Received: 20 July 2018 | Revised: 1 December 2018 | Accepted: 20 January 2019
DOI: 10.1111/jcal.12348

ARTICLE

WILEY Journal of Computer Assisted Learning

Analysing computational thinking in collaborative programming: A quantitative ethnography approach

Bian Wu[1] | Yiling Hu[1] | A.R. Ruis[2] | Minhong Wang[1,3]

[1] Department of Educational Information Technology, East China Normal University, Shanghai, China
[2] Wisconsin Center for Education Research, The University of Wisconsin-Madison, Madison, Wisconsin, USA
[3] KM&EL Lab, Faculty of Education, The University of Hong Kong, Hong Kong, China

Correspondence
Bian Wu, East China Normal University, No. 3663, North Zhongshan Rd., Shanghai, 200062, China.
Email: bwu@deit.ecnu.edu.cn

Funding information
Humanity and Social Science Youth foundation from the Ministry of Education of China, Grant/Award Number: 16YJC880085; Peak Discipline Construction Project of Education from East China Normal University; Eastern Scholar Chair Professorship Fund from Shanghai Municipal Education Commission of China, Grant/Award Number: JZ2017005; Division of Research on Learning in Formal and Informal Settings, Grant/Award Numbers: 1661036 and 1713110; Wisconsin Alumni Research Foundation; Office of the Vice Chancellor for Research and Graduate Education at the University of Wisconsin-Madison

Abstract
Computational thinking (CT), the ability to devise computational solutions for real-life problems, has received growing attention from both educators and researchers. To better improve university students' CT competence, collaborative programming is regarded as an effective learning approach. However, how novice programmers develop CT competence through collaborative problem solving remains unclear. This study adopted an innovative approach, quantitative ethnography, to analyze the collaborative programming activities of a high-performing and a low-performing team. Both the discourse analysis and epistemic network models revealed that across concepts, practices, and identity, the high-performing team exhibited CT that was systematic, whereas the CT of the low-performing team was characterized by tinkering or guess-and-check approaches. However, the low-performing group's CT development trajectory ultimately converged towards the high-performing group's. This study thus improves understanding of how novices learn CT, and it illustrates a useful method for modeling CT based in authentic problem-solving contexts.

1 | INTRODUCTION

Computational thinking (CT)—the approach to solving authentic problems like a computer scientist or software engineer—is deemed a critical competence for 21st-century knowledge workers. CT competence is often developed through computer programming education in both K-12 and higher education contexts (Czerkawski & Lyman, 2015; Lye & Koh, 2014), and collaborative work has long been recognized as an important mechanism for developing CT (Lindsjørn, Sjøberg, Dingsøyr, Bergersen, & Dybå, 2016). Especially for novice programmers, collaborative programming can offer opportunities to develop collective understanding of computational problems, plan alternative computing solutions, receive peer teaching, build collaborative knowledge, and engage in authentic programming practices such as planning, coding, monitoring, and testing (Kafai & Burke, 2013; Teague & Roe, 2008).

Research on collaborative programming as a pedagogical approach is extensive (see, e.g., Beck & Chizhik, 2013; Jehng, 1997; Shadiev et al., 2014; Williams, Wiebe, Yang, Ferzli, & Miller, 2002). However, most studies have examined only summative outcomes or learners' perceptions, motivations, and engagement, with little focus on the psychosocial process of collaborative programming and the development of CT competence (Emurian, Holden, & Abarbanel, 2008; Maguire, Maguire, Hyland, & Marshall, 2014; Serrano-Cámara, Paredes-Velasco, Alcover, & Velazquez-Iturbide, 2014). There is thus a lack of knowledge regarding the mechanisms through which novice programmers develop CT competence in collaborative programming contexts.

This study moves from a Piagetian perspective, or a focus solely on individuals' programming ability (Lister, 2016; Teague & Lister, 2014), to a Vygotskian perspective, in which CT competence is understood as socially developed and characterized by more than merely computing knowledge and skills. In doing so, our goal was to identify and model CT competence and CT development trajectories in a collaborative programming context. To do so, this study adopted a novel approach, quantitative ethnography, to identify and model CT competence. By documenting the ways in which novice programmers develop CT competence, the results of this study could inform the development of effective formative assessments, which would enable computer educators to better facilitate novice programmers' CT development.

2 | LITERATURE REVIEW

2.1 | CT and programming

The term computational thinking was first proposed by Wing (2006) to represent the widely applicable attitudes and skills needed not only by computer scientists and STEM professionals but also by everyone. However, interpretation of what CT entails has been heavily debated. For example, Chao (2016) regards CT as a problem-solving process involving *computational design* (i.e., understanding computational problems and designing computational solutions), *computational practice* (i.e., solving the problems), and *computational performance* (i.e., testing the solutions). This definition includes only the epistemic dimensions of computational problem solving. Brennan and Resnick (2012), in contrast, argue that CT includes *computational concepts* (i.e., core concepts of CT), computational practices (i.e., programming practices as one uses CT concepts), and *computational perspectives* (i.e., the values, attitudes, and perception of performing CT practices). Although no consensus has been reached on exactly what CT consists of, there is general agreement that CT includes multiple components, not all of which are cognitive. Therefore, CT competence should be conceptualized as a multilayer structure, which includes hierarchical and interrelated components.

Because of the publication of Wing's essay, there has been considerable interest in incorporating CT skills into both K-12 and higher education (Czerkawski & Lyman, 2015), and computer programming education is the most commonly applied approach (Lye & Koh, 2014). Previous studies on programming education that focused on novice programmers identified critical competencies in programming, including planning, program understanding, coding, and debugging. For example, planning, such as writing programme comments in advance or generating top-down modularization, is an early step to successful programming and a skill that novice programmers often lack (Wellons & Johnson, 2011). The ability to provide a summary of a piece of code rather than a line-by-line description—that is, a top-down approach to programming—suggests a higher level of competence (Lister, Simon, Thompson, Whalley, & Prasad, 2006); in contrast, tinkering, or guess-and-check processes that reflect a bottom-up approach, can support iterative exploration of key elements in programming (Berland, Martin, Benton, Petrick Smith, & Davis, 2013). However, due to the opportunistic and incremental nature of novice programming (Robins, Rountree, & Rountree, 2003), we still lack comprehensive knowledge of CT competence development at this programming learning stage.

Moreover, measurement of CT competence is most commonly based on summative assessment, such as questionnaires (Korkmaz, Çakir, & Özden, 2017) or test scores (Román-González, Pérez-González, & Jiménez-Fernández, 2017; Zhong, Wang, Chen, & Li, 2016), with little emphasis on formative assessment of programming processes to foster CT competence development. Besides, such assessments typically focus on individual outcomes. For example, adopting Brennan and Resnick's (2012) three-dimensional framework, Kafai et al. (2014) assessed individual students' understanding of core CT concepts. They found that remixing (i.e., including segments of existing code into their own design projects) played the most prominent role in CT competence development. However, their findings were constrained to individual learning, and the study assessed performance in the three CT dimensions separately. They suggested that future study should investigate the relations among computing concepts, computing practice, and computing perspective. Following this direction, Chao (2016) explored the interrelations among computational practice, computational design, and computational problem-solving performance to identify distinctive patterns of programming learning using cluster analysis. However, this analysis did not construe CT competence as developed both cognitively and socially and thus did not account for the social dimensions of learning.

2.2 | Collaborative programming

Collaborative programming is an instructional method in which a group of students work together to accomplish programming tasks. The instructional approach stems from cognitive and social constructive theories that view learning in a group as a process of actively and collaboratively constructing new knowledge based on prior knowledge and social interaction (Kalaian & Kasim, 2014). The approach also reflects the notion that cognition can be distributed across a group of people, tools, and artifacts (Berland & Lee, 2011; Mangalaraj, Nerur, Mahapatra, & Price, 2014).

