Constrained Level Generation through Grammar-Based Evolutionary Algorithms

Jose M. Font*[1], Roberto Izquierdo[2], Daniel Manrique[2], and Julian Togelius[3]

[1] U-tad, Centro Universitario de Tecnología y Arte Digital, C/Playa de Liencres 2-bis, 28290 Las Rozas, Madrid, Spain
jose.font@u-tad.com,
[2] Departamento de Inteligencia Artificial, Escuela Técnica Superior de Ingenieros Informáticos, Universidad Politécnica de Madrid, Campus de Montegancedo s/n, 28660 Boadilla del Monte, Madrid, Spain
r.iamo@alumnos.upm.es
d.manrique@fi.upm.es
[3] Dept. Computer Science and Engineering, New York University, 2 Metrotech Center, Brooklyn, 11201 New York
julian@togelius.com

Abstract. This paper introduces an evolutionary method for generating levels for adventure games, combining speed, guaranteed solvability of levels and authorial control. For this purpose, a new graph-based two-phase level encoding scheme is developed. This method encodes the structure of the level as well as its contents into two abstraction layers: the higher level defines an abstract representation of the game level and the distribution of its content among different inter-connected game zones. The lower level describes the content of each game zone as a set of graphs containing rooms, doors, monsters, keys and treasure chests. Using this representation, game worlds are encoded as individuals in an evolutionary algorithm and evolved according to an evaluation function meant to approximate the entertainment provided by the game level. The algorithm is implemented into a design tool that can be used by game designers to specify several constraints of the worlds to be generated. This tool could be used to facilitate the design of game levels, for example to make professional-level content production possible for non-experts.

Keywords: procedural content generation, genetic programming, evolutionary computation

1 Introduction

The problem of level generation is that of generating good level content for computer games, where "level" is the spatial content through which one or several player-controlled characters move. Depending on the game type, the level might be a dungeon [4], a map [14], a race track [22], etc. Level generation is an important problem for computer game development, as human authoring of this

type of game content constitutes a very large part of the development costs of modern computer games. Procedural generation of levels can provide large cost savings for game developers [20], and make game development feasible in small teams on limited budgets, but can also make new types of games possible that rely on user-adaptive run time content generation [16]. Additionally, the level generation problem has similarities with many other problems in automated design and creativity (from circuit design to music generation) and in many cases methods developed for one of these problems can be adapted to work on related problems.

A common and successful approach to level generation is to cast it as an optimization problem, so that a space of levels is searched for levels that best satisfy certain criteria; this is called "search-based procedural content generation" [21]. The literature contains numerous studies using evolutionary algorithms for this problem, but also methods based on e.g. Answer Set Programming [18].

While successful methods have been found for many simple domains, the general problem of level generation is hard [19] and for many domains we cannot yet provide effective and efficient level generators. Part of this is because it is hard to automatically judge the quality of a level [8], and the best we can do is provide several imperfect heuristic evaluation functions that might be partly conflicting [11]; at the same time there are a number of hard constraints that need to be taken into account, most obviously that the level needs to be possible to complete. As a result, the optimization algorithm will need find a balance between multiple objectives and constraints. Also contributing the difficulty of the level design problem is that the search space is often vast and disconnected, because levels in many games feature a large number of elements, and many configurations of these are impossible. It is therefore important to devise a level representation that minimizes the dimensionality of the search space while making good levels reachable.

Other desirable qualities of level generators are controllability, the ability of a designer or algorithm to control important characteristics of the results of the generation process, and diversity in the output of the generator. While user control is often addressed through adding objectives that can be tweaked by the generator, diversity can be achieved through modifications to the optimization algorithm [14]. However, such added objectives and modifications often come at the price of a drop in efficiency. One idea for improving search efficiency while guaranteeing diversity is to divide up the generation process in several phases. For example, one can use an optimization algorithm to optimize "templates" for levels, that another algorithm then expands into multiple different levels. This has the benefits that the templates do not need to include all details so the search space can be lower-dimensional, and that each template can give rise to multiple levels which helps diversity. In [7] the authors evolved agent-based systems that could generate levels; however, the evolved level generators could not guarantee any properties of the final levels, and user control was limited and indirect.

