Computers & Education 95 (2016) 202–215

Contents lists available at ScienceDirect

Computers & Education

journal homepage: www.elsevier.com/locate/compedu

Exploring students' computational practice, design and performance of problem-solving through a visual programming environment

Po-Yao Chao [a, b, *]

[a] Department of Information Communication, Yuan Ze University, Taiwan
[b] Innovation Center for Big Data and Digital Convergence, Yuan Ze University, Taiwan

ARTICLE INFO

Article history:
Received 7 September 2015
Received in revised form 24 January 2016
Accepted 25 January 2016
Available online 28 January 2016

Keywords:
Computer programming
Visual problem solving
Students programming patterns

ABSTRACT

This study aims to advocate that a visual programming environment offering graphical items and states of a computational problem could be helpful in supporting programming learning with computational problem-solving. A visual problem-solving environment for programming learning was developed, and 158 college students were conducted in a computational problem-solving activity. The students' activities of designing, composing, and testing solutions were recorded by log data for later analysis. To initially unveil the students' practice and strategies exhibited in the visual problem-solving environment, this study proposed several indicators to quantitatively represent students' computational practice (*Sequence*, *Selection*, *Simple iteration*, *Nested iteration*, and *Testing*), computational design (*Problem decomposition*, *Abutment composition*, and *Nesting composition*), and computational performance (*Goal attainment* and *Program size*). By the method of cluster analysis, some empirical patterns regarding the students' programming learning with computational problem-solving were identified. Furthermore, comparisons of computational design and computational performance among the different patterns of computational practice were conducted. Considering the relations of students' computational practice to computational design and performance, evidence-based suggestions on the design of supportive programming environments for novice programmers are discussed.

1. Introduction

Programming has been recognized as one of the important competencies that require students to use computational tools to address real-world problems in the 21st century (Einhorn, 2011; Grover & Pea, 2013; Yen, Wu, & Lin, 2012). Learning programming is not only a prerequisite for becoming a computer scientist, but it is also necessary for the practice of solving problems and designing systems (Palumbo, 1990; Robins, Rountree, & Rountree, 2003). Programming requires programmers to plan solutions to problems, precisely transform the plans into syntactically correct instructions for execution, and assess the consequent results of executing those instructions (Brookshear, 2003; Deek, 1999; Ismal, Ngah, & Umar, 2010). However, research revealed that at the conclusion of introductory programming courses, most students have difficulties in decomposing problems, developing plans and implementing their plans with programming languages to solve programming

* Department of Information Communication, Yuan Ze University, 135 Yuan-Tung Road, Chung-Li 32003, Taiwan.
E-mail address: poyaochao@saturn.yzu.edu.tw.

http://dx.doi.org/10.1016/j.compedu.2016.01.010
0360-1315/

problems (Lister et al., 2004; McCracken et al., 2001; de Raadt, 2007; Robins et al., 2003). Some of them lack adequate understanding of fundamental programming constructs, and most of them lack strategies for transforming programming problems into workable plans and algorithms (Deek, 1999; Kessler & Anderson, 1986; Li & Watson, 2011; de Raadt, 2007). This may be because the formal instruction in programming mostly focuses on students' mastery of a general-purpose programming language and adopts a programming tool that is intentionally designed for professional programmers (Deek, 1999; Ismal et al., 2010; Linn & Clancy, 1992; Robins et al., 2003; Xinogalos, 2012). The employment of the general-purpose programming language and the professional programming tool often drives the teachers and students to invest their efforts more on mastering programming language features than on developing design strategies for solving programming problems (Brusilovsky, Calabrese, Hvorecky, Kouchnirenko, & Miller, 1997; Deek, 1999; Linn, 1985; Pears et al., 2007).

Numerous studies have been devoted to research on instructional and environmental assistance for programming learning (Kelleher & Pausch, 2005; Winslow, 1996). Among the studies aiming to devise potential means for enhancing programming, an alternative approach to engaging students in solving computational problems (Edmonds, 2008) has been recognized as an effective way of cultivating students' programming constructs and skills (Liu, Cheng, & Huang, 2011; Ring, Giordan, & Ransbottom, 2008). This method often provides students with computational problems, which are specially designed to foster specific programming concepts or skills. In a scenario requiring students to solve a computational problem by exercising various programming knowledge and strategies, the students are expected to learn by formulating computer programs and systematically evaluating the consequent results (Deek, 1999). Many studies have also proposed alternative approaches to the students' difficulties in programming by the use of visual programming environments, such as Scratch and Alice (Cooper, Dann, & Pausch, 2000; Maloney, Resnick, Rusk, Silverman, & Eastmond, 2010), LighBot and PlayLOGO 3D (Gouws, Bradshaw, & Wentworth, 2013; Paliokas, Arapidis, & Mpimpitsos, 2011), or objectKarel and Jeroo (Sanders & Dorn, 2003; Xinogalos, 2012). These environments often adopt different visual programming elements that help novice programmers construct their programs or understand the process of program execution and the state of a problem (Green & Petre, 1996; Kelleher & Pausch, 2005; Navarro-Prieto & Canas, 2001). Research has revealed that visual programming environments could enhance novice programmers' engagement in programming tasks and help them demonstrate programming skills and problem solving strategies during the course of creating digital artifacts or solving programming problems (Cooper et al., 2000; Lye & Koh, 2014). Although visual programming environments are becoming important and have demonstrated their particular benefits to assist learning programming and problem solving (Lye & Koh, 2014), little is known about how novice programmers use a visual programming environment to learn to solve computational problems. Moreover, because constructing a computer program to solve a computational problem in a visual programming environment requires novice programmers to manipulate visual programming elements (e.g., control-flow blocks) to formulate and test a design solution to the problem (e.g., Gouws et al., 2013; Maloney et al., 2010), the programmers' behavior and strategies of solving computational problems in a visual programming environment may affect their performance of problem solving. Therefore, there is a need to further explore the novice programmers' behavioral patterns in a visual programming environment and investigate the difference in their strategies and performance of solving computational problems among different behavioral patterns.

Based on the aforementioned rationale, the purpose of this study is twofold. The first is to develop a visual problem-solving environment for programming learning and explore how novice programmers use it to learn to solve computational problems through interacting with the provided visual programming elements. To understand how novice programmers interact with the proposed environment to solve computational problems, the second purpose is to investigate novice programmers' visual programming behavior and strategies of computational problem-solving enacted in the visual problem-solving environment, as well as to examine the performance of computational problem solving among different patterns of visual programming behavior in computational problem solving activities. To this end, this study aims to answer the following research questions:

- What are the novice programmers' patterns of visual programming behavior exhibited in a visual programming environment to solve computational problems?
- Do the novice programmers' computational design and performance of solving computational problems differ in different patterns of visual programming behavior?

The results of this study may be of interest to interface designers attempting to design a specific programming learning environment for novice programmers. The results may also particularly interest teachers or educators who design formal instruction for students to foster their programming strategies (de Raadt, Watson, & Toleman, 2009) or computational problem solving skills.

2. Related works

2.1. Programming learning in visual programming environments

Many visual programming environments have been developed to provide novice programmers with visual supports in constructing programs and understanding programming constructs. For example, ToonTalk enables a programming environment in which users interact with visual objects, such as birds or cars in a city, to construct programs (Kahn, 1996). These

visual objects may assist users in understanding their programs by examining the state (e.g., appearance, behavior, or location) of the visual objects. However, the Scratch environment adopts a visual block language allowing users to construct scripts by dragging-and-dropping the language blocks and provides visual feedback showing the execution of scripts for the novices to comprehend how they work (Maloney et al., 2010). The LightBot environment proposed by Gouws et al. (2013) comprises a set of iconic instructions to help users compose programs. The environment also includes a visual robot to assist users in understanding programming constructs by allowing them to observe how the composed program would change the robot's behavior. The above programming environments share prominent features of visual programming languages and program visualization systems (Navarro-Prieto & Canas, 2001). The visual programming language provides novice programmers with visual elements, such as icons or symbols, to specify a program by dragging-and-dropping the visual elements instead of entering text-based statements. This may resolve novice programmers' difficulties in the syntax of programming languages and allows them to focus on the logic and structure involved in programming (Holvikivi, 2010; Kelleher & Pausch, 2005; Lye & Koh, 2014; Navarro-Prieto & Canas, 2001). However, the program visualization systems embody graphical elements and visual models to help novice programmers formulate conceptual ideas and realize the process and the consequence of executing certain computer instructions (Green & Petre, 1996; Navarro-Prieto & Canas, 2001). Thus, the visual programming environment could lower the barriers to programming, which can then assist novice programmers in developing and performing programming strategies (Kelleher & Pausch, 2005; Lye & Koh, 2014). It could be suggested that the use of visual elements assisting in constructing or understanding programs has great potential for novice programmers' practices and strategies of problem solving for programming learning.

