Proceedings, The Thirteenth AAAI Conference on
Artificial Intelligence and Interactive Digital Entertainment (AIIDE-17)

Evaluation of a Template-Based Puzzle Generator for an Educational Programming Game

Yihuan Dong, Tiffany Barnes
North Carolina State University
Raleigh, North Carolina 27606

Abstract

Although there has been much work on procedural content generation for other game genres, very few researchers have tackled automated content generation for educational games. In this paper, we present a template-based, automatic puzzle generator for an educational puzzle programming game called BOTS. Two experts created their own new puzzles and evaluated generator-generated puzzles for meeting the educational goals, the structural and visual novelty. We show that our generator can generate puzzles with expert-designed educational goals while saving experts more than 80% of creation time, and these puzzles exhibit structural and visual novelty compared to expert-created puzzles. The contribution of this work is defined and implemented the first template-based automatic puzzle generator that saves expert time while incorporating expert-designed educational goals and enhancing puzzle creativity.

Introduction

Game-based learning has been shown to be nearly as effective as one-on-one human tutoring (Chaffin et al. 2009; Eagle and Barnes 2009). A lot of educational games (Eagle and Barnes 2008; Smith et al. 2012; Hicks 2013) choose to use puzzles as educational content because, similar to doing homework assignments, solving puzzles is a natural way for players to learn and repeatedly practice an educational goal. Most educational puzzle games rely on expert-created puzzles. Experts usually take care to create a linear sequence of progressively more difficult puzzles to allow for learning.

Despite the intellectual benefit of having experts manually create puzzles, it has several problems. Firstly, creating high-quality game content often has a low production rate and requires a great deal of developer and/or educator time. Murray (Murray 2003) estimated that it takes approximately 300 hours to create a single hour of educational content for an intelligent tutoring system, without considering the time devoted to game design and player immersion that would be required to make an effective educational game. Secondly, the low production rate leads to the challenge for experts to create enough puzzles to provide replayable experience, which is a key component of a successful educational game (Prensky and Prensky 2007). There is often no way to replay the same game to review or practice the skills learned, without playing the exact same puzzles. Thirdly, designing puzzles for educational games is tedious and requires considerable effort to make a set of distinct puzzles with the same educational goal for players to practice, even for expert designers. The puzzle designer has to consider both including educational goals and satisfying the structural constraints of good puzzles, and this takes considerable expert time. Lastly, human imagination is limited (Togelius et al. 2011), making it hard for experts to create a variety of interesting-looking puzzles that meet the target educational goals.

To make the puzzle creation process for educational games more efficient, we propose a *template-based puzzle generator*. We define a template-based puzzle generator as an automatic generator that reads in an user-written *template* encoded with goal-oriented requirements and then generates different puzzles that meet the requirements defined by the template. Using template provides experts a controllable and consistent way to design educational goals. The generator frees experts from worrying about the puzzle's non-educational, structural constraints while helping generate new, novel structures for puzzles, which allows novice players to practice the same concept repeatedly without having to replay the same puzzle.

This work is a first step towards using a template-based method in content generation for educational games. We implemented a template-based puzzle generator (referred to as puzzle generator or generator later) for an educational game called BOTS (Hicks et al. 2015). The puzzle generator allows designers to write a structural template encoded with specific educational goals and generates different puzzles for BOTS that satisfy the template. To test the benefit of the generator, we investigate three research questions:
RQ1-Time: Can the generator save expert time?
RQ2-Learning: Can the generator preserve learning goals?
RQ3-Novelty: Can the generator help experts enhance puzzle creativity?

Related Work

There have been a lot of studies on generating game content for entertainment games (Khalifa and Fayek 2015; Togelius et al. 2011; Hendrikx et al. 2013). Smith (Smith, Whitehead, and Mateas 2010) developed a mixed-initiative design tool named Tanagra for 2D platformer games like

Super Mario Bros. The tool allows level designers to place constraints by manipulating exact geometry placement on a continuously running level generator that automatically fills the rest of the level while guaranteeing playability. Antonova (Antonova and others 2015) applied Answer Set Programming (ASP) to develop Portal game levels that satisfy physics constraints. The generator can only be used as a partial offline level creation tool since the generated puzzles need to be verified by a human designer. Mourato et al. (Mourato, dos Santos, and Birra 2011) investigated in using Genetic Algorithms for generating levels for a platformer game called Prince of Persia in the attempt to achieve higher expressivity and less linearity than rhythm-based approaches. Dormans (Dormans 2010) used generative grammar and shape grammar to generate mission and space for adventure games.

