Strategic Features and Terrain Generation for Balanced Heroes of Might and Magic III Maps

Jakub Kowalski, Radosław Miernik, Piotr Pytlik, Maciej Pawlikowski, Krzysztof Piecuch, Jakub Sękowski
Institute of Computer Science, *University of Wrocław*
Wrocław, Poland
jko@cs.uni.wroc.pl, radekmie@gmail.com, piotrcpytlik@gmail.com,
pawlikowski.maciek@gmail.com, kpiecuch@cs.uni.wroc.pl, sequba@gmail.com

***Abstract*—This paper presents an initial approach to a generic algorithm for constructing balanced multiplayer maps for strategy games. It focuses on the placement of so-called *strategic features* – map objects that have a crucial impact on gameplay, usually providing benefits for the players who control them.**

The algorithm begins with constructing a logical layout of the map from the perspective of a single player. We use a novel approach based on graph grammars, where rules do not add new features but are initially constrained by the content of the start node, which simplifies their construction. To introduce other players' space, the single-player graph is multiplied and partially merged. The result is projected onto a grid by combining Sammon mapping, Bresenham's algorithm, and Voronoi diagrams. Finally, strategic features are arranged on the map using an evolutionary search algorithm.

We implemented the proposed method in a map generator which we are developing for the popular strategy game Heroes of Might and Magic III. We also point out how this approach can be adapted to other games, e.g. StarCraft, WarCraft, or Anno.

***Index Terms*—procedural content generation, map generation, graph grammars, heroes of might and magic**

I. Introduction

Procedural content generation (PCG) has been used in games since the introduction of home computers. Examples include Beneath Apple Manor from 1978, or released in 1980 Rogue. However, the progress of PCG techniques did not go hand in hand with the progress in other areas of game development. Nowadays, most popular commercial video games, especially AAA titles, either do not use procedural generation or use relatively simple constructive generative techniques [1], which are not especially interesting from the research point of view.

World generation in Civilization VI [2] is based mainly on Perlin noise and simple heuristics for objects placement. Dungeons in Diablo III [3] consist of handcrafted tiles that are combined together using a simple generator, and later populated with randomized treasures and monsters. The levels in Spelunky [4] always contain 16 rooms whose layouts are selected from a set of predefined templates. To increase visual diversity, random obstacles are placed according to the templates' indications. Lastly, monsters are put in spots determined by heuristic rules.

On the other hand, academia-rooted PCG research often focuses on more advanced techniques, such as evolutionary algorithms, simulation-based evaluations, or neural networks [5]. However, their implementations usually have a purely scientific character, as they are applied mostly to test-bed games and tools [6], [7], [8], [9], [10], [11], [12], [13], indie games [14], or reimplementations of older titles [15], [16], [17]. There are, of course, notable exceptions, including level generation for Angry Birds [18], Cube 2 [19], [20], or StarCraft [21]. Some academic approaches even turned out into commercial (usually independently published) games, e.g. Galactic Arms Race [22], Petalz [23], or Darwin's Demons [24]. Nevertheless, it is often a difficult and laborious task to meet all of the requirements and generate maps that are playable in an already existing, commercially published game.

Our ambition is to develop a map generator for Heroes of Might and Magic III (HoMM3) [25] which could be widely used by the game's community. This paper describes the first step towards this goal. We are presenting an algorithm that generates a balanced map layout, projects it onto a grid, and ensures a fair placement of the most important map features, mainly towns and mines.

One of the main reasons for our work is, as we believe, a community need for such a tool. Although the game is nearly 20 years old, it is considered the best part of the entire Heroes of Might and Magic series and is still widely played. There are multiple fan-made mods and extensions, including Horn of the Abyss, In the Wake of Gods, and VCMI – an attempt to entirely rewrite the original HoMM3 engine. Additionally, in 2015, Ubisoft released the HD Edition of the game ($642,075 \pm 24,614$ sold copies according to the Steam Spy website[1]). Sadly, this version does not include expansions and lacks the random map generator.

Existing template-based generators usually require a skilled user to modify generator's behavior. Instead, we aim to propose a method that will be able to produce reliable maps based on simple user preferences. We provide necessary background about map generation approaches in the next section. Our algorithm is fully described in Section III. Section IV contains details of our implementation, which is tailored for Heroes of Might and Magic III and yields playable maps. We discuss the results produced by our algorithm, present the arguments for

[1]https://steamspy.com/app/297000, retrieved March, 2018

its generality, and give a perspective of the future research in Section V. Finally, we conclude in Section VI.

II. Related Work

A. Procedural Map Generation in Games

In this section, we provide a brief overview of the PCG research and algorithms relevant to this study. For the comprehensive survey on procedural generation and its role in games, we refer to textbooks [5], [26].

Graph grammars ([27], [28]) allow to combine a powerful and intuitive concept of grammatical rules with a capacious graph representation. This approach is often used for creating non-linear story-driven levels, where the level structure strongly follows the generated plot. In other words, all possible paths towards the objectives, along with the events that can be encountered along the way, are constructed as a randomly generated graph.

In general, application of graph grammar rules is a complicated task, as the left-hand part of a rule has to match a proper subgraph, which then has to be modified in-place according to its right-hand side. To make generated structures more predictable, additional constraints are usually placed upon the maximum number of application of certain rules or resources that those rules can place. This causes the entire architecture to become significantly more complex.