Collaborative programming has long been regarded as an effective way to support programming performance (Johnson & Johnson, 1999). Denner, Werner, Campe, and Ortiz (2014) argue that working collaboratively is especially useful in terms of building CT competence and computer programming knowledge for novice learners. Collaborative programming can also help students develop a higher level of confidence in their own problem-solving abilities (Beck & Chizhik, 2013). Other benefits of collaborative programming include more efficient work, fewer defects in programming solutions, greater engagement, and increased programming knowledge and skills, to name but a few (Williams & Kessler, 2002). However, we still lack knowledge about how group dynamics help coordinate different roles in collaborative problem solving, establish shared understanding through conflict negotiation, address the challenges of authentic programming tasks, and foster the development of CT competence (Lister, 2016; Teague & Lister, 2014).

In terms of analysing collaborative programming processes, two challenges need further investigation. First, analysis of recorded logs of programming can identify no more than the basic knowledge and skills of CT (Wang & Hwang, 2012). The presence of a code element in a programming artifact does not necessarily indicate a deep understanding of programming (Brennan & Resnick, 2012). To better examine deep learning, including computational practices and perspectives, student could be asked to verbalize their thought processes while programming (Ericsson & Simon, 1998), and their on-screen programming behaviour could be captured and analysed (Lye & Koh, 2014). Second, different dimensions, such as cognitive, metacognitive, and social or emotional factors, are often analysed separately without viewing them as interrelated and co-developed competences (Khosa & Volet, 2014; Kwon, Liu, & Johnson, 2014). To investigate the development of CT competence holistically, examining the connections among different components of CT competence during collaborative programming is needed. Shaffer (2012) argues that professional knowledge, skills, practices, and perspectives are interconnected through an *epistemic frame*, a particular way of framing, investigating, and solving problems in some domain. Thus, one's expertise is characterized in part by how the particular knowledge, skills, values, and other elements of the domain are integrated, which manifests in the actions and interactions of individuals in authentic problem-solving contexts. Learning to adopt the epistemic frame of a particular domain—in this case, computer programming—ultimately fosters identity development, as students shift from programming students to student programmers. Importantly, this process of identity development is facilitated by learning environments in which students can take consequential action and reflect on those actions with peers and more experienced others, in authentic contexts. Making practice meaningful and showing the applicability of domain competence can shape one's long-term interests and personal identity in the target domain, which, in turn, can motivate oneself to improve knowledge and skills through more deliberate practice (Foster & Shah, 2016).

2.3 | Quantitative ethnography

To analyse the processes of learning and expertise development in collaborative programming, researchers need to study the interactions of collaborating programmers (Lin & Liu, 2012). Leung (2002) suggests that ethnographic accounts would contribute significantly to our understanding of social learning processes such as collaborative programming (see also Denner & Werner, 2007). However, traditional ethnographic studies are based on laborious qualitative analyses and thus cannot easily be conducted at scale. To address the difficulty of analysing large amounts of ethnographic data to identify meaningful patterns for both pedagogical and assessment purposes, Shaffer (2017) developed a method known as *quantitative ethnography*.

Quantitative ethnography, which is described in detail in Shaffer (2017), is a method that combines statistical inference with the interpretive power of qualitative, grounded analysis. Importantly, quantitative ethnography addresses a critical problem in assessing complex and collaborative thinking. When students work in groups to solve problems, researchers can assess either the work of the group or the work of a specific member of that group. In collaborative problem solving, however, it is impossible to understand the work of an individual without accounting for the contributions of the other members of the group.

Epistemic network analysis (ENA; Shaffer, Collier, & Ruis, 2016), a quantitative ethnographic technique for measuring and visualizing complex, collaborative thinking, addresses this challenge by modeling learning in terms of the connections individual students make—in their speech or actions—among key skills, knowledge, values, and decisions. That is, it measures the development of an epistemic frame. Importantly, the connections in an ENA model can be between things that the individual does independently, or between things that an individual does and things that other members of the group say and do. ENA models learning as a network of connections, where each individual's network includes the relevant connections that he or she made in the context of the activities of the group. This makes it possible to assess collaborative problem solving without specifying explicitly a correct sequence of problem-solving steps.

In this study, we use ENA to model the complex, collaborative thinking of students engaged in a collaborative programming project in order to understand how novices develop CT competence in collaborative contexts.

2.4 | Purpose and research questions

To understand how novice programmers learn to think like programmers when working collaboratively to develop software, this study investigated novice programmers' collaborative programming processes and strategies for computational problem-solving enacted in programming projects. Specifically, it explored the development of individual CT abilities in relation to different kinds of collaborative programming behaviour across the different development stages. To this end, this study aimed to answer the following research questions:

> RQ1: What CT patterns do novice programmers exhibit when they collaborate on the development of a software application?
>
> RQ2: Do novice programmers follow different trajectories of CT competence development based on the collaborative programming activities of their group?

3 | METHOD

3.1 | Participants

Forty-seven year-one students in an educational technology major from a Chinese university attended CS1, introduction to C++

programming. They were all novice programmers with little or no programming experience. The students were asked to form collaborative programming groups with three to four students per group. This exploratory study selected two groups for a case study. Each group's work was recorded (screen-capture), transcribed, coded, and analysed.

3.2 | Research design and procedure

In this course, the instructor designed four collaborative programming projects as part of the course assignments. The projects were located in the curriculum after the required programming knowledge, such as branching, looping, and creating methods, had been taught via lecturing and in-class programming practice. In addition, the project description was accompanied by a brief summary of relevant programming knowledge. The projects were sequential, becoming more complex as students learn more advanced programming and problem-solving techniques. The course had one classroom session per week, and each project assignment lasted 4 weeks.

Each student group worked together on each project using the same computer. These sessions occurred after class, and each project required two to three sessions of 45 to 60 min each. Group members were requested to play the roles of "driver," "navigator," and "monitor"; that is, one member typed the code, one planned solutions, and one watched for coding errors, respectively. Participants could also rotate their roles during a session.

Figure 1 briefly summarizes the two projects analyzed in this study. The first project asked students to develop a text-based fish pond simulation. The pond, fish, bait, and fishing hook were displayed as text characters on the console. All these objects in the fish pond were moving in a required pattern; for example, four fishes moving from the left edge of the fish pond to the right edge horizontally, then appearing again on the left edge, like the pond was wrapped around. The learning goal of the first project was to improve students' procedural-oriented programming ability and fundamental knowledge about basic control structures such as loops and conditions, primitive data types, strings, and multidimension arrays, as well as defining and calling custom methods.

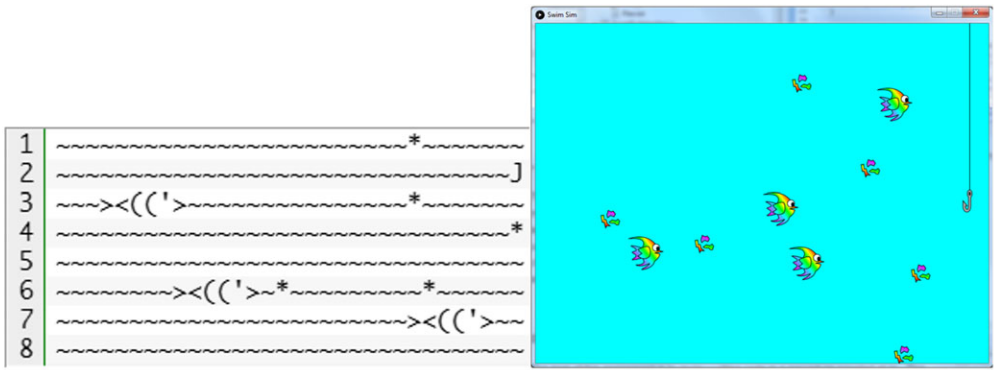
In the second project, the students were asked to create a graphical user interface (GUI) version of fish pond. All the objects within this microworld were no longer characters but became images in a window. Further, user interaction was added; for example, mouse clicking in the window will cause the fish hook to appear in the same place as the mouse was clicked. The learning goal of the second project was to reorganize a real-time simulation into a more general form by making use of instantiable objects of predefined types. Students were also required to apply provided classes and objects for GUI programming and a callback method to facilitate human interactions with simulation.