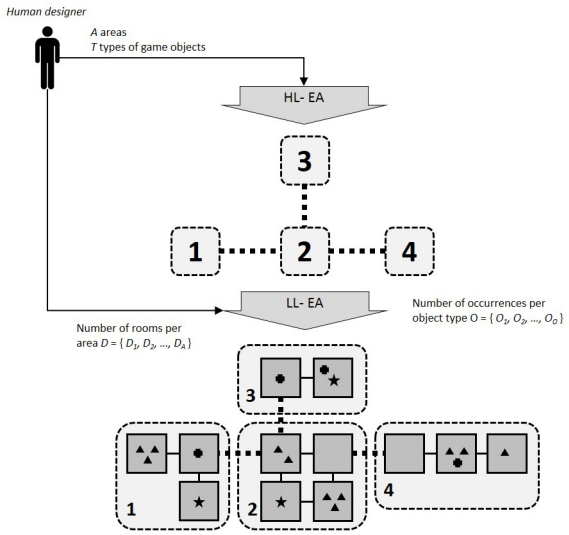

Fig. 1. Overview of the two-step Evolutionary Algorithm. HL-EA generates an acyclic graph of areas, each of those is later evolved by the LL-EA in order to be replaced by a cyclic graph of rooms.

Another approach to level generation is based on grammar expansion, where the general shape of a level is encoded as a grammar, and through variations in how the grammar is expanded variations of a level that still conform to structure decided by the grammar can be produced. The roots of producing content that could be used in games go back at least to L-systems, which were proposed plant generation by Prusinkiewicz and Lindenmayer [15]. Dormans brought grammar-based PCG to game level, devising a method for generating Zelda-like dungeons using grammar expansion, where both dungeon structure and quests were generated together [4]. Others have used grammar-based PCG for generating other kind of game levels, such as van Linden [10], or integrated grammar-based generation into mixed-initiative authoring tools [6].

In the above work, the grammars used for level generation were designed by humans. It is also possible to evolve grammars for content generation, as for example demonstrated by Ochoa who evolved L-systems for plant generation [13]; Shaker et al. used grammatical GP to evolve Super Mario Bros levels [17]. However, these approaches to evolving grammars for content generation do not take the ability of grammars to produce multiple levels from the same design constraints into account.

In this paper, we introduce a two-stage method for creating levels that adhere to strong level design constraints, while allowing for considerable a diversity and designer control and a fast generation time. The core idea is to generate grammars that evolve levels through grammar-guided genetic programming. In a second step, these levels are expanded to yield complete maps that obey con-

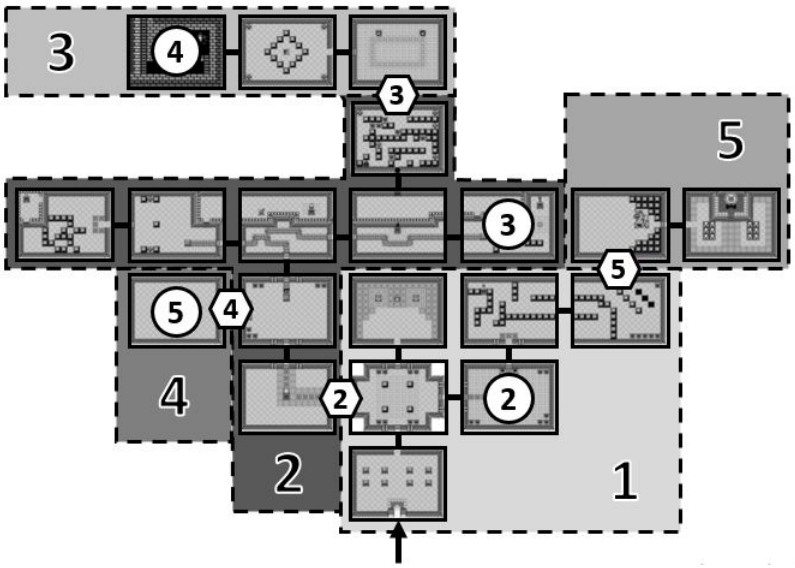

Fig. 2. Gnarled Root Dungeon, from The Legend Of Zelda: Oracle of Seasons [12]

