AI Design Document

Procedural generation of levels for a stealth video game

Riccardo Graziosi (matr. 961382)
riccardo.graziosi@studenti.unimi.it

Artificial Intelligence for Video Games
A. Y. 2020/2021

Table of contents

1. Introduction

The goal of the project is to create a procedural content generation (PCG) system that produces playable and interesting levels for a simple 2D stealth video game. In particular, the system must generate the whole level content, namely the game field as well as the enemies.

The following chapter serves as a brief design document for the game whose levels will be generated.

Chapter 3 contains a thorough explanation of the design behind the PCG system, starting from a high-level overview of how the game is represented internally by the system to how the generation and validation algorithms work. Moreover, the system is also classified using common PCG taxonomy.

Chapter 4 is about the implementation details of the strategies and techniques explained in Chapter 3.

Lastly, Chapter 5 contains some closing thoughts about the quality of the content generated by the system. Moreover, achieved goals are stated, alongside areas of the project that would benefit from further work.

To conclude, an appendix contains useful information to navigate the source code of the project.

2. Game design overview

The generated levels are for a 2D top-down stealth game. This chapter describes a particular game specifically designed to work well with the PCG system developed, but other slightly different games may work as well with little or no modification.

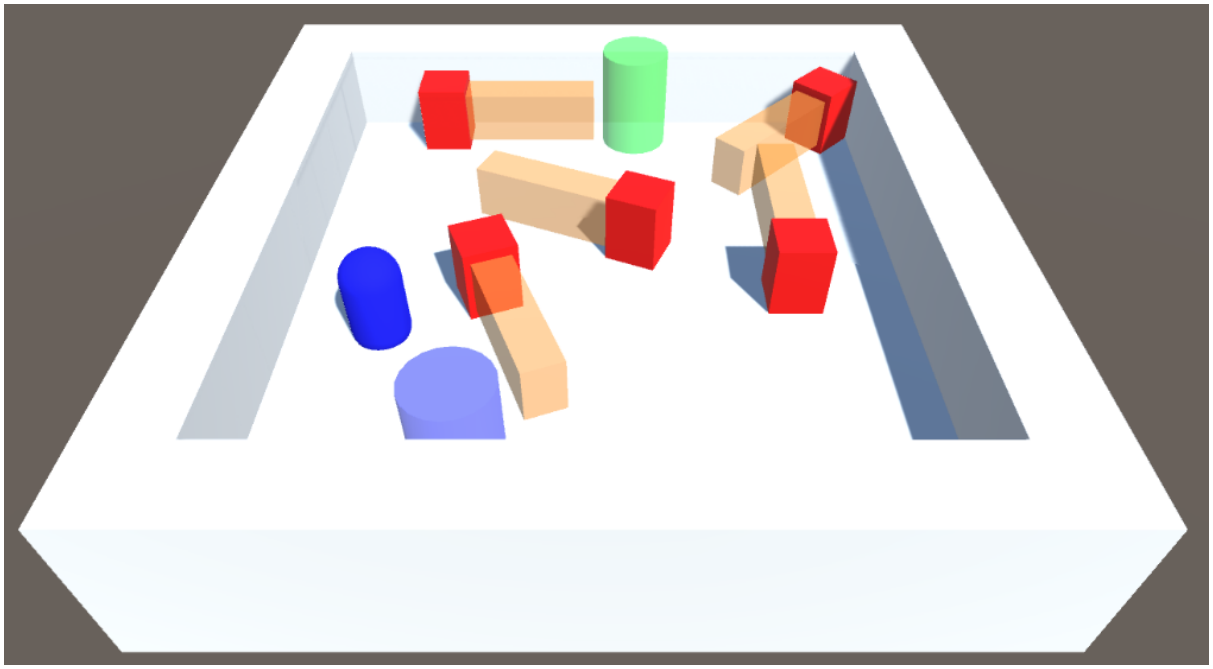
A level simply consists of a rectangular shaped room. A point at one side of the room serves as the spawn point for the player, while another point, at the opposite side of the room, serves as the goal the player must reach to complete the level.