Structured programming languages, such as C++ or Java, are widely adopted by engineers or educators to solve problems or teaching introductory programming (Pears et al., 2007). The representation of a computer program in a structured programming language is influenced by Pascal language, which was proposed by Niclaus Wirth and designed to facilitate development of computer programs by using basic control flow structures (Wirth, 1971). With a structured programming language, a computer program can be represented by the combination of three control flow structures: sequence, selection, and repetition (Böhm & Jacopini, 1966; Deitel & Deitel, 2011). The ability of applying the control flow structures and combining the structures in a meaningful way is fundamental and crucial to solving a computational problem through programming (Grover & Pea, 2013). Among the control flow structures, the sequence represents a list of step-by-step instructions to be followed one after the other (Brennan & Resnick, 2012; Deitel & Deitel, 2011). Sequencing characterizes a component of planning involving arranged actions or computer instructions in the order that produces accurate effects (Bers, Flannery, Kazakoff, & Sullivan, 2014; Suthers, 1994) and is essential to many disciplines, such as literacy or mathematics (Bers et al., 2014). The selection control flow structure represents the if-then like control structure that specifies conditions to execute different parts of computer instructions (Brennan & Resnick, 2012; Deitel & Deitel, 2011). It involves the establishment of relationships between conditions and the corresponding computer instructions, which serves as a crucial element of conditional and logical reasoning (Kurland, Pea, & Clement, 1986; Seidam, 1981). The repetition control flow structure enables the process of repeating a set of computer instructions, which requires the construction of an iterative plan involving the identification of the repeating instructions and the condition governing the end or continuation of the repetition (Brennan & Resnick, 2012; Deitel & Deitel, 2011; Rogalski & Samurcay, 1990). It could contribute to algorithmic thinking by engaging learners in recognizing patterns of repetition (Futschek & Moschitz, 2011). Visual programming environments are believed to support the understanding and adoption of the control flow structures because the environments contain visual elements to reduce unnecessary syntax difficulties and assist novices in visualizing the effects of the adopted control flow structures (Kelleher & Pausch, 2005; Lye & Koh, 2014; Navarro-Prieto & Canas, 2001). On the other hand, testing solutions is a prerequisite for success in programming, representing the validation of the computer program immediately after the program has been modified by the user (Deek, 1999; Gourlay, 1983). It involves the users' comprehension of the computer program and anticipation of how the modified program would work (Cross, Hendrix, & Barowski, 2002; McCauley et al., 2008). In other words, the comprehension of a modified program may reduce the need to test the program frequently (Green & Petre, 1996). In this study, the aforementioned control flow structures and the testing activity were employed to facilitate students' development of different methods of applying control flow structures and support students' learning of constructing computer programs by solving computational problems. Furthermore, by developing a visual programming environment that facilitates and visualizes the computational practice, the study could gain understanding of how novice programmers solve computational problems with the aid of the graphical elements.

2.2. Problem solving in visual programming environments

Strategies of problem solving, such as planning or designing solutions, are considered to be core competencies of computer science education and are central to programming success (Barg et al., 2000; Fee & Holland-Minkley, 2010; Hazzan, Lapidot, & Ragnis, 2011; Pears et al., 2007). Problem solving strategies for programming could be domain-specific because they address the need to use techniques, such as structured decomposition, to analyze given facts, such as input and output and to formulate steps leading to a problem's solution (Deek, 1999; Ismal et al., 2010). In other words, problem solving through programming requires programmers to use both problem solving strategies and program development skills (Deek, 1999; Gomes & José, 2007; Hazzan et al., 2011; Linn & Clancy, 1992). Therefore, to cultivate these strategies and skills, problem solving approaches are often adopted to assist novice programmers in learning programming (e.g., Gomes & José, 2007; Kiesmüller, 2009; Ring et al., 2008). The approaches typically include computational (or algorithmic) problems that are

carefully designed to enable a situation where a learner should apply particular sets of programming concepts (e.g., sequences, loops, or conditionals) or strategies (e.g., finding an average) to solve the problems. In this situation, learners are asked to identify the initial goals and states of a problem and then formulate an algorithm specifying a series of steps to solve the problems (Deek, 1999; Hazzan et al., 2011).

A visual programming environment has the potential to support problem solving because it provides graphical elements representing problems or program states for a novice to comprehend a problem and assess the current program state (Cooper et al., 2000; Kelleher & Pausch, 2005; Navarro-Prieto & Canas, 2001). For example, many visual programming environments, such as Jeroo (Sanders & Dorn, 2003), LightBot (Gouws et al., 2013), or Hour of Code (Code.org, 2015), embody various virtual worlds in which graphical elements serve as integral parts of a computational problem. By interacting with the graphical elements, learners are more likely to identify important features of a computational problem and compose their planned structures for the problem (Lye & Koh, 2014; Navarro-Prieto & Canas, 2001). Given the importance of problem-solving strategies and the potential of graphical elements, it is suggested that the core design of the visual programming environment should consider support for cultivating problem solving strategies and the inclusion of graphical elements to facilitate problem solving processes.

According to Deek (1999) and Hazzan et al. (2011), problem solving through programming involves the strategies related to planning and designing solutions. The planning strategy assists programmers in decomposing a problem goal into a collection of intermediate sub-goals, which may be further decomposed into more fine-grained sub-goals (Hazzan et al., 2011; Ismal et al., 2010). The research shows that programmers tend to formulate their plans and goal structures based on their knowledge of programming plans (Rist, 1991; Robins et al., 2003). The programming plans (or algorithmic patterns) are segments of code to achieve a common goal (Davies, 1993; Muller & Haberman, 2008; Rist, 1991). Given a computational problem, the number of sub-goals may reflect the granularity of the decomposition performed by programmers to tackle the problem. Based on stepwise refinement, the larger is the number of sub-goals, the finer-grained the produced sub-problems are (Pennington & Grabowski, 1990). On the other hand, the design strategies could help programmers organize and refine the components of their solution strategies and transform each sub-goal into corresponding algorithms (Deek, 1999; Hazzan et al., 2011; Soloway, 1986). In this study, visual programming elements that denote parts of a solution are employed to represent the components of solution plans in the visual programming environment. A problem solver organizes and refines the visual programming elements to formulate their design of solution plans.

According to Soloway (1986), several useful composition strategies are commonly adopted in organizing programming plans to achieve sub-goals. Among these strategies, abutment and nesting are two important plan composition methods. The abutment strategy represents the method of gluing two programming plans together back to front in sequence (Soloway, 1986). This involves applying programming plans and creating a sequential relationship between plans during the transformation of sub-goals into algorithms. The nesting strategy represents the method of embedding one programming plan into another (Soloway, 1986). The nesting of programming plans often enables a new programming plan that has a different structure from the previous two programming plans. Considering problem solving through programming, technical supports in the decomposition of a problem and the composition of programming plans may have potential in assisting novices to learn the planning and designing of solutions. Therefore, in this study, the aforementioned planning and design strategies for solving computational problems are supported and employed to represent different design strategies of computational problem solving in the visual programming environment.

Although many visual programming tools have been proposed in previous studies (e.g., Flannery et al., 2013; Gouws et al., 2013; Kölling, 2010; Maloney et al., 2010; Xinogalos, 2012), there is little research examining novice programmers' computational practice, design, and performance of solving a computational problem in a visual programming environment. Furthermore, exploring how novice programmers of different computational practice patterns utilize a visual programming environment to solve problems may provide insights into the development of more supportive environments for assisting novice programmers in learning programming and problem solving.

