Towards the Automatic Recognition of Computational Thinking for Adaptive Visual Language Learning

Kyu Han Koh, Ashok Basawapatna, Vicki Bennett, Alexander Repenning
University of Colorado at Boulder
{Kyu.Koh, Ashok.Basawapatna, Vicki.Bennett, Alexander.Repenning}@colorado.edu

Abstract

Visual programming languages can be used to make computer science more accessible to a broad range of students. The evaluative focus of current research in the area of visual languages for educational purposes primarily aims to better understand motivational benefits as compared to traditional programming languages. Often these visual languages claim to teach students computational thinking concepts; however, although the evaluations show that students may exhibit more enthusiasm, it is not always clear what computational thinking concepts students have actually learned. In this paper we attempt to develop a visual semantic evaluation tool for student-created games and simulations that goes towards depicting the computational thinking concepts implemented by the students. Through semantically analyzing a given student's created projects over time, this visual evaluation tool, called the Computational Thinking Pattern (CTP) graph, can possibly indicate the existence of computational thinking transfer from games to science simulations.

1. Introduction

There is growing evidence that visual programming languages can be effective in teaching children how to program. The ability to at least reduce syntactic problems – for instance not being a mere semicolon away from total programming disaster – combined with the do-it-yourself spirit of the Web 2.0 generation have resulted in a renaissance of visual programming. The anticipated educational benefit of visual programming is mostly of a motivational nature. Visual programming [1,2], drag and drop style programming, or more generally end-user development [3] makes programming simpler.

The motivational level of visual programming supported computer science learning, including game design, can be amazingly high; but the question of what it is that students actually learn emerges. For instance, in the case of the AgentSheets authoring tool [16,18], the large number of students instructed in schools is due to the fact that the Scalable Game Design [4] curriculum has been embedded into existing required courses that formerly consisted of teaching applications, such as word processing and keyboarding. In these classes most students are now required to participate in game design modules. Surprisingly, even with the forced exposure, the motivational numbers are extremely high. For instance 78% of the girls and 68% of the boys stated that they would like to continue taking a game design course [5]. These kinds of programs exploring motivational dimensions are well supported through funding organizations, including the National Science Foundation. Unfortunately, the educational dimensions of these investigations are much weaker, making what students actually learn about computer science through the use of visual programming languages less clear.

Computational Thinking [6,7] has become the buzzword in educational research. Early attempts to define computational thinking, such as the panel on computational thinking at the National Academies of Sciences, [8] suggests that consensus is not yet imminent. Many school districts with whom we are working have given us feedback that they have heard the term "computational thinking," but do not understand what it is. Even though educators may not share a definition of "computational thinking," their expectations of its effects overlap. Educators would like to walk into a game design-based class in computer science education and ask a student, "Now that you can make 'Space Invaders', can you also program a science simulation?" This is a bold question revealing a highly ambitious goal of transfer, including a testable condition. If all that students learn is to use simplified visual programming languages to make nothing but games, without the ability to apply more general skills to applications such as computational simulations, then computational thinking did not occur.

Whatever computational thinking may be, to educators, it should allow students to apply a computational skill set to a diverse set of problems; this would be a true test.

To assess computational thinking we need to be able to recognize various kinds of computational thinking skills. Many existing attempts have been limited in nature to skill investigations mostly at the syntactic level. Some investigations explore the specific benefits of visual programming languages in comparison to textual languages [9]. At the syntactic level, these comparisons are extremely difficult because visual and textual languages may not match up very well. The few studies that do find comparable languages, e.g., Logo and Scratch, [10] reveal mixed benefits and are unable to find simple causal connections between visual languages and learned skills. This is not a surprise to the visual language research community, which has extensively discussed these kinds of challenges. Perhaps more disappointing is the intrinsic limitation of syntax level analysis. The discussion has been limited to form (syntax), rather than the meaning (semantics) of a program. Largely this is not because people are uninterested in semantics, but because semantics are intrinsically hard to infer from existing programs.

