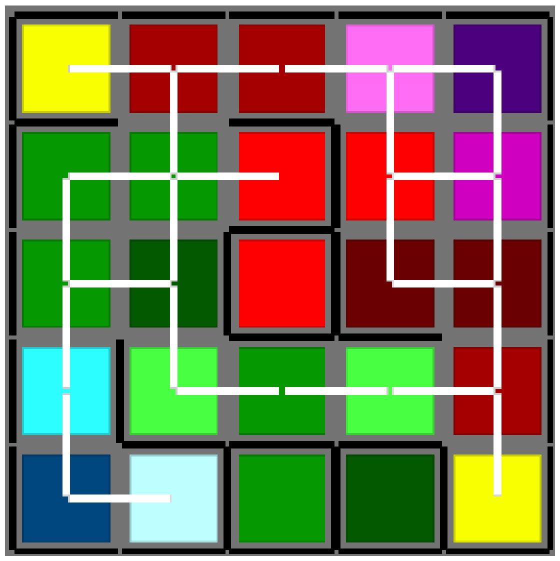
DBGA Programming Challenge

Programmatic square grid generation.

Amedeo Cuttica

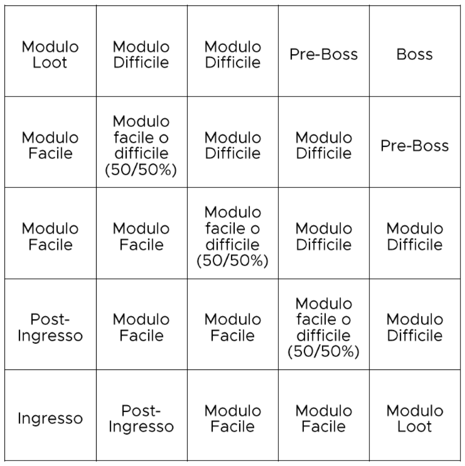
February 2021



Contents

1 Problem Description

It is required to build a system for the programmatic generation of square grid levels, composed of adjacent cells and walls between and around them. Cells in the grid are characterized by different types, and the scheme for cell types, at least in the case of $5\,x\,5$ grid, is the following:



Modulo Loot	Modulo Difficile	Modulo Difficile	Pre-Boss	Boss
Modulo Facile	Modulo facile o difficile (50/50%)	Modulo Difficile	Modulo Difficile	Pre-Boss
Modulo Facile	Modulo Facile	Modulo facile o difficile (50/50%)	Modulo Difficile	Modulo Difficile
Post-Ingresso	Modulo Facile	Modulo Facile	Modulo facile o difficile (50/50%)	Modulo Difficile
Ingresso	Post-Ingresso	Modulo Facile	Modulo Facile	Modulo Loot

Figure 1: Reference for cell type scheme.

Within the specified type, the specific cell variant has to be chosen at random among the provided ones.

In addition, of all of the interior walls of the grid, that is excluding walls that surround the grid perimeter, only a specified amount of them should be *closed*, in the sense that it prevents the passage between the two adjacent cells. The rest of them should be an *open wall*, which is basically equivalent to a *door*.

Specifically, doors should be generated so that the level satisfies the following requirements:

- **LootCell**s and the **BossCell** should always be reachable starting from **EnterCell**.
- **EnterCell** and **BossCell** should always be connected via a *door* to their adjacent post/pre cells.

2 Generation Procedure

We split the problem resolution into two distinct tasks, independent from each other. The first task consists in the generation of the square grid, together with all interior and exterior walls, and the assignment of cell types. The second task