3. Method

3.1. The visual problem-solving environment for programming learning

There are many types of visual programming environments for novice programmers that highlight the distinct benefits to programming behaviors and problem-solving strategies. Nevertheless, this study focuses on the exploration of how novice programmers use the provided visual programming elements to learn by solving a computational problem. To explore novice programmers' visual programming behaviors and strategies, a visual problem-solving environment for programming learning was specifically designed and developed in line with the research purposes of this study. In the environment, a novice is allowed to learn the process of solving a computational problem from example instructions on how a robot picks flowers or cleans barriers (e.g., stones) for a farmer. As shown in Fig. 1, the environment can be divided into four main parts: the robot's world, the instruction library, the program composer, and the execution button. First, the robot's world includes a robot, a farmer, instruction cards, and a set of flowers or stones. The robot performs actions (e.g., picking flowers or moving to specific places) based on the computer instructions contained in the instruction cards. A novice can place the instruction cards in the appropriate grid cells to inform the robot to perform the actions in different regions of the robot's world. Second, the instruction library contains visualized instruction blocks representing the computer instructions to control the robot's

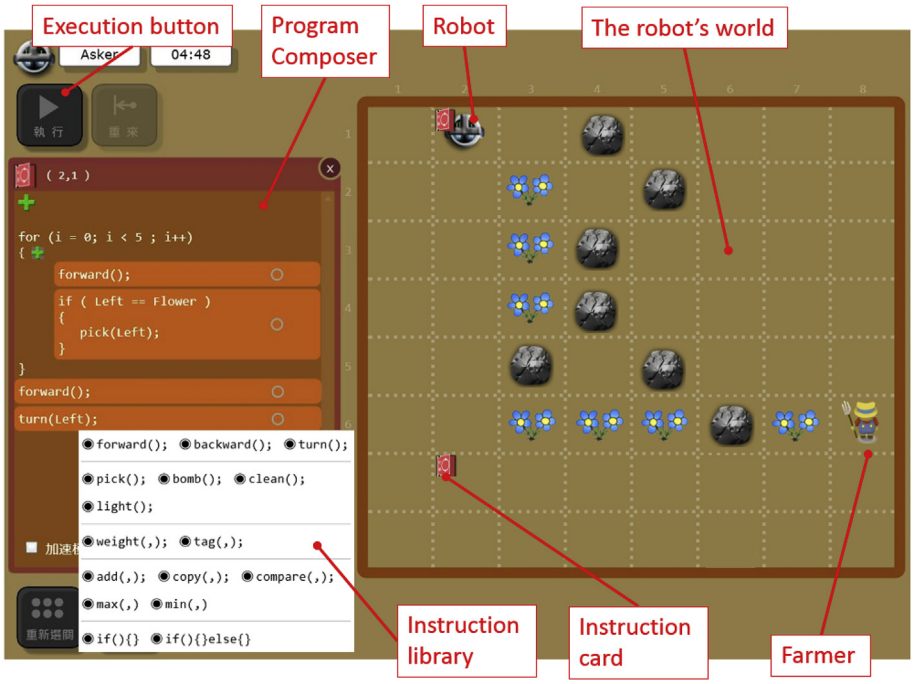

Fig. 1. The interface of the visual problem-solving environment for programming learning.

behavior. Because the environment currently focuses on the supports in learning of basic control flow principles and mechanisms, there are four control-flow blocks (sequence, selection, simple iteration, and nested iteration), designed according to the aforementioned fundamental control flow structures (Böhm & Jacopini, 1966; Deitel & Deitel, 2011), and 6 command blocks (move forward, move backward, change direction, light signals, pick flowers, and clean grass) in the instruction library. In this regard, the aforementioned iteration control-flow structure is further divided into simple and nested iteration control-flow blocks because the two types of control-flow are quite different in their repetition structures (Moreno, 2012). It is appropriate to purposefully identify them as two different control-flow structures for novice learners. Third, the program composer is an editor window where a novice assembles computer instructions contained in an instruction card. The assembly involves selection or drag-and-drop of the visualized instruction blocks to represent a chuck of computer instructions. An instruction card in the visual programming environment is a user-created visual programming elements used to help a novice plan the structure of a solution by dividing a solution into different parts. Each part of the solution is represented by an instruction card. A novice inserts visualized instruction blocks into an instruction card to implement a solution plan. For example, as shown in Fig. 1, a user created two instruction cards to navigate and collect flowers based on the distribution of flowers. Each instruction card contains chunks of computer instructions dealing with flowers or stones within a region in the robot's world. In this example, the user decomposes a goal of collecting all flowers into two sub-goals with each of them dealing with part of the flowers within a region. The user's instruction cards together form a complete computer program that gives instructions to the robot on picking flowers. Finally, the execution button is a graphical element that enables the robot to execute the computer instructions in the instruction cards. The environment provides novice programmers with various kinds of visual feedback to help them comprehend how the computer instructions would move the robot and change the states of the graphical objects in the robot's world. The environment also provide the novices with feedback describing their performance of solving computational problems in terms of the number of flowers collected and the number of visualized instruction blocks used. The feedback also provides the comparison between the requirements of a computational problems and a novice's performance, which may help the novices assess the effectiveness of their design solutions.

The proposed visual programming environment, such as JEROO or Karel the robot (Pattis, Roberts, & Stehlik, 1995; Sanders & Dorn, 2003), employs various visual elements and feedback to assist novices to implement computer programs and understand programming concepts. Additionally, to enable the approach that incorporates programming learning with problem solving, the proposed environment further provides visual programming elements that assist novice programmers to represent design strategies and offers feedback of assessment that helps novices evaluate their design solutions.

The aforementioned features of the visual programming environment are dedicated to promoting programming learning in a problem-solving manner. The environment provides a novice with computational problems that require the robot to collect all the flowers in the robot's world. To solve the computational problem, the novice must examine the distribution of flowers in the robot's world and determine whether there are any recognizable patterns of the distribution, such as the

repetition of flowers. After that, the novice may start to set a goal of reaching all flowers and may attempt to decompose the goal into sub-goals. For each sub-goal, the novice can create an instruction card and assemble the visualized instruction blocks to implement plans for the sub-goals. For example, Fig. 2 shows different design solutions of collecting all flowers for a computational problem. Fig. 2(a) shows the design that divides the L-shape region containing flowers and stones into two sub-regions (denoted by two double-arrow lines) and uses two instruction cards to represent the divisions. Each instruction cards contains a sequence control-flow block to achieve the plans of moving to and picking flowers. The two sequential control-flow blocks in this design are abutted together to form a complete computer program. Fig. 2(b), another design solution, demonstrates the inclusion of two simple iteration control-flow blocks ('for' loops), each of which repeats the actions of moving to and picking flowers three times. In this design, the two simple iteration control-flow blocks are also abutted together to form a complete computer program. In the design of Fig. 2(c), for each instruction card, a selection control-flow block ('if' condition) is nested in a simple iteration control-flow block, leading to a new pattern that can iteratively select accurate targets to process. Fig. 2(d) shows the design that treats the L-shape region as a whole and uses only one instruction card to represent the solution plan. In the instruction card, a selection control-flow block is nested into a nested iteration control-flow block (nested 'for' loop) to traverse the entire region and select accurate targets. The new pattern repeats a set of instructions ten times. The proposed visual problem-solving environment for programming learning provides various visual programming elements for a novice programmer to produce goals, design algorithms, compose programs, and test the programs. The assistance in the visual programming environment may further help novices explore different design solutions and evaluate the consequence of the design solutions.

3.2. Participants and procedure

The participants in this study were 158 college students majoring in information communication from three classes at the same university in northern Taiwan. The participants were 41% male and 59% female students aged from 18 to 21. They were enrolled in a course introducing the principles and methods of C++ language programming and were instructed by the same teacher who had a computer science major and more than 10 years of teaching experience. During the first 7 weeks of the course, the participants were taught programming concepts including variables, constants, if-then conditions, for loops, and nested for loops.

The research procedure started in the eighth week of the programming course and was included as a part of the course. The procedure included introductory, training, and practice phases, which were conducted in four 60-min periods, with a one

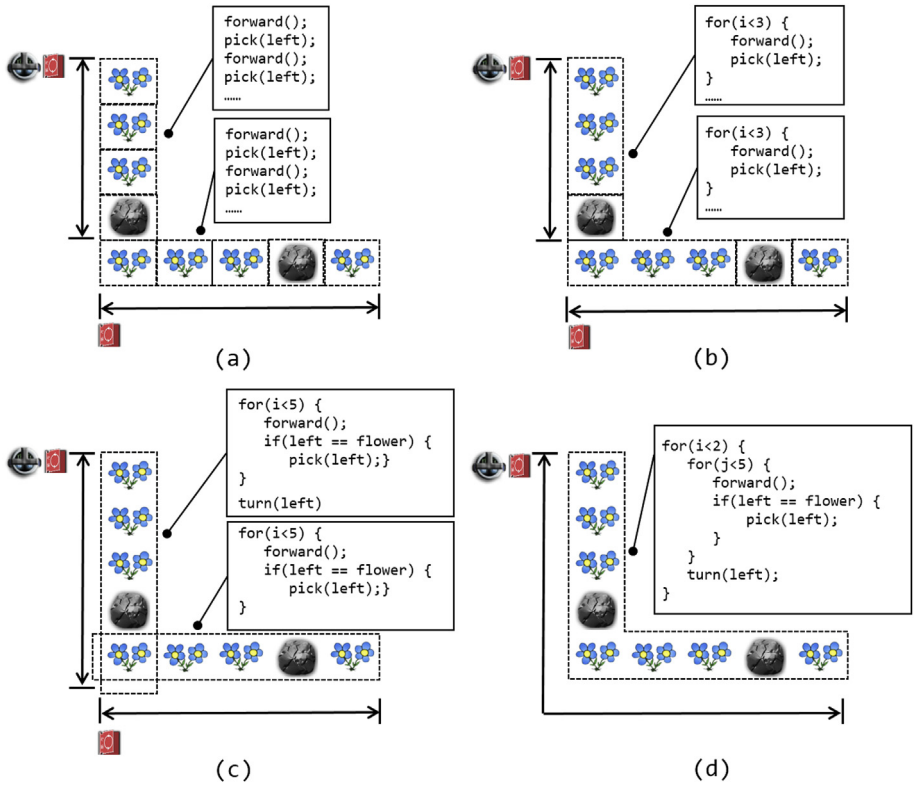

Fig. 2. The different solutions for a computational problem.