Graph grammars are especially convenient for generating RPG-style one-player maps. A good example is a mission graphs generator for Dwarf Quest [14]. It is based on evolutionary search, where grammatical rules are applied as mutation operators. Evolved mission graphs are converted into Dwarf Quest levels consisting of pre-made rooms mapped to nodes using so-called layout solver. Another approach, presented in [12], generates Zelda-like levels via two-step evolutionary algorithm: the first phase generates an acyclic high-level graph of areas, while the second evolves, for each area, a low-level cyclic graph consisting of rooms and doors.

An important issue is mapping the abstract graph structure into concrete locations while preserving its characteristics. In [29], the problem has been restrained to mapping trees into a low-resolution grid (one node per square) and solved via a recursive backtracking algorithm. If no mapping solution has been found, the tree is discarded and regenerated.

An approach of generating RTS-style maps via multiobjective evaluation has been presented in [6]. The method had since been extended and used to generate playable maps for StarCraft, however, it commonly produces maps “not looking very StarCraft-like” [21]. The genotype consists of player base locations, resource (gas and minerals) locations, and information about impassable areas. The algorithm tries to optimize maps mainly for their fairness (various resource-related metrics like distance, ownership estimation, safety) and strategic aspects (metrics related to choke points and possible unit paths). Recently, a new idea of using neural networks to predict the placement of StarCraft map features based on the existing map data has been proposed and tested in [30].

Lastly, we want to raise the subject of using cellular automata (CA) as a map-filling algorithm. It is often needed to populate an area with obstacles (water, impassable rocks, etc.) in a way that is random, fast, and still somewhat controllable. In [31], cellular automata have been utilized to generate surprisingly lifelike cave-rooms. In particular, the authors perform an exhaustive study on the influence of the CA parameters on the style of the obtained caves. In the genetic algorithm for generating Dune 2-like maps [17], an interesting approach to cellular automata-based genotype-to-phenotype mapping is used. Instead of direct map representation, the evolutionary process modifies only the CA parameters (e.g. size of a Moore-neighbourhood, activation threshold, number of iterations), which are later used to fill the map with rock, sand, and indirectly other features like player starting zones.

B. HoMM3 Map Generation

A random map generator had been introduced into HoMM3 with the first expansion – Armageddon’s Blade. Despite many faults it was, and still is, commonly used (its exclusion from HoMM3 HD Edition caused many complaints from the players). The generator on its default settings tends to produce highly unbalanced maps filled with items that are rarely used by human map-makers (e.g. Pandora’s Boxes with very random rewards), and valuable treasures that often lie unprotected. One of its bugs can even occasionally make a player starting position completely surrounded by obstacles.

However, by overriding generator’s template, it is possible to modify, to some limited extent, the algorithm’s behavior. This allows to handcraft a structure similar to our map layout (see Section III-C), which contains information about the zones’ characteristics and relative positions. Many alternative templates have been made and shared among the community[2].

The improvements of the random map generator are important aspects of HoMM3 mods. Horn of the Abyss provides a graphical template editor offering a wide array of settings[3]. Recently, a new web-based map generator had been published[4]. It is a work-in-progress version and still does not support all map features (e.g. water, underground). It defines its own format of templates, which allows for a very high level of customization and gives a lot of control over the resulting maps.

III. Methodology

Let us start with the definitions. A *zone* represents consistent map area with the same purpose, style, and level of challenge. Zone is defined by its *class*, i.e. (*type*, *level*) pair. The higher the level, the more challenging and rewarding the content of the zone should be. The zones are *local* if they are easier accessible for one player, i.e. it should be safe to explore for that player. The *buffer* zones are the areas separating different players’ local areas. Buffers are equally accessible by at least

[2]http://forum.heroesworld.ru/showpost.php?p=564869&postcount=57, retrieved March, 2018 (in Russian)

[3]https://www.facebook.com/h3hotaofficial/posts/1447460758611590

[4]http://www.frozenspire.com/MapGenerator/Index.html, retrieved March, 2018

two players, so in these zones the multiplayer fighting should take place. The *goal* zone is a special buffer, limited to one per map, existing only in maps with specific winning conditions (capture town, defeat monster, acquire artifact).

The content of zones is represented by *features*. There are strategic features, like *town* and *mine*, or more technical features: *outers* and *teleports*. The *value* of a feature describes its detailed content. We distinguish player's main town, other towns they initially possess, towns dependent on surrounding factions, and two types of random towns (chosen either by our generator or in-game randomizer). There are *base* mines (for wood and ore), *primary* mines (for faction-dependent most important resources), *gold* mines, and *random* mines.

Outer represents a connection with other player's part of the map. Teleport is a special kind of outer: a hyperedge joining all of its occurrences within the map via two-way monoliths. The value of an outer is its level – representing difficulty in the same way as levels of zones and influencing the strength of guarding creatures. The value of a teleport is its level and identifier. Teleports with the same identifier are joined together and there can be at most 4 distinct teleports on the map.

