Unique Dungeon Design:
Computational creative methods for generating video game dungeons

Brandon Atkinson
Computer Science Department
BYU
bj.atkinson92@gmail.com

Ryan West
Computer Science Department
BYU
west.ryan.k@gmail.com

Abstract

Video games are becoming one of the most popular forms of modern entertainment. Along with this growth in popularity has come an exponential increase in both the quality of games, as well as the cost to produce them. Much of this cost is due to the large amount of content that must be hand crafted by expert designers. In this paper, we propose a computational creative system to automate this process of content creation. Specifically, we focus on a creative system capable of designing unique and interesting dungeon levels for games. Though our system is abstract and can be applied to many different games, we focus on two main game engines: Tile World (a modern remake of the popular Chip's Challenge game) and a custom game written in PuzzleScript. We also provide several examples of our generated levels and evaluate their strengths and weaknesses.

Introduction

Video games have become a highly popular form of entertainment in today's world. Along with their growth in popularity, the budgets for modern AAA videos games have exploded. One of the reasons for this high cost is the difficulty of producing large quantities of high quality content. Highly skilled designers and artists need to be brought onto a project to create fun and interesting content for a player to experience. In addition to designers, play testers and quality assurance individuals need to be hired to assess a game's content in order to iteratively improve its design. This costly and time intensive process has been somewhat alleviated in some modern games by leveraging procedural generation. Games like Minecraft and No Man's Sky have been able to produce vast and interesting interactive game worlds for players. Importantly, procedural generation allows these games to accomplish all this without heavy reliance on designers to hand craft those worlds.

Using procedural generation is not a new technique in video games. In fact, it has been used successfully for many decades in games like Rouge and Elite. Their developers leveraged procedural generation to make games filled with surprising and interesting challenges. Some game genres, however, have been unable to effectively leverage procedural generation. While Rouge-likes and exploration games have seen huge progress in procedural generation, more intricately designed games, such as Metroidvanias and puzzle games, have been unable to successfully leverage such techniques. In this work, we seek to overcome the challenges unique to puzzle and Zelda-like games by leveraging procedural generation combined with computational creativity methods to create interesting dungeon designs. While this work borrows much from the procedural generation space, we feel that it is more than just a simple rule based system and is, in actuality, computationally creative. We believe our system is computationally creative for two reasons: the techniques we use to create levels are derived from computational creativity literature, and the levels generated by our system often look almost as good as those designed by a human creator using the same limited set of mechanics, exhibiting complicated puzzles that increase in difficulty throughout the level set.

Game Engines

Our system is designed to be abstract enough to interface successfully with many different games. In order to achieve and validate this, we generate levels for 2 distinct games. The first is Tile World which is a modern remake of the classic puzzle game Chip's Challenge. The second is a custom game created using a game engine called PuzzleScript. The key features of these two games are described below.

Chip's Challenge Chip's Challenge is a top-down puzzle game that was first released for the Atari Lynx in 1989 (Orland 2015). The game consists of a two-dimensional grid of tiles which form obstacles for the player. Players must navigate through puzzle obstacles in order to collect all of the chips and reach the exit. while players progress through the level set, level difficulty increases and more game mechanics are steadily added. The original game included a large set of tiles which provided a wide variety of game mechanics such as enemies, teleportation, and slippery ice. Our system, however, only uses a small subset of the game's original mechanics since we wanted to focus more on the quality and creativity of our system's output rather than recreating all of the complicated game mechanics.

PuzzleScript PuzzleScript is a simple game engine focused on helping developers prototype puzzle games. We leveraged this tool to create a simple game with mechanics

Tile	Name	Mechanic
⬜	Empty	Can always be visited
⬛	Wall	Can never be visited
🟦	Exit	Reach this to finish level
▦	Chip	Must collect all to finish level
▣	Barrier	Must collect all Chips to cross
🔑	Key	Opens Lock of same color
🔒	Lock	Opened by Key of same color
🔥	Fire	Passable with Fire Boots
👢	Fire Boots	Allows crossing of Fire
🌊	Water	Passable with Flippers
🩴	Flippers	Allows swimming in water
🟫	Block	Push into water to allow crossing

Table 1: Chip's Challenge tiles and mechanics.

similar to those present in Chip's Challenge. The PuzzleScript game has the player collect keys and power-ups, as well as collectables called crystals. In order to beat a level, the player must navigate through the full level space and collect all the crystals in the level. PuzzleScript is a web based game engine, meaning that it was simple to use across various different platforms. This proved to be important as we used both a Windows machine as well as a Mac while developing this system. PuzzleScript was also simple to use and easy to iterate on the game's design allowing us to move faster and try a variety of different mechanics.

