IJCCS (Indonesian Journal of Computing and Cybernetics Systems)
Vol.17, No.1, January 2023, pp. 91~102
ISSN (print): 1978-1520, ISSN (online): 2460-7258
DOI: https://doi.org/10.22146/ijccs.81178

Progressive Content Generation Based on Cyclic Graph for Generate Dungeon

Muhammad Anshar[*1], Raden Sumiharto[2], Moh. Edi Wibowo[3]
[1]Magister Ilmu Komputer Departemen Ilmu Komputer Dan Elektronika, (FMIPA) Universitas Gadjah Mada Sekip Utara, Bulaksumur, Yogyakarta 55281 Indonesia
[2]Departemen Ilmu Komputer dan Elektronika (FMIPA) Universitas Gadjah Mada Sekip Utara, Bulaksumur, Yogyakarta 55281 Indonesia
e-mail: ***[1]Muhammad.anshar@mail.ugm.ac.id**, [2]r_sumiharto@ugm.ac.id, [3]mediw@ugm.ac.id

Abstrak

Dungeon merupakan level dalam game yang terdiri dari kumpulan ruang dan pintu, yang terdapat rintangan di dalamnya. Untuk membuat level yang bagus, memerlukan waktu yang banyak. Dengan Procedural Content Generation (PCG), dungeon bisa dibuat secara otomatis. Salah satu pendekatan dalam PCG untuk membuat level adalah progressive. pendekatan progressive menghasilkan timeline sebagai representasi interaksi di dalam game. Representasi graph timeline yang berbentuk satu garis lurus bagus untuk konten endless runner. Tapi untuk konten dungeon, level yang terbentuk sangat linear. Pada penelitian ini representasi timeline diubah menjadi cyclic graph. Cyclic graph dibentuk menggunakan algoritma graph grammar. Penelitian ini bertujuan untuk membangun dungeon yang tidak linear dan minim dari jalan buntu. Untuk menghilangkan linearitas pada dungeon, maka percabangan dalam dungeon perlu dibentuk. Tahapan yang dilakukan dalam penelitian ini adalah perancangan graph grammar rule, generate graph sebanyak populasi, evaluasi graph berdasarkan nilai fitness, dan pembentukan dungeon. Ada empat fungsi yang digunakan untuk menentukan nilai fitness, yaitu shortest vertices, average duration, replayability, dan variation. Dungeon yang dihasilkan dengan pendekatan progressive berhasil meminimalisir linearitas pada dungeon. Pembentukan dungeon sangat bergantung pada rule grammar yang membentuknya. Dengan adanya proses evaluasi, dungeon yang linear hasil dari grammar rule dapat diminimalkan.

Kata Kunci - *Dungeon, Generative Grammars, Graph, Procedural Content*

Abstract

Dungeon is a level in the game consisting of a collection of rooms and doors with obstacles inside. To make a good level, takes a lot of time. With Procedural Content Generation (PCG), dungeons can be created automatically. One of the approaches in PCG to create levels is progressive. the progressive approach produces a timeline as a representation of the interactions in the game. A graph timeline representation that is in the form of one straight line is good for endless runner content. But for dungeon content, the levels are very linear. In this research, the timeline representation is changed to a cyclic graph. The cyclic graph is formed using a graph grammar algorithm. This research aims to build a dungeon that has not linear and is minimal dead ends. To eliminate linearity in dungeons, branching in dungeons needs to be formed. The steps carried out in this research are designing graph grammar rules, generating graphs as many as the population, evaluating graphs based on fitness values, and building dungeons. Four functions are used to determine the fitness value: shortest vertices, average duration, replayability, and variation. Dungeons produced with a progressive approach manage to minimize linearity in dungeons. Dungeon formation is very dependent on the rule grammar that forms it. With the evaluation process, linear dungeons resulting from grammar rules can be minimized.

Received January 10th,2023; Revised January 31th, 2023; Accepted January 31th, 2023

Keywords - *Dungeon, Generative Grammars, Graph, Procedural Content*

1. INTRODUCTION

Dungeons in games are environments that consist of a collection of rooms and doors, which contain obstacles such as monsters and puzzles [1], [2]. To build a dungeon that meets the criteria of a game, takes a relatively long time depending on the size and criteria needed. Procedural Content Generation (PCG) can provide a solution to this problem.

PCG is programmatic content creation with a certain algorithm that uses a random process. Content created by PCG is built automatically with unpredictable results depending on the application of certain constraints on the algorithm. PCG is often used to create content in games, and dungeons are no exception. PCGs can create more content, faster, and cheaper than humans [3], [4].