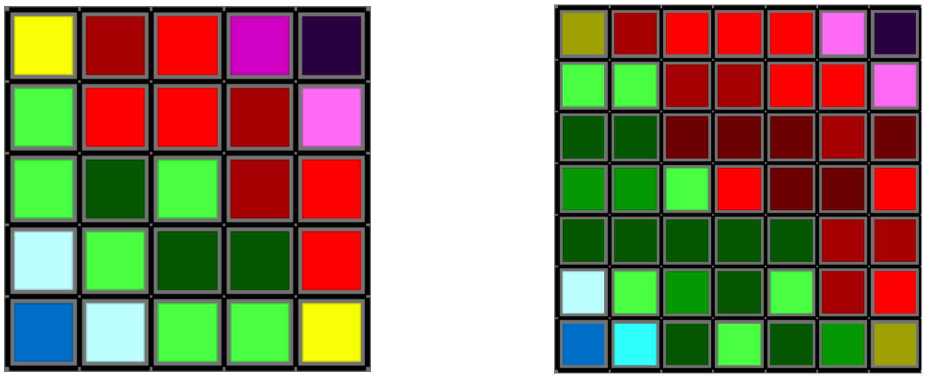


Figure 2: Example of grid with different dimensions but same scheme.

will then consist in the generation of a path, that is the opening of doors, connecting all of the necessary cells. The two tasks may be considered independent as they both directly depend on cell positions or, equivalently, on cell indexes.

2.1 Grid Generation

In the following, the terms cell and wall will refer to instantiated GameObjects, that we keep record of via the scripts/objects Wall and Cell, respectively, for each one.

Cell Generation. Regarding the positioning of the different types of cells in the grid, we assume what we may refer to as a *strict scheme*. In particular, we assume that, regardless of the grid dimension, the configuration is the following:

- **Enter** in the lower-left corner.
- **PostEnter** in the two cells adjacent to **Enter**
- **Boss** in the upper-right corner
- **PreBoss** in the two cells adjacent to **Boss**
- **Loot** in the two remaining corners
- **Easy** or **Hard** on the *upper-left to lower-right diagonal*
- **Easy** in the remaining cells below the diagonal
- **Hard** in the remaining cells above the diagonal

Cells are stored in a *dimension* · *dimension* bi-dimensional array. In fact, cells are created by iterating over rows and columns of such an array and, at each slot, instantiating a new cell. The world position of each cell directly depends on its row and column indexes. Similarly, based on the above scheme, the cell type can be derived based only on row and columns indexes. In other words, at the moment of creation, the type of the cell that is about to be created is known a priori. Then, given the type, the system picks one random prefab among the ones provided by the designer for that specific type.

Wall Generation Once all of the cells have been created, we proceed with the generation of walls between and around them. In particular, for a s*dimension x dimension* square grid, we have:

$$\begin{aligned} NumInteriorWalls &= 2 \cdot x \cdot dimension \cdot (dimension - 1) \\ NumExteriorWalls &= 4 \cdot dimension \\ NumWalls &= 2 \cdot dimension \cdot (dimension + 1) \quad . \end{aligned}$$

The system proceeds as follows: it first iterates over grid rows and, for each row, builds the $dimension + 1$ vertical walls required. Then, it repeats the same procedure over grid columns for horizontal walls.

In the present editor implementation, we quite strongly separate the generation of interior and exterior walls. This is absolutely not necessary, and possibly non-optimal, but it highlights the marked conceptual difference that there exists between the two. Furthermore, since interior wall are the only ones that can later be opened, that is transformed into doors, during the path generation procedure, they are the only ones that we keep record of.Interior walls are stored in a $dimension \cdot 2 \cdot (dimension - 1)$ bi-dimensional array of the form shown below: note that, in the case of a $5\,x\,5$ grid, 5 array slots are unused.

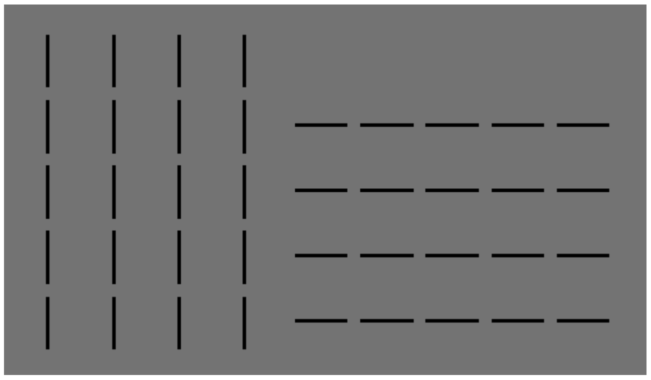


Figure 3: Configuration of the array storing interior walls.

