Creating adjustable cave-like terrain using cellular automata in the context of procedural content generation of strategy game maps

Project report

Jakub Sękowski

23rd February 2017

Introduction

Maps are vital components of level design for many types of games (e.g. first-person shooters, real-time strategy games and flight simulators) and their careful design (manual or procedural) contributes vastly to player experience. There are several reasons [...] why procedural generation of maps is important for game development. First, having an inexhaustible source of new maps means that levels become less predictable, which contributes to the players' curiosity and the game's life-span. Second, [...] given that content is represented in an efficient manner and the procedural content generation (PCP) algorithm is parametrizable in the right ways, maps can be adjusted to match player needs and abilities [...]. Finally, PCG can be used as an assisting authoring tool for complementing human creativity and level design expertise existent within commercial game development. [3]

In this report, I present the results of studies on automatic terrain generation in a practical setting. I tried to build a framework around cellular automata to create a cave-like terrain shape given realistic constraints. Also, I identified a handful of useful parameters, which can be adjusted in order to control the visual style of the resulting map.

The context

My work is a part of a project intended to build and maintain a comprehensive map generator for the classic turn-based strategy game Heroes of Might and Magic III [2]. The project is run by a group of students and employees of University of Wrocław (Institute of Computer Science). The main principles are fairness of the generated maps, sensible and significant logical structure and variety achieved through parameter adjustments as well as random decisions.

The terrain generation is expected to be somewhere in the middle of the process of building the map, which is briefly depicted in the following paragraphs.

We start from the map settings given by the user, including map size, number of players and their associated castles, resources richness, difficulty indicator, etc. Then, we translate these settings into the generator parameters used throughout the process.

The earliest stage focuses on the generation of an abstract map layout, which determines the flow of the game and placement of the most important objects (towns, mines) and ensures the map fairness. The layout is represented as a graph, the nodes of which are the map's zones and edges are connections between them. Each node has a number of properties associated with it, e.g. the essential game objects that are to be placed in the corresponding zone, the amount of treasures to be found there, the level of villain guarding access to the zone, etc. The layout graph as well as all this properties is generated using a kind of the formal grammar system parameterized with the values based on the user input.

Next, each zone is mapped onto the grid as a defined chunk of open space, the shape of which is based on the distorted Voronoi diagram. In order to achieve the intended connectivity schema, corresponding to the edges in the graph, the zones are surrounded by non-traversable borders except for the precisely positioned walkable cells, acting as portals between selected neighbouring zones.

Additionally, we have an intention for the terrain to be highly controllable in order to fulfil the typical kinds of requirements in the level design process. Thus, similarly to non-traversable zones' borders, we allow for a cell to be declared as necessarily traversable. The main purpose behind this idea is to be able to set up the paths between the important game facilities, which are guaranteed to remain walkable.

The map prepared in this manner is input to the terrain generator, which is the main subject of this study. After that, the map has to be filled with the rest of the content - treasures, monsters, and other game objects. This is planned to be done in a heuristic-based way i.e. expected player's level is to be computed in for each zone of the map and the game objects placed in that zone are to be selected deliberately to suit the player's needs and

abilities at this point.

It is worth to noticing that the model is general and it can be used for most of the strategy video games, not only Heroes of Might and Magic III.

Task specification

The map is represented as a two-dimensional grid of cells, each of which can be in one of the four feasible states: *black*, *white*, *super-black* or *super-white*. Here is the semantics of these four states:

- *white* and *super-white* cells represents traversable terrain;
- *black* and *super-black* cells are non-traversable;
- cells that are *super-white* and *super-black* are special in a way that their state is immutable; i.e. there is a reason for these cells to be traversable or non-traversable respectively and therefore they must remain such throughout the process.