Few studies focused on content generation for educational games. Smith et al. (Smith et al. 2012) compared 6 implementations of a level generator for an online educational game called Refraction. They demonstrated that hard constraints are easy to incorporate into suitably designed generative processes while yielding high-quality puzzles with good expressive power. In their later work (Smith, Butler, and Popovic 2013), they found that the distractor pieces in the generated puzzles may introduce undesirable solutions that do not demonstrate the target concept. They addressed this problem by making two design-oriented extensions to answer set programming and offered a general way to search for all possible solutions to high-complexity problems. Hicks (Hicks 2013) investigated generating puzzles for an educational game called BOTS using player generated content. which can produce a variety of different puzzles. The limitation with this approach is that the majority of BOTS players are novices, which makes it hard to ensure the educational quality of the puzzles they create. Also, it is hard even for experts to create puzzles with the build-in puzzle editors, as the expert needs to consider how to achieve educational goals while satisfying structural constraints of valid puzzles.

BOTS

BOTS[1] is an educational puzzle game designed to introduce fundamental programming concepts to players. It teaches players handle repetitive patterns using *loops* and *functions*.

Figure 1 shows an example of BOTS game interface. On the top right of the interface is the command panel. Players can drag and drop commands into the program panel on the left to create a program that controls a robot. The display panel on the bottom right shows the physical appearance of the puzzle to be solved. After creating a program, the player can click the "play" button on the top left of the interface to have the robot execute the program. The execution stops with a warning message if an error occurs or with a congratulation message if the puzzle is solved.

A *puzzle* in BOTS consists of a robot that executes player's program, gray blocks where the robot can travel on, and a number of yellow circle switches that serve as destinations. Many puzzles also include blue boxes that the robot can pick up, carry, and put down. To solve a puzzle, a player needs to construct a program that guides the robot to press all the switches at the same time, either by placing a box on them or by standing on top of them.

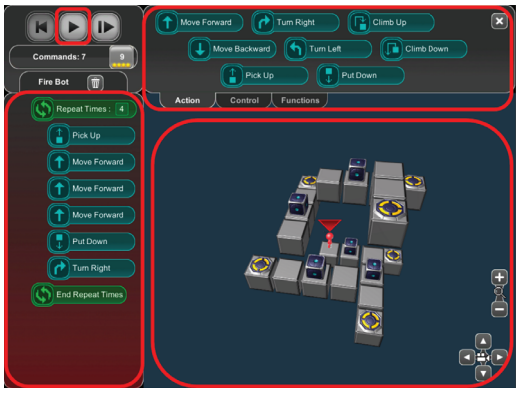


Figure 1: A screenshot of the BOTS game interface.

The robot is capable of *eight basic actions*: move "forward" and "backward" one block, "turn left" and " turn right", "climb up" and "climb down" one block, "pick up" and "put down" a box in front of the robot. These eight basic action commands suffice to solve any valid puzzles in BOTS. In addition, there are two control flow operations for repeating blocks of commands: the "repeat" command and the named "function" command. The "repeat" command, as shown in Figure 1, iterates through the actions it contains a number of times set by players. BOTS provides players with six empty functions with names from A to F. Players can edit the content of these functions and make function calls just like using the basic action commands.

The objective of the game is for players to solve the puzzle using a solution that has the least number of commands, which is referred to as the optimal solution. The optimal solution often contains the concept that the puzzle designer wants the players to practice, primarily the use of "repeat" and "function" commands. The game uses a point system to encourage players to look for opportunities in the puzzles to optimize their solutions with loops and functions in order to have a minimal number of commands — the fewer commands used, the higher the score.