week interval between each period. First, in the introductory phase (the first period), the teacher explained the purpose of applying the visual problem-solving environment to the support of the computational problem-solving activity as well as demonstrating the interface of the environment. Second, in the training phase (the second and third periods), the participants were provided with several computational problems and were asked to practice solving computational problems in the visual problem-solving environment. This phase aimed to allow the participants to acquire basic knowledge and skills of applying visualized control-flow and command blocks to the computational problems. Therefore, more diverse programming behavior and design strategies exhibited by the participants could emerge in the visual programming environment. Finally, in the practice phase (the fourth period), a task-driven approach was employed to urge the participants to perform the activities of computational problem solving. The participants enrolled in the programming course needed to complete the given assignment. The assignment was to "Help the robot collect all the flowers in the given five maps with the least possible number of command blocks". The five maps corresponded to five computational problems that required the students to design algorithms, as well as assemble control-flow blocks and command blocks to solve the problems. All participants' activities throughout the procedure were tracked and stored as log data for later analysis. The participants were informed that their performance on the assignment would be considered as part of their grade for the course. In this regard, the course-related assignment and procedure were expected to encourage the students' engagement in the computational problem-solving activity.

3.3. Data collection and analysis

The data collected in this study included log data about the participants' practice, strategies, and performance of computational problem-solving activities. Table 1 reveals the definition of each indicator representing the computational problem-solving activities in the visual problem-solving environment. These indicators could be categorized as three dimensions: computational practice, computational design, and computational performance.

The first five indicators represent the computational practice of adopting control-flow blocks and testing the consequence of generated computer instructions. They denote the behavior of using fundamental control structures (Wirth, 1971) to compose a computer program as a solution to a computational problem and testing the solution for the validation of the computer program. The indicators of *Sequence* and *Selection* correspond to the use of the sequence and selection control-flow blocks, respectively. The indicators of *Simple iteration* and *Nested iteration* correspond to the use of the simple iteration and nested iteration control-flow blocks, respectively, and represent different structures of iteration patterns (Moreno, 2012). The former represents a one-dimensional iteration structure, such as AAA demonstrating the repetition of element A. The latter represents a two-dimensional iteration structure, such as AAADAAAD showing the repetition of a compound element AAAD and the repetition of the element A in the compound pattern AAAD. Higher values of the abovementioned four indicators indicate more computational practice of using the corresponding control-flow blocks. The four indicators of adopting control-flow blocks were calculated by inspecting users' final computer programs. For example, there are two sequence control-flow blocks in the case of Fig. 2(a), two simple iteration and two selection control-flow blocks in case Fig. 2(c), and one nested iteration and one selection control-flow block in case Fig. 2(d). The indicator of *Testing* shows the average frequency of which a participant tested the consequence of executing a computer program immediately after generating or revising the program. It was computed by inspecting the log data related to the frequencies of executing computer programs during the process of the visual programming. The indicator shows the ratio of the number of which a user executes computer programs to test the consequence during problem solving and the number of visualized computer instructions (control-flow and command blocks) generated by a user in his/her final computer programs. A lower value of the indicator *Testing* may show fewer tests on

Table 1
The indicators of programming activities for problem solving in the visual programming environment.

Indicator	Definition
Computational practice	
Sequence	The number of sequence control flow blocks used by participants
Selection	The number of selection control flow blocks used by participants
Simple iteration	The number of simple iteration control flow blocks used by participants
Nested iteration	The number of nested iteration control flow blocks used by participants
Testing	The average frequency of acts testing the consequence of a programmed computer instruction
Computational design	
Problem decomposition	The number of instruction cards used to separate the solutions of problems into different parts of computer instructions
Abutment composition	The number of combinations where two flow control blocks are glued together back to front in the solutions of problems
Nesting composition	The number of combinations where one flow control block was embedded into another one in the solution of problems
Computational performance	
Goal attainment	The number of flowers collected by a participant through executing his/her computer program developed for solving a computational problem
Program size	The average number of command blocks and control-flow blocks used in a participant's computer program to solve a computational problem

computer instructions composed by a participant, which may imply that the participant tested their computer instructions based on chunks of the instructions rather than line by line.

The abovementioned five indicators (i.e., *Sequence*, *Selection*, *Simple iteration*, *Nested iteration* and *Testing*) were employed to represent the participants' activities in the visual programming environment. The participants' frequencies of the five indicators were analyzed by cluster analysis to explore their patterns of computational practice for solving computational problems. The Ward's minimum variance method was first adopted to identify number of clusters, followed by the *k*-mean cluster analysis on the identified cluster number (e.g., Lin & Tsai, 2012). An ANOVA analysis was employed to distinguish different computational practice patterns by comparing the five indicators among different clusters. Moreover, the post hoc tests were also employed to examine the significance of all possible pair-wise comparisons among clusters for the interpretation of the clusters. It should be noted that the ANOVA analysis was conducted for descriptive purposes as it has been employed in many cluster analysis studies (e.g., Hou, 2015; Lin & Tsai, 2012; Pintrich, Anderman, & Klobucar, 1994).

The dimension of computational design consisted of three indicators to reveal the strategies of decomposing problems and the strategies of composing control-flow blocks. The indicators denote the plan activity of decomposing a problem into sub-problems and the design activity of formulating solutions by abutting or nesting visualized control-flow blocks for the sub-problems. As shown in Table 1, the indicator of *Problem decomposition* represents the number of sub-problems decomposed by a student for the following design and implementation of solutions. The instruction cards generated by the student in the proposed environment reveals the division of a solution to a problem and the decomposition of the problem. A higher value of the *Problem decomposition* indicator showed more fine-grained sub-problems produced in the process of decomposing problems. The indicators of *Abutment composition* and *Nesting composition* represent, respectively, the participants' methods of adjoining control flow patterns sequentially or inserting one control flow pattern into another. The abovementioned three indicators were calculated by inspecting log data and users' final computer programs. For instance, in the case of Fig. 2(c), the problem was decomposed into two sub-problems. For the two decomposed sub-problems, two instruction cards were adopted to implement the solutions. In this case, there is one abutment composition (between two simple iteration control-flow blocks) and two nesting compositions (between simple iteration and selection control-flow blocks).

On the other hand, the indicators of *Goal attainment* and *Program size* aimed to reveal the performance of computational problem solving in terms of achieving quantitative or qualitative requirements of computational problems in the given assignment. The *Goal attainment* indicator demonstrated the effectiveness of a design solution in terms of the accurate collection of flowers. A higher value of the *Goal attainment* indicator showed greater effectiveness in terms of achieving the quantitative requirements. For the achievement of the qualitative requirements, the *Program size* indicator was adopted to reveal the quality of the participants' design solutions in terms of the instructions' efficiency. A smaller value of the *Program size* indicator shows higher efficiency of the participants' design solutions. The two computational performance indicators were calculated by inspecting users' final computer programs. For instance, in the case of Fig. 2(d), a novice programmer collected seven flowers (*Goal attainment*) by generating six visualized instruction blocks (*Program size*).

The indicators representing computational design and performance were further analyzed. For descriptive purpose, a series of ANOVA analyses were employed to examine the interrelationships between computational design strategies and performance in the visual programming environment.

4. Results

4.1. Descriptive statistics of the computational problem-solving activities

Table 2 reveals the descriptive statistics of 10 indicators regarding the computational problem-solving activities. With regard to the dimension of computational practice, the results showed that the participants used more simple iteration ($M = 8.56$, $SD = 4.13$) and sequence ($M = 4.84$, $SD = 3.33$) than selection ($M = 2.04$, $SD = 4.83$) and nested iteration ($M = 1.25$,

Table 2
Statistical results of computational practices, computational design strategies, and computational performance.