In general, PCG is grouped into Constructive, Search-based, and Generate-and-test approaches. The approach commonly used to generate dungeons is the constructive approach because the constructive approach is considered fast and predictable [5]. A constructive approach is an approach that produces content once executed, then the process is completed without further content evaluation [6]. The constructive approach has the disadvantage of guaranteeing that the content produced will meet the constraints (if any) or not because there is no evaluation step. In this case, the search-based approach can handle it. The search-based approach is a PCG approach that adapts the evolutionary algorithm. The search-based approach will look for solutions (in this case content) that are in accordance with the constraints. The search-based approach has a weakness in its speed, the search-based approach is much slower than the constructive approach and is too slow to use in real-time [7]. Research tried to combine the diversity, flexibility, and ability to guarantee the playability of a search-based approach with the speed of a constructive approach called the progressive approach.

A progressive approach is an approach that makes the timeline the basis for the content produced. The timeline in the progressive approach is a series of graphs that represent the sequence of interactions in the game. The process of generating timelines is carried out using a search-based approach. Then the timeline was changed to a level with a constructive approach. But a timeline that is in the form of a straight line is not suitable to be a base for making dungeons.

Timeline representation doesn't have big problems in a puzzle or endless runner genre games. But in games with dungeon-shaped levels, the level that is formed is a level with a linear form and can cause deadlocks that force the player to frequently turn back if there is a mission that requires the player to return to the starting point.

Research [8] introduces a cyclic graph generator, which is a generator for producing circular graphs. Representation of a cyclic graph is a graph that has branches, and edges that are not always bidirectional, and rotates. To overcome the problems in the progressive approach. The timeline will be changed to a cyclic graph.

The research being conducted is to build a generator using a progressive approach. By changing the timeline to a cyclic graph at the timeline generation stage, you can create a dungeon that is controlled according to constraints, that is, without many dead ends and not linear.

2. METHODS

The research being conducted is to make a dungeon generator with a progressive approach based on a cyclic graph. The research was started by designing a dungeon generator to generate a dungeon model. dungeon generator will be used in games that can be played by users. Then, the generator is tested by changing parameter values to analyze the performance of the resulting dungeon model. In general, Figure 1 illustrates the flow of research to be carried out.

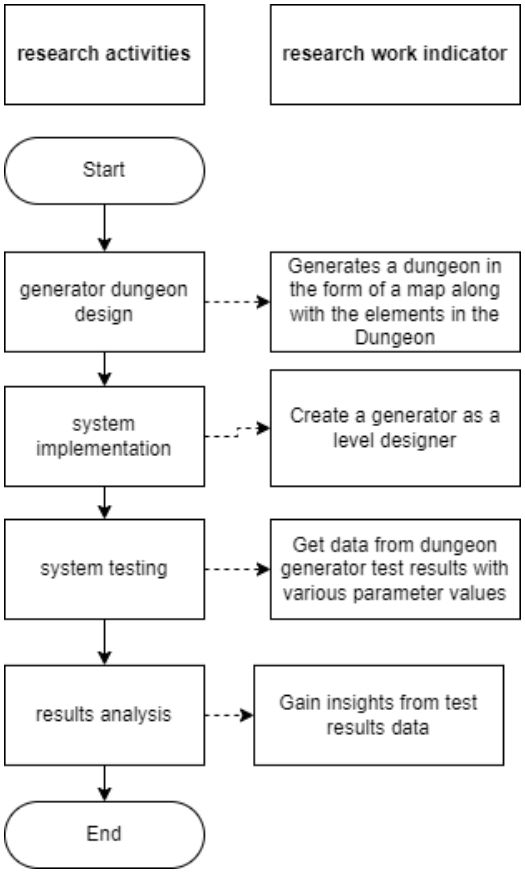

Image 1 Research plan

2.1 Dungeon Generator Design

The dungeon generator made in this research uses an approach inspired by the progressive approach, so the dungeon generator architecture is divided into two layers, namely the mission generator and the map generator. In the mission generator layer, several missions are generated and evaluated based on the fitness function to get the optimal mission. The approach applied to the mission generator layer is search-based. Meanwhile, in the map generator layer, missions that are only in the form of a series of interactions will be mapped into dungeons. The approach applied to the map generator layer is Constructive. In general, the dungeon generator design is depicted in Figure 2.

2.2 Mission Generator Design

The mission design in this research is a graph consisting of nodes and edges. The graph structure used in this research consists of nodes and edges. Each node can contain multiple edges, a unique ID, an alphabetic mission symbol, and a temporary marker number to change the shape of the graph. The alphabet used in this research can be seen in table 1. Meanwhile, each edge on the mission has a unique ID, start pointer, and end pointer, as well as edge type information.