Generator Design

Figure 2 provides an overview of the structure of the template-based puzzle generator. The generator has four components: a *template Parser*, a *solution Generator*, a *constraint Checker*, and a *program file formatter*. A while loop and an array that stores generated puzzles are introduced into the generator to control the number of puzzles to generate and ensure uniqueness of generated puzzles.

To generate puzzles, an expert first determines the educational goals, and considers what solution program structures achieve those goals. Once the educational goals are set, the expert encodes the structure of the solution program into a

[1] http://bots.game2learn.com/

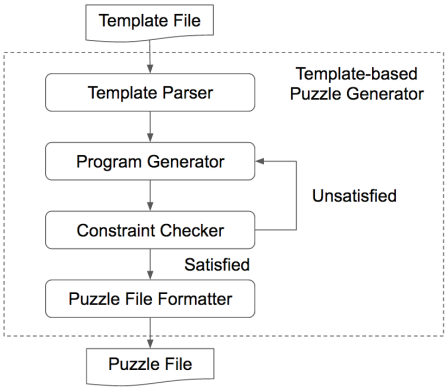


Figure 2: The structure of the template-based puzzle generator

template file as the input of the template-based puzzle generator. Upon receiving the template, the template parser checks the template's validity, creates a template object, and passes this object to the solution generator. The solution generator generates valid solution programs that meets the template structure, and then passed the solution programs to the constraint checker to make sure all of the constraints for good puzzles are satisfied. If any constraint is violated, the Constraint Checker will abandon the solution and ask the solution generator to generate a new one. Otherwise, the solution program is sent to the puzzle formatter where it is converted to a puzzle file for the BOTS game. The generator stops if it cannot generate a puzzle after a certain number of attempts and suggests that the expert should adjust the template. More detailed design decisions for each component are explained in the following subsections.

Template Parser

The template parser determines if the template file is valid, parses the template file into a template object and passes the template object to the Solution Generator.

The *template* is a file that encodes the expert-desired educational goals and requirements. Experts can use two types of statements to compose a template easily: *action statements* and *control statements*. The action statements are the names of the eight basic action in BOTS. The control statements, including function statement, repeat statement and wildcard statement, are for experts to design control structures and missing pieces in the template. The function statement defines a function with a specified function name and function content. Once defined, a function can be called without rewriting its content. The repeat statement defines a loop structure with the number of times to repeat and the repeat content. The wildcard statement is a placeholder for a sequence of random basic action commands for the Solution Generator to generate.

Figure 3 shows an example of finished template file. This template first defines a function named "A" that has the robot do 4 expert-specified actions. Then the template defines a repeat statement that iterates 4 times doing a sequence of 5 unspecified actions. Finally, it calls function "A" again.

```
1    function A
2        pick up
3        turn right
4        put down
5        turn left
6    end function
7    repeat 4
8        wildcard 5
9    end repeat
10   function A
```

Figure 3: An example of a template file

Solution Generator

The solution generator takes in a template object, generates content for all the wildcard statements to produce a *valid* solution program.

The generation of wildcard statement content uses a check and simulate fashion. The solution generator uses a 3D array of boolean values to represent the presence of game elements inside each cubic space in the BOTS game environment. We call this 3D array the *world* and each cubic space in the world a *cube*. When generating wildcard contents, the solution generator first simulates each statement in the template in the world (recursively if the statement is a control statement). Upon seeing a wildcard statement, it follows the following procedure to generate the wildcard content:

1. Get all valid next-step actions given the world state.
2. Randomly pick a valid next-step action, simulate the action in the world, and append it to the wildcard content.
3. The generator repeats step 1 until the number of actions reaches the specified length.

If an action could not be performed due to violating puzzle structural constraints or if no valid next-step action is available during the generation process, the Solution Generator will abandon the whole program and start a new attempt. The successfully generated solution program is then sent to the Constraint Checker for quality assurance.

Constraint Checker

The constraint checker is responsible for enforcing the quality constraints on the generated solution programs.

