span on two characters, you can write:

```
  unsigned x=4;
  printf("%2u", x);
  // It will print " 4"
```

A.2.2. *GLI with its specific states (provided to Group D)*

A possible GLI to respond to this problem is given in Fig. 22. It highlights two zones. The first one contains spaces before the first date, and the next one contains the dates.

It is also intended to suggest (via the $7k$ index value) that a new line should start when `day_index` (before being incremented) is a multiple of 7.

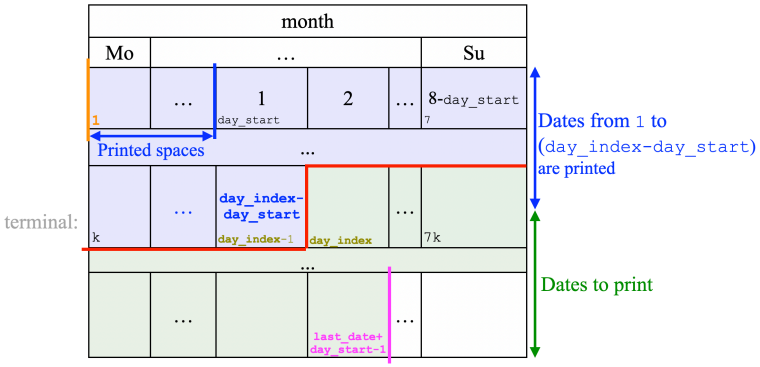


Figure 22. The GLI for solving the second problem (Display a calendar).

From this GLI (General State), we derived the Initial and Final States, given in Figures 23 and 24. Fig. 23 shows that `day_index` should be initialized to 1 while Fig. 24 illustrates that the exit condition is `day_index == last_date+day_start`. That makes sense since `day_index−1` reflects the number of characters (being a date or an initial space) that have been printed. Once `day_start−1` initial spaces have been displayed as well as `last_date` dates, that means that the calendar has been completed. This can be summarized through `day_index−1 ==day_start−1+last_date`, which is equivalent to the exit condition illustrated in the figure.

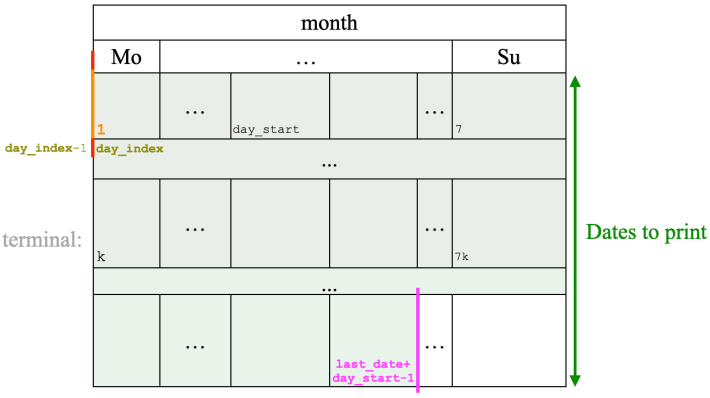


Figure 23. The Initial State of the GLI for solving the second problem (Display a calendar).

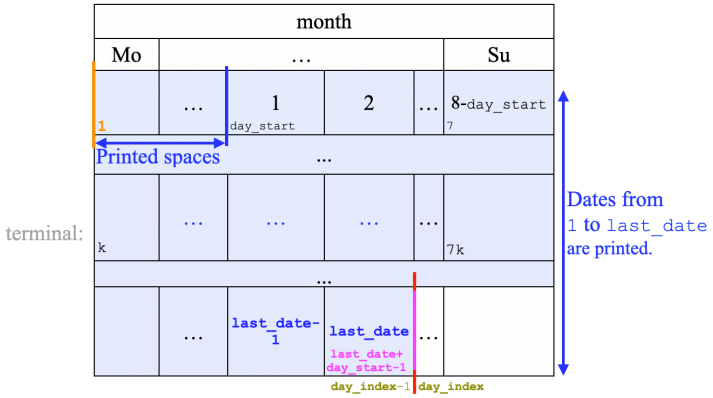


Figure 24. The Final State of the GLI for solving the second problem (Display a calendar).

A.2.3. *Fill-in-the-blank GLI (provided to Group B)*

From Fig. 22, a fill-in-the-blank GLI could be derived. It is illustrated in Fig. 25. Red boxes should host expressions while green boxes expect a symbol from a predefined list. The list of symbols students can drag and drop is given in Table 5.