Such a data structure allows us to retrieve any of the interior walls based on the row and column indexes of the two cells the wall is separating. In fact, indexes of two adjacent cells necessarily differ by one unit in one and only one of the two dimensions, that is row and column. Based on which of the two dimension such a difference appears in, we know whether the wall we are looking for is vertical or horizontal, and hence we know which side of the above array we should look into.

Note that up to this point, the only random procedure is the one regarding the choice of a specific cell prefab among the ones available for the cell type in question. All of the other procedures, namely cell positioning, cell type assignment and wall creation, are deterministic.

2.2 Path Generation

Once the square grid has been generated, and all of the interior walls are *closed* the next task is represented by the opening of walls in the grid in order to satisfy the necessary cell connection requirements. In this regard, the problem can be reformulated as: **25 walls have to be removed so that all the four grid corners are connected by a path of open walls.**

Our approach for such a task consists in the *exploration* of the grid and the opening of walls along the way. Wall opening is hence equivalent to path generation, and the goal is to generate paths that connect cells as required. The basic idea is the following: if we generate a path connecting the bottom-left cell and the top-right cell, and then generate paths that connect the two remaining corner with the path we just generated, then we are done, at least up to constraints on the exact number of walls that are to be opened.

In the following we present to different methods, both based on this approach. The two methods differ on the limitations posed on the way the traverses between opposite gird corners are performed. As a result, the two methods present marked differences in terms of generation power and guarantees of algorithm termination.

Direct Traverse Method As mentioned, the general procedure consists in traversing the grid between opposite corners, hence connecting all of the corners together.

In this first method, we impose that during one specific traverse, it is not possible to *turn back*. In practice this means that, for each specific traverse, only two kind of moves, or equivalently two directions, are allowed. Note that, if this is the case, then we are guaranteed to complete a traverse from one corner to the opposite one in exactly $2 \cdot (dimension - 1)$ moves, half of the moves being of one kind and half of the other. The order is irrelevant.

For instance, in the case of traverse from the bottom-left corner to the top-right corner and using a $5 \cdot 5$ grid as reference, with any combination of 4 right moves

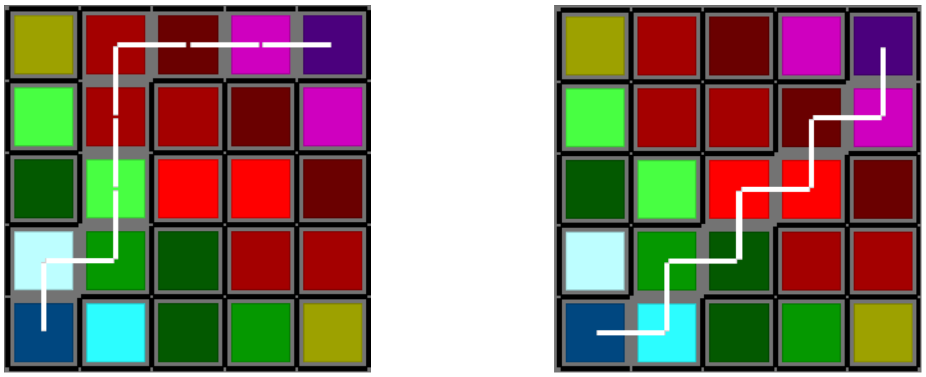


Figure 4: Direct traverses from bottom-left to top-right corners.

and 4 up moves we end up at our destination.

Once this first traverse has been completed, we then need to connect the other two corners to the generated path, hence connecting all of the four corners as required. Thanks to the topology of our square grid, we are guaranteed, while traversing along this second diagonal, to eventually intersect the previously generated path. More specifically, in order to keep the method more general, for this second step we do not generate a single traverse from one corner to the opposite one: instead, we generate two distinct traverse from each of the two corners and stop them as soon as they intersect the path generated by the first traverse.

