Two-step Constructive Approaches for Dungeon Generation

Michael Cerny Green
Game Innovation Lab
New York University
Brooklyn, NY
mcg520@nyu.edu

Ahmed Khalifa
Game Innovation Lab
New York University
Brooklyn, NY
ahmed@akhalifa.com

Athoug Alsoughayer
Game Innovation Lab
New York University
Brooklyn, NY

Divyesh Surana
Game Innovation Lab
New York University
Brooklyn, NY

Antonios Liapis
Institute of Digital Games
University of Malta
Msida, Malta
antonios.liapis@um.edu.mt

Julian Togelius
Game Innovation Lab
New York University
Brooklyn, NY
julian@togelius.com

ABSTRACT

This paper presents a two-step generative approach for creating dungeons in the rogue-like puzzle game MiniDungeons 2. Generation is split into two steps, initially producing the architectural layout of the level as its walls and floor tiles, and then furnishing it with game objects representing the player's start and goal position, challenges and rewards. Three layout creators and three furnishers are introduced in this paper, which can be combined in different ways in the two-step generative process for producing diverse dungeons levels. Layout creators generate the floors and walls of a level, while furnishers populate it with monsters, traps, and treasures. We test the generated levels on several expressivity measures, and in simulations with procedural persona agents.

CCS CONCEPTS

• **Computing methodologies** → Planning for deterministic actions; Game tree search; • **Applied computing** → **Computer games**.

KEYWORDS

procedural content generation, level generation, automated game playing, expressive range analysis

ACM Reference Format:
Michael Cerny Green, Ahmed Khalifa, Athoug Alsoughayer, Divyesh Surana, Antonios Liapis, and Julian Togelius. 2019. Two-step Constructive Approaches for Dungeon Generation. In *FDG '19: Procedural Conten Generation Workshop, August 26-30, 2019, San Luis Obispo, CA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/1122445.1122456

FDG '19, August 26-30, 2019, San Luis Obispo, CA

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

While research on level generation focuses on level generators based on stochastic search [14], constraint solving [11, 12], or machine learning [13], level generation in published games is mostly carried out via constructive algorithms. Unlike generate-and-test processes, constructive generators do not evaluate and re-generate output; for example, cellular automata and grammars can be used for constructive generation, as well as more freeform approaches such as diggers [10]. Such generators are computationally lightweight since they do not evaluate their generated output. This allows games to quickly create endless variations to game-play by generating maps as in *Minecraft* (Mojang 2011), weapons as in *Borderlands* (Gearbox 2009) or NPCs as in *Skyrim* (Bethesda 2011) in real-time. However, choosing the right algorithm for the design constraints seems to be an art rather than a science. A better understanding of the properties of different families of generators as well as how they can be combined could help advance this situation.

One way to better understand the properties of constructive generation techniques is to systematically investigate their differing performance when applied to different aspects of level generation. For example, we can divide up the task of generating a level into different phases and use different constructive algorithms for each phase. Multi-agent generation processes have been explored for terrain generation [4]; however, in this case, generation was controlled by multiple agents given explicit areas/types of terrain to generate, and agents could manipulate each others' finished products. In this paper, the second step of the process builds off of—but does not attempt to change—the result of the first.

Many mobile games use procedural content generation (PCG) to quickly generate content—with varying degrees of success. Flappy Bird (dotGears 2013), Doodle Jump (Lima Sky 2009), Downwell (Moppin 2015), Polytopia (Midjiwan AB 2016), and Temple Run (Imangi Studios 2011) are just a few of many examples. This paper compares several constructive generation approaches that produce levels for the rogue-like puzzle game MiniDungeons 2 [8]. The fast generation afforded by constructive approaches allows the game to create new levels with minimal lag even on a mobile device, for which MiniDungeons 2 is intended. The novelty of this approach is the use and analysis of a two-step process for generating the level's architecture first, using a standalone *layout creator*, and distributing the game objects on that architecture based on game-specific

rules using a *furnisher*. This allows for different combinations of creators and furnishers and can also work with manually created architectures (furnished automatically) or vice versa. While creators only place walls or floor tiles and use tried-and-tested algorithms popular in rogue-like dungeon generation [10], furnishers must account for the interactions and dynamics of different game objects. Moreover, since constructive generators do not test the final result, the rules used in the different furnishers must ensure that the level can be completed but also viable for different playstyles. To test the latter, artificial agents acting as play personas from prior work [6] are used to test the generated levels. The variety of personas allows us to assess the generated content from multiple perspectives.

