Performance Improver for Block Generator in 1010! Using AND-OR Tree

Livia Andriana Lohanda, Samuel Lukas, and Irene Astuti Lazarusli

***Abstract*—1010! is a tile-matching game for Android and iOS. Players aim to fill a board's entire row or column using blocks from block holder to empty the board, thus create more space for the next blocks. These blocks are randomly-generated, thus occasionally block holders will hold blocks that cannot fit into the board by any combination and cause the game to end. This loss can be avoided using a two-stage performance improver consists of validator and unfit block changer. Validation begins by creating an AND-OR tree as a basis for validation's flow, taking block set's permutation and block's position to the board into account. If the block set is deemed unfit, block changer algorithm will calculate the heuristics for each block based on frequency and location of failure in the validation stage, then change the block with the highest heuristic with random block and redo the validation stage with new block set. After measuring performance on five gameplays, running only the validation stage results in an average of 3.95 ms, while doing both stages ran in 14.48 ms on average.**

***Index Terms*—AND-OR tree, mobile game, random generator, tile-matching.**

I. Introduction

A. About 1010!

1010! is a tile-matching mobile game from Gram Games, released at Play Store and App Store since August 2014 [1]. It was highly similar to Tetris [2], but instead of forcing players to tackle falling blocks with increasing speed into the board, 1010! provided three blocks in a block holder which can be fitted anywhere in the board. Players will receive points for inserting filling board with blocks and clearing completed rows or columns. The game will be over if player cannot insert any blocks into the board. A sample game can be seen in Fig. 1.

One of the main problems in 1010! is when a block holder contained at least one block that could not fit into the board, no matter how players strategize to put them in, thus causing a game over situation. The problem lies in the block generator, which randomly generates three blocks every time the block holder gets empty without any validator to check its fitness to the board. This cause a flaw in the game design.

Some solutions that has been proposed is to add undo button [3], which was rejected by other players because it reduce the game difficulty and very abusable. Other recommendation was weighted random [4], which has been used in Gram Games' original version of 1010!. This paper discusses another approach: include validation after block generation, which trigger the generator to change the unfit block based on heuristic point calculation.

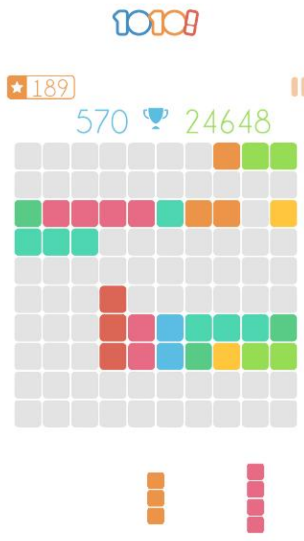


Fig. 1. A screenshot of 1010! in iOS platform.

B. Scope of Analysis

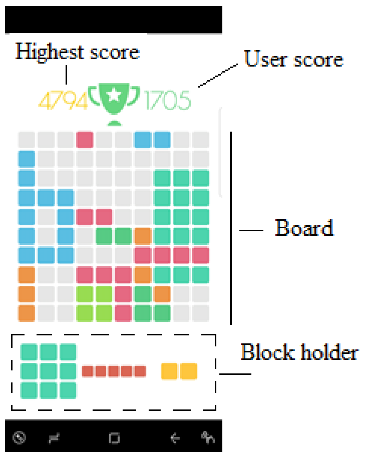


Fig. 2. A screenshot of Klooni 1010! in Android platform.

The improver will be developed based on an open source version of 1010! called Klooni 1010! initiated by LonamiWebs and hosted in their GitHub [5], as seen in Fig. 2. This project is written in Java using libGDX, a cross-platform open source Java-based game development framework, usually used to develop 2D Android games [6].

The validation stage will use AND-OR tree as its skeleton, and the improver only focuses on ensuring every member of any randomly generated block set is solvable, without considering score in its decision making. Also, the improver

Manuscript received May 3, 2019; revised July 22, 2019.

