International Journal of Computer Science Education in Schools, April 2018, Vol. 2, No. 2
ISSN 2513-8359

Using Grey-based Mathematical Equations of Decision-making as Teaching Scaffolds: from an Unplugged Computational Thinking Activity to Computer Programming

Meng-Leong How[1]
Chee-Kit Looi[1]

[1]National Institute of Education, Nanyang Technological University, Singapore

DOI: 10.21585/ijcses.v2i2.24

Abstract

Computational Thinking (CT) is pervasive in our daily lives and is useful for problem-solving. Decision-making is a crucial part of problem-solving. In the extant literature, problem-solving strategies in educational settings are often conveniently attributed to intuition; however, it is well documented that computer programmers might even have difficulty describing about their intuitive insights during problem-solving using natural language (such as English), and subsequently convert what has been described using words into software code. Hence, a more analytical approach using mathematical equations and descriptions of CT is offered in this paper as a potential form of rudimentary scaffolding, which might be useful to facilitators and learners of CT-related activities. In the present paper, the decision-making processes during an unplugged CT activity are delineated via Grey-based mathematical equations, which is useful for informing educators who may wish to explain to their learners about the various aspects of CT which are involved in the unplugged activity and simultaneously use these mathematical equations as scaffolds between the unplugged activity and computer code programming. This theoretical manuscript may serve as a base for learners, should the facilitator ask them to embark on a software programming activity that is closely associated to the unplugged CT activity.

Keywords
grey-based mathematical equations, decision making, computational thinking, scaffolding for teaching, computer software programming, unplugged computational thinking activity

1. Introduction

In computer programming education, there might be an overemphasis on students' acquisition of the syntax of a programming language; often at the expense of development of problem-solving skills (McGill & Volet, 1997). Somers (2017) notices that programmers do not work on a problem directly. He quoted Nancy Leveson, a professor at the Massachusetts Institute of Technology who has been studying software safety for 35 years, who explains, "The problem is that software engineers don't understand the problem they're trying to solve, and don't care to. The reason is that they're too wrapped up in getting their code to work. The serious problems that have happened with software have to do with requirements, not coding errors." Hence, development of problem-solving skills should be a high priority for computer programming education.

Griffin (2016) points out that it is important for computer programmers to develop a mental model of a notional machine (du Boulay, O'Shea, & Monk, 1981), which is a rudimentary model that describes the instructions of a computer program for problem-solving. Strong interest in how the computer programmer could develop this mental model (by researchers such as Grover, Pea, & Cooper, 2015; Hu, 2011; Selby, 2013; Wing, 2008), have more precisely explicated this mental model of a notional machine into what is now known as Computational Thinking (CT). A generally accepted definition of CT is still developing (Selby, 2013); even the very definition of individual constituents of CT such as the concept of abstraction is still evolving (Cetin & Dubinsky, 2017). Nevertheless, in the present paper, for the purpose of "operationalising" CT concepts for utilisation of mathematical equations in decision-making and problem-solving using a computer programming language, we follow the conceptual framework offered by Gouws et al. (2013) who have more concisely elucidated CT concepts specifically for the field of education. The constituents of this mental model of CT offered by Gouws et al. (2013) include decomposition, algorithmic thinking, abstraction of data and

functionality, evaluation, and generalisation. Decomposition refers to the process of breaking down a problem into multiple steps, in order to solve it. Algorithmic thinking refers to the repetitive execution of patterns of instructions, which might involve loops for iteration or recursion. Abstraction of data and functionality refers to the notion of representations in data storage and the manipulation of those data in functions. Generalisation refers to the ability to create adaptable solutions that are reusable for a wider range of problems. Evaluation is the ability to select the best solution for a given problem, as well as to identify and correct errors.

CT is pervasive (Bundy, 2007); in our daily tasks, CT is particularly useful for problem-solving (Barr, Harrison, & Conery, 2011). Indeed, CT is indispensable to problem-solving in the real world, and is also considered to be essential in education (Wing, 2008). Efforts have already been made in many studies to delineate which aspects of CT might be explicitly learnt by a person who has participated in screen-based activities (by researchers such as Grover, 2015; Israel, Pearson, Tapia, Wherfel, & Reese, 2015; Monteiro, Salgado, Mota, Sampaio, & de Souza, 2016; Selby & Cynthia, 2015).

Visualisation of code can also be in the form of physical or kinaesthetic activities (also referred to as unplugged activities); not just on screen-based devices. Zagami (2012) posits that the computer programmer could understand programming concepts better from visualisation of how code works. Research into non-screen based unplugged activities (by researchers such as Bell et al., 2009; Cortina, 2015; Paul Curzon et al., 2014; Feaster, Segars, Wahba, & Hallstrom, 2011; Rodriguez, 2015; Taub, Armoni, & Ben-Ari, 2012; Taub, Ben-Ari, & Armoni, 2009; Thies & Vahrenhold, 2012; Thies & Vahrenhold, 2013) have demonstrated that they might potentially help learners to understand computing concepts kinaesthetically as they solve problems in the real world.

Research Problem

In the extant literature, decision-making strategies in problem-solving used by learners in educational settings might often just be conveniently assumed by educators to be naturalistic (Zsambok, 2014), or simply intuitive (Metcalfe & Wiebe, 1987; Pretz, 2008). However, it is well documented that computer programmers might have difficulties in describing about their intuitive insights during problem-solving using natural language (such as English), and subsequently convert what has been described using words into software code. In a previous study by Kordaki, Miatidis, and Kapsampelis (2008), the students constructed an algorithm intuitively in an activity using coins, but most of them had problems describing the procedure which they had used, when they tried to express it in natural language (English) and pseudocode. Boticki, Barisic, Martin, and Drljevic (2013) also observed that students might have difficulties translating their thoughts into a form that could be used in computers.

Decision making is that thinking which results in the choice among alternative courses of action; problem solving is that thinking which results in the solution of problems (Taylor, 2013, p. 48). Further, Taylor (2013) also points out that the processes in decision-making are also important to problem-solving (p. 48). In the extant literature, besides the seminal work into computational models of decision-making done by Busemeyer and Johnson (2004), far too little attention has been paid to decision-making in problem-solving skills for computer programming education, and in particular for CT.

Wing (2008) proffers that CT is a form of analytical thinking which shares with mathematical thinking, engineering thinking, and scientific thinking in similar ways in which we might approach the understanding and modelling of real world phenomena, in order to solve problems. She points out that many sciences and engineering disciplines also rely on simulations of mathematical models of physical processes found in nature. Mathematical modelling has also been utilised in the field of education by educators, students, and researchers (Stillman, Blum, & Biembengut, 2015). In education, mathematical modelling has been employed as a strategy for building up systems of knowledge (D'Ambrosio, 2015), for students who do not solve problems independently to share and refine mathematical models through dual modelling teaching (Kawakami, Saeki, & Matsuzaki, 2015), for exploring interconnections between real-world and application tasks (Ng & Stillman, 2015), and as visualisation tactics for solving real-world tasks (Brown, 2015).