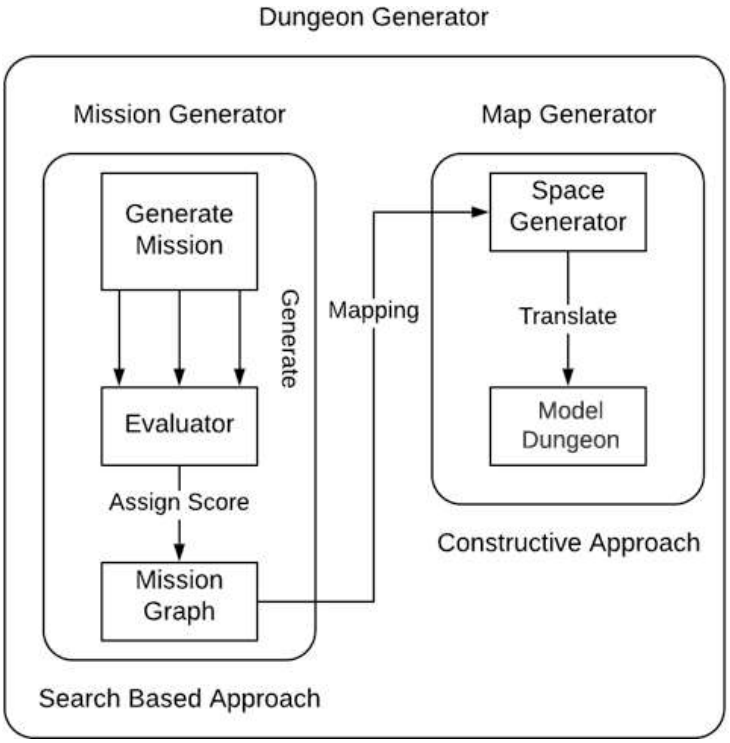

Figure 2 Generator Dungeon Flow

Table 1 Alphabet symbols and their meanings

No.	Alphabet	Meaning	No.	Alphabet	Meaning
1	S	Start	6	K	Key
2	G	Goal	7	R	Reward
3	T	Task	8	St	Secret
4	O	Obstacles	9	?	Any node
5	L	Lock	10	I	Init

The design of the mission generator in this research uses the generative grammar method. If generative grammars usually use rewriting rules for strings, in the mission generator the rewriting rules are done for graphs. In graphs, generative grammars will rewrite one or several nodes that are connected. The edges are the recipe replaced with different node and edge structures, as shown in Figure 3 (the numbers in the nodes are used as markers for which nodes will be changed according to the rules of left and right sides). The process of replacing the graph structure into a new graph is as follows:

1. Select the graph or node to be replaced randomly.
2. The selected nodes are numbered according to the rule on the left side.
3. All edges between nodes are removed.
4. The numbered node is then replaced by its equivalent on the right side of the rule.
5. Then the nodes on the right side of the rule that are not the same as the nodes on the left side of the rule are added to the graph.
6. Then add edges to the new nodes that are included in the graph, according to the rules on the right side.

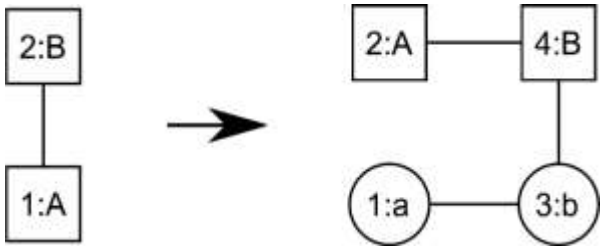

Figure 3 examples of graph grammars rules

The rewriting rules for the mission generator used in this research consist of five groups of rules, namely, init, extend, secret, obstacles and rewards, and keys and locks. The init rules are the writing rules for cyclic graph initiation. Graphs formed from init rules do not have nodes that only have one edge, so when other nodes are inserted, the graph remains cyclic. In the init rules, start and goal nodes are also formed which are the entry and destination points for players. then there are extend rules, namely rules for expanding tasks in various forms. The secret rules and the obstacles and rewards rules are the same rules that insert nodes, the difference is in the secret rules, the nodes that are inserted are secret nodes, while the obstacles and rewards rules that are inserted are obstacle nodes and reward nodes. For the keys and locks rule, the key and lock nodes will be inserted into the mission as mutually dependent nodes. Node lock can only be passed if the node key has been passed. The rules of keys and locks also have a variety of forms, namely branching, cyclical and linear. The mission rules are illustrated in Figure 4.

There are several types of edges in the mission in this research, the first is the requirement is a link between two nodes which is depicted by a line and an arrow at the end, the second is a key relation which is a marker of a lock that can be opened with a key that points to it and is depicted with a dotted arrow.

2.3 Recipe for Generating Content

Recipe (recipe) is used to control the graph to be generated. In generative grammar, the iteration stops when there are no more non-terminal alphabets. However, in this research there is no non-terminal alphabet, so recipes are needed to limit iterations. The recipe is a constraint on the number of a rule group to be executed. Here is an example of the recipe:
1 x init. 3 x extend. 1 x secret. 5 x obstacles and rewards. 3 x keys and locks.