Methods

Our system is designed as a single Creator which orchestrates the generation, validation, and evaluation of level sets (see figure 1). The Generator, Validator, and Evaluator are, in turn, comprised of smaller sub-components as well. The Generator is made up of a Graph Generator for creating a genotype of the mission, a Level Space Generator which creates random partitions in a level's map, and a Genotype to Phenotype Translator which converts the genotypic mission graph and partitioned level map into a complete level. The Validator is comprised of a Solver and a Sokoban Solver which verify the solvability of a level. The Evaluator is comprised of individual metrics which gauge the quality of a level.

Graph Generator

The Graph Generator is the first element in our system. This module generates what we refer to as a mission graph. A mission graph is an abstract representation of a level. It represents the connections between different critical elements of the level. In our system we consider critical elements to be the start and end of a level, keys, locks, and level collectables. Each of these elements must be visited by the player in order to complete the level. Thus the level graph represents the path a player must take in order to complete the level. However, this path is not a physical path, rather it is a conceptual path representing tasks the player must complete without any regard as to the spatial location of those tasks.

The graph generation process builds a graph recursively by continually adding locks to the graph. At each step, the generation adds one or more locks as children of the current node. Then the process moves randomly backwards through the graph to find an appropriate location to place a corresponding key. This ensures that a key can never be located behind the door it is intended to open, preventing a vast number of failed graphs being generated. After several iterations of this progressive building, the algorithm terminates and an end node is added.

The final step of generating the mission graph is ensuring that all branches of the graph are required. The algorithm traverses each node of the graph and determines if it contains a required collectable (e.g. a key or power-up). If there is no required collectable at the end of a path, one is added.

Level Space Generator

While the Graph Generator is responsible for producing a genotypic representation of a mission, the Level Space Generator is tasked with randomly partitioning the level space. The partitioned space acts as a landscape upon which a mission will be placed. A subsequent module in the system is tasked with mapping the mission graph into the level space created by this generator.

The Level Space Generator is a relatively simple component of the system but it still contains 3 distinct steps. In the first step, the Generator draws random rectangles of different sizes in the level. The rectangles are filled with empty tiles and have a border of wall tiles. This produces a level filled with many differently shaped partitions or rooms. While this is enough content for the subsequent modules to fit a mission in the space, it isn't very interesting because the level space looks too clean and perfect; therefore, the Level Space Generator manipulates the level space even further to make it more interesting. The next step is to add noise to the level space in the form of randomly placed wall tiles. This added noise increases the apparent complexity of the level space and also provides obstacles for the player to navigate while playing the level. The final step of the Level Space Generator is to potentially mirror the level space across either the x-axis, y-axis, or both. By adding symmetry to the level space, levels appear more visually appealing to players, increasing their perceived value and design intentionality.

Genotype to Phenotype Translator

The Genotype to Phenotype Translator is responsible for taking the mission graph genotype and a partitioned level space and converting it into a completed level. Originally, the Translator randomly placed mission elements in hopes that they made would form a valid mission (missions are filtered based on solvability in the Validator). However, that

