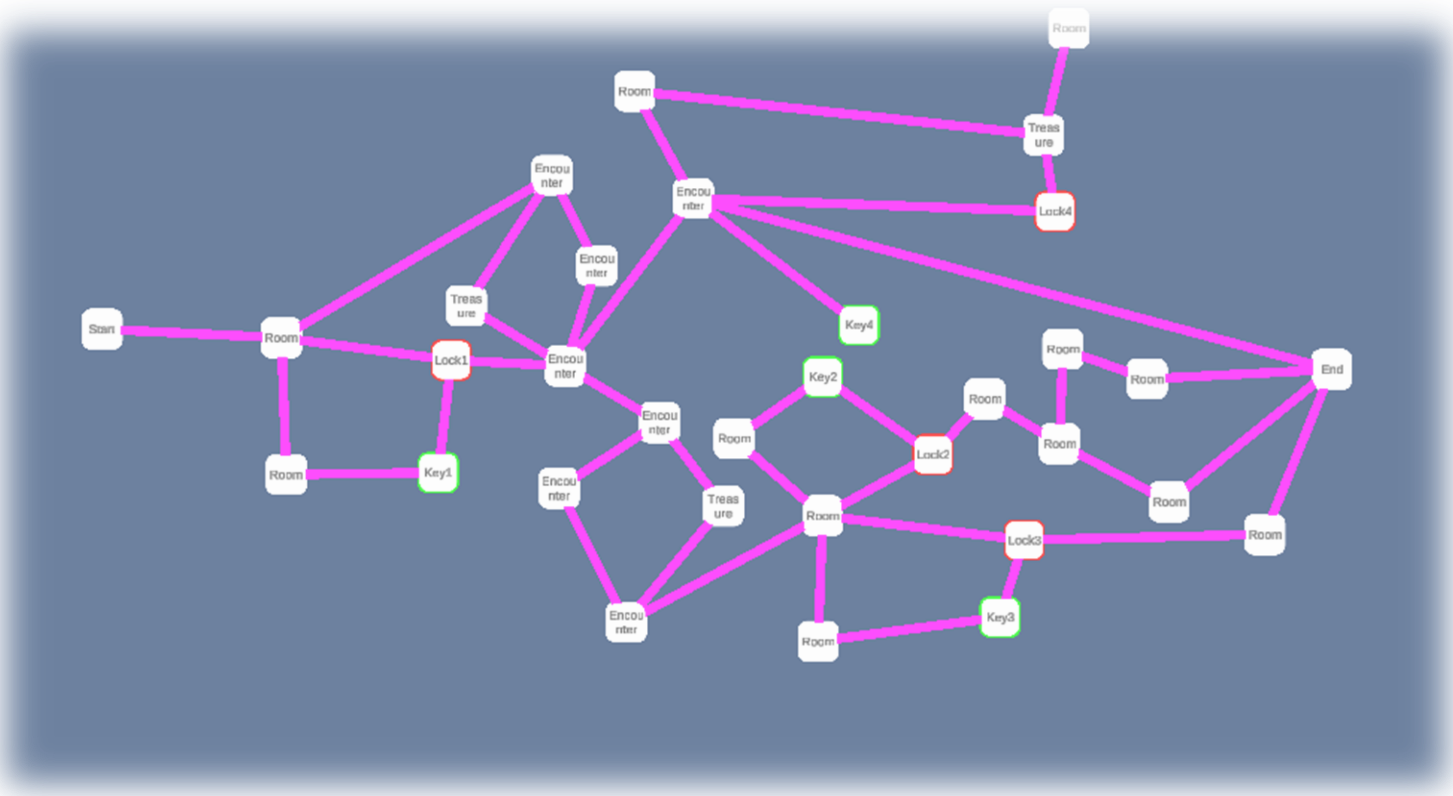
Lu Nicholas

Graph grammars rule complexity

Graduation work 2022-2023

Digital Arts and Entertainment

Howest.be



CONTENTS

ABSTRACT

This paper will go into detail of how and why graph grammars should be used. First a deeper look is taken into how graph grammars work and why they should be used over other procedural generation techniques. After that we take a look into level design, because the designer has a lot of control of the generation some understanding over level design should be reached to know what to aim for. To test everything a tool was built to utilize graph grammars to generate levels. Experimenting is done with the tool to find out how one can start using grammar rules to get the desired results.

Keywords: Games, Procedural content generation, level design, generative grammars

INTRODUCTION

With the rise in popularity of the roguelike genre, more and more games started using procedural content generation to create their levels. There are a lot of different methods/algorithms for the generation of game levels in games, but since the levels are procedurally generated there is a limit to how much the design of the level can be influenced to fit different situations. With the usage of graph grammars during the generation process the level will always stick to rules created by the designer. With those rules the designer has a lot of power over how the generation process of the level will proceed.

This paper will provide a general overview of the usage/workings of graph grammars and will go more in depth on what the complexity and limitations of graph grammars are for a good level generation.

The application of graph grammars will end with the generating of the graph and will not go into creating an actual level with the generated graph, but methods for creating levels will be lightly looked into.

LITERATURE STUDY

To understand more about the essence of graph grammars the next topic will look into how they originated, explain how graph grammars work and how they are used to generate levels. Compared to other procedural content generation techniques we will find out what graph grammars excel at and where it lacks.

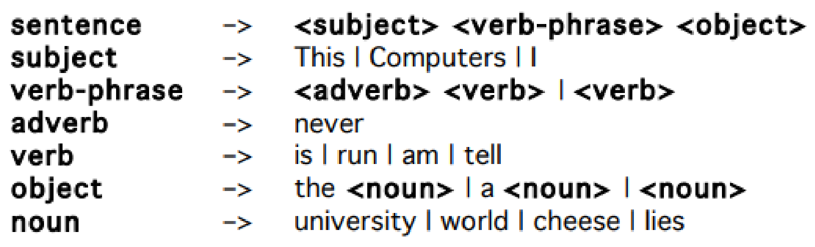
1. PROCEDURAL LEVEL GENERATION

1.1. GENERATIVE GRAMMARS

Generative grammars are the basis of all grammar generated content, it can have many different forms and uses.

1.1.1. FORMAL GRAMMAR

A grammar specifies a set of rules that describes how to either generate extra content or modify already existing content into different structures according to a set of rules. These grammars were first used in formal grammars, which is used in defining the syntax and semantics of a language. The rules that construct these grammars would define the structure of the sentence, and then go more in depth to what these structures could exist out of. A sentence's structure can exist out of subjects, verb-phrases and an object. Then again, the structure can go more into detail such as verb-phrases existing out of adverbs and verbs, all the way until the lowest level is reached and exact words are chosen to represent the content.(fig. 1) Some of the major reasons to introduce these grammars to graphs is that it can control the entire layout of the graph and its subgraphs just as formal grammars.



sentence	->	**<subject> <verb-phrase> <object>**
subject	->	This I Computers I I
verb-phrase	->	**<adverb> <verb>** I **<verb>**
adverb	->	never
verb	->	is I run I am I tell
object	->	the **<noun>** I a **<noun>** I **<noun>**
noun	->	university I world I cheese I lies

Figure 1: example of formal grammar rules
Taken from [1]

1.1.2. GRAPH GRAMMARS

Unlike the sentence structure used in formal grammars, the rules in graph grammars are made with graphs. Some basic graph theory knowledge is needed to understand how the grammars are formulated and used to generate levels.

GRAPHS

Graphs are often utilized in the mathematical field as they easily represent data. They are useful for analyzing interconnected data and have many techniques for solving real-world problems[6]. These graphs exist out of a collection of nodes which are points of interest that are connected to one another through edges.[6] There can be many different types of graphs, nodes and edges. Nodes can represent many different types. The nodes in this project can represent an encounter with an enemy, a room containing treasure or even the room that contains the final boss of the level. Edges can be directed or undirected meaning that it can only be traversed one way or bi-directional, this is most often represented with an arrow. If there is an arrow it is considered directed and can only be traversed in one direction, while a lack of an arrow indicates it is bi-directional. A graph with undirected edges would be an undirected graph while a graph with directed edges would be a directed graph(fig. 2), on top of being directed or undirected a graph can be a multigraph if multiple different edges are allowed between the same two nodes, if this is not allowed it would just be a simple graph(fig. 3). For the upcoming study on the level design and complexity of grammar rules, an undirected simple graph is used, due to this the levels generated are not as complex as they would be with a directed graph.

