Adaptive puzzle generation for Computational Thinking

Marco Scirea[1]

University of Southern Denmark, Denmark msc@mmmi.sdu.dk
marcoscirea.com

Abstract. This paper describes a system to generate puzzles with a difficulty degree that adapts to the player. The puzzle is designed with the objective of being used by young pupils, and it is mainly a planning/sequencing task, which is considered one of the aspects of computational thinking. The system is powered by a constrained multi-objective algorithm (NSFI-2Pop) – which evolves the sequences of actions necessary to solve the puzzle – combined with a stochastic algorithm that translates the sequences in playable levels. We also present a pilot evaluation of the system, which seems to indicate that the levels presented to the player are perceived as having an increasing difficulty.

Keywords: Procedural Content Generation · Evolutionary Algorithms · Computational Thinking.

1 Introduction

Computational thinking (CT) – being able to express problems and solutions in ways that a computer could execute – is becoming increasingly important in young pupils' education, given the increasing digitisation of our world.

Puzzles have been a common method to introduce young pupils to computational thinking [13]. The issue we want to address in this paper is that these puzzles, whether in digital or paper format, only present a static set of challenges and difficulty progression. This approach might work for some pupils, but wouldn't it be better for learning and concept assimilation if we could tailor the puzzles to fit each student's learning? The traditional way to do this, for example in the class, would be for the teacher to create new levels or new puzzles. This is however unpractical and very time consuming, especially if the quality of the new puzzles has to be consistent and high. In the case of digital puzzles it would also require the CT teachers to be programmers, as well as skilled game designers. So our goal here is also to alleviate the workload of teachers, and make it easier for them to assign digital puzzles to their pupils, as part of their augmented classroom [1] or as flipped classroom materials [8].

This paper will focus on logical puzzles in which the solution is unique and represented by a specific sequence of actions. Such puzzles can range from classical problems such as "river crossing puzzles" to puzzles that are closer to computer science topics (e.g. *Knight Tour*[1]is a puzzle where the player has to make a knight visit each city in a map/graph once and only once). Examples of such puzzles can be seen in games such as river-crossing puzzles, various puzzles appearing in the *Professor Layton* (Level-5, Matrix Software, 2007-2017) series (e.g. the *Toy Car minigame* in *Professor Layton and the Unwound future* [Level-5, 2008]), *Sokoban* (Thinking Rabbit, 1982), escape-the-room games (e.g. the *Zero Escape* game series [Spike Chunsoft, 2009-2016]), most adventure games (e.g. *The Curse of Monkey Island* [LucasArts, 1997]), and many more. Many puzzles leave more freedom to the player in how to solve them, allowing for more creative problem-solving, but we decided to focus on puzzles with predefined solutions since they are more controllable and allow us to make sure the player has mastered specific concepts when they are able to solve the problem.

In this project we want to generate puzzles (and their solutions) that require the player to reason about sequences of actions, their effect, and how a goal can be achieved in optimal way. Moreover, in order to solve the puzzles, the player has to simulate the effect of a sequence of instructions; when the produced outcome diverges from the player's intuition, the player will be forced to redesign the solution, and improve his or her understanding of the effect of the instructions on the state of the game. These are central aspects of CT: the player has to decompose the problem, formulate an algorithmic sequence of instructions to solve the puzzle, and possibly recognise similar situations from previous puzzles [18]. We decided therefore to evolve puzzle solutions, so our game will always propose the players: i) a solvable puzzle, ii) which include different kinds of challenges, iii) and have an adaptive difficulty level.

Finally, to make the system adapt to the player we introduce a simple player model that keeps track of: what puzzles the player is able to solve as a measure of what concepts have been internalised, and how long does the player take to solve each challenge, giving us a measure of how difficult the puzzle is perceived. This model influences the second and third objective described above, so that the target for the generative system is adjusted as the player is observed playing.

2 Background

2.1 Procedural Content Generation

Procedural content generation is an active field that focuses on generating content for games through AI methods [9]. There has already been some research in generating content for some puzzle games, but without keeping in account the learning aspect [4]. Some examples include: level generation for the *Cut the Rope* (ZeptoLab, 2010) puzzle game [17], generation of *Sokoban* levels [11], and generation of narrative puzzles for adventure games [7]. Moreover, the type of adaptive content generation that this paper describes can be seen as an instance of the experience-driven procedural content generation framework (*EDPCG*) [19], where the game adaptation mechanism generates puzzles with particular elements and challenges in response to the player's actions.