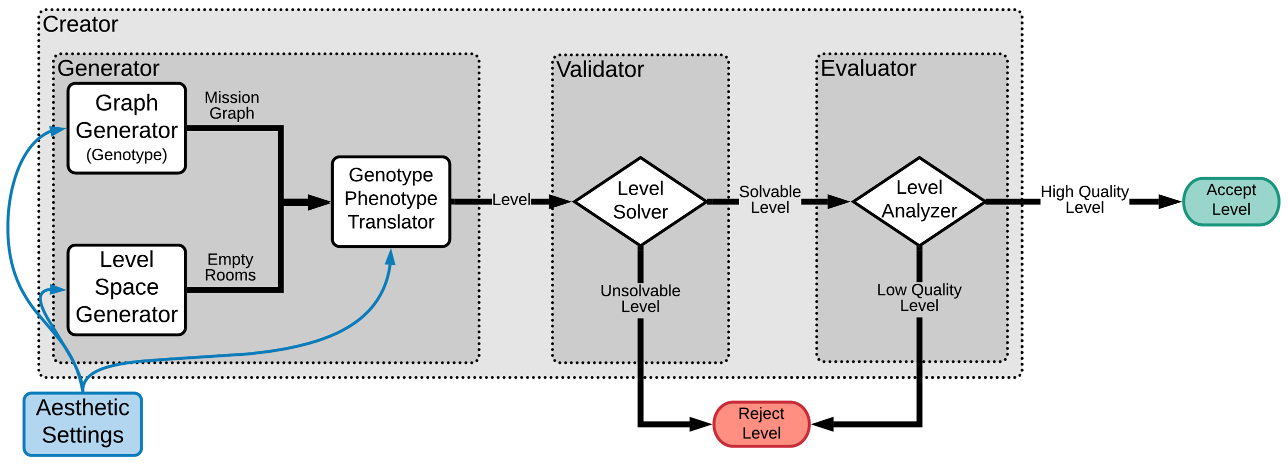

Figure 1: An overview of our our dungeon level creation system. The system starts with the Generator which creates an abstract mission graph. This graph is then mapped onto a generated level space to create a level. This level is then evaluated, first to determine if the level is solvable, and then to determine the quality of the level. The Creator module then repeats this process several times to create a set of levels that exhibit interesting level designs, as well as a progressive difficulty curve.

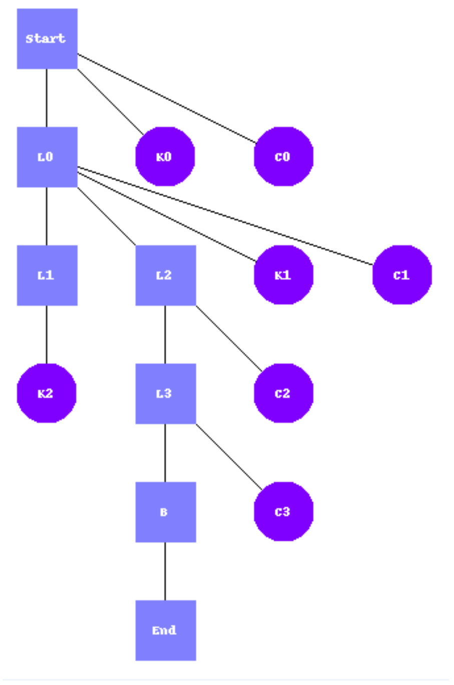

Figure 2: A level graph is a representation of the critical path through the level. Importantly, it does not represent the spatial relation between critical elements but rather a dependency graph where higher elements must be completed before lower elements in the graph. Squares in this figure represent locks and circles represent collectables like keys and power-ups

method proved to be very computationally expensive as it generated many unsolvable levels before finally stumbling upon a solvable level. This system could occasionally generate small levels with simple mission graphs, but large levels with complex missions were out of the question. When randomly placing mission elements in a level, the set of generated unsolvable levels grows far more rapidly than the set of solvable levels. That means that the probability of randomly generating a solvable level became vanishingly small as size and complexity increased, thus making complex levels was near impossible.

In order to overcome this limitation, we had to study the problem we were trying to solve and understand the deeper computer science principles that were at play. Throughout our time working on the project, our understanding of the problem expanded in several leaps of inspiration. The final fact that we realized about this problem was that instead of trying to transfer the genotypic mission graph directly onto the level space, we could abstract the level space to a graph representation. We constructed a spatial graph (a graph representing the level space) by treating each room as a node and adding edges between nodes when rooms were adjacent to one another. Once the level space was converted to a graph, our problem shifted from fitting a graph into a 2D grid of a level (a problem for which no off-the-shelf algorithms exists) to the problem of finding a graph inside a graph. That exact problem has already been solved and is called the subgraph isomorphism problem. Specifically, the problem is defined as determining whether a subgraph of graph A is the same shape as graph B. Since this algorithm is already implemented in the NetworkX graph analysis package (Hagberg, Schult, and Swart 2008), we were able to use their implementation of the algorithm to perform this step. Once the Genotype-Phenotype Translator obtains a mapping from

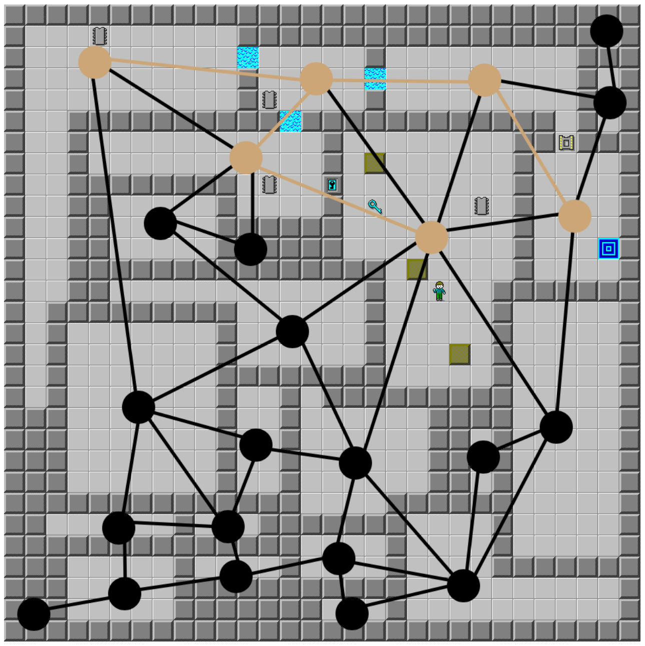

Figure 3: An illustration of how a simplified mission graph is remapped into a level space. The level space is converted into a spatial graph of nodes representing rooms and edges connecting adjacent rooms. A subgraph of the spatial graph is found which is isomorphic with the mission graph. The gold highlighted nodes and edges represent the portion of the spatial graph which is isomorphic with the mission graph.