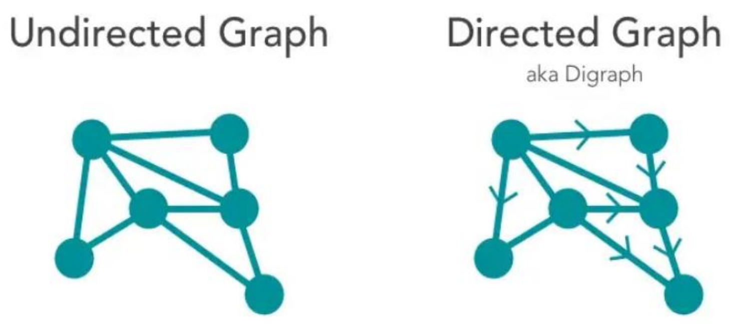


Figure 2: example of an undirected graph vs a directed graph
Taken from https://towardsdatascience.com/graph-theory-basic-properties-955fe2f61914

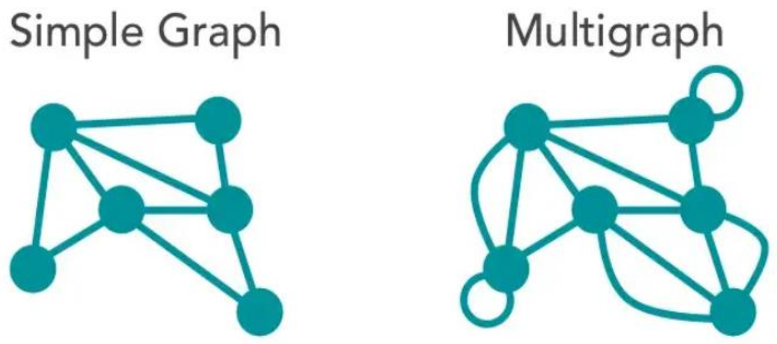


Figure 3: example of a simple graph vs a multigraph
Taken from https://towardsdatascience.com/graph-theory-basic-properties-955fe2f61914

GRAMMARS

The rules used for graph grammars consists of 2 subgraphs. A subgraph is a smaller graph that can be found as part of a graph. The first subgraph is what must be found in the graph and the second subgraph is what will replace the found subgraph. These grammar rules can range from rearranging existing nodes in the graph(fig. 5) to adding in new nodes. These new nodes could be other types such as locked nodes or nodes containing keys(fig. 6). When a part has been replaced this can open up opportunities for other rules to start being able to apply as well. With these grammar rules designers can make sure that some rooms are always followed up by whatever is desired.

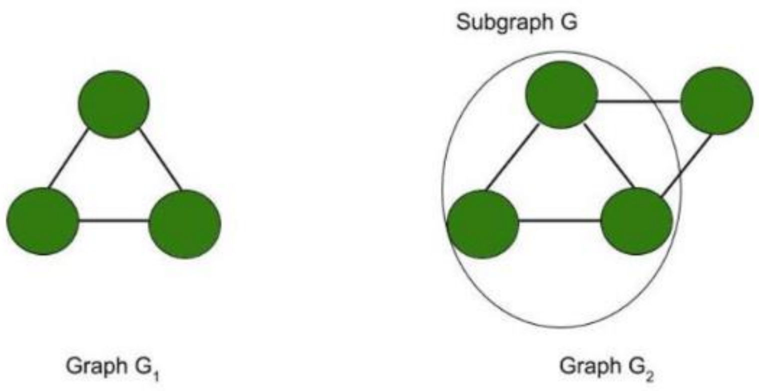


Figure 4: example of a graph G_1 that is a subgraph inside graph G_2 taken from: https://www.geeksforgeeks.org/

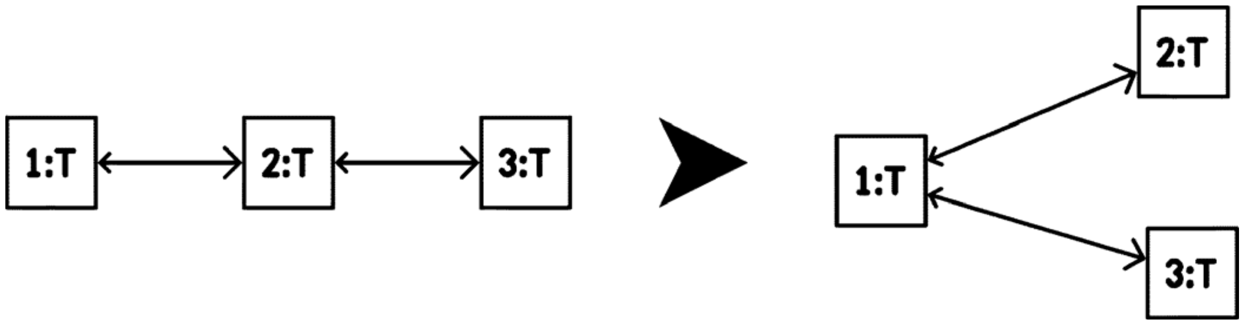


Figure 5: rearrange rule, the numbers are used to identify corresponding nodes on the left and right hand side[2]

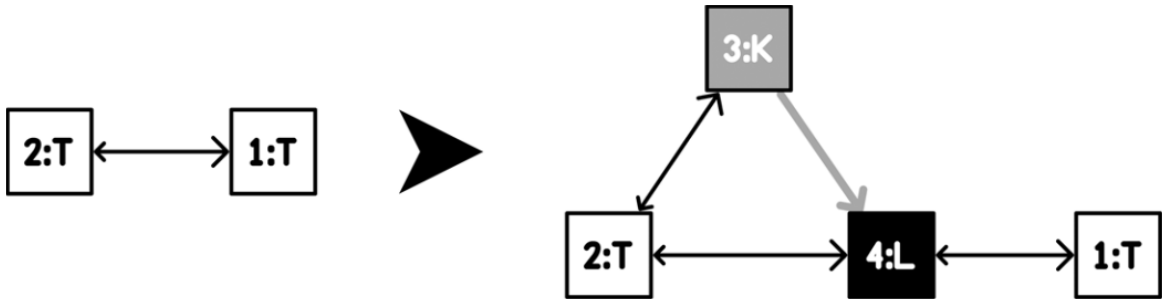


Figure 6: rule that adds a key and a lock, colors are only used to help identify node types[2]

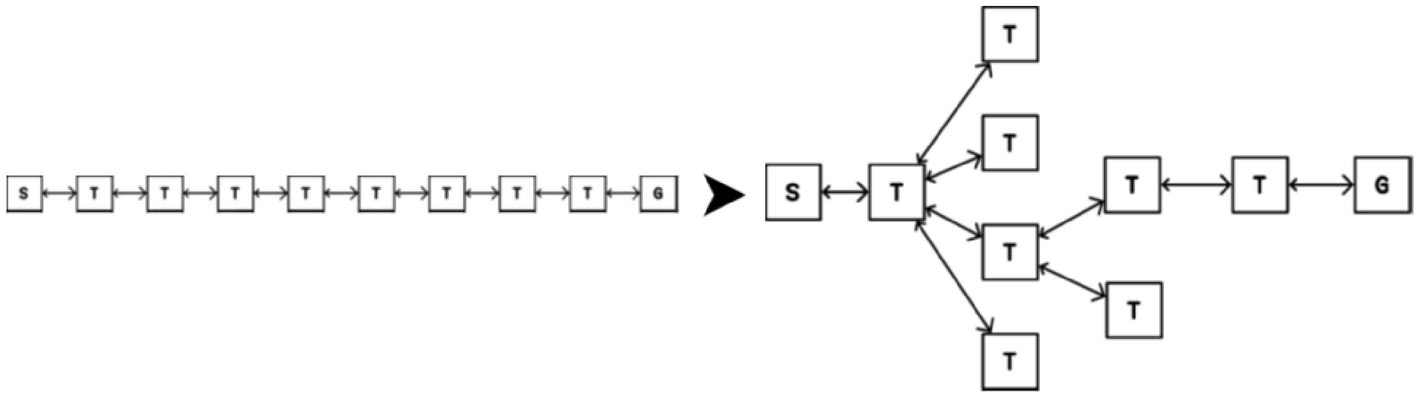
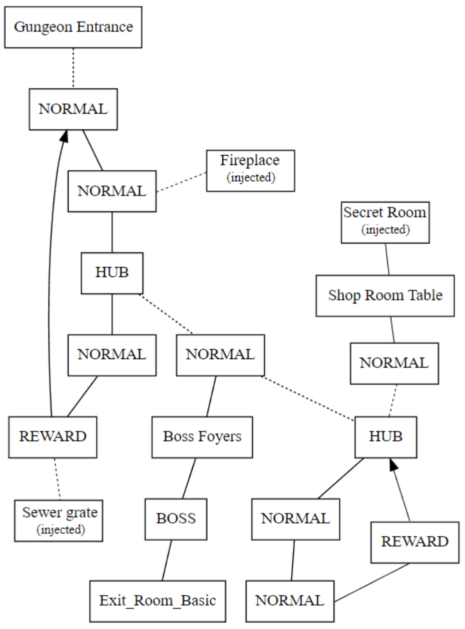


