The Depth-First Search Column by Column Approach on the Game of Babylon Tower

Romi Fadillah Rahmat , Harry, Mohammad Fadly Syahputra, Opim Salim Sitompul, Erna Budhiarti Nababan
Department of Information Technology, Faculty of Computer Science and Information Technology,
Universitas Sumatera Utara,
Medan, Indonesia
romi.fadillah@usu.ac.id, harry_masterquiz@students.usu.ac.id, nca.fadly@usu.ac.id, opim@usu.ac.id

***Abstract*—Babylon Tower is a three-dimensional puzzle which consists of six discs that are arranged piled up. Each disc consists of six columns of small balls along the side in six kinds of different colors. Babylon Tower can be played by rotating and sliding it. The goal of the game of Babylon Tower is to sort each ball of the same color on the same column and sort the balls in each column based on the brightness of the color of the ball. There are two methods that can be used to find a solution of the game of Babylon Tower, namely disc by disc and column by column. The method proposed in this research is depth-first search column by column. In this research, it is shown that the proposed method is capable of finding step by step to reach a solution of the game of Babylon Tower.**

Keywords—Babylon Tower; three-dimensional puzzle; column by column; depth-first search

I. INTRODUCTION

Babylon Tower is one of the three-dimensional puzzle that was invented by Endre Pap and patented in 1982. Babylon Tower can be played by rotating and sliding it, so that Babylon Tower has a very large number of possible ball positions and also a very large number of possible ways of solving it [1].

There are many studies that have been done to solve various kinds of puzzle game using various kinds of algorithm. Several searching algorithm such as frontier search, disk-based search, parallel processing, pattern database heuristic and breadth-first heuristic search are used to solve the problems in the game of four-peg Towers of Hanoi [2]. Another research was done on a method that can be implemented in the game of Japanese puzzle, also known as nonogram [3]. The method consists of two phases. The first phase is done by determining the cells that would be colored by using some logical rules. In the second phase, the depth-first search algorithm will be applied to solve those remaining cells. The research on Japanese puzzle has also been done by using rule-based method and best-first search algorithm [4]. A heuristic approach called Self-Adapting Harmony Search (SAHS) was implemented in the game of Kakuro [5]. SAHS algorithm can be improved if there is an initial calculation step to determine the possible combinations of sums to obtain the desired number. Chaotic harmony search algorithm has been used to develop flower pollination algorithm that was implemented in Sudoku [6]. In that research, the proposed method was capable of finding the better way of solving Sudoku puzzle.

Game is a real-time, dynamic and interactive computer simulation [7] that consists of a virtual world simulator that processes real-time data, shows the outputs visually, and controls the mechanism [8]. Data structures in a game development are used to represent objects in a game, and when they are combined with algorithms, the data can be processed efficiently and effectively [9] which may affect the running time [10]. Tree is a type of data structure that can be used to maintain the data hierarchy [9] in a path finding method [11]. In order to search solutions from a tree where there is no additional information about the state that will occur after a step is taken, we can use blind search [12] which is mainly consists of breadth-first search and depth-first search algorithms [13].

One of engaging idea is to solve Babylon Tower by combining column by column algorithm with another searching algorithm. Depth-first search has advantages in terms of memory usage that breadth-first search [13]. In order to give the first look result, we try to combine depth first search with column by column algorithm in order to solve this game. The other contribution is, we need to develop and implement Babylon tower in Android Platform, which will need a several steps in our general architecture.

II. METHODOLOGY

A. General Architecture

Fig. 1 shows the general architecture of the proposed method that consists of a series of steps that have been done to design and implement depth-first search column by column algorithm in the game of Babylon Tower.

B. Modelling

1) Disc Model

A Babylon Tower consists of six discs, in which four of them that reside in between another two discs, have the exactly same shape. Hence, there will be three types of disc model that are needed to be built, they are disc model that resides at the top position, disc model that resides at the bottom position, and disc model that resides in between another two discs.

Disc model that resides at the top position is built from a cylinder model with a diameter of 1.9 units and a height of 0.8 units. The upper part of the cylinder model is formed slightly

curved. Next, there are six gaps added around that cylinder model. Those gaps are formed apened at the lower part and closed at the upper part with the shape of half circle. The disc model for the top position is shown in the fig. 2.