A. Generation Parameters

First, the user needs to specify desired map characteristics. Apart from necessary settings like map size, players' specification, or winning condition, we defined eight parameters influencing generator's behavior. All the parameters have values from 1 to 5, where the default 3 means "standard". The most important for the algorithm are:

- *welfare* – higher values mean more resources and mines,
- *towns* – higher values mean more towns placed,
- *branching* – higher values mean more connections between zones,
- *focus* – lower values mean more player-vs-player map, while higher values focus on player-vs-environment,
- *zonesize* – higher values mean bigger zone areas (which is equivalent to lesser number of zones placed).

Knowing the map specification, we start by defining the content of the map without knowledge about its layout. We do it from the perspective of one player, i.e. we enumerate the zones he will encounter without going into other players' territory. The generator uses a set of parameterized, randomized rules. For instance: number of zones depends on the map size and zonesize parameter, maximum zone level depends on the map size and overall estimated difficulty of the map, and strong player-vs-player focus results in a smaller number of local zones compared to the number of buffers.

Let us start with an example. For M-size, 4-player map generated with default parameters, we can obtain the following zone classes: $(local, 1)$, $(local, 3)$, $(buffer, 2)$, $(buffer, 4)$.

We generate features in a similar way. However, each feature is already associated with a zone class. Although we do not know an exact zone to place a certain feature, we know what class of challenge this feature should belong to. These classes are calculated based on feature values (e.g. gold mines have a higher probability to be placed in high-level zones) and given parameters (low player-vs-player focus forbids putting outers in low-level local zones, winning condition "capture town" requires town in the goal zone, etc.).

Continuing our example, we can obtain a starting town (T_{START}) and two base mines (M_{BASE}) in $(local, 1)$ zone, two random mines (M_{RND}) in $(local, 3)$, a random (T_{NEUT}) town in $(buffer, 4)$, and an outer edge in each buffer. (See Figure 1a.)

B. Logic Map Layout

The *Logic Map Layout* (LML) graph consist of nodes representing zones and edges representing connections between the zones. Each node contains a multiset of zone classes and a multiset of features (with a proper class associated). LML defines a logical structure of the map and is constructed using a novel variant of a graph grammar algorithm.

An LML node is *inconsistent* if it contains a feature associated with a class which is not present in the node. We require our graph to always be consistent. A node is *final* if it is consistent and contains only one zone class.

We initialize our graph structure with one node that consists of all zone classes and features computed from the generation parameters, as shown in Figure 1a. In our algorithm, graph generation process comes down to making all nodes final. Thus, our approach does not require any additional constraints checking, as all we do is a redistribution of the nodes' content.

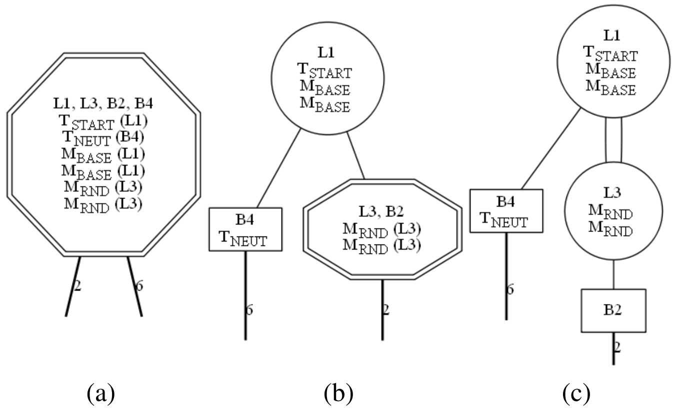

Fig. 1. Constructing LML. Nodes contain classes and features, bolded lines are outer edges with levels. Initial node is presented in Fig. 1a. Fig. 1b contains one of the intermediate steps (after two successful applications of rule (1)). The final LML is shown in Fig. 1c.

Each grammar rule has a weight assigned. In every step, one rule is chosen via the roulette-wheel selection. If its preconditions match, the rule is applied. Otherwise, the graph remains unchanged. Currently, we use four production rules (the weight in parentheses is either a constant or a parameter value):

1) (15) For the first non-final node, divide its content by pushing out a new node containing zones larger than a random pivot.
2) (15) For the first non-final node, if it only contains n zones with the same class, divide its content into n new final nodes and put them at the same depth.

3) (*branching*) Duplicate random edge (if there is only one edge between the nodes).
4) (*branching*) Connect two random, previously not connected, nodes (only local-local or nonlocal-nonlocal).

Although this set can be extended infinitely by adding more and more sophisticated rules, the ones we defined cover most types of reasonable graphs, while remaining relatively simple.

The effectiveness of this grammar comes from a proper ordering of the zone classes. We order zones of the same type by their level, and otherwise we have: local $<$ buffer $<$ goal. This way rule (1) always creates a new node containing a buffer zone if the original node had one. It prevents a local zone from being placed "after" a buffer (counting from the starting position). Rule (1) alone will eventually give us trees where any non-final node contains zones of only one class.

Thus, the rule (2) splits such nodes by making several new nodes all containing only one zone (so they become final). The new nodes have all or only some of the base node's edges (depends on branching parameter). The main issue here is fair features redistribution. We defined heuristic values for every type of feature. Thus, for each feature in the base node, we insert it into the copy that has the lowest value at that moment. The ordering in which the features are considered depends on the zone's type. For local zones, we distribute towns and mines first, while for the other types we prioritize outers and teleports. The role of remaining two rules is to extend otherwise tree-like graphs with cycles and multiedges.