Figure 7: The effect of a single rule(fig2.) being applied repeatedly on the same graph[2]

LEVEL GENERATION

The only thing the generation process does is create a graph that represents the layout of the level with different types of rooms. The level itself still has to be generated. This is where graph grammar end, because the generation of the level itself is separate from graph grammars and require their own technique. There is no single way to translate the graph to an actual level, it heavily depends on what type of world is desired. The graph layout is still just a layout, it is up to the designer to decide what this will be translated into. A classic dungeon style where nodes can be different sized rooms connected by corridors(fig. 8) or even an open world approach where the nodes are islands connected by bridges(fig. 9). Connections also do not necessarily have to be physically present in the level, rooms could stick to each other with the edge only indicating what rooms are accessible(fig. 10).



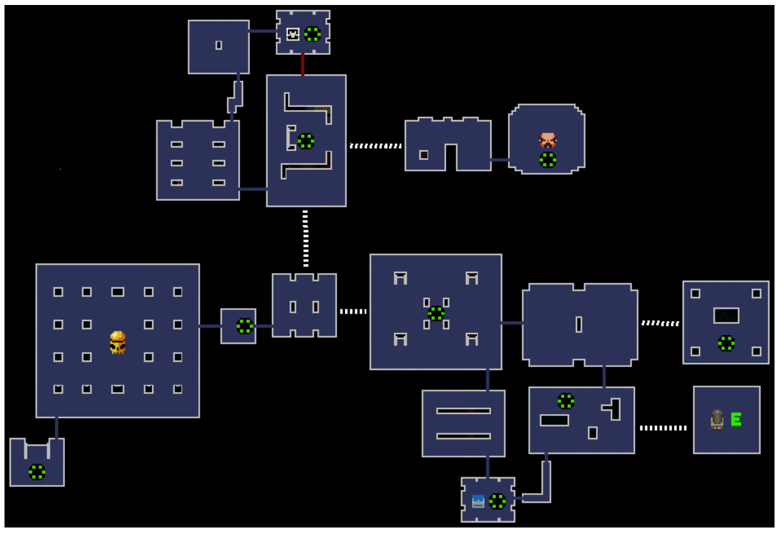


Figure 8: Example level with corresponding graph taken from the game “Enter the gungeon”
Taken from: https://www.boristhebrave.com/2019/07/28/dungeon-generation-in-enter-the-gungeon/

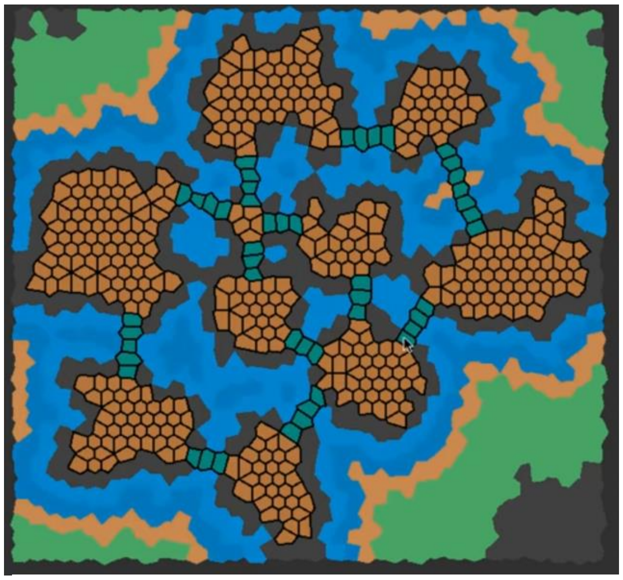


Figure 9: nodes being island with edges being bridges[7]

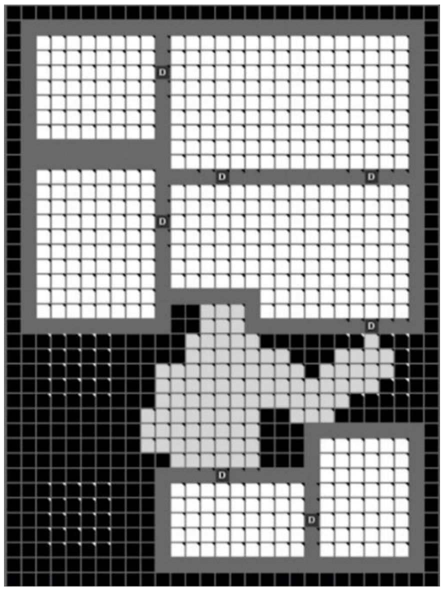


Figure 10: rooms connected without corridors

1.1.3. OTHER APPLICATIONS

Using generative grammar for games does not end at generating levels, some other interesting procedural content generation uses for generative grammars besides level generation includes things such as generating an economics system where nodes can represent either goods or a process that produces other goods, with edges representing the flow of the goods[8]. Another example of the use of generative grammars is the procedural generation of missions. A mission can have a basic sequence of events. Those events then get split up into tasks needed for completing the mission, such as transforming a fetch quest into a discovering and fetch variant. The splitting up of the events is more like a formal grammar setup then it is to a graph grammar.[9]

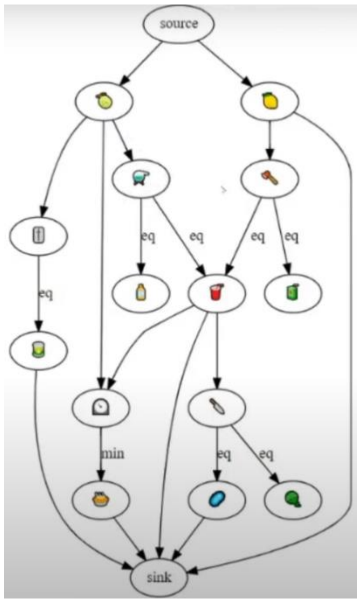


Figure 11: example of goods flow[8]

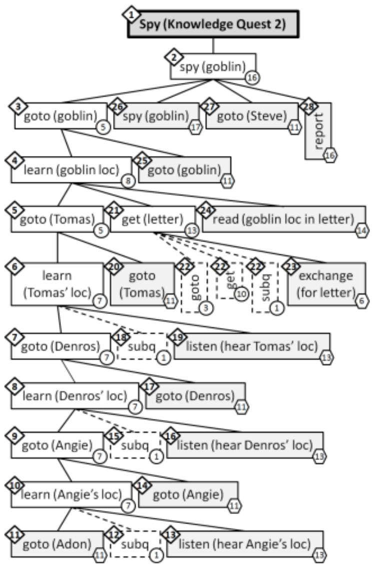


Figure 12: Analysis of a procedurally generated spy quest [9]

1.2. SIMPLE ROOM PLACEMENT

This technique is one of the more basic ways of procedural level generation. In its most basic form it is quite random and there is not much control to influence the end result. Random rooms get placed when there is space and stops placing when enough rooms have been reached. The next step is to connect these rooms. The most common method is to create a triangulated graph of all the rooms and create a minimum spanning tree to see which rooms get connected in the end. These connections are not necessarily the final connections, extra connections can be randomly added again for more interesting levels. This method of connecting rooms ensures that all rooms are reachable and that there are not too many connections. With this generation technique everything is mostly left to chance, a designer does not have many ways to alter the generation process.