The field of semantics program analysis is extensive including reverse engineering and design intent recovery [11]. Even with a limited scope, for instance, the ability of a compiler to detect code that could run in parallel instead of sequential [12], semantic discovery can be difficult. In educational settings, educators tend to make the best from the situation by working with rubrics that provide a checklist for required behavior of a game. For instance, the Stanford School of Education developed a rubric based approach to grade middle school students building games with AgentSheets [13]. These checklists work well for teachers. Teachers run a student's game while checking off the behavior that the game is supposed to exhibit. For instance, in Frogger, the user should be able to control the position of the frog using the cursor keys, an element of an evaluation rubric. Unfortunately, rubric based approaches do not work well, if at all, for open ended programming assignments, in which students can build a nearly unlimited range of game or simulation designs.

In this paper we describe an early version of an automatic approach to computational thinking pattern recognition based on a semantic analysis called "Program Behavior Similarity (PBS)". This approach can, with some probability, recognize the presence of a semantic level pattern without program execution. This paper will introduce the notion of computational thinking patterns, describe a way to recognize these patterns, and show a number of examples ranging from middle school to graduate school level games and simulations. The aim of this paper is to develop an early framework to recognize computational thinking skills in a way that is application domain independent. This idea is important for visual language learning as it pertains to human centric computing. For example, the ability to automatically detect computational thinking patterns enables the adaptation of visual language curricula to individual students. Ideally, we would demonstrate that these kinds of patterns not only exist for many different applications, but also, that we can teach these patterns in a way that could detect transfer, and consequently start to shed some light on the question, "Now that you can program 'Space Invaders', can you program a science simulation?"

2. Approaches to Semantic Program Recognition

Visual language based authoring tools have been created and used in a number of application domains, specifically in game design [13], computational science [14] and robotics [15]. The computational thinking pattern spiral (Figure 1) [5] depicts these application domains and the skills transfer process over time. The Computational Thinking (CT) Spiral embodies:

- A collection of computational thinking patterns specifying common object interaction that can be found in a number of domains including game design, computational science and robotics.
- A spiral pathway suggesting an iterative approach to introduce and connect these concepts. For instance, random movement in game design is conceptually similar to Brownian movement in physics.
- Ordered from simple computational thinking patterns such as the collision of objects to highly advanced ones such as Maslow's hierarchy of needs. These concepts build upon each other within the spiral.
- Implies increased connectivity among the three computer science areas of robotics, computational science and game design.

Ideally, learning would begin with the simplest concepts and progress to the more complex. The CT Spiral exemplifies the process of how learning transfer can be detected through the use of computational thinking patterns as they build and combine with each other. In our experience, over the course of the semester in various game design classes, we have informally noticed an increase in computational thinking complexity and possible evidence of transfer among students [5]. Overall, just as the spiral depiction offers an insightful view into the computational thinking transfer process, so does a student's use of increasingly complex pattern combinations.

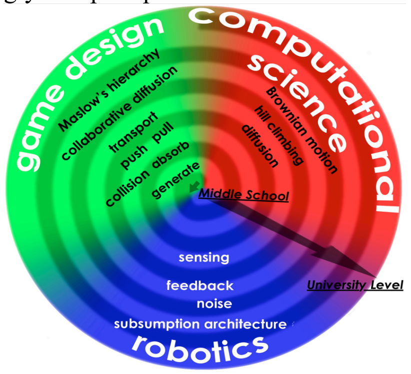

Figure 1: Computational Thinking Pattern Spiral exemplifies computational thinking concepts from the simple to the complex [4]

The aim of this research is to develop an adaptive mechanism able to support the teaching of visual programming. The ability to recognize computational thinking patterns is a first step towards the assessment of computational skills in a way necessary to solve challenges in applications such as game design, computational science and robotics. Employing the notion of flow [16] a human centric approach would be able to adapt to specific needs of users by balancing design challenges with the necessary skills. This would allow novices to gradually pick up programming skills starting with simple application such as basic game designs and then, guided by the system, to move towards more sophisticated games and computational science simulations.

2.1. Program Behavior Similarity

As part of the Scalable Game Design project we have collected thousands of games built by middle school students [5]. We have also collected more sophisticated games and science simulations from teaching a game design course to university undergraduates and graduates. To assess learning among such diverse projects necessitates the need to quantify skills beyond a collection of motivational data. For example, being able to automatically recognize the computational thinking skills outlined in the CTP spiral (Figure 1), would enable students and teachers to receive direct feedback as to a given student's skill level and learned concepts.

