Tetropica - Project report

A project about creating an automatic puzzle generator of solvable levels in the game concept Tetropica.

DM2904 - Individual Course in Media Technology

Hannes Vestberg

Abstract - English

Tetropica is a puzzle game concept that challenges the player to solve levels of grid filled tiles using different sets of Tetris blocks. The game created within this project utilizes a level generator of solvable levels to be able to deliver a large range of challenges for the player to solve. The generator was constructed in modules and was responsible for both the creation of a level and to ensure that it was solvable. The solutions to the puzzles were made using a distance matrix and different block placement algorithms to both ensure that a solution was found and to find the best solution to any given level. The resulting work in this project a fully functional generator of solvable levels that for the most part generated levels with only a single solution.

Abstract - Svenska

Tetropica är ett pusselspelskoncept som utmanar spelaren att lösa banor som består av brickor i ett rutnät med hjälp av tetrisblock. Spelet som skapades i detta projekt använder sig utav en generator av lösbara banor som ger spelaren en stor mängd olika utmaningar för spelaren att lösa. Generatorn var konstruerad i moduler och var ansvarig för att både skapa olika banor och att se till så att de var lösbara. Lösningarna till pusslen skapades med hjälp av en distansmatris och med algoritmer som kalkylerar olika placeringar av tetrisblock för att både se till så att en lösning hittades men även för att hitta den bästa lösningen till en given bana. Det resulterade arbetet inom projektet var en fullt funktionell generator av lösbara banor som för det mesta genererar banor med endast en lösning.

1. Introduction

Puzzle games are popular and one of the most recognized genre of games. But one problem with the creation of puzzle games is the time consuming task of creating new content for the game manually. With complex puzzle rules this task becomes even more difficult, and a puzzle generator may be wanted. The advantage of using a generator to create game content is that they are often vastly more efficient than a human. A puzzle that a human may need minutes to an hour to create, a generator may only need fractions of a second to generate. However, creating a puzzle is often easy, but ensuring that the level can be solved is a much harder task. Therefore the generator must have the criteria to generate only solvable levels to ensure that the content can be used in a game.

Puzzle games can differ in several ways, one being the possible solutions a given puzzle can have. Sudoku is a popular puzzle game where for each given board, only one solution is valid. A simple level generator for such a puzzle would be to start with a finished board and then systematically remove numbers on each row until the desired difficulty is reached, while still leaving enough numbers to be able to solve the puzzle. This generator would not only be fast but also ensuring that every outputted level would be solvable.

But some puzzles does not only have a single solution and will require a more complex generator. Due to having multiple solutions the generator may have a harder time to start with a solved puzzle and then add the challenges of the puzzle, but may rather need to generate a whole new puzzle and try to solve it after the creation. This way the generator limits the number of different solutions to a given puzzle to better generate puzzles with a desired difficulty. A higher count of solutions to a puzzle generally equates to an easier puzzle. Depending on the type of puzzle game, this behavior may not be wanted, and the solution count of a puzzle should always be as low as possible so that other parameters of the puzzle will be the deciders of the puzzle difficulty.

1.1 Tetropica game concept

Tetropica is a 2D puzzle game concept developed during this project that involves the user solving puzzles using Tetris blocks. It takes game concept inspiration from popular traditional games such as Tetris [1] and Bomberman [2], but also graphical and layout inspiration from recent games such as Unblock Me [3] and Candy Crush [4]. In Tetropica, the user will for each level be presented with a tile filled grid with variable sizes and a set of Tetris blocks. And in contrast to the original Tetris, by placing blocks the user removes tiles from the grid instead of adding them. The grid always starts with two empty tiles in opposing sides, one start point and one end point. To solve the puzzle the user must make a path from the start point to the end point with the set of Tetris block given to the player. The user can only place a block adjacent to any empty tile, and some tiles in the grid have special functionalities, for instance hindering the player to place a block over it, making some paths invalid.