mission nodes to room nodes, it places tiles representing the mission obstacles into the level. It places locks in the walls that separate rooms and places keys, Sokoban blocks, and collectables in the rooms that come after their parent nodes in the mission graph. Once all of the tiles corresponding to mission nodes are placed, the level is complete.

Validator

Although the Genotype to Phenotype Translator generates complete levels, there is no guarantee that these levels are solvable. Because unsolvable levels prevent the player from progressing through the game, our system rejects all levels that are unsolvable to ensure the player doesn't run into any issues during gameplay. It is the job of the Validator to determine whether a given level is solvable.

Solving a level is a complicated process, requiring the system to both understand a valid sequence of progression in the mission graph as well as the layout of the level itself. The system must simulate the movement of a character through the level, keeping track of the changes of state that occur when a player obtains keys and items, opens doors, and pushes blocks. As the system simulates a player in the level, it must ensure that every node in the mission graph is reachable at the correct time. If the player can't move to reach the next key or lock in the mission graph, the level is impossible to solve. This could occur in the unlikely case that a key was accidentally placed behind the door it was intended to unlock or the case that it is impossible to push a Sokoban block to the correct position due to an obstruction. As the system is checking to ensure that all the nodes are reachable at the correct time, it also checks to ensure that nodes that occur later in the mission graph are not reachable. When nodes are reached too soon, the level does not correlate very well with the mission graph and the player can solve the level without having to complete all of the tasks in the mission graph. It is essential that the system also checks for these trivial solutions because otherwise the Creator would be biased towards creating levels that weren't as complex as it expected.

As the system simulates the movements of the player, it needs to understand how to travel from point A to point B in a level. Our Level Solver implements the A* path finding algorithm to calculate those paths. The calculation is relatively simple when the player is just walking from one place to another. The player is only allowed to move to empty tiles, tiles with items that it can pick up, and tiles for which it has the proper items to visit (such as the red key for the red lock, or the flippers for the water). However, the task of path finding is much harder when the player must push a block from point A to point B. In order to determine if a block can be pushed a certain direction, the system must check whether the player can move from their current position to the opposite side of the block in order to push it. This turns into a double nested path finding algorithm where the path of the block is planned by planning the movements of the player around the block.

Evaluator

The final step of the Creator system is the Evaluator. The purpose of the Evaluator is to determine the overall quality of the levels produced by the Generator. Our Evaluator attempts to determine the difficulty of a level as well as how fun the level is. Since it is difficult for a computational system to determine what constitutes as fun, our Evaluator uses surrogate metrics to estimate these values. The Evaluator measures these attributes of the level and its solution and uses a weighted sum of those values to produce a score representing the level quality. Unfortunately, the relative weights assigned to each attribute were simply our best guess as to what was most important for a good level, rather than being based on any rigorous analysis. We had plans to record level statistics and then have other people play the levels. These players would then give each level a rating to help us identify any correlations between our metrics and a human player's assessment of level quality. Unfortunately, we didn't have enough time to complete such a study.

The metrics we used in our Evaluator are as follows:

- Number of steps in the level's mission
- Average number of moves required to complete each step
- Number of times the player must change direction while pushing Sokoban blocks

We chose these metrics because we believe that they are good surrogates for the quality and difficulty of a level. We used the number of steps in a level's mission because the number of steps required to solve a puzzle is usually an indicator of how difficult the puzzle is. The average number of moves required to complete each step is also an indication that the steps are more difficult. For example, if a key is located only 3 tiles away from the lock that it opens, then it won't be very difficult to find it and complete that step. The final metric we use to determine level quality is the number of times the player must change direction while pushing Sokoban blocks. While this metric may seem strange, we argue that higher values for this metric indicate that block pushing puzzles are more difficult. A Sokoban puzzle which requires the player to change direction pushing a block requires the player to navigate around obstacles and plan their movements more than puzzles which require the user to push the block in straight lines. Overall, we found that these metrics aligned well with our expectations when we played through levels ourselves.