The authors of the present paper do concede that using mathematical equations to model a phenomena and subsequently converting those mathematical equations into computer programming code is nothing new; in fact, it is a part of Computational Science (Humphreys, 2004). In computational science, computational models are usually presented as mathematical models because they can be analysed and even run as simulations using computers to better understand the characteristics of the phenomenon being studied. For example, computational

fluid dynamics (Chung, 2010) refers to the computational modelling of fluid dynamics. Computational finance (Ugur, 2008) refers to the computational modelling of financial-related systems. Computational biology (Waterman, 1995) refers to the computational modelling of biological-related systems. In the same token, computational thinking (Wing, 2006) could be construed as the computational modelling of thinking. Laudable efforts have been made by computational thinking researchers (such as Lu & Fletcher, 2009; Weintrop et al., 2016) to illustrate key concepts in computational thinking which involved the use of mathematical equations; however, these studies seem to be solely focused on which aspects of computational thinking were involved when a person encounters a mathematical formula or algorithm. Currently, there is a dearth of computational models about the "thinking" part of computational thinking in the extant literature. The present paper purports to explore this "thinking" part of computational thinking via the decision-making portions of problem-solving; first from the perspective of a human learner playing with the programmable toy mouse in an unplugged computational thinking activity, and subsequently from the perspective of a computer programmer who is programming the software version of a self-navigating mouse that can autonomously reach its objectives.

Since CT purports to enable humans to analyse problems, and to communicate the corresponding solutions using computational terms that are ultimately meant for computers to comprehend and execute, it follows that mathematical modelling might be well suited for understanding the decision-making processes involved during an unplugged CT activity. The mathematical model can also be utilised as a valuable resource should the facilitator choose to ask the learners to implement a software programme to describe the decision-making strategies that might be involved.

In the present paper, the decision-making processes during an unplugged activity, which utilises a programmable toy mouse as a simple example, are depicted via mathematical equations, to elucidate which aspects of CT might be involved in decision-making during problem solving. The analytical approach of using mathematical equations and descriptions of CT is offered in this paper as a potential form of rudimentary scaffolding between CT concepts and software programming, which might be useful to facilitators and learners of CT-related activities.

The rest of this paper is organised as follows: in the next section, the basic building blocks for the mathematical modelling of decision-making during the unplugged CT activity will be presented. Using the conjectures from these mathematical equations, the aspects of CT involved, together with some examples of Python programming code that correspond to the mathematical equations, will be presented in the discussion section. In the present paper's hypothetical scenario, the programmer needs to implement parts of the Python code to create a software-based self-navigating mouse that can avoid obstacles and autonomously move towards its goals. Finally, the direction of future research will be presented in the conclusion section.

2. Mathematical modelling of decision-making in unplugged ct activity

In the context of this research, a simple case of an unplugged CT activity which involves a programmable toy mouse (see Figure 1) will be used. In this unplugged CT activity, the learner operating the programmable mouse must evaluate the possible consequences when trying to achieve the pre-determined objectives. Since computational modelling usually involves the use of mathematical models, it follows that using mathematical models to depict decision-making in problem-solving might allow us to clearly see the aspects of computational thinking involved. A grey-based approach (Deng, 1989; Liu & Lin, 2010; Liu, Yang, & Forrest, 2016) of mathematical modelling is utilised in this paper, because it excels in modelling phenomena in situations where there might be uncertainty, scarcity of quantitative data, or incomplete information; situations which learners often find themselves in during problem-solving exercises. Further, although the entire maze could be easily observed by a human, a software-based self-navigating mouse that the programmer is trying to create – having sensors only on its front, left, right, and rear – can only detect whether there is any object in its immediate vicinity. It does not have a "bird's eye view" of the entire maze. Also, the programmer might need to implement the features to enable the software-based mouse to "decide" whether an object is an obstacle or a goal it is trying to reach. It might also need to learn from its previous attempts and "predict" the next step that it needs to take. Accordingly, we use the conceptual notion of "black" to indicate completely unknown information, "white" to indicate completely known information, and "grey" to indicate partially known and partially unknown information (Liu & Lin, 2010, p. 15). Grey-based mathematical equations are

used because they could potentially be used to address issues of uncertainty that might be countered when a programmer is trying to implement the self-navigating features of the software-based mouse in computer code.

Figure 1: Unplugged activity which uses a programmable toy mouse

The basic building blocks for modelling decision-making during problem-solving, which is adapted from the Grey Models of Decision Making, first developed by Liu & Lin (2010, p. 197) into the context of the programmable toy mouse in this unplugged CT activity, is presented as follows:

Let the four key elements of decision-making be events, countermeasures, effects, and objectives. Let the totality of all events which might be encountered by the programmable toy mouse in the unplugged activity be denoted as

$$E = \{e_1, e_2, \ldots e_n\} \quad (1)$$

where e_i represents the ith event within the set of events, where $i = 1, 2, 3, \ldots, n$, such that the event e_1 precedes e_2 and e_2 precedes e_3 and so forth.

A countermeasure (Oxford Living Dictionaries, 2017) is an actionable process or choice that could be taken (or conversely, not taken) to mitigate the effects of an event. The term "countermeasure" is used instead of "action" because of the implication that it could be taken or not taken after considerations during the decision-making process. The totality of all countermeasures is defined as the set of countermeasures, denoted as

$$C = \{c_1, c_2, \ldots, c_m\} \quad (2)$$

and c_j represents the jth countermeasure within the set of countermeasures, where $j = 1, 2, \ldots m$. Actions taken by the programmable mouse (see Figure 2) can be considered as countermeasures, with the action "forward" denoted as c_1, "rotate left" denoted as c_2, "rotate right" denoted as c_3, and "reverse" denoted as c_4. Thus, the set of countermeasures for the programmable toy mouse is denoted as

$$C = \{ c_1, c_2, c_3, c_4 \} \quad (3)$$

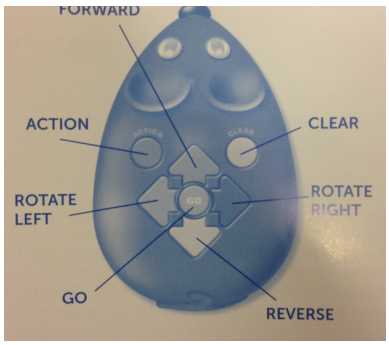

Figure 2: Buttons on the programmable toy mouse

The set of decision schemes $S = E \times C$ can be represented by the Cartesian product

$$E \text{ x } C = \{ (e_i, c_j) \mid e_i \in E, c_j \in C \} \quad (4)$$

of the set of events E, and the set of countermeasures C, where each pair of decision scheme $s_{ij} = (e_i, c_j)$, for any $e_i \in E$, $c_j \in C$. Hence, a set of decision schemes for the programmable toy mouse, which represents the totality of all the various combinations of moves it can make, is denoted as

$$S = E \times C = S_{ij} = (e_i, c_j) = \{ S_{11}, S_{12}, \ldots, S_{14}, S_{21}, \ldots, S_{24}, S_{31}, \ldots, S_{34}, \ldots \} \quad (5)$$

A set of decision schemes not only can be used to represent the totality of the various ways the programmable toy mouse can move that an individual learner has considered, if there is only one learner. Especially noteworthy is, it can also be used to represent the totality of the various ways of solving the problem that a group of learners has discussed about together, if there is more than one learner involved, such in this instance, where many learners are involved in the decision-making process to discuss how best to move the programmable toy mouse.

