Overview of modern algorithms for world procedural generation in computer games

Ivan F. Laitaruk, Tetyana O. Hryshanovych

Lesya Ukrainka Volyn National University, 13 Voli Ave., Lutsk, 43025, Ukraine

Abstract

This article analyzes the most common algorithms and approaches of worlds' procedural generation that are used today, and considers real use cases of these methods in order to obtain the best understanding of the use of these algorithms in specific situations. There are also the estimations of algorithms' time complexities to determine whether they are justified with high-quality output virtual environment. Among such algorithms are graph grammar, Fortune's algorithm, Perlin noise construction, cellular automaton construction, genetic algorithm and others.

Keywords

Graph grammar, Voronyi diagram, Fortune's algorithm, Minkowski metric, Perlin noise, cellular automaton, genetic algorithm

1. Introduction

The repetitiveness of game elements leads to fast indifference from players, so sometimes developers decide to add "randomly" generated objects, textures, storylines, quests, worlds, etc. Data generation using algorithmic functions with randomness is called procedural content generation (PCG). The issue of this approach is that such algorithms are frequently resource-intensive and unpredictable in the generated content.

The purpose of the work is to assess the algorithmic complexity of world procedural generation algorithms: graph grammar, Fortune's algorithm for constructing Voronyi diagram, Perlin noise, cellular automaton and genetic algorithm. It is also necessary to analyze the use of distance metrics in the Voronyi diagram, fractional Brownian noise, and physics simulation as approaches to procedural generation.

2. Generative grammar

Generative grammar was originally used to create phrases by specifying rules to select a lexeme (term) from a set of logically correct options [1]. Therefore, it is possible to form a given number of grammatically correct constructions. Based on this, other grammars were developed such as graph rewriting[1] and form grammar.

2.1. Graph grammar

Graph grammar is commonly used for world-building of different types of dungeons, castles, and other structures for role-playing (RPGs) or strategy games. It considers the world to be represented as a graph [2]. Firstly, an initial graph (randomly selected from a template) is defined, and then rules

CS&SE@SW 2024: 7th Workshop for Young Scientists in Computer Science & Software Engineering, December 27, 2024, Kryvyi Rih, Ukraine

laitaruk.ivan2024@vnu.edu.ua (I. F. Laitaruk); hryshanovych.tatiana@vnu.edu.ua (T. O. Hryshanovych)
https://vnu.edu.ua/uk/staff/hryshanovych-tetyana-oleksandrivna (T. O. Hryshanovych)
0009-0002-9832-7759 (I. F. Laitaruk); 0000-0002-3595-6964 (T. O. Hryshanovych)

© 2025 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

[1] Graph rewriting can sometimes be used instead of graph grammar.

CEUR-WS.org/Vol-3917/paper21.pdf

for subgraph transitions from one to another are sequentially applied. Typically, rewriting rules are specified manually to ensure logical transitions.

The most popular approach to graph rewriting is the algebraic approach, in which the double pushout and the single pushout methods are defined. The **double pushout** method involves using rules of three graphs: a left-hand L, a right-hand R, and a context K (often called the interface). The graphs L and R specify a transition using the context K: $L \supseteq K \subseteq R$. There is a homomorphic subgraph to L in the graph G and all elements that are present in L but not in K are removed from G. After that, the elements of R that are not in K are glued to G (figure 1). This method is strict, which allows to avoid cases where conflicting changes are applied to the same subgraph. The **single pushout** method is simplified and operates only with graphs L and R. Elements of G that do not correspond to graph R are removed, and new elements from R are added to G (figure 2). This method is more flexible, but can cause issues with compatibility of changes.

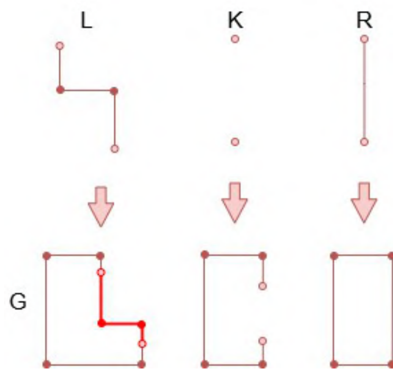

Figure 1: Double pushout example. Subgraph L is matched with part of graph G and with a context K, this part can be replaced to subgraph R.

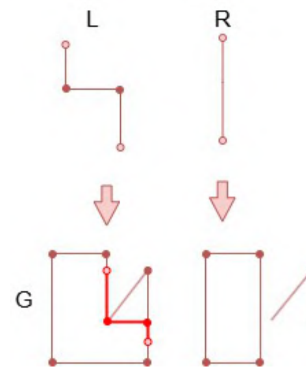

Figure 2: Single pushout example. The absence of context K can cause redundancy in result graph.

