# Learning from Others' Codes in Game-based Robot Programming: Behavioral Patterns and Outcomes

**Shintaro MAEDA[a*], Kento KOIKE[b], & Takahito TOMOTO[c]**
[a] *Graduate School of Information and Computer Science, Chiba Institute of Technology, Japan*
[b] *Faculty of Engineering, Tokyo University of Science, Japan*
[c] *Faculty of Innovative Information Science, Chiba Institute of Technology, Japan*
*front4.shintaro@gmail.com

**Abstract:** This study introduces a game-based programming environment for developing algorithmic thinking through iterative refinement. The system features "Score-based Gradual Worked Examples," allowing learners to view others' codes that achieved slightly higher scores. An analysis of behavioral logs from 49 university students using K-means clustering revealed seven distinct patterns. Learners who frequently referenced others' codes while making incremental changes showed greater learning gains. Expert evaluation suggests that presenting codes based on algorithmic similarity may be more effective than score-based selection alone. These findings can inform the design of adaptive support in programming education.

**Keywords:** Computational thinking, algorithmic thinking, worked examples, trial-and-error

## 1. Introduction

In programming education, students must develop not only syntax knowledge but also the ability to design and execute problem-solving procedures through computational thinking and algorithmic thinking (AT) (Lockwood et al., 2016; Wing, 2006). Visual programming environments like Code.org have shown promise in promoting AT through iterative trial-and-error processes (Choi, 2022; Papert, 1980). However, traditional puzzle-type tasks with single correct answers may limit continuous improvement once a working solution is found.

To address this limitation, we designed a game-based programming task that encourages score maximization rather than finding a single correct answer (Maeda et al., 2024a). This open-ended approach motivates learners to continuously refine their algorithms for better performance. However, novice learners require support, such as worked examples (WE) to progress effectively. Although WE can reduce cognitive load and facilitate learning (Sweller, 2020; Zhi et al., 2019), presenting complete solutions may lead to mere imitation without meaningful trial-and-error learning.

We propose "Score-based Gradual Worked Examples," where learners can view peer-generated codes that achieved slightly higher scores. This approach addresses two key challenges: (1) satisfaction with suboptimal solutions, by presenting incremental improvements rather than complete answers, and (2) limited support, by scaling support through peer-generated examples rather than instructor-authored ones. While previous classroom implementation showed overall score improvements (Maeda et al., 2024a), individual differences in system usage and learning outcomes remain unexplored. Therefore, this study analyzes behavioral patterns to address the following research questions (RQs). RQ1: In a class using the proposed step-by-step "Score-based Gradual Worked Examples," what trial-and-error characteristics are observed among learners, and how are these differences related to learning effectiveness? RQ2: For learners who do not achieve satisfactory learning outcomes even after viewing the WE, what factors might have influenced their learning?

## 2. Literature Review

AT—the ability to construct new algorithms to solve problems (Futschek, 2006)—requires learning environments that support repeated trial-and-error (Grover & Pea, 2013; Papert, 1980). Game-based platforms like Code.org and Scratch effectively promote AT through iterative challenges (Hsu & Wang, 2018), encouraging natural skill development through repeated practice. However, without clear guidance, learners may abandon trial-and-error when unable to understand or modify their programs. WE addresses this by reducing cognitive load through solution demonstrations (Sweller, 2020), which are proven to be effective in programming education (Zhi et al., 2019). Yet conventional WE remain static and uniform, limiting adaptation to individual needs. Learners also tend to skip high-quality examples without meaningful engagement (Zhi et al., 2019). Recent research has explored adaptive WE presentation methods. Toukiloglou and Xinogalos (2023) used fuzzy reasoning to adapt WE to learning progress; however, utilization was inconsistent and examples were fixed to specific problems rather than learner-constructed algorithms. Effective WE must be perceived as "helpful" (Muldner et al., 2023), yet learners often find them unhelpful when the "distance from the task" is too great (Wang et al., 2021). To address these limitations, our system automatically presents peer-generated code examples incrementally, matching examples to learners' current performance and facilitating continuous trial-and-error through contextually relevant support.