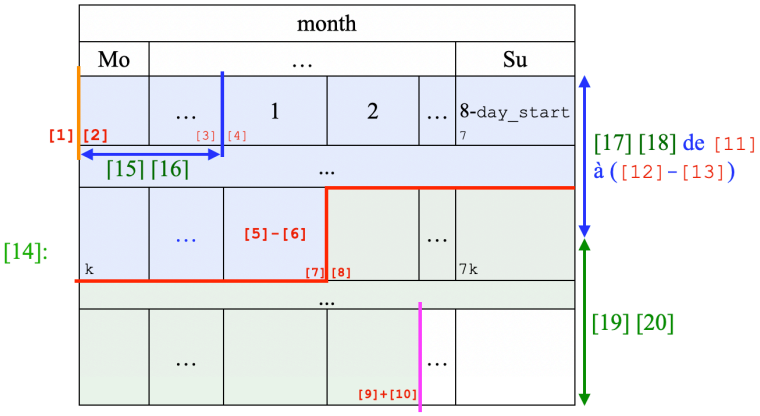


Figure 25. The fill-in-the-blank GLI for solving the second problem (Display a calendar).

1. Already browsed	8. Natural values	15. Already displayed	22. $\neq$
2. Spaces	9. Modified	16. To compute	23. Values
3. Years	10. Terminal	17. Already computed in	24. Still to free
4. Months	11. To browse	18. All equal	25. $=$
5. Interval of	12. Unchanged	19. Display of	26. $<$
6. Still to display	13. $>$	20. Calendar result	27. To be filled with
7. Result of computation	14. Sum computed in	21. Dates	

Table 5
GLI: Possible choices for solid green-outlined boxes (for Problem 2).

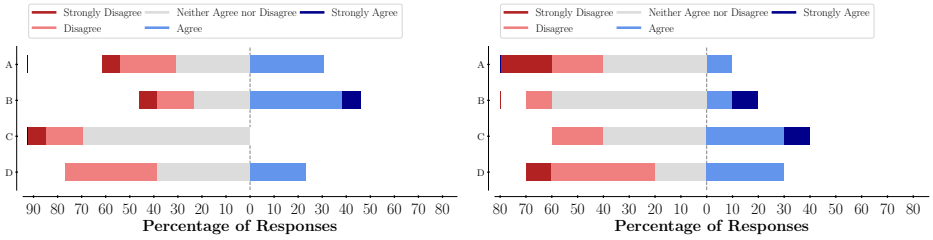
A.2.4. *Code snippet solving the problem*

Students from all the four groups were expected to produce the following code (or an equivalent version):

```
unsigned short day_index=1;

while(day_index<last_date+day_start){
  if(day_index<day_start){
    printf("␣␣␣");
    day_index++;
  }else{
    printf("%2hu␣", day_index-day_start+1);
    if(!(day_index%7))
      printf("\n");
    day_index++;
  }
}
```

B. Validation of the two problem statements



(a) Problem 1 (Compressing an array).

(b) Problem 2 (Drawing a calendar).

Figure 26. Claim: *"To what extent did the problem sound complicated for you?"*

Fig. 26 shows that most students did not find the problems too hard to solve. In Problem 1 (resp. 2), less than 50% (resp. 41%) found it complex, across the four groups. Going further, Fig. 27 illustrates the number of lines of code students wrote, in each group. The dotted vertical line represents the minimum number of lines required to solve the problem. It was based on the instructor's solution. And the solid line shows the number of lines (20) we targeted to hold students' solutions. We can see that a limited proportion of students (5% (resp. about 21%)) wrote more than 20 lines in Problem 2 (resp. 1), confirming that the problems were meeting quite well the expected number of lines.

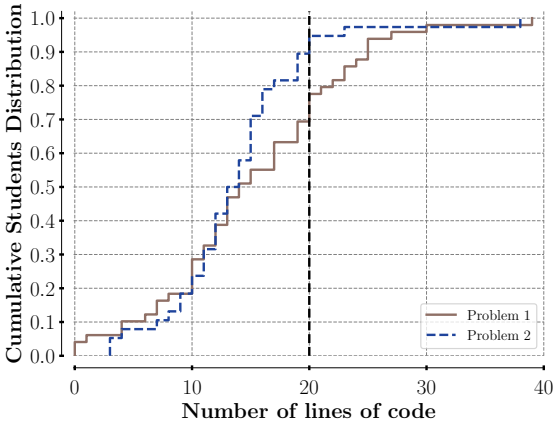


Figure 27. Distribution of students with respect to the number of lines in their code, where 20 lines was the targeted number of lines for students.

C. Baseline knowledge of participants

To measure the baseline knowledge of the participants in each group, we use the grades they obtained to the exam question related to GLIBP.