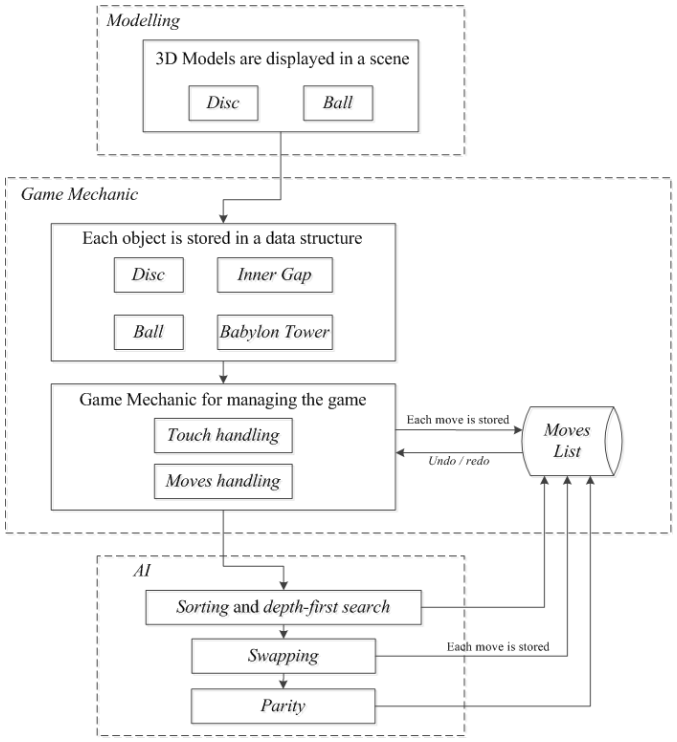

Fig. 1. General Architecture

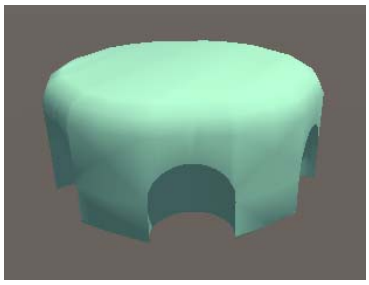

Fig. 2. Disc model at the top position

Disc model that resides at the bottom position is initially built in the same way as building disc model for the top position, then the model is rotated 180° around the Z-axis so that the opening of the gaps are facing upwards. Next, there is a hole that being made to connect two gaps in the opposite positions. The disc model for the bottom position is shown in the fig. 3.

Fig. 3. Disc model at the bottom position

Disc model that resides in between another two discs is built from a cylinder model with a diameter of 1.9 units and a height of 0.5 units. Next, there are six gaps added around that cylinder model. Those gaps are formed opened at bith lower part and upper part of that cylinder model. The disc model that resides in between another two discs is shown in fig. 4.

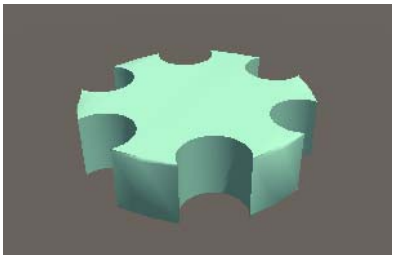

Fig. 4. Disc model that resides in between another two disc

2) Ball Model

In a Babylon Tower, there are 36 balls in six different types of color (black, yellow, red, green, brown, blue). Each type of color consists of six balls with different brightness level, starting from the brightest to the palest. Ball models are built from a sphere model with a diameter of 0.5 units. Then the color of each ball model is set according to the needs. Those 36 ball models are shown in fig. 5.

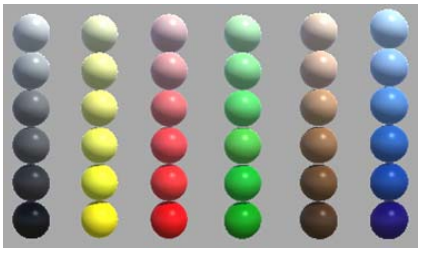

Fig. 5. Thirty six ball models that are used in a Babylon Tower.