The final LML graph has been presented in Figure 1c. To visualize the process, Figure 1b contains an exemplary middle step, before applying rule (3) to duplicate an edge and rule (1) to the only remaining non-final node.

C. Multiplayer Logic Map Layout

In the next step, we need to compute a layout for the entire map, including all players and all zones. We call the resulting structure *Multiplayer Logic Map Layout* (MLML). To create this graph, we make a copy of LML for each player. Then, we join these duplicated LML's via the outer edges and merge certain buffer zones. In doing so, we want to obtain a graph which is connected and isomorphic from the perspective of every player (i.e. the node containing player's main town).

We say the MLML zones belonging to different players are *corresponding* if they were created as a copy of the same LML zone. We call a newly added edge *valid* when it connects two zones of the same level. First, we attempt to connect all the players' graphs with valid edges between buffer zones, to ensure the final graph will be connected.

After using the buffer zones' outer edges, we add valid edges between local zones, connecting the graph if still needed, and simply distributing them randomly and evenly otherwise. This can be optimized by keeping track of the added edges for each players' corresponding zones and ensuring all players have similar edges. Remembering which corresponding zones had a connection added between them lets us provide a much higher chance for the final graph being isomorphic.

After distributing all outer edges, we merge certain buffer zones, to simplify the graph and allow for a larger buffer zone to replace several corresponding buffer zones. This is done by restricting ourselves to a graph made up of the newly added edges. We search through this graph for sets of corresponding buffer zones, which only have connections between each other.

If such sets exist, each one can be merged into a single buffer zone. While merging such zones, we say the merged zone has a size equal to the sum of sizes of the original buffers, up to a maximum of 3 times the base size (this is an arbitrary limit, which has proven to work well enough during tests). After these merges are finished, we want to verify that the final graph is isomorphic as observed by the players by using a rooted tree isomorphism algorithm[5].

MLML for our example has been presented in Figure 2.

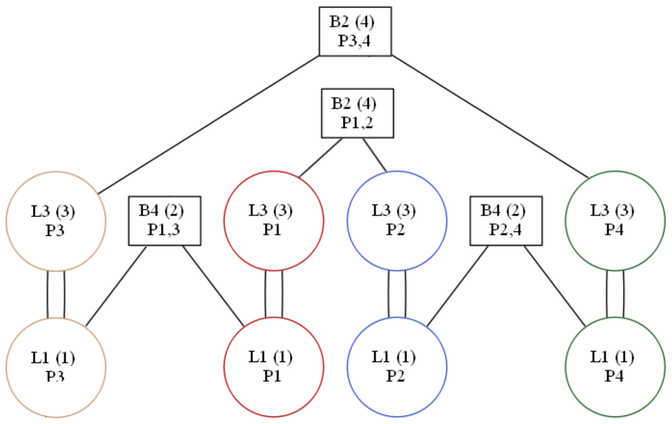

Fig. 2. Generated MLML. Numbers in braces identifies the corresponding LML zones. Each zone contains information about accessing them players.

D. Mapping MLML to Grid

Our task is to project the MLML graph onto a grid, i.e. each vertex has to become a separate zone on a game map. We are bound by three constraints: (a) zones representing connected vertices should be directly accessible from one another, (b) vertex size should represent zone area, and (c) the map should not contain too much unutilized space. We divide this process into two stages. First, we embed the graph points in a planar space. Then, we calculate a modified Voronoi partitioning based on the result. The visualization of each step is presented in Figure 3.

Solving the first constraint algorithmically is not a trivial task. Instead, we chose to pursue a data science approach and employ Sammon mapping [32] for creating a graph embedding. Sammon mapping is a data visualization method deriving from multidimensional scaling. Given a set of points and a matrix d^*_{ij} of relative distances between them, it embeds the points in a low-dimensional space by minimizing a stress function

$$E = \frac{1}{\sum_{i<j} d^*_{ij}} \sum_{i<j} \frac{(d^*_{ij} - d_{ij})^2}{d^*_{ij}},$$

[5]https://groupprops.subwiki.org/wiki/Rooted_tree_isomorphism_problem

where d_{ij} is a distance matrix for the embeddings. Defining a distance between two vertices as a length of the shortest path between them allow us to use this method to draw a graph on a plane [33].

In order to fulfill the (b) requirement, we start with reshaping the graph. We split each vertex of size s into a cycle of s subvertices. Edge (p, q) in the original graph is translated to a random connection between the cycles for p and q. After embedding the obtained graph using Sammon mapping, we still need to ensure the (c) property. Consequently, the output is rotated and cropped. Afterward, we use a gravity-like mechanism to fill sparse regions by pulling the points closer to the map edges.

Note that this method relies on heuristics, and the desired proportions between areas are only roughly maintained. Nevertheless, we found the results satisfying for the task.

At this point, we have a good basis for the partitioning of a map space. To prepare for that, we add a sparse virtual grid above the map grid. This virtual grid covers the map grid and consists of *sectors*, which are rectangles of equal size, each containing a group of map squares.

We call two sectors *direct neighbors* when they are next to each other horizontally or vertically, but not diagonally. For every zone, we want to have a group of sectors, where each one is a direct neighbor of at least one other sector in that zone. This ensures that every zone has a connected set of sectors.