2.4 Functions of Fitness Mission

The mission generator generates as many mission graphs as the population, but not all mission graphs proceed to the next stage. Each mission graph is evaluated first based on its fitness value. The fitness function is used to find which mission is good enough to be converted into a game level. Each generated mission is sorted based on the calculated value of the fitness function. The mission that has the highest score will be converted into a game level. Adapt from research [9] with several additional functions, there are four supporting functions used in this research to determine the fitness value, namely:

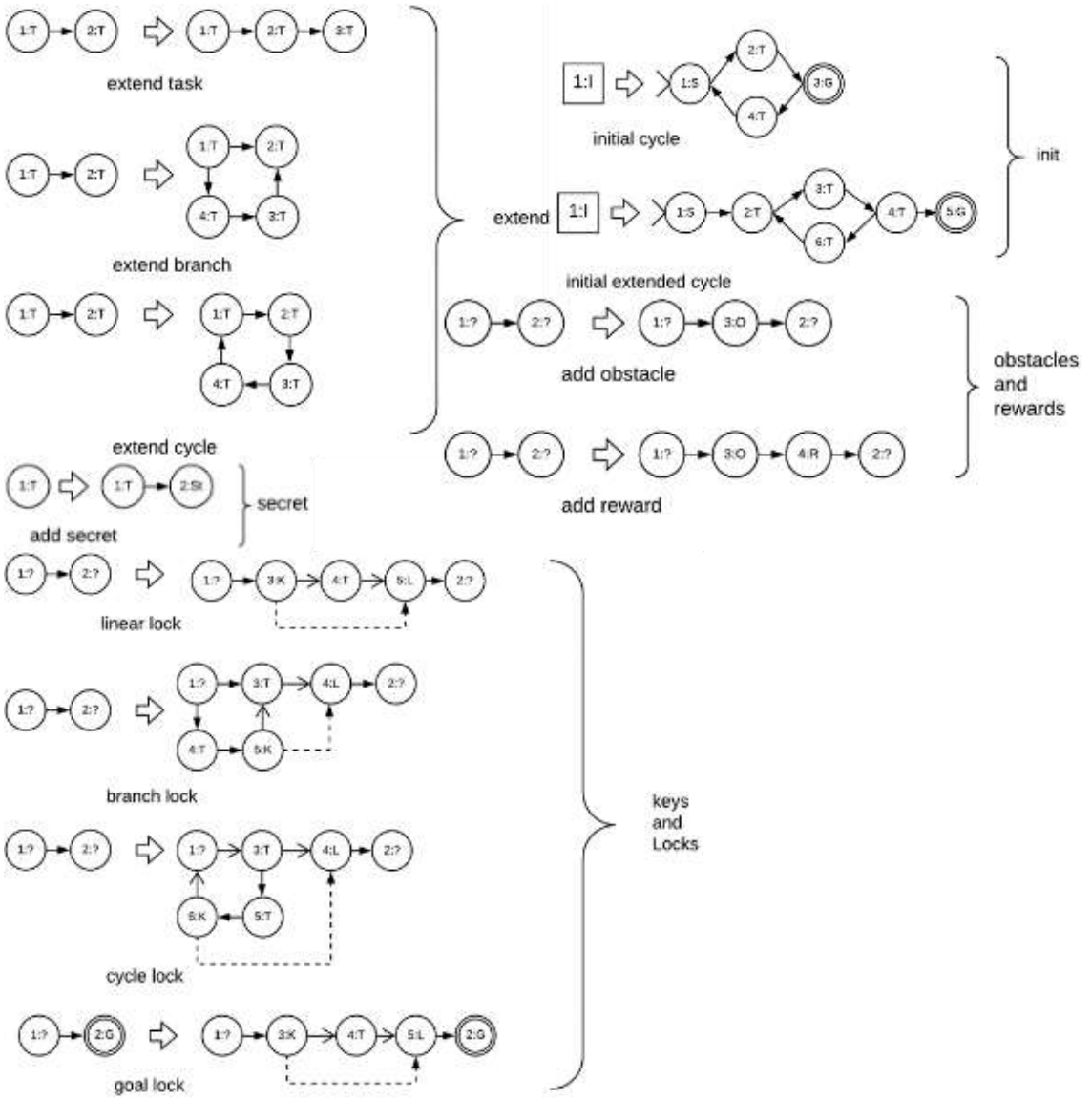

Figure 4 Rule missions

- Shortest vertices (f_{sv}). Shortest vertices is the value of the minimum number of vertices required from vertex start to vertex goal V_s, then normalized based on the preferred value for Shortest vertices p_{sv}. The preferred value is the percentage value of the minimum number of vertices to reach the vertex goal defined by the user. Where V is the number of vertices in the graph.

$$f_{sv} = 1 - \left| \frac{\frac{p_{sv}}{100} * V - V_s}{V} \right| \tag{4.1}$$

- Average Duration (f_{ad}). Average Duration is the average weight on each edge which is normalized based on the preferred value for the weight. In this research, there are three possibilities for the weight value at the edge, namely one as a short representation, two as an ideal representation, and three as a long representation. So, the preferred value used is a constant number two. Where the $\sum$w is the number of weights and E is the number of edges.