In this unplugged activity, the facilitator can first explain to the learners that at each step, the programmer must decide whether to make the mouse move forward, or to rotate left or right, or to move in reverse. In our mathematical model, however, each "step" that the mouse takes can be technically considered to be an event. For example (see Figure 3), each step shall be technically referred to as an event in the computational model, and in each step, there is a corresponding countermeasure, which can be Forward, or Rotate Left, or Rotate Right.

For example, at event e_1, the programmable toy mouse might not be blocked by any obstacle in front, behind, or on its sides, so the programmer can choose to deploy countermeasure c_1 to move the mouse one square forward. At event e_2, the programmable mouse might encounter an event where the goal (the cheese) is located to its left, so the programmer needs to deploy the countermeasure c_2 to rotate left, and so forth.

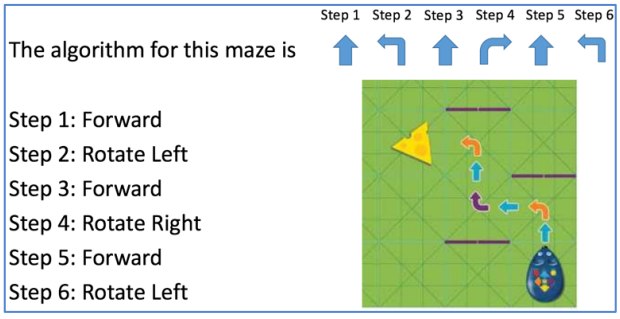

Figure 3: Example of a path utilised by a learner for the programmable toy mouse

Effect values of decision schemes

To encourage the learners to participate in this unplugged activity, the facilitator might like to consider explaining to them that there will be a competition where each team is required to discuss about the best steps for the mouse to take, before using the mouse. The rules of the competition can be presented to them in a slide (see sample slide in Figure 4).

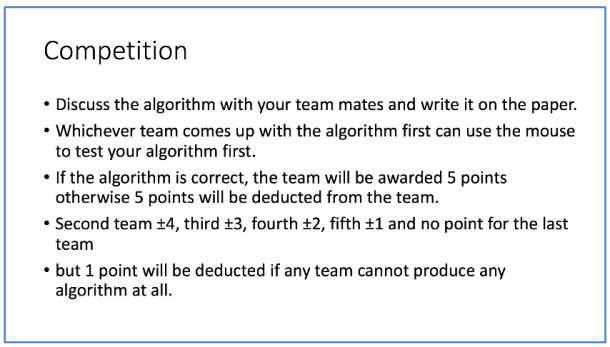

Figure 4: Rules of the competition for the teams of learners participating in the unplugged CT activity

Effect value of decision scheme

Let a set of decision schemes be denoted as

$$S = \{ S_{ij} = (e_i, c_j) \mid e_i \in E, c_j \in C \} \quad (6)$$

Further, let $u_{ij}^{(k)}$ represent the effect value of the set of decision scheme S_{ij} with respect to the objective k, where R represents the set of all real numbers. If we suppose that $u_{ij}^{(k)}: S \longmapsto R$, that is, the effect value $u_{ij}^{(k)}$ can be mapped from the set of decision schemes to a set of real numbers with respect to the objective k, it follows that a particular decision scheme s_{ij} can also be mapped to an effect value, which can be denoted as $s_{ij} \longmapsto u_{ij}^{(k)}$. For example, in the context of the points award system of the competition in this unplugged CT activity (see Figure 4), the learners in each team are encouraged to discuss among themselves to propose a solution (a particular decision scheme which is selected from a set of decision schemes) for their team and try it on the programmable toy mouse. Hence, the effect values of the teams' decision schemes can be directly manifested in the real values of the points that they score (or lose).

Equivalent countermeasures

If two different actions (also referred to as countermeasures) of the programmable toy mouse can contribute to achieving the same objective, we can denote it as follows: if the countermeasures c_j and c_h are equivalent with respect to objective k, it can be denoted as $c_j \cong c_h$. Hence, the set with equivalence class of countermeasure c_h to the event e_i with respect to objective k can be denoted as

$$C_i^{(k)} = \{ c \mid c \in C, c \cong c_h \} \quad (7)$$

Suppose the two effect values $u_{ij}^{(k)}$ and $u_{ih}^{(k)}$ are equivalent, then this effect equivalence can be denoted as $u_{ij}^{(k)} = u_{ih}^{(k)}$. For example (see Figure 5), the effect value in Decision Scheme A can be considered to be equivalent to the effect value in Decision Scheme B, because the mouse takes the same number of countermeasures to reach the cheese.

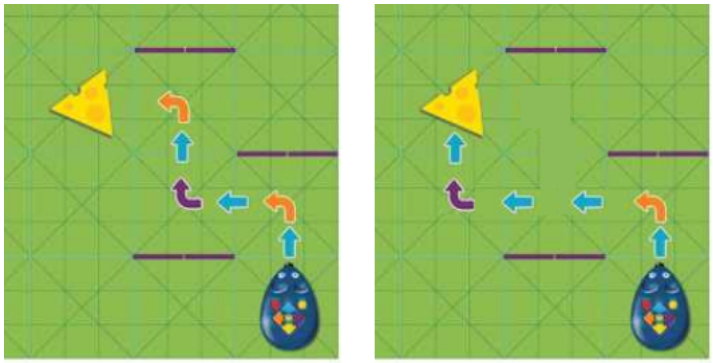

Figure 5: Example of equivalent effect values

Superiority of a countermeasure

If a countermeasure is considered to be better than another countermeasure for the programmable toy mouse, it can be denoted as follows: if the effect value $u_{ij}^{(k)}$ is greater than the effect value $u_{ih}^{(k)}$, it can be denoted as $u_{ij}^{(k)} > u_{ih}^{(k)}$. If the countermeasure c_j is superior to c_h in response to event e_i with respect to objective k, it can be denoted as $c_j \succ c_h$. Hence, it follows that the superior set of countermeasures c_h to the event e_i with respect to objective k can be denoted as