3.3 | Data collection and coding scheme

We collected conversation data from the two selected groups during their collaborative programming sessions. Camtasia, a screen cast software, was used to record the programming process in visual studio 2015 IDE environments for C++ programming, group conversation during collaborative programming, and webcam video of all group members. Thus, the programming actions are coordinated with the groups' problem-solving conversations. The video recordings were used for speaker recognition and to identify emotions and gestures. We collected 473 min of video in total.

To code the verbal interactions of the groups, the transcription of interaction was divided into conversational turns, defined as a change of speaker. We obtained 1,533 conversational turns from the audio–video data. Each conversational turn could be coded with one or more codes. Data were coded by two independent researchers, resulting in a good strength of agreement (kappa value for each code above 0.8). Differences in scoring were resolved through discussion.

We adapted Brennan and Resnick's (2012) three-dimension CT framework as a coding scheme to analyze the collaborative conversations (see Table 1). The first dimension (computational concepts) refers to the elements such as sequences, loops, conditionals, operators, and data structures that are present in many programming languages; the second dimension (computational practice) refers to activities, such as being incremental and iterative, reusing and remixing, testing and debugging, as well as modularizing and abstracting, that designers use to create programs; and the third dimension (computational identity) focuses on the perception of computing, that is, how students see themselves within the computing field and links to their future career, such as expressing and questioning.



```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~J
~~~><(('>
~~~~~~~~~~~~~~~~~~~~~*~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~><(('>~*~~~~~~~~~~~~*~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~><(('>~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

FIGURE 1 Interface of project 1 (left) and project 2 (right) [Colour figure can be viewed at wileyonlinelibrary.com]

TABLE 1 Coding scheme of computational thinking

Dimension	Codes	Definition
Concepts	Sequence (C. Sequence)	A series of individual steps or instructions that can be executed by a computer
	Loops (C. Loop)	A mechanism for running the same sequence multiple times
	Conditions (C. Condition)	the ability to make decisions based on certain conditions, which supports the expression of multiple outcomes
	Operators (C. Operator)	Provide support for mathematical, logical, and string expressions, enabling the programmer to perform numeric and string manipulations
	Data (C. Data)	Concerns about data types and storing, retrieving, and updating values of data
Practice	Incremental and Iterative (P. Increment and Iterative)	Identifying key concepts in programming, planning, and implementing design are adaptive and iterative process, decomposing complex tasks into simple subtasks and goals.
	Testing and Debugging (P. Test and Debug)	addressing programming problems and dealing with unexpected programming outcomes
	Reusing and Remixing (P. Reuse and Remix)	Finding ideas and code to build upon, creating things much more complex than students could have created on their own
	Abstracting and Modularizing (P. Abstract and Module)	Recognizing patterns and generalizing from specific instances
Identity	Expressing (I. Express)	See computation as more than something to consume, computation as a medium for self-expression
	Questioning (I. Question)	Negotiate the realities of the technological world regarding computing affordances and limitations

3.4 | Data analysis approach

To answer RQ1 (What CT patterns do novice programmers exhibit when they collaborate on the development of a software application?), the study employed discourse analysis (Gee, 2005) on the excepts of conversations during different stages of collaborative programming. To provide statistical warrants that support the qualitative discourse analysis and answer RQ2 (Do novice programmers follow the different trajectories of CT competence development based on the collaborative programming activities of their group?), we then conducted an ENA on all verbal interactions of the two groups.

The process of creating epistemic network models is explained in detail elsewhere (Shaffer, 2017; Shaffer et al., 2016; Shaffer & Ruis, 2017), but in brief, ENA uses a sliding window to construct a network model for each turn of talk in the data. Connections in the network are defined as the co-occurrence of codes in the current turn of talk and codes within the recent temporal context, which we defined as each line plus the four previous lines based on our qualitative analysis of the data (a window size of 5 turns of talk). Co-occurrence of elements in a given window of discourse is a good indicator of socio-cognitive connection (Landauer, McNamara, Dennis, & Kintsch, 2007; Lund & Burgess, 1996; Wu, Wang, Spector, & Yang, 2013). The resulting networks were aggregated for all turns of talk for each unit of analysis (individual in a group), such that each individual was represented by a vector whose elements were the number of co-occurrences between each pair of codes for that person. We normalized the matrix of co-occurrence vectors to account for variation in the amount of talk between individuals and performed a dimensional reduction via singular value decomposition.

Networks were visualized using two coordinated representations: (a) plotted points representing each individual's network, where the point indicates the location of an individual's network in the projected space (or ENA space) created by the first two dimensions of the dimensional reduction and (b) a weighted network graph in which the nodes correspond to codes, and the thickness of the edges is proportional to the relative frequency of connection between two codes. The positions of the network graph nodes are fixed across all networks in the model, and those positions are determined by an optimization algorithm that minimizes the difference between the plotted points and their corresponding network centroids. Thus, points that are located toward the extremes of a dimension have network graphs with strong connections between nodes located on those extremes. As a result, dimensions in ENA space distinguish individuals (or collectively, groups) in terms of connections between codes whose nodes are located at the extremes of the dimensions. All analyses were conducted with the ENA web tool (http://app.epistemicnetwork.org).

4 | RESULTS

Groups' programming performance was determined through assessment of all groups' programming artifacts based on four criteria, including code correctness, function completeness, code cleanness, and design creativity, with each criterion scored from 0 to 5 points. The full score was 20, and the mean score of all 15 groups was 14.50 (*SD* = 4.19). We divided groups into high-performing and low-performing conditions based on whether their score was greater than or lower than the mean, respectively. For this exploratory study, two groups were selected for both quantitative and qualitative analysis, with one from high-performing groups (HighGp; total score 18.6) and the other from the low-performing groups (LowGp; total score 13.84). The HighGp included three male students, called GZR, ZMZ, and LYX. The (LowGp) included two male students and one female student, called WXL, ALM, and CJM (female). All names are pseudonyms in this study.

For each group, we analysed qualitatively three tasks, that is, the text-based fish pond printout, making code reusable, and the GUI-based fish pond simulation, for each group during two different projects. We explored collaborative programming behaviours using discourse analysis first and then conducted a quantitative ethnographic analysis based on our qualitative findings.