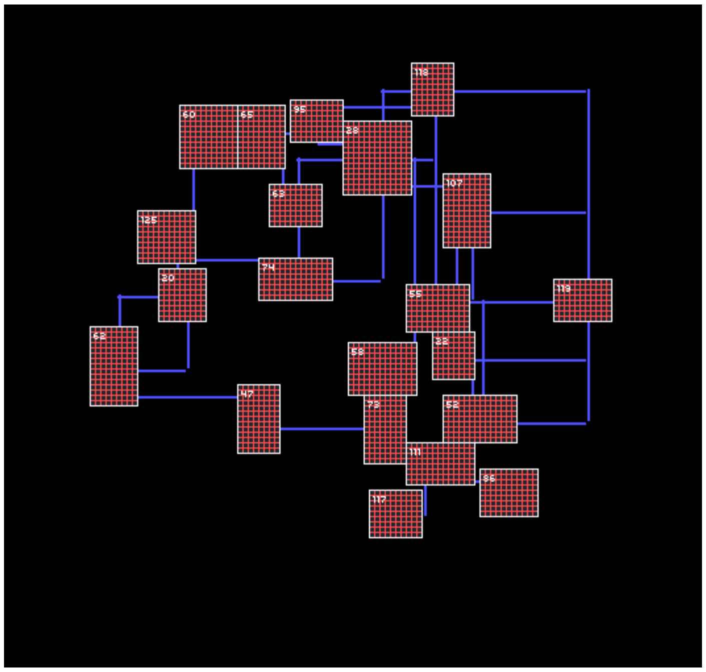


Figure 13: end result of Simple room placement
Taken from www.gamedeveloper.com

1.3. CELLULAR AUTOMATA

Cellular automata is a tile based way of generating levels, like for graph grammars a set of rules is set in place that will decide how the level will generate. But a big difference with graph grammars, apart from cellular automata not utilizing graphs, is that in cellular automata every tile only looks at its respective neighbours instead of the level as a whole. A selected tile will decide what will happen with itself such as becoming a wall or empty according to its immediate neighbours. The generation starts by filling the tile grid with walls, hand placed or through a seed, then every iteration each tile will apply the rules and decide what will happen with itself. This can go on for however many iterations are desired. With this technique a level designer already has a lot more influence as the rules can be set to whatever is needed. When rules are made, tiles will always react the same way to a setup. This means the end result will always be the same if the same seed/rules are used. A negative is that with a lot of iteration it becomes harder to predict what will happen the next iteration. This form of generation also does not guarantee that every section in the grid is reachable, although this problem can be fixed by removing all the unconnected areas with a post generation fix. A popular example of a cellular automaton is “Conway’s game of life” which is used for simulations. This simulation has a specific set of rules(fig. 14) and is not used for level generation, but the same rules could be applied for level generation. This technique is mainly used to generate cavern like structures and not dungeon like levels.

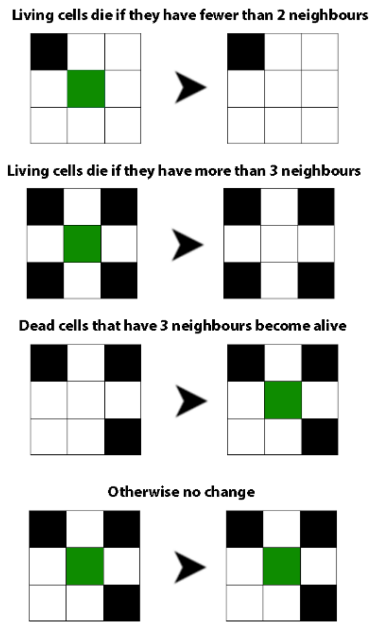

Living cells die if they have fewer than 2 neighbours
Living cells die if they have more than 3 neighbours
Dead cells that have 3 neighbours become alive
Otherwise no change

Figure 14: rules used for conway’s game of life

1.4. DRUNKARD'S WALK

A Drunkards walk is a generation technique where a whole grid is filled with walls and a random point in the grid gets selected to be empty, from this point on a random cardinal direction is chosen and that tile will also be marked empty. Keep choosing random directions until satisfied and eventually whole sections of the grid will be opened up. With this technique every part of the map is guaranteed to be reachable. This form of level generation is mostly used for cave-like levels. There is a possibility to steer the algorithm in certain directions more than others to have more control but that is where the control mostly ends.

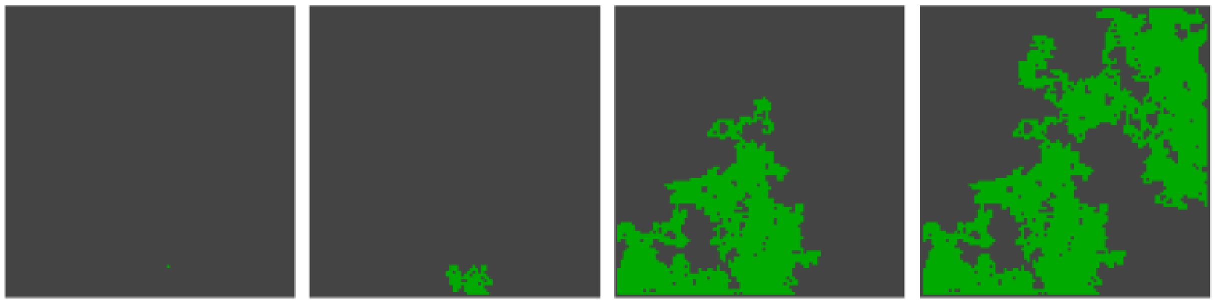


Figure 15: Drunkard Walk level initial state, level after 101 steps, level after 2001 steps and the final state of 4001 steps[4]

1.5. PROS/CONS OF GRAPH GRAMMARS

Graph grammars are very effective and versatile and can be used in many different ways to generate content for games, but for now this will only focus on the level layout generation. Compared to previous explained procedural content generation techniques, graph grammars excel at creating vastly different maps depending on what is needed in the gome due to the control the designer has creating the grammars. Being able to control the layout by implementing what rooms should exist at what position followed up by whatever is desired. Nodes can be injected at exact positions, such as having a shop as the third room. Different paths each with their own difficulty that all lead to the same outcome can be implemented. Other generation techniques do not have this form of control. But creating these grammars is not easy at all and requires a good understanding on grammars to predict what the end result could look like. But when a good understanding is reached and the designer knows how to use the grammars, creating a lot of different levels with different needs gets a lot easier and is worth all the effort of understanding grammars and creating them.

2. PROCEDURAL LEVEL DESIGN

What should be the end result of using graph grammars, how should a level look like after it is done generating? This will mostly be different for every game/genre depending on what is needed for the gameplay and story. This chapter will go more into exploration dungeon like levels.

2.1. BRANCHING TREE

A common form of generation is branching out like a tree structure. Starting from a single path the level grows out with branches to create different paths and make the level feel less monotonous than a single path from start to end. This does create a backtracking problem where if you get to the end of a branch the player has to run back to one of the origin points of the branch. Of course this does not have to end with just a tree like structure, the designer can put in extra rules to edit the base structure, branches can connect with each other to create cycles or create a one-way path from the end of the branch to the beginning to avoid backtracking. The basis of a tree is easier to generate but also a lot more limiting.