The time complexity of graph rewriting mostly depends on the sizes of an input graph G and a left-hand graph L, since finding subgraph homomorphism is an NP-complete problem, that can lead to exponential complexity. Therefore, the time complexity of searching for a mapping position in a graph G by a graph L is $O(V_G * V_L)$ in simple cases, and $O(V_G^{V_L})$ in the worst case, where V_G is the number of vertices in G and V_L is the number of vertices in L. The time of applying a rule is $O(V_R)$, however, it is usually a constant value. Considering that rewriting a graph processes n rewriting rules, the time complexity is $O(n * V_G * V_L)$ in the simple case and $O(n * V_G^{V_L})$ in the worst case. The space complexity depends on the size of the graph G and all rewriting rules – $O(n * (|L| + |K| + |R|))$, where n is the number of rules, $|L|$, $|K|$, $|R|$ are the sizes of graphs L, K, and R respectively. When applying the single pushout, the context size K will be absent in the space complexity.

Generating graph rewriting rules can be a challenge, as all rules must modify the graph in a way that can actually be generated. Paul Merrell proposes a new method for generating graph rewriting rules [3] (figure 3-4).

The first step is to select graphs that will serve as examples for generation. It is required that the result must be locally similar to the examples. This means that every part of the generated graph should be similar to the example at small scales, but its large-scale structure can be completely different. The examples are broken down into the primitives and glued together by all possible ways. The primitives and all glued graphs are organized into a hierarchy of subgraphs. If the new subgraph can be simplified to the one higher in the hierarchy, then a rewrite rule can be formed (new subgraph $\leftrightarrow$ simpler subgraph) (figure 5).

Graphs L and R are mutually gluable if they have the same boundary string: $\partial L = \partial R$. The **graph boundary string** is described by positive $\wedge$ and negative $\vee$ turns of the subgraph and also its half-edges (figure 6). If the positive and negative turns are consecutive, they can be reduced: $a \wedge x \wedge \vee = a \wedge x$. If the numbers of half-edges, positive and negative turns are the same, then the boundary strings of

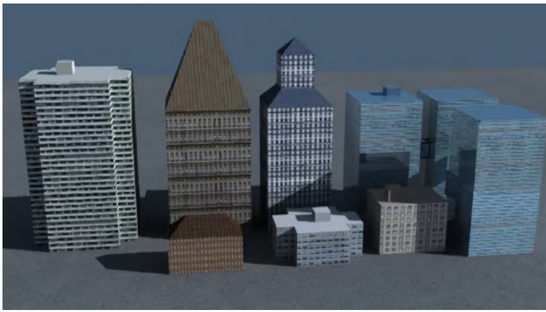

Figure 3: Examples for Paul Merrell's graph rewriting algorithm.

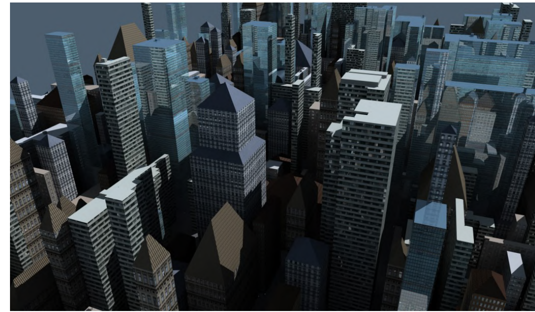

Figure 4: Generated city with graph rewriting based on examples.

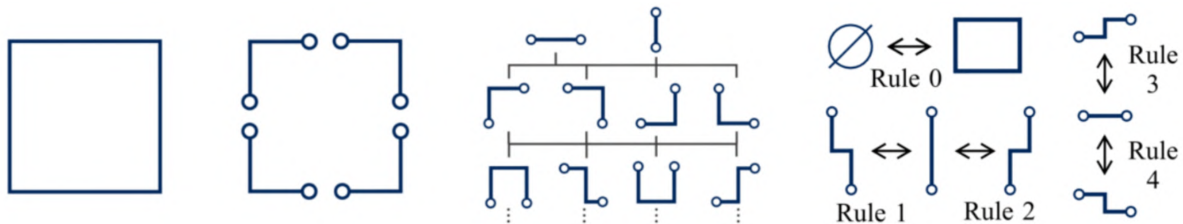

Figure 5: Input example, breaking it down into primitives, constructing a hierarchy, and forming graph rewriting rules with Paul Merrell's method (from left to right).

the subgraphs are also the same. Due to the reason that all our graphs are planar, they can be glued in linear time. There are two operations to directly glue subgraphs: loop glue and branch glue. They can be considered more in Paul Merrell's publications.

After applying a rewriting rule, there is a process of randomly defining the edge lengths of the graph. If the lengths are organized so that the graph is planar, then the decision is made, otherwise, the edge lengths are redefined.

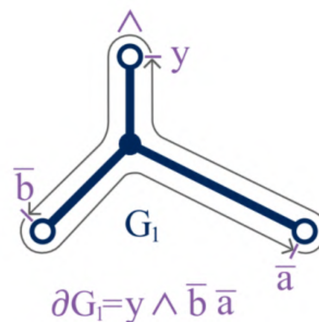

Figure 6: Graph boundary string of subgraph G_1 by Paul Merrell. $\partial G_1 = y \wedge \overline{b}\,\overline{a}$ means that through the path of graph G_1 we meet half-edges y, $\overline{b}$ and $\overline{a}$, and one positive turn $\wedge$. A starting point does not matter: $y \wedge \overline{b}\,\overline{a} = \wedge\overline{b}\,\overline{a}y = \overline{b}\,\overline{a}y\wedge$.