## 3. Method

### *3.1 Participants and Procedure*

This study analyzed log data from a classroom implementation that involved 49 second-year engineering students who used the system during a 40-minute session (10-min pre-test, 20-min system use with code viewing features, 10-min post-test). This approach has shown significant score improvements (Wilcoxon signed-rank test, $p < 0.005$, Cohen's $d > 0.8$) as reported in our previous work (Maeda et al., 2024a). To understand how different usage patterns of the "Score-based Gradual Worked Examples" feature influenced these learning outcomes, we extracted and analyzed behavioral indicators from the system logs.

### *3.2 System Description*

The learning environment requires learners to construct algorithms for a virtual robot that performs seeding and harvesting tasks. To maximize crop yield, learners must design effective action sequences and conditional branching, fostering AT development. The system scores and ranks algorithms, allowing learners to view others' codes that was "one rank higher"—a restriction ensuring examples remain sufficiently aligned with learners' current performance. Figure 1 shows the system features (Learners write code in C#).

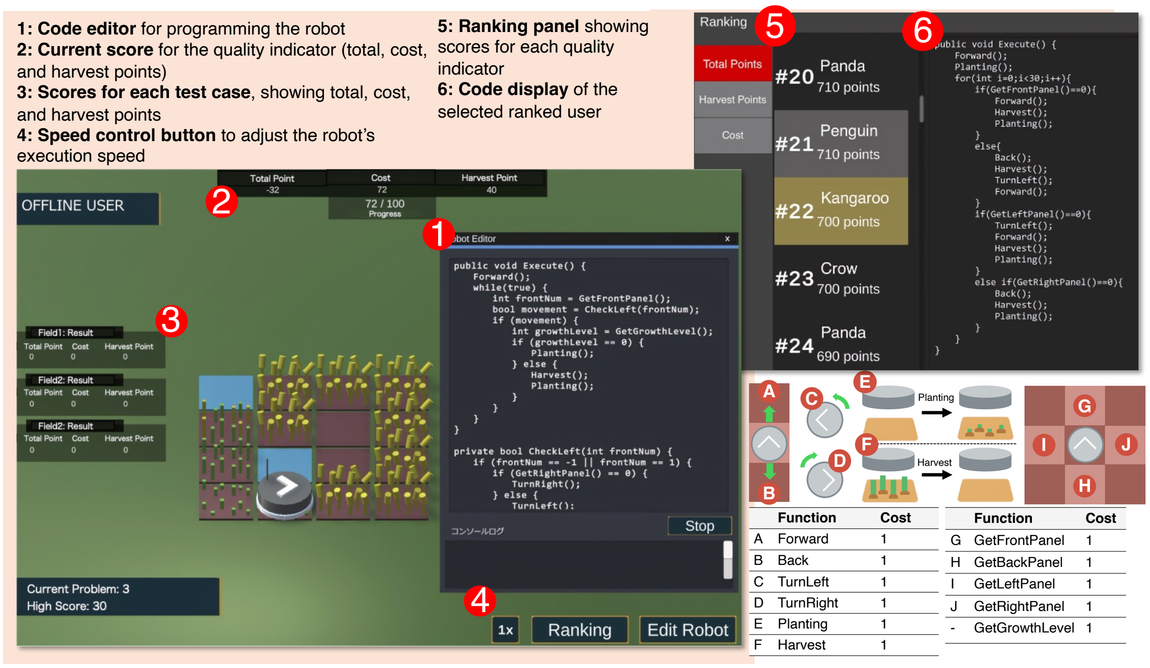
**Game-based programming task designed to encourage score acquisition:** The system is structured as a harvesting game with a virtual robot. Evaluation metrics like productivity, cost, and efficiency are used to assign scores. Productivity is indicated by the number of crops harvested, and cost is indicated by the number of robotic actions. Efficiency is indicated by the value obtained by subtracting cost from productivity (Maeda et al., 2024a). Improving one's score thus becomes equivalent to engaging in trial-and-error to develop more effective algorithms.

**Ranking feature:** The system displays each learner's score in relation to their peers. This approach not only promotes self-evaluation but also encourages the setting of new goals and increases motivation by emphasizing relative achievements rather than absolute performance.