Furthermore, we want every pair of zones which are connected in the MLML graph, to have at least one pair of sectors (one from each zone), which are direct neighbors. This will allow us to later specify that the two zones have an edge between these two sectors.

In our approach, we assign each zone a starting sector, by taking the average of the zone vertices obtained with Sammon mapping and choosing the sector which holds this position.

After assigning each zone a starting sector, we go through the edges from MLML and attempt to connect each pair of zones with a chain of direct neighbors. First, we calculate a path between both starting sectors using a Bresenham algorithm[6]. This ensures that we always have direct neighbours along the way. Next, we traverse the path and try to find a chain of sectors, which start from one of the zones, ends in the other, and has only empty sectors along the way. If such a chain exists we assign the sectors fairly to both zones. So, ultimately, a path starts in one zone, goes along sectors of this zone, and then continues in sectors of the second zone.

Now that we have assigned sectors for all of the zones, we proceed to fill the map grid with a basic Voronoi method. Inside every sector, we generate three random control points with the sector's id. We only allow the points to be generated at a certain distance away from the sector's sides. We assign each grid square the id of the control point which is closest to the center of the square. We only have to take into account the current and neighboring sectors, because further control points can never be closer.

[6]https://www.redblobgames.com/grids/line-drawing.html#stepping

While testing the algorithm, we observed that using too few control points caused neighboring sectors to be separated by another sector. However, when using too many control points, the sector boundaries immediately took the form of a standard grid, without any irregularities. The three random control points allow the Voronoi grid some randomness, while not restricting the sectors to have any specific shape.

Because of the allowed movements in HoMM3, the borders between sectors of different id's can not have diagonal gaps. To decide which grid squares must form a border, we iterate over the squares and compare a candidate to each of its 8 neighbors. When comparing the square with a neighbor, we check if both squares have an id of an assigned sector and if the candidate square has a larger distance to its control point. If it does, it is changed to a wall. Otherwise, it remains unchanged and we continue through the map. The gates between two adjacent zones are placed in zone border tiles such that adjacent squares belong only to one of that zones or are neutral.

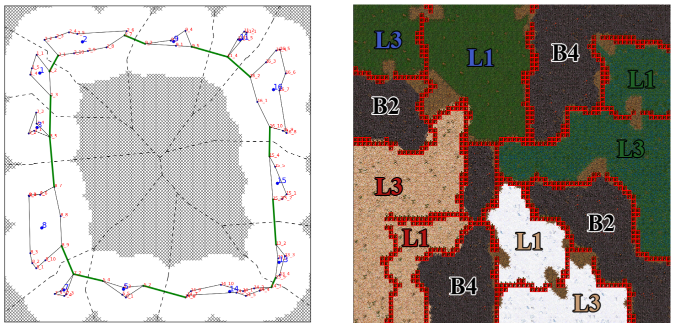

Fig. 3. On the left, result of the Sammon mapping. On the right, final partitioning of a grid, including gates between the appropriate zones. (For this partitioning example, we distorted the graph embedding results to forcefully fill the entire map.)

E. Strategic Features Placement

To guarantee a decent level of balance, we have to place the strategic features such that all players have similar access to the corresponding objects in the corresponding zones. If, for example, one of the players has a mine near their town, while all the others have it on the far end of the zone, this will result in a large difference in their early advancement.

The task can be defined as follows. We are given a set of corresponding zones and, for each of them, coordinates of *entrances*, i.e. tiles where a player can enter the zone. Given a set of strategic features, the following distances should be preserved:

- between corresponding features in corresponding zones (feature-to-feature),
- between corresponding objects and corresponding entrances in corresponding zones (entrance-to-feature),
- if the zone is a buffer, between k-th nearest object for any two entrances in the zone leading from lower-level zones (for k less equal than the number of features).

The first two rules focus on fairness between corresponding zones. The last rule ensures that different players arriving from their entrances to one particular zone will encounter a strategic feature in a similar distance.

We aim to optimize the features placement via an evolutionary algorithm. A genotype contains the exact position of every feature. We start with computing all valid feature spots in a zone and use them to randomly generate an initial population.

To compute the evaluation function, the distances between all pairs of objects are calculated by BFS. For individuals with overlapping features, the fitness is infinity. Otherwise, we try to minimize the sum of squared differences between corresponding distances included in the above list, e.g. we minimize the discrepancy between the distance from the entrance to the mine in zone A and the distance to the corresponding objects in zone B, assuming A and B are corresponding.

For breeding, we choose $\sqrt{n}$ best individuals and perform a uniform crossover on every possible pairing. The mutation operator replaces each position in an offspring genome with a valid random position. We discard all identical individuals and preserve best individual obtained so far (i.e. elitism of size 1). The algorithm stops after a given amount of time or if no better solution has been found in a number of past iterations.

Assuming we have the features placed, we can finally place the roads. We calculate a minimum spanning tree connecting all features and entrances on the zone and set the corresponding tiles as road tiles in this zone and all corresponding ones. The example is shown in Figure 4.