The contributions of this paper include the division of dungeon generation into two phases and several algorithms that work in each phase; a novel agent-based furnisher; and play-testing the resulting levels with procedural personas.

2 BACKGROUND

We start our exploration in procedural level generation of Minidungeons 2 by reviewing how constructive map generation has been done in other games. Based on Shaker et al. [10], popular methods for generating dungeons include binary state partitioning, cellular automata and digger agents. A *digger agent* is placed in a dungeon filled with impassable blocks (often in a random position), and removes the block it is in while moving to adjacent tiles following random or rule-based strategies. *Cellular automata* are popular methods used to generate organic and smooth-looking maps, including islands and caves [9]. Cellular automata work in iterative steps; in each step they change a tile based on patterns in its adjacent tiles. Rules regarding adjacent tiles and how they affect the current tiles can be elaborate or include stochasticity; the most straightforward way to create caves, however, is to change the current tile to match the majority of its adjacent tiles [10].

Grammars have proven to be successful in generating adventure game levels due to their formal structure which can be intuitively interpreted and edited by human designers. Dormans [5] uses graphs to generate missions as an initial description of a level, then transforming that into the rendered space. The whole process can result in a highly automated yet controlled way for map generation. This research was extended with a highly controllable automated system of model transformations [5].

While constructive methods take a straightforward approach to generation and have a long history in the game industry, to the best of our knowledge no one has extensively analyzed the combination of different methods that iteratively generate map layouts and distribute game objects. This paper presents nine unique creator-furnisher combinations and analyzes patterns in resulting levels.

3 THE MINIDUNGEONS 2 GAME

Minidungeons 2 (MD2) is a 2D rogue-like dungeon crawler [8] in which the player controls a hero and tries to find the exit of a dungeon, while encountering a variety of monsters and objects along the way. The level is set on a 10×20 tile grid, where each tile is either impassible (wall) or passable (floor). Floor tiles can contain interactable objects (*treasures, potions, traps, portals*) or game characters, subdivided into enemies (*goblins, goblin mages, blobs, ogres, minitaurs*) and the *hero* controlled by the player. LOS stands for Line of Sight, i.e. if this entity has a clear sight-line to the designated target.

Goblin	Move towards hero if in LOS
Goblin Mage	Hurl bolts at hero in LOS if within 3 tiles
Blob	Move toward closest hero or potion if in LOS; power up if colliding with potion or another blob (removing collided object)
Ogre	Move towards closest hero or treasure if in LOS; empty treasure if colliding with it
Minitaur	Move towards hero anywhere in the dungeon, following A* pathfinding; can not be killed, only stunned for 3 rounds

Table 1: Movement strategies of monsters.

All game characters have a preset number of hit-points (HP) and deal damage when they collide with the hero or each other; goblin mages also hurl bolts that deal damage at range. When a game character runs out of HP, they die. To win, the player must move the hero to the exit without dying; the hero moves to adjacent tiles, except if stepping into a portal in which case it instantly transports to the other portal (a level has two portals, or none). Unlike its predecessor [7], in MD2 monsters' damage and movement behaviors are deterministic (see Table 1). This brings MD2 closer to a puzzle game where the player must plan ahead how to reach the exit without dying as well as collecting as much treasure or killing as many monsters as possible. The personas used in the analysis of the creator-furnisher combinations are done using agents from another project in Minidungeons 2 [6].