4.1 | Text-based fish pond printout

The first task in the first project asked students to print a fish pond with tilde characters representing water in the size of 32 columns and eight rows on the console (see Figure 1 left). The HighGp planned to complete the task in a top–down way, from computing expressions to definition of the data and method, and then to knowledge about method parameters and the scope of the variables (see Table 2). First, GZR (#1) and LYX (#2) identified the task goal and decomposed the goal into two computing steps, that is, pond generation and pond printout. Then, ZMZ (#3) asked how to achieve the first step with programming methods. They then began discussing what they knew about methods and calling (i.e., computing concepts), followed by more details of how the method can manipulate data and in what scope (i.e., computing practice). Overall, the HighGp built connections between I. Express, P. Abstract and Module, C. Data, and C. Sequence at this stage.

In contrast, the LowGp struggled with the technical implementation of computing expressions, such as how to apply the concept of a loop to realize the fish pond, at an early stage of programming (see Table 3). They failed to differentiate the computing difference between variable assignment and printout, for example, WXL (#8) and WXL (#12). They took some time to consolidate their prior conceptual knowledge of nested loops, for example, WXL (#8), CJM (#9), and WXL (#10), and two-dimensional arrays, for example, ALM (#11) and WXL (#12). Therefore, the LowGp mainly developed connections between I.express and C. Data or C.Loop.

TABLE 2 High-performing group discourse in text-based fish pond visualization

Seq. #	Utterance	Codes
1 GZR	We need to define fish pond using an array and create a function to fill the fish pond with water.	C.Data, P.Abstract&Module, I.Express
2 LYX	We need to fill water into fish pond first and then print it out.	C.Sequence, I.Express
3 ZMZ	Where should we put this function, inside the main function or outside it?	C.Sequence, P.Abstract&Module
4 LYX	You can define this function outside the main function and then call it inside.	C.Sequence, P. Abstract&Module
5 GZR	Let us write this fill Fish Pond function first. We need to think about how many parameters it needs.	P. Abstract&Module
6 LYX	First, the array should be one of them, and then water should be a char type data which is also a parameter, is not it?	C.Data, P.Abstract&Module
7 GZR	So how many parameters should we need?	P.Abstract&Module
8 ZMZ	Can we define the fish Pond array just inside this function instead of in the main function?	C.Data, P.Abstract&Module
9 GZR	Main function must also have an array.	C.Data, P.Abstract&Module
10 LYX	The array is needed globally.	C.Data
11 GZR	Because you need to output in the main function.	P.Abstract&Module

TABLE 3 Low-performing group discourse in text-based fish pond visualization

Seq. #	Utterance	Codes
1 ALM	Then we need to fill the array with water.	I.Express
2 WXL	Yes, we will define an array. Because the array contains character, so we need to define a char-type array, right?	C.Data
3 CJM	A very big size...	C.Data
4 ALM	The pond size is 32*8 and we need to fill it with water.	C.Data, I.Express
5 WXL	Yes, we fill the water first and then covered using fish at some place. Now we use for loop to fill with water.	I.Express, C.Loop
6 ALM	How do you plan to fill the pond using for loop?	I.Express, C.Loop
7 CJM	Two nested for loop.	C.Loop
8 WXL	Just output it using printf, something like that. We use the first for loop to print the row	C.Loop
9 CJM	First loop is row or column?	C.Loop
10 WXL	No difference, just cover every single row and column. Here is the problem, our row and column are all numbers, now we need...	C.Loop, C.Data
11 ALM	It's just the change of address, we need to fill something into this address. I suggest we keep assigning water to i and j. right?	C.Data, C.Operator
12 WXL	No. we need to print water in the for loop. We do not assign it to i and j, but a[i][j], some place in the array. We fill each place with water and when i and j change, the place also changed accordingly.	C.Data, C.Operator, C.Loop

4.2 | Make code reusable

During project one, the idea of abstraction was introduced so that students were required to revise their programme to be more general and robust. The HighGp, therefore, created three functions to place the fish, bait, and fish hook in the pond, respectively. Then, they thought it was too redundant and were trying to figure out how to combine these three functions into a more general method (see Table 4). They shared a common goal of creating a general method based on previous codes, for example, LYX (#1) and ZMZ (#2) but came up with different plans, reflecting their different understandings of abstraction. After GZR (#6) pinpointed incongruities in the other two members' plans, ZMZ (#7) realized the difference and agreed with LYX's solution. After reaching this shared understanding of abstraction, LYX (#8) further proposed to identify and address differences among methods so as to define a general method. Therefore, the major connections built by the HighGp during this stage were within computing practice (i.e., P. Increment and Iterative, P. Reuse and Remix, and P. Abstract and Module).

Unlike the HighGp's programming strategy, the LowGp first wrote all source code in the main function including object placement, object movement, and object overlapping detection. Then, they spent more time rewriting their code using the bottom–up method in this encapsulation phase (see Table 5). For example, WXL (#7) engaged primarily in remixing rather than abstraction, which is less efficient and more error prone. WXL (#3) and WXL (#9) kept tinkering with the whole programme to delete useless temporal variables and modify the name of variables. The LowGp thus primarily made the connection between P. Increment and Iterative and C. Data.

TABLE 4 High-performing group in make code reusable

Seq. #	Utterance	Codes
1 LYX	The placeFishInPond function seems the most complex. How about we adjust the new function based on this one?	P.Increment&Iterative
2 ZMZ	Hmmm, I think it makes sense. let us do it. We need to write three sections, one is about fish, one about baits and one about fish hook. let us check what needs to be modified in the fish part.	P.Reuse&Remix, P.Increment&Iterative
3 LYX	Regarding the placing fish section, we determine the position of fish tail first and add one step in the second loop to print the whole fish on the screen. We can do the similar here. Because baits and fish hook both are represented by one character, that is to say, the second loop only run one time.	I.Express, C.Loop, C.Data
4 ZMZ	That is to say we actually do not need the second loop for the remaining two sections.	P.Reuse&Remix, C.Loop
5 LYX	It does not mean we do not need it. We just need to define a variable to represent times of repeat in the second loop so that we can change according to the length of an object.	C.Data, C.Loop, P.Abstract&Module
6 GZR	I think you two have different thoughts. ZMZ means to combine three function into one by copying the code of three and paste together, while LYX's point is to achieve the three functionalities using one method by adjusting the value of its parameters. So which one we should choose?	P.Reuse&Remix, P. Abstract&Module
7 ZMZ	I think LYX's solution is more reasonable and simpler. Mine is more redundant.	P.Abstract&Module
8 LYX	So we should find out what are the differences among these three and try to address them.	P.Abstract&Module

TABLE 5 Low-performing group in make code reusable

Seq. #	Utterance	Codes
1 ALM	We need to separate the code into different part, such as fish, fish pond, fish hood, and movement, right?	P.Abstract&Module
2 CJM	We can separate the code about fish pond first.	P.Reuse&Remix
3 WXL	Now we need to modify our code. Here we do not need to declare variables x, y, and m.	P.Increment&Iterative, C.Data