**Code viewing function:** To foster continuous trial-and-error learning, learners are only allowed to view the code of another learner who holds a rank just above theirs. This "one-rank-higher" restriction ensures that the code being referenced is similar in complexity, making it more accessible and relevant.

Since the learners' code is used, differences in quality and syntax are expected to arise. Thus, beginners who are learning programming are expected to learn from others' codes who are at a similar level. Additionally, learners who already have some ability to write code are also expected to learn from the codes of others who are at a similar level. It should be noted that this system focuses only on the behavior of robots, or algorithmic evaluation (scores), and therefore does not evaluate syntactic style.

Figure 1 (left) shows the robot planting and harvesting interface, where harvest scores depend on crop growth. Since each command incurs costs, efficient algorithm design is essential. The total score equals harvest score minus operational costs. The ranking panel (right) displays peer scores and allows viewing code one rank above.


**1: Code editor** for programming the robot
**2: Current score** for the quality indicator (total, cost, and harvest points)
**3: Scores for each test case**, showing total, cost, and harvest points
**4: Speed control button** to adjust the robot's execution speed
**5: Ranking panel** showing scores for each quality indicator
**6: Code display** of the selected ranked user

```
public void Execute() {
    Forward();
    while(true) {
        int frontNum = GetFrontPanel();
        bool movement = CheckLeft(frontNum);
        if (movement) {
            int growthLevel = GetGrowthLevel();
            if (growthLevel == 0) {
                Planting();
            } else {
                Harvest();
                Planting();
            }
        }
    }
}

private bool CheckLeft(int frontNum) {
    if (frontNum == -1 || frontNum == 1) {
        if (GetRightPanel() == 0) {
            TurnRight();
        } else {
            TurnLeft();
```

```
public void Execute() {
    Forward();
    Planting();
    for(int i=0;i<30;i++){
        if(GetFrontPanel()==0){
            Forward();
            Harvest();
            Planting();
        }
        else{
            Back();
            Harvest();
            TurnLeft();
            Forward();
        }
        if(GetLeftPanel()==0){
            TurnLeft();
            Forward();
            Harvest();
            Planting();
        }
        else if(GetRightPanel()==0){
            Back();
            Harvest();
            Planting();
        }
    }
}
```

| | Function | Cost |
|---|---|---|
| A | Forward | 1 |
| B | Back | 1 |
| C | TurnLeft | 1 |
| D | TurnRight | 1 |
| E | Planting | 1 |
| F | Harvest | 1 |

| | Function | Cost |
|---|---|---|
| G | GetFrontPanel | 1 |
| H | GetBackPanel | 1 |
| I | GetLeftPanel | 1 |
| J | GetRightPanel | 1 |
| - | GetGrowthLevel | 1 |

*Figure 1.* Example of the Code Visualization Interface and Ranking System.

### 3.3 Overview and Selection of Behavioral Indicators

To understand system usage patterns and their impact on learning, we analyzed behavioral log data using seven key indicators: (1) #Exec: Code execution count after modifications (excluding repeated runs) (2) ΔTotal: Cumulative behavioral changes across executions (3) Δ/Exec: Average behavioral change per execution (ΔTotal/#Exec) (4) Maximum score: Highest total score achieved (5) #Ref: Frequency of viewing others' codes during learning (6) Ref/Exec: Reference rate per execution (#Ref/#Exec) (7) Learning gain: Normalized gain (Hake, 1998): g = (post−pre-score)/(max−pre-score), where max = 3284 points. These metrics collectively assess trial-and-error authenticity and the utilization of others' code examples from multiple perspectives.

## 4. Results

Table 1 presents the basic statistics for each indicator across three phases: pre (Pre), during system-use learning (Learn), and post (Post). While learners cannot access WE features in the Pre and Post phases, these features were enabled in the Learn phase. Additionally, the same questions were used for Pre, Post, and Learn. #Exec increased from 13.2 (Pre) to 24.2 (Learn) and 15.0 (Post), with the Learn phase showing higher activity likely due to having double the exercise time (20 vs. 10 minutes).