A good puzzle in BOTS needs to meet several requirements. First, the puzzle needs to be solvable, with at least one straightforward solution consisting of only the 8 basic actions. Second, the puzzle should not have game elements that aren't needed to solve the puzzle, as those are considered distractions to the students in an educational setting. Thirdly, the solution for an expert-created puzzle needs to be optimizable with loops and/or functions. Here, optimizable means that the solution program can be made short with the use of loop and/or function statements. Lastly, expert-created puzzles should also have affordances for students — the educational goal should be easily visible in the puzzle layout.

We separate the constraint checker from the solution generator for two reasons. First, the solution generator does not

guarantee the overall quality and may include undesirable action sequence like meaningless cancellation of actions in generated solution program (e.g. "forward" action immediately followed by "backward" action). Secondly, unlike the action constraints in the solution generator, which can be obtained directly from BOTS, the constraints in the constraint checker are customized with the expert knowledge of game content. Separating the solution generator and the constraint checker makes a better modular design.

The constraint checker uses a set of expert predefined hard constraints that can be applied to all generated puzzles. We classified these constraints into three categories: *program constraints*, *world constraints* and *educational constraints*. Each category contains 2 constraints. The program constraints look for undesirable action sequences in the solution program, such as immediate cancellation of actions. The world constraints look for undesirable structures or states in simulated worlds, for example if the generated puzzle initializes with all switches pressed. The educational constraints ensure the educational quality of the generated puzzles, for example no unintentional loops and functions introduced into the generated solution program and no unused game elements that distract the player. Note that this is not a sufficient set of constraints that guarantees the generation of good puzzles. However, these constraints already yield a high success rate, which will be discussed later.

If any constraint is not satisfied, the constraint checker will abandon the program and ask the solution generator to generate another solution program. Otherwise, it sends the solution program to the puzzle file formatter.

Puzzle File Formatter

The puzzle file formatter is responsible for converting solution programs into usable puzzle files for BOTS. The formatting process goes through three steps. First, the puzzle file formatter executes the solution program to get a final state of the world. Then, it normalizes the world to the minimum cubic space needed for the generated puzzle. Finally, it extracts game elements from the world and encodes them into a puzzle file that meets the BOTS requirements.

Method

This section describes the two experiments we did to answer our three research questions.

Experiment 1: Puzzle Creation

The first experiment investigates how much time the puzzle generator can save for experts when generating puzzles, regardless of the educational requirements (RQ1-Time).

First, two experts created puzzles following given requirements using the program-based puzzle editor, and they recorded time used to create each puzzle. One of the experts is the first author and the other expert is experienced with creating educational puzzles for BOTS. They each created 12 puzzles with the following constraints: three of the puzzles needed to have *short loops* of 6 actions that iterate 4 times; three needed to have *long loops* of 12 actions that iterate 4 times; three needed to have *short functions* of 6 actions called 3 times; and three needed to have *long functions* of 12 actions called 3 times each. One expert created puzzles for loops first and the other expert created puzzles for functions first. All created puzzles needed to have box interactions because box interaction is an important component that highlights the puzzle goals and increases both the difficulty to create and to solve those puzzles. Then, the two experts created puzzles following the same requirements and order using our template-based puzzle generator, and recorded time they used to write the templates, run the puzzle generator and ensure the puzzles files contain valid puzzles that can be opened in BOTS.We compared the time the experts used to hand-author the puzzles and the time they took to do the same task using the generator.

In addition, we evaluated the performance of the puzzle generator. We recorded the average time of 100 runs for the generator to generate 1, 3, 5, 10, 15 and 20 different puzzles respectively for the 4 requirements described above. We also manually checked the good puzzle rate in the last run of the 20-puzzle group (a total of 80 puzzles) as an estimate of the success rate using the constraints described above.

Experiment 2: Expert Puzzle Rating

The second experiment evaluated how effective the generator is at generating puzzles for the desired educational goal (RQ2-Learning) and how novel the generator-generated puzzles are (RQ3-Novelty).

In this experiment, we asked the same two experts to independently rate the 12 puzzles the puzzle generator generated in experiment 1 for them respectively, by filling out an evaluation questionnaire for each puzzle. Below we list the 9 questions asked in the questionnaire. All questions except for Q3 - Q5 were on a 4-point Likert scale, "Not at all xxx", "Not xxx", "xxx", "Extremely xxx". Q1 - Q3 focused more on the experts' opinions about the physical appearance of the generated puzzles and the rest of the questions focused more on the educational aspects.