4 ALM	Why?	
5 WXL	Because they do not exist in this encapsulated method.	P.Increment&Iterative
6 CJM	Now we only need to fill fish pond. We only need to assign the element of the array with water.	I.Express, C.Data
7 WXL	And we need to be careful we should change all names of the same variables here.	P.Increment&Iterative
8 ALM	To pass arguments to the method	C.Data
9 WXL	We named this parameter water, so we should change all variable a in the method to water. We should encapsulate and revise these methods one by one.	C.Data, P.Increment &Iterative

4.3 | GUI-based fish pond simulation

In the second project, students were required to change the text-based solution into a GUI-based fish pond simulation. To add more entertaining features to this game, GZR in the HighGp proposed to change the hook movement from constant speed into acceleration. The following excerpt shows how they debug a problem with achieving this goal (see Table 6). Initially, they focused on the data operation mechanism, for example, LYX (#2), ZMZ (#6), LYX (#7), ZMZ (#10), LYX (# 11), and GZR (#12), and adopted a hypothetico-deductive approach to troubleshoot the problem, for example, LYX (#2), LYX (#5), and ZMZ (#6). Even though they were lacking key debugging skills, such as breakpoints and detailed conceptual knowledge of graphics rendering, their reasoning and argumentation strategies helped build these competences. That is to say, they were developing connections between other CT components and P. Test and Debug.

The LowGp had also developed new knowledge going into the second project (see Table 7) by examining programming artifacts in the first project, for example, WXL (#1). They successfully aligned their practice of class definition and object calling with object-oriented thinking in programming, for example, WXL (#3), WXL (#5), and CJM (#6). In other words, they began re-planning their programme from the top–down and made connections between computing practice (P.Abstract and Module and P. Increment and Iterative) and computing identity (I.Express and I.Question).

4.4 | Epistemic network modelling

To corroborate the above qualitative findings and further explore the differences between the two groups' CT competence, we conducted an ENA analysis on the coded conversation data. As shown in Figure 2, each point is the centroid of a students' epistemic network of CT competence; the squares are the means of all group members' centroids, and the black boxes are the 95% confidence intervals for the means. The first dimension (X) accounted for 34% of the variance in the data. The second dimension (Y) accounted for 27% of the variance in the data. To compare the means statistically, we computed an independent-samples *t* test. The difference on the first dimension was statistically significant: MeanHighGp = 0.24, MeanLowGp = −0.24, *t* = 4.14, *p* = 0.01, Cohen's d = 3.38. There was no significant difference on the second dimension.

Because each network (Figure 2) and the corresponding mean networks (Figure 3) were in the same projection space, we can interpret the meaning of the first and the second dimensions of this space based on the node positions in Figure 3. As we can see, P. Reuse and Remix and P. Increment and Iterative are located on the left side of the projection space; these codes correspond to *bricolage*, a process described by Turkle and Papert (1990) as an interaction between programmer and programme, navigating through missteps and planning little more than a step ahead. P. Test and Debug and C. Loop are located on the right side of the projection space, and these codes

TABLE 6 High-performing group in graphical user interface-based fish pond simulation

Seq. #	Utterance	Codes
1 ZMZ	It moves very very slow, does not it?	P.Test&Debug
2 LYX	Anyhow, dy should be a positive number, because it must move upward. But it does not. I think it might because we define dy as an integer. Suppose it is 50, which divided by 500 is 0.1 so that dy is zero. And it will keep zero which means it will not move.	C.Data, C.Operator, I.Express
3 GZR	Can we change it to double?	C.Data
4 LYX	Then it also affects the following integer array of hook's position.	C.Data
5 LYX	Can we use double type to express pixel on the screen?	C.Data, I.Question
6 ZMZ	What about we do not divide dy by 500?	C.Operator
7 LYX	This is the formula for updating hook speed. If we multiply the coefficient by 10, it might move very very fast.	C.Operator, I.Express
8 ZMZ	We can give it a try.	P.Test&Debug
9 GZR	I am afraid double type data will increase computation dramatically and take up a lot of memory space. But let us put it aside and try using double type first.	C.Data, I.Question
10 ZMZ	They move now! But very very slow. I still think we can change the coefficient smaller, say, 250.	P.Test&Debug, C.Operator
11 LYX	Well, 500 is to make sure the final speed will not be too fast. Let us wait to see it reaches the top. If the final speed is still very slow. We can change the coefficient then. Hmmm, it seems so...	P.Test&Debug, I.Express, C.Operator
12 GZR	Try 50. I think we can change to 50 and return to use integer.	C.Operator, C.Data
13 LYX	Let us run it.	P.Test&Debug
14 GZR	Looks much better.	P.Test&Debug
15 ZMZ	But it seems a little bit difficult.	I.Express
16 GZR	Difficulty makes player feels more fun. I think the effect is great.	I.Express, I.Question

TABLE 7 Low-performing group in graphical user interface-based fish pond simulation

Seq. #	Utterance	Codes
1 WXL	Yes, we have object generation, object placement, and object movement, right? We can call this method from outside. If we hold procedure-oriented thinking, we can put all codes inside ofApp, but if we want to adopt object-oriented thinking, we can put them into an object. We declare these functions with a public access and then call them from another cpp file	P.Reuse&Remix, P. Abstract&Module I.Question
2 ALM	OK. So we have a moveFish, how to write this method?	P.Abstract&Module
3 WXL	We usually declare a method in the head file and put definition in the source file. moveFish has no return value, so it's void. And it has no argument to pass, it just changes the position.	P.Abstract&Module
4 ALM	Anything else?	
5 WXL	We also need to use those object's variables. As our instructor said, an object's property should keep private, so other object cannot access directly. So we need to define method to achieve this. Here in this project, we need methods to get position values of fish, baits and hook.	P.Increment&Iterative, I. Question, P. Abstract&Module, I. Express
6 CJM	What is the difference between these methods and moveFish?	P.Abstract&Module
7 WXL	I think there are two differences. moveFish has no return value, but here we need to return double-type position value. Second, moveFish is for moving operation without passing arguments. But here, for example, getFish, we need an parameter to pass the fish number so that we can get the position value of a specific fish.	P.Abstract&Module, P. Increment&Iterative

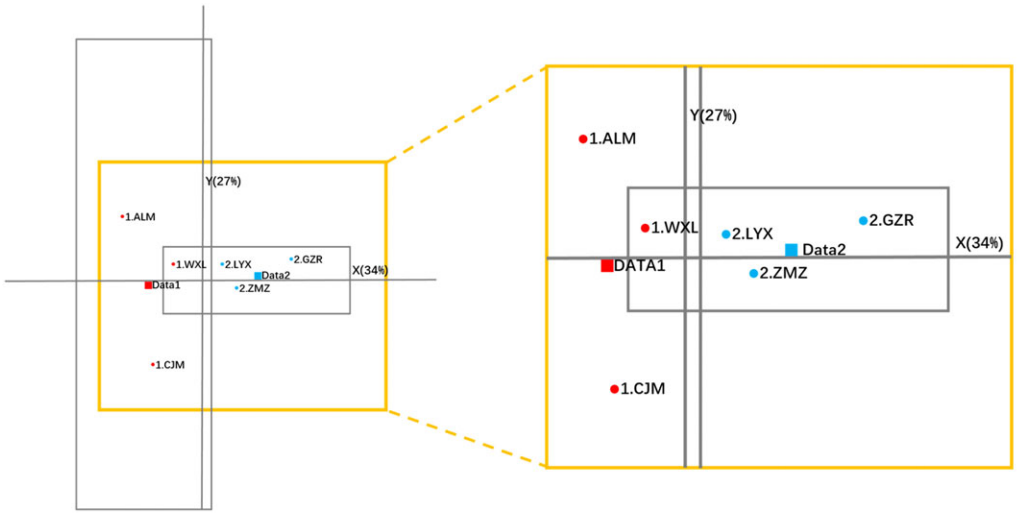


FIGURE 2 Students' epistemic networks of computational thinking competence in projection space. The group that generate high-quality programming artifact (in blue) and low-quality one (in red) [Colour figure can be viewed at wileyonlinelibrary.com]

correspond to tracing interactive programme code, a fundamental skill for novice programmers to develop (Lister, 2016). Hence, the first (X) dimension can be interpreted as the bricolage versus tracing dimension. Similarly, we can interpret the second (Y) dimension as the construction versus reasoning dimension because the CT codes to the top and the bottom of the space are related to creating computational artifacts (C.Data, I. Express and P. Increment and Iterative) and operational reasoning (P.Reuse and Remix, C. Loop, and C.Operation; Lister, 2011; Wilkerson-Jerde, 2014), respectively.