The goal of the terrain generator is to transform the map in such a way that *super* cells remain untouched and the rest is shaped as a reasonably looking strategy game map. More formally:

- INPUT: an arbitrarily filled up grid of cells and a number of parameters, whose possible values and meaning is going to be specified during the process of algorithm design
- OUTPUT: the new grid that is compatible with the input on the *super-white* and *super-black* cells, and each of remaining ones is either *white* or *black*; it is supposed to respect the input parameters and simultaneously maximise the subjective aesthetic value when interpreted as a cave-like terrain

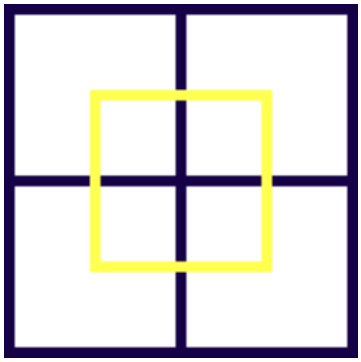

Figure 1: Sample map for testing purposes. Dark blue and yellow indicates *super-black* and *super-white* cells respectively. *White* cells are to be changed by the generator.

As a test case, I am going to use a 33x33 sample input map divided by *super-black* borders into four 15x15 zones. In order to guarantee the connectivity between them throughout the process, middle points of the neighbouring zones are connected with *super-white* line segments through the narrow gaps in the borders (see Figure 1). The actual inputs are about to be structured similarly except for the possibly more sophisticated shapes of zones and the connectivity schema (not every border is going to have a gap).

Cellular automata

A cellular automaton is a discrete model developed in the field of artificial life. It consists of (possibly infinite) grid of cells and a fixed rule for changing their states. For each cell there is a set of other cells, defined relative to the specified one (the centre) called its neighbourhood. Typically, the rule of state transition is a mathematical function of a specific cell's current state and its neighbourhood i.e. it depends only on its arguments, it is the same for every cell, it is deterministic and it does not change over time.

The simulation can be run for an arbitrary initial generation (each cell has some state assigned to it). Then, in each step, the new generation is created by applying the transition rule to every cell in the grid simultaneously.

The two most commonly used types of neighbourhood are the von Neumann neighbourhood and the Moore neighbourhood. The former consists of the four adjacent cells. The latter includes the von Neumann neighbourhood and additionally the four remaining cells surrounding the cell whose state is to be calculated. [4] See Figures 2 and 3. A notable well-known example of a cellular automaton is the *Conway's Game of Life* [5], whose rule is based on the Moore neighbourhood.

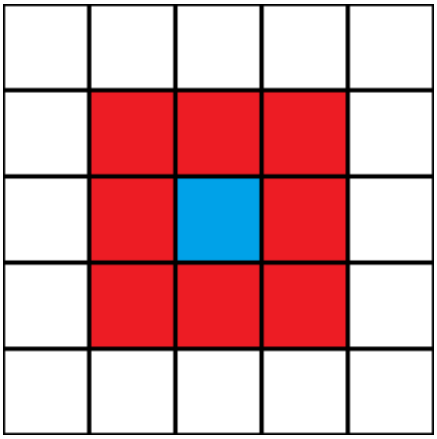

Figure 2: *The red cells are the Moore neighbourhood for the blue cell.* [4]

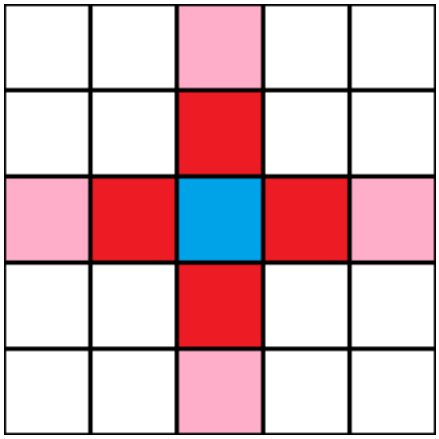

Figure 3: *The red cells are the von Neumann neighbourhood for the blue cell. The extended neighbourhood includes the pink cells as well.* [4]

A special class of cellular automata is called *outer totalistic cellular automata. The state of each cell in a [outer] totalistic cellular automaton is represented by a number (usually an integer value drawn from a finite set), and the value of a cell at time t depends only on the sum of the values of the cells in its neighborhood [...] including the cell itself [...] at time t - 1.* [4]

The model I am using in this study is indeed an *outer totalistic cellular automaton*. Cells in states *white* and *super-white* correspond to integer 0.