$$f_{ad} = 1 - \left| \frac{\sum w - (E * 2)}{E} \right| \quad (4.2)$$

- Replayability (f_r). Replayability is the number of branching vertices V_b divided by all vertices V.

$$f_r = \frac{V_b}{V} \quad (4.3)$$

- *Variation* (f_v). Variation is the number of edges that connect two vertices of different types, then divided by the total number of edges. Where E_d is the number of edges connecting different vertices and E is the number of edges in the graph.

$$f_v = \frac{E_d}{E} \quad (4.4)$$

After all the functions supporting the fitness value are obtained, then the fitness value can be calculated. The fitness value function is the sum of shortest vertices, Average Duration, replayability, and variation divided by the number of supporting functions.

$$f = \frac{f_{sv} + f_{ad} + f_r + f_v}{4} \quad (4.5)$$

2.5 Space Generator Design

If missions are the abstract structure of a series of player actions, then space is the abstract structure of dungeons. Space contains information such as position and what elements exist in a dungeon [10]. The space graph has three types of nodes. Each node in the space graph can be specified by using a letter indicating the type of the node. There are three types of space nodes (also see figure 9) :

- Place. Place is a field that has a real position both on the graph and level. Places can vary from one game to another. It can be a room, a platform, or a zone. Places are represented as open circles which can be sized to determine their dimensions within the level.
- Lock. Locks are a special type of place that cannot be accessed until the player has activated or obtained the appropriate lock. Lock has two states: locked and unlocked.
- Game elements. Game elements are game features that can be placed in a place such as items, characters, or creatures. Game elements are represented as small circles within the place circle. Element game also has an alphabet as a marker for a certain feature. The following game elements are often found in a game such as entrance, goal, key, each marked with the alphabet 'e', 'g', and 'k'. These elements represent game objects that implement mission logic.

The space graph can have many relations represented by different types of edges. Some of them depend on their implementation in the game. Common relations that are often encountered are as follows:

- Path. Path indicates that players can move freely between two places (places). A path can be traversed in both directions. Path is represented by an arrow pointing to a space.
- Gate. Gate is a relation that connects two places or game elements with a place. Unlike a path, it can only be traversed in one direction. A gate is represented as a solid arrow with the arrowhead blocked by a perpendicular line indicating the direction of travel.
- Window. A window is a relationship indicating that a player can see a place from another place, but cannot directly travel to the place seen from the window. The window is represented as a dotted arrow connecting the two places. Arrows point towards observable places.
- Lock. The lock relationship connects game elements with locks. When connected to a lock, the source game element serves as the key to unlock it. A lock can have multiple keys. The lock relation is represented as a dotted line that ends in an open circle next to the lock.

In this research, the space graph was initiated with a mission graph generated from a mission generator. In the space graph representation, all nodes in the mission graph are changed to game elements except for the task node which is changed to place.

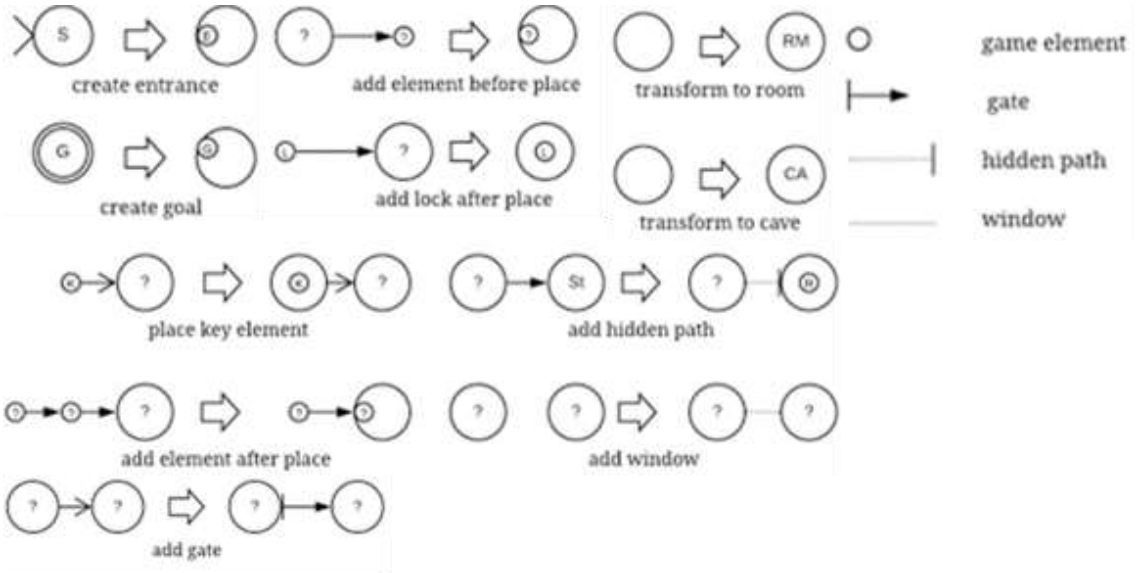

Figure 9 Space grammars rules for making maps

2.6 System Implementation

The resulting Dungeon is only a map model and cannot be played yet. In this research, the resulting dungeon model is a graph data structure in json format.