Indicators	Range	*M*	*SD*
Computational practice			
Sequence	0–20	4.84	3.33
Selection	0–26	2.04	4.83
Simple iteration	0–17	8.56	4.13
Nested iteration	0–11	1.25	1.87
Testing	0.12–1.02	0.46	0.18
Computational design			
Problem decomposition	5–30	14.65	4.63
Abutment composition	4–36	19.54	7.54
Nesting composition	0–26	1.91	3.18
Computational problem-solving performance			
Goal attainment	8–27	23.91	4.90
Program size	5.25–19	11.74	2.72

SD = 1.87) control-flow blocks in solving the computational problems. The results also showed that the participants, on average, tested a single programmed instruction 0.46 times (*SD* = 0.18). This may indicate that most participants tended to test their code by a chunk of instructions rather than by a single instruction. Referring to computational design, in Table 2, the indicator of *Problem decomposition* showed that the participants produced 14.65 (*SD* = 4.63) subparts of solutions. This may indicate that the participants would generally divide one computational problem into two or more sub-problems and formulate corresponding solutions. The results also showed that the participants demonstrated more *Abutment composition* (*M* = 19.54, *SD* = 7.54) than *Nesting composition* (*M* = 1.91, *SD* = 3.18). These results suggest that the participants, in the visual problem-solving environment, were more likely to generate solutions to the sub-problems by adjoining control-flow blocks rather than nesting the control-flow blocks. Regarding their computational performance, the indicators of *Goal attainment* and *Program size* showed that the participants, on average, collected 23.91 (*SD* = 4.9) flowers and used 11.74 (*SD* = 2.72) command blocks to solve a computational problem.

4.2. Exploring students' patterns of computational practice in the visual programming environment

To explore the students' patterns of computational practice in the activity of problem solving, the study first employed the Ward's minimum variance method to identify number of clusters, followed by the *k*-mean cluster analysis on the identified cluster number. Finally, this procedure resulted in a four-cluster solution to the cluster analysis because the solution yielded the clearest distinctions among and provided more meaningful explanations for the different patterns of computational practices (i.e., *Sequent approach*, *Selective approach*, *Repetitious approach* and *Trial approach*). For descriptive purpose, separate ANOVAs and post hoc tests with the cluster group as an independent factor were conducted to determine the difference of the participants' computational practices among different clusters by comparing the indicators. Table 3 shows the numbers of participants, the mean values of the computational practice indicators in each cluster and the comparison of the post hoc tests. The results showed that there were significant differences among the four clusters for the indicators of *Sequence* ($F = 51.29$, $p < 0.001$), *Selection* ($F = 135.66$, $p < 0.001$), *Simple iteration* ($F = 91.68$, $p < 0.001$), *Nested iteration* ($F = 8.16$, $p < 0.001$) and *Testing* ($F = 4.99$, $p < 0.01$). Moreover, the post hoc tests suggested that the four clusters have the potential to interpret the difference in the computational practice patterns of participants' computational problem solving in the visual problem-solving environment. Therefore, the participants could be classified into four major groups as follows:

4.2.1. Sequent approach to computational practice

As shown in Table 3, cluster 1 includes 44 participants accounting for 27.8% of the study sample. When compared with the other clusters, the frequency of the *Sequence* activity exhibited by the participants in cluster 1 was significantly higher than that of any other cluster. Moreover, the number of the indicator *Simple iteration* was also significantly higher than that of cluster 2 and cluster 4. However, with regard to the indicators *Selection* and *Nested iteration*, the numbers shown by cluster 1 were significantly lower than those of cluster 2 and cluster 4. These results reveal that the participants in this group tended to compose solutions with a relatively linear progression of computational practice, such as sequence or simple iteration, but tended to exert less effort to use more advanced practices, such as selection or nested iteration. The participants in this cluster exhibited a sequential approach to computational practices while dealing with computational problems. The participants seemed to employ a relatively fundamental approach that require merely simple control-flow blocks to implement their design solutions. They appeared to be unable or unwilling to apply more advanced control-flow blocks to their design solutions.

4.2.2. Selective approach to computational practice

The second cluster includes nine students accounting for 5.7% of the study sample, which is the smallest group among the four clusters. When compared with the other clusters, the numbers of *Selection* and *Nested iteration* exhibited by the participants in cluster 2 were significantly higher than those of any other cluster. In particular, the frequency of *Testing* was significantly lower than those of the other three clusters. In addition, they demonstrated relatively low values for the indicators of *Sequence* and *Simple iteration*, which to some extent reveals a reverse pattern to that of cluster 1. They also tended to create complex repetitious conditions by using selection and nested iteration tactics rather than by merely using simple ones. They revealed a selective and divergent approach to computational problem solving. The participants appeared to

Table 3
The cluster of students' patterns of computational practice.

	Sequence	Selection	Simple iteration	Nested iteration	Testing
(1) Sequent approach (n = 44) *M*/*SD*	8.55/2.94	0.84/2.18	8.73/2.61	0.52/0.76	0.49/0.15
(2) Selective approach (n = 9) *M*/*SD*	1.89/3.18	18.56/4.95	6.78/3.63	3.0/3.5	0.4/0.15
(3) Repetitious approach (n = 68) *M*/*SD*	3.29/1.55	0.62/1.80	11.6/2.37	1.06/1.36	0.42/0.15
(4) Trial approach (n = 37) *M*/*SD*	4.0/2.69	2.08/3.28	3.22/2.27	2.03/2.52	0.54/0.22
F(ANOVA)	51.29***	135.66***	91.68***	8.16***	4.99**
Post hoc tests (LSD tests)	1 > 2, 1 > 3, 1 > 4 4 > 2	2 > 1, 2 > 3, 2 > 4 4 > 1, 4 > 3	1 > 2, 1 > 4 2 > 4 3 > 1, 3 > 2, 3 > 4	2 > 1, 2 > 3 4 > 1, 4 > 3	1 > 3 4 > 2, 4 > 3

**$p < 0.01$, ***$p < 0.001$.

mainly combine selection and nested iteration control-flow blocks. The selection control-flow blocks assisted the participants to implement different action plans corresponding to different conditions, and the nested iteration control-flow blocks help the participants implement plans that heavily reused a small set of computer instructions. The participants were likely to produce a more flexible and efficient solution.

4.2.3. Repetitious approach to computational practice

The third cluster includes 68 students accounting for 43% of the study sample, which forms the largest group among the four clusters. Among the clusters, cluster 3 had the highest number of *Simple iteration* and a relatively low number of *Selection*. This cluster also had a significantly lower number of *Testing* than cluster 1 and cluster 4. These results reveal that the participants in this group mainly applied a simple repetitious approach to computational practices while performing the computational problem-solving activity. The participants appeared capable in discovering repeated patterns, such as a line of flowers, drawn from the distribution of flowers in the computational problem and applying simple iteration control-flow blocks to implement a repeated set of computer instructions for the repeated patterns. The participants' frequent use of simple iteration control-flow blocks and the lower number of testing their solutions may reveal that they may be familiar with the consequent effects of applying simple iteration control-flow blocks.

4.2.4. Trial approach to computational practice

Finally, the fourth cluster accounts for 23.4% of the study sample ($n = 37$). Akin to cluster 2, the students in this cluster had relatively higher numbers of *Selection* and *Nested iteration* than those in cluster 1 and cluster 3. However, this cluster had a significantly higher number of *Testing* than cluster 2 and cluster 3. The cluster also had a relatively higher frequency of *Sequence* than cluster 2. These results reveal that the participants in this cluster tended to combine various computational practices in the problem-solving activity and frequently employed a trial approach to testing the results of generated instructions. The participants seemed more frequent to try the application of different control-flow blocks to the implementation of their solutions and evaluate the consequence of their implementation. The combination of control-flow blocks and the frequent trials may reveal that the participants may have difficulties in implementing basic control-flow blocks or applying complex combination of control-flow blocks.

4.3. The comparison of computational design strategies among the patterns of computational practice

To examine the relationships between computational practice and computational design strategies, a series of ANOVAs and post hoc tests with cluster group as an independent factor were conducted to compare the indicators of computational design among different clusters (i.e., *Problem decomposition*, *Abutment composition* and *Nesting composition*). The results reveal significant differences in the indicators of computational design among the four clusters. As shown in Table 4, the clusters of *Sequent approach* and *Repetitious approach* revealed significantly higher numbers for the indicators of *Problem decomposition* and *Abutment composition* but a lower number of the indicator *Nesting composition* than the clusters of *Selective approach* and *Trial approach*. These results revealed that the participants in the *Sequent approach* or *Repetitious approach* clusters tended to decompose problems into finer-grained subparts and were more likely to compose solutions for these subparts by abutting control-flow blocks.