$$C_{ih}^{(k)} = \{\, c \mid c \in C, c \succ c_h \,\} \tag{8}$$

Superiority of a decision scheme

After discussion amongst the learners in their respective teams, if a set of emergent decision scheme is considered to be superior by the team members, it can be expressed as follows: if the effect value $u_{ij}^{(k)}$ must be greater than the effect value $u_{ih}^{(k)}$ to achieve the objective, it can be denoted as $u_{ij}^{(k)} > u_{ih}^{(k)}$. If the decision scheme s_{ij} is superior to s_{hl} with respect to objective k, it can be denoted as $s_{ij} \succ s_{hl}$ and hence the set of superior decision scheme can be denoted as

$$S_{hl}^{(k)} = \{\, s \mid s \in S, s \succ s_{hl} \,\} \tag{9}$$

For example (see Figure 6), if decision scheme A is contributed by Learner A, decision scheme B is contributed by Learner B, and decision scheme C is contributed by Learner C, and suppose in the group discussion, the three learners come to a decision that decision scheme A is superior because the programmable toy mouse would be able to reach the cheese if this is chosen, we can say that decision scheme A is superior compared to decision schemes B and C.

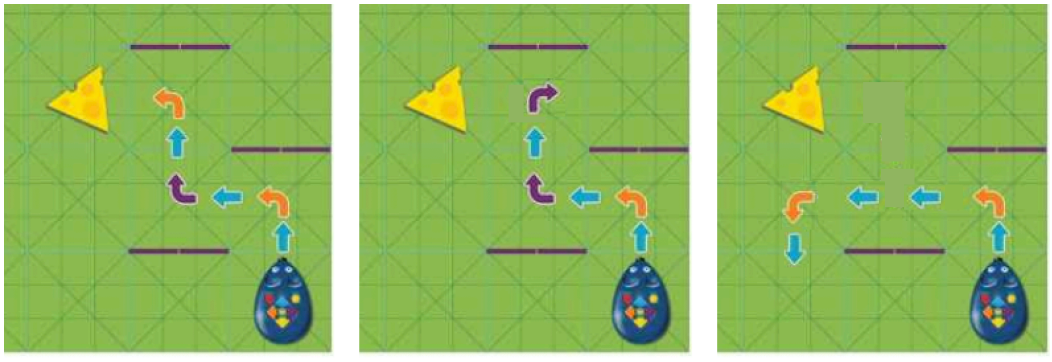

Figure 6: Example of a superior decision scheme

Threshold values of decision effects

In the competition within this unplugged activity, the teams of learners are motivated to discuss and present their perceived solution quickly, so that they can score higher points; however, if their solution of the algorithm is incorrect, they might be penalised too, so this can be expressed as follows: let $d_1^{(k)}$ be the upper threshold value (the points that the team can score if its solution is correct), and $d_2^{(k)}$ be the lower threshold value (the points that the team can score if its solution is incorrect) of the decision scheme s_{ij} with respect to the single objective k, and r be the value between the range of $d_1^{(k)}$ and $d_2^{(k)}$. It follows then that the one-dimensional grey target for objective k can be denoted as

$$S^1 = \{ r \mid d_1^{(k)} \leq r \leq d_2^{(k)} \} \tag{10}$$

and a satisfactory effect value with respective to objective k can be denoted as

$$u_{ij}^{(k)} \in [d_1^{(k)}, d_2^{(k)}] \tag{11}$$

Decision-making with multiple objectives

Suppose $u_{ij}^{(k)}$ represents the effect value of decision scheme s_{ij} with respect to a single objective k. If s_{ij} is a feasible decision scheme which can contribute to achieving the objective k, it can be denoted as $s_{ij} \in S^1$. This applies to situations which involve a single objective.

For grey targets of decision-making with multiple objectives, if there are two objectives for instance (see Figure 7), we can assume that $d_1^{(1)}$ and $d_2^{(1)}$ to be the lower and upper threshold values of the decision effects of objective 1, where $r^{(1)}$ represents the value between the range of $d_1^{(1)}$ and $d_2^{(1)}$. We can also assume $d_1^{(2)}$ and $d_2^{(2)}$ to be the lower and upper threshold values of the decision effects of objective 2 , where $r^{(2)}$ represents the value between the range of $d_1^{(2)}$ and $d_2^{(2)}$. Hence, the grey target of two-dimensional decision-making can be denoted as

$$S^2 = \{ r^{(1)}, r^{(2)} \mid d_1^{(1)} \leq r^{(1)} \leq d_2^{(1)}, d_1^{(2)} \leq r^{(2)} \leq d_2^{(2)} \} \tag{12}$$

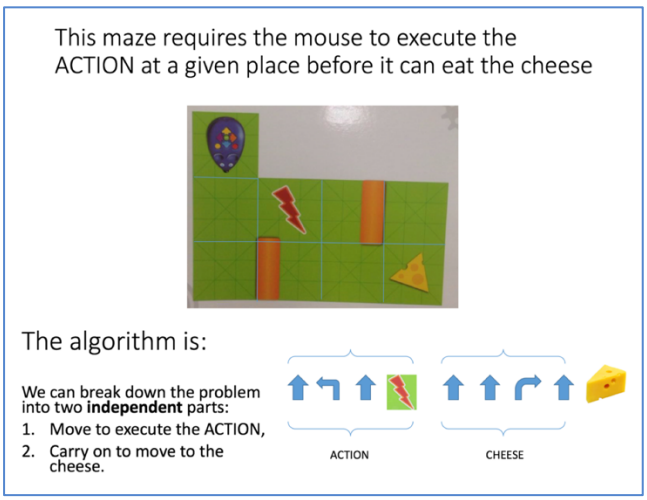

Figure 7: Multiple objectives of the programmable toy mouse

If this effect vector of s_{ij} satisfies the effect value u_{ij} such that $u_{ij} = \{u_{ij}^{(1)}, u_{ij}^{(2)}\} \in S^2$, then s_{ij} can be considered to be a superior decision scheme with respect to objectives 1 and 2. It also follows that c_j can be considered to be a superior countermeasure for event e_i with respect to objectives 1 and 2.