4 CONSTRUCTIVE GENERATORS FOR MD2

The generative process for MD2 levels is split into two steps: the first step generates the architectural layout of the dungeon; the second step furnishes it with game objects and monsters. Three generators are built for the first step, identified as *layout creators*, and determine which tiles in the dungeon will be passable (floor) or impassable (wall). Once the layout has been created (in whichever fashion), it is furnished with game objects (monsters, treasure, potions, traps, portals, the entrance and the exit). Three furnishers are introduced and evaluated in this paper, identified as *Game Element furnishers*.

4.1 Layout Creators

Based on Shaker et al. [10], three layout creators were designed following popular methods for constructive approaches: constraint-based, cellular automata, and agent-based creators. All creators ignore the border of the 10×20 MD2 grid which is always filled with walls, and operate on a grid of 8×18 tiles.

4.1.1 Constraint-based: The constraint-based creator (CC) is inspired by the TinyKeep dungeon generator and collision detection systems [1]. The generator spawns a random number of rooms with a random width and height. At every step, the locations of the rooms are modified by separating colliding rooms from each other, either until no more collisions occur and all the rooms are within the bounds of the map or until 100 iterations have passed. This

Entrance	Always within 8 tiles of one end of the **LP**
Exit	Within 5 tiles of the opposite end of the **LP** from the entrance
Treasure Chests	Surrounded by 2 or more walls, with a preference towards 3 walls
Potions	Scattered randomly across the map
Portals	One placed 5-10 tiles from Entrance, the other 5-10 tiles from Exit; at least 10 tiles from each other
Traps	On or around (within 1 tile) the shortest path between the Entrance and Exit
Goblins	One side be a wall
Goblin Mages	Must be adjacent to a Goblin
Ogres	4-8 tiles away from a Treasure chest in LOS
Blobs	4-8 tiles away from a Potion in LOS
Minitaur	4-8 tiles away from Entrance

Table 2: Constraint-based furnisher object placement rules

Entrance/Exit	Any tile where the other object is not present in a 5 tile neighborhood
Treasure Chests	At least 3 neighbors must be walls using 1 tile neighborhood
Potions	At most 3 neighbors are populated with objects using 1 tile neighborhood
Portals	One portal must neighbor the Entrance, the other portal must neighbor the Exit using 3 tile neighborhood
Traps	5 or more neighbors must be populated with walls or other objects using 1 tile neighborhood
Goblins	4 or more neighbors are walls using 1 tile neighborhood, and none of the tiles in a 3 tile neighborhood are goblins
Goblin Mages	1 or more neighbors must be a Goblin using 3 tiles neighborhood
Ogres	None of the tiles in 1 tile neighborhood can be walls
Blobs	At least 1 of the tiles in a 3 tile neighborhood is a Potion
Minitaur	1 of the tiles in a 3 tile neighborhood is the Entrance

Table 3: Cellular Automata furnisher object placement rules

generator initializes anywhere from 8 to 16 rooms with a width between 4 to 6 tiles and a height between 4 to 8 tiles.

4.1.2 Cellular Automata: Based on the binary cellular automata cave generator in Shaker et al. [10], the MD2 cellular automata creator (CAC) initially populates all 8×18 tiles with either wall (45% chance) or floor tiles. Cellular automata changes each tile's type to the type of the majority of its neighbors (considering the 8 closest neighboring tiles). In MD2, this resulted in a single 'island' of floor tiles in the center. To counter this, a rule was added that checks if the map consists of more than 75% floor tiles, in which case the map is filled with more wall tiles and another step of cellular automata is applied. This process will continue until there are less than 75% floor tiles. This method can create multiple isolated 'islands' of free space, which the player would be unable to reach. The largest empty space takes precedence. Any smaller islands (defined as isolated, empty tiles smaller than the largest space) are filled with walls.

4.1.3 Agent-based: The agent-based creator (AC) was formulated much like the digger agent described by Shaker et al. [10]. The 8×18 tile grid initially is filled with walls. The agent is then placed on a random tile, converting it to a floor tile. The agent then randomly selects a direction to travel in, moves forward one tile and converts it into a floor tile. The process continues in steps; in every step the agent has a chance to change direction, increasing the probability (+5%) each step that it does not. This process continues until the map contains a number of floor tiles, which is randomly selected in the beginning to be within the range of 75 to 95 tiles.