3) Babylon Tower Model

The models that have already been built are arranged in such a way so that each model is placed in its position. Fig. 6 shows the arranged models of Babylon Tower. Disc model at the bottom position is placed at the center coordinates. A disc model that is located directly above another disc model is placed at the coordinates in which the value of Y-axis is 0.5 greater than the value of Y-axis of the disc below it.

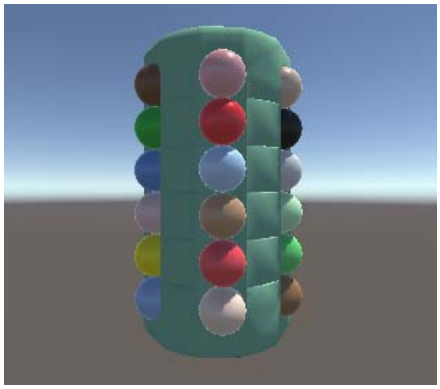

Fig. 6. Babylon Tower in 3D Model

Each ball model that is located in a disc is placed at the coordinates in which the Y-axis has the same values as the Y-axis of the disc. The ball that is located at the first column is placed at the X-axis coordinate of 0 and Z-axis coordinate of 0.86. The ball that is located at the next column is placed at the same coordinates, but is rotated around the Y-axis with the multiples value of 60°.

C. Data Structure

In order to represent each object that exists in the game of Babylon Tower, it is necessary to provide a class for each type of object.

1) Ball

There are several variables in the Ball class, including gameObject variable with the data type of Game Object that is used to store 3D model and its position, color variable with the data type of integer that is used to store a value that defines the ball's color, and brightness variable with the data type of integer that is used to store a value that defines the brightness level of the ball. There is also a static variable in the Ball class with the data type of array of string that stores the color's name. The methods in the Ball class are used for accessing the data that are stored in each variable in the Ball class from another class.

2) Disc

There are several variables in the Disc class, including gameObject variable with the data type of Game Object that is used to store 3D model and its position, balls variable with the data type of array of Ball that is used to store the ball objects in a disc, and row variable with the data type of integer that is used to store a value that defines the row position of a disc. The methods in the Disc class are used to move the position of each ball in a disc while rotated, and used for accessing the data that are stored in each variable in the Disc class from another class.

3) Inner Gap

Inner Gap is used to store the information about the ball that is residing inside the Babylon Tower. Inner Gap class contains a variable that is used to store the Ball object that is residing inside the Babylon Tower, and a variable that is used to store a value that defines the column position of that ball. The methods in the Inner Gap class are used to move the position of the ball inside Babylon Tower while the disc at the bottom position is rotated, and used for accessing and replacing the data that are stored in each variable in the Inner Gap class from another class.

4) Babylon Tower

Babylon Tower is used to store and manage each element in the Babylon Tower. Hence, there are several variables that are required in the Babylon Tower class, including discs variable with the data type of array of Disc that is used to store discs objects in the Babylon Tower, and innerGaps variable with the data type of array of InnerGap that is used to store a ball object that is inside the Babylon Tower. The methods in the Babylon Tower class are used to rotate a disc, used to push a ball into the disc at the bottom position, used to bring back the ball from inside the Babylon Tower, used to slide the balls in the same column, used to randomize the position of the balls, and used for accessinf and replacing the data that are stored in each variable in the Babylon Tower class from another class.

D. Game Mechanic

Each game application has its own rules and methods that are designed to interact with the states or conditions that happened in that application. Hereinafter those rules and methods are referred to as "game mechanic". The game of Babylon Tower is designed and built to be able to detect touch input by the user. Therefore, it is necessary to design a game mechanic that is capable of detecting various kinds of touch at the screen of the device and capable of detecting which object is touched. In addition, the game mechanic is also capable of deciding which kind of move that is done to the Babylon Tower according to the touch input by the user.

1) Touch Handling

Touch handling is a part of the game mechanic that is used to manage each touch input from the user. The screen position that is touched, the move direction of the finger, and the displacement made by the finger that is touching the screen are the key factors that decide what kind of move done to the Babylon Tower.