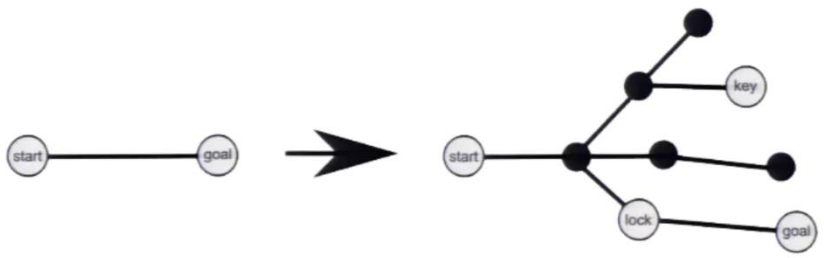


Figure 16: branching tree structure[12]

2.2. CYCLIC GENERATION

While branching structure adds cycles after the main generation, cyclic generations' entire focus is on getting cycles. Focusing generation on creating cycles gives more of a natural feel to the levels. This comes due to how cycles are used everywhere and not only in games. Cycles can be found everywhere even in real life. From the layout of a building, to parks and road networks, humans have the tendency to use cycles.[5] This prompted level designers to put in cycles everywhere in their games from multiplayer shooters that have multiple paths of attacking and ambushing opponents to single-player games where you have to traverse dungeons and unlock paths. When experimenting with creating levels, cyclic levels will be tried to be achieved for the most natural feeling levels.

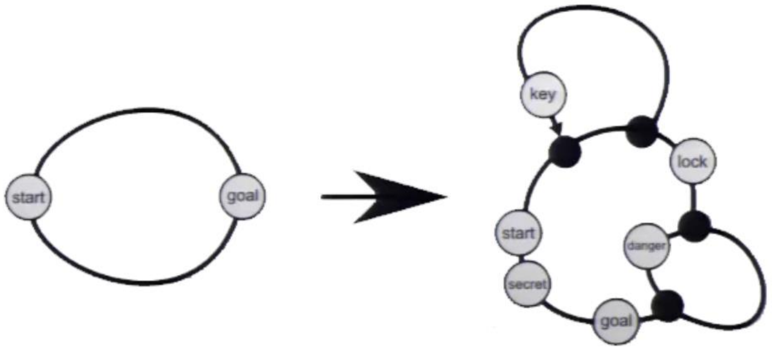


Figure 17: cyclic generation[12]

CASE STUDY

1. INTRODUCTION

To figure out more about the complexity of rules and how many rules are needed to create well designed cyclic levels using graph grammars. Some trial and error had to be done with a graph grammar generation system to create a lot of different test levels. So the start of, this study will explain the making of the tool used for creating levels.

2. SOFTWARE

Unity version 2021.3.15f1 was used for the project due to the existence of scriptable objects and a plugin called "Edgar" that has a basic graph scriptable object to use.

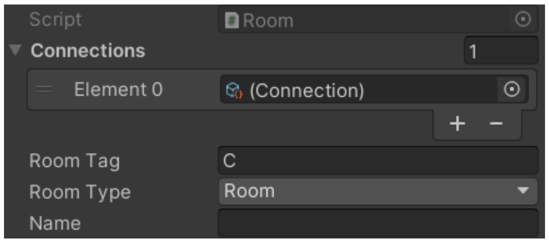
Visual studio 2019.

3. GRAPH GENERATION

The method used for finding subgraphs and the graph rewriting will be explained in the following section. To get more understanding of how the replacing and finding works, which could create more of an insight to how rules should behave and be created.

3.1. GRAPHS

A graph object that is used in the grammar rules can simply be created by right clicking in the asset browser and creating one via the create tab. Double clicking inside the graph creates a node. When the node is selected some important changes can be made to the node(fig.18). The room type can be changed to one of the room types present in the enumeration combo box. Many more of these types can be added easily in the code. Another important variable is the Room tag, this string will define how the connections of a node should react when getting replaced. This system of implementing for controlling edges has been chosen to give full control over how the edges on a node will react when that node is changed. This tagging system will get better explained in one of the next sections. Those were the most important settings for the node, there are some variables that don't have real impact that are residues from the Edgar plugin[10] that gets used. Holding control when drag clicking from one node to another will connect those nodes.



Script	Room
Connections	1
Element 0	(Connection)
Room Tag	C
Room Type	Room
Name	

Figure 18: settings available for nodes

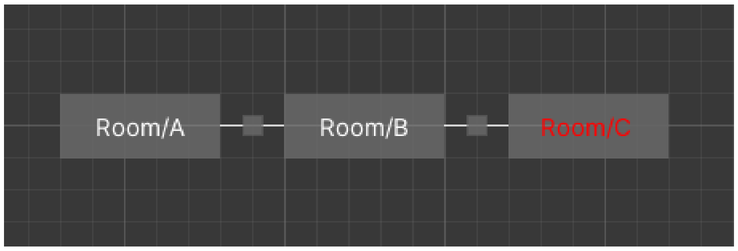


Figure 19: example of how a graph could look for a rule

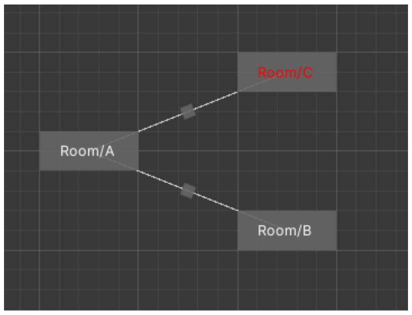
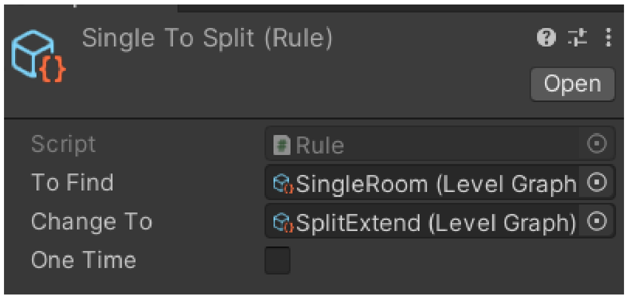


Figure 20: example graph

3.2. GRAMMARS

The grammar rules are made from scriptable objects which stores data as assets for the project to use at run time. A grammar will take two graphs, a “To Find” graph and a “Change To” graph as well as a bool that can be toggled on if the rule should only apply once. An important factor to the tool is that when a rule adds a new key, a corresponding lock should be added as well in the same rule. This must happen so the system knows what key opens what door. After the key and the lock have been added some rules can be used to separate the key further from the lock, since they have to be added at the same time they will be relatively close together so some separating would not be bad.

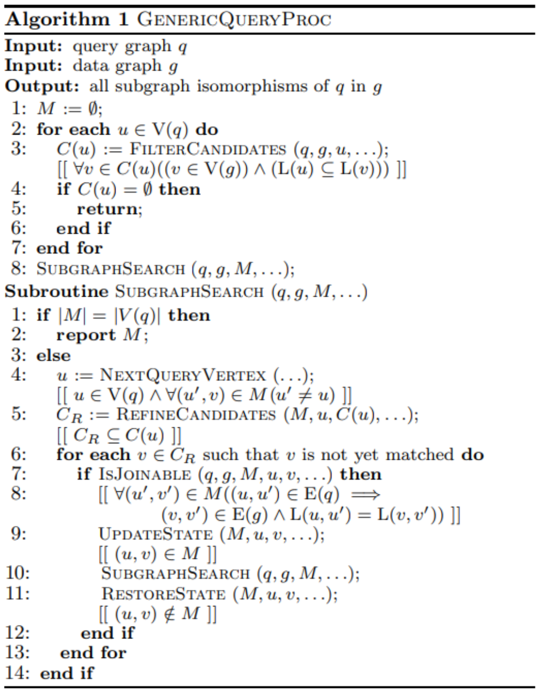


Script	Rule
To Find	SingleRoom (Level Graph)
Change To	SplitExtend (Level Graph)
One Time	☐