Black and *super-black* are seen as 1. The rule is determined by a single number only - a survival threshold. The cell x is *black* in time t if and only if the sum over its neighbourhood in time $t-1$ is not less then the threshold value, otherwise it is *white*. The rule applies to non-*super* cells only, since *super* states are immutable.

The algorithm

The generation algorithm works as follows:

1. It reads the input map filled with *black*, *white*, *super-black* and *super-white* cells.
2. Then, to each non-*super* cell, regardless of its current state, it assigns *black* or *white* at random with **p** being the probability of a *black* state.
3. It produces the next generation of automaton, by computing the new states for every cell **x** simultaneously, according to the following rule:
   - If **x** is *super-black* or *super-white*, then its state is preserved in the next generation.
   - Else,
     - let $value(\mathbf{y})$ be 1 if cell **y** is *black* or *super-black* and 0 if it is *white* or *super-white* for every cell **y**;
     - let **V** be the sum of cell values over the neighbourhood of **x**;
     - new state of **x** is *black* if $\mathbf{V} + \mathbf{s} \times value(x) \geqslant \mathbf{t}$ and *white* otherwise.
4. Previous step is repeated until **i** iterations is reached.
5. The **i**-th generation of automaton is returned as a result.

It is easy to see, that this procedure depends on a handful of parameters:

- **p** - the probability of *black* cell in the initial generation of automaton;
- **N** - the neighbourhood type (e.g. Moore, von Neumann);
- **s** - the "self weight" i.e. the degree with which the current state of a cell contributes to its next state calculation;
- **t** - the "survival threshold" i.e. minimal value of $\mathbf{V} + \mathbf{s} \times value(x)$ necessary for a cell to become (or remain) *black*[1];
- **i** - the number of generations of cellular automaton to be computed.

[1] See the algorithm description for details.

For the sake of completeness, the random seed used for determining the cells' states in the initial generation should be considered an additional parameter. A set of parameters defined in this manner guarantees an extremely desirable property: the same collection of parameters' values (including the random seed) provided to the algorithm will cause the same map to be regenerated for a particular input.

Evaluation

I have generated the terrain for the sample input map with a vast variety of the parameters' values. I used both the Moore and von Neumann neighbourhoods. The exact method as well as the parameters are thoroughly described in *The algorithm* section. The source code is available in the GitHub repository.[1]

In order to highlight the way the parameters affect the result, I reused the initial generations of the automaton as much as possible, since this is the only nondeterministic stage of the procedure. In particular, for each value of parameter **p** all the generated maps are based upon the same initial state.

Also, I decided to adjust the value of **t** automatically to suit the selected set of **p**, **s** and **N** values. The reason is that, as preliminary experiments have shown, at most two **t** values make sense given the other parameters. Usually, a single one can be named that is optimal for generation. Less values lead to maps almost completely filled with *black* cells, whereas greater ones produce maps nearly entirely *white*. In order to pick the right **t** value I was generating three iterations of automaton for $\mathbf{t} = 1, 2, 3, 4...$ and selecting the least **t** producing the map for which the number of the *white* cells is not less than the number of the *black* cells.

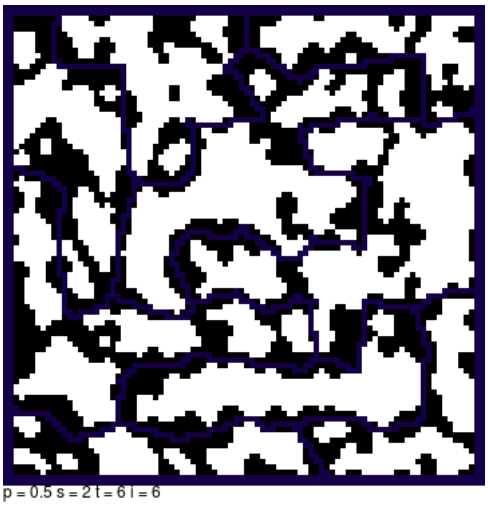

Figure 8: The terrain generated for a realistic-shape input. No *super-white* cells were present so the caves' connectivity is low. For the results like this one the further post-processing is necessary to create tunnels between the rooms.

