Web App for Beginners Coding Gamified Learning

João Lírio, Murilo Couceiro, José Pedro Baptista
201705254 , 202003377 , 201705255
$FEUP - TACS$

***Abstract*— Technology has and will continue to have a fundamental aspect in our lives. Coding is being introduced in younger crowds, and to assist the learning process, more accessible types of learning are in development. This paper presents a game for this purpose. It is a game for finding a path in a map with blocks to move the character that translates to the code that would perform the same moves. This code is then generated from the blocks and run, in order to interact with the game. Some other functionalities, such as code and path optimization, were also implemented, and were accomplished using a parser combinator and genetic algorithms, respectively.**

I. INTRODUCTION

Technology plays an essential role in almost every activity of today children's lives. With this younger generation starting to look at technology, it is important to create tools to learn interactively and easily.

This paper presents an application for this purpose, specifically for programming. This app is a game where a player must reach a specific destination, avoiding an enemy and obstacles through the map. Users code by a drag-and-drop system, placing blocks in order and getting the equivalent piece of code generated for this action. The rest of the document specifies, in this order, the blocks and code generator associated for each, the implementations of the genetic algorithm for path.

II. DEVELOPMENT ENVIRONMENT AND TECHNOLOGIES USED

The application was developed using web technologies such as Typescript, React, React-Konva and Node.js.

We chose to use Typescript as the main programming language for this project, because it adds static typing on top of Javascript, which makes it easier to detect and solve some bugs in compile time that otherwise would only be found while executing. As Typescript code can be transpilled to Javascript, it can also be used for the same purposes of Javascript. React was chosen to develop the user interface because everyone in the group was familiar with it, and Typescript has type support for it. React-konva is a package that integrates konva, the Javascript 2d canvas library, in React and was used to display the game.

III. CODE GENERATION

Being the objective of the app presented in this paper to help users learn the basics of coding, there must be a way of showing what code they would have to write and execute in order to perform the moves represented by the selected blocks. The interface is very simple and intuitive, as this is for beginners. In this section we explore our generation of code with each block.

A. Available Blocks and parameters

The available blocks translate to very simple in-game actions, and point to some aspects of coding such as methods and loops. The available blocks are:

Block name	Description
Move	Moves the player 1 tile forward
Turn	Turns the player
Repeat	Repeats the inner actions n times
Wait	The player stays still

Block name	Parameter : type
Move	-
Turn	rotateTo : ”right” or ”left”
Repeat	nIterations : number
Wait	-

B. From the blocks to the code

The generated code is solely based on the blocks the user selected and its parameters. The Repeat block may hold more blocks inside itself and thus, the code it generates includes the code from its inner blocks. In some blocks, such as Turn and Repeat, the user must fill in its parameters to make it valid.

Assuming every block is valid, after generating the code from the blocks the user placed, we end up with a working algorithm that interacts with the game and/or player. In order to achieve this behaviour, we created a function in our Game class for each block type (with the same parameters). For instance, a Move block would generate code that would call the movePlayer function of the game object, i. e. ”this.g.movePlayer()”. Every other block follows the same logic to generate its code, with the exception of the Repeat block. As this block corresponds to a loop, it simply generates a for loop using its parameter - nIterations - and for its body, it appends the code generated by each of its inner blocks.

At this point, our main objective had already been achieved, however, in order to apply some more concepts we learnt, we decided to optimize the generated code.

1) Parser Combinator: To optimize the code, we created a parser combinator using the *Parsimmon* library. The parser receives a line of code that corresponds to a block of code, and it can be a forward movement, a turn, a wait or a repeater

of some of this movements. When the parser receives an expression, it detects what type of expression it is and returns the block of code that corresponds to the expression. Although this step was not strictly necessary (we could use the code blocks we already had), we decided to parse the generated code in order to apply another concept that we were told.

2) Optimizations: After some time testing the application and watching the code being generated, we noticed that sometimes there were a lot of repeated code statements being generated. So, we decided to optimize this code by using for-loops. Basically, we made a function that analyzes every block of code generated and when it detects a sequence of repeated lines of code, creates a for-loop with that repeated line of code inside of the "for", substitutes that code by the loop and make it run the number of times it was repeated.

IV. Genetic Algorithms

After completing the path required to complete the game, the player can try to optimize this path. When optimizing a path, we can consider the traditional algorithms that compare every possibility to find the best solution. However, these algorithms can be very time-consuming. This problem opens up space for another set of algorithms, such as applying a genetic algorithm to perform this task. In general, a genetic algorithm has five essential components : (1) an encoding method that is a genetic representation (genotype) of solutions to the program. (2) A way to create an initial population of chromosomes. (3) The fitness function. (4) The genetic operators (crossover and mutation) that alter the genetic composition of offspring during reproduction [1]. For this case, creating the initial population has the advantage of having the user-created path. The initial population is 100 times the solution.

A. Genotype

The genotype generated for the algorithm is a list of characters. It represents the path by the first letter of the block's name, except for the turn block with the first letter of its direction. The block move turns into the character M. Turn block, right and left, turns into R and L, respectively. Finally, the wait block turns into a W. When there is a for block, the elements inside it are unfolded, so the genotype created contains the path created by the user move by move.

B. Fitness Function

We want to optimize the path, so the fitness function must be inverse to the path size. Bigger the path, the smaller the fitness of the genotype. To classify the solution's fitness, we first check if it reaches the final destination. If it does not, the solution fitness is zero, and it has no interest. If it does, we calculate the inverse, $1/pathsize$, following the logic mentioned earlier.

C. Mutation Function

Mutations are what start to create changes to the path. For this purpose, we wanted to shorten the path, so one of the options for this mutation was to remove a random element. One change in the path can make the algorithm reach its destination or not, so another option was replacing a random element in the path. Adding an element can also help, creating different possible solutions.

One of these three options is selected randomly, so each mutation results from one of these operations. Although their selection is random, they do not have the exact probabilities. There is a 40% chance of removing or replacing an element and a 20% chance of adding one.

D. Crossover Function

The crossover function allows the creation of a set of different mixed solutions from the mutations. Our crossover function receives two paths and crosses them. It does this by going through one of them and, with 85/15% chance, stays with its element or replaces it with the other block's element, respectively.

V. CONCLUSIONS

In conclusion, we think the game is attractive and easy to understand, but most importantly, it allowed us to apply some of the contents learned in the course unit. For beginners, it allows the easy understanding of how small pieces of code should be written and some functions and objects understanding by the lines of code generated. Some other concepts could have been implemented, such as if statements or while loops. Although the current implementation allowed rapid implementation of these concepts, we have focused our development on additional concepts other than code generation. Code optimization is a relevant aspect utilized in this game. Optimizing the code written, for example, with for loops, is helpful for users who do not use it and should to simplify its code. A parser combinator was developed to check the code generated to help the optimization. However, it may have other utilities in the future, such as allowing the user to write its own code and then check its transaction to blocks. Unmoving, a genetic algorithm was also developed. In some cases, it does find a better solution, but not the most practical one. Better results could have been reached if we had a more radical mutation and crossover method.

These methods and concepts were applied in this game context but could be very important for more advanced code optimization, so we believe that the work developed has significant importance.

References

[1] A. Y. Hamed, "A genetic algorithm for finding the k shortest paths in a network," *Egyptian Informatics Journal*, vol. 11, no. 2, pp. 75–79, 2010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S111086651000023X