Different kinds of enemies populate the room and roam around following a specific repetitive pattern. Enemies are made of their square shaped body plus their cone of vision. Each enemy is characterized by its position, rotation and vision length. As a result, there can be quite a few types of enemies: some that stay still in a certain position and rotation, some that just look around maintaining their position, some that patrol back and forth along a direction, etc.

The player spawns at the spawn point and then is free to move along all four directions. No other actions are allowed. Moreover, movement is not subject to forces (e.g. friction).

The objective of the player is to reach the goal point while avoiding enemies. If an enemy touches or sees the player, then the latter is brought back to the spawn point.

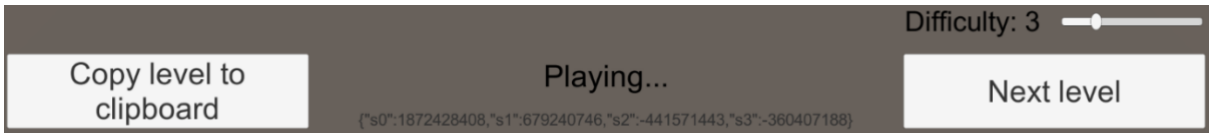
The following picture depicts a typical level of the game. The spawn point is the light-blue cylinder, the goal is the green cylinder, enemies have red body and orange field of vision, and the player is the blue capsule.



How to play

Key	Action
WASD (or arrow keys)	Move the player.
Spacebar (keep pressed)	Change camera perspective.

User interface



The user interface is functional to test the PCG system. In particular, there is:

- A “Next level” button, to generate a new level. The current level is lost. The new level is generated on the fly and the player can play it immediately. The difficulty of the newly generated level is based upon the value of the slider “Difficulty”.
- A “Difficulty” slider, to set the difficulty level of the next generated levels.
- A “Copy level to clipboard” button, to copy the level seed (or, using Unity’s naming, Random.state) to the device clipboard. This seed can be saved somewhere and used later to play the same level again. The seed must be entered from the Inspector of the Unity Editor, in the “Seed” field of the “Generator” game object. Note that to reproduce a level, both the same seed and difficulty level must be chosen.

3. AI design

Classification of the PCG system

Following a taxonomy of PCG proposed by Togelius and later revised by Shaker (see Procedural Content Generation in Games, N. Shaker, 2016, chapter 1.6), the PCG system can be classified as:

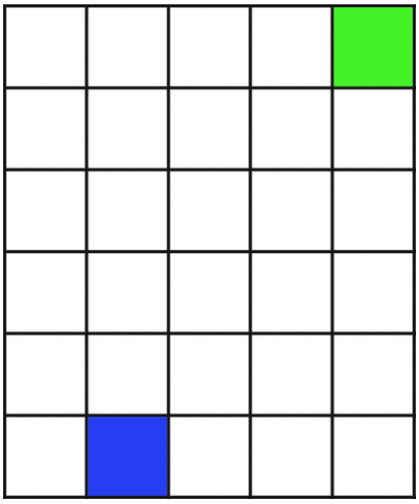
- Online, as the PCG algorithm is supposed to work during game execution. This constraint poses a bound on computation time. On the other hand, the system could also be seen as offline, since the developer of a game can use it to generate levels and use them as a starting point to be later modifier.
- Necessary, since the output of the algorithm is a necessary content: the whole level.
- Parametrized by a difficulty level and a seed.
- Generic and not adaptive, because the system does not take the player's past behaviour into account when generating new levels.
- Deterministic, since given the same parameters and seed the system always outputs the same result.
- Generate-and-test, as a lot of steps in the PCG algorithm generates something and then tests if every constraint is still satisfied.
- Automatic, since the system creates the level from scratch to finish, without permitting a mixed authorship paradigm.

Internal representation of the level

Even though the game is supposed to be a real time one and the player can move continuously pixel by pixel, the internal representation that the PCG system uses is discrete both in time and space.