A handful of notable examples of the maps created are presented in Figures 4, 5, 6 and 7. Also, Figure 8 shows the terrain generated for the input Voronoi diagram[2], which looks far more realistic.

[2]see *The context* section for details

Another thing about this way of terrain generation is that the cellular automata are extremely efficient. Computing 6 iterations on 128x128 map using the von Neumann neighbourhood takes only 0.09s on 1.86 GHz CPU. The fact is worth mentioning since it makes the method suitable for real-time generation.

Conclusion

As the results show, the cellular automata are the fast and efficient method for generating cave-like terrain on a typical strategy game maps. They can be controlled easily through the convenient notion of immutable cells, which are guaranteed to remain (non)-traversable. The rest of the terrain produced fits these constraints quite naturally. However, earning the specific visual "style" of the map requires careful adjustments of the parameters.

Below, I present the observations I was able to make about each of the parameters, which should be helpful in the process of tweaking and adjusting.

- **N** (the neighbourhood type) is probably the most important parameter. Picking the Moore neighbourhood results in smoothly shaped cave-like structures, which can be seen in Figures 4 and 6. Whereas, the von Neumann neighbourhood produces more right angles and thus the terrain generated resembles more of a labyrinth or a complex of peculiar buildings (see Figures 5 and 7). It looks like some kind of a man-made object.
- In fact, nearly all of the values for parameter **p** can be used successfully. The near-edge ones, can produce slightly more unexpected results, but not necessarily unwanted.
- **s** affects the "frequency" of a structure. As its value increases the caves' walls become less smooth, there are more rooms disconnected from the main area and the accidental obstacles start to emerge in the middle of the open space. It is well shown in Figure 6. The von Neumann neighbourhood works better with slightly greater values of **s** than the Moore neighbourhood would prefer.
- Using the large values of **i** in most cases is not necessary, since the automaton gets to the "stable" state quickly (see Figures 4 and 5). For this study $\mathbf{i} = 6$ was the maximum, but often 2 or 3 iterations were sufficient.
- Parameter **t** is not really an adjustment, since given the remaining ones, its suitable value may be easily computed. Making this procedure an automatic step is highly recommended, since it decreases the search space substantially.

However, there is a necessary requirement that is impossible to be met through the parameters' adjustments only. All the empty space on the map (i.e. the *white* and *super-white* cells) should be reachable by the players, so the terrain cannot block the access to any cave. In order to achieve that, a post-processing of the resulting maps will be needed. For each room disconnected from the main open area a tunnel is to be created. The tunnel is defined to be a connection (path) to the closest *white* or *super-white* cell disconnected form the specified room. How exactly the tunnels are to be constructed is a design-related decision. One of the possible ways is to use the shortest connection that does not intersect with a *super-black* border, which can be computed by the breadth-first search algorithm (BFS) easily.

References

[1] Jakub Sękowski. *H3MapGen: Cellular Terrain* repository. `https://github.com/sequba/h3mapgen-cellular-terrain`.

[2] New World Computing. *Heroes of Might and Magic III*. The 3DO Company, 1999.

[3] Lawrence Johnson, Georgios N. Yannakakis, and Julian Togelius. *Cellular automata for real-time generation of infinite cave levels. (PCGames '10)*, ACM. `DOI=http://dx.doi.org/10.1145/1814256.1814266`.

[4] Wikipedia: *Cellular automaton*. `https://en.wikipedia.org/wiki/Cellular_automaton`. Version from 2017-01-05.