The following are the steps taken in managing the touch input:

- Initialize is_rotating to false, is_camera_rotating to false, selected_disc to zero, selected_ball to null, touch_reference to center coordinates, and total_rotation to zero.
- Check whether the user starts touching down the screen. If yes check whether the user touches any object. If yes, set is_rotating to true, set selected_disc to the disc touched by the user and set selected_ball to the ball touched by the user (if any). If no, set is_camera_rotating to true. Finally, set touch_reference to the coordinates of position touched by the user.
- Check whether the user is still touching the screen. If yes, initialize touch_offset to the coordinates of position touched by the user substracted by the touch_reference. If the value in is_rotating is true, add the touch_offset to the total_rotation and rotate each ball in the selected_disc. If

the value in is_camera_rotating is true, rotate the point of view of the Babylon Tower based on the touch_offset. Finally, set touched_reference to the coordinates of position touched by the user.

- Check whether the user's finger starts leaving up the screen. If yes, check whether the value in is_rotating is true or not. If true, initialize rotation_value to the number of columns of the disc rotated by calculating the total_rotation. If the rotation_value equals zero, check whether the selected_ball is null or not. If the selected_ball is not null, check whether the selected_disc is at the bottom and there is no gap. If yes and the ball can be pushed, then push the ball into the disc. If no and the ball can be slided, then slide the ball towards the gap. Finally, set is_rotating to false, set is_camera_rotationg to false, set total_rotation to zero, and set selected_ball to null.

2) Moves Handling

Each move that is done by user will be checked if the Babylon Tower is already solved or not. The basic moves that can be done to the Babylon Tower are rotating a disc (rotate), pushing a ball into the disc (push), sliding balls at a column towards the existing gap (slide), and bringing back a ball from inside the disc (pop). Each time the basic move is done, the values that are stored in the affected objects and their visual display in the scene will be updated. Each move will also being stored in a list called moves list. There are three type of moves that can be stored in the moves list, including rotate, push and slide.

For the "rotate" move, a disc will be rotated clockwise as far as a certain value. When the undo process is done, the disc will be rotated counter-clockwise as far as that value.

For the "push" move, the ball will be pushed into the disc, then all balls above it will be slided down covering the gap produced by the pushed ball. When the undo process is done, the first move is to slide all balls at the column towards the gap at the top position, then a ball behind the gap at the bottom postion will be brought back.

For the "slide" move, the ball will be slided from its initial position towards the gap so that the gap position will move to the ball's initial position. After the ball is being slided, it is needed to be checked if there is a ball behind the recent gap position. If there is, then the ball will be brought up covering that gap. When the undo process is done, if the last move is bringing back the ball inside the disc, then the reverse of the move is pushing that ball back into the disc. Next the sliding move can be done to return the balls to their previous positions.

E. AI

Before the algorithm is executed, there will be specified the ball color for each column. Then will be checked if the ball that resides at the first column of the bottom position has the similar color as the color specified for the first column. If not, then the color for the first column will be exchanged with the color of another column that has the similar color as that ball. Next that ball will be pushed into the disc so that a gap will appear at that column. Then the column by column algorithm can be executed.

1) Sorting and depth-first search

Sorting phase will group the balls with the similar color to the same column regardless to their brightness level. In this phase, there will be used a recursive function to repeatingly search a ball from another column that has a similar color as the color of the column that contains a gap. After the desired ball is found, some procedural steps will be executed to move that ball position to the column that has the similar color, so that the gap position will be moved to another column. Those processes will be repeated until each column is placed by the balls with a similar color.

During the sorting phase, it is necessary to be checked if the step taken would end up in a dead end. To avoid the dead end, it is needed a depth-first search algorithm that is used to step back to the previous state while the game is reaching the dead end, then looking for another step available.

The following are the steps taken in this phase:

- Start a method which require gap_column (the column that contains a gap) as a parameter and return boolean type of value.
- Initialize flag to false. If the sorting phase is completed, return true. Else if there is no gap, return false. Else, for each column except gap_column and for each ball in a column, check whether the color of the ball is the same as the color of the gap_column. If yes, move the ball to the gap by running several procedural steps and set the flag to the value obtained by running the same method recursively.
- Finally, if the flag is false, undo the last move taken.