On the contrary, the participants labeled as the *Selective approach* or *Trial approach* clusters tended to decompose problems into coarser-grained subparts and were more likely to compose solutions for the subparts by nesting control-flow blocks. Furthermore, as shown in Table 4, the participants in the group of *Selective approach* had higher numbers of all indicators regarding computational design strategies than those in the *Trial approach* group. This result may indicate that although the *Selective approach* and *Trial approach* clusters shared a similar tendency in terms of the decomposition of problems and the composition of control-flow blocks, the former may approach more sub-problems by nesting control-flow blocks than the latter.

4.4. The comparison of computational performance among the patterns of computational practice

To examine the relationships between computational practice and computational performance, a series of ANOVAs and post hoc tests with cluster group as an independent factor were conducted to compare the indicators of computational

Table 4
Comparison of computational design strategies of the different clusters.

	Problem decomposition	Abutment composition	Nesting composition
(1) Sequent approach (n = 44) *M*/*SD*	17.8/3.65	21.43/6.37	0.75/1.20
(2) Selective approach (n = 9) *M*/*SD*	11.67/3.71	16/8.2	9.11/7.18
(3) Repetitious approach (n = 68) *M*/*SD*	15.96/2.67	23.22/5.15	1.4/1.83
(4) Trial approach (n = 37) *M*/*SD*	9.24/3.7	11.37/5.79	2.46/3.06
F(ANOVA)	53.138***	35.605***	27.482**
Post hoc tests (LSD tests)	1 > 2, 1 > 3, 1 > 4 2 > 4 3 > 2, 3 > 4	1 > 2, 1 > 4 2 > 4 3 > 2, 3 > 4	2 > 1, 2 > 3, 2 > 4 4 > 1, 4 > 3

**$p < 0.01$, ***$p < 0.001$.

performance among different clusters (i.e., *Goal attainment* and *Program size*). As shown in Table 5, the *Trial approach* had a significantly lower number of the *Goal attainment* indicator and a higher value of the *Program size* indicator than the *Sequent approach* and the *Repetitious approach*. The participants who adopted the *Trial approach* collected the least flowers and produced relatively inefficient instructions, which implies relatively lower effectiveness and efficiency in performing the given assignments.

With regard to the *Selective approach* and the *Repetitious approach*, the participants in these clusters showed a significantly lower value of the *Program size* indicator than the other two clusters. Additionally, the *Selective approach* cluster had the lowest value of the *Program size* indicator. These results could suggest that both the *Selective approach* and the *Repetitious approach* clusters produced relatively efficient computer programs, but the former cluster did better than the latter. Moreover, the participants in the *Repetitious approach* demonstrated higher numbers of *Goal attainment* than those in the *Trial approach* cluster but had no difference in this indicator with the other two clusters. These results could suggest that, when compared with the *Trial approach* cluster, the *Repetitious approach* cluster participants could meet relatively more quantitative requirements of problems with relatively efficient computer instructions.

Finally, the participants in the *Sequent approach* cluster had a significantly higher number of *Goal attainment* than those in the *Trial approach* cluster. They also had a significantly higher value of the *Program size* than those in the *Selective approach* and *Repetitious approach* clusters. These results demonstrate that the participants in this cluster could also meet relatively more quantitative requirements of the problems. However, they tended to produce relatively inefficient computer instructions.

5. Discussion and conclusion

This study proposed a visual problem-solving environment aiming to assist students' programming learning by analyzing, designing, implementing, and evaluating solutions to computational problems. Because many studies have suggested the potential of visual programming environments for programming learning (Cooper et al., 2000; Lye & Koh, 2014; Maloney et al., 2010), the visual problem-solving environment proposed in this study further allowed the students to solve computational problems by iteratively formulating diverse programming strategies in a visualized and constructive way. Thus, the purpose of programming learning changes its emphasis to the understanding of programming constructs and the application of programming strategies rather than merely focusing on the features or syntax of programming languages (Muller & Haberman, 2008; de Raadt et al., 2009). For this reason, this study examined the effectiveness of the proposed environment by exploring the interrelations among students' computational practice, computational design, and computational problem-solving performance in the computational problem-solving activities.

This pilot study initially proposed ten indicators categorized as computational practice, computational design, and computational performance to display the features of computational problem-solving activities. For example, the participants in this study exhibited the highest numbers of simple iterations among the use of four control-flow blocks. They also tested the results of program execution according to a chunk of instructions. This may imply that the participants could easily identify the situation of adopting simple iteration control-flow blocks in the visual problem-solving environment. This tendency is in line with the finding that novice programmers are relatively familiar with applying loop concepts in a visual programming environment (Maloney, Peppler, Kafai, Resnick, & Rusk, 2008). The visual programming elements provided by the visual problem-solving environment may further support the decomposition of computational problems, the assembly of control flow structures, and the testing of chunks of novices' self-generated instructions (Maloney et al., 2010).

Students' computational practice patterns and computational design are crucial to their quality of learning programming in educational settings (Brennan & Resnick, 2012; Lye & Koh, 2014). By analyzing the students' computational problem-solving activity exhibited while interacting with the environment, this pilot study initially identified four different patterns of computational practice (i.e., *Sequent approach*, *Selective approach*, *Repetitious approach* and *Trial approach*). For example, the participants labeled as adopting the *Selective approach* tended to be more capable of adopting selection and nested iteration control-flow blocks than other clusters for resolving computational problems in the visual problem-solving environment. Concerning computational design strategies, they tended to decompose problems into relatively coarser-grained subparts and represent solutions by nesting selection control-flow blocks into simple iteration or nested iteration

Table 5
Comparison of the computational performance of the different clusters.

	Goal attainment	Program size
(1) Sequent approach (n = 44) *M*/*SD*	24.18/5.34	12.45/2.0
(2) Selective approach (n = 9) *M*/*SD*	23.33/5.57	8.05/1.82
(3) Repetitious approach (n = 68) *M*/*SD*	25.15/3.77	11.36/1.43
(4) Trial approach (n = 37) *M*/*SD*	21.46/5.31	12.49/4.26
F(ANOVA)	4.386**	9.139***
Post hoc tests (LSD tests)	1 > 4	1 > 2, 1 > 3
	3 > 4	3 > 2
		4 > 2, 4 > 3

**$p < 0.01$, ***$p < 0.001$.

control-flow blocks. This type of nesting, which achieved a meaningful programming plan that enabled iterative, simultaneously selective processing of every target item (e.g., flowers or stones), is believed to improve the flexibility and efficiency of the designed solutions (Johnson, 1986). The outcomes of problem solving further evidenced the superiority of this kind of composition. That is, the participants who adopted the *Selective approach* showed the most efficient design solutions among the four clusters. The proposed visual programming environment may help the participants of *Selective approach* adopt relatively advanced design strategies that could lead to efficient and flexible design solutions to the computational problems. The cluster played a particular role in supporting programming learning because the participants' features of solving programming problems appear similar to those of the 'effective novice' suggested by Robins et al. (2003, p. 165), who can apply effective programming strategies without excessive effort or assistance. The pattern of the *Selective approach* provides valuable insights into the behavioral and strategic patterns that may be employed by effective novices.

The participants categorized as adopting the *Trial approach* also attempted to use selection and nested iteration control-flow blocks and compose their design solutions by nesting the control-flow blocks. Nevertheless, they finally produced relatively ineffective and inefficient computer programs. Although the participants of *Trial approach* attempted to try relatively advanced design strategies, they appeared to have most difficulties in producing good computational performance in the visual programming environment. This may suggest that in the proposed visual programming environment the adoption of selection control-flow blocks and the employment of nesting composition strategies may be insufficient for the good performance of solving computational problems. Research shows that the strategies of pattern composition, particularly for the nesting method, are the main factors in students' difficulties solving programming problems (Soloway, 1986; Spohrer & Soloway, 1986). The participants tended to adopt the nesting method and show relatively frequent testing of solutions. In this regard, some participants may have difficulties applying their nested blocks to solving computational problems. Further examination of errors or revisions made by novice programmers during the process of programming can be helpful in revealing the kinds of the difficulties or in finding other possibilities that lead to frequent testing of solutions and poor computational performance. To support the development of effective design strategies, it is suggested that particular scaffolding is needed to assist the participants in mastering this nesting composition method. For instance, integrating explicit educational instructions on the composition of programming plans (Muller & Haberman, 2008; de Raadt et al., 2009) with visualized program tracking mechanisms (e.g., Rowe & Thorburn, 2000) may help the participants deeply understand how nested control-flow blocks work and what the subsequent effects are.