Q1. How complex does the structure of the puzzle look?

Q2. How interesting does the puzzle look?

Q3. Does the puzzle have a novel structure that you didn't think of when creating puzzles?

Q4. How long does it take you to solve the puzzle with only basic actions (in minutes and seconds)?

Q5. How much time does it take for you to solve the puzzle using loops/functions (in minutes and seconds)?

Q6. How visually apparent is the loop/function structure in the puzzle layout?

Q7. How hard do you think it is for a novice to solve the puzzle with only basic actions?

Q8. How hard do you think it is for a novice to solve the puzzle with loops or functions?

Q9. How valuable is the puzzle w.r.t. learning how to use loops/functions?

Results and Discussion

Results for Experiment 1

Table 1 shows the total time the experts took to create the puzzles for each requirement in the first experiment. The "validation" in the third and fourth column refers to the time needed for the experts to open the puzzle in BOTS to verify if the puzzle is a valid, playable puzzle (though not necessarily a good, educational puzzle).

Table 1: Two experts average total time cost (in seconds) to create the 3 puzzles for each educational goal.

Requirement	Expert by Hand	Generator w/ Validation	Generator w/o Validation
S Loop	203	58	22
L Loop	625	68	23
S Func	373	109	63
L Func	734	113	52
Total	**1935**	**348**	**160**

Looking at each educational goal, we found that experts took more than twice the time to finish the "long" puzzle groups , indicating that larger puzzles are more time-consuming and presumably more difficult to create than smaller puzzles. Comparing the expert times between the two educational goals, we found that puzzles teaching functions took longer for the experts to create than puzzles teaching loops, even though the number of function calls is one less than the number of loop iterations. We think this is because creating puzzles for functions requires time to make additional design decisions, such as what actions to put in between function calls. The experts showed the same pattern when writing templates for the generator: writing templates for functions took about 35 seconds longer than writing templates for loops, where the former took about 55 seconds on average and the latter took about 20 seconds.

In general, experts took an average of 1935 seconds (about 32 minutes) to create twelve puzzles. In contrast, our template-based puzzle generator greatly reduced the time needed to create twelve educational puzzles under the same requirements to only 348 seconds (about 6 minutes) including puzzle validation and reduced to only 160 seconds (about 3 minutes) without the validation process. This shows that the template-based puzzle generator is able to save more than 80% of the expert time needed for puzzle generation and validation. Moreover, if experts have access to pre-defined templates, the work that used to take 30 minutes could potentially be done in merely seconds.

Table 2 shows the results of the performance evaluation on a laptop with Intel Core i5 1.3 GHz processor. A Wilcoxon signed-rank test shows that the time needed to generate one puzzle for short loops and short functions is significantly shorter than that for long loops and long functions (Both $Z = 2.2014, p = 0.03125$). The time does not change much with the increase of the number of distinct puzzles to generate, possibly because the number of puzzles required is far less than the whole solution space of the template. However, generating 20 distinct puzzles is already enough for one template. After evaluating the 80 puzzles from the 20 puzzles group, 93.75% puzzles were good puzzles. All the bad puzzles, which had several unnecessary blocks, were had the Long Loop group, indicating additional constraints needed for the group.

Table 2: The average time (in seconds) used to generate one puzzle for each experiment.

# of Puzzles	1	3	5	10	15	20
S Loop	0.29	0.26	0.26	0.28	0.28	0.31
L Loop	0.40	0.38	0.45	0.44	0.39	0.46
S Func	0.29	0.22	0.26	0.26	0.27	0.22
L Func	0.41	0.41	0.38	0.35	0.38	0.37

Results for Experiment 2

Table 3 shows the distribution of the expert evaluation responses. Note that the ratings of Q1 - Q8 do not necessarily mean the generated puzzles were good or bad. The distribution presents the expressive range of the puzzle generator from the experts' point of view.