[5] Wikipedia: *Conway's Game of Life*. `https://en.wikipedia.org/wiki/Conway's_Game_of_Life`. Version from 2017-01-10

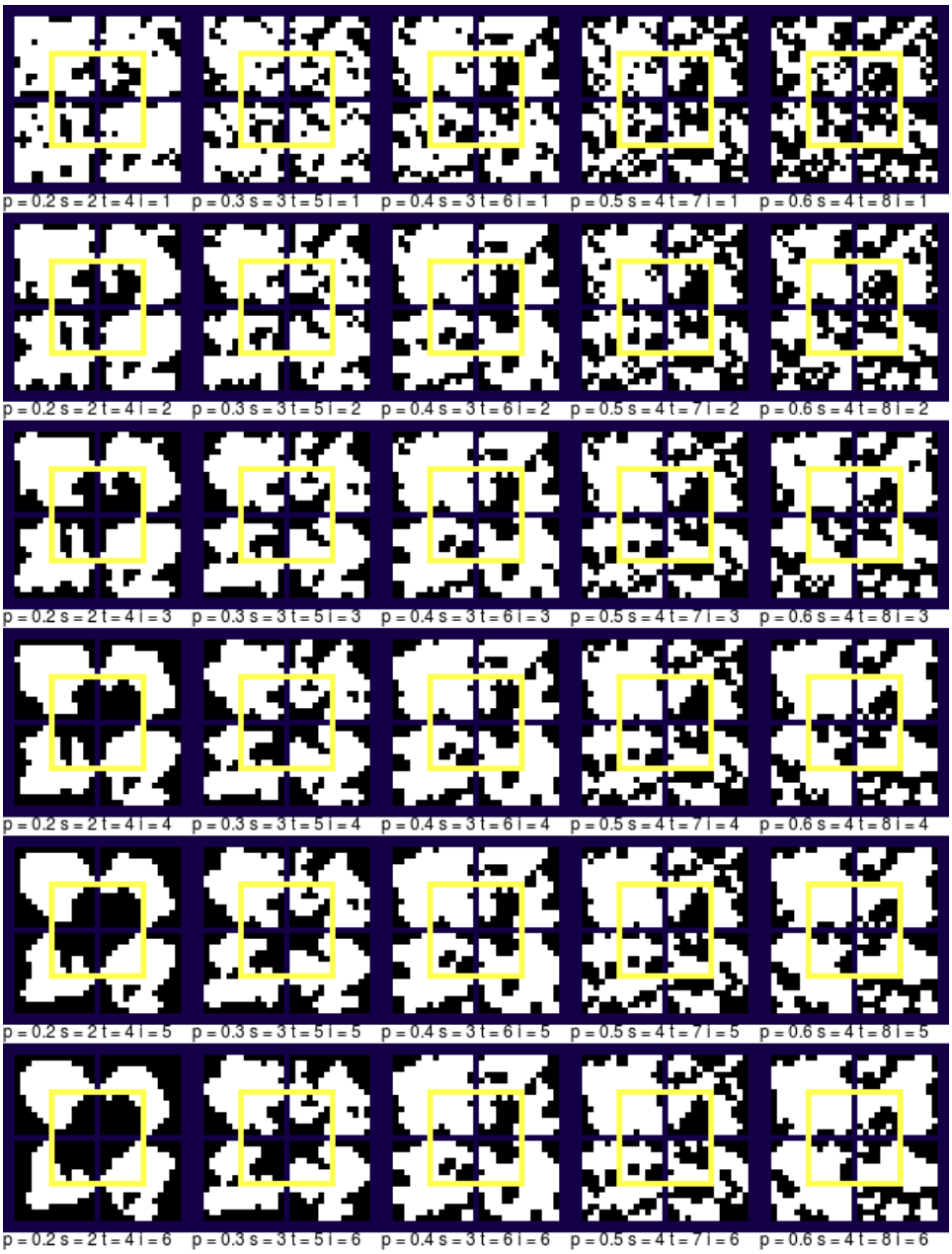

Figure 4: Six steps of evolution of the cellular automaton for the Moore neighbourhood and a number of different (**s**, **p**) pairs. (**i** = 1,2,3,4,5,6). Parameter **t** has been computed automatically.