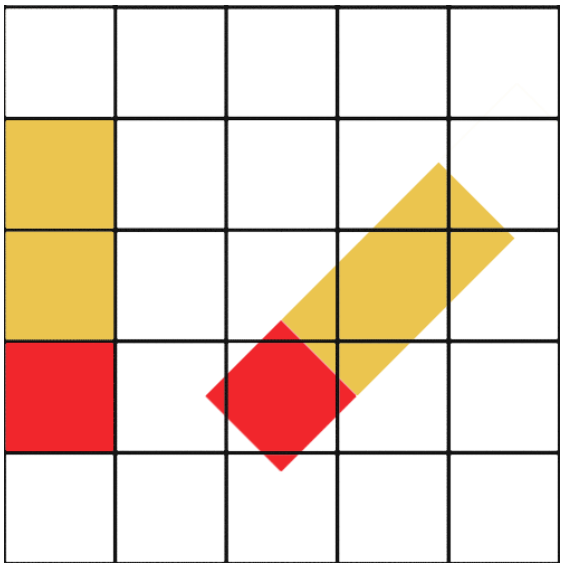
As a result, the game field can be represented as a simple M x N rectangular grid (where M is the number of rows and N the number of columns), where M >= N. The spawn point and goal point are represented as cells of this grid: in particular, the former is a randomly selected cell of lowest row while the latter Is a randomly selected cell of the highest row. Note that since the number of rows is greater or equal to the number of columns (M >= N) then either (1) the game field is a square (M = N) or (2) the game field is rectangular and such that the player must travel along its longest dimension to reach the goal.

An example is shown in the following picture, where M=6, N=5, the spawn point is depicted in blue and the goal point in green:



Like everything else, enemies are represented in a discrete manner. In particular, they are composed of their body (which occupies just a single tile of the grid) plus their cone of vision of length L (which is just a Lx1 rectangle that is rotated in the same direction the enemy is). Enemies are characterized by their position on the grid (x, y) and their rotation α which can be one of the 8 main directions {0°, 45°, 90°, 135°, 180°, 225°, 270°, 315°}. 0° means facing downward (towards the spawn point), and the other values are referred to a clockwise rotation from 0°.

Some care must be taken for enemies that are facing towards any of the four diagonal directions. In fact, as it can be seen from the following picture, the spatial discretization works perfectly fine when the enemy is rotated 0°/90°/180°/270° whereas some approximation errors must be handled when the enemy is rotated 45°/135°/225°/270°. The body of the enemy (depicted in red) and the vision ray (depicted in orange) do not fit the tiles of the grid perfectly: in both cases, tiles occupied by small portions of the enemy are disregarded as a whole and tiles occupied by big portions of the enemy are considered fully occupied. Notice that, in the picture, both enemies have a vision length of 2, but while the enemy facing forward occupies two full tiles, the diagonally oriented one fully occupies just one tile: in fact, it must be taken into consideration that, if the tile edge has length 1, then the tile diagonal has length $\sqrt{2}$.



In the game, each enemy follows a pattern of movements. In the internal representation, a pattern is just a collection of multiple enemy states (position + rotation). All enemy states last for the same amount of time: 1 time step.

For example, the following table contains the representation of an enemy that, staying still, watches at direction 90° and then 0°, repeatedly:

	State 1	**State 2**
Position	(3, 4)	(3, 4)
Rotation	90°	0°

Note that discretization of time takes place here: while during the game the enemy rotates smoothly from 90° to 0° during 1 time step (which may be, for example, 1 second of real time), the internal representation swaps abruptly from 90° to 0° at the beginning of the next time step.

For the sake of simplicity, the number of states an enemy can have must be a power of 2. This constraint avoids the explosion of the number of configurations when there is more than one enemy. For example, suppose that there are two enemies, one with 4 states and one with 3 states. The total number of configurations of the two enemies together would be 12. Under the constrain stated above, the total number of configurations is always equal to the number of states of the enemy with the highest number of states.