C.1. *Distribution of exam grades*

Fig. 28 shows the actual distribution of students' grades in each group, to the exam question where students had to apply GLIBP. In Problem 1, distributions seem similar from one group to another with a slight disparity, namely because some students did not participate in the experiment. The disparity is even more pronounced in Problem 2. The figure also suggests that students from Group 1C (equivalent to Group 2A – see Fig. 5) feel more comfortable with our GLIBP approach, seeing their higher grades. This is further supported by Fig. 26, showing that students from this group found both problems relatively easy to solve, regardless of their GLI exposure.

To verify that this does not threaten the validity of the A/B/C/D comparison, we conducted a Kruskal-Wallis test on these grades across groups. The test revealed no statistically significant difference between groups for either problem ($\chi^2(3) = 0.34$, $p = 0.953$ in Problem 1; $\chi^2(3) = 3.17$, $p = 0.367$ in Problem 2), supporting the use of these groups for comparative analysis.

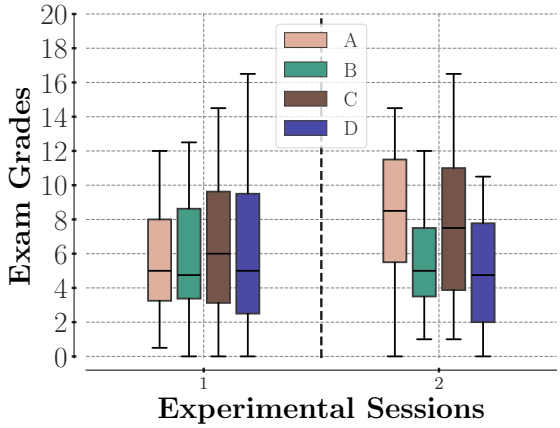


Figure 28. Students' grades to the exam question related to Graphical Loop Invariant Based Programming.

C.2. *Exam question used to measure baseline knowledge*

The exam question addressed to students prior to the two experimental sessions was the following:

Exam question involving GLIBP

In this question, we consider positive integers (i.e., $\in \mathbb{N}$) in decimal base and the conversion of these integers to binary base. We want to write a procedure allowing us to transform into binary base a positive integer, $number$, expressed in decimal base. The binary form of the integer is stored in an array. Thus, the integer 244 is represented, in binary base, by the following array `T` (assuming an 8-bit representation):

`T` = | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |

It is important to note that, in array `T`, the first indices are used to store the most significant bits of the binary representation of $number$, while the last indices concern the least significant bits. We therefore want to write a **procedure** allowing us to convert a positive decimal integer to binary. The prototype of the procedure is as follows:

```
1 void decimal_to_binary(int *T, int N, unsigned int number);
```

where `T` is the result array whose cells have all previously been initialized to 0, `N` is the size of the array (the array is large enough for the conversion) and $number$ is the decimal number. We ask you to implement the procedure `decimal_to_binary()` respecting the following constraints:

- your code must be based on a single `while` loop;
- it is forbidden to use an intermediate array;
- you cannot use any library.

Here are the questions you should answer:

1. Provide the Graphical Loop Invariant based on which you are going to construct your code. Be as rigorous and complete as possible.
2. Indicate the variables you need to declare to solve the problem, as well as their initial value. Justify them based on the Graphical Loop Invariant.
3. Indicate the exit condition of your loop. Justify it based on the Graphical Loop Invariant.
4. Provide the code of the procedure `decimal_to_binary()`.