The game includes two modes, one "Puzzle mode" with tested levels put in an appropriate difficulty curve, and one "Endless mode" with unlimited automatically generated solvable puzzles. Levels in the "Puzzle mode" are created using the generator of solvable puzzles as well and stored as seeds to the generator.

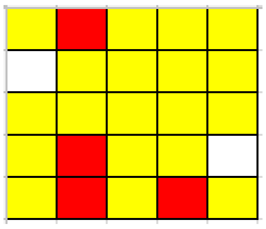


Figure 1. One typical Tetropica puzzle, with the yellow tiles destructible and the red tiles indestructible. The user must place Tetris blocks adjacent to the empty white tiles to make a path between the two starting empty tiles.

The automatic generation of solvable levels with the best solution given a set of blocks is an advanced NP-problem that often requires an approximation solution, and is the main area of research within this project. To be able to generate a solvable level you need an algorithm that understand how to play the game.

1.2 Motivation

An automatic content generator can help to drastically reduce the amount of work needed to expand the game content. But for the generation to be considered effective, the generator must also output quality content close to or at the same standard as if the content was designed by a designer. This project aims to create a level generator that both creates solvable levels while still ensuring challenging gameplay. What is to be considered challenging within the scopes of this project will be determined by how many solutions are found to be available for each puzzle related to their grid size, with a lower solution count indicating a harder and a more desired challenge for the level.

The research to create an automatic generator for the puzzle game Tetropica will give insight into how to create a puzzle game with a complex ruleset and how to make algorithms that take these rules into consideration when generating solvable levels. The results of this research may be used in future work as a reference to how such an algorithm may be functioning.

1.3 Research Question

"How can an automatic level generator of solvable levels be created in the puzzle game Tetropica?"

1.4 Limitations

The report will not cover the design of the game Tetropica, but only the development and implementations of the already finished concept of the level generator of Tetropica.

2. Related Work

Procedural content generation systems are used extensively in game development today [5]. As the gaming player base keeps expanding, game developers face a large scalability challenge to meet the content demands of the players of their games. Procedural content generators may address these challenges by assisting the designers and developers in the form of automation tools but can also be embedded into games as fully automated content generators. They are often produced to ease the creation of a large space of artefacts such as in-game items, equipment and levels in a quantitative manner that artists and designers do not have the time to individually create [6]. But they can also open up new possibilities of smart content design where the game analyses a player's playstyle and generate content to maximize enjoyment of the played level in real time [7]. This would be almost impossible for a human designer to do, and even if it were possible, it would require extreme amount of manual labor. So in this respect the system is not only replacing the designer, but also outperforming them in time, efficiency and player satisfaction aspects.

A procedural content generator can be very useful in the creation of content for a puzzle game [8], however work must be put into the generator to ensure that the resulting output of such a generator will be fun and challenging for the player. In contrast to many other uses for procedural content generators where quantity is the main focus, a puzzle game often have nothing except its content, and therefore the quality of the output from the generator is at least of equal importance. A game that has this problem is the game Refraction.

Refraction [9] is a mathematically educational puzzle game where the player solves entangled spatial and mathematical challenges using complicated puzzle design. Creating these levels by hand would be too much work to ensure that the puzzles always are solvable and no so called "shortcut solutions" exist, meaning unwanted solutions. Therefore the creation of a level generator would be appropriate in this game, but due to the complicated puzzle design of the game many criteria are set on the generator, including making challenging puzzles at the same quality as if a designer would have made them. Tetropica faces many of the same obstacles as in the Refraction project.

The area of procedural content generators is well researched and considered one increasingly important area in human-computer interaction design [10], but level generators in games that ensure a solvability guarantee on its output levels are a more advanced topic that is less researched on [9]. Even less researched are the generators that guarantee that output levels are challenging as well, with only a single solution.