Reference frequency (#Ref) varied widely (mean 3.8, SD 3.4), but reference rate per execution (Ref/Exec) remained stable at approximately 20%. While average scores improved significantly from 79.4 to 1,279.1, suggesting learning effectiveness, the relationship between behavioral indicators and achievement remained unclear from basic statistics alone, necessitating further analysis.

Table 1. *Basic Statistics*

| | **Pre** | | **Post** | | **Learn** | | **Pre-Post** | |
|---|---|---|---|---|---|---|---|---|
| | **Mean (SD)** | **Min-Max** | **Mean (SD)** | **Min-Max** | **Mean (SD)** | **Min-Max** | **Mean (SD)** | **Min-Max** |
| #Exec | 13.2 (10.3) | 0.0-48.0 | 24.2 (14.8) | 4.0-68.0 | 15.0 (8.8) | 4.0-43.0 | - | - |
| ΔTotal | 4.0 (3.2) | 0.0-17.4 | 8.0 (5.3) | 0.3-27.4 | 4.5 (3.1) | 0.0-16.7 | - | - |
| Δ/Exec | 0.3 (0.2) | 0.0-0.6 | 0.3 (0.1) | 0.1-0.6 | 0.3 (0.2) | 0.0-0.7 | - | - |
| Max Score | 79.4 (435.5) | –676.0 -1568.0 | 815.4 (721.7) | –500.0 -2760.0 | 1279.1 (924.2) | –370.0 -3284.0 | -- | - |
| Learning Gain | - | - | - | - | - | - | 0.4 (0.3) | -0.1 (-1.0) |

Note: Negative scores may occur when harvesting fails or when movement costs exceed the harvest value

K-means clustering was performed using five normalized indicators (#Exec, ΔTotal, Δ/Exec, Score, #Ref), yielding seven clusters based on silhouette score optimization. Table 2 presents the behavioral characteristics of each cluster.

Table 2. *Behavioral Indicators in Each Cluster*

| **Cl.** | **Name** | **N** | **#Exec** | **ΔTotal** | **Δ/Exec** | **Max Score** | **#Ref** |
|---|---|---|---|---|---|---|---|
| 0 | HTR | 2 | 52.5(0.7) | 12.5(0.5) | 0.2(0.0) | 669.0(420.0) | 10.0(1.4) |
| 1 | MAT | 17 | 17.3(5.2) | 6.4(1.9) | 0.4(0.1) | 566.8(366.3) | 2.4(1.9) |
| 2 | PAS | 6 | 9.7(5.1) | 2.7(1.4) | 0.2(0.1) | 29.3(319.4) | 0.8(1.0) |
| 3 | SIR | 9 | 22.0(5.2) | 10.3(2.8) | 0.5(0.1) | 1154.9(504.2) | 7.3(2.8) |
| 4 | EFS | 4 | 13.3(6.5) | 2.8(2.1) | 0.2(0.1) | 2194.0(517.1) | 6.0(2.9) |
| 5 | PTM | 7 | 44.3(12.3) | 7.6(2.5) | 0.2(0.1) | 1004.6(913.6) | 2.0(2.0) |
| 6 | BRV | 4 | 42.3(11.1) | 20.9(5.3) | 0.5(0.1) | 650.5(418.2) | 4.3(4.8) |

Three high-performing clusters emerged: Self-improver & Reference (SIR) showed high Ref/Exec and Δ/Exec, making incremental code modifications while frequently referencing others' codes. Efficient-Strategist (EFS) achieved the highest scores despite low #Exec, suggesting effective use of others' codes within limited trials. Heavy-trial & Reference (HTR) had high #Exec and #Ref but low reference density, limiting its effectiveness.

Four clusters showed limited success: Passive (PAS) exhibited minimal engagement across all metrics. Persistent-trial Mixed (PTM) and Moderate-activity (MAT) both showed high trial activity but low reference usage, with MAT (n=17) displaying conscious algorithmic modifications but variable outcomes. Bulk-revision (BRV) made large code changes (high ΔTotal and Δ/Exec) but with moderate reference usage, potentially losing focus through excessive modifications.