2) Swapping

Swapping phase aims to swap the position of the balls at each column until each ball at each column is sorted based on its brightness level. This phase is considered complete if the brightness level of each ball at the third row from top to the bottom row has already been sorted based on its brightness level.

The following are the steps taken in this phase:

- For each column except gap_column, check whether each ball has already been placed to the nearest position to its actual position, which means the difference between the ball position to its actual position is at most one. If not, for each row, run several procedural steps to swap each ball until it is placed to the nearest actual position.
- For each column except gap_column, check whether each ball in the third row until the bottom row has already been sorted or not. If not, for each row from bottom row to the third row, run several procedural steps swap each ball that is not placed on its actual position.

3) Parity

Parity phase will be executed if the balls at the top two rows are not entirely sorted. The condition that is required to execute the parity phase is there must be pairs of columns that have not been sorted, because each time the parity phase is executed, it will produce a pair of column that is sorted. So, before parity phase is executed, it is needed to be counted the number of columns that have not been sorted. If the number is

odd, then will be executed another steps before the parity phase can be executed. The steps meant are rotating the disc at the top position at 60°, pushing in the ball at the first column, followed by repeating the sorting phase. After the sorting phase has been done, the number of unsorted columns must be even, so that the parity phase can be executed.

The following are the steps taken in this phase:

- For each column, initialize column_1 to -1 and column_2 to -1. Search two columns that have not been sorted and set the values to the column_1 and column_2 respectively.
- Initialize bottom_disc_rotated to false. This variable is used to determine whether the bottom disc need to be rotated in order to run parity steps.
- If the first inner gap column does not equal to column_1 and column_2, initialize column_3 to the first inner gap column. Else if the second inner gap column does not equal to column1 and column_2, initialize column_3 to the second inner gap column. Else, set bottom_disc_rotated to true, rotate the bottom disc around 60°, and initialize column_3 to the first inner gap column.
- Run parity steps and check whether the bottom_disc_rotated is true or not. If true, rotate the bottom disc around 300°.
- If there are other unsorted columns, run this method again recursively.

III. Result and Discussion

A. Implementation

Game of Babylon Tower was built using Unity software version 5.0.0f4 using C# programming language. The output targeted was an application for Android platform (.apk). The following will be described the results of application execution of the game of Babylon Tower. Fig. 7 shows the main menu of the application.

Fig. 7. Main Menu

The first option of the main menu is “play” that is used to start a new game. By selecting “play” option, user can select one of the three difficulty levels of the game, there are easy, medium, or hard. The difficulty level selected by user will decide how many columns of Babylon Tower will be randomized. After selecting the desired difficulty level, main menu view will be switched with game scene view as shown in the fig. 8.

Next, user can interact with the randomized Babylon Tower to play it. Every step taken by the user is stored in the list and the number of steps that have already been done by user will be shown below the Babylon Tower. On the upper part of the game scene, there is a timer that determines the time taken by user to solve the game. User can navigate against the steps that are stored in the list through the buttons that are located on the lower part af the game scene. User can run the animation of the steps that are stored in the list by touching the play button. While the animation is running, the play button is switched with the pause button that is used to pause the running animation. User can decide to solve the game with AI by touching the solve button. By touching the solve button, the animation will be played to show the step by step taken by AI to solve the game started from the beginning state. User can go back to the main menu by touching the home button.

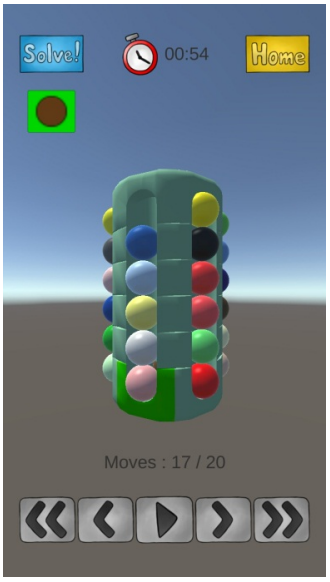

Fig. 8. Game Scene

B. Testing

The algorithm is considered successfully find a solution if the algorithm is capable of grouping each ball with the similar color in the same column and sorting each column according to the brightness level of each ball. Testing had been done as much as ten times for the randomized Babylon Tower for each different difficulty levels. The number of steps taken by AI and the time required to reach the solution for each test case are recorded during the testing. Table 1 shows the number of steps taken and table 2 shows the time required to reach the solution for each test case.