3. Space distribution

The **space distribution** methods refer to the algorithms that divide the entire game space among individual generation elements (e.g., rooms). Such algorithms were originally used in computer graphics for texture generation, and later found wider application.

3.1. Voronyi diagram

The **Voronyi tessellation**[2] [4] describes the entire territory, dividing it so that any point r inside a separated element is the closest to the corresponding partition point a: $dist(r, a) < dist(r, b)$, if a and b are partition points of the diagram. This can be achieved, if we draw perpendicular bisectors of the lines connecting each pair of randomly given points and choose the part of the bisector that does not intersect with others (figure 7):

$$V(a_i) = \bigcap_{\substack{1 \leq j \leq n \\ j \neq i}} h(a_i, a_j) \tag{1}$$

where a is partition point, $h(a_i, a_j)$ is half-plane formed by drawing a perpendicular bisector and containing a_i and not containing a_j. However, there are better algorithms for constructing a Voronyi diagram that do not require a pairwise search of all points.

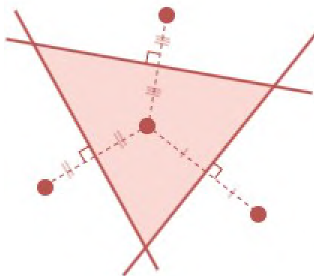

Figure 7: Constructing a Voronyi tessellation cell using perpendicular bisectors.

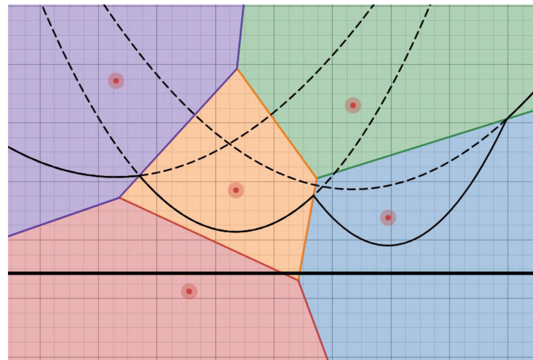

Figure 8: Constructing a Voronyi tessellation using the sweep line of Fortune's algorithm.

Fortune's algorithm [4] suggests using a line that sequentially "sweeps" all points of the diagram (figure 8). If the line passes the partition point, then the **site event** is triggered. During this event, a new arc (parabola) is created, which is a representation of this partition point. The intersection of the arcs gradually grows and outlines a new edge of the Voronyi diagram. If the middle arc at the place of three arcs shortens and compresses to one point of the circle, then the **circle event** is triggered. At this moment, the middle arc is removed and the vertex of the diagram is formed. Since the number of events processed by the algorithm is n, and the complexity of operations with data structures (priority queue of events and arc tree) of each event takes $O(\log n)$ time, then this algorithm has a time complexity $O(n \log n)$, where n is the number of partition points. Due to the storage of the previously mentioned data structures, the space complexity of the algorithm is $O(n)$.

Using various distance metrics, it is possible to achieve versatile partitions of the Voronyi tessellation. The Manhattan (figure 9) or Euclidean (figure 10) distance are the most common metrics, but they lead to straight edges of the diagram. An interesting approach is to use the **Minkowski metric** [5], which will make the edges curvilinear. To calculate the distance between points $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_n)$ the formula can be used:

$$dist(A, B, p) = \left(\sum_{i=1}^{n} |a_i - b_i|^p \right)^{\frac{1}{p}} \tag{2}$$

If $p = 1$, this is the Manhattan distance, $p = 2$ is the Euclidean distance. When $p > 2$ (but not too big), the edges of the Voronyi diagram become curvilinear (figure 11). This can be used, for example, to generate biomes – different areas in the game world, which are defined by climate, terrain, vegetation, fauna, etc. It is possible to determine the belonging of each group of Voronyi cells to a specific biome

[2] More popular name 'Voronoi diagram' is replaced with ukrainian correct transliteration option. Also names 'Voronyi tessellation', 'Voronyi decomposition' or 'Dirichlet tessellation' are commonly used.

using algorithms to combine the cells in a shared entity. Muzzin [6] describes generating biomes for a two-dimensional strategy game with rectangular or hexagonal cells in his paper “How to Use Voronoi Cells for Strategy Game Maps”. Another approach to generate biomes with indirect edges is to use gradient noise.

Since the Voronyi diagram generation algorithm is used only to distribute the game space, it is not universal for various world generation and does not provide a wide range of input parameters:

1. The number of partition points;
2. Algorithm for random placement of the partition points (for example, each point in a separate square cell);
3. Parameter p according to the Minkowski metric.

To summarize the Voronyi diagram’s usability, it can be helpful for developing the real-time strategies (RTS), simulation, survival, and exploration games.