There are several observations from Table 3. First, if we split the 4-likert scale options in the middle into positive ratings (right side) and negative ratings (left side), the experts had fairly even numbers of positive ratings and negative ratings for Q1, Q6, Q7 and Q8. This indicates that the generator can generate puzzles with varying visual complexity, apparentness of the educational goal, and difficulty to construct both straightforward and optimal solutions. Interestingly, people would normally consider that constructing a straightforward solution to a puzzle is not very difficulty. However, both the experts felt it would be difficult to construct straightforward solutions for half of the generated puzzles. This is because it becomes harder to mentally track the robot's status (location, facing direction, etc) as the length of the straightforward solution grows. One usually needs to have the robot execute the solution several times while constructing a long straightforward solution.

Moreover, results for Q2 and Q9 show that all but one generator-generated puzzles were rated at least interesting and valuable at teaching targeted educational goal. This means the puzzle generator is capable of generating interesting looking puzzles and preserve expert-designed learning goals in generated puzzles. A Wilcoxon signed-rank test on the responses of Q4 and Q5 shows that it took significantly longer for the experts to construct straightforward solutions than optimal solutions for the generator-generated puzzles ($Z = 4.2859, p = 1.192e - 07$). While this does not necessarily mean the puzzle will lead to learning, the significant time difference between the two solutions may encourage players to construct optimal solutions instead of using straightforward solutions when playing the puzzles.

Lastly, we evaluated novelty of the generator-generated puzzles. The puzzles in the top row of Figure 4 are not novel because of the similar spiral structure, while the puzzles in the bottom row are novel because they have very different visual structure. According to the responses to Q3, 8/12 and 9/12 generated puzzles had novel structures compared to the puzzles the experts created themselves, indicating that the

Table 3: Results from expert puzzle evaluation questionnaire. The bar chart presents with "Not at all xxx" on the left and "Extremely xxx" on the right, except for Novelty (Q3) which has "No" on the left and "Yes" on the right.

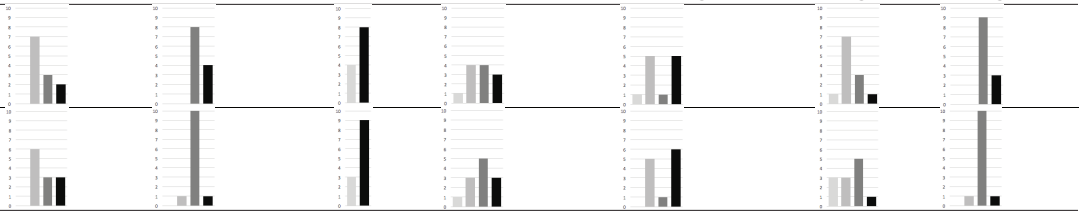
	Complexity (Q1)	Interestingness (Q2)	Novelty (Q3)	Apparrentness (Q6)	Difficulty Straightforward (Q7)	Difficulty Optimal (Q8)	Educational Value (Q9)
Expert 1							
Expert 2							

Figure 4: Expert-created puzzles (top) and generator-generated puzzles (bottom) for short loop requirement.

template-based puzzle generator can help experts discover novel puzzle structures.

We had two interesting observations when comparing the generator-generated puzzles with expert hand-authored puzzles. The first interesting finding is that, as shown in Figure 4, the experts created different puzzles with very similar visual structures for some educational goals. A possible explanation is that the experts' imaginations may be bounded by their mental set on how to present the educational goal. This mindset may limit experts' imaginations and make it hard to think of puzzles with very different appearances. From this perspective, the template-based puzzle generator can help experts discover new ideas and new puzzle structures.

Another interesting finding is that, unlike the generator-generated puzzles, no expert-created puzzles used reverse patterns, such as "backward" action or turn-around-and-go-back in their puzzles. Similarly, experts also never interacted with two or more boxes in the loop/function content of their solution program. However, these are important components that may make the puzzles more fun and more challenging to solve by creating alternative solution programs for players to discover the optimal one. The reason may be that designing these components requires additional cognitive load for experts to memorize previously traveled paths to append corresponding actions. In contrast with expert-created puzzles, 5/12 and 3/6 generator-generated puzzles had reverse patterns and multiple box interactions (only the 6 puzzles with long content length are long enough to have multiple box interactions), indicating that the template-based puzzle generator can help expert design novel puzzles with these patterns.