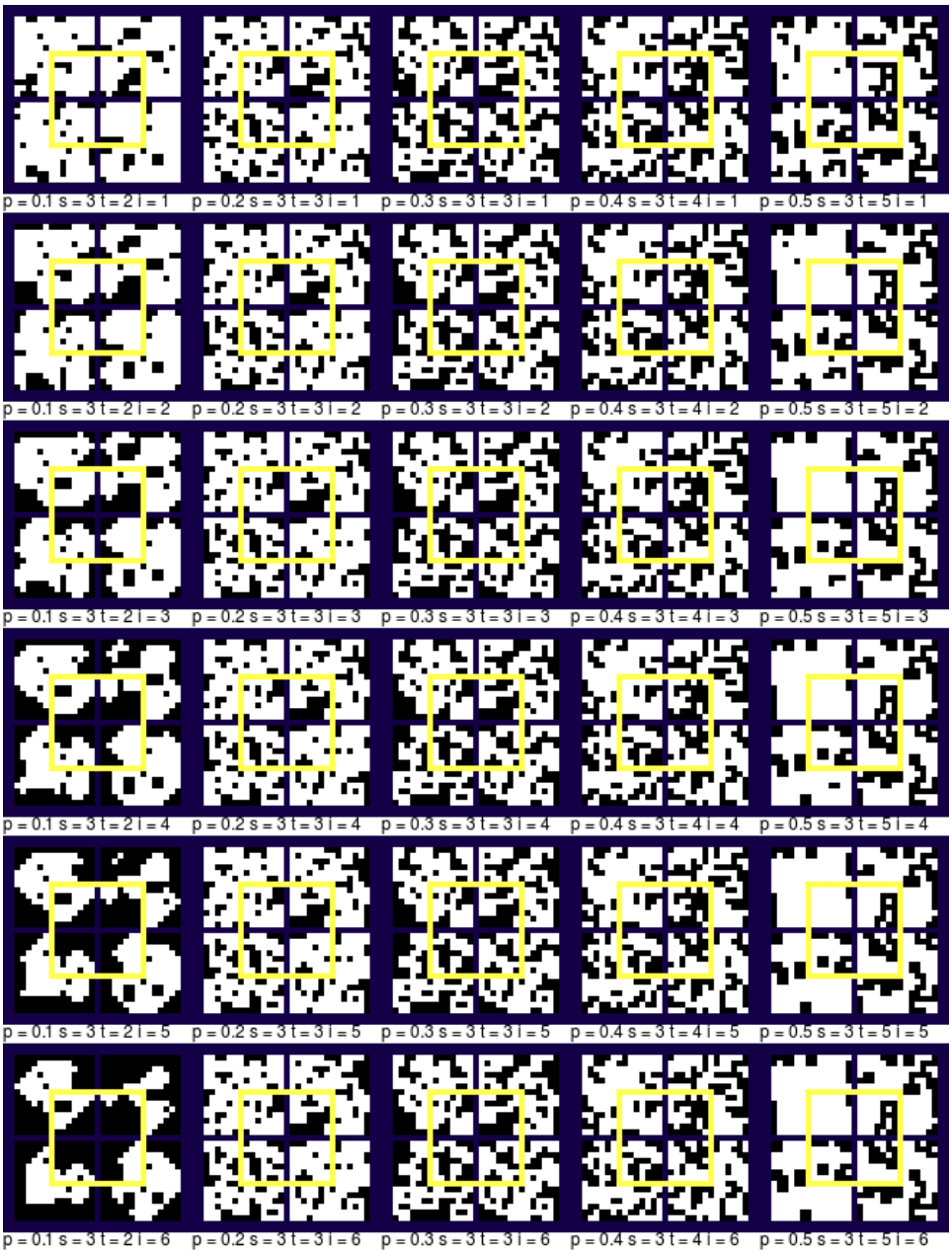

Figure 5: Six steps of evolution of the cellular automaton for the von Neumann neighbourhood and a number of different values of **p**. (**s** = 3 **i** = 1,2,3,4,5,6). Parameter **t** has been computed automatically.