Figure 21: variables able to set for a grammar rule

3.3. SUBGRAPH ISOMORPHISM

When the program has selected a to apply, the most important step will happen. Subgraph isomorphism is a task in determining if with two given graphs the first one is contained in the second one as a subgraph. The algorithm used for this is a version of a Generic Subgraph Isomorphism Algorithm(fig. 22) that is modified to fit the needs for this project. One of the modifications applies makes the algorithm find every instance of the subgraph instead of only finding the first instance of the subgraph. This version of the algorithm uses a mapping system to map the nodes in the subgraph to every node that it potentially could be in the graph. After every node in the subgraph is mapped the algorithm will start checking the connections of the mapped nodes and look if they connected correctly according to the subgraph. Every found subgraph will then be saved and one will randomly get chosen to represent the graph of the grammar.



```
Algorithm 1 GenericQueryProc
Input: query graph $q$
Input: data graph $g$
Output: all subgraph isomorphisms of $q$ in $g$
 1: $M := \emptyset$;
 2: for each $u \in \mathrm{V}(q)$ do
 3:     $C(u) :=$ FilterCandidates $(q, g, u, \ldots)$;
        [[ $\forall v \in C(u)((v \in \mathrm{V}(g)) \wedge (\mathrm{L}(u) \subseteq \mathrm{L}(v)))$ ]]
 4:     if $C(u) = \emptyset$ then
 5:         return;
 6:     end if
 7: end for
 8: SubgraphSearch $(q, g, M, \ldots)$;
Subroutine SubgraphSearch $(q, g, M, \ldots)$
 1: if $|M| = |V(q)|$ then
 2:     report $M$;
 3: else
 4:     $u :=$ NextQueryVertex $(\ldots)$;
        [[ $u \in \mathrm{V}(q) \wedge \forall (u', v) \in M (u' \neq u)$ ]]
 5:     $C_R :=$ RefineCandidates $(M, u, C(u), \ldots)$;
        [[ $C_R \subseteq C(u)$ ]]
 6:     for each $v \in C_R$ such that $v$ is not yet matched do
 7:         if IsJoinable $(q, g, M, u, v, \ldots)$ then
 8:             [[ $\forall (u', v') \in M ((u, u') \in \mathrm{E}(q) \implies$
                    $(v, v') \in \mathrm{E}(g) \wedge \mathrm{L}(u, u') = \mathrm{L}(v, v'))$ ]]
 9:             UpdateState $(M, u, v, \ldots)$;
                [[ $(u, v) \in M$ ]]
10:             SubgraphSearch $(q, g, M, \ldots)$;
11:             RestoreState $(M, u, v, \ldots)$;
                [[ $(u, v) \notin M$ ]]
12:         end if
13:     end for
14: end if
```

Figure 22: Generic Subgraph Isomorphism Algorithm [11]

3.4. GRAPH REWRITING

The last step to complete the graph grammar procedure is replacing the found subgraph with the graph that the grammar rule contains. There are many complex ways to calculate how the connections should react when the subgraph gets replaced. Such as the double-pushout or single-pushout approach, but I have opted for a more self-made approach that was inspired by Joris Dormans paper about graph grammars[2]. The method used in this project works with a tag system where the designer must specify for each node what node it corresponds with in the second graph of the grammar. This way the designer has full control of how the connections outside of the subgraph will react. As example when trying to add a node between two rooms, the first graph in the grammar will specify what we are looking for, in this case two connected nodes(fig. 23) so that we can create one in between them. These nodes must be tagged so the connections they have outside of the given “to Find” graph can be remembered. When creating the second graph in the grammar, the tagging of “A” and “B” can make a difference, either the new node will be in between the two nodes or branch off. Let’s say in this case the room tagged “A” is connected to the start and the room tagged “B” is connected to the end, that means that in the second graph the node we tag “A” will be connected to the start and the node tagged “B” to the end, this can give different result depending on if the middle node or the last node is tagged “B”. If the middle node is tagged “B” that means the middle node will be connected to the end, and the newly added node will only be connected to the node tagged “B”, but if the new node is added between A and B it will be added in between those two nodes and work as originally wanted. (fig. 24)

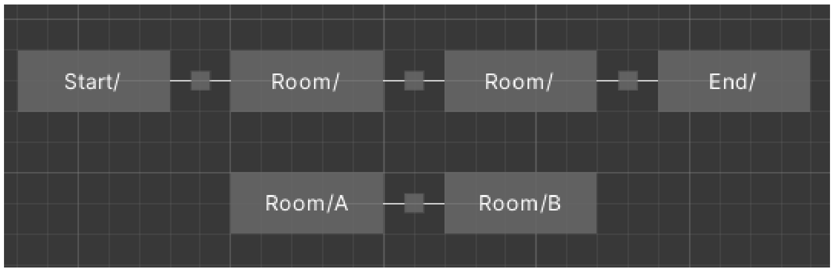


Figure 23: finding two rooms in a smiple straight line graph

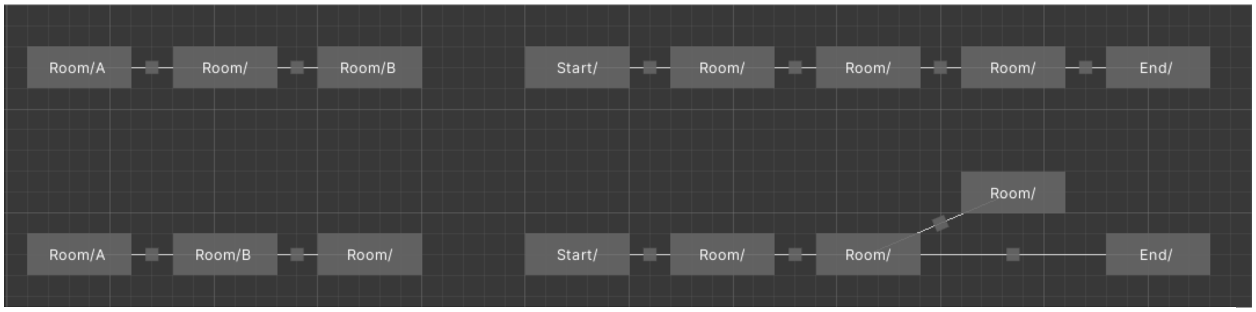


Figure 24: the difference that tags makes on a simple rule together with the result it would make if they were applied on figure 23

EXPERIMENTS & RESULTS

Levels will be created using the graph grammar system and try to get a layout with different cycles, different difficulty paths and aim for levels used such as in “Enter the Gungeon” or “Unexplored” but on a bigger scale. Good level design is subjective but bad level design can be noticed quickly.

1. GRAMMAR COMPLEXITY

The experimentation on creating rules will focus on the complexity of said rules. This will look into how complex a rule should be and how many of those rules are needed to create levels.

1.1. COMPLEX GRAMMAR RULES

1.1.1. RULE COMPLEXITY

Example of what could be considered complex rules are the creation of a hub inside a dungeon, a space where the player has different paths he can choose to go down to. When starting with a simple graph(fig. 25) different preset complex rules(fig. 26) can be chosen to represent the hub. One rule does not suffice to be considered an entire level but it represents more the layout of what a section in the level will look like. Made some grammar rules that would fall under the complex category and created some test levels with those rules. These levels work with some ambiguous node types such as hard paths and easy paths so the difficulty is a lot easier to control. These paths will then be entirely created by the designer to have more control of where the harder paths(fig. 27) and the easier paths would form in the level. As well as the difficulty of those paths. When creating the rules the designer has an easier time controlling the size of the level, the size is as big as the designer makes all the graphs.