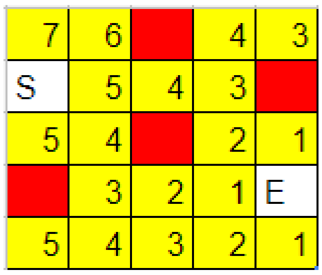
3. Methods

The game Tetropica is designed to be developed in the game engine Unity in the programming language C#, with mobile platforms in focus, but with a user interface that can also be used for PC. The development of the game consists of typical stages associated with game development: the design document, user interface sketches, design of the graphical objects, game functionality development, animations and visual effects and special for this project will be the level generator development. Within this section, only the development of the level generator will be covered as it is the only part of the development process relevant to answering the research question.

The level generator will consist of several modules that together make up the generation of a solvable level. The first part of these modules is the actual map generator that generate the grid of tiles that the player must find its way through. The generation of a map in games that utilize procedural map generators can be done in at least two different ways. Either the map generator generates a path from the start point to the end point before populating the map with obstacles or the other way around, i.e generates the obstacles first and then try to find a path through the level. The first option is generally simpler as no additional calculations are needed once the path has been placed. The only criteria is that the generator cannot place obstacles on the path. Tetropica however uses the second method to try to limit the amount of sub solutions to a level. If Tetropica used the first method there is a much greater risk that the obstacles are placed in a way that enables the user to use another path through the obstacles, which is not wanted as it reduces the wanted difficulty of a level.

To first generate the grid of tiles, the generator takes in game settings as input (size of grid and the count of obstacles) together with the seed of the map as an integer. The algorithm then takes a random position in the grid and rolls the dice whether the tile should be a normal tile or an obstacle (with the game settings in mind). This continues until the grid has been totally filled, except for two tiles on opposite sides of the grid. These are the start and end tiles that the players need to find a path through. Notice that this algorithm does not ensure that the level is solvable, but simply arbitrary generates a grid of tiles that could be impossible to solve. To check if the map could be solvable the grid is sent to the next module of the level generator called solution generator.

The solution generator is responsible for three different tasks: first it must check if the input grid can be solved, secondly it must tell the map generator to generate a new grid of tiles if no solutions are found, and thirdly it must calculate the best solution it can given a certain time limit and store it to be available to the player when the game starts. The solution generator takes the newly created map and a seed as inputs and starts calculating solutions to the map. The first step in the process of finding a solution to the grid is to generate a distance matrix. The matrix will be of the same size as the grid of tiles and each element in the matrix will contain a value. The value will correspond to the distance from the end tile, making the matrix element increase for each tile away from the end tile.

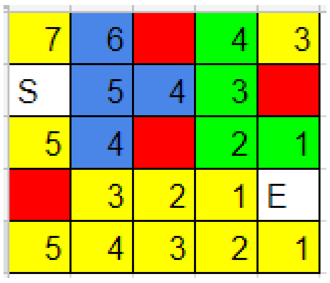


7	6	🟥	4	3
S	5	4	3	🟥
5	4	🟥	2	1
🟥	3	2	1	E
5	4	3	2	1

Figure 2. Displaying how the distance matrix would look like for the given grid. With S representing the starting tile and E representing the end tile.

The next step of the solution generator is to find a way through the grid with a randomly given Tetris block with the use of this distance matrix. The generator tries all available placements and rotations of the block in the grid, while storing the total values of all the elements in the distance matrix the block occupies. After all valid placements have been tested and stored, the generator picks the best one with the lowest value and applies it to the map. Then a new block is generated to be placed. This process continues until a path is found from the start tile to the end tile, or until no more valid placements can be found, where the solution generator resets the map to its original state and tries again with different tiles. Between each placement of blocks, the algorithm checks for a path from the start tile to the end tile using a recursive depth-first search of the empty tiles in the grid. This algorithm ensures that if there are currently multiple found paths from the start tile to the end tile, it will always use the smallest path through the empty tiles in the grid.

The generator tries to find solutions to the map ten times and stores a valid solution where it was found. Once all tests have finished the solution generator then picks the solution or path with the lowest total value i.e. the most efficient path from the starting tile to the end tile. This level has now been proven to be solvable using the blocks present in this path. The blocks are then saved to the level and will be given to the player once the game starts. The path that solves the level will also be stored with the level so that the player may receive hints where to put the blocks if they are stuck.