[1] https://teachinglondoncomputing.org/puzzles/

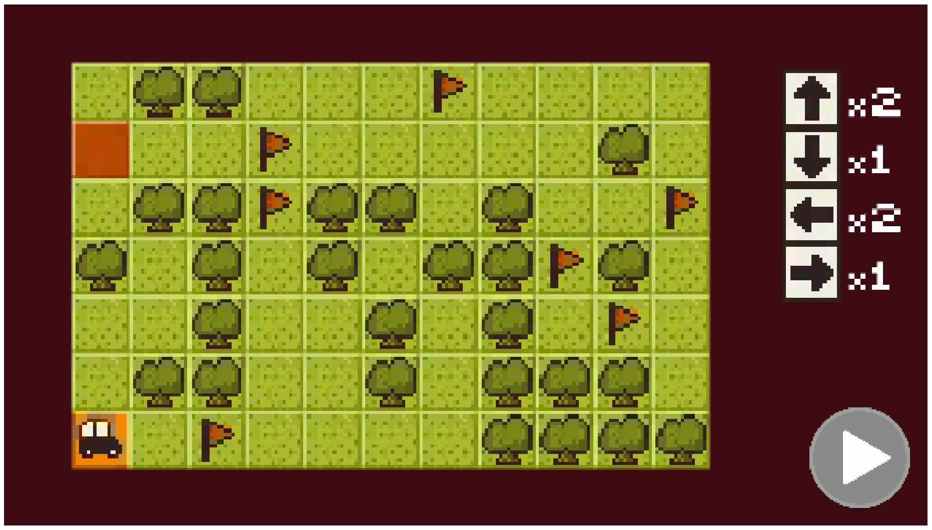


Fig. 1: An example of how the puzzle is presented to the player. See Figure 2 for the solution to this puzzle.

2.2 Computational thinking

Computational thinking has been defined as "a universally applicable attitude and skill set everyone, not just computer scientists, would be eager to learn and use" [18]. That can be seen as a very vague definition, and there is still much debate over a more precise definition of what computational thinking is, and what are its components.

Brennan and Resnick developed a definition of computational thinking that involves three key dimensions: "*computational concepts*(the concepts designers employ as they program), *computational practices*(the practices designers develop as they program), and *computational perspectives*(the perspectives designers form about the world around them and about themselves)"[2]. The computational concepts they identify are: *sequences*, *loops*, *parallelism*, *events*, *conditionals*, *operators*, and *data*. Computational practices are defined as *being incremental and iterative*, *testing and debugging*, *reusing and remixing*, and *abstracting and modularizing*.

In this paper, our system mostly focuses on *sequences* concepts and on *being incremental and iterative* (since our system generates more and more complex problems), and *testing and debugging* (since the game allows the pupils to plan their solutions and then see how it works, debug it, and improve on it).

2.3 Evolutionary computation

To generate the puzzles we use an evolutionary algorithm, this is a family of algorithms for global optimisation inspired by biological evolution. These algorithms can create highly optimised solutions for a wide variety of problems and moreover are able to create not just one solution but a wide range of different, but similarly optimised ones. In particular, we use a Multi-Objective Optimisation (MOO) approach: this is defined as the process of simultaneously optimising multiple objective functions. In most multi-objective optimisation problems, there is no single solution that simultaneously optimises every objective. In this case, the objective functions are said to be partially conflicting, and there exists, a number (possibly infinite) of Pareto optimal solutions. To understand what makes a solution better than another the concept of Pareto dominance is introduced: this is a binary relation between two solutions where one solution is Pareto dominant with respect to another solution if, for all objectives, it improves on the other solution.

Many search/optimisation problems have not only one or several numerical objectives, but also a number of constraints – binary conditions that need to be satisfied for a solution to be valid. A number of constraint-handling techniques have been developed to deal with such cases within evolutionary algorithms. The Feasible/Infeasible 2-Population method (FI-2POP) [12] is a constrained evolutionary algorithm that maintains two populations evolving in parallel, where feasible solutions are selected and bred to improve their objective function values, while infeasible solutions are selected and bred to reduce their constraint violations. In each generation, individuals are tested for constraint violations; if they present at least one violation they are moved to the 'Infeasible'population, otherwise they are moved to the 'Feasible' population. An interesting feature of this algorithm is that the infeasible population influences, and sometimes dominates, the genetic material of the optimal solution. Since the infeasible population is not evaluated by the objective function, it does not become fixed in a sub-optimal solution, but rather is free to explore boundary regions, where an optimum solution is most likely to be found.