Our system generates batches of about 20 levels and rates each one. Then it picks the top 5 levels from the level batch. The Evaluator only keeps about 25% of the levels that it generates in order to ensure only its best levels are presented to the player. While that may appear to be a waste of resources, we feel it makes sense to keep this small percentage of levels to ensure the average output from the system is high quality.

Once the Evaluator discards low quality levels, it uses the scores given to the levels to estimate reasonable time limits for each level. It assigns this time limit estimate to each level to encourage players to play more quickly. This improves the quality of all the levels because the player feels an added level of difficulty and intensity while solving puzzles under the time limit pressure. In reality the puzzles aren't too difficult, but under a strict time limit, each level becomes a race against the clock.

The final step of the Evaluator is to sort the levels into a level set. Most puzzle games are designed to include a progression from easier puzzles to harder ones. Our system tries to mimic this by sorting the puzzles based on their quality metrics and by the number of steps it takes to solve them. This ramp up in difficulty across a set of levels makes the level set appear more intentional in its design.

Aesthetic Settings

Our system relies on a variety of aesthetic settings used to influence the final output of game levels. These are hyper parameters configured by a user. Importantly, they do not provide direct control over the generation process. Rather, these settings influence the final output of the system by modifying internal probability distributions of the system. These settings are largely used to set upper and lower bounds on various aspects of level generation as well as modifying the likelihood of various generative events. These aesthetic settings are broken down into 4 major groups.

Level Space Aesthetic Level space aesthetics are settings used to modify the generation of the levels physical layout. Examples of aesthetic settings are `x_mirror_probability` and `y_mirror_probability` which control the probability of mirroring a level along an axis. Level space aesthetics also includes `rectangle_count` which changes how densely rooms are place in the level and `noise_percentage` which controls how "noisy" level layouts appear. Again, none of these settings directly control the final layout of the level but instead simply guide the algorithm to produce appropriate level layouts.

Mission Graph Aesthetic These settings are used to influence mission graph generation. This includes settings such as `min_depth` and `max_depth` which influence how large the generated mission graph is. `branch_probability` is another setting used to influence how often the level structure branches. Usually, levels that exhibit lots of branching are more difficult for players. This means that while we don't have direct control over how hard a mission our graphs will be, its final difficulty can be somewhat influenced using these settings.

Mission Aesthetic These settings are used to enhance the mission once the level space and mission graph have been combined into an actual level. They key settings in this section are the `fire_spread` and `water_spread` probabilities. When a level is built there is a chance that the system will include hazards like water and fire to make the level more interesting. To make these hazards appear more natural they are allowed to spread out and cover larger areas of the level. By adjusting these values we can modulate how "watery" and "fiery" the generated levels are.

Tweaker Aesthetic Tweaker aesthetics are the smallest subset of the aesthetic settings. This section includes only one value, `should_fill_unused_space`. This is a post processing step that fills all unused space in the generated levels. Doing this can reduce visual clutter in a level but can also make levels less visually interesting.

Results

As previously discussed, the output of our system is a level set which usually consist of 5 individual levels. These levels are arranged according to their estimated difficulty, easier levels first followed by harder levels. The intention is that this raising difficulty will follow a traditional difficulty curve. Difficulty curves are a theoretical tool used by game developers to ensure that levels increase in difficulty in line with a player's ability. Importantly, a good difficulty curve ensures the ramp up should be neither too fast nor too slow. If levels become too difficult they risk leaving a player frustrated and unable to progress. If level difficulty grows too slowly, however, the player may become bored and leave the game without finishing it.

While our system for establishing a difficulty curve is not very refined, it provides fairly good results. The levels in the set tend to start off linear and fairly easy to solve. They then start to branch and include more of our mechanics making them more difficult. Final levels in the set tend to have fairly long levels requiring players to explore, utilize spatial reasoning, and work quickly to successfully beat the level. Importantly, these levels include a tight time limit meaning that although puzzles may not be too difficult to solve, there

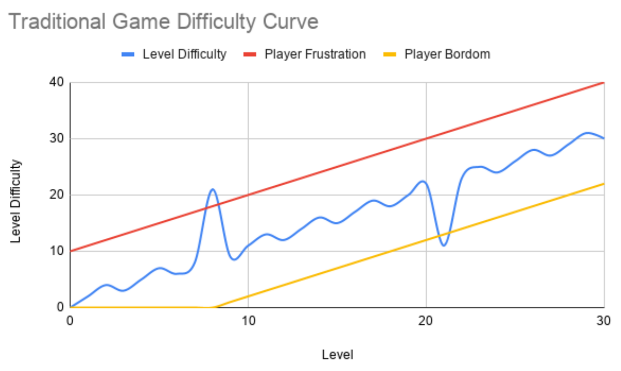

Figure 4: Difficulty curves are a common theoretical tool used in game development. Ideally, this curve generally trends up as later levels in the game are progressively more difficult. Additionally, an ideal difficulty curve follows the progress of the player such that no level is so difficult as to frustrate or so easy as to bore the them.

is always pressure on the player to move quickly, keeping the challenge from becoming mundane.