Livia A. Lohanda, Samuel Lukas, and Irene A. Lazarusli are with the Informatics Department, Universitas Pelita Harapan, Tangerang, Indonesia (e-mail: livia.lohanda@uph.edu, samuel.lukas@uph.edu, irene.lazarusli@uph.edu).

DOI: 10.7763/IJCTE.2019.V11.1254

must be able to change unfit block with another block that makes the block set solvable.

Lastly, since 1010! is a mobile game, the improver must work under 100 milliseconds (ms).

II. Literature Review

A. Puzzle Game Design

According to Schell, there are ten principles that makes a puzzle game design as a good design [7, pp. 243–251]:

1) Make the goal easily understood
2) Make it easy to get started
3) Give a sense of progress
4) Give a sense of solvability
5) Increase difficulty gradually
6) Parallelism let the players rest
7) Pyramid structure extends interest
8) Hints extend interest
9) Give the answer
10) Perceptual shifts are a double-edged sword

Out of ten principles, 1010! has implemented six first principles. 1010! has a clear goal and easy to understand with instructions available in the download page. The score is also updated instantly after inserting one block into the board, so the progress is visible. Principle #4 and #5 are represented by the block sets; fitting all members into the board and clearing blocks give players a sense of solvability. Moreover, more blocks in the board means less space to fit another block in the board, so players must figure out how to clear more space. Regarding to principle #6, since there is no timer in 1010!, players can pause the game whenever they want to and resume it after rest time.

In addition to the 6 principles of puzzle game design, 1010! relies on Zeigarnik effect to cause addiction towards the game. [8]. Usually after reaching a certain checkpoint, players will save their progress and exit the game. However in 1010!, inserting a block into the board not only can clear the board, but also create new unfinished rows and columns, thus players feel as if they can never "complete" the game and fueled to finish a round until reaching game over [9].

Although 1010! has implemented good design principles, having a block generator with possibilities of generating unsolvable block set violates principle #4. If players found unsolvable block sets too often, they will feel like wasting time to solve an unsolvable problem and finally decide to give up [7], (p. 248). Hopelessness after repeatedly facing unsolvable problems can worsen player's ability in decision making [10].

Lastly, regarding to principle #3, implementation of performance improver should not hinder the game to update instantly, since higher latency can affects performance [11]. At the rate of 13 ms, human brain can already perceive what happened to the screen [12]. To maintain the illusion of animation, a screen must be updated at least every 100 ms, while a person can unpreparedly respond to stimulus within a second [13]. If there is no feedback from the game after one second, players will perceive it as a delay. This means the improver should run under one second; even better if it runs under 100 ms to avoid lagging animation.

B. AND-OR Tree

To validate a block set, we must search for a sequence of actions that can make a block set fits into the board. Since the solution is a sequence, our searching algorithm must consider various possible action sequences. We follow up one option and putting others aside for later, in case the currently chosen option does not lead to the solution [14].

Usually the search for solution is mapped into a tree, where nodes represent a subproblem with root node as its initial state and leaf nodes as goal states. In 1010!, the environment is nondeterministic, as inserting certain block in certain position can fill the board or clearing lines. Therefore, we will use AND-OR tree since it works on problems with nondeterministic environment.

An AND-OR tree is slightly different from normal tree or any directed graph used in problem solving. There are two types of nodes: OR nodes and AND nodes. For an OR node, their child nodes would not be influenced by other child nodes, thus if one child node has been marked as resolved, the parent node will be marked as resolved, too. Meanwhile in an AND node, all child nodes indicates choices determined by the environment, which means we must have a plan to solve each child nodes and therefore all child nodes in an AND node must be marked as solved before the parent node is marked as resolved, too. [15].