When dealing with constrained optimisation problems, the approach is usually to introduce penalty functions to act for the constraints. Such an approach favours feasible solutions over the infeasible ones, potentially removing infeasible individuals that may lead to an optimal solution, and finding solutions that can be considered local optimum. There have been many examples of constrained multi-objective optimisation algorithms [14, 10, 6, 3]. In this paper we use an algorithm called Non-dominated Sorting Feasible-Infeasible 2 Populations (NSFI-2POP) [16, 15], which combines the benefits of maintaining an infeasible population, free to explore the solution space without being dominated by the objective fitness function(s), and finding the Pareto optimal solution for multiple objectives. This algorithm is essentially a combination of FI-2POP and NSGA-II [5].

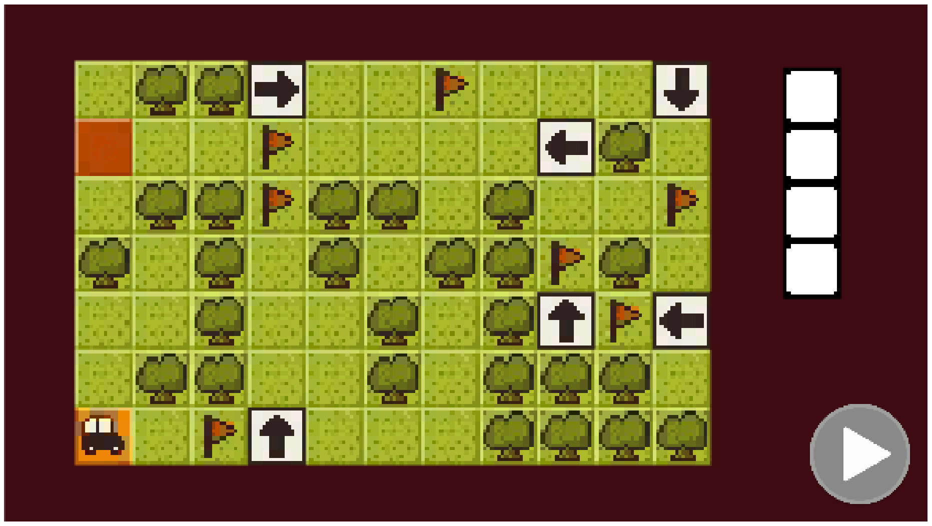


Fig. 2: The solution to the puzzle in Figure 1.

3 The system

In this section we discuss the details of the implementation of the system, in particular describing how does the game work, how does the puzzle generation work, and how we model the current solving ability of the player.

3.1 The game

We have chosen to recreate a mini-game from *Professor Layton and the Unwound future* [Level-5, 2008]); the game's goal is for a toy car to reach the goal square, while collecting all the flags on its path (see Figure 1). Looking at Figure 1, the car sprite indicates that the starting direction is to the right, the red tile is the goal that the car has to reach, and the flags have to be collected on the way. On the right the direction tiles that the player can drag-and-drop on the game map can be found. Note that direction tiles can only be placed on empty spaces. On the bottom right the play button starts the simulation, so that the player can observe if plan works as expected.

The player interacts with the game by positioning the direction tiles within the game grid, and then by pressing the ”play” button the player is able to observe the plan being executed. A small modification to the original game is that, as the car walks over the direction-change tiles, these are picked up, this can create more complex puzzles when the sequence is long enough (see Figure 12). Note that the player has no way to interact with the game once the simulation has started (apart from aborting it), which means that the player has to create and “test” the plan in its head before being able to see the actual results.