Another important aspect of our generated levels is their aesthetic appeal. Video games are an inherently visual entertainment medium and the way a level looks is an important aspect of its design. Often times the visual design of a level goes a long way to communicate the intentions of the designer. Our system uses several tricks to improve the appeal of our generated levels. We use parameters like hazard spread, level mirroring, and level noise to ensure a level looks and feels intentionally designed. Figure 5 presents several examples of levels generated by our system. These levels demonstrate the ability of our system to generate a variety of interesting and unique levels using a wide variety of mechanics.

One of the final additions to our system was the Sokoban system. While this seemed like a minor addition to the dungeon designer, it proved to be exceptionally important in improving the output of our levels. While the Sokoban puzzles are, necessarily small and simple, they add a surprising amount of variety to the levels. These puzzles often require very specific movements in order to solve and break up the pace of quick exploration to demand a slower more thoughtful approach to move forward. Requiring the player to switch between the two tasks of exploring and solving Sokoban puzzles greatly enhanced the player's enjoyment.

Evaluation

It is important to note that, while our system is impressive, it is not yet as capable as a professional designer would be. The levels our system creates only exhibit a small fraction of the available mechanics present in our various game engines. They also occasionally lack the intentionally one would expect from a professional and can sometimes appear poorly thought out (such as when a key is placed directly beside the lock that it opens). These levels also lack the difficulty that

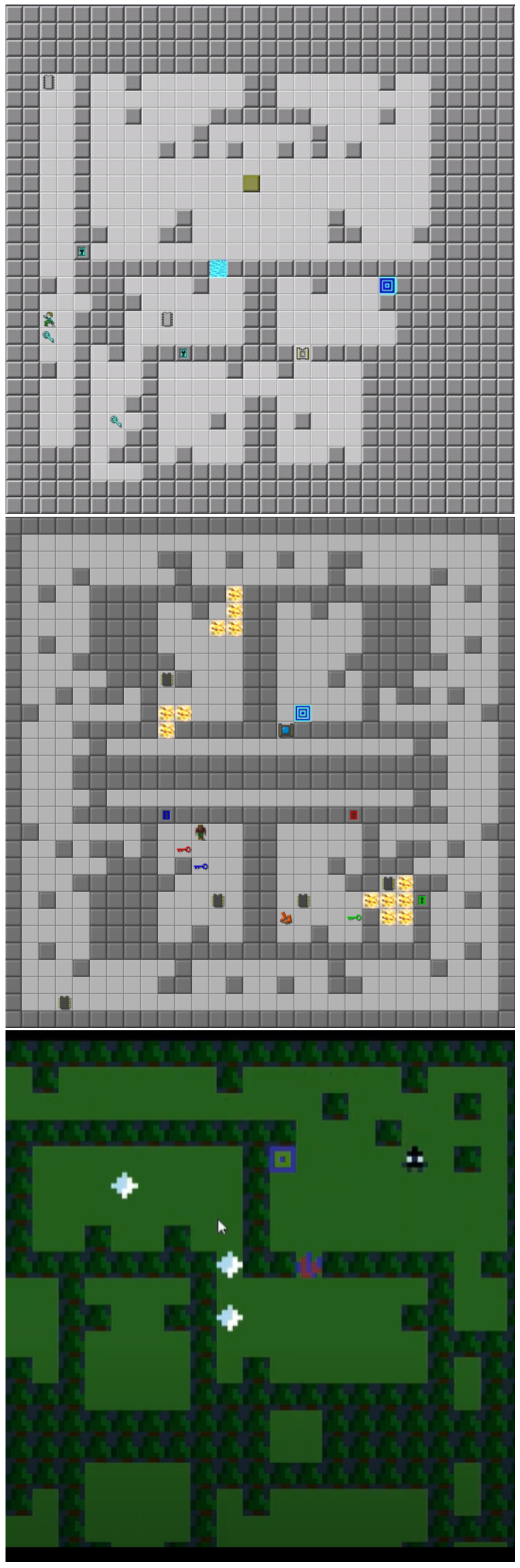

Figure 5: Examples of levels generated by our system. The first two levels are run in Tile World and show that our system is able to generate interesting levels of various sizes including various elements such as fire hazards, Sokoban block pushing and key/door pickups. The second level is an example of level mirroring which we can use to make a level feel more intentional in its design. The third image is an example of a level generated for PuzzleScript featuring collectable crystals and power-ups.

a player might expect in the final levels of a puzzle game. While many of the levels we generated did offer a moderate degree of challenge, none of them exhibited the truly devious and tricky puzzles often present in other influential entries in this particular game genre. While this means our system can easily generate enjoyable levels for a new player, an expert player would likely find our system trivial to outsmart.