🟨 7	🟦 6	🟥	🟩 4	🟨 3
S	🟦 5	🟦 4	🟩 3	🟥
🟨 5	🟦 4	🟥	🟩 2	🟩 1
🟥	🟨 3	🟨 2	🟨 1	E
🟨 5	🟨 4	🟨 3	🟨 2	🟨 1

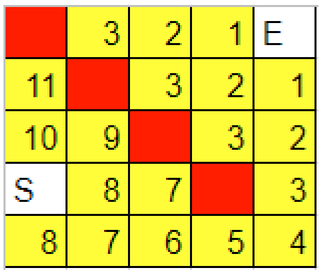
Figure 3. Displaying one valid solution to the level with the use of two Tetris blocks colored in blue and green. The total value for this solution is equal to all the values the blocks occupy in the distance matrix. In this instance it is 6 + 5 + 4 + 4 + 3 + 4 + 2 + 1 = 29.

4. Results

The implementations of the method explained above was functional but required some iterations to work efficiently. One problem that was found with the first implementation was that many of the levels had more than one solutions, often with less than all the blocks given to the player at the start of the level. The levels contained sub solutions within its solution. This was due to the way the solution generator generated the solutions with random blocks. It became apparent that it was a mistake to let the generator randomly select a block to place on the map where the player had access to all of the blocks to place at any given time in a level. The solution generator tried to solve the level with another ruleset than the player, changing the difficulty of the level. One common anomaly of this algorithm was the placements of Tetris blocks on abnormal side paths that often did not contribute to the path to the end tile. The algorithm would in such an instance try to place a randomly selected Tetris block on a path to the end tile, but failed, forcing the placement to a spot as close to the end tile as possible but with no guarantees that the block would be useful.

This issue was solved through an iteration on how the solution generator algorithm functioned. Instead of generating a new block, all Tetris blocks were instead tested with all available placements and rotations to match the perfect Tetris block to solve the level. This change to the algorithm proved to work more efficient than the first implementation since all possible Tetris blocks were tested, increasing the possible paths to the end tile. This new version of the solution generator was more calculation heavy due to having to test all Tetris blocks instead of the previous version where it was only one block. This required a compromise to optimize the generation, and the number of total solutions tested on each level were reduced from the previous ten tries to only one. Even though this reduction may seem extreme, the quality of the solution of the new generator would not change if it were to regenerate a new solution, since all Tetris blocks are tested and therefore the best possible solution to a level is selected on every try.

One other problem was found with the distance matrix. The algorithm that created the distance matrix did not take the obstacle tiles into consideration, and regarded these tiles as ordinary ones. This resulted in some strange placement behavior from the solution generator where obstacle tiles blocked of major parts of the level. The solution to this problem was to change the algorithm that created the distance matrix to a breadth-first iteration from the end tile throughout the grid, not passing through obstacles. This meant that the solution generator could place blocks more efficiently with regards to larger groups of obstacles.



🟥	3	2	1	E
11	🟥	3	2	1
10	9	🟥	3	2
S	8	7	🟥	3
8	7	6	5	4

Figure 4. Displaying how the new distance matrix algorithm calculated the distance with obstacles into consideration.

After these iterations the resulting work in this project is the implementation of the game concept of Tetropica with an automatic generator of solvable levels. The game contains the two game modes "Puzzle Mode" with selected levels with a predefined difficulty setting and the "Endless Mode" with a randomly generated level with a user-selected difficulty. The game settings range from "Easy" with a 5 by 5 grid, "Medium" with a 7 by 7 grid and "Hard" with a 9 by 9 grid. All levels are generated with an integer seed and a difficulty setting, which means that a map with two different difficulty settings could use the same seed and look different, totaling a number of 2,147 483 647 levels per difficulty setting in the game.