PCG algorithm

The following pseudo code shows a high-level overview of how the generation algorithm works:

```
Start:
// Map generation
Randomly draw map dimensions (M and N)
Randomly position spawn and goal points

// Enemy generation
Generate enemies until level is non trivial
If (level is solvable) then finish; else go back to Start
```

The PCG algorithm enforces primarily two constraints on the generated level: solvability and non-triviality.

A level is said to be solvable if the player can reach the goal. The algorithm that checks whether a level is solvable simulates gameplay using the internal representation explained in the previous section. More information about the implementation of this algorithm is provided in the next chapter.

The solvability constraint is vital to the PCG system since a level that cannot be solved should never be proposed to the player. However, the solvability of a level does not say anything about whether that level is interesting or fun to play. The other constraint, non-triviality, is a bare minimum check that the level is at least a little interesting and not extremely easy or trivial.

A level is said to be trivial if there exist a path from the spawn point to the goal point that is always free from enemy surveillance. If a trivial path exists, then the player can just follow it without considering any of the enemies and complete the level too easily. The triviality check algorithm looks for trivial paths and places enemies along them, in order to make the level nontrivial. As for the solvability algorithm, more in-depth information can be found in the next chapter.

PCG parameters

The PCG system receives as input a difficulty parameter (integer number between 1 and 10) and a seed. The difficulty parameter is just directly proportional to the size of the game field and the number of enemies.

Given the same difficulty and the same seed, the PCG algorithm always produces the same level.

PCG output

The PCG system outputs some data containing information about the game field (its dimensions, position of spawn and goal points) and the enemies (position, rotation and vision length for every state of every enemy).

In order to test the level, this data must be given as input to a script that actually instantiates the needed entities in the game and animates them. Even though it is technically not part of the PCG system, the architecture used in this project for testing the levels out is depicted in the following picture.

4. Implementation details

Data structures for level representation

The game field, being a regular rectangular grid, can be stored using just two integers: one for the number of rows and one for the number of columns.

Positions can be stored using a data structure that represents 2D points/vectors. In particular, Unity's Vector2Int built-in data structure has been adopted. For example, spawn and goal points use this representation.

There are also cases where an entity occupies more than one position: for example, it may be useful to refer to all the positions an enemy is occupying/watching (i.e. both with its body and its field of vision). In order to store multiple positions, a set (in particular, a C# HashSet) of points has been used. Set operations also make it easy to perform unions and intersections while keeping the univocity of elements in the set, which are all desirable properties for a lot of the algorithms implemented.

Enemy generation algorithm

Enemy generation starts from a template that defines the general behaviour of the enemy. For example, there are templates for fixed enemies, enemies that only rotate around themselves and patrolling enemies. When an enemy has to be placed, a random template is chosen and an enemy from that template is created.

To decide the position of the enemy, the generation algorithm computes a set containing all the available tiles on which an enemy could be placed. The constraints to satisfy are the following: an enemy must not be placed such that it occupies either the spawn or the goal point, an enemy must not be placed very near or on top of another enemy, if a trivial path is present an enemy must be placed such that it blocks that path. Certain templates of enemies impose other constraints on position, such as that the enemy should occupy a centre position with respect to the game field.

The rotation of an enemy is usually decided based on its position relative to the game field: the goal is to rotate enemies toward the most natural orientation possible. For example, if an enemy is placed against a wall, it should not look at the wall, but rather in the opposite direction. If an enemy has multiple states with N different rotations, then the N most natural rotations are used.

It may happen that, given the constraints above, an enemy of a certain template cannot be placed. In this case, the enemy generation is aborted, and the algorithm tries to place an enemy starting from another template.

The number of enemies generated for a level depends on multiple factors. Firstly, a simple formula based on the difficulty and the size of the game field determines the base number of enemies to create. After these are generated, if the level happens to be trivial, all enemies necessary to make the level nontrivial are added.

Solvability algorithm

The goal of this algorithm is to return whether the level is solvable, given all the information about it (game field size, spawn and goal points and enemies). Part of the algorithm works in a similar way to the flood-fill algorithm used by image editing software.