4.2 Game Element Furnishers

Once the architecture is created by one of the above generators (or by a human creator), all game elements are added through another process identified as the *furnisher*. Three furnishers are used in this paper; their differences are primarily in the rules for placing objects, and whether objects can change their location after being placed. Game elements added by the furnishers are the entrance (where the hero starts from), exit, treasures, potions, portals, traps and monsters.

4.2.1 Constraint-based: In the constraint-based furnisher (CF), each game element is constrained to areas of the map with specific characteristics. Before placing any object, the furnisher finds the *longest path* (**LP**) that exists between any two points on the map. Elements are added iteratively, with specific elements such as the entrance and exit added first. As each element is added, the map is scanned for suitable locations that satisfy that element's constraints. The furnisher then randomly selects a suitable location among these to place the element. In-depth rules are found in Table 2.

4.2.2 Cellular Automata: Similar to the CA creator, the cellular automata furnisher (CAF) populates a map with game objects based on each tile's state and its neighboring tiles' states. The difference between the furnisher and the creator is that the furnisher visits tiles in random order instead of sequential, has a variable size neighborhood which allows CAF to access tiles that are more than 1 tile away, and has a restriction on the number of each placed object (one entrance, two traps, etc) so it will not overpopulate the dungeon. All floor tiles in the map are checked iteratively; if the tile is empty and fulfills the neighborhood requirements of Table 3, the corresponding game object is added to map. Since the neighborhood requirements are mutually exclusive, a tile can have only one object type.

4.2.3 Agent-based: The Agent-based furnisher (AF) differs from previous ones as each object takes turns moving around the map by itself. Every game element is given its own heuristic for movement priorities (see Table 4) and is randomly placed somewhere on the map. Every "turn", all objects move around the map according to what they see and their proximity to other objects. For example, treasures actively seek out Goblins to guard them, and Goblins attempt to hide from other Goblins. All objects operate on a simple one-step-lookahead. After 45 turns of movement (based on preliminary testing), the map is considered furnished.

Entrance/Exit	Move as far apart from each other as possible
Treasure Chests	Moves closer to goblins in Line-Of-Sight (LOS)
Potions	Move randomly around the map
Portals	Move as far apart from each other, the Entrance, and the Exit
Traps	Move away from other traps and goblins in LOS. Move towards treasure in LOS
Goblins	Move away from other goblins in LOS
Goblin Mages	Move toward goblins in LOS and away from other Goblin-Mages within LOS
Ogres	Move away from Ogres in LOS (within 6 tiles), and move within 4 tiles of Treasure in LOS
Blobs	Move within 4 tiles of other Blobs and Potions in LOS
Minitaur	Move as far away as possible from the Entrance and Exit

Table 4: Agent-Based furnisher object placement rules

5 EVALUATION

In order to evaluate the different patterns favored by each of the three creators and the three furnishers, each creator produces 1000 layouts for each furnisher, which then furnishes each layout once. The result is 9000 MD2 levels for all creator-furnisher combinations. Statistical tests in this section use $p < 0.05$ via Student's two-tailed t-test assuming unequal variance.

5.1 Differences between generators

Figure 1 shows three layouts created by the three creators, which are then furnished by each of the three furnishers. There are several consistent patterns among generator combinations: levels always have around 20 game elements (monsters comprise slightly less than half of those), potions are a bit more than half the number of monsters and a bit less than double the number of treasure chests. Differences in the layouts are also obvious, while the way in which each layout affects each furnisher less so. This section attempts to quantitatively analyze these patterns in creators, furnishers, and their combination.

5.1.1 Layout Creators: Observing the layouts created by each creator, we decided to use the number of floor tiles, the length of the longest path between the player's starting position and the exit, and the number of wall chunks as metrics to analyze the generated layouts. The number of floor tiles just counts the number of empty tiles in the generated map, the longest path length calculates the number of moves that a player needs to traverse the map, and the number of wall chunks calculates the number of isolated wall segments in the generated layouts.