D. Relationships between the GLI quality (general and specific states) and the code

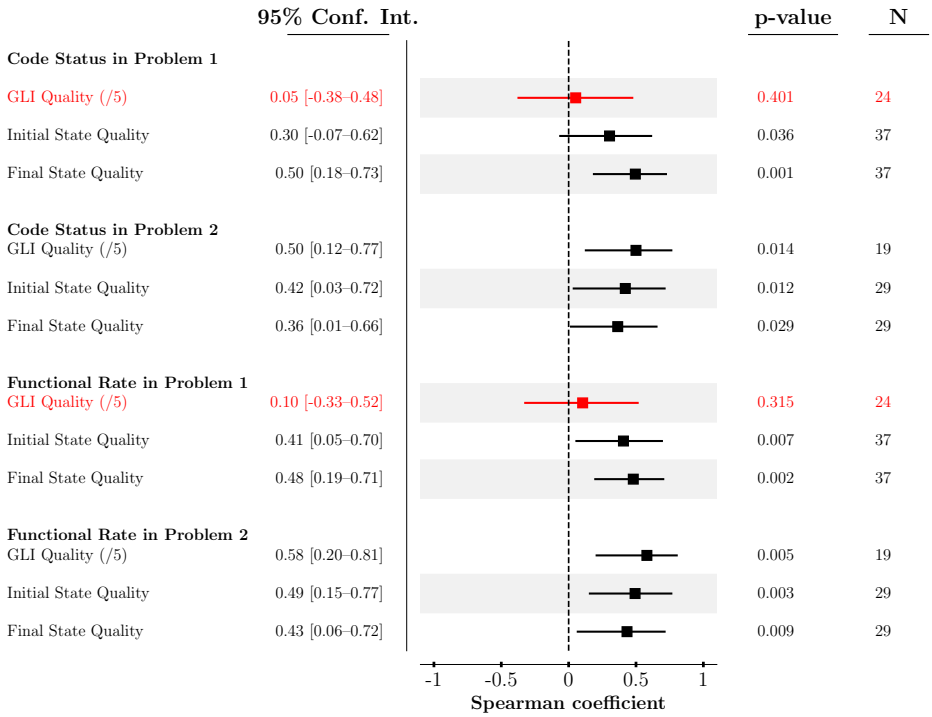


Figure 29. Exploratory one-tailed Spearman correlations between GLI quality and code metrics, with 95% bootstrap confidence intervals.

To complement the results to RQ2, Fig. 29 presents the Spearman correlation coefficients between the quality of the GLI diagrams and the code metrics (code status and functional rate), for both problems. As this analysis is exploratory, the reported one-tailed Spearman tests are intended to identify potential positive associations rather than provide confirmatory evidence. We performed a one-tailed test, as we assume a positive relationship between GLI quality and code quality, and confidence intervals were computed using a bootstrapping approach. The confidence intervals are reported to illustrate the uncertainty associated with the estimated correlations, especially in cases where the interval crosses zero. The sample size (N) for each correlation is also reported on the right of the figure. To compute these values, we used the *SciPy library* [8].

As an exploratory analysis, this first set of findings shows that better code status and higher functional rates generally correlate with higher-quality GLI representations. Most correlation coefficients vary from 0.30 to 0.58, reflecting small-to-medium effect

[8] https://docs.scipy.org/doc/scipy/tutorial/stats/

sizes according to Cohen's conventions; most, but not all, 95% CIs exclude zero. Two correlations carry high uncertainty. They are highlighted in red in the figure and correspond to the GLI (General State) quality in Problem 1 where, for both code status ($r = 0.05$, $95\%\,\text{CI} = [-0.38, 0.48]$, $p = 0.401$) and functional rate ($r = 0.10$, $95\%\,\text{CI} = [-0.33, 0.52]$, $p = 0.315$), the 95% CIs broadly span zero, indicating no meaningful relationship. Given the exploratory nature of this analysis, these results should be interpreted with caution and used to generate hypotheses rather than conclusions.

The uncertain relationships appearing in Fig. 29 are clarified in Fig. 30, detailing the frequency of each combination of values between the GLI quality (/5) and the functional rate (/5). It shows that, in Group B, eight students provided a GLI with reasonable quality (between 2.5 and 3) while being unable to provide complete code (leading to a functional rate equal to 0 and weakening any linear relationship between the two metrics). On the other hand, in Group C, two students were able to correctly code the solution while failing to provide a correct GLI. Both these students mentioned they spent less than five minutes on it. They likely preferred reasoning directly on the code without relying on the GLI.

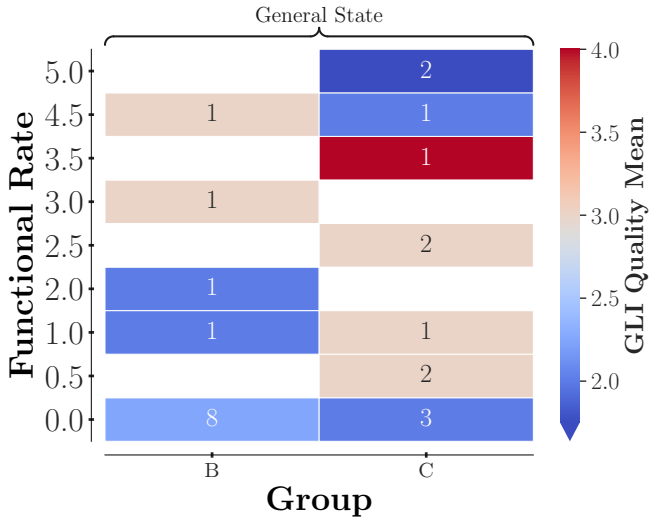


Figure 30. How the functional rate is related to the quality of the GLI for Problem 1? This pair of metrics was selected based on the red lines in Fig. 29. The numbers in each rectangle represent the number of students getting this code status/functional rate, in a given group.