To interpret the statistical difference between these two groups, we plotted their mean networks (see Figure 3). The HighGp has more connections in the upper right area of the projection space, reflecting the stronger connections to P. Test and Debug, and C. Loop, whereas the LowGp has more connections in the lower left area of the projection space, reflecting stronger connections to I. Question, P. Reuse and Remix, P. Increment and Iterative, and C. Sequence. Both groups had strong connections to I. Express. In other words, the HighGp focused more on computing artifact construction and code tracing whereas the LowGp focused more on algorithmic reasoning and code bricolage.

Next, we examined students' competence development by analysing the trajectories of their epistemic networks across different stages of two sequential projects. The HighGp completed the first project and the second project both in two sessions, whereas the LowGp completed the first project in three sessions and the second one in two sessions. The time spent on each project by two groups are reported in Table 8. These differences in time allocation for the programming projects were determined by the students.

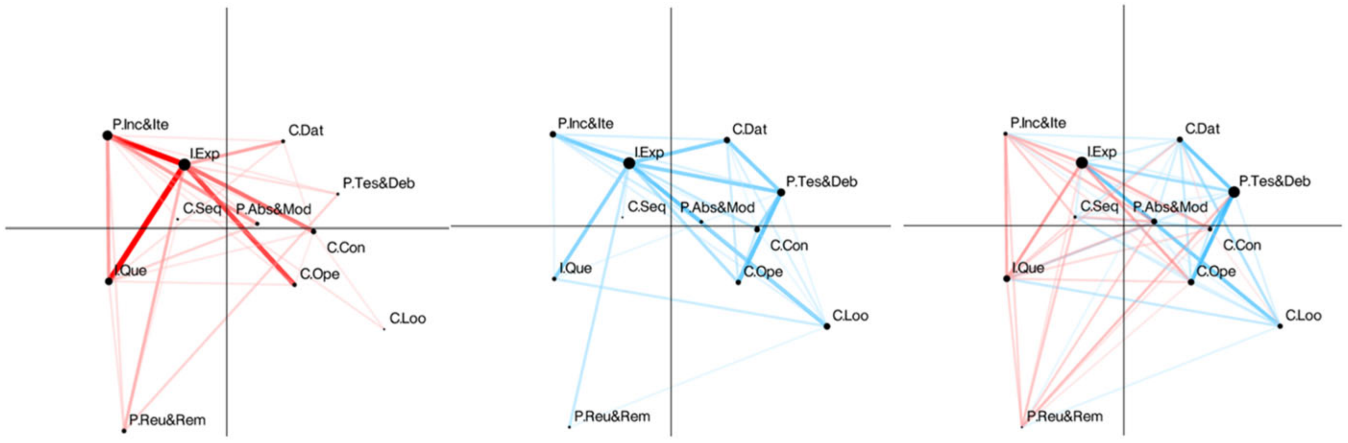


FIGURE 3 Mean network of low-performing group (in red) and high-performing group (in blue) and the difference network [Colour figure can be viewed at wileyonlinelibrary.com]

The network trajectory of the HighGp's members consists of four points, and the trajectory of the LowGp's members consists of five points, with each point representing the centroid of the CT competence network at a certain programming stage. Figure 4 shows that the two groups had different trajectories in CT competence development, whereas members of each group had relatively similar trajectories. Overall, the two groups' trajectories were generally convergent.

Although two groups' initial CT competence networks were quite different, with the HighGp characterized by stronger connections to P. Abstract and Module and the LowGp characterized by stronger connections to P. Reuse and Remix, the trajectories showed the convergence of these two groups towards similar CT competence networks, that is, networks characterized by more balanced connections among different CT components.

By plotting mean networks for each point in this projection space as shown in Figure 3, we explored the changes in students' epistemic networks to see how often different CT components connected with each other at different stages of programming learning comparatively. Here, we selected GZR and LYX in HighGp and CJM in LowGp to compare novice programmers' CT competence development both between and within groups. Table 9 shows their individual networks at the first two sessions and the fifth session of programming.

In the first stage, CJM from the LowGp had strong connections between C. Data, P. Reuse and Remix, and P. Test and Debug, whereas both GZR and LYX in the HighGp had strong connections between P. Abstract and Module, C. Data, I. Express, and other components. This suggests that the HighGp focused more on top–down approaches to framing the problem, data representation, and computing planning,

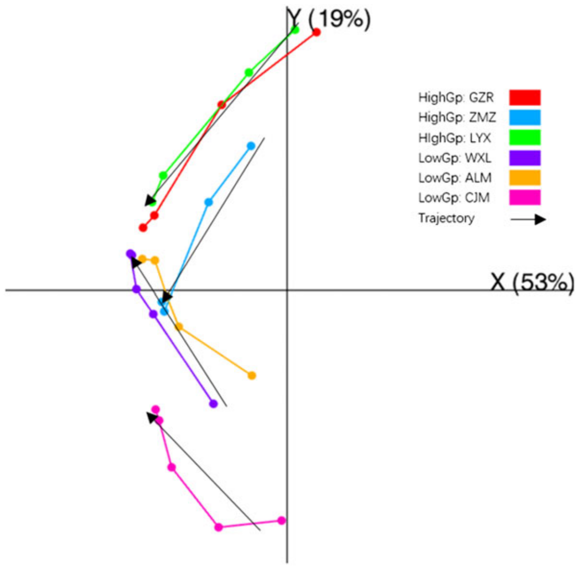


FIGURE 4 Trajectory of group members' network model in a projection space [Colour figure can be viewed at wileyonlinelibrary.com]

whereas the LowGp used a trial-and-error approach of data manipulation and reusing.

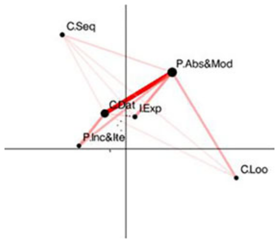
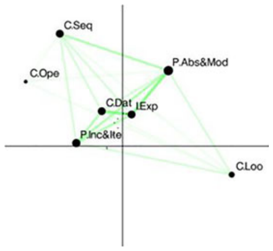
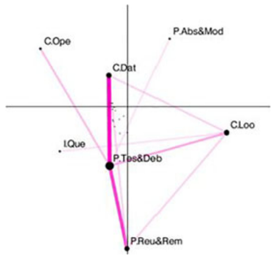
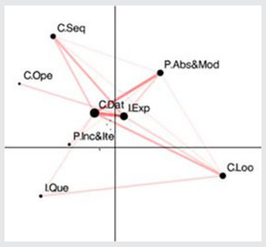
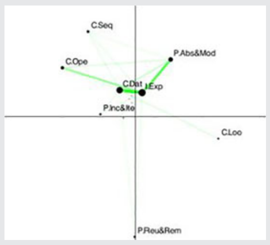
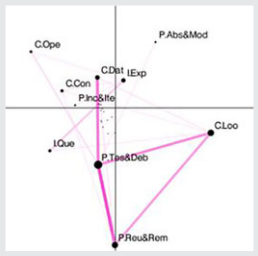
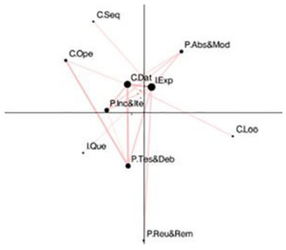
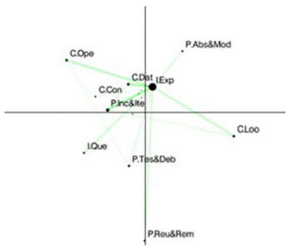
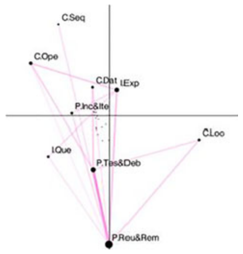
In the second stage, GZR and LYX exhibited stronger connections among C. Data, I. Express, and P. Abstract and Module as well as more connections to C. Operation. Besides, new components like I. Question and P. Reuse and Remix had emerged in their networks, respectively. Meanwhile, new components like P. Increment and

TABLE 8 Time spent on different sessions of two sequential projects by two groups

	Project 1			Project 2	
	Sesssion 1 (min)	Session 2 (min)	Session 3 (min)	Session 4 (min)	Session 5 (min)
HighGp	40	67	-	64	45
LowGp	48	58	36	47	68

Note. HighGp: high-performing group; LowGp: low-performing group.

TABLE 9 Computational thinking competence networks in different stages for students from two groups

	Stage1	Stage2	Stage5
GZR in HighGp			
LYX in HighGp			
CJM in LowGp			

Iterative, C. Condition, and I. Express emerged in CJM's network. The similarity between CJM's network and the HighGp members' thus increased.

In the fifth stage, all three students' network had more connections with I. Express, P. Test and Debug, P. Reuse and Remix, and P. Increment and Iterative, which suggested that both HighGp and LowGp developed more comprehensive CT competence networks. For instance, both groups were learning how to reuse predefined functions or improve programmes to express new design (i.e., connections between I. Express, P. Reuse and Remix, and P. Increment and Iterative) and how to debug unexpected variable outcomes (i.e., connections between P. Test and Debug and C.Data).

5 | DISCUSSION

This study examined how novice programmers develop CT competence by interacting with each other during collaborative programming activities. The findings from our qualitative discourse analysis revealed that the HighGp adopted a top–down perspective. They planned conceptual and expressive design first and then analytically achieve this design goal using computational artifacts. In contrast, the LowGp adopted a bottom–up perspective at the beginning but transformed to a top-down approach by the end. They tended to consider the affordances and limitations of technical implementation to search for a better solution and to refine their software design. Whereas previous studies of novice–expert difference suggested that programming experts use top–down strategies (Robins et al., 2003), our study revealed that CT competence can be developed in collaborative contexts even when students begin with a bottom–up approach. This research thus suggests that collaborative programming in project-based learning contexts can help novice learners develop CT competence via heterogeneous trajectories.

Our quantitative modelling of the groups CT, using epistemic network trajectories as well as individual networks at different time periods, help deepen our understanding of CT competence development in high and LowGps. First, the group mean networks provide a holistic view in terms of what competence components were strengthened more for different groups. As we can see from the difference network in Figure 3, the HighGp developed more skills in integrating different data type definition and manipulation with looping structure application to formulate complex functions and kept sharpening debugging techniques during programming (i.e., tracing skills). On the contrary, the LowGp, during the whole programming process, focused more on the sequence of their programme and kept reusing their code to make minor change (i.e., tinkering activity). They also kept questioning the expected outcomes without developing the debugging technique. These findings were consistent with previous studies that tracing skills reflect an intermediate level of programming (Lister, Fidge, & Teague, 2009), whereas tinkering behaviour often suggests an ad hoc trial-and-error approach in novice programming (Berland et al., 2013).