In trying to formulate an evaluative tool that would be able to distinguish increased pattern combination complexity within student created games, both syntactic and semantic formats were considered. Syntactic evaluation usually focuses on the form or structure of computation. This type of evaluation does not appear to be discriminating enough to show potential increases in computational thinking skills, which would lead to transfer. A semantic evaluative tool that could compare simpler games and projects, produced during the beginning of the semester, with games and projects, created near the end of the semester, could possibly reveal learning transfer. As students start creating science simulations based on natural phenomena that employ her/his knowledge, gained from the game design curriculum, this evaluative tool could point to the existence of transfer within these games.

Related research, such as Lewis [10], compares two visual language programs (Scratch and Logo) for student game authoring. For this example, much of this research protocol centers on motivational questions to determine if there was any difference between each of the two software programs. Only knowledge of individual programming pieces, apart from the larger context of the program, were used to compare the two programs. Studies such as this one that evaluate on a purely syntactic level might show knowledge of individual concepts, but do not evaluate the student's ability to use that knowledge within multiple contexts. Syntactic evaluations are not useful for detecting high-level computational thinking knowledge or learning transfer. A semantic evaluation tool could be very useful for more accurately indicating the transfer of learned knowledge in this respect.

One way to compare and profile code on more of a semantic level, as an alternative to just counting program primitives such as loops, is to look for higher level patterns that could be indicative for the meaning of a program. A similar approach called Latent Semantic Analysis (LSA) [17] is used to find semantic information in natural languages by comparing text. Computer languages, including visual languages, can be subjected to the same idea. Just like natural languages, computer languages are based on the notion of statements consisting of grammatical structure. On one hand, computer languages should be simpler to deal with as their syntactic rules tend to be less irregular. In LSA stemming is a fundamental problem, which is not relevant to computer language because verb conjugations are non-existent. Functions and primitives of computer languages are comparatively simple. On the other hand, the approach described here shares some of the documented shortcomings of LSA such as the Bag-of-Words problem preventing the recognition of semantics based on word order.

Our first attempt at a semantic type evaluation involved using the rules within different AgentSheets games to develop a profile of these games. AgentSheets programs consist of user created "agents," which are the game characters. For example, in the game Frogger, a user creates a different agent for the frog, truck, street etc. Every agent in AgentSheets consists of depictions that specify how the agent looks, and behaviors that are rules dictating how the agent acts in a given situation. All behaviors in AgentSheets are implemented using "If/Then" conditional statements [18]. AgentSheets enables the use of 16 different conditions and 23 different actions, in combination, to create behaviors for any given agent. In Frogger, for example, to make the frog move in four different directions involves four "key-pressed" conditions associated with four "move" actions, one for each direction. Therefore, to make the frog move up, a student would program "If the keyboard up key is hit, then the frog moves up." The rules to make the frog move every direction are depicted below in Figure 2.

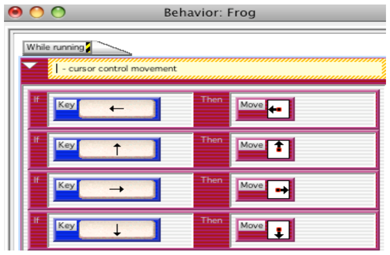

Figure 2: The cursor controlled move conditions and action behaviors for the frog.

With the 23 conditions and 16 actions, it is possible to represent each game as a vector of length 39, wherein each element of the vector represents how many of each individual conditions and actions are used to implement a given game. Using these vectors, any game created in AgentSheets can be compared to any other game through a high dimensional cosine calculation for similarity as depicted in Equation 1. The cosine is zero if the unit vectors are orthogonal and one if they are the same direction.

$$PBS(u,v) = \frac{\sum_{i=1}^{i=n} u_i v_i}{\sqrt{\sum_{i=1}^{n} u_i^2} \cdot \sqrt{\sum_{i=1}^{n} v_i^2}}$$

Equation 1: The Program Behavior Similarity (PBS) is obtained by finding the angle in-between of two n-dimensional vectors, u and v.