Suppose $d_1^{(1)}$, $d_2^{(1)}$, $d_1^{(2)}$, $d_2^{(2)}$, … where $d_1^{(s)}$ and $d_2^{(s)}$ represent the lower and upper threshold values of decision effects with respect to objectives 1, 2, … , *s*. A grey-target with a *s*-dimensional decision-making scheme can be denoted in Euclidean space as

$$S^s = \{ r^{(1)}, r^{(2)}, \dots, r^{(s)} \mid d_1^{(1)} \le r^{(1)} \le d_2^{(1)}, d_1^{(2)} \le r^{(2)} \le d_2^{(2)}, \dots, d_1^{(s)} \le r^{(s)} \le d_2^{(s)} \} \quad (13)$$

Hence, if s_{ij} is a superior decision scheme, where $u_{ij}^{(k)}$ represents the effect value of the decision scheme s_{ij} with respect to the objective *k*, and *k* = 1, 2, …, *s*, then effect vector can be denoted as

$$u_{ij} = \{u_{ij}^{(1)}, u_{ij}^{(2)}, \dots, u_{ij}^{(s)}\} \in S^s \quad (14)$$

These grey-targets of decision-making represent the locus of superior effects. In reality, however, it might be almost impossible to achieve absolute optimization of an outcome. Nevertheless, in analysis we endeavour to strive for the quasi-optimal outcome, where the decision scheme and its corresponding countermeasures are quasi-optimal; which is to say, they are the best choices among the available decision schemes and their corresponding countermeasures. As a cautionary note, however, the quasi-optimal solution presented by a team of learners might not always be the correct solution; in fact, it could even be incorrect.

Time Series: Memories of Sets of Decision Schemes

So far, the discussion has focused only on static decision schemes with a fixed moment in time. Grey-based decision-making can also focus on changes of the decision effect over time (Liu et al., 2016). Let us suppose that in a hypothetical scenario, the facilitator might wish to consider asking the learners to develop a software-based self-navigating mouse that can perform autonomous problem-solving in a series of different maze challenges, not just in one maze challenge. Instead of static decision schemes, the concept of time can now be included; as time advances forward, the changing decision effects can also be considered.

Memory plays an important role in problem solving (Reber & Kotovsky, 1997). In the software-based self-navigating mouse's multiple attempts at problem-solving, which involves the notion of time, it has to "remember" the consequential effects of its previous attempts, before another attempt is made to solve a similar problem in the future. As such, the facilitator might also consider asking the learners to implement a rudimentary type of memory into the software-based self-navigating mouse, so that it can "remember" its previous moves in the form of a time series. Suppose a set of events is represented by $E = \{ e_1, e_2, e_3, \dots, e_n \}$, a set of countermeasures is represented as $C = \{ c_1, c_2, \dots, c_m \}$, and the set of decision schemes is represented by $S = \{ s_{ij} = (e_i, c_j) \mid e_i \in E, c_j \in C \}$, then it follows that the time series of decision effect of the decision scheme s_{ij} with respect to the objective *k* can be denoted as

$$u_{ij}^{(k)} = (u_{ij}^{(k)}(1), u_{ij}^{(k)}(2), \dots, u_{ij}^{(k)}(h)) \quad (15)$$

This section has described some mathematical equations that might be used to depict the decision-making processes that might be involved in the programmable toy mouse in the unplugged CT activity. The next section will present some of the CT concepts and their corresponding Python programming code that might be useful as scaffolds for the teacher in the facilitation of a discussion with the students about CT concepts in decision-making and problem-solving, and how to implement them using computer programming code.

3. Discussion

So far, these grey-based mathematical equations have tried to depict the decision-making processes that might be involved in the unplugged CT activity to educe (meaning: to draw out) the problem-solving abilities of the learners. Sometimes, after an unplugged CT activity has been conducted, the facilitator might wish to continue with a code writing activity for the learners, if they already have experience in software programming. For example, the facilitator might consider asking the learners to write code to implement the mouse in software, in such a way so that it has autonomous decision-making abilities. Besides text syntax-based programming languages such as Python, C++, and Java, software programmes can also be developed using mathematical equations. Currently, a software which utilises symbolic computing and can accept mathematical symbols as part of its programming syntax to create simulations is Mathematica (Wolfram Research Incorporated, 2017). Should the facilitator choose to explicitly explain the CT concepts involved in the unplugged CT activity to the learners, so that by analogous association, they can programme decision-making capabilities in the software-based autonomous self-navigating mouse, the following information might be useful for the programmers, regardless of the choice of programming language to be used. In the current section, some suggested snippets of computer programming will be offered as illustrations to show how the afore-mentioned mathematical equations of decision-making can be used as scaffolds by the teacher for possible discussions of computer programming with the students. The Python programming language is used in the present paper, because it has become quite popular for learning programming in schools. It had also gained traction as a programming language for development work in machine learning, deep learning, and artificial intelligence. In the present paper, the Python code snippets are not intended to be complete solutions; they merely serve as scaffolds for the teacher to discuss about programming concepts and CT concepts with the students. This section purports to make explicit the CT concepts for implementing decision-making capabilities in the software-based self-navigating mouse.

Computational Thinking in Decision-making: Abstraction

The CT concept of abstraction of data can be applied to the events (see Table 1) which can be represented as lists or arrays in the software-based self-navigating autonomous mouse. For the purpose of keeping this example simple, the number of events is assumed to be 5. It is also assumed that in this game, the self-navigating mouse is not allowed to reverse. The values inside each event are assumed to be some data "sensed" by the software version of the self-navigating mouse, perhaps via machine-vision or via proximity sensors. Let us assume that this software-based self-navigating mouse has three sensors, one on its front, one on its left side, and one on its right side. In each event that the mouse encounters, the event e_i can be represented as an array with 3 values, each from its front, left, and right sensor respectively. For example, if there is a cheese in front of the mouse, its value would be: 2, if there is an obstacle on its left, its value would be: -1, if there is no object on its right, its value would be: 0. Hence, it can be represented as an array in Python code e_1 = [2, -1, 0]. Therefore, if it encounters 5 events, it can be represented in Python code as follows:

Table 1: Mathematical equation of an array of events and its corresponding Python code

From Equation 1: A set of events	*Example of corresponding Python code*

$E = \{e_1, e_2, \ldots e_5\}$	`# create array`
	`from numpy import array`
	
	`# Each array of Event e1 to e5 filled with`
	`# data from Front, Left & Right sensors`
	`e1 = [0,-1,0]; e2 = [0,1,0]; e3 = [0,0,0]`
	`e4 = [-1,0,0]; e5 = [0,2,0]`
	
	`# create array of a Set of Events`
	`E = [e1, e2, e3, e4, e5]`
	`a = array(E)`
	
	`# display array`
	`print(a)`

The CT concept of abstraction of data can be applied to the events (see Table 2) by representing them as an immutable tuple or alternatively as an array in the software-based self-navigating autonomous mouse. For the purpose of keeping this example simple, the number of events is assumed to be 4, where actions taken by the

self-navigating autonomous mouse can be considered to be countermeasures, with the action "forward" denoted as c_1, "rotate left" denoted as c_2, "rotate right" denoted as c_3 and "reverse" denoted as c_4. For ease of computation by the software-based self-navigating autonomous mouse, the value of c_1 is 1, the value of c_2 is 2, the value of c_3 is 3, and the value of c_4 is 4.