straints while varying in a number of dimensions as permitted by the designer. We demonstrate the method through a system that evolved dungeons for Zelda-like games, complete with lock and key puzzles that pose challenges for other methods.

2 The Evolutionary World Designer

The proposed system is an evolutionary tool that generates worlds for adventure games from scratch. This software displays an intuitive graphic interface that allows setting up and running an evolutionary algorithm (EA). The EA follows a two-step sequence, where two different evolutionary processes run sequentially to define, respectively, the high-level and low-level structures of a game world.

As it is depicted in Figure 1, the high-level evolutionary algorithm (HL-EA) inputs the number of areas set by a human designer, and outputs a numbered acyclic graph of interconnected areas. For each area in this graph, a low-level evolutionary algorithm (LL-EA) runs. Each LL-EA inputs the number of rooms that the human designer set for its related area, as well as the different kinds and number of objects that it must contain. Therefore, each LL-EA outputs a cyclic graph of interconnected rooms, which contains a set of objects that populate the game world (i.e. monsters, chests, keys). The whole set of interconnected areas and their underlying sets of interconnected rooms composes a sufficient structure for codifying a world for an adventure game.

Figure 2 shows the map called Gnarled Root Dungeon, from The Legend of Zelda: Oracle of Seasons [12], represented as a two-level structure. This map will

be used as an explanatory example during the following sections, also showing that the proposed system is capable of coding actual adventure-game maps.

The first level is an acyclic graph whose nodes are the areas (dotted lines) named 1, 2, 3, 4, and 5. Each area contains a low-level cyclic graph whose nodes are rooms. Edges represent doors that connect pairs of rooms. Notice that edges do not imply direction, because doors can be traversed in both ways. Some doors connect a pair of areas as well, named with the number of the forthcoming area (in hexagons). Keys, named after the hexagon door they open (in circles), are required to open those doors, allowing the player to move to the next area. This stands as a lock and key game mechanic typical from adventure games. Lock and key mechanisms serve to connect missions and spaces, translating strong prerequisites in a mission into spatial constructions that enforce the relationships between tasks [1].

Though lock and key mechanisms can adopt several forms, for the purpose of this research they are implemented as actual locks and keys in the game map. The player starts in the first room of area 1 (pointed by an incoming arrow). Areas are named from 1 to 5, indicating the order in which the player must traverse them. This way, the player enters the map in area 1, needing to find the key to area 2 before he can move to that area. This process repeats until the adventure ends in the last room of area 5 (the one at the right), triggering some final event (e.g. facing the final boss).

The presented EAs use context-free grammars (CFG) in order to codify the syntactic rules that produce the languages of valid high and low-level structures (individuals) that comply with the depicted game worlds. The following sections describe the proposed codification system as well as the fitness evaluation functions used by the EAs.

3 Evolutionary Algorithm Encoding Scheme for Adventure Games

The evolutionary system represents a world for adventures games as a high-level acyclic graph whose nodes contain low-level cyclic graphs. These structures are represented as individuals in the populations of the different EAs involved in the searching process by means of context-free grammars (CFG). Thus, these EAs are grammar-guided genetic programs, engineered by Whighams crossover and mutation operators [23, 3]. Each EA operates over a fixed size population, where new individuals are obtained by crossover, mutation and at random during every evolutionary iteration. After evaluating every new individual, the population is replaced by elitism. Evolution stops after a given number of iterations without an improvement on the best fitness score.