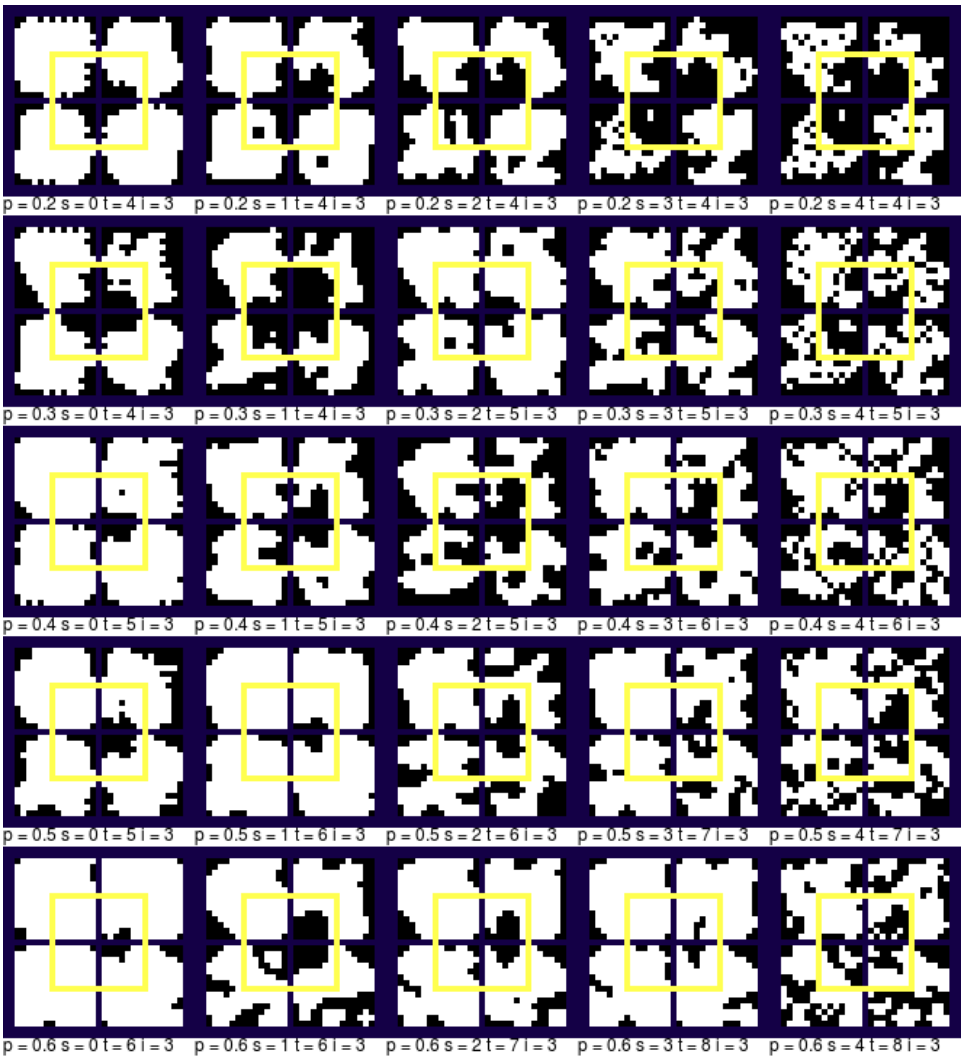

Figure 6: Comparison of results for different values of **p** and **s** for Moore neighbourhood and $\mathbf{i} = 3$. Parameter **t** has been computed automatically.