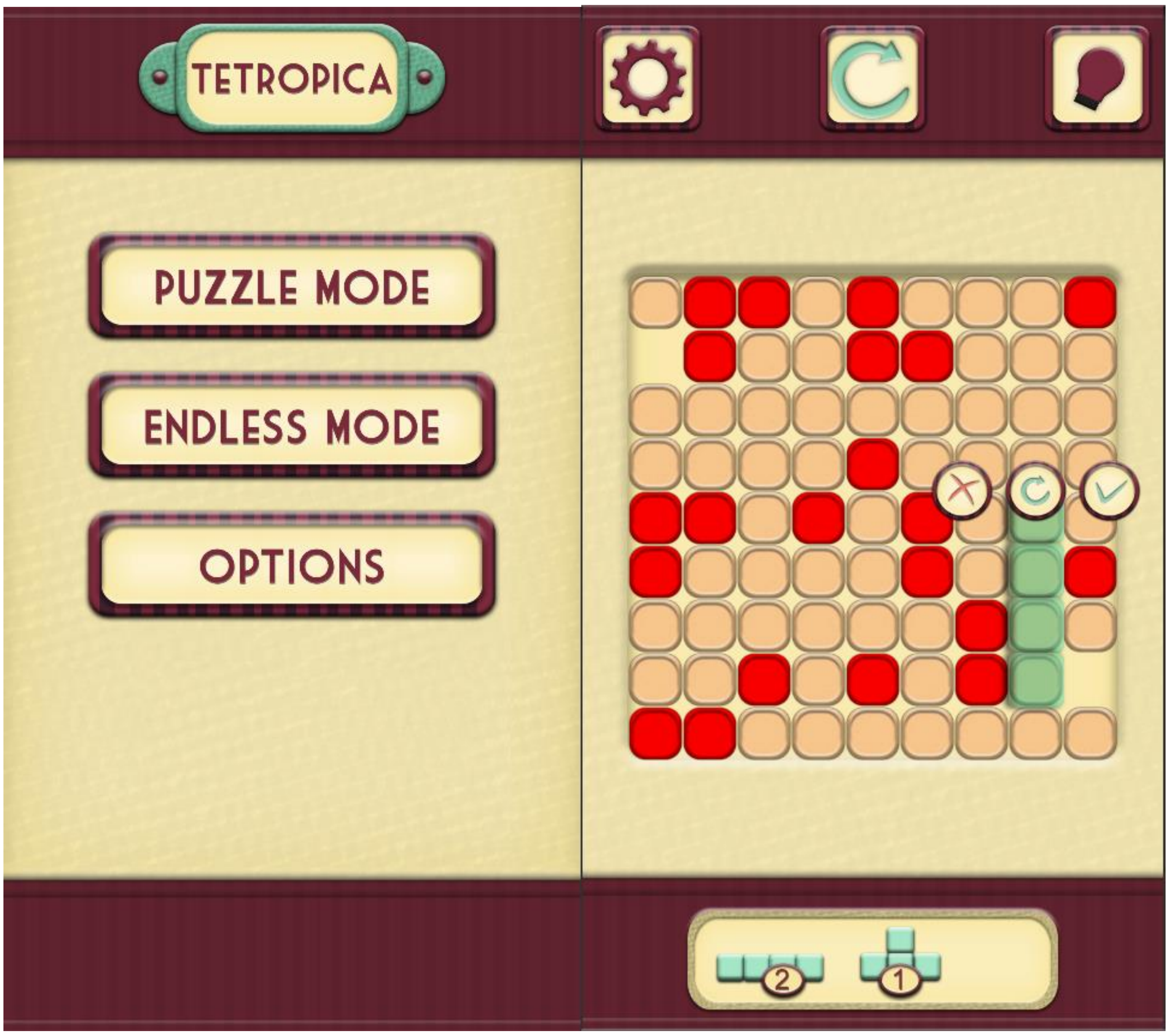


Figure 5 and 6. Displaying the main menu and gameplay of the game Tetropica during a block placement.