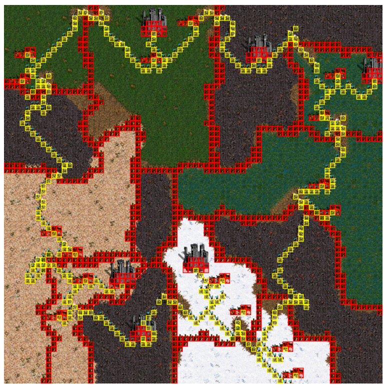
Fig. 4. The map after the placement of strategic features and roads (red tiles are zone borders, yellow tiles are roads)

F. Filling Space with Cellular Automata

It is characteristic for HoMM3 maps to contain irregularly placed obstacles, which allow placing treasures nearby and effectively guarding them. Since usually the exact locations of the obstacles do not heavily influence the map's properties, we can safely use cellular automata to fill the space randomly, and, if required, do some small fixes later.

In the standard case, cellular automata operate on two types of tiles: white and black. In each step, every cell can change its color according to the rules. In our case, we need to fill the interior of the zones without erasing borders separating zones or blocking roads set by the feature placing algorithm. Thus, we added two additional colors: *super-white* (which works as white but cannot be blacked) and *super-black* (which works conversely). This is more general then, e.g. overriding tiles with roads and borders after CA step because we are able to additionally parametrize automata behavior by giving separate weights to these special colors.

IV. Implementation

After all the steps presented in the previous section the main structure of the map is complete. The visualization of the example run we have described is shown in Figure 5.

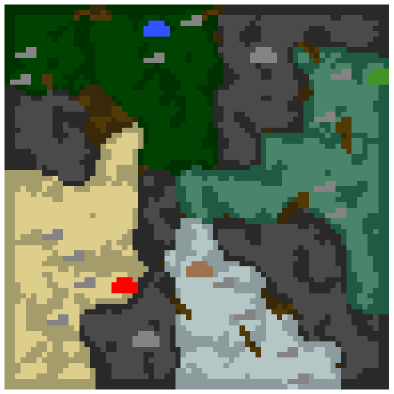
Fig. 5. The in-game minimap of the example map we have generated. This is an M-size map, for 4 players, and with all generation parameters set to default value.

Our map generator is written mainly in Lua, and partially in C++ and Python. We used Löve[7] for GUI. To generate maps in proper format, we used (and slightly fixed) C++ homm3tools library [34], for which we developed Lua API[8].

Graph embedding and visualization was done with NumPy[9], SciPy[10], and Matplotlib[11] Python packages. We used open source implementation of Sammon mapping from Github[12].

Evolutionary algorithm for features placement was run with population of size 100, mutation rate of 0.01, and time limit set to 1 second.

V. Discussion

We proposed a method that, in theory, can generate balanced map layouts for Heroes of Might and Magic III. The algorithm is highly modular, which proved to have both advantages and disadvantages. On one hand, we were able to independently develop and improve individual components. The randomized nature of some of them gives an opportunity to run them several times and pass the best result to the next step. It also means we can obtain varied results using the same parameter set. The partial visualizations of the process would not be possible without splitting the generator into separate phases.

On the other hand, ensuring truly balanced outputs turned out to be very hard. One of the biggest challenges comes from

[7]https://love2d.org
[8]https://github.com/radekmie/homm3lua
[9]http://www.numpy.org/
[10]https://www.scipy.org/
[11]https://matplotlib.org/
[12]https://github.com/tompollard/sammon

the cumulation of errors. Each step heavily depends on getting a reasonable input. If the previous phase fails to produce it, we usually have no way to fix it, since the components are almost completely independent. For example, MLML step does not know if the graph it is making can be nicely drawn on a plane. This sometimes results in scenarios, where Sammon mapping cannot provide a good embedding, and the final map is useless. Introducing some backtracking could help with this issue, but it is going to impossible to completely avoid the problem without some coupling between phases.

We also need to mention that at this point the balancing is only theoretical, and we did not have the opportunity to actually test the maps by playing them. The game HoMM3 is heavily based on moment-to-moment exploration, so getting the full experience requires all of the game-specific, low-level features to be present.

Although we developed our algorithm mainly for the purpose of generating Heroes of Might and Magic maps, it is generic in nature and can be adapted to other productions. The algorithm requires the existence of zones and strategic features, but these concepts occur commonly, and their equivalents can be easily defined in many strategy games.

If we consider real-time strategy game StarCraft [35], [36], the partitioning into zones is not so clear. However, we can still keep main routes as edges between some distinguished map areas and specify their terrain style (level of openness, high ground flag, the existence of chokepoints). We can identify resources (minerals, rich minerals, vespene gas) and controllable Xel'Naga Towers as strategic features. Destructible rocks could be encoded similarly to outers.

Another famous RTS, Warcraft III [37], is even more suitable for our method, mainly because of the number and significance of potential strategic features. This category includes not only gold mines, but also all neutral buildings (taverns, mercenary camps, marketplaces, etc.), creeps (neutral monsters of various strength that give experience and item rewards), and even teleports (called way gates).

As a slightly different example, we will mention Anno 1602 [38], an economic strategy game, usually taking place on a map consisting of multiple islands. Each island naturally maps to a zone, where resources, like gold or iron, are its features. Other strategic features may include island-specific crops, like cocoa and cotton, or the size of an island. The distinction between local and buffer zones is conventional in this case. We can simply assume that local zones form an archipelago of islands closest to the specific player, while buffer zones are islands equally distant from more than one player.

A. Future Work

The goal of a future work is to improve and extend the algorithm, and to finish the remaining parts of our HoMM3 map generator. In particular, we want to develop evaluation functions to estimate output quality after each stage of the procedure. It will let us to use generate-and-test approach, i.e. run each step several times and choose the best outcome.