Firstly, an array listing all possible configurations of the enemies is computed. For enemy configuration i, it is intended how every enemy is positioned and rotated at time step i, hence which tiles of the map are controlled by the enemies and which are not at time step i. As a result, we have an array that, given the index of a time step, returns a set containing the tiles occupied by all the enemies in that time step.

Now that the algorithm knows movement patterns of all enemies, it can simulate player movement. Starting from the spawn point, a flood-fill like algorithm is executed in order to simulate all possible player movements, i.e. the simulation moves the player along all the four main directions for as long as the duration of a time step consent. If a tile is found to be controlled by an enemy, then the flood-fill does not continue in that direction.

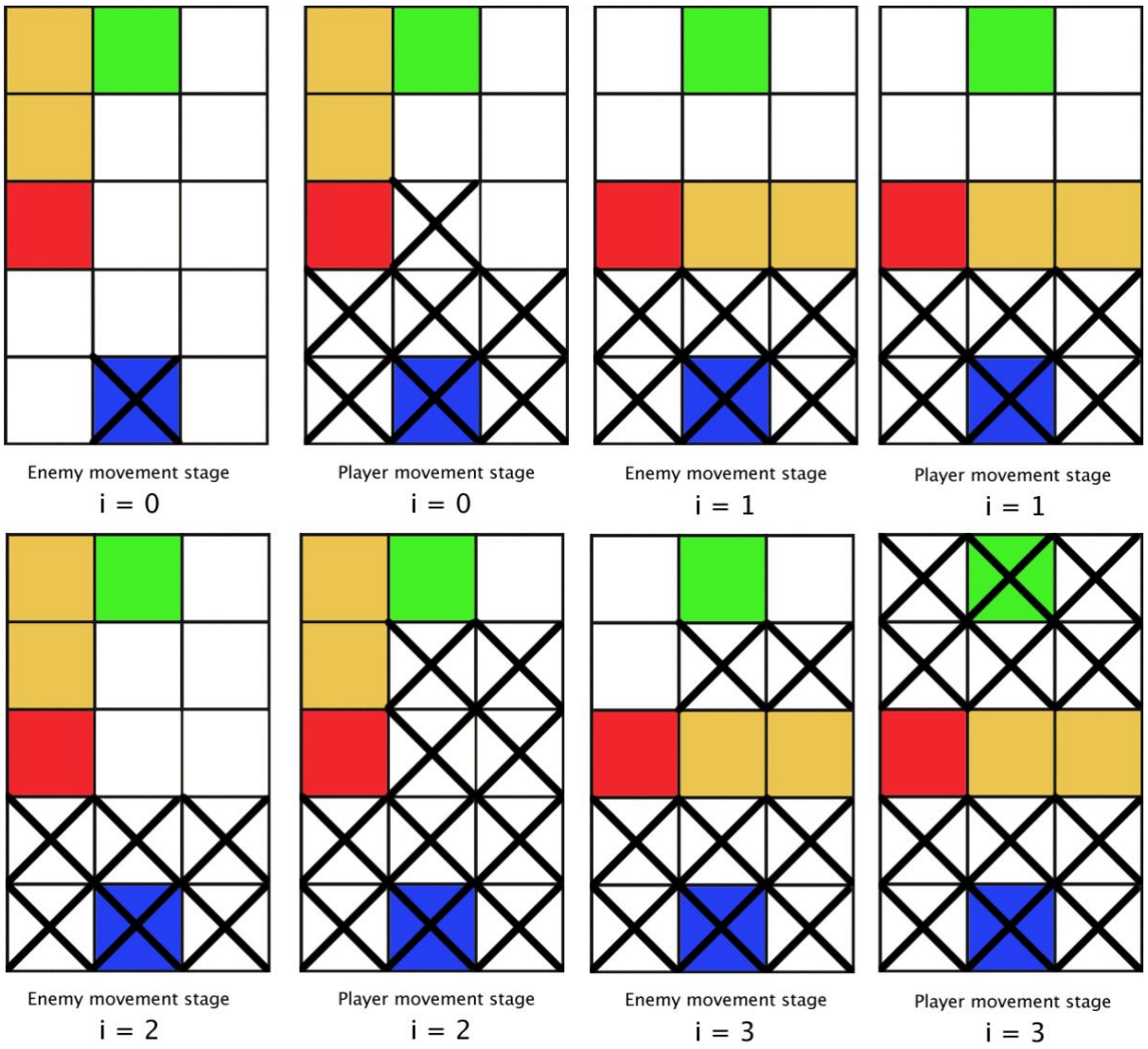
For example, if a speed of 3 is given to the player (i.e. the player can move 3 tiles in 1 time step) and we just started out the simulation (so the player could possibly be only in the spawn point), then the player simulation would mark as possible positions reached all the tiles that have distance (to be precise, Manhattan distance, since flood-fill is performed only along the four main directions) less or equal to 3 from the spawn point and that are not unreachable because of enemies.