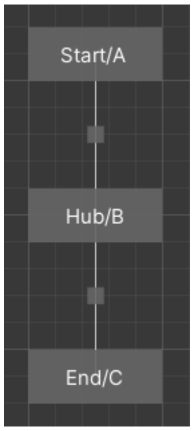


Figure 25: simple hub start

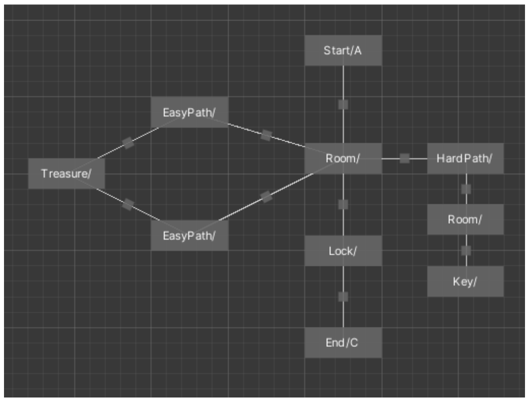


Figure 26: example of a complex graph starting from a hub

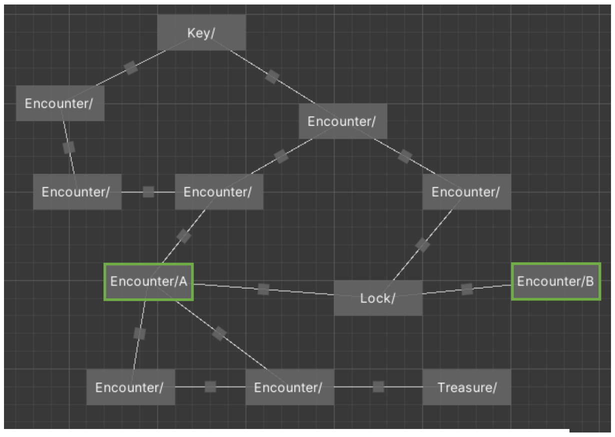


Figure 27: example of a hard path where path gets enter from the node marked ‘A’ and destination is the node marked ‘B’

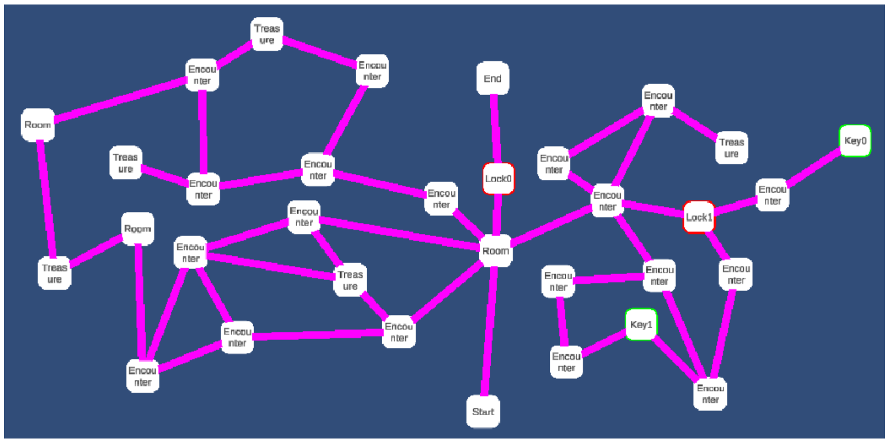


Figure 28: example 1 level after using complex rules

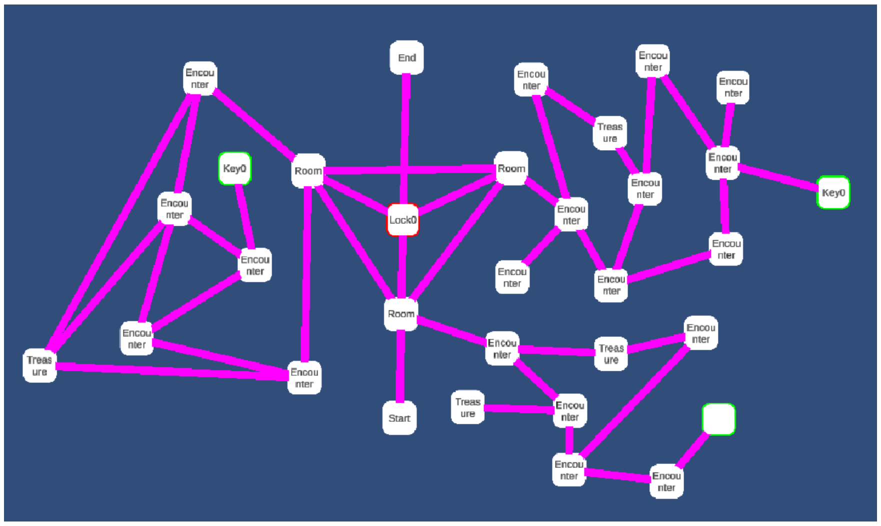


Figure 29: example 2 level after using complex rules

1.1.2. RULE COUNT

A lot less of these complex rules get applied, this happens because the complex rules are very specific and are only found once or twice in the graph. The designer has full control on how many rules get applied without putting a limit on how many rules can get applied in total. This comes from how the designer has an easier time predicting all the possibilities of how rules will generate. The only things adding more complex graphs does is add variety when generating levels multiple times using the same set of rules. That also means that the chances of getting the exact same level gets smaller with each rule added, but the possibility is still relatively high. Depending on how many nodes can generate different sets, a lot of rules will be needed to get a good variety of different levels.

1.2. SIMPLE GRAMMARS RULES

1.2.1. RULE COMPLEXITY

Simple rules will be a lot less complex and only exist out of a handful of nodes. But getting the rules just right to achieve a well made level is already a lot harder than using complex rules. This comes from how much harder it becomes to predict how the rules will react to one another. It also becomes a lot harder to limit the size of the entire level as stopping the rules completely by getting the level to a specific point is hard to predict with this type of ruleset. A major advantage is the uniqueness, if there is enough variety the chance of getting the exact same level will be very small so each playthrough will be different, but this comes at the cost of some control and predictability. Simple rules also do not change so much in one go, this make it more likely that other rules will still apply. Rules keep getting applied and keep opening up other opportunities for other rules. But some rules should be created to limit the generation in certain parts of the graph. When limiting the graph predicting of what can happen becomes a lot easier, and with those rules the graph does not keep endlessly generating in a concentrated area.

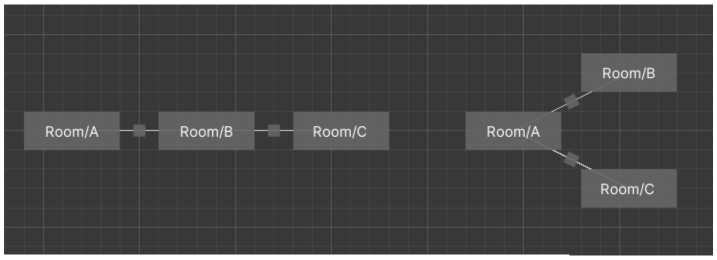


Figure 30: example rule of splitting

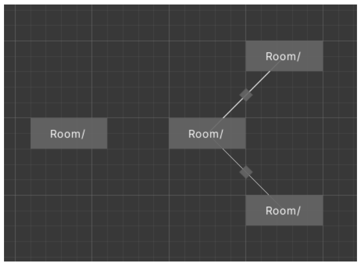


Figure 31: example rule of adding 2 nodes

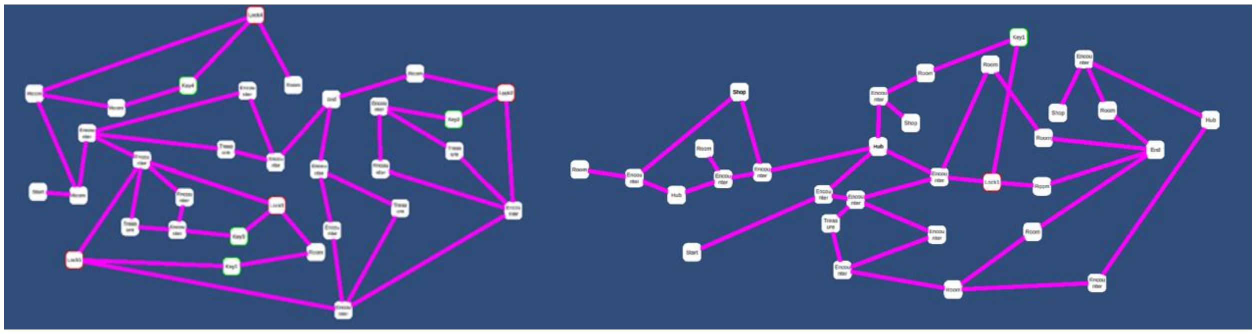


Figure 32: two different examples of levels that used the same ruleset

1.2.2. RULE COUNT

Here again more rules add more variety, but with the simple rules the impact is much bigger. Due to how simple the rules are, a single rule could create something interesting if it is able to keep applying itself on the graph. The amount of rules used for previous shown levels(fig. 32) are only a total of 10 rules, which could be argued is quite a lot but with just 10 rules 2 vastly different levels came out of it and a lot more can be expected. The shown levels are not the best while there are some basic loops, keys and locks it still feels off, so increasing the amount of rules even more would make the end result even more complex.