With regard to the groups of *Repetitious approach* and *Sequent approach*, the participants usually adopted sequence and simple iteration control-flow blocks. Unlike the participants in the other two groups (i.e., *Selective approach* and *Trial approach*), they tended to design solutions by decomposing problems into more fine-grained subparts and then compose solutions by adjoining sequence and simple iteration control-flow blocks. The proposed visual programming environment may assist the participants of *Repetitious approach* or *Sequent approach* to adopt relatively simple design strategies that may partially achieve good computational performance. The simple design strategies appeared to help the participants meet quantitative requirement of computational problems but may also produce relatively inefficient design solutions. The participants' design strategies of decomposing problems and composing control-flow blocks may be affected by their knowledge of control-flow structures (Robins et al., 2003). They may possess insufficient knowledge of when and how to decomposing problems for the application of more complex control-flow blocks, such as selection or nested iteration control-flow blocks. The assistance that helps the participants acquire knowledge of applying nesting method and complex control-flow structures or reminds them to consider the efficiency requirement of computational problems can be beneficial to the learning of constructing efficient computer programs. For this reason, particular scaffolding or problem setting could be employed to improve the quality of decomposition during the design phase of problem solving, for instance, imposing efficiency constraints on the size of self-generated computer programs, such as explicitly asking a novice to solve a computational problem within a maximum number of computer instructions (e.g., Code.org, 2015). Another example is to provide students with hints about using specific control-flow blocks during problem solving, such as a scaffolding that automatically assesses students' use of specific control-flow blocks and provides them with suggestion of more appropriate control-flow blocks immediately after the participants test their design solutions in the visual programming environment. With these supports, it is believed that the students can learn the nesting method of composing different control-flow blocks and make progress toward becoming effective novices. In addition, it is noticeable that the participants who adopted the *Sequent approach* showed less simple iteration computational practice and more testing of solutions when compared with those participants who took the *Repetitious approach*. This difference may reveal that the participants in the *Sequent approach* group did not have as much mastery over the simple iteration control-flow blocks as the *Repetitious approach* participants. Therefore, before introducing the nesting of control-flow blocks to the *Sequent approach* participants, scaffolding that helps them transform the use of sequence control-flow blocks into the use of iteration ones could be beneficial to their mastery of iteration control-flow blocks.

A visual problem-solving environment for programming learning provides students with technological supports in learning programming through solving computational problems. The students learned to adopt effective computational practice and design strategies by interacting with the environment. The proposed visual problem-solving environment assisted students in modeling, simulation, and problem solving, which is believed to be beneficial to the development of computational thinking (Lye & Koh, 2014). The visual problem-solving environment provided students with rich and visual programming elements for meaningful interactions between the students and the environment. The meaningful interaction revealed different practical patterns, which reflects diverse engagement that leads to different computational design

strategies and performance of computational problem solving. For example, the students employing a more selective approach to computational practice tended to use the more advanced design strategy of nesting different control flow structures, which may produce relatively effective and efficient computer programs. In this sense, students who are motivated to engage in meaningful interaction with the visual problem-solving environment may develop more effective strategies of computational design and then have better performance of computational problem solving. Therefore, the scaffolding for meaningful interactions may cultivate students' programming skills and design strategies.

The results of this study show that visual problem solving through programming constitutes an effective approach to assisting novice programmers to learn programming and computational design strategies. The cluster analysis adopted in this study identified different patterns of computational practices and design strategies in solving computational problems. However, the classes and fields of computational problems conducted in this study are specific, which may limit the emergence of behavioral patterns and design strategies. The generalization of the computational practice patterns and the findings presented in this study are limited. Future studies need to explore novices' programming behavior of solving computational problems of different classes. Additionally, this study was still a small-scale investigation. Further work needs to be undertaken with a larger sample to provide additional evidence. Specifically, because there has been renewed interest in introducing programming to younger students in recent years (Grover & Pea, 2013), it would be worthwhile to explore how the visual problem-solving environment could influence younger students' learning of programming and problem-solving skills. In addition, according to the results of the cluster analysis, the participants demonstrated different practical patterns and adopted diverse design strategies to solve problems. Given the difference in the visual problem-solving activity of the participants, further research on the design of visual programming environments should consider such individual differences. Furthermore, to support novice programmers' visual problem-solving activities, a visual problem-solving environment for programming learning integrated with specific scaffolds should be designed to facilitate and guide visual programming processes in consideration of all of the principal computational practice and design strategies involved in programming. Future research should analyze different aspects of novice programmers' processes of visual programming. Activities engaged by novice programmers during the process of programming, such as inspecting errors, revising plans, or tracing codes, could provide more insight into the behavioral patterns and design strategies exhibited by novice programmers. Further investigation is also necessary to explore the integration of the visual problem-solving environment into a programming instruction and to explore the effects on programming behavior and performance. It would also be worthwhile to integrate game elements into the visual environment and to explore the influence of the game elements on the learners' experience and motivation to solve computational problems.

Acknowledgements

This research was partially funded by Ministry of Science and Technology, Taiwan, under grant number MOST 103-2511-S-155-001-MY2

References

Barg, M., Fekete, A., Greening, T., Hollands, O., Kay, J., Kingston, J. H., et al. (2000). Problem-based learning for foundation computer science courses. *Computer Science Education, 10*(2), 10–128.

Bers, M. U., Flannery, L., Kazakoff, E. R., & Sullivan, A. (2014). Computational thinking and tinkering: exploration of an early childhood robotics curriculum. *Computer & Education, 72*, 145–157.

Böhm, C., & Jacopini, G. (1966). Flow diagrams, turing machines, and languages with only tow formation rules. *Communication of the ACM, 9*(5), 366–371.

Brennan, K., & Resnick, M. (2012). New frameworks for studying and assessing the development of computational thinking. In *Paper presented at the annual American educational research association meeting, Vancouver, BC, Canada*.

Brookshear, J. G. (2003). *Computer science: An overview*. Boston: Pearson Education, Inc.

Brusilovsky, P., Calabrese, E., Hvorecky, J., Kouchnirenko, A., & Miller, P. (1997). Mini-languages: a way to learn programming principles. *Education and Information Technologies, 2*, 65–83.

Code.org. (2015). *Hour of Code website*. Retrieved from http://code.org/learn.

Cooper, S., Dann, W., & Pausch, R. (2000). Alice: 3D tool for introductory programming concepts. *Journal of Computing Sciences in College, 15*(5), 107–116.

Cross, J. H., II, Hendrix, T. D., & Barowski, L. A. (2002). Using the debugger as an integral part of teaching CS1. In *Paper presented at the 32nd annual frontiers in education, Boston, Massachusetts*.

Davies, S. P. (1993). Models and theories of programming strategy. *International Journal of Man-Machine Studies, 39*, 237–267.

Deek, F. P. (1999). The software process: a parallel approach through problem solving and program development. *Computer Science Education, 9*(1), 43–70.

Deitel, P., & Deitel, H. (2011). *C++ how to program*. New Jersey: Pearson Education, Inc.

Edmonds, J. (2008). *How to think about algorithms*. New York: Cambridge University Press.

Einhorn, S. (2011). *Microworlds, computational thinking, and 21st century learning*. Logo Computer System Inc, White Paper. Retrieved from http://www.microworlds.com/.

Fee, S. B., & Holland-Minkley, A. M. (2010). Teaching computer science through problems, not solutions. *Computer Science Education, 20*(2), 129–144.

Flannery, L. P., Kazakoff, E. R., Bonta, P., Silverman, B., Bers, M. U., & Resnick, M. (2013). Designing ScratchJr: support for early childhood learning through computer programming. In *Paper presented at the 12th international conference on interaction design and children (IDC '13), New York, NY, USA*.

Futschek, G., & Moschitz, J. (2011). Learning algorithmic thinking with tangible objects eases transition to computer programming. *Informatics in Schools, Contributing to 21st Century Education, 7013*, 155–164.

Gomes, A., & José, A. (2007). An environment to improve programming education. In B. Rachev, A. Smrikarov, & D. Dimov (Eds.), *Proceedings of the 2007 international conference on computer systems and technologies. Rousse, Bulgaria*.