Briefly, the algorithm iteratively calculates all positions the player can reach in 1 time step starting from the actual positions it has reached (at the start, just the spawn point), then moves the enemies to next step and removes player from positions occupied by enemies, move player again, and so on.

More formally, the algorithm works in a loop like the following:

- Initialize the set of player possible positions (playerPositions) to the spawn point.
- Loop until playerPositions contains the end point or maximum amount of step is performed:
    - [Enemy movement stage] Access the configuration of enemies in time step i and remove from playerPositions the positions that coincide with enemy positions.
    - [Player movement stage] Starting from playerPositions, update it adding all possible positions the player can reach in 1 time step (this is the flood fill part of the algorithm).
    - Go to next time step (i = i + 1)
    - Repeat

The following example shows the algorithm at work in a simple 5x3 level with just one enemy having two states (same position, rotation is 180° in the first state and 270° in the second). Tiles marked with X are the aforementioned playerPositions, i.e. all the positions the player may have reached in that time step. Note that in the picture, the two stages of the algorithm (enemy movement and player movement) are depicted separately. In this specific example the algorithm returns true since the goal is reachable.



Edge cases

Since the internal simulation is discrete in time and space, whereas the game is continuous both in time (enemies do not suddenly move from one state to another, but rather interpolation occurs) and space (a tile may be half occupied by an enemy), the solvability algorithm, as implemented now, is not always accurate.

The most frequently occurring edge case, not handled by the algorithm, can be found in the previous picture (from i=2 to i=3): in the game, the enemy does not rotate instantaneously from 180° to 270°, but rotates continuously degree by degree. As a result, the field of vision

of the enemy actually touches the two tiles (2, 4) and (3, 4) (those near the goal point marked with X at i = 3) when changing state, so the player would be eliminated if in those positions.

A solution to this problem may consist in removing from the possible player positions not only the next enemy configuration, but the all the tiles between the current and the next configuration, found using interpolation. Another solution can be to increase the number of states of the enemies in order to define their movements in a more fine-grained way: in this way, the approximation errors between discrete and continuous domains reduce.

To keep the algorithm simple, these solutions have not been adopted. Anyway, in practice, the algorithm is almost always able to correctly tell whether a level is solvable or not.

Another discrepancy between the real game and the internal simulation concerns the player movement. In the game, the player can also move diagonally but the flood-fill part of the solvability algorithm considers only the four main directions. This choice was taken on purpose in order to make the algorithm more conservative, thus mitigating the edge case explained above. Moreover, both approaches (4-way flood-fill and 8-way flood-fill) were tested, but it was proven empirically that 4-way flood-fill improved algorithm reliability.

Triviality algorithm

The goal of this algorithm is to return whether the level is trivial, given all the information about the level. A level is said to be trivial if there exist a path that is walkable by the player in every configuration of the enemies.

In order to check for triviality, a single set containing all tiles occupied by all enemies in every configuration must be computed. Then the set representation is converted into a graph representation (8-connected, with weight $\sqrt{2}$ for diagonal edges and 1 for horizontal/vertical ones) and given as input to the A* algorithm for pathfinding. The latter could either return a path, which means that the level is trivial, or that there is no possible path, which means that the level is nontrivial.