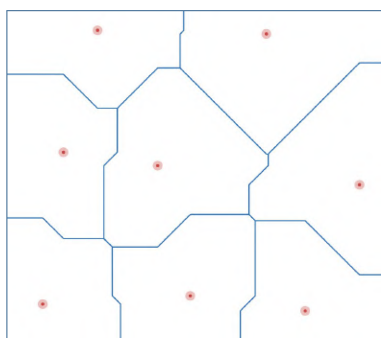

Figure 9: Voronyi diagram using manhattan distance, $p = 1$.

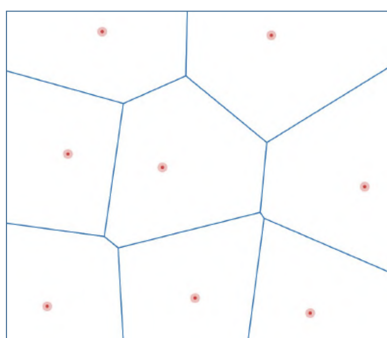

Figure 10: Voronyi diagram using euclidean distance, $p = 2$.

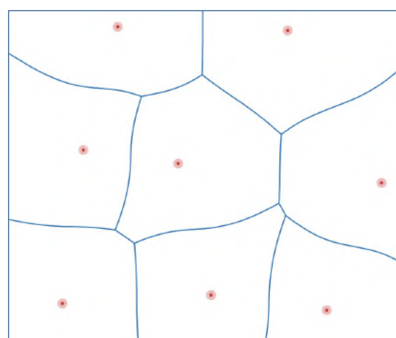

Figure 11: Voronyi diagram using Minkowski distance of $p = 3$.

3.2. Gradient noises

The idea of **gradient noises** is to create a field divided into cells and distribute randomly specified gradients, where the value in the intermediate cells is found by interpolation of dot products of cells.

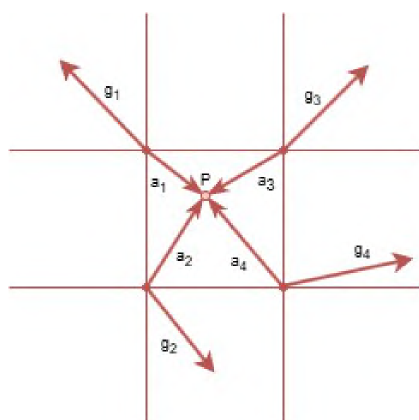

Figure 12: Graphical representation of Perlin noise calculation at point P.

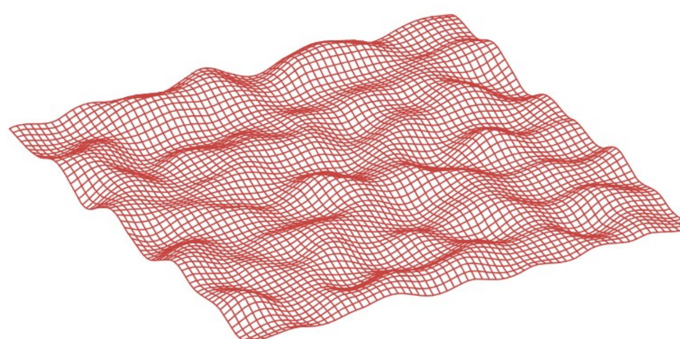

Figure 13: Graphical interpretation of Perlin noise as height maps, where 0 is the lowest height, 1 is the highest height.

Perlin noise [7, 8, 9] is one of the simplest gradient noises. Firstly, the entire space is divided into a grid, each vertex of which is represented of a random gradient $\{\overrightarrow{g_1}, \overrightarrow{g_2}, \ldots, \overrightarrow{g_{2^n}}\}$, where n is the number of dimensions. Then for every single point P we calculate the vectors from four nearest vertices of the grid to the point P: $\{\overrightarrow{a_1}, \overrightarrow{a_2}, \ldots, \overrightarrow{a_{2^n}}\}$. Next step is finding dot products of $\{\overrightarrow{g_1}, \overrightarrow{g_2}, \ldots, \overrightarrow{g_{2^n}}\}$ and $\{\overrightarrow{a_1}, \overrightarrow{a_2}, \ldots, \overrightarrow{a_{2^n}}\}$ and these values are interpolated. For example, considering a two-dimensional space, this process consists of interpolating the upper left and the lower left products $v_1 = lerp(\overrightarrow{a_1} * \overrightarrow{g_1}, \overrightarrow{a_2} * \overrightarrow{g_2})$

and the upper right and the lower left products $v_2 = lerp(\overrightarrow{a_3} * \overrightarrow{g_3}, \overrightarrow{a_4} * \overrightarrow{g_4})$, and after that finding the interpolation of v_1 and v_2 (figure 12). Therefore, we obtain a value from 0 to 1, which can be graphically interpreted at point P and all other points (figure 13).

The time complexity of Perlin noise depends on the number of points n_p and dimensions n. We have already seen that if $n = 2$, then the number of operations for one point is 4. Therefore, the time complexity of Perlin noise is $O(n_p * 2^n)$.