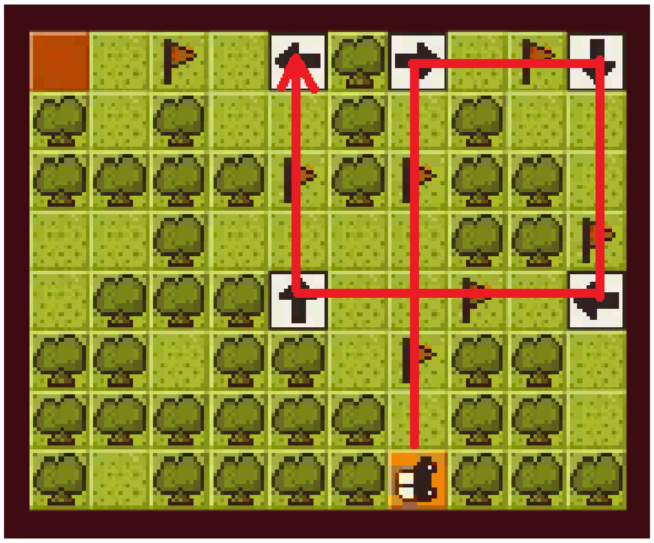


Fig. 3: A visualization of what we consider a *loop* in the context of this game

As discussed in section 2.2, our system mostly focuses on *sequences* concepts, on *being incremental and iterative*, and on *testing and debugging*.

While creating lengthy sequences is the most obvious challenge, puzzles can also include various other features, such as “loops” and recurring paths. By loops we don’t mean in the programming sense, but when the player has to make the car follow a path that makes it go through all directions before coming back to the starting one (e.g [left, up, right, down, left], also see Figure 3).

3.2 Generation of puzzle from solution

The evolutionary algorithm generates a set of directions that represent the solution of the puzzle. That is an abstraction, and not a complete level as can be seen in the previous figures. To generate the final level the set is passed to a semi-stochastic algorithm which is described in Figure 4.

As can be observed in Figure 4, the algorithm has a stochastic component, meaning that from the same solution different boards can be created (see Figure 5).

3.3 GA

This section describes details of the evolutionary system where domain-specific choices had to be made that deviate from the more general NSFI-2POP structure defined in section 2.3.

```
L: list of moves
increase: increase in chance of changing direction
P = empty list to hold the path
    carCoordinates = {0,0}
    foreach move m in L:
        changeProbability = 0
        while random number > changeProbability:
            carCoordinate = new position moving by m
            add carCoordinate to P
            changeProbability += increase
        Add an objective in the segment the car has traversed
    spawn car at {0,0}
    spawn goal at last position in P
    foreach tile t not in P:
        if random number > 0.3
            spawn obstacle in t
```

Fig. 4: Level generation algorithm: this transforms the evolved sequence of moves into the actual visual level

Genome representation The evolutionary genome consists of a number of values that represent the moves necessary to solve the puzzle. These values are left, right, up, down, and correspond to the tiles that the player has to place on the board (see Figure 2). The size of the genome is variable, since we want to be able to generate puzzles with different and possibly quite long sequences.

Constraints We have two constraints in this problem: we do not want to create solutions which contains opposite directions one after the other (e.g [..., left, right,...]) nor solution that contain the same direction more than once conse-

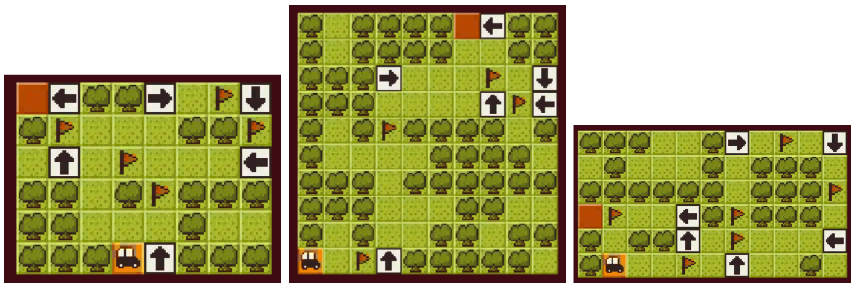


Fig. 5: Three examples of puzzles that are represented by the same solution $\{U, R, D, L, U, L\}$, showing how the system can express the same problem in a number of different ways

quently (e.g [..., left, left,...]). The opposite case is the most crucial, since it would represent an impossible solution (see Figure 6).

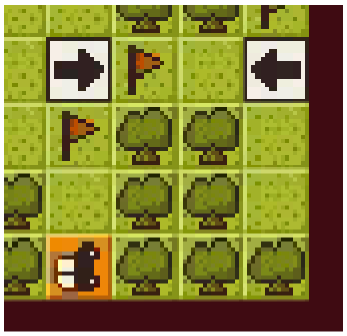


Fig. 6: Example of a situation we would like to avoid and use as a constraint to the evolution

The second case, where we have the same direction more than once consequently, is less critical, but it amounts to the same solution if the two duplicates where one:

$$[up, \textbf{left}, \textbf{left}, up, right] \equiv [up, \textbf{left}, up, right]$$

As such the feasability equation we define is:

$$Feasibility = -\sum_{i=0}^{n-1}(Opposite(i, i+1) + Same(i, i+1))$$

where n is the lenght of the genome.