Although our system has not reached professional human level performance, it still achieves impressive results. As mentioned previously, our system is capable of quickly generating many levels that both are interesting and appropriately challenging for a novice player. In fact, many of the levels our system produced were surprising even to us (the system's creators). This ability for the system to produce artifacts that surprise even us, the creators, is a strong indication that our system is actually computationally creative.

Our system also contains many of the key elements discussed in other computational creativity papers. In (Ventura 2017), the author explains that a computational creative system should have both genotypic and phenotypic representations and a module which translates one representation into another. Our system uses mission graphs as a genotypic representation of the level and contains a module which converts that mission graph into an actual level (the phenotype). Another aspect of our system was inspired by ideas about evaluation found in (Ritchie 2007). In this paper, the author asserts that there are two essential criteria in evaluating the creativity of artefacts produced by a system: ”quality” and ”typicality”. The author goes on to explain that ”quality” measures to what extent the artefact is a high quality product, while ”typicality” measures to what extent the artefact is an example of the class of creativity it is supposed to represent. Our system performs both forms of evaluation outlined by Ritchie. We assert that our Validator assesses the typicality of a level by analyzing whether the level is, in fact, solvable or not. If the level is not solvable, the Validator rejects it because it is not a representative of what a level really is: a solvable puzzle. We also argue that our Evaluator is performing the second type of evaluation mentioned by the author. The Evaluator measures certain attributes of the final level to decide whether it is difficult, fun, and interesting. This analysis amounts to an evaluation of the quality of the level.

One important aspect of our system that should be addressed is the various limitations that were imposed upon it. These were due in large part to the tight time constraints we had on this project. For example, we chose to limit the number of mechanics our generator could handle. Specifically, we chose to remove game mechanics that introduced randomness to the solution, such as moving enemies. This was necessary as it allowed us to make quicker progress on several fronts, including assessing a level's win-ability, difficulty, and general structure. Another aspect of the system that was simplified was the number of levels in a level set. Usually, level counts for puzzle games could reach into the triple digits. Our system chooses to simplify this significantly by only generating a set of 5 levels at a time. Again, this simplified the process of building the basic system and allowed us to iterate on the system's design faster. While many of these concessions were made early in the process of building our system, we kept our system flexible enough that many of these mechanics could be added into our system in the future. We will discuss this further in our future works section.

Another important consideration when evaluating our system is the restrictions on the size of levels we can generate. When mapping the generated graph into a level space, we use an algorithm designed to solve the subgraph isomorphism problem. Unfortunately, the subgraph isomorphism problem is known to be NP-complete. This means that the scalability of our system is limited. Given that the levels we are currently outputting are no larger than 30 x 30 tiles this is not a problem. However, we would have to make several changes to our system to handle significantly larger levels.

Future Work

One simple addition we would like to make to our system would be to add more game mechanics. Making more game mechanics available to the system would improve the levels created by the system by increasing the variability of the obstacles that players encounter. It would also allow for a greater range of difficulty and creativity in generated levels. Some early candidates for mechanics to add are hazards like ice, and doors tied to on/ off switches.

Another interesting expansion to our system would be to incorporate additional criteria into our level quality evaluation system. Currently we only assess a level's quality based on analysis of several attributes of the level's solution. While these factors provide a good baseline, incorporating factors like similarity to previous levels could greatly improve the Evaluator's performance.

One area which we started to explore but didn't have enough time to incorporate in the final project was that of automatic aesthetic metric weighting. We generated a large number of sample levels using our system and recorded the individual metric values that our system reported for the levels. Then we recruited several play testers to play through the levels and give their own ratings for the quality of the levels. However, the process of recruiting people and setting up their individual computers for play testing took too long to complete and we didn't have time to collect results from everyone. If we had collected enough results from players, however, we would have used that data to have our system perform regression between the recorded metric values and player's scores of those levels using a neural network. This analysis would have provided us with weight values we could use in our Evaluator in order to improve the correlation between what our system deemed as high quality levels and what actual players reported. This would improve the quality and creativity of our system's output because the system would be incorporating feedback from actual people.