5. Discussion

In its current state, the level generator does generate solvable levels that, for the most part, only have a single solution. Some cases of levels with multiple solutions do however exist on some of the smaller levels. This is due to the solution generator scoring system of placements using the distance matrix not being able to project the best possible placement as a human would on small distances. As many of the possible placements on a small map would have the same score, the algorithm does not check each placement if a solution is found if a block would be placed in that specific position. One possible fix for this behavior that's not currently implemented in the game is to check after each block have been tested with all possible placements, and the best ones stored, is to also check if there are two or more placements with the same score and test if either of those placements would solve the level and prioritize those who do. This implementation would end up complicating the algorithm a bit and slow down the process of level generation, so optimizations of the code may be needed for it to work efficiently.

But for larger maps of sizes 7 to 9 in dimension, the algorithm does generate levels with rarely any other solution than the one the solution generator found. This makes these map dimensions optimal for the current state of the game. The computing time for creating a map ranges from around 20 milliseconds to up to 800 milliseconds, on a desktop computer. On some seeds and due to some unfortunate random outcomes, the generation of levels takes a very long time, up to several seconds on some of the larger maps. On these seeds the grid of tiles seems to be generated without any solution and is regenerated several times before one gets accepted. Because the generator is seeded means that for every try to generate that particular level, the same procedure will run and it will fail until it finds the first acceptable map using that seed. This behavior could be optimized by checking for a possible path right after the grid of tiles has been generated to avoid trying to find a solution to a level that is impossible to solve.

6. Conclusion

The implementation of the seeded generator of solvable levels does work as intended after a few iterations on how the generation and placements of blocks were processed inside the algorithm. The generator generates levels with only a single solution for the most part and therefore fulfill the requirement that the generator both should generate solvable levels but also levels that are challenging for the player. The optimal map size with regards to the number of possible solutions to the level and the occurrences of long generation times on larger maps are 7 by 7 to 9 by 9 grid sizes. Maps smaller than these dimensions have possibilities of subset solutions within the given Tetris blocks but could be fixed by and iteration on how the solutions are generated, but will have a negative effect on generation computing time.

Some optimizations may be needed for the algorithm to work efficiently on mobile devices as the generation takes up to a second on a desktop computer. But the generation of solvable levels still is fully functional on these devices.

7. References

[1] "Tetris," Alekséj Leonídovitj Pázjitnov, 1984. [Online]. Available: https://tetris.com/play-tetris. [Accessed 13 December 2017].

[2] "Bomberman," Hudson Soft, 1983. [Online]. Available: https://en.wikipedia.org/wiki/Bomberman_(1983_video_game). [Accessed 13 December 2017].

[3] "Unblock Me," Kiragames Co., Ltd., [Online]. Available: https://itunes.apple.com/us/app/unblock-me-classic-block-puzzle-game/id315019111?mt=8. [Accessed 13 December 2017].

[4] "Candy Crush," King, [Online]. Available: http://candycrushsaga.com/en/. [Accessed 13 December 2017].

[5] M. Hendrikx, S. Meijer, J. Van Der Velden and A. Iosup, "Procedual content generation for games: A survey," ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM), New York, 2013.

[6] J. Fisher, "How to Make Insane, Procedural Platformer Levels," Pwnee Studios, 2012. [Online]. Available: https://www.gamasutra.com/view/feature/170049/How_to_Make_Insane_Procedural_Platformer_Levels_.php. [Accessed 2017 December 13].

[7] D. Ashlock, "Automatic generation of game elements via evolution," Computational Intelligence and Games (CIG), Copenhagen, Denmark, 2010.

[8] S. Colton, "Automated Puzzle Generation," Division of Informatics, University of Edinburgh, 2002.

[9] A. M. Smith and E. P. Z. Butler, "Quantifying over Play: Constraining Undesirable Solutions in Puzzle Design," Center for Game Science, Dept. of Computer Science & Engineering, University of Washington, 2013.

[10] G. N. Yannakakis and J. Togelius, "Experience-Driven Procedual Content Generation," *Transactions on Affective Computing,* vol. 2, no. 3, pp. 147-161, 2011.

8. Figures