2.7 System Testing and Analysis

Dungeon generators that have been made will be tested with various recipe combinations. Dungeon results from testing were analyzed for their influence on deadlocks and branching in dungeons.

3. RESULT AND DISCUSSION

In this research, testing was carried out by executing various recipes on the generator. Each recipe is executed 100 (one hundred) times with a population of 5 (five) dungeons in each trial. Then the resulting dungeon from the recipe will be evaluated for dead ends and forks in it.

3.1 Preparation of the Recipe

There are three scenario recipes made to test the generator and see the results of the dungeon being built. Each recipe aims to test the generator's ability to generate dungeons. Each dungeon generated from the recipe is evaluated for branching and the presence of dead ends.

Tabel 2 *recipe* 1

Recipe 1		Recipe 2		Recipe 3	
No	Rule	No	Rule	No	Rule
1	init cycle	1	init extended cycle	1	Random init
2	Extend task	2	Extend task	2	Random extend
3	Extend task	3	Extend task	3	Random extend
4	Extend branch	4	Extend cycle	4	Random extend
5	Extend cycle	5	Secret	5	Secret
6	Obstacle	6	Obstacle	6	Obstacle

7	Reward	7	Reward	7	Reward
8	Linear lock	8	Linear lock	8	Random lock
9	Branch lock	9	Linear lock	9	Random lock
10	Cycle lock	10	Cycle lock	10	Random lock
11	Goal lock	11	Goal lock	11	Random lock
12	Space rule	12	Space rule	12	Space rule
13	Duplicate key	13	Duplicate key	13	Duplicate key
14	Duplicate lock	14	Duplicate lock	14	Duplicate lock
15	Move lock forward	15	Move lock forward	15	Move lock forward

In recipe 1, each rule is executed once except for the init extended cycle. In recipe 2 all branch rules are removed, this is to find out whether without branch rules branches in dungeons can still be formed. And recipe 3 has a new rule that is intended to produce dungeons from the same patterns, namely the random rule. Random init aims for the generator to execute a randomly selected init rule. Likewise for random extend and random lock.

3.2 Generator Testing

Tests carried out using recipe 1 result in the dungeon being made, it can be ascertained that it has branching and is unlikely to have a dead end. It can be seen in the graph in figure 10 that out of a hundred attempts, the number of deadends in the dungeon is zero. This proves the generator can make dungeons 100% deadlockless. As for branches, there are always branches in dungeons with a minimum of 3 branches and a maximum of 7 branches. This is because in the recipe there are executing the extend branch rule and branch lock rule. Without the secret rule and extended init rule, the resulting dungeon has no dead ends. Dungeon results can be seen in Figure 11.

Tests performed using recipe 2 always produce a deadlock. As can be seen in Figure 12, out of a hundred trials, there is always a deadend with a minimum of one and a maximum of three. Whereas branches are like the dungeon in recipe 1, the minimum number of branches created in dungeons is 3 and the maximum number is 7. Dead ends that are always formed are limited to the secret room, which is the room where the vertex secret is located. But it is also possible to form a deadlock in a room containing a start or entrance vertex and a room containing a goal vertex if neither of these spaces is selected by the extend rule. For branches in dungeons, even though there are no branch rules in the recipe, dungeons always have branches. Because the extend cycle rule and cycle lock indirectly make branches at the selected vertex. An example of a dungeon generated using recipe 2 is shown in Figure 13.

Tests carried out using recipe 3 produce dungeons that are difficult to predict, because there is no certainty about the formation of cycles and branches. All extend rules and keylock rules are randomly selected. Seen in figure 14 the range of the longest number of branches compared to the dungeons produced from recipe 1 and recipe 2, with a minimum number of branches of 1 and a maximum number of branches of 8. The graph also shows that the flat lines formed are not long, this shows the dungeons made by the generator is always different each time it is run. If all the selected extend rules and keylock rules are rules that do not form cycles and branches, then the resulting dungeon has no branches. But this can be minimized by evaluating the fitness of each graph. If of all populations there is a graph that contains branches, certainly, the graph that does not have branches will not continue to become a dungeon. It is not certain that the resulting dungeon will have branches because there is a possibility that none of the population graphs will have branches. An example of a dungeon generated using recipe 3 in figure 15.