The A* implementation uses the Diagonal distance (also known as Octile distance) as estimator, which is an always underestimating heuristic for an 8-connected graph. This ensures that the path returned by A* is an optimal path, which is not strictly needed for this use case, but other non-always underestimating heuristics like Manhattan distance were tested and no performance gains were noticed. Moreover, Diagonal distance was chosen instead of Euclidean distance because it better approximates the real distance (it always underestimates, but to a lesser extent than Euclidean), hence reducing computation time.

Another note on the implementation of A* is that to handle the infinite value, a very big number has been used (namely, float.MaxValue, which is 3.4*10^38). This is more than enough for the present use case, since maps up to around 2.4*10^38 x 2.4*10^38 could be created, which is obviously ridiculously big for a game like this one.

Any algorithm that checks whether it is possible to go from node X to node Y of a graph could have been used in place of A*, and even be more computationally efficient. The choice of A* is justified by the fact that this algorithm returns a natural path that resembles

the one a real player would take when playing, so the new enemies generated to block this path will be positioned in a more natural way as well.

5. Conclusions

Evaluating the quality of a content generator is not an easy task, because it does not only depend upon objective and measurable metrics, but also on subjective and variable factors like the design of the target game, player tastes, and so on.

As far as measurable goals are concerned, the most important one for the present PCG system is that the generated level must be solvable. Even though it is not always the case with the present solvability algorithm, it can be proven empirically that a false positive is very rare (in a test with 100 levels that passed the solvability algorithm check, only 1 of them was found to be unsolvable in practice), and in any case different solutions have been proposed that should make the algorithm 100% accurate.

Another measurable goal of the system was to achieve low enough computational time, since the algorithm must execute real time during gameplay. This goal has been achieved, since the time required to generate a level varies from a minimum of few hundred milliseconds to a maximum of about 1 second. The maximum is reached when generating small levels, since they usually happen to be unsolvable even with a small number of enemies, and generation must start again from scratch.

The other goal of the PCG system, which was to create interesting and fun levels, is not easily measurable: a questionnaire should be devised and submitted to a significant number of players. Anyway, the generated levels are guaranteed to be always nontrivial, which assures that, at least to some degree, they are fun to play.

Further work

- Optional or additional goals can be added, like tokens or keys to collect. This can be done easily by placing them in a position that is reachable but in a nontrivial way. This addition adds a level of complexity to the game and forces the player to explore the game field more thoroughly, whereas, as of now, in certain levels the goal can be reached even by skipping large chunks of the map.

- The difficulty parameter, as of now, is pretty naïve, since it only influences game field size and number of enemies. Other metrics to evaluate difficulty could be taken into consideration, for example the number of steps needed by the solvability algorithm to reach the goal, how many times the player must go backward, and so on.

- The PCG algorithm focuses on generating a single square field (a room), but a game may use it to generate a whole dungeon with multiple rooms. An algorithm for procedural generation of dungeons (like agent-based dungeon growing) can be used, then each room can be populated independently from others using the PCG system presented in the present document.

- Right now, enemies are generated starting from a set of predefined templates, but it would be possible to devise a procedure to also generate templates procedurally. This will bring more variety to enemies.

Appendix: source code organization

The source code of this project is inside the Assets/Scripts folder. In particular:

- Generator.cs contains the level generation algorithm.
- Verifier.cs contains both the solvability algorithm and the triviality algorithm. The latter relies on A*, which is implemented by the scripts of folder AStarAlgorithm.
- *EnemyFactory.cs are the templates for enemies. They all implement the interface IEnemyFactory.cs. The file EnemyFactoryUtility.cs contains utility functions used by the templates.
- Enemy.cs and Map.cs are classes representing enemies and the game field respectively.
- Instantiator.cs is the script that, given PCG output, instantiates the game objects for the actual level.
- Orchestrator.cs animates enemies instantiated by the Instantiator.