A one-way ANOVA revealed significant differences in learning outcomes between clusters ($F(6,42)=4.781$, $p=0.0008$, $\eta^2=0.4058$). Post-hoc analysis (Tukey's HSD) showed EFS significantly outperformed PAS ($p=0.0005$), while MAT outperformed EFS ($p=0.0016$). Despite lower #Exec and ΔTotal, EFS achieved the highest scores through high Ref/Exec—frequently referencing others' codes within limited executions while making incremental modifications. These findings suggest that reference density (Ref/Exec) rather than trial quantity (#Exec) is the key factor for improving learning outcomes, though causal relationships require further investigation given the small cluster sizes.

## 5. Discussion and Future work

Cluster analysis revealed that learning effectiveness depended more on the quality of the trial-and-error than quantity. EFS and SIR clusters demonstrated that high reference density (Ref/Exec) combined with incremental modifications led to superior outcomes, despite fewer overall trials. Conversely, HTR showed that frequent references without meaningful integration yielded limited gains, while PAS's minimal engagement and MAT/BRV's low reference usage resulted in poor performance.

These findings address both RQs. For RQ1, reference density emerged as more critical than trial frequency for effective learning. For RQ2, HTR's case demonstrated that reference volume alone is insufficient—the quality of engagement matters. This aligns with Wang et al.'s (2021) identified barriers: comprehension difficulties and modification challenges, which intensify when the presented code significantly differs from learners' own solutions. These barriers likely influenced the observed behavioral patterns and outcomes, suggesting that code examples should be closely aligned with learners' current approaches to facilitate effective adoption.

To reduce learning barriers, we propose presenting codes based on two similarity types: (A) syntactic format and (B) algorithmic proximity—aligning with code clone classifications where Types I-III represent textual similarity and Type IV represents functional similarity (Roy & Cordy, 2007). Our findings suggest that algorithmic similarity facilitates better understanding and integration into learners' trial-and-error processes.

In a preliminary study (Maeda et al., 2024b), we quantified algorithmic similarity using F-values based on execution behaviors and presented both algorithm-based and conventional score-based rankings to three expert programmers (a professor specializing in programming education and two industry professionals). Expert evaluation strongly favored algorithm-based ranking (mean: 5.67/6) over conventional ranking (mean: 3.67/6). Experts noted that algorithmically similar codes were "helpful for development" and "used the same strategy," while conventional rankings showed "differences too large for beginners" risking "code copying rather than learning." Thus, algorithmic similarity-based presentation could effectively replace the current "one-rank-higher" restriction.

Recent advances in deep learning, such as SANN for generating code vectors (Hoq et al., 2025), enable presentation of syntactically similar programs that match learners' coding levels rather than expert syntactic patterns. Combining both similarity dimensions could reduce code quality as a confounding factor and enable clearer behavioral pattern interpretation. This understanding would support personalized strategies, such as presenting examples aligned with learners' behavioral tendencies or offering alternative approaches when learners stagnate—ultimately enabling more adaptive, individualized learning support.

This observational study has some limitations. First, it cannot establish causal relationships between code viewing and learning improvements. While supplementary analysis (Maeda et al., 2025) showed score improvements when learners incorporated elements from others' codes, it remains unclear which code types are most "adoptable" and "effective." Therefore, the hypothesis that algorithmically and syntactically similar codes facilitate learning requires further empirical validation. Further, some clusters, such as HTR and BRV, remain an issue for the future due to their small sample sizes. Second, this study evaluated a single robot programming task, limiting generalizability to other programming domains or diverse learner populations, which warrants future investigation. The algorithmic similarity approach requires validation through actual learner testing beyond expert evaluation.

This study has implications for future research. First, it is not possible to evaluate how learners read and understand the code of others. In contrast, in evaluations based on previous research, a survey was conducted on learners who viewed the codes of others to investigate how their own code changed, and it was confirmed that a certain number of learners incorporated parts of others codes into their own, in a way that was not imitation (Maeda et al., 2025). Furthermore, this led to an improvement in scores. Another avenue for future research is investigating the learning effects of sharing others' codes with large score differences.