Equation 1, allows for a simple comparison of every game based on the rules used. The high dimensional cosine similarity comparison of games is robust to two games having the same proportion of rules, but having these rules in differing numbers. In such cases, a syntactic analysis would categorize the games as different. This makes the use of the high dimensional cosine less of a purely syntactic evaluation and closer to a semantic-type evaluation. Therefore, if two games use the same exact rule set or rules in the same exact proportion to one another, the similarity score between the two games will be one. On the other hand, if completely different rules are used, the similarity between the two games will be zero.

The following is an example of two games with a high similarity score.

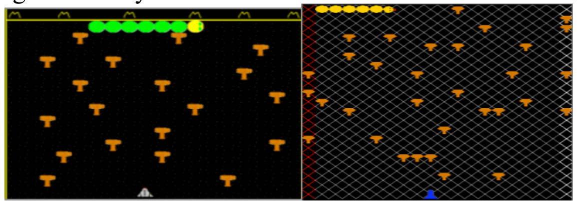

Figure 3: Two similar Centipede Games with a similarity score of 0.89

The two games in Figure 3 are implementations of the classic arcade game Centipede. These games look similar, and, in general, two normally programmed Centipede games should have a very close similarity score[1]. This is in-fact the case, as the two above games have a similarity score of 0.89. The power of similarity score analysis gives us an initial metric in order to compare two games. However, this comparison is still low level, and furthermore, does not give us a meaningful explanation as to what computational thinking patterns might be used in a given game implementation or give insight into the existence of transfer.

Although low-level, calculating program behavior similarity, as shown in the previous section, is one approach for semantic evaluation. However, the differences between programmers' problem solving approaches and programming styles may result in an inaccurate semantic analysis. For example, the two Centipede games below (Figure 4), look similar, play similar, but have a low similarity score of 0.43. So why do these two similar games have such a low similarity score? The two games have differences both in their problem solving approaches and programming styles (Table 1), giving an imprecise semantic game analysis. Accidently, these two Centipede games have almost same number of rules, but their agents' rules are programmed in a different manner. Both of them have a 'mushroom' agent, ie: a section of the centipede, when hit by a laser, turns into a mushroom. Centipede A uses two rules to implement the 'mushroom' agent while Centipede B uses six rules (Figure 5 and Figure 6). This differences in implementation produces a low PBS score between the two games despite both games having almost the same number of rules and similar gameplay.

[1] 'Normally programmed' refers to a minimum of unnecessary rules.

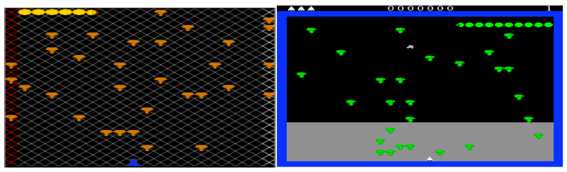

Figure 4: Two Centipede Games with a low similarity score of 0.43 (Centipede A: Left, Centipede B: Right)

Table 1: Structure of Centipede A and B

	Centipede A	Centipede B
Number of Agent Classes	8	19
Number of Depictions	13	35
Number of Methods	26	38
Number of Rules	107	129