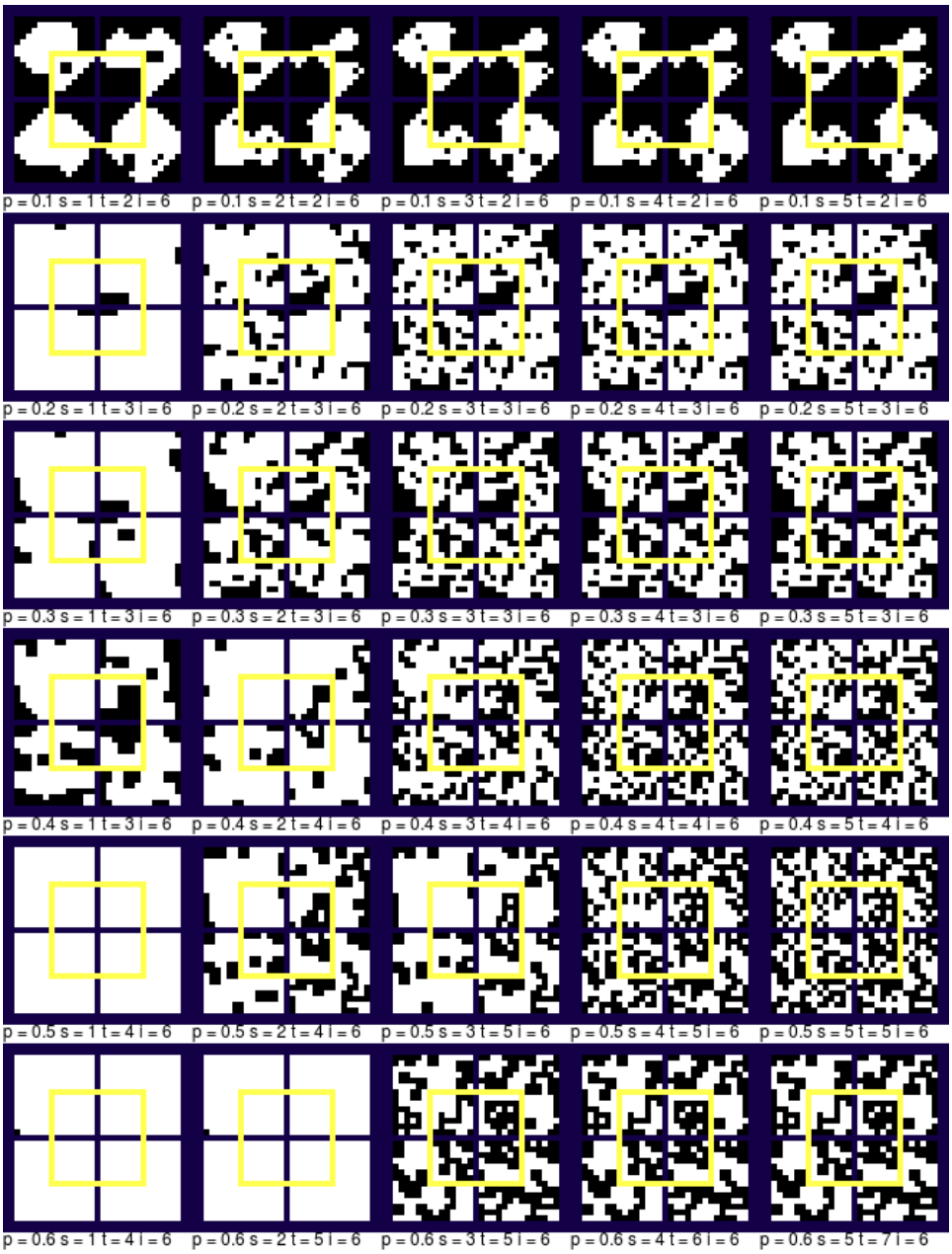

Figure 7: Comparison of results for different values of **p** and **s** for von Neumann neighbourhood and $\mathbf{i} = 6$. Parameter **t** has been computed automatically.