## Acknowledgements

This work was supported by JSPS KAKENHI Grant Number JP25K21362.

## References

Choi, W. C. (2022). The influence of code.org on computational thinking and learning attitude in block-based programming education. *Proceedings of the 2022 6th International Conference on Education and E-Learning*, 235–241. https://doi.org/10.1145/3578837.3578871

Futschek, G. (2006). Algorithmic thinking: The key for understanding computer science. In R. T. Mittermeir (Ed), *Informatics Education – The Bridge between Using and Understanding Computers (Vol.* 4226, pp. 159–168*). Springer Berlin Heidelberg*. https://doi.org/10.1007/11915355_15

Grover, S., & Pea, R. (2013). Computational thinking in K–12: A review of the state of the field. *Educational Researcher*, *42*(1), 38–43. https://doi.org/10.3102/0013189X12463051

Hake, R. R. (1998). Interactive-engagement versus traditional methods: A six-thousand-student survey of mechanics test data for introductory physics courses. *American Journal of Physics*, *66*(1), 64–74. https://doi.org/10.1119/1.18809

Hoq, M., Patil, A., Akhuseyinoglu, K., Brusilovsky, P., & Akram, B. (2025). An automated approach to recommending relevant worked examples for programming problems. *Proceedings of the 56th ACM Technical Symposium on Computer Science Education*, *1*, 527–533. https://doi.org/10.1145/3641554.3701951

Hsu, C.C., & Wang, T.I. (2018). Applying game mechanics and student-generated questions to an online puzzle-based game learning system to promote algorithmic thinking skills. *Computers & Education*, *121*, 73–88. https://doi.org/10.1016/j.compedu.2018.02.002

Lockwood, E., & Asay, A. (2016). Algorithmic thinking: An initial characterization of computational thinking in mathematics. *North American Chapter of the International Group for the Psychology of Mathematics Education*.

Maeda, S., Koike, K., & Tomoto, T. (2024a). Code visualization system for writing better code through trial and error in programming learning: Classroom implementation and practice. *International Conference on Computers* in *Education*. https://doi.org/10.58459/icce.2024.5006

Maeda, S., Koike, K., & Tomoto, T. (2024b). Code-sharing platform in programming learning: A proposal for a strategy-aware code-sharing methodology. *International Conference on Human-Computer Interaction*, 338–348.

Maeda S., Koike K., & Tomoto T. (2025). Proposal and classroom practice of game task type virtual robot programming to promote writing “better code” through trial and error [Manuscript submitted for publication].

Muldner, K., Jennings, J., & Chiarelli, V. (2023). A review of worked examples in programming activities. *ACM Transactions on Computing Education*, *23*(1), 1–35. https://doi.org/10.1145/3560266

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. Basic Books, Inc.

Roy, C. K., & Cordy, J. R. (2007). A survey on software clone detection research. *Queen’s School of Computing Technical Report* 2007-541. Queen’s University, Canada.

Sweller, J. (2020). Cognitive load theory and educational technology. *Educational Technology Research and Development*, *68*(1), 1–16. https://doi.org/10.1007/s11423-019-09701-3

Toukiloglou, P., & Xinogalos, S. (2023). Adaptive support with working examples in serious games about programming. *Journal of Educational Computing Research*, *61*(4), 766–789. https://doi.org/10.1177/07356331231151393

Wang, W., Kwatra, A., Skripchuk, J., Gomes, N., Milliken, A., Martens, C., Barnes, T., & Price, T. (2021). Novices’ learning barriers when using code examples in open-ended programming. *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education*, *Vol.* 1, 394–400. https://doi.org/10.1145/3430665.3456370

Wing, J. M. (2006). Computational thinking. *Communications of the ACM*, *49*(3), 33–35. https://doi.org/10.1145/1118178.1118215

Zhi, R., Price, T. W., Marwan, S., Milliken, A., Barnes, T., & Chi, M. (2019). Exploring the impact of worked examples in a novice programming environment. *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 98–104. https://doi.org/10.1145/3287324.3287385