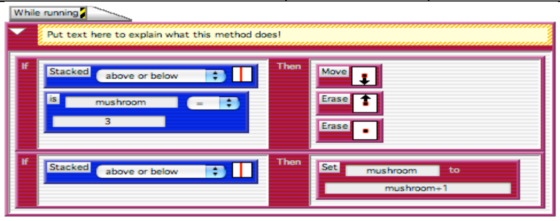

Figure 5: Rules and Conditions of a Mushroom Agent in Centipede A

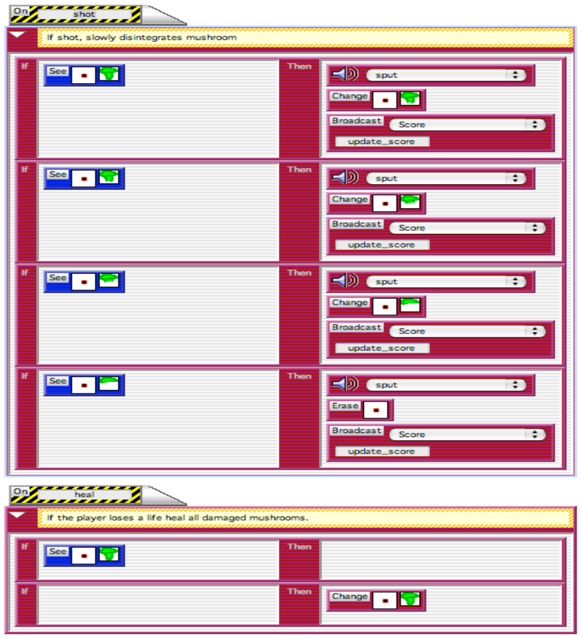

Figure 6: Rules and Conditions of a Mushroom Agent in Centipede B

Based on this low similarity score these games would seem unrelated, which would be misleading. Thus, a higher-level approach to semantic evaluation that could detect the specific computational thinking patterns that constitute a given game would be more desirable. In order to fill this void, we developed the Computational Thinking Pattern (CTP) Graph as an attempt at a higher-level approach.

2.2. Computational Thinking Pattern Graph

The CTP graph illustrates the amounts and kinds of computational thinking patterns implemented in a given game. Figure 7 depicts two CTP graphs that identify the nine most popular computational thinking patterns providing tangible semantic game information that cannot be found through more syntactic means. These nine CT patterns are the result of a survey of game collections and science simulations that have been developed over a number of years. The CT patterns are lined up in a clockwise direction in order of implementation difficulty. In order to compare the CTP graphs, the positioning of the computational thinking patterns remain in the same order in any given CTP graph. The internal rationale of the CTP graph is an extension of the Program Behavior Similarity score. The CTP graph is drawn by calculating the PBS score between a given AgentSheets project and nine representative canonical patterns. Each canonical pattern form represents one computational thinking pattern such as ‘cursor control’, ‘generation’, etc. These canonical patterns can be found on the Scalable Game Design Arcade (SGDA).

On the CTP graph, the score for each vector, multiplied by 10, depicts the PBS score between a given game and each canonical pattern (computational thinking pattern). Also, the score for each vector represents how much a certain computational thinking pattern is employed in a given game. So, if a game has features, which are not in the CTP graph structure, the CTP graph will not analyze those features. Therefore, the remaining computational thinking patterns would be a smaller portion of the graph. Consequently, undetected features will lower the PBS score of those computational thinking patterns. As a result, the CTP graph for that game will be smaller than a game that employs only the computational thinking patterns within the CTP graph structure.

Though the top and bottom images in Figure 7 look the same size, they are scaled differently. This is due to the fact that the greatest computational thinking pattern value in the top image is 8 whereas the greatest computational thinking pattern value in the bottom image is 4. This difference in scaling is more apparent in Figure 8 wherein the two CTP graphs are overlapped.

When comparing these two Centipede games above, using the CTP graph, the graph reveals more accurate analysis (Figures 7 and 8). Though these two games may use different implementations, they employ the same computational thinking patterns because they are the same game. Consequently, the CTP graph gives us the true picture of the underlying semantic meaning of these games.

The CTP graph can help users, such as teachers or students, more effectively interpret and evaluate games. Furthermore, this CTP graph is automatically generated when a student submits her/his game to the SGDA giving instant feedback [7]. The authors of SGDA have used the system to collect around 2500 AgentSheets projects including arcade games such as Frogger, Sokoban, Centipede etc. and various science simulations from the participants of the Scalable Game Design project [4]. Students can get instant semantic evaluation feedback right after he/she submits his/her project to SGDA through the CTP graph and students have the ability to compare their games to the other AgentSheets project on the SGDA.