Gourlay, J. S. (1983). A mathematical framework for the investigation of testing. *IEEE Transactions of Software Engineering, SE-9*(6), 686–709.

Gouws, L., Bradshaw, K., & Wentworth, P. (2013). Computational thinking in educational activities: an evaluation of the educational game light-bot. In *Paper presented at the 18th ACM conference on innovation and technology in computer science education, Canterbury, England*.

Green, T. R. G., & Petre, M. (1996). Usability analysis of visual programming environment: a 'cognitive dimensions' framework. *Journal of Visual Language and Computing, 7*, 131–174.
Grover, S., & Pea, R. (2013). Computational thinking in K-12: a review of the state of the field. *Educational Researcher, 42*(1), 38–43.
Hazzan, O., Lapidot, T., & Ragnis, N. (2011). *Guide to teaching computer science: An activity-based approach* (2nd ed.). London: Springer.
Holvikivi, J. (2010). Conditions for successful learning of programming skills. In N. Reynolds, & M. Turcsányi-Szabó (Eds.), *Vol. 324. Key competencies in the knowledge society* (pp. 155–164).
Hou, H.-T. (2015). Integrating cluster and sequential analysis to explore learners' flow and behavioral patterns in a simulation game with situated-learning context for science courses: a video-based process exploration. *Computer in Human Behavior, 48*, 423–435.
Ismal, M. N., Ngah, N. A., & Umar, I. N. (2010). Instructional strategy in the teaching of computer programming: a need assessment analyses. *The Turkish Online Journal of Educational Technology, 9*(2), 125–131.
Johnson, W. L. (1986). *Intention-based diagnosis of novice programming errors*. London: Morgan Kaufmann Publishers, Inc.
Kahn, K. (1996). ToonTalk an animated programming environment for children. *Journal of Visual Language and Computing, 7*, 197–217.
Kelleher, C., & Pausch, R. (2005). Lowering the barriers to programming: a taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys, 37*(2), 83–137.
Kessler, C. M., & Anderson, J. R. (1986). Learning flow of control: recursive and iterative procedure. *Human-Computer Interaction, 2*, 136–166.
Kiesmüller, U. (2009). Diagnosing learners' problem-solving strategies using learning environments with algorithmic problems in secondary education. *ACM Transactions on Computing Education, 9*(3), 17:11–26.
Kölling, M. (2010). The greenfoot programming environment. *ACM Transactions on Computing Education, 10*(4), 14:1–21.
Kurland, D. M., Pea, R. D., & Clement, C. (1986). A study of the development of programming ability and thinking skills in high school students. *Journal of Educational Computing Research, 2*(4), 429–459.
Linn, M. C. (1985). The cognitive consequences of programming instruction in classrooms. *Educational Researcher, 14*(5), 14–29.
Linn, M. C., & Clancy, M. J. (1992). The case for case studies of programming problems. *Communication of the ACM, 35*(3), 121–132.
Lin, C.-C., & Tsai, C.-C. (2012). Participatory learning through behavioral and cognitive engagements in an online collective information searching activity. *International Journal of Computer-Supported Collaborative Learning, 7*(4), 543–566.
Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., et al. (2004). A multi-national study of reading and tracing skills in novice programmers. *SIGCSE Bulletin, 36*(4), 119–150.
Liu, C.-C., Cheng, Y.-B., & Huang, C.-W. (2011). The effect of simulation games on the learning of computational problem solving. *Computers & Education, 57*, 1907–1918.
Li, F. W. B., & Watson, C. (2011). Game-based concept visualization for learning programming. In *Paper presented at the third international ACM workshop on multimedia technologies for distance learning, Scottsdale, AZ, USA*.
Lye, S. Y., & Koh, J. H. L. (2014). Review on teaching and learning of computational thinking through programming: what is next for K-12? *Computers in Human Behavior, 41*, 51–61.
Maloney, J., Peppler, K., Kafai, Y., Resnick, M., & Rusk, N. (2008). Programming by choice: urban youth learning programming with scratch. In *Paper presented at the 39th SIGCSE technical symposium on computer science education Portland, OR, USA*.
Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The scratch programming language and environment. *ACM Transactions on Computing Education, 10*(4), 16:1–15.
McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L., et al. (2008). Debugging: a review of the literature from an educational perspective. *Computer Science Education, 18*(2), 67–92.
McCracken, M., Kolikant, Y. B.-D., Almstrum, V., Laxer, C., Diaz, D., Thomas, L., et al. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *Inroads, 33*(4), 125–140.
Moreno, J. (2012). Digital competition game to improve programming skills. *Educational Technology & Society, 15*(3), 288–297.
Muller, O., & Haberman, B. (2008). Supporting abstraction processes in problem solving through patter-oriented instruction. *Computer Science Education, 18*(3), 187–212.
Navarro-Prieto, R., & Canas, J. J. (2001). Are visual programming languages better? The role of imagery in program comprehension. *International Journal of Human-Computer Studies, 54*, 799–829.
Paliokas, I., Arapidis, C., & Mpimpitsos, M. (2011). PlayLOGO 3D: a 3D interactive video game for early programming education. In *Paper presented at the third international conference on games and virtual worlds for serious applications, Athens, Greece*.
Palumbo, D. B. (1990). Programming language/problem-solving research: a review of relevant issues. *Review of Educational Research, 60*(1), 65–89.
Pattis, R. E., Roberts, J., & Stehlik, M. (1995). *Karel the robot: A gentle introduction to the art of programming*. New York: Wiley.
Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., et al. (2007). A survey of literature on the teaching of introductory programming. *ACM SIGCSE Bulletin, 39*(4), 204–223.
Pennington, N., & Grabowski, B. (1990). The tasks of programming. In J.-M. Hoc, T. R. G. Green, R. Samurcay, & D. J. Gilmore (Eds.), *Psychology of programming* (pp. 45–62). Lodon: Harcourt Brace Jovanovich.
Pintrich, P. R., Anderman, E. M., & Klobucar, C. (1994). Intraindividual difference in motivation and cognition in students with and without learning disabilities. *Journal of Learning Disabilities, 27*(6), 360–370.
de Raadt, M. (2007). A review of Australasian investigations into problem solving and the novice programmer. *Computer Science Education, 17*(3), 201–213.
de Raadt, M., Watson, R., & Toleman, M. (2009). Teaching and assessing programming strategies explicitly. In *Paper presented at the 11th Australasian computing education conference (ACE 2009), Wellington, New Zealand*.
Ring, B. A., Giordan, J., & Ransbottom, J. S. (2008). Problem solving through programming: motivating the non-programmer. *Journal of Computing Sciences in College, 23*(3), 61–67.
Rist, R. S. (1991). Knowledge creation and retrieval in program design. *Human-Computer Interaction, 6*, 1–46.
Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: a review and discussion. *Computer Science Education, 13*(2), 137–172.
Rogalski, J., & Samurcay, R. (1990). Acquisition of programming knowledge and skills. In J.-M. Hoc, T. R. G. Green, R. Samurcay, & D. J. Gilmore (Eds.), *Psychology of programming* (pp. 157–174). Lodon: Academic Press.
Rowe, G., & Thorburn, G. (2000). VINCE–An on-line tutorial tool for teaching introductory programming. *British Journal of Educational Technology, 31*(4), 359–369.
Sanders, D., & Dorn, B. (2003). Classroom experience with JEROO. *Journal of Computing Sciences in College, 18*(4), 308–316.
Seidam, R. H. (1981). The effects of learning a computer programming language on the logical reasoning of school children. In *Paper presented at the annual meeting of the American education research association, Los Angeles, CA*.
Soloway, E. (1986). Learning to program=learning to construct mechanisms and explanations. *Communication of the ACM, 29*(9), 850–858.
Spohrer, J. C., & Soloway, E. (1986). Novice mistakes: are the folk wisdoms correct? *Communication of the ACM, 29*(7), 624–632.
Suthers, D. D. (1994). Strategies for sequencing as a planning task. In *Paper presented at the 7th international generation workshop, Kennebunkport, Maine*.
Winslow, L. E. (1996). Programming pedagogy–A psychological view. *ACM SIGCSE Bulletin, 28*(3), 17–22.
Wirth, N. (1971). The programming language Pascal. *Acta Informatica, 1*(1), 35–62.
Xinogalos, S. (2012). An evaluation of knowledge transfer from microworld programming to conventional programming. *Journal of Educational Computing Research, 47*(3), 251–277.
Yen, C.-Z., Wu, P.-H., & Lin, C.-F. (2012). Analysis of expert's and novice's thinking process. *Engaging Learners through Emerging Technologies, Communication in Computer and Information Science, 302*, 122–134.