Figure 2 shows the expressive range of our three layout creators. It is obvious that the constrained-based creator generates levels with the longest path compared to the other two techniques. Also, it generates maps with the lowest number of isolated wall chunks. We think that the constraints satisfaction guarantees that rooms are not cutting each other, allowing for a longer path and fewer isolated wall chunks. On the other hand, CAC creates maps with the highest number of empty tiles which is due to the rules which govern the growth of cellular automata. Both AC and CAC generate

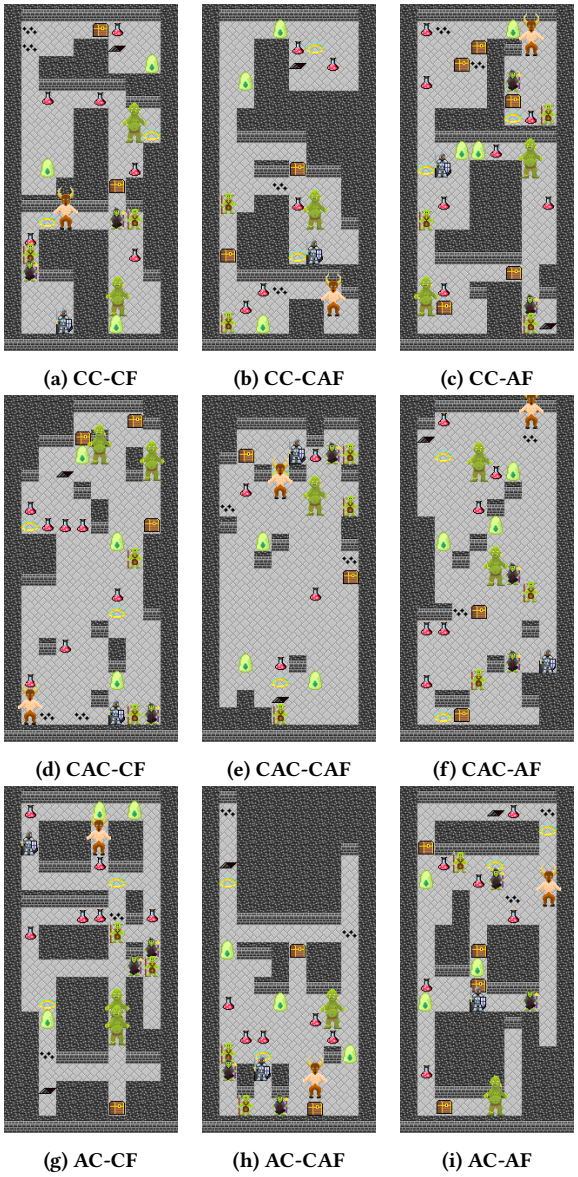
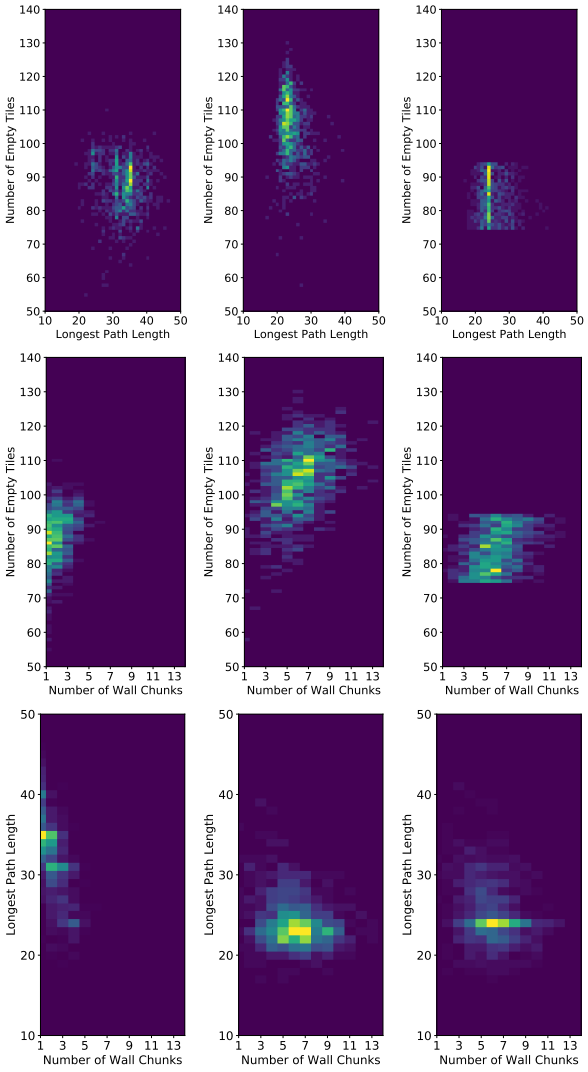