The main reason for choosing a CFG driven EA is to prevent the system from generating syntactically non-valid individuals (unfeasible solutions, worlds), without using any kind of repair operator. This operator would raise the overall computational cost by parsing, and possibly fixing, any individual generated during the initialization, crossover, and mutation steps.

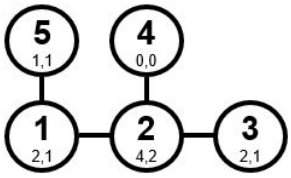

Fig. 3. High-level acyclic graph of the map Gnarled Root Dungeon.

The CFG to be used in each EA depends on the constraints set by the human designer on the systems interface. For this reason, a CFG is deterministically generated ad-hoc from the set of constraints to feed its correspondent EA. The overall process is as follows: A single CFG that becomes the input of the HL-EA is automatically generated from the features set by the human designer. Then, the HL-CFG generates the language of all possible high-level acyclic graphs that match the constraints defined by the designer. Once the optimal (satisfying) high-level graph has been evolved, a different LL-EA starts for each area in the high level graph. To do so, again, a LL-CFG is automatically generated for each LL-EA from the designers parameters to avoid the generation of unfeasible maps. The proposed CFG encoding schemes for cyclic and acyclic graphs is explained in the following subsections. It extends the work presented in [5] about coding acyclic graphs for evolving Bayesian Network architectures.

3.1 High-Level Representation

Figure 3 shows the high-level acyclic graph that codes the areas, their connections (doors), and their amounts game objects in the Gnarled Root Dungeon:

- Each node is named after a single area. It contains the number of the area, as well as the kinds and amount of game objects distributed among the rooms of the area. In this example, there are only two kinds of game objects (treasure chests and monsters). Area 1 contains 2 chests and 1 monster. Area 4 contains neither chests nor monsters.
- The edges show the existing doors between pairs of areas. Notice that, though the final area (5) is directly connected to the first one (1), the player cannot access it until he finds the 5th key located in area 4. This order is strict, and this high-level graph representation does not accept cycles in order to preserve it. Adding a connection between 2 and 5 would create a cycle that allows the player to move directly from 2 to 5, before finding the required 5th key in 4. This would make areas 3 and 4 become useless parts of the map.

In order to encode graphs like this as individuals of a grammar-guided genetic programming system (sentences belonging to the CFG), the proposed encoding scheme includes the meaningful information of the graph as a sequence of natural numbers and separators. The sentence that encodes the high-level graph of the Gnarled Root Dungeon map is 2:1 ; 4:2 ; 2:2:1 ; 2:0:0 ; 1:1:1.

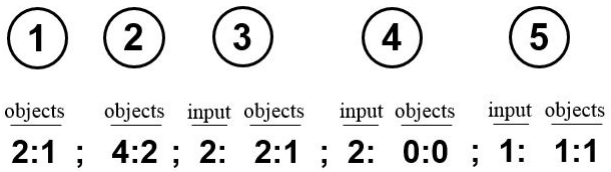

Fig. 4. Example of the proposed encoding scheme for high-level graphs.

Figure 4 explains the meaning of every number in this sentence: it is composed by five sections, one per area (1, 2, 3, 4, 5), separated by ;. The first number of each section points to the area that is connected to this one (input connection), excepting areas 1 and 2. Notice that the label input is missing. In both cases, information about their input connections is not provided because their connections are implicitly represented: area 1 will never have an input connection (because it is the first area in the map), and area 2 will always have an input connection from area 1. Due to the absence of cycles, areas cannot have more than one input connection, though they can have multiple output connections. Each of the remaining numbers indicates the amount of game objects placed in this area.