Table 2: Mathematical equation of an array of countermeasures and its corresponding Python code

From Equation 3:	*Example of corresponding Python code*

A set of 4 countermeasures
 	
$C = \{ c_1, c_2, c_3, c_4 \}$
	`# create array`
	`from numpy import array`
	
	`# declare variables c1 to c4`
	`# and initialize them with values`
	`# 1=Front; 2=Left; 3=Right; 4=Reverse`
	`c1 = 1; c2 = 2; c3 = 3; c4 = 4`
	
	`# create a tuple of Countermeasures`
	`# as we want the values to be immutable`
	`C = (c1, c2, c3, c4)`
	
	`# convert the tuple C into an array`
	`Countermeasures = array(C)`
	
	`# display array`
	`print(Countermeasures)`

Besides representing data as variables in lists or arrays, data can also be presented in the form of a Cartesian structure in Euclidean space during the decision-making process (see Equation 5) in the software-based self-navigating autonomous mouse. This Cartesian structure can be easily created using Python; it involves the usage of vertically stackable arrays to create a multi-dimensional matrix (see Table 3). Multi-dimensional matrices are also referred to as tensors, which could be used to store data of different data-types "sensed" by the software-based self-navigating autonomous mouse from its sensors. In real-world practical applications, tensors – which are useful for storing massive amounts of digital data from pixels of images, audio data, spatial data, and so forth – are the cornerstone of data structures in artificial intelligence programming-related software such as TensorFlow, Theano, and Keras.

Table 3: Mathematical equation of a multi-dimensional matrix and its corresponding Python code

From Equation 5:	*Example of corresponding Python code*

An E by C dimension Cartesian structure formed from the equation which represents Events and Countermeasures.
 	
$E \times C = S_{ij}$
$= \{ S_{11}, S_{12}, \dots, S_{14}, S_{21}, \dots, S_{24}, S_{31}, \dots, S_{34}, \dots \}$
	`# create array with vstack`
	`from numpy import array`
	`from numpy import vstack`
	
	`# initialize the variables`
	`e1 = [0,-1,0]; e2 = [0,1,0]; e3 = [0,0,0]`
	`e4 = [-1,0,0]; e5 = [0,2,0]`
	`c1=1; c2=2; c3=3; c4=4`
	
	`# create first array`
	`E = array([e1,e2,e3,e4,e5])`
	`print(E)`
	
	`# create second array`
	`C = array([c1,c2,c3])`
	`print(C)`
	
	`# vertical stack`
	`S = vstack((E, C))`
	`print(S)`

Computational Thinking in Decision-making: Evaluation

The CT concept of evaluation could potentially be applied in this manner: let us suppose that the software-based self-navigating mouse has autonomous decision-making capabilities to select the best quasi-optimum solution using the decision-making process to determine the superiority of a countermeasure (see Equation 8). The following is a simple contrived example of using Python code to calculate the magnitude of an array of possible countermeasures, so that a "superior" countermeasure can be determined:

Table 4: Python code to calculate the magnitude of an array to determine which direction the self-navigating mouse should take

From Equation 8: Superiority of a Countermeasure $C_{ih}^{(k)} = \{ c \mid c \in C, c \succ c_h \}$	*Example of corresponding Python code to calculate and compare the magnitudes of three arrays of events data collected by the self-navigating mouse's sensors on its front, left, and right, to determine which is the superior way to be taken by the self-navigating mouse; that is, the superior countermeasure (direction) that would allow that higher value to be manifested.*

```
from numpy import array

# the arrays of Events e1, e2, e3 are each filled
# by data from Front, Left, and Right sensors
e1 = [0,-1,0]; e2 = [0,1,0]; e3 = [0,0,0]

sum_e1 = sum(e1)
sum_e2 = sum(e2)
sum_e3 = sum(e3)

#Output the magnitude of each array
print ("Sum of e1: " +str(sum_e1))
print ("Sum of e2: " +str(sum_e2))
print ("Sum of e3: " +str(sum_e3))

if ((e1 >= e2) & (e1 >= e3)):
    print ("The mouse should go STRAIGHT")
elif ((e2 >= e1) & (e2 >= e3)):
    print ("The mouse should go LEFT")
elif ((e3 >= e1) & (e2 >= e2)):
    print ("The mouse should go RIGHT")
```

The CT concept of evaluation can also be applied by the software programmer to determine the superiority of a decision scheme (see Equation 9). Usually, in a self-navigating autonomous machine, the evaluation is not determined by the human programmer, but by the machine itself using algorithms that are useful for machine vision, image pattern recognition, path finding, and so forth. In practical terms, the programmer might simply need to "feed" the data (most probably contained in the data structure of a matrix) to the machine learning or deep learning algorithm. Comparison of numerous multi-dimensional matrices (multiple decision schemes) can then be performed by the machine learning or deep learning algorithm inside the self-navigating autonomous mouse to determine the "superiority" of a decision scheme (one single matrix) in a set of decision schemes (numerous matrices).

Computational Thinking in Decision-making: Decomposition

The CT concept of decomposition could be applied in decision-making with multiple objectives (see Equation 12) in the software-based self-navigating mouse. For instance, if there are two objectives, and if the self-navigating mouse is required to reach the Action symbol, as well as the cheese symbol on the board, it would be required to break down the steps to be taken to achieve those two objectives.

Computational Thinking in Decision-making: Algorithmic Thinking

In addition to the CT concept of decomposition in decision-making for multiple objectives, the CT concept of algorithmic thinking, in conjunction with the concept of utilising the threshold values of decision effects (see Equation 13) could be applied to the software-based self-navigating mouse so that it can "think"

about using similar methods for reaching multiple objectives, even though the objectives may look different. For example, the same method that the software-based self-navigating autonomous mouse can use to reach the Action symbol, that is, by comparing countermeasures to determine which one is superior (for instance, one path which uses fewer steps), can also be applied to reach the cheese. In terms of practical application, the software programmer might wish to consider using a Time Series (see Equation 15) so that the events, countermeasures, decision schemes, and decision effects considered and taken (or considered but not taken) by the software-based self-navigating mouse can be "stored" for analysis to determine which next step to take (forward, left, or right).