1.3. MIXED GRAMMARS RULES

1.3.1. RULE COMPLEXITY

These set of rules will be a mix of previous explained simple rules and a slight variation of the complex rules. The complex rules should be changed in a way that the small rules will apply after the complex rules have been added to the level. This way the big outline of the complex rules set the basis of the level and the smaller rules will make the level feel more unique by altering the results of the complex rules. That way the importance of how well thought out small rules are is not as complex as they are on their own. Because the bigger complex rules are in charge of the layout such as deciding where difficult and easy paths should end up while the small rules purely exist for creating uniqueness.

1.3.2. RULE COUNT

The number of rules will be both the amount of complex rules and small rules added together, but both rule groups do not need as many rules as they each need separately. When it came to complex rules the amount needed was for more variety, but since the smaller rules will add the variety this time not as many complex rules are needed. When the designer wants more different base layouts then more complex rules should be created and added. The amount of smaller rules is also a bit less than it needs on its own, because now they do not need to make an entire level layout, this is what the complex rules are for. The main use for these smaller rules is changing the layouts of complex rules so they are not the same every new instance.

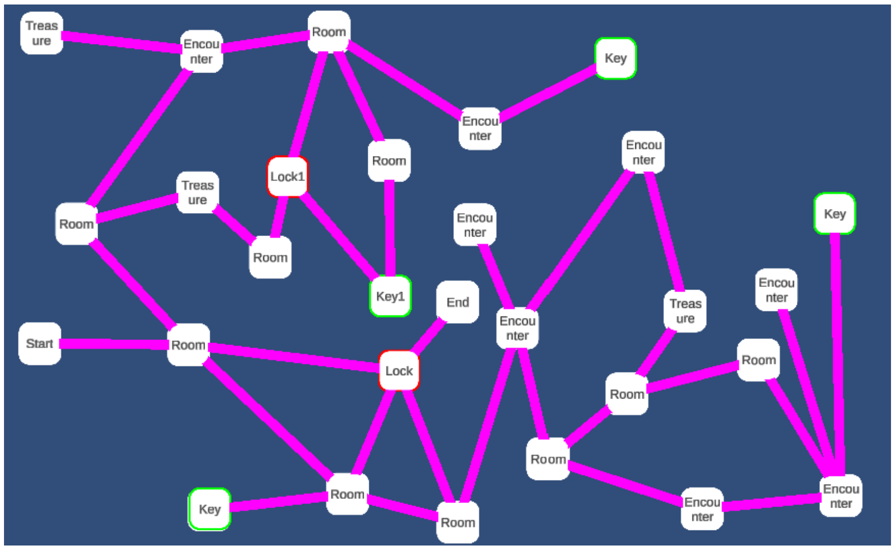


Figure 33: Example of level using a mix of both rules

2. RESULTS

2.1. COMPLEX RULES RESULT

When compared to the other results creating levels using complex grammar rules should be utilized the least. A lot more effort is needed to make variety because the designer is basically just designing full parts of a level. Although this way gives the most control over the level the uniqueness is not something fitting for this type of level generation. The chances of getting the same level are too high and getting lower chances comes at the cost of letting the designer make more sub-area level layouts. This comes down to almost just creating your own levels, graph grammars do not have much to do with this style of generating and other procedural generators that are more focused on mixing around existing sub-areas could be used.

2.2. SIMPLE RULES RESULT

This method will create unique levels almost every time when setup with enough rules, but creating good levels using smaller grammar rules is a lot harder. When using simpler rules a lot of learning and experimenting of how they will react with each other is needed if high quality levels are expected. After spending quite some time testing a lot of different rules, there still is a feeling that better results can be reached. But when knowing what can be reached when utilizing the best of graph grammars and trying to reach that point, just playing around with the tool can be quite interesting. The most diverse and unique levels will be created using this, but this is also the hardest way of creating them.

2.3. MIXED RULES RESULT

This is using the best of the complex rule with the utility of the simple rules. While some level design has to be done to create the basis it nearly does not have to be as perfect as opposed to only using complex rules. The basis created here is mainly setting the main paths that the player can go down. The designed part of the level will get modified enough by the smaller rules to introduce the randomness and uniqueness in the level. This will give an easier time than only using simple rules and much more control over the basic layout.

DISCUSSION

There are many ways of utilizing graph grammar for level layouts each having their pros and cons when using them. It is up to the designer to learn the ins and outs of how to use and make the best of it. Nothing can be perfect and of course certain generated levels will feel better than others. Creating a set of grammar rules that will create the perfect level with a different structure each time will be close to an impossible task. Due to the graph grammar generation tool being self-made there are some limitations to what rules are possible, which made it took longer to experiment and get results. Being able to immediately generate amazing levels with graph grammars is hard to achieve, experimentation will definitely have to be done to get used to the system. Previous study gives a good starting point on how one could start using graph grammar and build upon it even further.

CONCLUSION & FUTURE WORK

CONCLUSION

Graph grammars are amazing for level layout generation definitely when compared to some of the more simpler level generators. There is a big barrier to entry as there are no public tools for using graph grammars or explicit tutorials on how to exactly make one. After being able to create levels using graph grammar the added research and experimenting needed to understand the concept and utilize the technique effectively makes graph grammars hard to use. There is no definite way on how to use graph grammars, different techniques result in different outcomes each having their own good and bad points.

When trying to have the most control where variety matters little, very complex specific rules offers the best. If you are prepared to spend a lot of time on figuring out what impact rules have on the generation and make countless of setups to reach something great with amazing variety, simple rules are a good choice. Lastly when more control is needed and layouts cannot be fully random, a mix of both complex rules and less complicated ones can give an easier time to create layouts. But why limit to only complex or simple rules, limiting oneself causes one to just narrow the view of how levels can be generated. There will be times when complex rules are needed and there will be times when simple rules are needed, having the best of both choices is in my opinion the obvious winner when trying to create any level.

FUTURE WORK

The tool is definitely still a bit basic and could use some upgrades. The layout for using the tool is a bit hard to grasp. The layout together with the graphing system could both use some updating for ease of use.

There are a couple limitations on how the tagging system works and could definitely be improved or be totally revamped. The system is confusing and needs quite an explanation to understand.

Depending on how big the levels get there might need to happen some changes to the subgraph isomorphism algorithm. Currently not the most efficient algorithm is used, and when the level gets quite big there are some framerate drops bound to happen.

The next great step would be to make a conversion to actual playable levels, I would want to make it a more classic dungeon style with rooms connected by corridors, but I will play around with some different styles and see what can be reached. When the complete generation and translation is made, implementing it in a game would be amazing.

BIBLIOGRAPHY

[0] Ehrig, H. (2002). *Introduction to graph grammars with applications to semantic networks.*

[1] Johnson, M., & Zelenski, J. (2012). *Formal Grammars.*

[2] Dormans, J., & Bakkes, S. (21011). *Generating Missions and Spaces for Adaptable Play Experiences.*

[3] MATHEMATICAL GAMES The fantastic combinations of John Conway's new solitaire game "life"

[4] Naußed, D., & Sapokaite, R. (2021). *Evaluation of Procedural Content Generators for TwoDimensional Top-Down Dungeon Levels.*

[5] Youtube video: AI and Games, The Secret Behind Unexplored: Cyclic Dungeon Generation

[6] IJREAS. (n.d.). *INTRODUCTION TO GRAPH THEORY.*

[7] Youtube video: naughtyyt, Graph Grammar based Procedural Generation for a Roguelike

[8] Gritter, M. (2020). *Procedurally Generating Economics with Graph Grammars (and Math).*

[9] Doran, J., & Parberry, I. (2011). *A Prototype Quest Generator Based on a Structural.*

[10] Nepožitek, O. (2022). *Edgar - Unity*. Retrieved from https://ondrejnepozitek.github.io/Edgar-Unity/

[11] Lee, J., Kasperovics, R., Han, W.-S., & Lee, J.-H. (2012). *An In-depth Comparison of Subgraph Isomorphism* Algorithms in Graph Databases.

[12] Youtube video: BUas Game, EPC2016 – Joris Dormans – Cyclic Dungeon Genetation