The two funcions in the above equation are both Boolean ones, returning either 0 or 1 depending if the constraint is satisfied or not. The Feasibility function can return a number between $[0, -2(n-1)]$, where 0 means that the individual satisfies all constraints and can consequently be transferred from the infeasible population to the feasible one.

Fitness objectives The objectives used to evolve the individuals in the feasible population are two: the length of the sequence, and the amount of challenges (*loops*, see Section 3.1). These objectives do not follow a maximization or minimization problem (e.g. we do not want to evolve towards infinitely large sequences, since it does not make sense to present the player such puzzles). Instead both objectives have a target number, which is variable as time progresses and the player solves puzzles.

These numerical targets are controlled by the player model (see section 3.4), and represent the current difficulty calculated as the one fitting the player's current skill level.

Selection operator The feasible population (i.e. that running NSGA-II) utilises a binary tournament selection operator: two random individuals are chosen from the population and compared. The individual that dominates the pair is selected as a parent for a crossover operator, this operator is executed twice to obtain the two parents required by a crossover operator. In the event neither individual dominates the pair, a parent is chosen randomly among the pair.The infeasible population uses a roulette-wheel selection operator: the selection is a stochastic process where individuals have a probability of becoming parents for the next generation proportional to their fitness. In this way individuals with higher fitness are more likely to be selected while individuals with lower fitness have a lesser chance, however they may have genetic material that could prove useful to future generations and are therefore preserved.

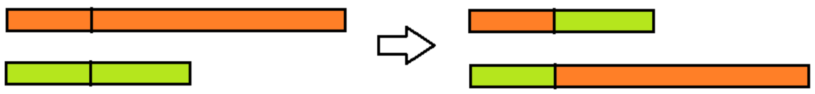


Fig. 7: A graphical representation of the crossover process: the two parents are recombined by cutting them at a common index to create two new offspring

Crossover operator Both populations adopt a simple single point crossover operator: meaning that a cutting point is chosen, both genomes are split at that index and new individuals are created by recombining the resulting sequences (see Figure 7).

One adjustment to the canonical single point crossover is necessary, given that in our case the genome length is variable. Quite simply the cutting point is chosen as a number between [0, smallestN), where smallestN is the length of the smallest of the selected genomes. This ensures that the cut always happens at an index that does exist for both genomes. As it can be noticed also in Figure 8, this operator creates two new individuals that have the same length of the parents. Some more complex crossover operators could create more variation in length sequence; in this first study we decided to use a very commonly used one, since experimentally it doesn't seem to create an issue in how evolution proceeds.

Mutation operator The mutation operator gives each gene a probability 1/l, where l is the genome length, to mutate. This ensures that on average only one gene will mutate but allows for more than one or no mutation to occur. The mutation itself applies one of these operations to the gene g: g changes to represent another direction, g is removed from the genome, a new gene g' (with a randomly chosen direction) is added to the genome after g.

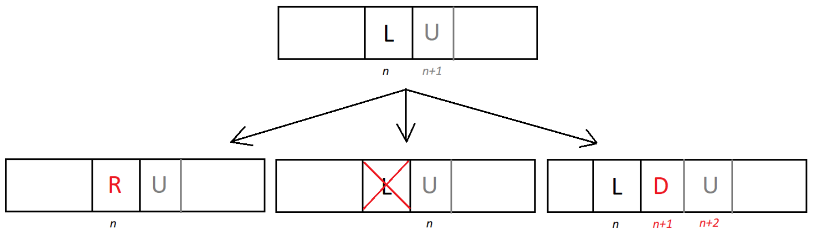


Fig. 8: A representation of the three mutation operators used by our system: the first randomly changes the gene n (in the example from **L**eft to **R**ight), the second removes gene n, and the third inserts a new gene after gene n. Note how the latter two operators change the size of the individual.

Implementation details Other implementation details used to obtain the results discussed later in the paper are here summarised. Our system uses an **elitist strategy**, meaning that a specified number of the best individuals from the current population is allowed to carry on to the next one without being altered.

The parameters used are:

- Population size: 500
- Generation number: 1000
- Elitist factor: 10%
- Mutation rate: 1/n for each gene, where n is the genome length

We do not present an analysis of the running time of the algorithm but, with these parameters, we obtain a solution in ~ 10 seconds.