Table 5: Python code to utilise a Time Series and make a one-step prediction

From Equation 15: Data stored in a Time Series $u_{ij}^{(k)} = (u_{ij}^{(k)}(1), u_{ij}^{(k)}(2), \dots, u_{ij}^{(k)}(h))$	*Example of corresponding Python code which utilises the concept of Time Series to apply the CT concept of generalisation to analyse the pattern in the data, and subsequently make a one-step prediction*

```
# load Auto-regression model from file and make a one-step prediction
from statsmodels.tsa.ar_model import ARResults
import numpy

# load model
model = ARResults.load('model.pkl')
data = numpy.load('data.npy')
last_observation = numpy.load('observation.npy')

# make prediction
predictions = model.predict(start=len(data), end=len(data))

# transform prediction
predicted_value = predictions[0] + last_observation[0]
print('Prediction: %f' % predicted_value)
```

Computational Thinking in Decision-making: Generalisation

Finally, the CT concept of generalisation in decision-making might be implemented in the software-based self-navigating mouse in a manner that can combine the sets of events, countermeasures, decision schemes, and objective effects into a decision-making algorithm, so that they can be utilised in problem solving, for example, to autonomously predict the best route to take in new mazes. Practically, one of the ways might be for the programmer to consider implementing a LSTM (Long Short-Term Memory) recurrent neural network, which is a cornerstone of Machine Learning/Deep Learning for predicting new sequences (and in this context: paths) based on older data. The programmer may wish to consider implementing code to develop persistence (a form of memory) and perform analysis on the data from events, countermeasures, decision schemes, and decision effects, so that the self-navigating mouse can develop its own algorithmic thinking. More information about coding LSTM recurrent neural networks in Python can be perused at Brownlee's (2016) website.

4. Conclusion and future research

Grey-based mathematical equations have been utilised in the present paper to depict what might be involved in decision making during an unplugged CT activity. Mathematical modelling of decision-making might contribute to addressing a gap in the extant literature of CT research that has insofar not been studied much. An analytical approach using mathematical equations and descriptions of CT has been offered in this paper as a potential form of rudimentary scaffolding, which might be useful to facilitators and learners of CT-related activities. The mathematical equations of the decision-making processes posited in this theoretical manuscript may serve as a base for programmers, regardless of the programming language they prefer, should the facilitator wish to ask the learners to embark on a software programming activity that is closely associated to the unplugged CT activity.

Indeed, teachers/instructors might not need the mathematical equations in the present paper to teach an activity such as navigating in the maze. They might, however, find them to be useful as scaffoldings if software programming by the learners is involved after the conclusion of the unplugged CT activity. The hypothesis is that, if the teachers are exposed to a math model, they can be made aware of what the decision options are, and how to interpret the actions and results provided by students. Further, they might be more aware of the

ramifications of the unplugged activity through its representation as a mathematical model. We hope future research can explore this hypothesis. Better still, if some instructors can create the model or fragments of the model, they can become even more conversant of the content knowledge to be taught and can build on the model to do the programming of the algorithm.

The existence of the problem-solving conceptual framework that has come to be referred to as computational thinking cannot be in doubt; however, what that structure is, might be another matter that is worthy of further research and exploration. As researchers seek to understand more about the various aspects of computing education, the utilisation of mathematical modelling might play a significant role in CT by, for example, describing it in more formal terms via mathematical equations to uncover aspects of CT that might be useful for programmers; should the need arise to implement them systematically in software code.

Acknowledgements
Support for this paper was provided by the project grant for: Researching and developing pedagogies using unplugged and computational thinking approaches for teaching computing in the schools (Project Number: OER 04/16 LCK)**.** Many thanks to Peter Seow, Longkai Wu, and Liu Liu for their help in designing and conducting the programmable toy mouse unplugged activity in a classroom.

References

Barr, D., Harrison, J., & Conery, L. (2011). Computational Thinking: A Digital Age Skill for Everyone. *Learning and Leading with Technology*, *38*(6), 20–23.

Bell, T., Alexander, J., Freeman, I., & Grimley, M. (2009). Computer Science Unplugged: School Students Doing Real Computing Without Computers. *Journal of Applied Computing and Information Technology*, *13*(1), 20–29.

Boticki, I., Barisic, A., Martin, S., & Drljevic, N. (2013). Teaching and learning computer science sorting algorithms with mobile devices: A case study. *Computer Applications in Engineering Education*, *21*, 41–50.

Brown, J. P. (2015). *Visualisation Tactics for Solving Real World Tasks*. (G. A. Stillman, W. Blum, & M. S. Biembengut, Eds.), *Mathematical Modelling in Education Research and Practice: Cultural, Social and Cognitive Influences*.

Brownlee, J. (2016). Making Predictions with Sequences. Retrieved February 6, 2018, from https://machinelearningmastery.com/sequence-prediction/

Bundy, A. (2007). Computational Thinking is Pervasive. *Journal of Scientific and Practical Computing*, *1*(2), 67–69.

Busemeyer, J. R., & Johnson, J. G. (2004). Computational models of decision making. In *Blackwell handbook of judgment and decision making* (pp. 133–154).

Cetin, I., & Dubinsky, E. (2017). Reflective abstraction in computational thinking. *Journal of Mathematical Behavior*, *47*(November 2016), 70–80. https://doi.org/10.1016/j.jmathb.2017.06.004

Chung, T. J. (2010). *Computational fluid dynamics*. Cambridge university press.

Cortina, T. J. (2015). Broadening Participation: Reaching a broader population of students through "unplugged" activities. *Communications of the ACM*, *58*(3), 25–27. https://doi.org/10.1145/2723671

Curzon, P., McOwan, P. W. P., Plant, N., & Meagher, L. R. (2014). Introducing teachers to computational thinking using unplugged storytelling. *Proceedings of the 9th Workshop in Primary and Secondary Computing Education*, 89–92. https://doi.org/10.1145/2670757.2670767

D'Ambrosio, U. (2015). Mathematical Modelling as a Strategy for Building-Up Systems of Knowledge in Different Cultural Environments. In G. A. Stillman, W. Blum, & M. S. Biembengut (Eds.), *Mathematical Modelling in Education Research and Practice: Cultural, Social and Cognitive Influences* (pp. 35–44).

Deng, J. (1989). Introduction to Grey System Theory. *The Journal of Grey System*, *1*, 1–24.

du Boulay, B., O'Shea, T., & Monk, J. (1981). The black box inside the glass box: presenting computing concepts to novices. *International Journal of Man-Machine Studies*, *14*, 237–249.

Feaster, Y., Segars, L., Wahba, S., & Hallstrom, J. (2011). Teaching CS unplugged in the high school (with limited success). *ITiCSE*, 248–252. https://doi.org/10.1145/1999747.1999817

Gouws, L. A., Bradshaw, K., & Wentworth, P. (2013). Computational thinking in educational activities. *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education - ITiCSE '13*, 10. https://doi.org/10.1145/2462476.2466518

Griffin, J. M. (2016). Learning by Taking Apart: Deconstructing Code by Reading, Tracing, and Debugging. *Proceedings of the 17th Annual Conference on Information Technology Education (SIGITE '16)*, 148–153. https://doi.org/10.1145/2978192.2978231

Grover, S. (2015). "Systems of Assessments" for Deeper Learning of Computational Thinking in K-12. *Annual Meeting of the American Educational Research Association*, (650).