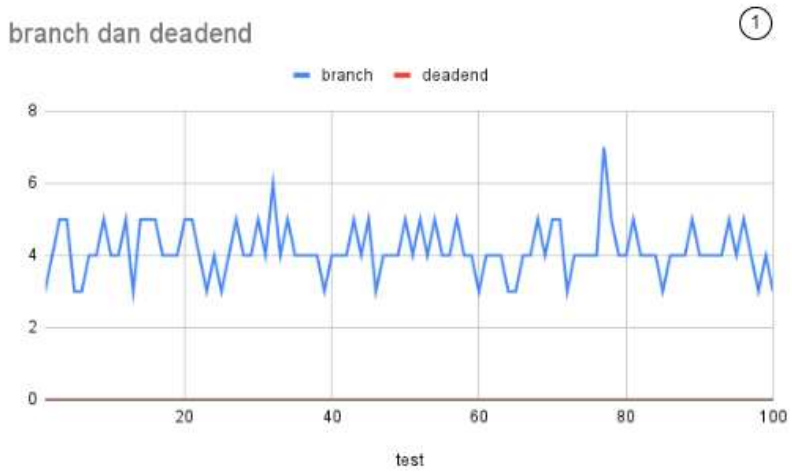

Figure 10 graphs of branches and deadends generated from recipe 1.

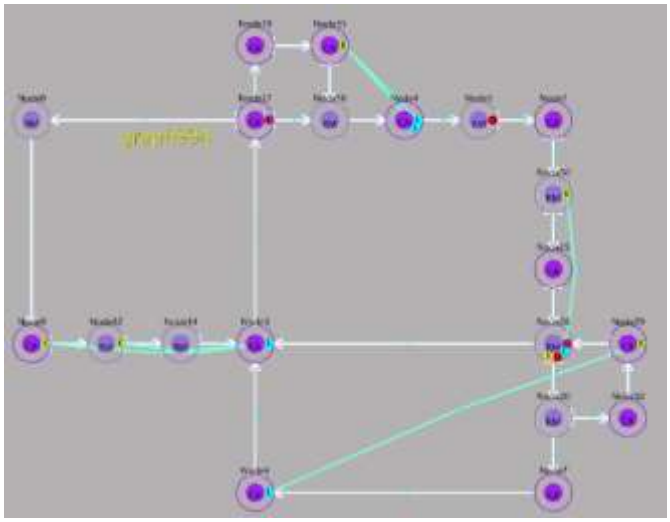

Figure 11 is an example of a dungeon generated from recipe 1.

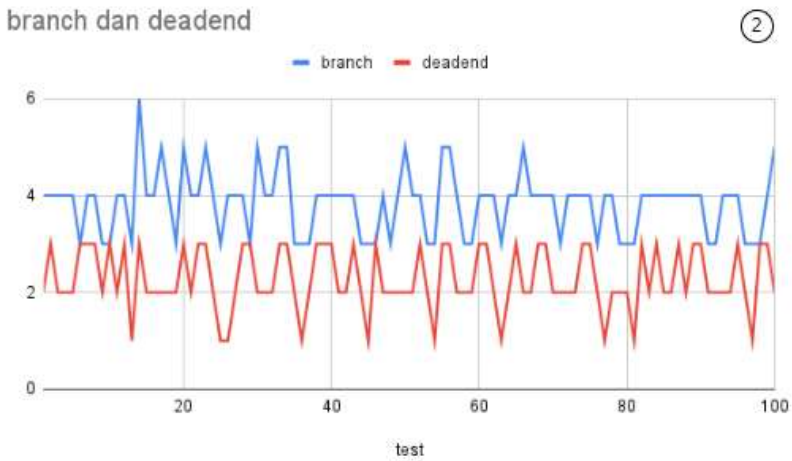

Figure 12 graphs of branches and deadends generated from recipe 2.

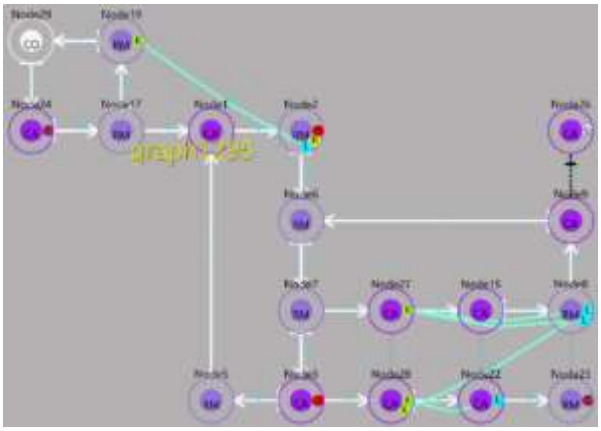

Figure 13 is an example of a dungeon generated from recipe 2.