III. System Design

A. Mathematical Modelling of 1010!

A block is consisted of connected cells and labelled with a row vector that is equals to a 2D matrix of cells $b(t,r) = \{cell_{xy}^{tr}\}$, and fulfills (1), with $t = \{0,1,\dots,8\}$ indicates a block type out of available nine types, and $r = \{0,1,2,3\}$ indicates block rotation ($0^o, 90^o, 180^o$, and 270^o). Labelling for every block type can be seen in Table I.

$$\mathrm{cell}_{\mathrm{xy}}^{\mathrm{tr}} = \begin{cases} \text{-1} & \text{empty cell} \\ \mathrm{t} & \text{filled with block} \end{cases} \tag{1}$$

A board is represented in a 2D matrix $A = \{a_{xy}\}$ where all members must fulfill (2). Cells in a board follow the block type placed in said cell.

$$a(x,\mathrm{y}) = \begin{cases} -1 & empty\ cell \\ cell_{xy}^{tr} & filled\ with\ block \end{cases} \tag{2}$$

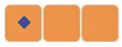
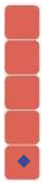
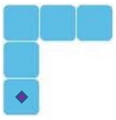
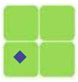
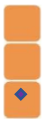
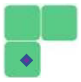
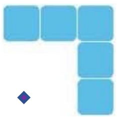
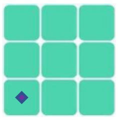
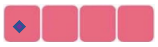
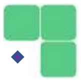
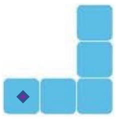
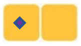
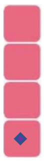
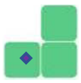
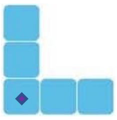
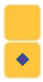
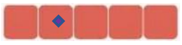
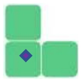
$$x, y = \{0,1,\dots,9\}$$

When inserting a block $b(t,r)$ into the board, the block's position will be referenced using a reference point $cell_{00}^{tr}$, which is the bottom leftmost cell. If $b(t,r)$ is placed on $a(x,y)$, that means in $a(x,y)$ there is a reference point $cell_{00}^{tr}$. All cells in board must fulfill (3).

$$\forall cell_{pq}^{tr} \neq -1 \land a(x+p, y+q) = -1 \tag{3}$$

Block holder is labelled as $holder(b_1, b_2, b_3)$ with b_i indicates the i-th block.

TABLE I: TYPES OF BLOCKS IN 1010!

Block	Label	Block	Label	Block	Label	Block	Label
	$b(0,0)$		$b(4,0)$		$b(6,1)$		$b(8,0)$
	$b(1,0)$		$b(4,1)$		$b(7,0)$		$b(8,1)$
	$b(2,0)$		$b(5,0)$		$b(7,1)$		$b(8,2)$
	$b(3,0)$		$b(5,1)$		$b(7,2)$		$b(8,3)$
	$b(3,1)$		$b(6,0)$		$b(7,3)$		

◆= position of referenced point ($cell_{00}^{tr}$) for each block

B. Algorithm for Validation

1010!'s board state is determined by two factors:

1) Permutation of blocks. There are three blocks in a block holder and each permutation uses three blocks, thus the amount of available permutations is $P(3,3) = 6$. These are labelled as $Permute = \{p_{ij}\}$ with $i \in \{0,1,\ldots,5\}$ and $j \in \{1,2,3\}$.

$$Permute = \begin{bmatrix} b_1 & b_2 & b_3 \\ b_1 & b_3 & b_2 \\ b_2 & b_1 & b_3 \\ b_2 & b_3 & b_1 \\ b_3 & b_1 & b_2 \\ b_3 & b_2 & b_1 \end{bmatrix}$$

2) Block's position. The same block in different location will cause different state. For example, if there are values of x_1, x_2, y_1, and y_2 that fulfills $x_1 \neq x_2, y_1 \neq y_2$, the state of board A after putting $b(t,r)$ in $a(x_1,y_1)$ will be different than in $a(x_2,y_2)$.

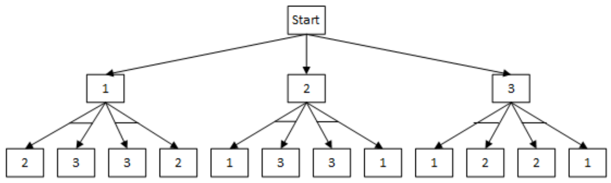


Fig. 3. The AND-OR tree used in validation stage.