Fractional Brownian noise (FBN) [9] extends the idea of classical Brownian motion, which describes the random movement of a particle. In general, FBN works by layering ordinary noise (e.g. Perlin noise) with varying amplitude and frequency parameters. One such layer is called an **octave** of Brownian noise. The amplitude controls the pitch of the noise, so with each next octave, this coefficient decreases so that each layer becomes smoother and less contrasting. The frequency increases with the transition to the next octave and adds rough small details. FBN has the fractal property of self-similarity: a change in scale partially repeats the original appearance of space.

Gradient noise is often used to define a height map of open world. The game Minecraft uses such algorithms for world procedural generation [10]. To do this, congruent generators are firstly used to create Perlin noise, which serve as octaves of the FBN. After that, the map is processed in a multilayer stack of operators similar to cellular automaton to define biomes (figure 14). Each layer takes the biome map from the previous one, adds some details and passes it to the next one. Also, each biome has the parameters of the average depth and its average deviation (figure 15). Minecraft procedural generation is a good example of open world PCG, developed over the years.

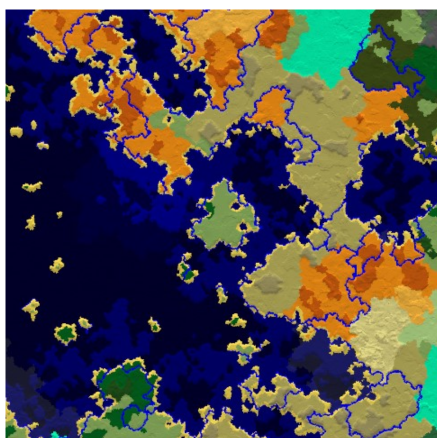

Figure 14: Result of generation the Minecraft world, top view.

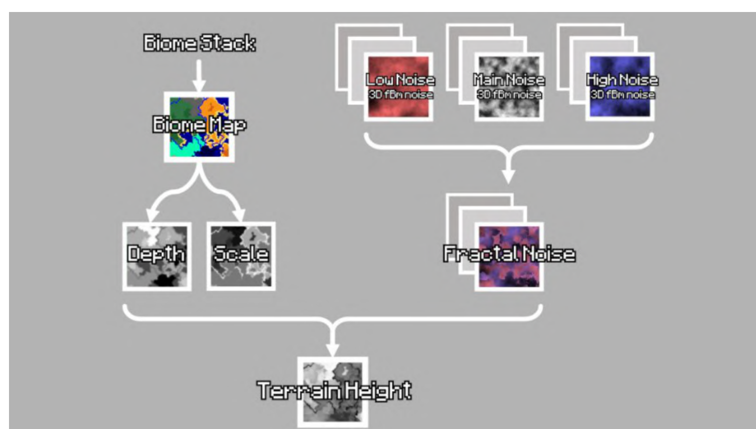

Figure 15: Diagram of terrain height map generation in Minecraft.

As it is seen, gradient noises are frequently used in 3D adventure, exploration, or survival games but generating randomized and smooth 2D terrains and obstacles for platformer games is also a common way of its exploitation.

4. Simulative algorithms

Simulative algorithms generate the world based on the reproduction of the change of a real object over time. In such algorithms, time is expressed in the number of iterations and plays a key role in the shape and distribution of the game space.

4.1. Cellular automaton

In a **cellular automaton** [11], the game space is divided into cells with their coordinates – $x_{i,j}$ for two-dimensional space. Each cell is defined by one of at least two states: {path, wall}. In the initial step, the state of each cell is randomly selected, after which an iterative process begins: the state of the

cell of the next generation $x_{i,j}^{t+1}$ is determined by the state of this $x_{i,j}^{t}$ and the neighbor $N(x_{i,j}^{t})$ cells in this generation. This can be written as

$$x_{i,j}^{t+1} = f\left(x_{i,j}^{t}, N\left(x_{i,j}^{t}\right)\right), \quad x_{i,j} \in \{path, wall\} \tag{3}$$

There are two commonly used approaches to define neighbor cells: the von Neumann neighborhood and the Moore neighborhood. The **von Neumann neighborhood** (figure 16) defines neighbor cells as all those that have a common side with the current cell or the distance to which according to the Manhattan metric is equal to one (for the range of one, $r = 1$). The number of cells depends on the range r and can be calculated as $r^2 + (r+1)^2$. In contrast, the **Moore neighborhood** (figure 17) defines that neighbor cells are those to which the Euclidean distance is equal to one (for the range of one, $r = 1$). Similarly, the number of neighbor cells is $(2r+1)^2$ depending on the range r.

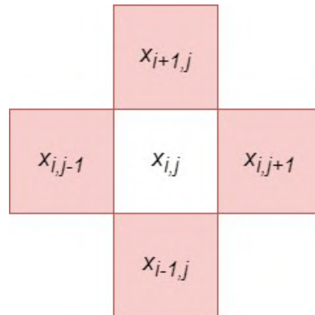

Figure 16: Red cells are neighbors according to the von Neumann neighborhood of $r = 1$.

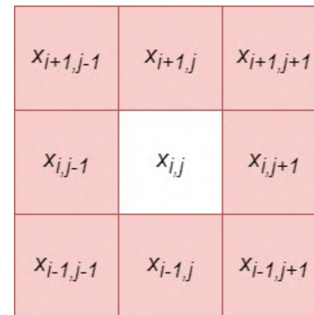

Figure 17: Red cells are neighbors according to the Moore neighborhood of $r = 1$.