Second, the two groups' network trajectories show different competence development pathways. For the HighGp, their CT network changed from focusing on function-level and class-level abstraction and modularizing at the beginning to creating and applying different types of data to achieve their expressive design and finally to keep improving their programming artifacts. The LowGp emphasized reusing and remixing at the very beginning, which suggests that they failed to make plans before programming and that they did not understand or think deeply about both the project problem and drafted code. They continued to question the requirement of the project and even the functionality of the programming environment regardless of syntax or logical problems in their code. However, by the end of the programming stage, the LowGp presented similar CT competence patterns to the HighGp s, which indicates that their CT competence developed through a different pathway. The trajectory of both groups reflected their heterogeneous CT competence patterns at the beginning but evolved towards convergence. This finding corroborates the argument that programming ability is something that is learned rather than something innate (Robins et al., 2003; Teague & Lister, 2014) and further suggests that students who struggle to learn programming can also develop their CT competence through collaborative learning.

Third, a closer analysis of individual networks across different programming stages offered detailed instances of CT competence development trajectories. This study reveals that students in the LowGp changed from debugging problems by reusing data and loop structure to reusing code to achieve incremental tasks of computing expression, whereas the HighGp's students changed from applying fundamental concepts to realize computational expression through abstraction to more balanced integration of computing expression, debugging, reusing, and iteration of programmes. These epistemic network findings provide statistical warrants for our discourse analysis results and contribute to our understanding of heterogeneous CT development trajectories between and homogeneous trajectories within these two groups.

6 | CONCLUSIONS

This study employed an innovative quantitative ethnography approach to provide evidence of differences in CT competence between high and LowGp s as well as their convergent development trajectories in collaborative programming, the results of which corroborates our qualitative findings from group discourse analyses. The study suggests that quantitative ethnography is a promising approach for understanding and modeling CT competence and its development. The method also addresses the challenge of inferring individual students' development of CT competence from collaborative learning, particularly when there is no knowledge about how professional expertise is developed in a specific domain nor about ways to separate individual performance from joint activities (Enyedy & Stevens, 2014; Lajoie, 2003).

However, this exploratory study also suffered from some limitations. First, the codes underlying the three dimensions of CT as proposed by Brennan and Resnick (2012) were adapted in this study to fit the designed programming project content, which may be different if they were applied to other programming contexts, such as projects in using thread and database techniques. This is, of course, beyond the novice level of programming, and adjustment of the coding would likely be necessary. Second, we only selected two groups and analysed their collaborative programming in two projects. Due to the small sample size, the model created from this study would not necessarily generalize to other contexts. In future study, we will increase our sample size and extend to other CT development contexts even without programming.

ACKNOWLEDGEMENTS

This work was funded in part by the Humanity and Social Science Youth foundation from the Ministry of Education of China (Grant Number 16YJC880085), the Peak Discipline Construction Project of Education from East China Normal University, Eastern Scholar Chair Professorship Fund from Shanghai Municipal Education Commission of China (Award Number JZ2017005), the National Science Foundation (Grant Number DRL-1661036, DRL-1713110), the Wisconsin Alumni Research Foundation, and the Office of the Vice Chancellor for Research and Graduate Education at the University of Wisconsin-Madison, USA. The opinions, findings, and conclusions do not reflect the views of the funding agencies, cooperating institutions, or other individuals. The authors also appreciate Prof. David Williamson Shaffer for his valuable comments on this manuscript.

ORCID

Bian Wu https://orcid.org/0000-0003-0391-2630

REFERENCES

Beck, L., & Chizhik, A. (2013). Cooperative learning instructional methods for CS1: Design, implementation, And Evaluation. *ACM Transactions on Computing Education*, *13*(3), 10–21.

Berland, M., & Lee, V. R. (2011). Collaborative strategic board games as a site for distributed computational thinking. *International Journal of Game-Based Learning*, *1*(2), 65–81. https://doi.org/10.4018/ijgbl.2011040105

Berland, M., Martin, T., Benton, T., Petrick Smith, C., & Davis, D. (2013). Using learning analytics to understand the learning pathways of novice programmers. *Journal of the Learning Sciences*, *22*(4), 564–599. https://doi.org/10.1080/10508406.2013.836655

Brennan, K., & Resnick, M. (2012). New frameworks for studying and assessing the development of computational thinking. Paper presented at the proceedings of the 2012 annual meeting of the American Educational Research Association, Vancouver, Canada.

Chao, P. Y. (2016). Exploring students' computational practice, design and performance of problem-solving through a visual programming environment. *Computers & Education*, *95*, 202–215. https://doi.org/10.1016/j.compedu.2016.01.010

Czerkawski, B. C., & Lyman, E. W. (2015). Exploring issues about computational thinking in higher education. *TechTrends*, *59*(2), 57–65. https://doi.org/10.1007/s11528-015-0840-3

Denner, J., & Werner, L. (2007). Computer programming in middle school: How pairs respond to challenges. *Journal of Educational Computing*

Research, *37*(2), 131–150. https://doi.org/10.2190/12T6-41L2-6765-G3T2

Denner, J., Werner, L., Campe, S., & Ortiz, E. (2014). Pair programming: Under what conditions is it advantageous for middle school students? *Journal of Research on Technology in Education*, *46*(3), 277–296. https://doi.org/10.1080/15391523.2014.888272

Emurian, H. H., Holden, H. K., & Abarbanel, R. A. (2008). Managing programmed instruction and collaborative peer tutoring in the classroom: Applications in teaching Java™. *Computers in Human Behavior*, *24*(2), 576–614. https://doi.org/10.1016/j.chb.2007.02.007

Enyedy, N., & Stevens, R. (2014). Analyzing collaboration. In R. K. Sawyer (Ed.), *The Cambridge handbook of the learning sciences* (pp. 128–150). New York, NY: Cambridge University Press. https://doi.org/10.1017/CBO9781139519526.013

Ericsson, K. A., & Simon, H. A. (1998). How to study thinking in everyday life: Contrasting think-aloud protocols with descriptions and explanations of thinking. *Mind, Culture, and Activity*, *5*(3), 178–186. https://doi.org/10.1207/s15327884mca0503_3

Foster, A., & Shah, M. (2016). Examining game design features for identity exploration and change. *Journal of Computers in Mathematics and Science Teaching*, *35*(4), 369–384.

Gee, J. P. (2005). *An introduction to discourse analysis theory and method* (2nd ed.). London: Routledge.

Jehng, J. C. J. (1997). The psycho-social processes and cognitive effects of peer-based collaborative interactions with computers. *Journal of Educational Computing Research*, *17*(1), 19–46. https://doi.org/10.2190/YHGG-RVGP-E60X-N9N3

Johnson, D. W., & Johnson, R. T. (1999). *Learning together and alone: Cooperative, competitive, and individualistic learning* (5th ed.). Boston: Allyn & Bacon.

Kafai, Y. B., & Burke, Q. (2013). The social turn in K-12 programming: Moving from computational thinking to computational participation. *Paper presented at the 44th ACM technical symposium on computer science education*.

Kafai, Y. B., Lee, E., Searle, K., Fields, D., Kaplan, E., & Lui, D. (2014). A crafts-oriented approach to computing in high school: Introducing computational concepts, practices, and perspectives with electronic textiles. *ACM Transactions on Computing Education*, *14*(1), 1–20. https://doi.org/10.1145/2576874