TABLE I. Testing Results Against The Number of Steps Taken to Reach The Solutionable Type Styles

Case	Difficulty level		
	Easy	***Medium***	***Hard***
1	165	322	362
2	111	344	403

3	191	255	391
4	160	309	429
5	95	304	406
6	192	274	474
7	237	265	336
8	105	272	489
9	116	278	502
10	141	325	515

TABLE II. TESTING RESULTS AGAINST THE TIME REQUIRED TO REACH THE SOLUTION

Case	Difficulty level		
	Easy (ms)	*Medium (ms)*	*Hard (ms)*
1	9.436607	13.94987	15.61999
2	8.593082	14.78124	15.67364
3	10.36525	13.05532	15.95926
4	9.3925	13.76057	16.25204
5	8.271217	13.64374	16.09182
6	10.04815	13.08036	20.22243
7	11.0476	12.01272	14.73069
8	7.363796	14.24599	17.25554
9	7.842064	13.42607	18.00632
10	9.407759	14.63747	18.58711

IV. CONCLUSION

Implementation of the depth-first search column by column algorithm on the game of Babylon Tower results in some conclusions which are: the proposed method is always capable of finding solution for the randomized Babylon Tower; the average number of steps taken for the proposed method to find the solution is 151 steps for easy difficulty, 295 steps for medium difficulty, and 431 steps for hard difficulty; the average time required for the proposed method to find the solution is 9.1768025 ms for easy difficulty, 13.659335 ms for medium difficulty, and 16.839884 ms for hard difficulty.

ACKNOWLEDGMENT

The authors acknowledge the suggestions and comments from both reviewers and editors that have helped improve the presentation of this paper.

REFERENCES

[1] J. Scerphuis : Babylon Tower. http://www.jaapsch.net/puzzles/ivory.htm

[2] R. E. Korf and A. Felner, "Recent progress in heuristic search: A case study of the four-peg towers of hanoi problem," Proceeding of International Joint Conference on Artificial Intelligence, pp. 2324-2329, 2007.

[3] M. Q. Jing, C. H. Yu, H. L. Lee and L. H. Chen, "Solving japanese puzzles with logical rules and depth first search algorithm," International Conference on Machine Learning and Cybernetics, vol. 5, pp. 2962-2967, 2009.

[4] D. Stefani, A. Aribowo, K. V. I. Saputra and S. Lukas, "Solving pixel puzzle using rule-based techniques and best first search," International Conference on Engineering and Technology Development, pp. 118-124. 2012.

[5] S. Panov and S. Koceski, "Solving kakuro puzzle using self adapting harmony search metaheuristic algorithm," International Journal of Engineering Practical Research, vol. 3, pp. 34-39, 2014.

[6] O. Abdel-Raouf, M. Abdel-Baset and I El-henawy, "A novel hybrid flower pollination algorithm with chaotic harmony search for solving sudoku puzzles," International Journal of Engineering Trends and Technology, vol. 7, 126-132, 2014.

[7] J. Gregory, Game Engine Architecture. A K Peters, Ltd: Massachusetts, 2009.

[8] D. Sanchez and C. Dalmau, Core Techniques and Algorithms in Game Programming. New Riders: Indianapolis, 2004.

[9] A. Sherrod, Data Structures and Algorithms for Game Developers, Charles River Media: Boston, 2007.

[10] M. T. Goodrich and R. Tamassia, Algorithm Design and Applications. Wiley: New Jersey, 2015.

[11] G. T. Heineman, G. Pollice and S. Selkow, Algorithms in a Nutshell. O'Reilly: Sebastopol, 2009.

[12] G. F. Luger and W. A. Stubblefield AI Algorithms, Data Structures, and Idioms in Prolog, Lisp, and Java for Artificial Intelligence. Pearson Education, Inc,: Boston, 2009.

[13] S. Russell and P. Norvig, Artificial Intelligence A Modern Approach. 3rd Edition. Pearson Education, Inc,: New Jersey, 2010.