Once all of the required corners have been connected, the remaining walls that need to be opened are chosen with different approaches. For example, for the problem specifics in question, that is a $5 \cdot 5$ grid and a number of walls to be opened equal to $40 - 15 = 25$, we will successfully connect the corners with 12 to 17 steps, so that we will be required to open 13 to 8 more walls. First of all, mandatory open walls for the bottom-left and top-right corners are opened, in the these are still closed after the precedent exploring. Then remaining walls, if present, are chosen at random among closed ones. Note that, with such an approach, we could possibly open walls connecting cells that can actually never be reached, thus virtually wasting the connection. For this specific reason, in the next traverse method an alternative approach for the opening of the remaining walls is provided.

It can be easily verified that there are paths that satisfy the problem requirements but cannot be generated via the above direct traverse method. In other words, there are correct levels configurations that the method is not capable of generating, because of the restrictions on the available moves while traversing the grid.

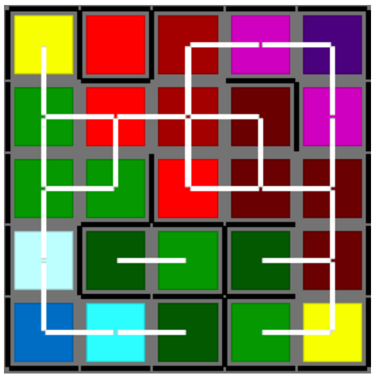
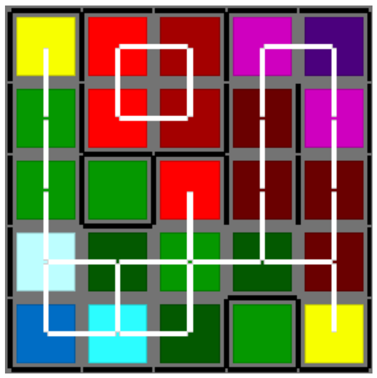



Figure 5: Examples of *isolated paths*.

Random Traverse Method We present a more general method for generating paths on the grid. This second method arises from removing the limitation on the possible moves that can be performed while traversing the grid. This means that, at each step, it is possible to go towards any of the adjacent cells, but for the previous one. Again, after a first *bottom-left to top-right traverse* has been completed, two additional traverses are started from the two remaining corners and terminated when an intersection with the first traverse occurs.

Then, remaining walls can either be opened as above, that is totally randomly, or by starting new random traverses. Specifically we start new traverses from any of the already traversed cell to any random cell in the grid, until the specified number of open walls is reached.

Now, regardless of the specific method used for the remaining walls opening, the two presented traversing methods, namely direct traverse and random traverse, have different generating powers. Clearly, this second method is able to generate every level generated by the direct traverse method. Further, we claim that this method is capable of generating any possible level satisfying problem requirements. Indeed, issues with the random traverse method are in terms of algorithm termination.

It is important to underline that this random traverse method suffers from two main issues: first of all, the random traversing from a source cell to a target cell is not guaranteed to terminate in a finite number of steps . Second, even if the traverses eventually comes to an end, we have no guarantee that the total number of opened walls/doors satisfy the problem constraints. The first problem can be mitigated by simply checking, each time a wall is opened, if the maximum number of walls that can be opened has been trespassed. Then, even if cycles are allowed, it is very unlikely for the path generating system to keep cycling without opening new doors. Hence, the algorithm is very likely to terminate

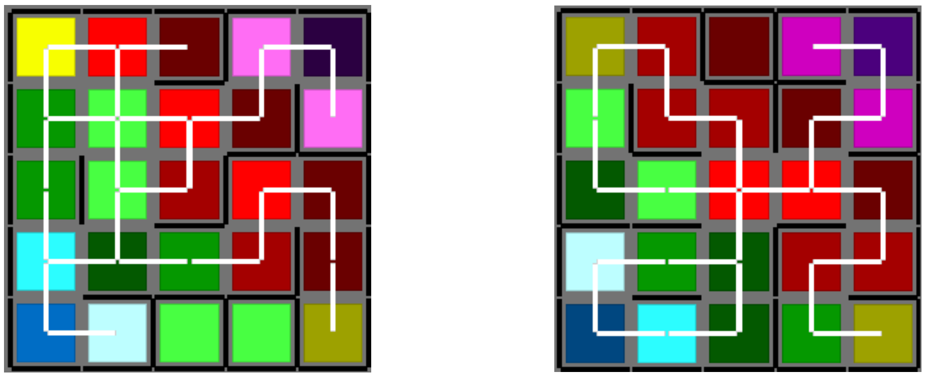


Figure 6: Examples of level configurations not achievable by Direct Traversing.