The validation begins as illustrated in Fig. 3. There are three OR nodes that can be chosen. If one of these nodes are solved, the whole tree is solved. Assumed we are going to test b_1. If it fits into the board, the tree will traverse the vertexes connected to b_1, which has two AND nodes to solve. Assume we are going to b_2. If b_2 fits in the board, b_3 must fits in the board too, else both nodes will be marked unsolved.

The validation algorithm can detect which block causes failure and where it happens. It can be used to detect which block must be changed if the validation fails. The failure is noted in a row vector $unfitBlock = \{ub_i\}$ with $i = \{0,1,\ldots,5\}$ and filled with the failure's position based on current permutation used in validation. Fig. 4 describes the validation algorithm.

```
1.  While the next permutation is still available:
    (a) Adjust $holder$ with current permutation
    (b) While there is still a location to insert the first block ($p_{i1}$),
        do:
        i.   Create new state after putting $p_{i1}$.
        ii.  While there is still a location to insert the second
             block ($p_{i2}$), do:
             A. Create new state after putting $p_{i2}$.
             B. If there is a location to put the third block
                ($p_{i3}$), return SUCCESS code and stop the
                improver.
             C. If there is none, note $p_{i3}$ into $unfitBlock$ and
                move $p_{i2}$'s location.
        iii. If there is no location for $p_{i2}$, note $p_{i2}$ into
             $unfitBlock$.
        iv.  Move the location of $p_{i1}$.
    (c) If there is no location for $p_{i1}$, note $p_{i1}$ into $unfitBlock$.
    (d) Use the next permutation.
2.  If there is no permutation left, return FAILURE.
```

Fig. 4. Validation algorithm.

C. Algorithm for Changing Block

Block changing is triggered when the validator returns FAILURE. To decide which block is going to be changed, the algorithm will calculate heuristic points from data collected at validation stage, which are determined by two factors:

1) How often the block fails the test. The more often it breaks the test, the bigger the value will be.
2) Position of failure. If it happens in the tree's leaf node, the heuristic points will be bigger than in the parent node.

The calculation from $unfitBlock$ is stored first into a 2D matrix $F = \{fail_{ij}\}$ with $i = \{1,2,3\}, j = \{0,1,\dots,5\}$ that follows (4).

$$fail(i,j) = \begin{cases} 0 & p(j, ub(j)) \neq b_i \\ 5 * ub(j) & p(j, ub(j)) = b_i \end{cases} \quad (4)$$

After filling matrix F, heuristic point is calculated based on (5).

$$b_i = \mathrm{argmax}_i \left(\sum_{j=0}^{5} Fail(i,j)\right) \quad (5)$$

IV. Implementation and Testing

A. Implementation

In LonamiWebs' Klooni 1010!, block generator was located in `takeMore()` function from `BlockHolder` class. All parts of the improver were collected in a helper class called `State` which used to convert the 2D matrix A into 2D matrix $State = \{s_{xy}\}$ where all integers representing cell colors were replaced with Boolean system to ease checking. In `takeMore()` after blocks generation, the block holder will be validated using `validateBlock()` from class `State`, which will call two function: `checkPermute()` and `changeBlock()`.

B. Performance Measurement

TABLE II: Results from Performance Measurement

k = *empty cells*	Only validate (ms)	Validate+change (ms)
$k < 21$	5.736	7.001
$21 \leq k \leq 30$	4.011	15.147
$31 \leq k \leq 40$	4.170	14.919
$41 \leq k \leq 50$	4.675	10.717
$51 \leq k \leq 60$	3.685	22.250
$61 \leq k \leq 70$	3.858	-
$71 \leq k \leq 80$	3.610	-
$k > 81$	3.648	-
Average	**3.948**	**14.479**

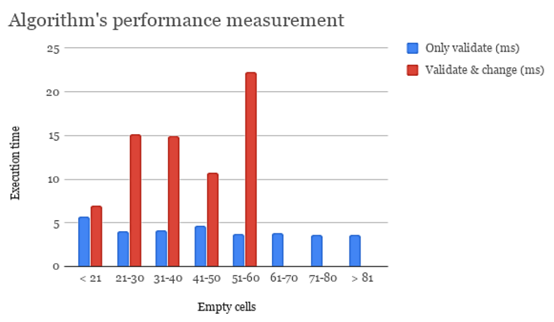


Fig. 5. Performance comparison.