Figure 1: Generated MD2 levels for different furnishers (one per column) on the same layout per creator (one per row).

maps with high amounts of isolated chunks, most likely due to the ability of the agent to cross its own path and the extra wall-seeding function in CA.

5.1.2 Furnishers: The different combinations of creators and furnishers result in 9000 MD2 levels, out of which 3000 levels are generated by each creator (and different furnishers) and 3000 levels by each furnisher (and different creators). In order to assess how each algorithm affects the placement of game elements, these levels are grouped by creator or furnisher and their average metrics are compared with those of other creators or furnishers respectively. A broad range of metrics has been explored, including the number and ratio of all game elements, the path from entrance to exit and

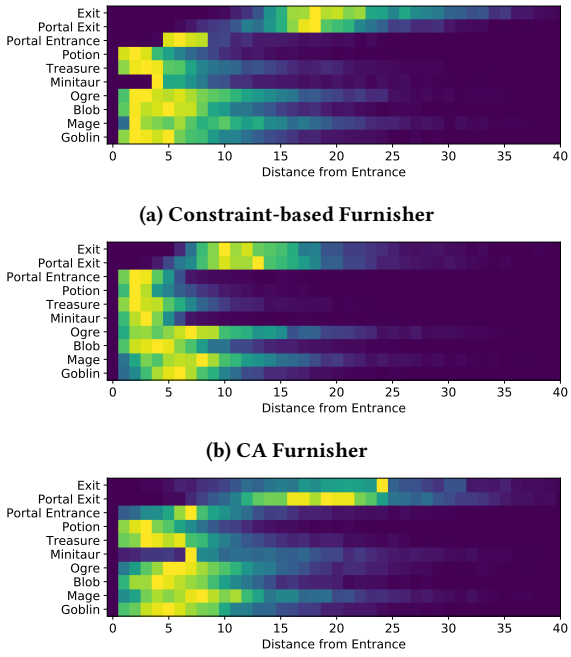


(a) CC (b) CAC (c) AC

Figure 2: The expressive range of the three different creators using the number of empty tiles, the longest path length, and the number of wall chunks.

the number of potions or treasures which are guarded (i.e. their paths from the entrance are blocked by a monster). For the sake of brevity, only a subset of significant differences are reported here.

Grouping MD2 levels by furnisher, Figure 3 shows the expressive range of these furnishers. We used the distance between the entrance and the various game objects as metrics for the analysis. The obvious difference that can be observed is that CAF has the smallest distance from entrance to exit or portal exit, compared to the two other furnishers. This is expected as CAF is the only furnisher which places both entrance and exit randomly with the only control being on neighboring tiles, while other generators either greedily maximize the distance between entrance and exit (AF) or place them along the longest path (CF). Similarly, the agent-based



(a) Constraint-based Furnisher

(b) CA Furnisher

(c) Agent-based Furnisher

Figure 3: Expressive range of the shortest distance from the entrance to all other game objects for each furnisher agent.

furnisher has the largest entrance-minitaur distance compared to the other two techniques. In maps furnished by AF, the minitaur is most always able to maximize the distance between the entrance and itself.