3.4 Player model

As mentioned already in section 3.4, player modeling can be split in two large families: model-based (top-down) and model-free (bottom-up) [19]. Our approach uses a model-based approach, meaning that we define a model and use the collected player data to determine the *state* of the player. Since the objective is to control the difficulty of the generated puzzles, the represented state is related to the player's ability to solve the previous puzzles.

For each puzzle we collect some information about the player, these are:

$$Info(p_n) = \{tries, sequenceLength, loops\}$$

where p_n is the n-th puzzle.

Based on the information collected from the gameplay, the target length and target loops of the evolutionary system (as described in section 3.3) are adjusted so that:

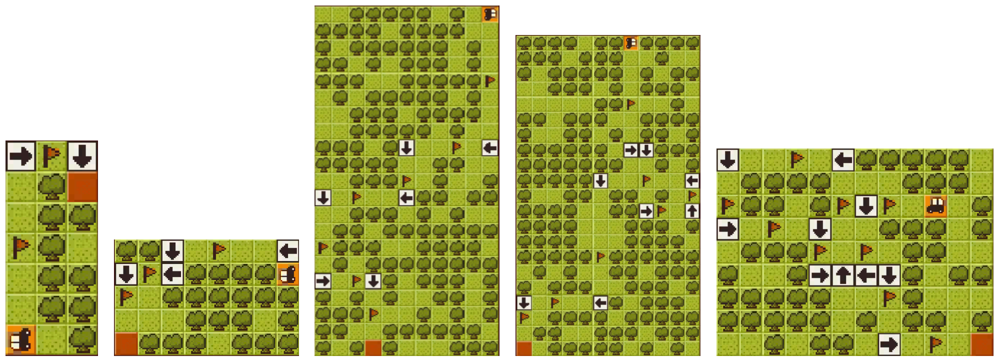


Fig. 9: Example of level progression as can be experienced by the player, starting with a target of $\{3, 0\}$ *(a)* and ending at $\{11, 2\}$ *(e)*

- The sequence target length is increased by 2 if the player solved the last puzzle with less than three tries.
- The target loops are increased by 1 if: 1) $targetLength > 5 * targetLoops + 3$ and 2) the player solved the last puzzle with less than three tries. The motivation behind the first part of the if comes from our definition of a loop (see section 3.1) which requires a specific sequence of five directions to be formed.

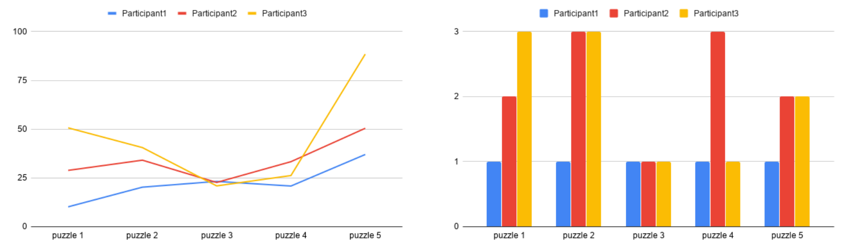
4 Results and discussion

This section provides and discusses generated puzzles for various targets, and provides an exploratory evaluation of the dynamic difficulty of the puzzles. This last one is provided by a pilot user study involving three participants.

4.1 Targeted evolution

Figure 9 shows five generated puzzles of increasing difficulty. As it can be observed, the system is able to produce quite different maps, and as the difficulty increases we see a transition from linear solutions to more complex and sprawling ones that do require deeper thought from the player.

We also would like to discuss Figure 12, this shows a puzzle evolved to have a quite high complexity. The puzzle appears quite complex even with the solution being displayed, it requires a clever use of the tiles, sometimes in quite unintuitive ways, and requires the player to not only think about where the tiles are placed, but which tiles would disappear once the car travels on them. The participants on the pilot study were asked after the test to try to solve this, and all three had to give up, citing that it was a much more complex task than



(a) (b)

Fig. 10: Visualization of the time the participants took in completing the 5 puzzles (in seconds) (a), and amount of tries needed (b)

the ones they tried before. That said, this puzzle is also an example that, while the system creates a solvable puzzle with this type of large solutions, there is room for improvement. In fact, you can observe around the centre of the map that there are two ”Down” tiles that appear one after the other in the sequence, which makes one of them superfluous. Possibly a post-processing task could be added to make sure such ”extra” tiles would be removed from the final puzzle.