We also plan to finish the full implementation of all HoMM3-specific features. Water and whirlpools should be implemented as the special type of buffer zones and teleport-like features. Underground map level should be formed by removing some buffer zones before mapping MLML onto a grid, and placing them below adjacent zones, so the subterranean gates can be placed in the overlapping areas. Grail should be placed in a buffer zone close to gameplay-based centers of the map, i.e. areas equally difficult to reach for all players.

We can also consider generating maps that are deliberately imbalanced. The simplest example for HoMM3 is a map containing AI-only players, who should have some handicaps (richer zones, more starting towns, etc.). This can be done by generating two different LML graphs – for human players and for AI-only players. Then, in the MLML phase, these graphs have to be merged in the right way. This is one of the non-trivial extensions we plan to implement and test in the future.

The remaining parts we need to include in our map generator are map aesthetics and low-level features placement. Apart from functional properties of the map, we should also take into account its visual aspects. HoMM3 contains various types of obstacles like trees, lakes, rocks, or mountains, ranging in size from 1×1 to 3×5 map tiles. To make generated map consistent and visually pleasant, the choice of obstacle should depend on its surroundings: terrain type, other obstacles, and strategic features. It is natural that sawmill should be near the trees and crystal cavern is placed in the mountains. Another example is a water wheel, which is a map object that should be placed on a river.

Zones and strategic features determine the outline of the game, but no less significant are low-level features scattered around the map: resources, artifacts, various special objects, and creatures guarding them. Their proper placement is the most important and challenging aspect of a future work, as the entire gameplay can be seen as a sequence of losses and gains. For example, a player loses some troops fighting wandering creatures and then picks up an artifact they were protecting.

In a less complicated domain, the solution could be to use evolutionary algorithms with balance-testing fitness function depending on AI agents simulating the players' behaviors [10], [39], [40]. In our case, to control the loss-gain loop on the map, we need to estimate players' capabilities based on our knowledge about the game mechanics.

It is for this reason that we introduced zone levels, as their semantics is closely correlated with player strength. Let us assume that we want a zone of level four to be attainable roughly on turn 15. To achieve that we need to estimate player's strength at that moment. It is possible knowing their starting town and the content of closer zones of lower levels. Thus, the remaining part is to place a proper creature at the entrance of this zone. We have developed simulation-based unit value estimation program for HoMM3[13], which is able to, for any amount and type of creature, estimate the number of other creatures needed to match its strength. We plan to

[13]https://github.com/maciek16180/h3-fight-sim

use these estimations to ensure the proper level of challenge when placing guarding creatures.

VI. Conclusions

In this paper, we focused on generating a balanced map layouts, with obstacles, partitioning into zones, and fair placement of the strategic features. We combined a variety of known methods including graph grammars, Voronoi diagrams, cellular automata, and evolutionary computation with novel approaches, like feature-redistribution graph grammar algorithm or MDS plus Bresenham-based layout solver.

The proposed algorithm was implemented in a map generator which we are developing for Heroes of Might and Magic III. Although it is not finished yet, it produces fully playable HoMM3 maps in proper `h3m` format. We argue that there is a community need for such a tool, especially because the recently released HD edition of the game does not contain the original map generator.

We presented a step-by-step description and visualization of our method, and discussed the details of its implementation. Lastly, we have shown that the presented algorithm is generic, and can be applied in other strategy games.

Acknowledgments

This work was supported in part by the National Science Centre, Poland, under the project number 2014/13/N/ST6/01817.

References

[1] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-Based Procedural Content Generation: A Taxonomy and Survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 172–186, 2011.

[2] Firaxis Games, *Civilization VI*, 2K Games, 2016.

[3] Blizzard Entertainment, *Diablo*, Blizzard Entertainment, 2012.

[4] D. Yu and A. Hull, *Spelunky*, Mossmouth, 2009.

[5] N. Shaker, J. Togelius, and M. Nelson, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2016.

[6] J. Togelius, M. Preuss, and G. N. Yannakakis, "Towards Multiobjective Procedural Map Generation," in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, ser. PCGames '10, 2010, pp. 3:1–3:8.

[7] A. Liapis, H. P. Martínez, J. Togelius, and G. N. Yannakakis, "Adaptive game level creation through rank-based interactive evolution," in *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, 2013, pp. 1–8.

[8] A. Liapis, G. N. Yannakakis, and J. Togelius, "Towards a Generic Method of Evaluating Game Levels," in *AIIDE*, 2014, pp. 30–36.

[9] P. T. Ølsted, B. Ma, and S. Risi, "Interactive evolution of levels for a competitive multiplayer FPS," in *2015 IEEE Congress on Evolutionary Computation (CEC)*, 2015, pp. 1527–1534.

[10] A. Liapis, C. Holmgård, G. N. Yannakakis, and J. Togelius, "Procedural Personas as Critics for Dungeon Generation," in *Applications of Evolutionary Computation*, ser. LNCS, 2015, vol. 9028, pp. 331–343.

[11] R. A. Agis, A. Cohen, and D. C. Martínez, "Argumentative AI Director Using Defeasible Logic Programming," in *Computer Games*, ser. CCIS, vol. 614, 2016, pp. 96–111.