5.2 Playability metrics

While structural differences shed light on the physical appearance of the levels, how these differences affect gameplay is an important next question. In order to gain some insight on how players would interact with generated levels, a number of artificial agents designed to represent different playstyles were used to playtest a sample of the 1000 levels generated for each creator-furnisher combination. These artificial agents, named procedural personas, use a variant of Monte-Carlo tree search [2] with an evolved exploration strategy described by [6]. Three personas are used in this paper: the runner (which prioritizes reaching the exit with fewest actions), the monster killer (which prioritizes killing the most monsters and reaching the exit) and the treasure collector (which prioritizes collecting the most treasure and reaching the exit). To limit simulation times, results are calculated on simulations of procedural personas in the 100 first levels generated by each creator-furnisher combination.

Figure 4 summarizes three important metrics which are targeted explicitly by the personas (one each). An initial observation is that completion rates are high for runner and killer personas. Unsurprisingly, personas perform well on the metric they prioritize: runners reach the exit, monster killers (MK) kill more monsters, and treasure collectors (TC) collect more treasures. However, the differences between them are more prominent for some generators than for

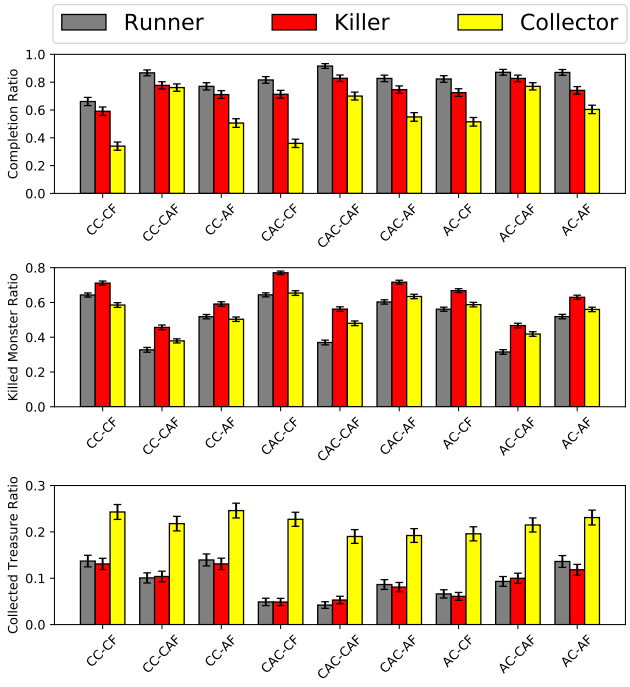


Figure 4: Metrics of each persona on playthroughs of 100 levels; error bars show the 95% confidence interval.

others: maps generated by CC-CF combination are the hardest to beat, and the biggest differences in treasures collected between the TC persona and the other two personas are found in CAC-CF combination. It is also notable that certain creator-furnisher combinations result in generally more monsters killed (e.g. all combinations with CF), while levels by the cellular automata furnisher generally result in fewer kills. Similarly, CA creators have the least collected treasures for Runner and MK personas; on the other hand, the starker differences with the TC persona in this metric are with the constraint-based furnisher.

6 DISCUSSION

Driven by the objective needs of the MiniDungeons 2 game, which must produce new content while also operating on a mobile device, we designed and experimented with a multitude of computationally lightweight generators. Since the generators populate the level in steps, this allows for many combinations of patterns in the levels. As shown in the experimental analysis, the different generators introduce different patterns, and while visible differences in levels of Figure 1 are mostly due to the layout creators, the number and placement of game objects seem less sensitive to differences among creators. Based on playthroughs of artificial agents, the generated levels allow for different strategies— although some strategies do not guarantee that the level can be finished. The CC-CF combination created the hardest maps for any persona to beat, suggesting that they create more of a challenge than other combinations. Some generator types in particular penalize or reward specific playstyles (e.g. killing monsters in CAF levels) while some creators make a clearer distinction *between* playstyles (treasure collection differences in CAC levels).