From this we can conclude that regardless of which neighborhood is used, the time complexity of one iteration of a cellular automaton in a two-dimensional space is $O(nr^2)$, where n is the number of cells, r is the range of neighbor cells. Since in the last iteration the whole space is remembered to form a game space of the next iteration, the space complexity of a cellular automaton is $O(n)$.

Cellular automaton approaches are widely used to generate open worlds and simulate the change in the position of fluids over time [11] (figure 18). In Noita, the environment responds dynamically to player actions. For example, if a player casts a fire spell, it can ignite nearby flammable objects, spread through adjacent combustible pixels, and potentially create chain reactions. Cellular automaton rules dictate how these effects propagate, often resulting in unpredictable and emergent behaviors that enhance the game's chaos and complexity.

Figure 18: A computer game Noita [12] that uses cellular automaton algorithms.

It is worth noting that generation a world with such algorithms does not necessarily guarantee the presence of a single connected path, but when given a well-chosen cell transition function from one

state to another, there is a growing tendency for paths and walls to be less distributed, which makes the game space more clustered. In addition, there are relatively few parameters that can be changed to control the generation process:

1. A percentage of inaccessible space (walls);
2. The number of iterations (generations);
3. Definition of neighbor cells;
4. The function of transition from one state to another.

Therefore, cellular automata do not provide sufficient control over world PCG, leading to a long process of trial and error to ensure specific gameplay features. However, they are perfect for procedural maze or dungeon generation in roguelike, survival, and puzzle games.

4.2. Genetic algorithms

Evolutionary **genetic algorithms** [13] are used to find the optimal solution to optimization problems. Each solution of such problem is expressed in the form of information encoded in a string – a **gene (chromosome)**. The set of all possible solutions is a population. The quality of a chromosome is calculated using a **fitness function**, which takes into account values that are important for the population. The transition to the next population occurs using the operations of crossing, mutation and selection, which leads to the fact that the average value of the fitness function gradually increases in the next iteration.

When interpreting this approach to world PCG, the ideas of generative grammar are used: a generator graph (figure 19), which determines the generation dependency of certain elements on each other, and a derivation tree (figure 20), which describes an individual in the current population. Each vertex in such a tree is a gene. A population is all possible generated worlds with specified dependencies.

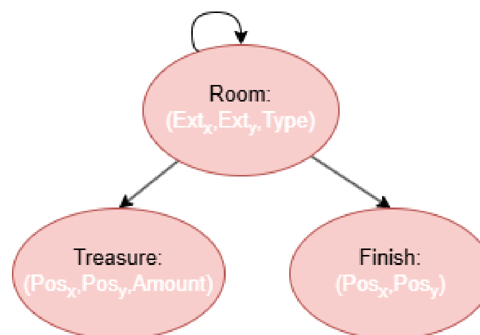

Figure 19: The generator graph with its parameters, consisting of a room whose position is either independent or depends on the position of other rooms, treasures and the finish line, which depend on the position of the rooms.

Mutation, which changes an individual independently of the others, and crossover, which relies on two “parents” to create an individual, are used to obtain new members of the population. These operations are applied with a certain probability: mutation only, crossover only, or both mutation and crossover.

There are three types of **mutation**: grow, cut and alter. Grow adds a random gene to a gene that has not reached the maximum number of children. Cut removes a random gene and the entire subtree that corresponds to it. Alter is characterized by rewriting the parameters of a particular gene to random ones.

During **crossover**, a random gene (along with the entire subtree) from one parent and a compatible gene from the other parent are selected. After exchanging the subtrees of these individuals, two new representatives of the population are formed (figure 22).

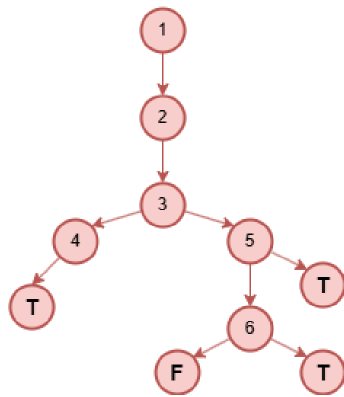

Figure 20: The derivation tree consisting of rooms (numbers), treasures (T), and a finish line (F).

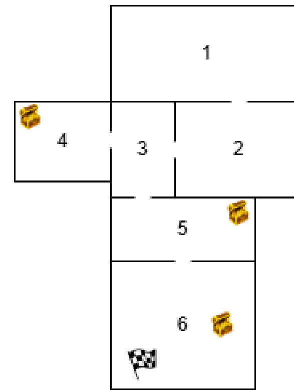

Figure 21: A model of possible game space for the derivation tree in figure 20.