4.2 Pilot user study

We developed a pilot user study to lightly assess the functioning of the system. The experiment consisted in the participants playing through five puzzles, and of a short survey afterwards. There were 3 participants, with average age 23.6, of which 2 males and 1 female. These were all university students (which can be assumed to have some knowledge of computational thinking), so it’s not a sample representative of our target audience, yet they could still give us some initial feedback, especially on playability. During gameplay we collected the time and amount of tries needed to complete the puzzles. The survey consisted in some basic demographic questions, and two specific ones about the experiment itself:

– *Did you feel a sense of progression?*
– *Were you familiar with this type of puzzle?*

Both questions were answerable using a 5-point Likert scale.

As can be observed from Figure 10a, there seems to be a common pattern with the time to complete the puzzles decreasing after the first puzzle and then increasing towards the fourth and fifth. We hypothesise that the initial decrease in time is due to the participants getting acquainted to the game/interface, and the following increase due to the increasing difficulty. Looking at Figure 10b we can observe a similar pattern in participants 2 and 3.

All the participants expressed with a relatively large amount of confidence that they perceived an increase in difficulty 11. Participant 1 was the only one that was quite familiar with this type of puzzle, which also is reflected into their performance: always solving the puzzle in one try, and very fast solving time also for the first puzzles. That said, we can observe that their solving time also looked like it was increasing towards puzzle 5.

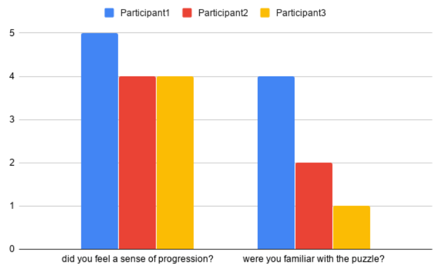


Fig. 11: Participants' answers to the survey

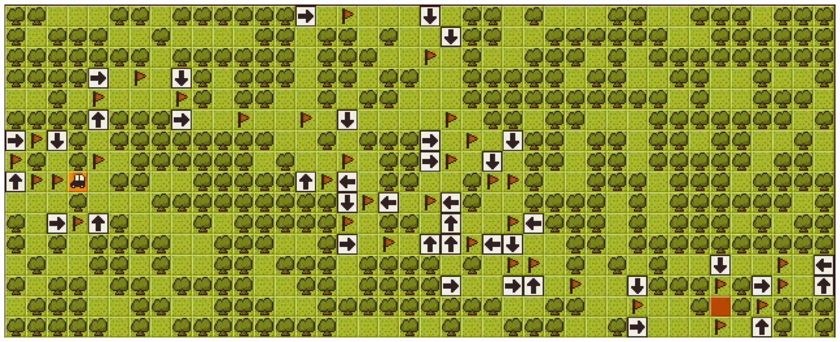


Fig. 12: Example of a generated puzzle with a very large target sequence length (40)

5 Conclusions

This paper presents a novel use of PCG for the generation of puzzles to help pupils learn in a more adaptive ways about some computational thinking concepts, especially sequences. The presented system is able to create a vast variety of puzzles of different difficulties, and can present the player with a progression from simple puzzles to very complex ones.

Of course, this work is still in its infancy, much could be expanded and improved on. In particular the current player modelling is quite simplistic, a more bottom-up approach to creating the model (based on collected player data) would likely lead to more flexible and personalised puzzle generation. We also present a very small user study, which shows some promising results, but that is not nearly enough to reach statistically sound conclusions. Moreover the participants have not been the target audience (young pupils). The next step of this research would be to conduct a more in-depth user-study with our target audience.

That said, we do believe there is a lot of potential in this and similar systems since they would solve one of the main problems educators encounter with using gamification tools for teaching/learning: the content is almost always static and limited! This use of PCG would allow the educator to use the tool for longer and possibly even ask it to generate specific content, without requiring the educator to have either design or programming skills (which is usually the case). Another thing we want to highlight, is that our system can generate a variety of different looking levels, even if they have the same solution, this might be interesting from an educational point of view, to see how groups of students might be able to figure out collectively that they can abstract problems that might look different to the same one.

In conclusion, we presented a system for generating levels of a puzzle game with adaptive difficulty. The difficulty curve is itself controlled by the player performance, and the game was designed with the objective to be able to teach some computational thinking concepts. The system is powered by a constrained multi-objective optimisation evolutionary method.