There are many directions for future work, but also on further analysis on how the patterns of each step of the generative pipeline affects patterns in the next. A more in-depth analysis using non-linear or possibly even computer vision techniques such as deep learning could shed more light on the dependencies of creator-furnisher pairings. Another idea would be to take a note from previous generative comparison work [3] to perform a more general comparison of metrics. Moreover, the artificial agents can be used as surrogate testers in order to fine-tune the placement rules of some objects, so that for instance the chance that runners can complete the generated levels increases while also increasing the number of treasures collected by treasure collectors (that currently rarely gain a third of all treasure in the level). We also believe that the Agent-based furnisher offers a fresh approach to the old problem of procedural map generation in games. In this paper, all agents move using simple one-step-look-ahead, but future work could install more rigorous tree search or localized optimization methods, allowing these agents to interact with one another in complex ways.

7 CONCLUSION

This paper presented nine generative algorithms that decide either on the architecture or on the game object distribution of levels for MiniDungeons2. By combining these creators and furnishers, a broad set of patterns emerges. Experiments have demonstrated how levels created by different combinations of creators and furnishers differ from each other, and how they affect the playthroughs of artificial agents acting as surrogates of human players.

ACKNOWLEDGEMENTS

Michael Cerny Green acknowledges the financial support of the SOE Fellowship from NYU Tandon School of Engineering. Ahmed Khalifa acknowledges the financial support from NSF grant (Award number 1717324 - "RI: Small: General Intelligence through Algorithm Invention and Selection.").

REFERENCES

[1] Adonaac. 2015. Procedural Dungeon Generation Algorithm. http://gamasutra.com/blogs/AAdonaac/20150903/252889/Procedural_Dungeon_Generation_Algorithm.php. Last Accessed: April 18, 2019.
[2] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4, 1 (March 2012), 1–43.
[3] Michael Cook, Jeremy Gow, and Simon Colton. 2016. Danesh: Helping bridge the gap between procedural generators and their output. (2016).
[4] Jonathon Doran and Ian Parberry. 2010. Controlled procedural terrain generation using software agents. *IEEE Transactions on Computational Intelligence and AI in Games* 2, 2 (2010), 111–119.
[5] Joris Dormans. 2011. Level design as model transformation: a strategy for automated content generation. In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*. ACM.
[6] Christoffer Holmgard, Michael Cerny Green, Antonios Liapis, and Julian Togelius. 2018. Automated playtesting with procedural personas with evolved heuristics. *IEEE Transactions on Games* (2018).
[7] Christoffer Holmgård, Antonios Liapis, Julian Togelius, and Georgios N. Yannakakis. 2014. Evolving Personas for Player Decision Modeling. In *Proceedings of the IEEE Conference on Computational Intelligence and Games*.
[8] Christoffer Holmgard, Antonios Liapis, Julian Togelius, and Georgios N. Yannakakis. 2016. MiniDungeons 2: An Experimental Game for Capturing and Modeling Player Decisions. In *Proceedings of the 10th Conference on the Foundations of Digital Games*.

[9] Lawrence Johnson, Georgios N Yannakakis, and Julian Togelius. 2010. Cellular automata for real-time generation of infinite cave levels. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. ACM, 10.

[10] Noor Shaker, Antonios Liapis, Julian Togelius, Ricardo Lopes, and Rafael Bidarra. 2016. Constructive generation methods for dungeons and levels. In *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, Noor Shaker, Julian Togelius, and Mark J. Nelson (Eds.). Springer, 31–55.

[11] Adam M Smith and Michael Mateas. 2011. Answer set programming for procedural content generation: A design space approach. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 3 (2011), 187–200.

[12] Gillian Smith, Jim Whitehead, and Michael Mateas. 2011. Tanagra: Reactive planning and constraint solving for mixed-initiative level design. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 3 (2011), 201–215.

[13] Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy K Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. 2018. Procedural Content Generation via Machine Learning (PCGML). *IEEE Transactions on Games* 10, 3 (2018), 257–270.

[14] Julian Togelius, Georgios N Yannakakis, Kenneth O Stanley, and Cameron Browne. 2011. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 3 (2011), 172–186.