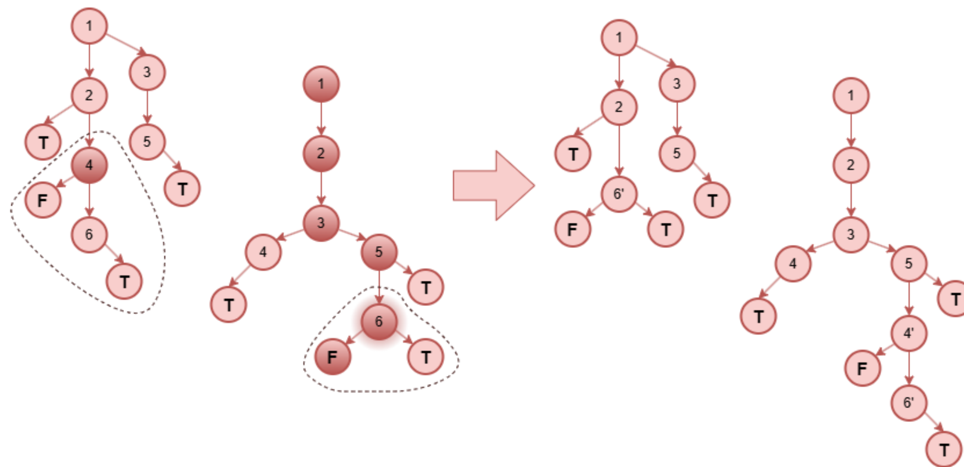

Figure 22: The derivation trees with tagged genes for crossover (left), the result of crossover (right). Numbers are rooms, T is treasure, F is a finish line. Node 4 was randomly chosen for crossover from the first tree and nodes 1, 2, 3, 5, 6, F are compatible to be switched from the second tree. Node 6 is randomly chosen from the second tree.

The selection of the next population can be performed with one of several methods to randomly choose the best individuals. Using roulette wheel selection algorithm, we choose an individual I with a probability $p_I = \frac{f_I}{\sum\limits_j f_j}$, where f_I is the value of the fitness function for this individual I and $\sum\limits_j f_j$ is the sum of all individuals fitness function values of current generation. The k-tournament selection method randomly selects k individuals from the population and keeps only those with the best fitness function values. In practice, this algorithm is implemented more often than the roulette wheel selection, since it lacks stochastic noise.

After a genetic algorithm processed, the graph should be virtually represented (figure 21). The graph representation algorithm can differ depending on game space type (e.g., building or open world) and they also use similar to a generator graph structure.

The time complexity estimation of genetic algorithms is often difficult because the overall complexity depends on the complexity of the genetic operations [14]. In general case, we can estimate it as $O(G * M * N * O(f) * O(s) * (O(co) + O(m)))$, where G is the number of generations, M is the number of an individual's genes, N is the size of a population, $O(f)$ is the time complexity of the fitness function, $O(co)$, $O(m)$ and $O(s)$ are the time complexities of crossover, mutation and selection operations respectively. If we assume that genetic operations have constant execution time, then the complexity of the genetic algorithm reduces to $O(GMN)$. It should be noted that the genetic algorithm

is a general approach. For example, sequential Monte Carlo methods and their modifications are used for many problems.

Genetic algorithms provide a wide range of parameters for controlling a generation process. If we consider the problem of generating sequentially connected rooms and rely on the given generator graph in figure 19, then the following parameters can be specified:

1. The area of the game space;
2. The number of rooms;
3. The size of rooms;
4. The number of treasures;
5. Rewards from treasures;
6. The position of treasures;
7. The position of the finish;
8. The distance from the start to the finish.

The number of parameters will grow rapidly as the complexity of the game mechanics increases. Therefore, a genetic algorithm is a universal approach that provides the ability to clearly specify the features for generation. Genetic algorithms are quite generic but the most popular its use cases cover simulation, strategy, combat (tower defense) games.

4.3. Physics simulation

If a generated world is abstractly represented by a graph, then **physics simulation** can be used to transition to an actual physical representation of the game map. Usually, this approach is helpful in action, adventure, and survival games, where player can interact with environment and cause world terrain changes.

By writing each vertex of the graph into a collider that describes its geometric volumes (a circle in 2D), you can simulate the use of a certain force on this graph with randomly given parameters, which will lead to different isomorphic options of generating one world (figure 23). Each vertex is completely incapable of deformation rigid body, but can be characterized, for example, by mass and velocity, which will allow to use momentum between bodies. For edges, a stiffness coefficient is specified, which allows to describe how much the final distance between vertices can differ from the initial one.

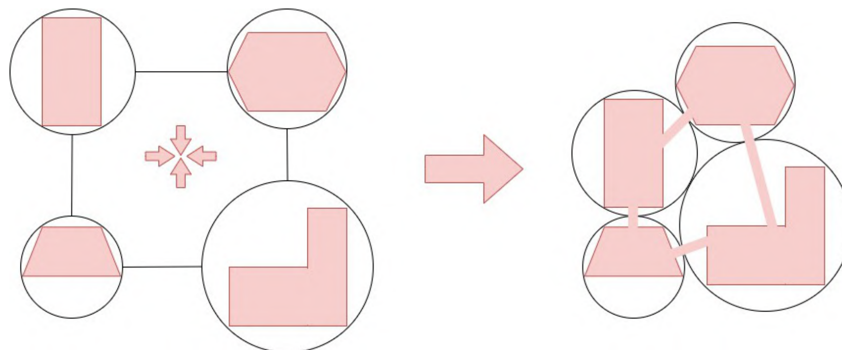

Figure 23: Example of graph representation using surface tension force simulation. Paths between rooms are generated at the contact points of the circles after the force has been applied.