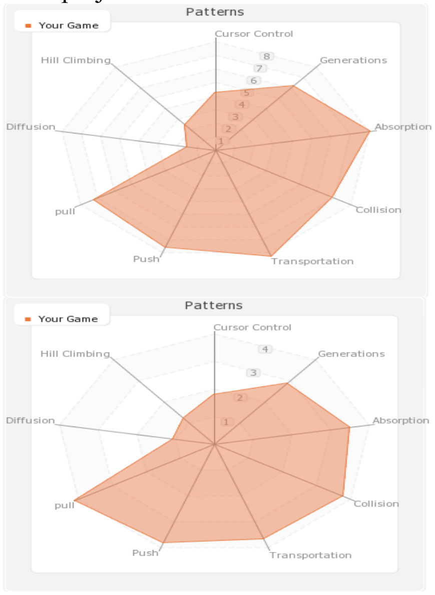

Figure 7: CTP Graph of Centipede A (top) and Centipede B (bottom)

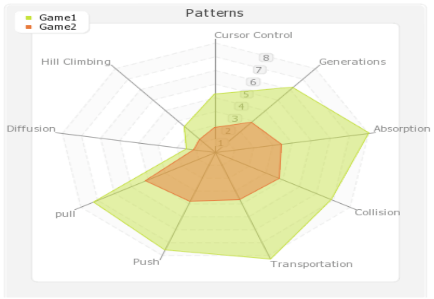

Figure 8: CTP graphs from Centipede A and B

3. Transfer

Bransford et al [19] describe knowledge transfer as the most common method for human beings to learn the necessary components of life. Transfer is defined as the ability to extend or use what has been learned in one context into a new context or to solve a new problem. Using this definition, all learning can be considered a form of transfer. Knowledge transfer can be aided by using multiple contexts (the more diverse settings, the better) to demonstrate new concepts to students. The new knowledge can then be retained by the student in a more abstract form. When new types of future situations occur, this knowledge can then be accessed by the student. Students are not normally able to transfer purely conceptual information to real world situations without help. Linking any concept to a single setting or context can also cause difficulty with transferring knowledge to new situations. So, although transfer is our preferred mode of learning and retaining new information, transfer cannot be assumed in any given context. Previous knowledge that students build upon can also enhance or deter the effort to assimilate new information. Consequently, the ability to detect possible knowledge transfer could benefit researchers in many disciplinary areas [19].

Since learning and knowledge from the field of Computer Science in general can potentially be integrated and used productively in many other disciplines, promoting the transfer of computer science knowledge into these areas could substantially enhance learning and research within the computer science field. Having a tool, which could detect the potential transfer of computer science knowledge, as well as to other disciplines, would tend to increase the breadth and validity of computer science research, and contribute to the growth of the field. The CTP graph could potentially demonstrate the existence of knowledge transfer, not just within related computer science fields (Figure 1), but across disciplinary lines.

The CTP graph was first developed as a means to offer feedback to students uploading their games to the SGDA. The SGDA served as a submission format for introductory game programming courses using AgentSheets. During the semester, students are exposed to simple computational thinking patterns; as the class progresses they are introduced to more complex and diverse computational thinking patterns. Towards the end of the class, students are given open-ended assignments. For these assignments students are encouraged to build on their initial knowledge from the class in order to create their games. For the final project, students often choose to create simulations that depict some natural phenomena. Semantically analyzing a given student's games from the beginning of the semester as compared to their final project (especially a science simulation), could offer an opportunity to discover potential knowledge transfer.

For instance, a chaos theory simulation created by one student (Figure 11) with the accompanying CTP graph, shows how he mixed and combined computational thinking patterns that he had learned and used when previously programming Sokoban (Figure

9) and Sims (Figure 10). The CTP graph of his science simulation is very similar to the combined CTP graphs of Sokoban and Sims (Figure 12). Consequently, for this student, the CTP graphs indicate that knowledge transfer has occurred.