Kalaian, S. A., & Kasim, R. M. (2014). Small-group vs. competitive learning in computer science classrooms: A meta-analytic review. In R. Queiros (Ed.), *Innovative teaching strategies and new learning paradigms in computer programming* (pp. 46–63). Hershey, PA: IGI Global.

Khosa, D. K., & Volet, S. E. (2014). Productive group engagement in cognitive activity and metacognitive regulation during collaborative learning: Can it explain differences in students' conceptual understanding? *Metacognition and Learning*, *9*(3), 287–307. https://doi.org/10.1007/s11409-014-9117-z

Korkmaz, Ö., Çakir, R., & Özden, M. Y. (2017). A validity and reliability study of the computational thinking scales (CTS). *Computers in Human Behavior*, *72*, 558–569. https://doi.org/10.1016/j.chb.2017.01.005

Kwon, K., Liu, Y. H., & Johnson, L. P. (2014). Group regulation and social-emotional interactions observed in computer supported collaborative learning: Comparison between good vs. Poor Collaborators. *Computers & Education*, *78*, 185–200. https://doi.org/10.1016/j.compedu.2014.06.004

Lajoie, S. P. (2003). Transitions and trajectories for studies of expertise. *Educational Researcher*, *32*(8), 21–25. https://doi.org/10.3102/0013189X032008021

Landauer, T. K., McNamara, D. S., Dennis, S., & Kintsch, W. (2007). *Handbook of latent semantic analysis*. Mahwah, NJ: Erlbaum.

Leung, W. C. (2002). Why is evidence from ethnographic and discourse research needed in medical education: The case of problem-based learning. *Medical Teacher*, *24*(2), 169–172. https://doi.org/10.1080/01421590220125268

Lin, J. M. C., & Liu, S. F. (2012). An investigation into parent-child collaboration in learning computer programming. *Journal of Educational Technology & Society*, *15*(1), 162–173.

Lindsjørn, Y., Sjøberg, D. I., Dingsøyr, T., Bergersen, G. R., & Dybå, T. (2016). Teamwork quality and project success in software development: A survey of agile development teams. *Journal of Systems and Software*, *122*, 274–286. https://doi.org/10.1016/j.jss.2016.09.028

Lister, R. (2011). Concrete and other neo-Piagetian forms of reasoning in the novice programmer. *Paper presented at the Thirteenth Australasian Computing Education Conference*.

Lister, R. (2016). Toward a developmental epistemology of computer programming. *Paper presented at the 11th workshop in primary and secondary computing education*.

Lister, R., Fidge, C., & Teague, D. (2009). Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. *Paper presented at the SIGCSE Bull*.

Lister, R., Simon, B., Thompson, E., Whalley, J. L., & Prasad, C. (2006). Not seeing the forest for the trees: Novice programmers and the SOLO taxonomy. *Paper presented at the 11th annual SIGCSE conference on innovation and Technology in Computer Science Education*, Bologna, Italy.

Lund, K., & Burgess, C. (1996). Producing high-dimensional semantic spaces from lexical co-occurrence. *Behavior Research Methods, Instruments, & Computers*, *28*(2), 203–208. https://doi.org/10.3758/BF03204766

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. https://doi.org/10.1016/j.chb.2014.09.012

Maguire, P., Maguire, R., Hyland, P., & Marshall, P. (2014). Enhancing collaborative learning using paired-programming: Who benefits? *AISHE-J: The All Ireland Journal of Teaching and Learning in Higher Education*, *6*(2), 1411–14125.

Mangalaraj, G., Nerur, S., Mahapatra, R., & Price, K. H. (2014). Distributed Cognition in Software Design: An Experimental Investigation of the Role of Design Patterns and Collaboration. *MIS Quarterly*, *38*(1), 249–274. https://doi.org/10.25300/MISQ/2014/38.1.12

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, *13*(2), 137–172. https://doi.org/10.1076/csed.13.2.137.14200

Román-González, M., Pérez-González, J. C., & Jiménez-Fernández, C. (2017). Which cognitive abilities underlie computational thinking? Criterion validity of the Computational Thinking Test. *Computers in Human Behavior*, *72*, 678–691. https://doi.org/10.1016/j.chb.2016.08.047

Serrano-Cámara, L. M., Paredes-Velasco, M., Alcover, C. M., & Velazquez-Iturbide, J. Á. (2014). An evaluation of students' motivation in computer-supported collaborative learning of programming concepts. *Computers in Human Behavior*, *31*, 499–508. https://doi.org/10.1016/j.chb.2013.04.030

Shadiev, R., Hwang, W. Y., Yeh, S. C., Yang, S. J., Wang, J. L., Han, L., & Hsu, G. L. (2014). Effects of unidirectional vs. reciprocal teaching strategies on web-based computer programming learning. *Journal of Educational Computing Research*, *50*(1), 67–95. https://doi.org/10.2190/EC.50.1.d

Shaffer, D. W. (2012). Models of situated action: Computer games and the problem of transfer. In C. Steinkuehler, K. Squire, & S. Barab (Eds.), *Games learning, and society: Learning and meaning in the digital age* (pp. 403–433). Cambridge, UK: Cambridge University Press. https://doi.org/10.1017/CBO9781139031127.028

Shaffer, D. W. (2017). *Quantitative Ethnography*. Madison, Wisconsin: Cathcart Press.

Shaffer, D. W., Collier, W., & Ruis, A. R. (2016). A tutorial on epistemic network analysis: Analyzing the structure of connections in cognitive, social, and interaction data. *Journal of Learning Analytics*, *3*(3), 9–45. https://doi.org/10.18608/jla.2016.33.3

Shaffer, D. W., & Ruis, A. R. (2017). Epistemic network analysis: A worked example of theory-based learning analytics. In C. Lang, G. Siemens, A. F. Wise, & D. Gasevic (Eds.), *Handbook of learning analytics* (pp. 175–187). (n.p.): Society for Learning Analytics Research.

Teague, D., & Lister, R. (2014). Longitudinal think aloud study of a novice programmer. *Paper presented at the the Sixteenth Australasian Computing Education Conference*.

Teague, D., & Roe, P. (2008). Collaborative learning: Towards a solution for novice programmers. *Paper presented at the tenth conference on Australasian computing education*.

Turkle, S., & Papert, S. (1990). Epistemological pluralism: Styles and voices within the computer culture. *Signs*, *16*(1), 128–157. https://doi.org/10.1086/494648

Wang, S. L., & Hwang, G. J. (2012). The role of collective efficacy, cognitive quality, and task cohesion in computer-supported collaborative learning (CSCL). *Computers & Education*, *58*(1), 679–687. https://doi.org/10.1016/j.compedu.2011.09.003

Wellons, J., & Johnson, J. (2011). A grounded theory analysis of introductory computer science pedagogy. *Journal on Systemics, Cybernetics and Informatics*, *8*(6), 9–14.

Wilkerson-Jerde, M. H. (2014). Construction, categorization, and consensus: Student generated computational artifacts as a context for disciplinary reflection. *Educational Technology Research and Development*, *62*(1), 99–121. https://doi.org/10.1007/s11423-013-9327-0

Williams, L., & Kessler, R. (2002). *Pair programming illuminated*. Boston, MA: Addison-Wesley Longman Publishing Co., Inc.

Williams, L., Wiebe, E., Yang, K., Ferzli, M., & Miller, C. (2002). In support of pair programming in the introductory computer science course. *Computer Science Education*, *12*(3), 197–212. https://doi.org/10.1076/csed.12.3.197.8618

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

Wu, B., Wang, M., Spector, M., & Yang, S. (2013). Design of a dual-mapping learning approach for problem solving and knowledge constructionin ill-structured domains. *Educational Technology & Society*, *16*(4), 71–84.

Zhong, B., Wang, Q., Chen, J., & Li, Y. (2016). An exploration of three-dimensional integrated assessment for computational thinking. *Journal of Educational Computing Research*, *53*(4), 562–590. https://doi.org/10.1177/0735633115608444

How to cite this article: Wu B, Hu Y, Ruis AR, Wang M. Analysing computational thinking in collaborative programming: A quantitative ethnography approach. *J Comput Assist Learn*. 2019;35:421–434. https://doi.org/10.1111/jcal.12348