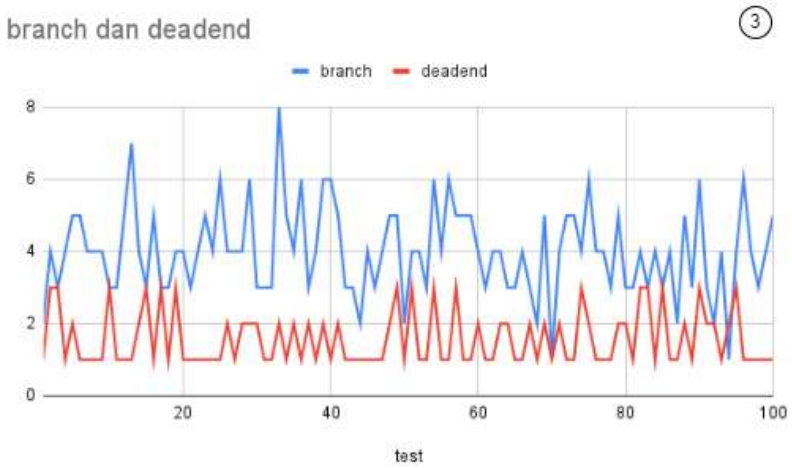

Figure 14 graph of branches and deadends generated from recipe 3.

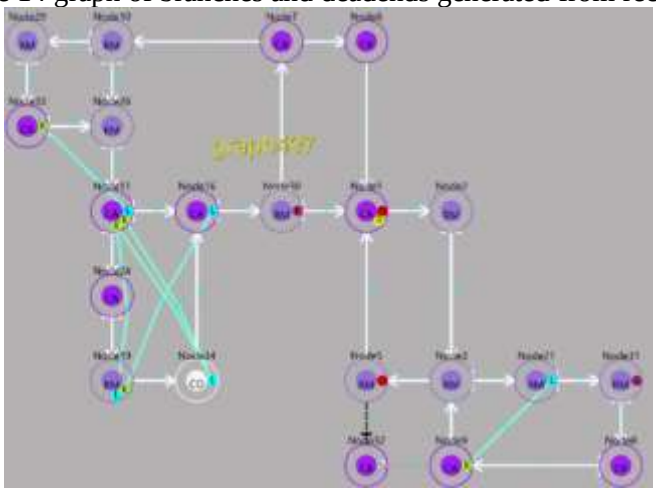

Figure 15 examples of dungeons generated from recipe 3.

The result of testing generators from various recipes is that the resulting dungeon specifications depend on the rules contained in the recipe. The dungeon generated by the generator is guaranteed to always form a cycle, because at the initiation stage a graph cycle has already been formed. Dungeons can still generate deadlocks, but the deadlocks formed are limited to starting rooms, ending rooms, and secret rooms. Dungeons always have branches if there are executing cycle rules or branch rules. To prevent dungeons from the same patterns from appearing frequently, you can include random rules in the recipe. With the random rule, each generator

creates a dungeon, reducing the possibility of a dungeon with the same number of branches being formed in a row. This can be seen from the graph in Figure 14 where the flat line is not long.

4. CONCLUSIONS

Based on the research and test results that have been carried out, it can be concluded that The dungeon generated by the generator is 100% deadlock free if the dungeon has no secret room. The dungeon that the generator produces 100% has branches with the number of branches depending on the rules used.

REFERENCES

[1] M. Niemann and M. Preuß, "Constructive Generation Methods for Dungeons," presented at the Seminar-Thesis in Procedural Content Generation for Games. Westfalische Wilhelms Universitat Munster, 2015.

[2] B. Lavender and T. Thompson, "The Zelda Dungeon Generator: Adopting Generative Grammars to Create Levels for Action-Adventure Games," 2015.

[3] I. Antoniuk and P. Rokita, "Procedural generation of multilevel dungeons for application in computer games using schematic maps and L-system," in *Intelligent Methods and Big Data in Industrial Applications*, Springer, 2019, pp. 261–275.

[4] J. Togelius, E. Kastbjerg, D. Schedl, and G. N. Yannakakis, "What is Procedural Content Generation?: Mario on the Borderline," in *Proceedings of the 2Nd International Workshop on Procedural Content Generation in Games*, New York, NY, USA, 2011, p. 3:1-3:6. doi: 10.1145/2000919.2000922.

[5] N. Shaker, A. Liapis, J. Togelius, R. Lopes, and R. Bidarra, "Constructive generation methods for dungeons and levels," in *Procedural Content Generation in Games*, Springer, 2016, pp. 31–55.

[6] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation," presented at the European Conference on the Applications of Evolutionary Computation, 2010, pp. 141–150.

[7] M. Shaker, N. Shaker, J. Togelius, and M. Abou-Zleikha, "A progressive approach to content generation," presented at the European Conference on the Applications of Evolutionary Computation, 2015, pp. 381–393.

[8] J. Dormans, "Cyclic Generation," in *Procedural Generation in Game Design*, AK Peters/CRC Press, 2017, pp. 83–96.

[9] D. Karavolos, A. Liapis, and G. N. Yannakakis, "Evolving missions to create game spaces," presented at the 2016 IEEE Conference on Computational Intelligence and Games (CIG), 2016, pp. 1–8.

[10] J. Dormans, *Engineering emergence: applied theory for game design*. Universiteit van Amsterdam [Host], 2012.