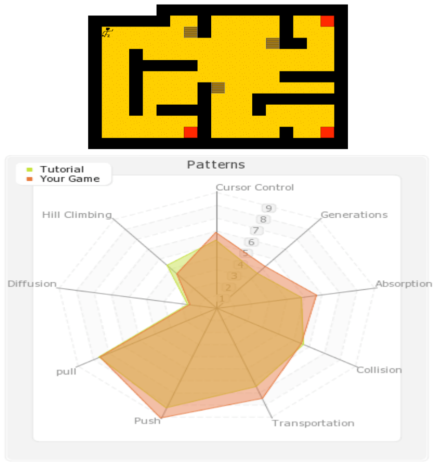

Figure 9: Screenshot and CTP Graph of Sokoban

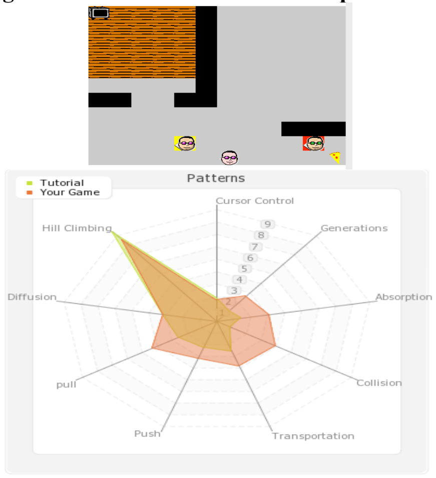

Figure 10: Screenshot and CTP Graph of Sims

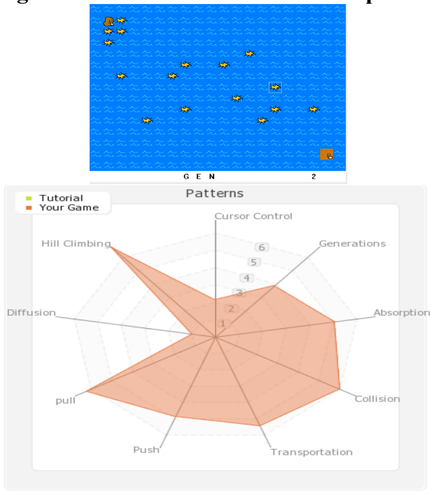

Figure 11: Screenshot and CTP Graph of Chaos Theory Simulation

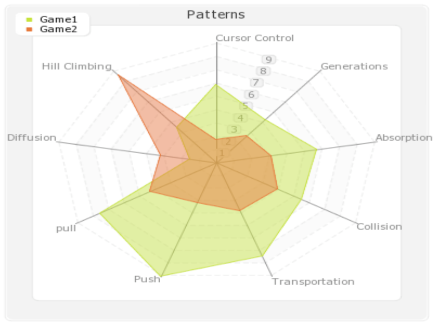

Figure 12: Comparison of CTP Graphs: depicts the Sims-Sokoban combination

4. Discussion

For the last decade, several visual programming languages have provided easy ways for young children to learn programming concepts and skills. Many of these visual languages successfully motivate students. However, visual language research has not focused on what kind of knowledge students have actually learned from creating these games. The CTP graph provides an initial way to assess specific knowledge accumulated by students within a given class.

As we have observed, the ability to detect computational thinking patterns is important for school teachers and students using visual languages for education. The CTP graph provides us with an initial means to answer the question "Now that the student can program Space Invaders, can the student program a science simulation?" Furthermore, the CTP graph has the ability to enable Human Centric Computing as teachers can get immediate feedback on their student's progress.

Limitations of the CTP graph include the arbitrary nature of the specified computational thinking patterns, the difficulty in differentiating similar patterns, and the number of computational thinking patterns chosen for the CTP graph. Among the chosen computational thinking patterns, a few, such as diffusion, are not depicted as accurately as the others, such as hill climbing. Although this anomaly needs to be further investigated, it does not taint the relative accuracy of the CTP graph, nor diminish its value for detecting the presence of knowledge transfer in these situations.

Analyzing computational thinking patterns in multiple combinations is a step closer to demonstrating the depth and breadth of students' knowledge. The semantic nature of the CTP graph allows us to evaluate and visualize a program's actual underlying meaning. A syntactic evaluation of a student's learning only shows the student's knowledge in a very limited context. Moreover, the implementation of a given student's previously learned computational thinking patterns to a scientific context gives us a clearer picture of how the student transferred new knowledge to a new situation, demonstrating that through the CTP graph comparison knowledge transfer exists.

4.1. Future Work

Although in most learning scenarios, knowledge transfer is often assumed, this transfer cannot be guaranteed to have actually taken place. The CTP graph is a better tool for evaluating knowledge transfer because the graph represents the CTP combinations as an observable, and definable outcome. The ability to detect knowledge transfer through the CTP graph, over the duration of a semester course, is a positive first step towards measuring transfer in other areas, and possibly other forms of learning.