Grover, S., Pea, R., & Cooper, S. (2015). Designing for deeper learning in a blended computer science course for middle school students. *Computer Science Education*, *25*(2), 199–237. https://doi.org/10.1080/08993408.2015.1033142

Hu, C. (2011). Computational thinking. *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education, ITiCSE '11*, 223–227. https://doi.org/10.1145/1999747.1999811

Humphreys, P. (2004). *Extending ourselves: Computational science, empiricism, and scientific method*. Oxford University Press.

Israel, M., Pearson, J. N., Tapia, T., Wherfel, Q. M., & Reese, G. (2015). Supporting all learners in school-wide computational thinking: A cross-case qualitative analysis. *Computers and Education*, *82*, 263–279. https://doi.org/10.1016/j.compedu.2014.11.022

Kawakami, T., Saeki, A., & Matsuzaki, A. (2015). *How Do Students Share and Refine Models Through Dual Modelling Teaching: The Case of Students Who Do Not Solve Independently*. (G. A. Stillman, W. Blum, & M. S. Biembengut, Eds.), *Mathematical Modelling in Education Research and Practice: Cultural, Social and Cognitive Influences*.

Kordaki, M., Miatidis, M., & Kapsampelis, G. (2008). A computer environment for the learning of sorting algorithms: Design and pilot evaluation. *Computers & Education*, *51*, 708–723.

Liu, S., & Lin, Y. (2010). *Grey Systems: Theory and Applications*. Berlin: Springer-Verlag.

Liu, S., Yang, Y., & Forrest, J. (2016). *Grey Data Analysis*. Singapore: Springer-Verlag.

Lu, J. J., & Fletcher, G. H. L. (2009). Thinking About Computational Thinking. In *SIGCSE '09 Proceedings of the 40th ACM technical symposium on Computer science education* (pp. 260–264). Chattanooga, TN, USA. https://doi.org/10.1145/1539024.1508959

McGill, T. J., & Volet, S. E. (1997). A Conceptual Framework for Analyzing Students' Knowledge of Programming. *Journal of Research on Computing in Education*, *6504*(December), 37–41. https://doi.org/10.1080/08886504.1997.10782199

Metcalfe, J., & Wiebe, D. (1987). Intuition in insight and noninsight problem solving. *Memory & Cognition*, *15*(3), 238–246. https://doi.org/10.3758/BF03197722

Monteiro, I. T., Salgado, L. C. de C., Mota, M. P., Sampaio, A. L., & de Souza, C. S. (2016). Signifying software engineering to computational thinking learners with AgentSheets and PoliFacets. *Journal of Visual Languages and Computing*, (February 2016), 1–21. https://doi.org/10.1016/j.jvlc.2017.01.005

Ng, K. E. D., & Stillman, G. A. (2015). *Exploring Interconnections Between Real-World and Application Tasks: Case Study from Singapore*. (G. A. Stillman, W. Blum, & M. S. Biembengut, Eds.), *Mathematical Modelling in Education Research and Practice: Cultural, Social and Cognitive Influences*.

Oxford Living Dictionaries. (2017). Countermeasure. Retrieved December 4, 2017, from https://en.oxforddictionaries.com/definition/countermeasure

Pretz, J. E. (2008). Intuition versus analysis: Strategy and experience in complex everyday problem solving. *Memory & Cognition*, *36*(3), 554–566. https://doi.org/10.3758/MC.36.3.554

Reber, P., & Kotovsky, K. (1997). Implicit learning in problem solving: The role of working memory capacity. *Journal of Experimental Psychology: General*, *126*(2), 178.

Rodriguez, B. R. (2015). *Assessing Computational Thinking in Computer Science Unplugged Activities*. Colorado School of Mines. https://doi.org/10.1017/CBO9781107415324.004

Selby, C. (2013). Computational Thinking : The Developing Definition. *ITiCSE Conference 2013*, 5–8.

Selby, C. (2015). Relationships: Computational Thinking, Pedagogy of Programming, and Bloom's Taxonomy. *Proceedings of the Workshop in Primary and Secondary Computing Education*, 80–87. https://doi.org/10.1145/2818314.2818315

Somers, J. (2017). The Coming Software Apocalypse A small group of programmers wants to change how we code—before catastrophe strikes. Retrieved October 3, 2017, from https://www.theatlantic.com/technology/archive/2017/09/saving-the-world-from-code/540393/

Stillman, G. A., Blum, W., & Biembengut, M. S. (Eds.). (2015). *Mathematical Modelling in Education Research and Practice: Cultural, Social and Cognitive Influences*.

Taub, R., Armoni, M., & Ben-Ari, M. (2012). CS Unplugged and Middle-School Students' Views, Attitudes, and Intentions Regarding CS. *ACM Transactions on Computing Education*, *12*(2), 1–29. https://doi.org/10.1145/2160547.2160551

Taub, R., Ben-Ari, M., & Armoni, M. (2009). The effect of CS unplugged on middle-school students' views of CS. *ACM SIGCSE Bulletin*, *41*(3), 99. https://doi.org/10.1145/1595496.1562912

Taylor, D. W. (2013). Decision making and problem solving. In *Handbook of organizations* (pp. 48–86).

Thies, R., & Vahrenhold, J. (2013). On plugging "unplugged" into CS classes. *Proceeding of the 44th ACM Technical Symposium on Computer Science Education - SIGCSE '13*, 365–370. https://doi.org/10.1145/2445196.2445303

Thies, R., & Vahrenhold, J. B. (2012). Reflections on outreach programs in CS classes: Learning objectives for "unplugged" activities. *SIGCSE12 Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, 487–492. https://doi.org/10.1145/2157136.2157281

Ugur, Ö. (2008). *An introduction to computational finance*. World Scientific Books.

Waterman, M. S. (1995). *Introduction to computational biology: maps, sequences and genomes*. CRC Press.

Weintrop, D., Beheshti, E., Horn, M., Orton, K., Jona, K., Trouille, L., & Wilensky, U. (2016). Defining Computational Thinking for Mathematics and Science Classrooms. *Journal of Science Education and Technology*, *25*(1), 127–147. https://doi.org/10.1007/s10956-015-9581-5

Wing, J. (2008). Computational thinking and thinking about computing. *Philosophical Transactions of the Royal Society of London: Mathematical, Physical and Engineering Sciences*, (July), 3717–3725. https://doi.org/10.1109/IPDPS.2008.4536091

Wing, J. M. (2006). Computational Thinking. *Communications of the ACM*, *49*(3), 33–35.

Wolfram Research Incorporated. (2017). Mathematica, Version 11.2, (2017). Champaign, IL.

Zagami, J. (2012). *Seeing is understanding: The effect of visualisation in understanding programming concepts*. Lulu.com.

Zsambok, C. E. (2014). *Naturalistic decision making*. Chicago: Psychology Press.