References

1. Billinghurst, M., Duenser, A.: Augmented reality in the classroom. Computer **45**(7), 56–63 (2012)
2. Brennan, K., Resnick, M.: New frameworks for studying and assessing the development of computational thinking. In: Proceedings of the 2012 annual meeting of the American educational research association, Vancouver, Canada. vol. 1, p. 25 (2012)
3. Chafekar, D., Xuan, J., Rasheed, K.: Constrained multi-objective optimization using steady state genetic algorithms. In: Genetic and Evolutionary Computation Conference. pp. 813–824. Springer (2003)
4. Colton, S.: Automated puzzle generation. In: Proceedings of the AISB'02 Symposium on AI and Creativity in the Arts and Science. Citeseer (2002)
5. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: Nsga-ii. IEEE transactions on evolutionary computation **6**(2), 182–197 (2002)
6. Deb, K., Pratap, A., Meyarivan, T.: Constrained test problems for multi-objective evolutionary optimization. In: International conference on evolutionary multi-criterion optimization. pp. 284–298. Springer (2001)
7. Fernández-Vara, C., Thomson, A.: Procedural Generation of Narrative Puzzles in Adventure Games: The Puzzle-Dice System. In: Proceedings of the The Third Workshop on Procedural Content Generation in Games. pp. 12:1–12:6. PCG'12, ACM, New York, NY, USA (2012). https://doi.org/10.1145/2538528.2538538, http://doi.acm.org/10.1145/2538528.2538538, event-place: Raleigh, NC, USA

8. Gilboy, M.B., Heinerichs, S., Pazzaglia, G.: Enhancing student engagement using the flipped classroom. Journal of nutrition education and behavior **47**(1), 109–114 (2015)
9. Hendrikx, M., Meijer, S., Van Der Velden, J., Iosup, A.: Procedural Content Generation for Games: A Survey. ACM Trans. Multimedia Comput. Commun. Appl. **9**(1), 1:1–1:22 (Feb 2013). https://doi.org/10.1145/2422956.2422957, http://doi.acm.org/10.1145/2422956.2422957
10. Isaacs, A., Ray, T., Smith, W.: Blessings of maintaining infeasible solutions for constrained multi-objective optimization problems. In: 2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence). pp. 2780–2787. IEEE (2008)
11. Khalifa, A., Perez-Liebana, D., Lucas, S.M., Togelius, J.: General video game level generation. In: Proceedings of the Genetic and Evolutionary Computation Conference 2016. pp. 253–259. ACM (2016)
12. Kimbrough, S.O., Koehler, G.J., Lu, M., Wood, D.H.: On a feasible–infeasible two-population (fi-2pop) genetic algorithm for constrained optimization: Distance tracing and no free lunch. European Journal of Operational Research **190**(2), 310–327 (2008)
13. O'Kane, L.: A Computational Thinking Puzzle (Jul 2016), http://www.icompute-uk.com/news/computational-thinking-puzzle/
14. Ray, T., Kang, T., Chye, S.K.: An evolutionary algorithm for constrained optimization. In: Proceedings of the 2nd Annual Conference on Genetic and Evolutionary Computation. pp. 771–777. Morgan Kaufmann Publishers Inc. (2000)
15. Scirea, M., Togelius, J., Eklund, P., Risi, S.: Metacompose: A compositional evolutionary music composer. In: International Conference on Computational Intelligence in Music, Sound, Art and Design. pp. 202–217. Springer (2016)
16. Scirea, M., Togelius, J., Eklund, P., Risi, S.: Affective evolutionary music composition with metacompose. Genetic Programming and Evolvable Machines **18**(4), 433–465 (2017)
17. Shaker, M., Sarhan, M.H., Naameh, O.A., Shaker, N., Togelius, J.: Automatic generation and analysis of physics-based puzzle games. In: 2013 IEEE Conference on Computational Inteligence in Games (CIG). pp. 1–8 (Aug 2013). https://doi.org/10.1109/CIG.2013.6633633
18. Wing, J.M.: Computational thinking. Communications of the ACM **49**(3), 33–35 (Mar 2006). https://doi.org/10.1145/1118178.1118215, https://doi.org/10.1145/1118178.1118215
19. Yannakakis, G.N., Togelius, J.: Experience-driven procedural content generation. IEEE Transactions on Affective Computing **2**(3), 147–161 (2011)