[12] J. M. Font, R. Izquierdo, D. Manrique, and J. Togelius, "Constrained Level Generation Through Grammar-Based Evolutionary Algorithms," in *Applications of Evolutionary Computation*, ser. LNCS, vol. 9597, 2016, pp. 558–573.

[13] A. Liapis, "Piecemeal Evolution of a First Person Shooter Level," in *Applications of Evolutionary Computation*, ser. LNCS, 2018.

[14] D. Karavolos, A. Liapis, and G. N. Yannakakis, "Evolving Missions to Create Game Spaces," in *IEEE Conference on Computational Intelligence and Games (CIG)*, 2016, pp. 1–8.

[15] N. Shaker, M. Nicolau, G. N. Yannakakis, J. Togelius, and M. O'Neill, "Evolving levels for Super Mario Bros using grammatical evolution," in *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, 2012, pp. 304–311.

[16] S. Dahlskog and J. Togelius, "Procedural Content Generation Using Patterns as Objectives," in *Applications of Evolutionary Computation*, ser. LNCS, vol. 8602, 2014, pp. 325–336.

[17] T. Mahlmann, J. Togelius, and G. N. Yannakakis, "Spicing up map generation," in *European Conference on the Applications of Evolutionary Computation*, 2012, pp. 224–233.

[18] M. Stephenson and J. Renz, "Generating varied, stable and solvable levels for angry birds style physics games," in *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, 2017, pp. 288–295.

[19] L. Cardamone, G. N. Yannakakis, J. Togelius, and P. L. Lanzi, "Evolving Interesting Maps for a First Person Shooter," in *Applications of Evolutionary Computation*, ser. LNCS, vol. 6624, 2011, pp. 63–72.

[20] D. Loiacono and L. Arnaboldi, "Fight or flight: Evolving maps for cube 2 to foster a fleeing behavior," in *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, 2017, pp. 199–206.

[21] J. Togelius, M. Preuss, N. Beume, S. Wessing, J. Hagelbäck, and G. N. Yannakakis, "Multiobjective exploration of the StarCraft map space," in *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, 2010, pp. 265–272.

[22] E. J. Hastings, R. K. Guha, and K. O. Stanley, "Evolving content in the Galactic Arms Race video game," in *IEEE Symposium on Computational Intelligence and Games*, 2009, pp. 241–248.

[23] S. Risi, J. Lehman, D. B. D'ambrosio, R. Hall, and K. O. Stanley, "Combining search-based procedural content generation and social gaming in the petalz video game," in *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2012, pp. 63–68.

[24] T. Soule, S. Heck, T. E. Haynes, N. Wood, and B. D. Robison, "Darwin's Demons: Does Evolution Improve the Game?" in *Applications of Evolutionary Computation*, ser. LNCS, vol. 10199, 2017, pp. 435–451.

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

[26] G. N. Yannakakis and J. Togelius, *Artificial Intelligence and Games*. Springer, 2018.

[27] J. Rekers and A. Schurr, "A graph grammar approach to graphical parsing," in *Proceedings of Symposium on Visual Languages*, 1995, pp. 195–202.

[28] J. Dormans, "Adventures in level design: Generating missions and spaces for action adventure games," in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, ser. PCGames '10, 2010, pp. 1:1–1:8.

[29] K. Hartsook, A. Zook, S. Das, and M. O. Riedl, "Toward supporting stories with procedurally generated game worlds," in *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, 2011, pp. 297–304.

[30] S. Lee, A. Isaksen, C. Holmgård, and J. Togelius, "Predicting Resource Locations in Game Maps Using Deep Convolutional Neural Networks," in *AIIDE*, 2016, pp. 46–52.

[31] L. Johnson, G. Yannakakis, and J. Togelius, "Cellular Automata for Real-time Generation of Infinite Cave Levels," in *Workshop on Procedural Content Generation in Games*, ser. PCGames '10, 2010, pp. 10:1–10:4.

[32] J. W. Sammon, "A Nonlinear Mapping for Data Structure Analysis," *IEEE Trans. Comput.*, vol. 18, no. 5, pp. 401–409, 1969.

[33] E. R. Gansner, Y. Koren, and S. North, "Graph Drawing by Stress Majorization," in *Graph Drawing*, 2005, pp. 239–250.

[34] J. Åkerblom. (2015) homm3tools – Tools for Heroes of Might and Magic III. [Online]. Available: https://github.com/potmdehex/homm3tools

[35] Blizzard Entertainment, *StarCraft*, Blizzard Entertainment, 1998.

[36] ——, *StarCraft II: Wings of Liberty*, Blizzard Entertainment, 2010.

[37] ——, *Warcraft III: Reign of Chaos*, Blizzard Entertainment, 2002.

[38] Max Design, *Anno 1602*, Sunflowers Interactive, 1998.

[39] J. Kowalski and M. Szykuła, "Evolving Chesslike Games Using Relative Algorithm Performance Profiles," in *Applications of Evolutionary Computation*, ser. LNCS, 2016, vol. 9597, pp. 574–589.

[40] F. de Mesentier, S. Lee, J. Togelius, and A. Nealen, "AI as Evaluator: Search Driven Playtesting of Modern Board Games," in *AAAI 2017 Workshop on What's Next for AI in Games*, 2017.