in a finite number of steps, whether by reaching the desired destination or by exceeding the available number of doors that can be opened. On the other hand, the second issue mentioned above is much more difficult to deal with, since we have no way to force the traversing algorithm to reach its destination with a maximum number of steps set a priori. Our solution in this case is to set a maximum number of tries and, in case the system fails to generate a level satisfying the requirements in such a number of tries, we simply generate a level using the direct traverse method (which, as explained above) is guaranteed to successfully generate a level in one single attempt.

3 Analysis of the proposed solution.

3.1 Topology

Both our traverse methods are based on the assumption, supported by the two dimensional grid topology, that when we traverse the grid from the second time (and on) we will necessarily eventually intersect a path, the one generated by the first traverse, and then connect all the necessary cells as required. This assumption fails when we pass from 2 up to 3 dimensions, that is with cube grids rather than square grids. Indeed, in three dimensions it is possible to build paths that connect opposed vertices and do not intersect with each other.

3.2 Grid dimension

In terms of scalability, the two exploring methods present major differences. Direct traverse is guaranteed to generate a path successfully connecting all of the four corners in a finite number of steps. The number of such steps, which is equivalent to the number of opened walls at least in the case of direct traverse, ranges from a minimum and a maximum value which clearly depends on the grid dimension. As long as the specified number of walls that are to be opened is in accordance with these values, the algorithm is guaranteed to terminate.

On the contrary, the number of steps that the random traverse method requires to traverse the grid is not bounded. For instance, while it is usually able to traverse a $5\,x\,5$ grid in a reasonable amount of time, the required number of steps could easily grow large as the grid dimension grows. For the same reason, stronger limitations on the number of walls that are to be opened strongly influence the probability for the random traverse method to successfully generate a *correct* level.

3.3 Cell type scheme

In the above we assumed what we called a *strict scheme* for the different cell types over the grid. In particular, we assumed that the four key cells that are to be connected together correspond to the four square corners. If this assumption fails, that is if we allow for special cells to be placed anywhere around the grid, other assumptions that we made us of for grid traversing fail with them.

In particular, traverse methods are based on the guarantee that traverses on the two diagonals intersect each other. If traverses are between any cell in the grid, it would be actually possible, for traverses after the first, to arrive at destination without intersecting any other path. While this doesn't represent a problem for random traversing, since it could be simply imposed to successive traverses not to stop until another path is reached, it actually gives direct traversing a chance to fail. In this case, since it is not possible to *turn back*, if a successive traverse goes past the first one without intersecting it, then it is not able to recover: the method will not successfully generate a level.

3.4 Graph structure

When speaking about traversing the grid we speak about going cell to cell at each step choosing among adjacent ones. This can be seen as the traversing of a graph where cells correspond to nodes ad the relation of adjacency in the grid corresponds to edges. Now since such a relation can be derived based on the respective row and column indexes of each cell, we do not actually need an object acting as edge in the graph and carrying information about the two cells it is connecting. Similarly, cells are not required to directly carry information about adjacent cells since, again, this kind of information can be retrieved starting from indexes in the cell matrix.

In the present implementation, the two traversing methods are characterized by a different notation for indexing cells and walls. In particular, the notation used for the random traverse method hides much of the underlying matrix structures and presents the traversing as more of a proper graph traversing. Still, it is important to note that this difference is only in terms of notation and the technique used to retrieve adjacencies and separating walls is the exact same, that is, based only on cell and wall indexes in their respective bi-dimensional arrays.

4 System workflow and instructions.

The system for the programmatic level generation can be tuned in multiple ways. Designers that are willing to use the system get to act at two distinct levels: by tuning script parameters and by providing their own prefabs for grid objects. In particular, following the order of level generation as presented above, they are required to make decisions at various steps as presented below.