Future research will include additional validation of computational thinking pattern recognition. The current model has been validated manually by comparing CTP graph output with human graders evaluating computational thinking patterns by playing games/simulations and looking at the source code. The CTP graphs have performed quite well. However, at this point we speculate that problems such as potential false positives could be reduced by deepening the level of analysis. Specifically, the current level of analysis stops at individual conditions and actions. The analysis does not drill further down into parameters to these actions and conditions which could be used to discriminate between similar patterns more effectively.

5. Acknowledgments

This material is based in part upon work supported by the National Science Foundation under Grant Numbers No. 0833612 and DMI-0712571. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

6. References

[1] T. Selker and L. Koved, “Elements of Visual Language”, *IEEE*, 1988, pp. 38-44.
[2] S. Chang, R.R. Korfhage, S. Levialdi, and T. Ichikawa, “Ten Years of Visual Languages Research”, *IEEE*, 1994, pp. 196-205.
[3] H. Lieberman, F. Paternò and V. Wulf (Eds), Use end-user development book, End User Development. New York, NY: Springer, 2006
[4] A. Repenning, D. Webb, A. Ioannidou, “Scalable Game Design and the Development of a Checklist for Getting Computational Thinking into Public Schools”, *Proc. SIGCSE’ 10*, ACM Press, WI, USA, 2010.
[5] A. Basawapatna, K.H. Koh A. Repenning, “Using Scalable Game Design to Teach Computer Science From Middle School to Graduate Schools”, ITICSE’10 Ankara, Turkey, 2010, In Press.
[6] P.B. Henderson, T.J. Cortina, O. Hazzan, and J.M. Wing, “Computational Thinking”, *Proc. SIGCSE’07*, KT, USA, 2007, pp. 195-196.
[7] J.M.Wing, “Computational Thinking and Thinking about Computing”, *Philosophical Transactions of the Royal Society A*, 366, 2008, pp. 3717-3725.
[8] National Academy of Sciences on Computational Thinking, *Report of a Workshop on The Scope and Nature of Computational Thinking*, National Academies Press, 2010
[9] Z. Les and M. Les, “Thinking, Visual Thinking, and Shape Understanding”, *Studies in Computational Intelligence*, 86, 2008, pp. 1-45.
[10] C.M. Lewis, “How Programming Environment Shapes Perception, Learning and Goals: Logo vs. Scratch”, *Proc. SIGCSE’10*, ACM Press, WI, USA, 2010.
[11] K.F. Moore and D. Grossman., “ High-Level Small-Step Operational Semantics for Transactions”, *Proc. POPL’08*, CA, USA, 2008, pp. 51-62.
[12] S.P. Reiss, “Semantics-Based Code Search”, *Proc. ICSE’09*, Vancouver, Canada, 2009, pp.243-253.
[13] S. Walter, B. Barron, K. Forssell, and C. Martin, “Continuing Motivation for Game Design”, *CHI 2007*, CA, USA, 2007, pp. 2735-2740.
[14] C. Johnson, S. Parker, D. Weinstein, “Large-scale computational science applications using the SCIrun problem solving environment”, *Proc. International Supercomputer Conference 2000*, 2000.
[15] J. Gindling, A. Ioannidou, J. Loh, O. Lokkebo, and A. Repenning., "LEGOsheets: A Rule-Based Programming, Simulation and Manipulation Environment for the LEGO Programmable Brick," Proceeding of Visual Languages, Darmstadt, Germany, IEEE Computer Society Press, 1995, pp. 172-179.
[16] A. Repenning and A. Ioannidou, “Broadening Participation through Scalable Game Design”, *Proc. SIGCSE 2008*, ACM Press, OR, USA, 2008
[17] T. K. Landauer, P. W. Foltz, & D. Laham, Introduction to Latent Semantic Analysis. *Discourse Processes,* 25, 1998, 259-284.
[18] A. Repenning and J. Ambach, "Tactile Programming: A Unified Manipulation Paradigm Supporting Program Comprehension, Composition and Sharing," *Proc. the 1996 IEEE Symposium of VL/HCC*, CO, USA, 1996, pp. 102-109.
[19] J.D. Bransford., A.L. Brown., & R.R. Cocking., *How People Learn: Brain, Mind, Experience, and School*, National Academy Press, Washington DC, USA, 2000.