5. Conclusions

Considered algorithms and approaches are computationally complex, since their growth varies from linearithmic to exponential. This is due to the fact that today's computers are quite powerful, so developers neglect optimization measures for such algorithms in order to ensure more correct and predictable world generation. Table 1 shows the generalized results of the analysis, which do not

compare the algorithms with each other. These methods are not interchangeable: usually a combination of different algorithms and approaches is used, as discussed in this article.

Table 1
General characteristics of world PCG algorithms and approaches.

World PCG method	Usage	Approaches/modifications	The time complexity estimation
Graph grammar	Generation of cities, castles, trees, dungeons	The single/double pushout, Paul Merrell's modification	$O(n * V_G * V_L)$ in general, $O(n * V_G^{V_L})$ in worst case: n rewriting rules, V_G – the number of G vertices, V_L – the number of L vertices
Voronyi diagram	Generation of rooms, dungeons, buildings, biomes	Manhattan metric, Euclidean metric, Minkowski metric	Iterating through all points: $O(n^2)$, Fortune's algorithm: $O(n \log n)$, n – the number of partition points
Perlin noise	Generation of height maps, biomes	Octaves of fractional Brownian noise	$O(n_p * 2^n)$, n_p – the number of points, n – the number of dimensions
Cellular automaton	Generation of dungeons, liquid movement simulation	Moore/von Neuman neighborhood	One iteration in 2D – $O(nr^2)$: n – the number of cells, r – neighborhood range
Genetic algorithms	Generation of buildings, dungeons, worlds tied to game logic	Fitness function, algorithms of crossover, mutation and selection operations	$O(GMN)$: G – the number of generations, M – the number of an individual's genes, N – the size of a population.

Declaration on Generative AI: The authors have not employed any Generative AI tools.

References

[1] J. McCollum, Generative grammars as a form of procedural content generation, The Shaggy Dev, 2022. URL: https://shaggydev.com/2022/03/16/generative-grammars/.

[2] J. McCollum, An introduction to graph rewriting for procedural content generation, The Shaggy Dev, 2022. URL: https://shaggydev.com/2022/11/20/graph-rewriting/.

[3] P. Merrell, Example-Based Procedural Modeling Using Graph Grammars, ACM Transactions on Graphics 42 (2023) 1–16. doi:10.1145/3592119.

[4] D. Mount, CMSC 754: Lecture 11 Voronoi Diagrams and Fortune's Algorithm, University of Maryland, 2020. URL: https://www.cs.umd.edu/class/spring2020/cmsc754/Lects/lect11-vor.pdf.

[5] V. Chugani, Minkowski Distance: A Comprehensive Guide, DataCamp, 2024. URL: https://www.datacamp.com/tutorial/minkowski-distance.

[6] J. F. Muzzin, How To Use Voronoi Cells for Strategy Game Maps, Medium, 2023. URL: https://medium.com/@jaemuzzin/how-to-use-voronoi-cells-for-strategy-game-maps-1deaee9a4b34.

[7] R. MacWha, Generating Digital Worlds Using Perlin Noise, Medium, 2021. URL: https://medium.com/nerd-for-tech/generating-digital-worlds-using-perlin-noise-5d11237c29e9.

[8] Suboptimal Engineer, What is Perlin Noise?, YouTube, 2023. URL: https://www.youtube.com/watch?v=7fd331zsie0.

[9] E. Dalefield, Distributed Architecture for Procedural Terrain Generation in Video Games, Master's thesis, Victoria university of Wellington, Wellington, 2024. URL: https://openaccess.wgtn.ac.nz/articles/thesis/Distributed_Architecture_for_Procedural_Terrain_Generation_in_Video_Games/25658514?file=45770220.

[10] A. Zucconi, The World Generation of Minecraft, 2022. URL: https://www.alanzucconi.com/2022/06/05/minecraft-world-generation/.

[11] T. Pagáč, Simulating Game Worlds Using Cellular Automata, Master's thesis, Masaryk University, Brno, 2022. URL: https://is.muni.cz/th/w12aw/SimulatingWorldsCA.pdf.

[12] P. Purho, Noita: a Game Based on Falling Sand Simulation, 2019. URL: https://80.lv/articles/noita-a-game-based-on-falling-sand-simulation/.

[13] V. Kraner, I. Fister, L. Brezočnik, Procedural content generation of custom tower defense game using genetic algorithms, in: I. Karabegović (Ed.), New Technologies, Development and Application IV, volume 233 of *Lecture Notes in Networks and Systems*, Springer International Publishing, Cham, 2021, pp. 493–503. doi:10.1007/978-3-030-75275-0_54.

[14] Y. Pyrih, Computational complexity evaluation of a genetic algorithm, Information and Communication Technologies, Electronic Engineering 4 (2024) 52–60. URL: https://doi.org/10.23939/ictee2024.01.052.