The basic configuration assumes a GameObject holding the following four scripts:

- **LevelBuilder**, which is responsible of the grid construction.
- **PrefabLoader** which is responsible for the loading of all prefabs from respective resources folders
- **LevelExplorer**, which is responsible for path generation/wall opening.
- **Level** which acts as a static holder for both cells and interior walls.

All of the generated GameObjects that will constitute the level will be children of this one.

4.1 LevelBuilder

dimension represents the grid dimension, in the sense that a matrix of $dimension \cdot dimension$ cells will be generated.

Cell Distance determines the distance in world units (meters) between centers of adjacent cells. Walls are simply placed midway. *Cell Scale* determines the amount each instantiated cell prefab will be scaled of, and hence its dimension.

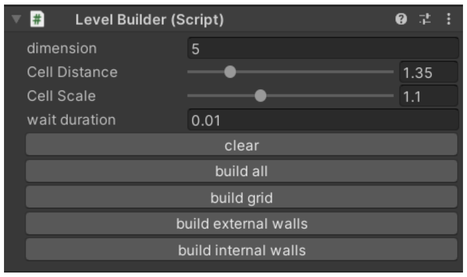

Level Builder (Script)
dimension 5
Cell Distance 1.35
Cell Scale 1.1
wait duration 0.01

Figure 7: Tuning for grid shape

Finally, *wait duration* regulates the time interval between each generation step. This allows a gradual level generation in edit mode, but is irrelevant when play mode is started.

4.2 PrefabLoader

As explained above, cell prefabs to be instantiated are chosen at random among the available ones for the specific cell type in question. These prefabs are loaded from multiple resources folder that the Designer is required to populate with prefabs of its choice.

On the other hand, the present implementation provides a single wall prefab variant, that is simply deactivated when opened/turned into a door. Still, a system equivalent to the ones for cell prefabs, implementing the same procedure for the loading of both wall and door prefabs, could be introduced. The only requirement for cell and wall/door prefabs is to hold a Cell and a Wall script respectively.

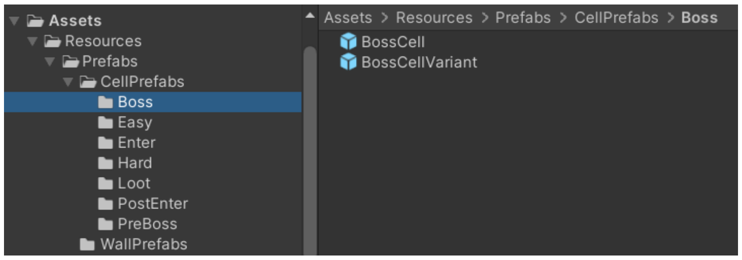


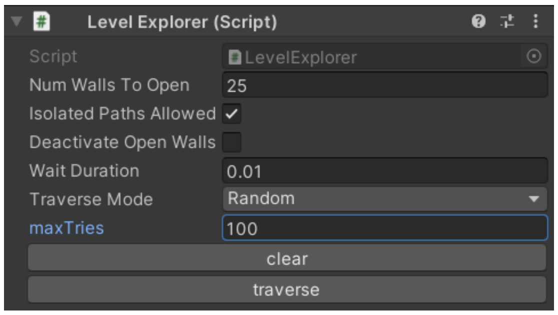
Figure 8: Example of resources folder for grid objects prefabs.

4.3 LevelExplorer

NumWallsToOpen defines the number of walls, among the interior ones, that need to be opened/replaced by doors. Referring to the original problem formulation, this value is complementary to the number of interior walls that need to be placed inside the grid.

IsolatedPathsAllowed defines which of the two methods for reaching the specified number of open walls presented above is used. Isolated paths refers to passages, that is open walls, between cells that are not reachable.

DeactivateOpenWalls is usually true for play mode but it could be useful to un-flag it in edit mode for visualizing the generated paths.



Script	LevelExplorer
Num Walls To Open	25
Isolated Paths Allowed	✓
Deactivate Open Walls	
Wait Duration	0.01
Traverse Mode	Random
maxTries	100

Figure 9: Tuning for map exploring/path generation.

TraverseMode defines whether direct exploring or random exploring is to be used. In case of random exploring, a maximum number of tries/attempts, conceded to the system for generating the level before switching to direct mode, has to be specified.