Limitation

One limitation to this study is that only two experts participated in the experiments, making it hard to claim statistical significance from the results. Secondly, we only focused on how satisfy the experts were with the generator-generated puzzles and did not have the expert rate the puzzles they created by themselves. Thus, we do not know if the experts' expectations for the generator are the same to their expectations for themselves.

Conclusion and Future Work

In this paper, we defined, implemented and tested a template-based puzzle generator on an educational game called BOTS. The generator allows experts to write educational goals into a template and uses the template to generate a number of puzzles with different appearances but with the same intended educational goal.

We compared the time needed for two experts to hand-author puzzles and use the puzzle generator to generate puzzles under the same requirements. We also asked the experts to evaluate the quality of the generator-generated puzzles. Our results show that our generator saved more than 80% of time for experts on puzzle generation; it generally generated good puzzles with expert-designed educational goals incorporated; it can help experts discover novel puzzle structures that they otherwise may not have been able to.

For future work, we would like to make the generated puzzle difficulty customizable in the template-based puzzle generator. Although the generator allows experts to specify educational goals with templates, the experts do not have control over the difficulty and/or complexity of the puzzles that the generator generates. However, this is an important requirement for creating a progressive learning experience automatically (Dewey 2007). We plan to apply machine learning technologies to generate puzzles with adjustable difficulty.

References

Antonova, E., et al. 2015. Applying answer set programming in game level design.

Chaffin, A.; Doran, K.; Hicks, D.; and Barnes, T. 2009. Experimental evaluation of teaching recursion in a video game. In *Proceedings of the 2009 ACM SIGGRAPH Symposium on Video Games*, 79–86. ACM.

Dewey, J. 2007. *Experience and education*. Simon and Schuster.

Dormans, J. 2010. Adventures in level design: generating missions and spaces for action adventure games. In *Proceedings of the 2010 workshop on procedural content generation in games*, 1. ACM.

Eagle, M., and Barnes, T. 2008. Wu's castle: teaching arrays and loops in a game. In *ACM SIGCSE Bulletin*, volume 40, 245–249. ACM.

Eagle, M., and Barnes, T. 2009. Experimental evaluation of an educational game for improved learning in introductory computing. In *ACM SIGCSE Bulletin*, volume 41, 321–325. ACM.

Hendrikx, M.; Meijer, S.; Van Der Velden, J.; and Iosup, A. 2013. Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 9(1):1.

Hicks, D.; Dong, Y.; Zhi, R.; Cateté, V.; and Barnes, T. M. 2015. Bots. In *CEUR-WS*.

Hicks, A. 2013. Bots: Harnessing player data and player effort to create and evaluate levels in a serious game. In *Educational Data Mining 2013*.

Khalifa, A., and Fayek, M. 2015. Literature review of procedural content generation in puzzle games.

Mourato, F.; dos Santos, M. P.; and Birra, F. 2011. Automatic level generation for platform videogames using genetic algorithms. In *Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology*, 8. ACM.

Murray, T. 2003. An overview of intelligent tutoring system authoring tools: Updated analysis of the state of the art. In *Authoring tools for advanced technology learning environments*. Springer. 491–544.

Prensky, M., and Prensky, M. 2007. *Digital game-based learning*, volume 1. Paragon house St. Paul, MN.

Smith, A. M.; Andersen, E.; Mateas, M.; and Popović, Z. 2012. A case study of expressively constrainable level design automation tools for a puzzle game. In *Proceedings of the International Conference on the Foundations of Digital Games*, 156–163. ACM.

Smith, A. M.; Butler, E.; and Popovic, Z. 2013. Quantifying over play: Constraining undesirable solutions in puzzle design. In *Proceedings of the International Conference on the Foundations of Digital Games*, 221–228.

Smith, G.; Whitehead, J.; and Mateas, M. 2010. Tanagra: A mixed-initiative level design tool. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, 209–216. ACM.

Togelius, J.; Yannakakis, G. N.; Stanley, K. O.; and Browne, C. 2011. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games* 3(3):172–186.