Fixing one certain issue with our level generator would improve our system's efficiency and, ultimately, improve the average quality of the levels that are generated. Currently, when we place Sokoban blocks in our level, we simply assign the block to a random tile in a certain room. However, since blocks must be pushed from their start position to a water tile, blocks are often placed in positions which lead to impossible to solve puzzles. Although these puzzles are filtered out by the Validator, the generation of many unsolvable levels results in low system efficiency since are large percentage of puzzles including Sokoban blocks are rejected. Also, the fact that a majority of the levels using Sokoban blocks are rejected while a majority of levels without Sokoban blocks are accepted results in a low percentage of output levels utilizing the mechanic. The solution which we propose for future work is to use an algorithmic approach based on the reversal of the Sokoban Solver process instead of using random placement. If our system were to use the Sokoban Solver to simulate pulling a block from its goal position to a position in the level, the Sokoban puzzle would be guaranteed to be solvable since the block arrived in its position by the reverse of the process which is used to solve it. Ensuring all puzzles which utilize Sokoban blocks are solvable would increase the efficiency of the system by reducing the number of rejected puzzles and also improve the quality of the output by producing more levels which feature block pushing mechanics.

One final idea we have is a modification to make the system design levels in a more human-like, creative way. When humans embark on a creative endeavor, their process is usually characterized by an iterative approach of designing, evaluating, and refining. Our system, on the other hand, approaches the problem by designing, evaluating, and discarding; we just throw out a majority of the levels that are generated and only keep the best. In order to improve the creative nature of the level design process, we propose a modification to the system so that it can refine the levels that it generates instead of discarding the poorly made ones. We believe that this could be implemented if we modified our Evaluator to not only produce a single score for the overall quality of the level, but to also provide scores tied to each section of the level. That way, the system could then understand which parts of the level are resulting in a lower score. Armed with that knowledge, the system could refine the level by changing the locations of mission elements or adding entirely new mission elements. This iterative refinement process where one module of the system reports which parts of an output need improvement then another module modifies the output to improve the quality is reminiscent of the backpropagation process found in neural networks. It would both improve the average quality of produced levels as well as transform the system's design process into a more creative workflow.

Conclusion

Video games are quickly becoming one of the most widely enjoyed types of entertainment. With the ever increasing numbers of games coming out each year companies are continually looking for ways to differentiate their games. One popular method for achieving this is by producing ever larger, more intricate and detailed worlds. This ever growing pursuit of more content has not come without challenges and costs. One way to counteract these growing costs is by leveraging procedurally generated content. While this technique has been successfully leveraged in many games, it has not seen successful use in the puzzle game and Zelda-like game genres. These games require much more thought and craft when creating levels and so require a system more robust and capable than simple procedural randomness. In this paper we present a computational creative method for producing novel, interesting, and most importantly fun to play dungeon levels.

The system creating these levels relies on the works of many from the computational creativity community as well as our own novel contributions. Importantly, this system has several steps of generation that are combined to create a potential level which is tested and evaluated for quality. Finally, these levels are assembled by our system into a set of levels that respects a traditional difficulty curve to offer players an interesting level of challenge. Ultimately, we claim that, while our system is not perfect and has much room for improvement, it is in fact creative and produces artifacts that reflect its creativity.

References

Hagberg, A. A.; Schult, D. A.; and Swart, P. J. 2008. Exploring network structure, dynamics, and function using networkx. Proceedings of the 7th Python in Science Conference. [Online; accessed 2020-04-20.

Oliphant, T. 2006–. NumPy: A guide to NumPy. USA: Trelgol Publishing. [Online; accessed 2020-04-20.

Orland, K. 2015. How an early '90s windows gaming classic was unearthed after years in limbo.

Ritchie, G. 2007. Some empirical criteria for attributing creativity to a computer program. *Minds and Machines* 17:67–99.

Ventura, D. 2017. How to build a cc system. In *ICCC*.

Virtanen, P.; Gommers, R.; Oliphant, T. E.; Haberland, M.; Reddy, T.; Cournapeau, D.; Burovski, E.; Peterson, P.; Weckesser, W.; Bright, J.; van der Walt, S. J.; Brett, M.; Wilson, J.; Jarrod Millman, K.; Mayorov, N.; Nelson, A. R. J.; Jones, E.; Kern, R.; Larson, E.; Carey, C.; Polat, İ.; Feng, Y.; Moore, E. W.; Vand erPlas, J.; Laxalde, D.; Perktold, J.; Cimrman, R.; Henriksen, I.; Quintero, E. A.; Harris, C. R.; Archibald, A. M.; Ribeiro, A. H.; Pedregosa, F.; van Mulbregt, P.; and Contributors, S. . . 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17:261–272.