Algorithm's performance was measured using `System.nanoTime()` from Java API with Nox emulator as the device (1 GB RAM and 1 CPU core allocated). In five gameplays, the testing will measure `validateBlock()`'s execution time in nanoseconds and classified them based on amount of empty cells in the board and whether `changeBlock()` was called. Table II and Fig. 5 showed that the more "crowded" the board were, the longer time needed for the algorithm to work. The average time needed for validation stage only was 4 ms, and average time for validation and changing stage was 14.5 ms.

V. Conclusion

In this paper, AND-OR tree had been successfully implemented during validation stage to ensure solution for any generated block sets. In case the validator found no possible solution, the generator will replace said unfit block according to heuristic calculation. Also, performance improver for 1010!'s block generator had been implemented and managed to run under 100 milliseconds. Typically, validation stage took 4 ms to run, and changing stage took 14.5 ms in average, since it also ran validation more than one time.

Future improvement can be focused on better method to determine which block will be changed in case of any unfit block set, while retain challenging for players and unburdening for devices.

References

[1] Gram Games, "1010! Press release," *Gram Games*, Istanbul, 2014.

[2] M. Kosoff. (2015). Here's how to play the game that's so addictive it's destroying people's sleep cycles. *Business Insider*. [Online]. Available: http://www.businessinsider.sg/how-to-play-1010-mobile-game-2015-11/?r=US&IR=T

[3] Oktomus. (2017). Feature request: Add an undo button. Issue #19. LonamiWebs/Klooni1010. *Github*. [Online]. Available: https://github.com/LonamiWebs/Klooni1010/issues/19

[4] Hitechcomputergeek. (2017). None of the set of three pieces matches. Issue #11. LonamiWebs/Klooni1010. *Github*. [Online]. Available: https://github.com/LonamiWebs/Klooni1010/issues/11

[5] Lonami, *1010! Klooni*, 2017.

[6] J. Bose, *LibGDX Game Development Essentials*, Birmingham: Packt, 2014.

[7] J. Schell, *The Art of Game Design: A Book of Lenses*, 2nd ed. Florida: Taylor & Francis Group, 2015.

[8] B. Zeigarnik, *On Finished and Unfinished Tasks*.

[9] M. Wu, "Game sophistication analysis: Case study using e-Sports Games and TETRIS," Japan Advanced Institute of Science and Technology, 2018.

[10] K. Starcke, J. D. Agorku, and M. Brand, "Exposure to unsolvable anagrams impairs performance on the IOWA gambling task," *Front. Behav. Neurosci*., vol. 11, p. 114, 2017.

[11] M. Claypool and K. Claypool, "Latency and player actions in online games," *Commun. ACM*, vol. 49, no. 11, p. 40, Nov. 2006.

[12] M. C. Potter, B. Wyble, C. E. Hagmann, and E. S. McCourt, "Detecting meaning in RSVP at 13 ms per picture," *Attention, Perception, Psychophys*., vol. 76, no. 2, pp. 270–279, Feb. 2014.

[13] S. K. Card, G. G. Robertson, and J. D. Mackinlay, "The information visualizer, an information workspace," in *Proc. the SIGCHI Conference on Human Factors in Computing Systems Reaching through Technology - CHI '91*, 1991, pp. 181–186.

[14] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. New Jersey: Prentice Hall, 2010.

[15] G. Levi and F. Sirovich, "Generalized and/or graphs," *Artif. Intell*., vol. 7, no. 3, pp. 243–259, 1976.

Livia A. Lohanda was born in 1998 in Tangerang, Indonesia. She took her bachelor of computer science at Universitas Pelita Harapan (UPH) from 2015. Since she was an undergraduate student, she has joined the School of Information Science and Technology UPH as a laboratorium assistant. Her current research interest is artificial intelligence.