The HL-CFG that has been automatically generated for this example is able to produce the language of all possible acyclic graphs containing 5 areas and two kinds of game objects. Figure 5 shows the generic template that generates the corresponding HL-CFG given A the number of areas, T the number of kinds of game objects, and D, a list with the number of rooms per area: $D = D_1$, ... , D_A., all set by the human designer. S is the axiom of the grammar, NTS, the set of non-terminal symbols, TS, the set of terminals, and $P_{A,T,D}$ the set of production rules that generate the language. The axiom S produces A sections for each area, separated by ; where inputs and number of objects will be placed. Non-terminal symbols Z_i and C_{ij} produce, per section and respectively, its input connection and its amount of game objects for each kind. Notice that the first two sections produced by the axiom do not contain the symbol Z_i, because their connections are implicitly encoded. Z_i symbols always produce connections to preceding areas, and C_{ij} symbols always produce a number of game objects lower than or equal to the amount of rooms in the area.

3.2 Low-Level Representation

The low-level representation encodes the content of every area in the high-level representation. Thus, every node in the high-level graph leads to a different low-level graph. Figure 6 shows the low-level graph that represents the content in area 1, where nodes and edges encode rooms and (unlocked and bidirectional) doors, respectively. The information coming from node 1 in the high-level graph is the following: area 1 must have connections (doors) to areas 2 and 5, as well as two game objects of the first kind (monsters) and one object of the second

$G_{A,T,D} = (S, NTS, TS, P_{A,T,D})$
$NTS = \{C_{11}, C_{12}, \ldots, C_{1T}, C_{21}, C_{22}, \ldots, C_{2T}, C_{31}, C_{32}, \ldots, C_{3T}, \ldots, C_{AT}, Z_3, \ldots, Z_A\}$
$TS = \{:, ;, 0, 1, \ldots, 9\}$
$P_{A,T,D} = \{$
$S ::= C_{11} : C_{12} : \ldots : C_{1T};$
$C_{21} : C_{22} : \ldots : C_{2T};$
$Z_3 : C_{31} : C_{32} : \ldots : C_{3T};$
$\ldots$
$Z_A : C_{A1} : C_{A2} : \ldots : C_{AT}$
$Z_3 ::= 1|2$
$\ldots$
$Z_A ::= 1|2| \ldots |A-1$
$C_{11} ::= 0|1|D_1$
$\ldots$
$C_{1T} ::= 0|1|D_1$
$\ldots$
$C_{A1} ::= 0|1|D_A$
$\ldots$
$C_{AT} ::= 0|1|D_A$
}

Fig. 5. Generic HL-CFG template for A number of areas, T kinds of game objects, and the list of rooms per area $D = D_1, \ldots, D_A$.

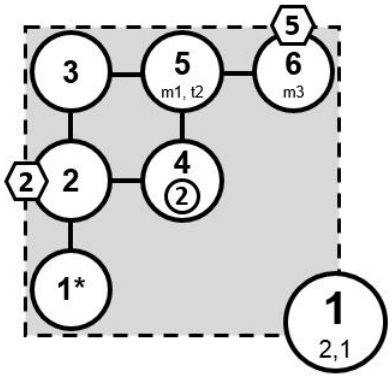

Fig. 6. Low-level cyclic graph for area 1 of the Gnarled Root Dungeon map.

kind (treasures). Area 1 has two additional implications: there must be an initial room, and one room must contain the second key: the key to area 2.

The graph in Figure 6 depicts all this information, being room 1 the entry point to the map (marked with an asterisk), and rooms 2 and 6 those which lead to areas 2 and 5 (respectively). Two monsters can be found in rooms 5 and 6, as well as one treasure chest in room 5. Many subtypes of monsters and treasures can be represented, that is what the numbers in m1, m3, and t2 stand for. The key to area 2 is hold in room 3. In this level, numbers do not imply any kind

of order, and cycles are allowed. Doing this increases the diversity of paths that players can follow during the gameplay. The proposed low-level encoding scheme employed in the LL-EA encodes this cyclic graph as the sentence: 01 : 010 : 0011 : 00001 ; 1 : 2 : 6 : 4 ; 5 : 6 ; m1 : m3 ; 5 ; t2. This sentence contains several sections separated by ;:

- The first section 01 : 010 : 0011 : 00001 is a binary codification of the input connections from room 3 to 6, separated by :. Room 1 will never have any input connections and room 2 only connects backwards to room 1, so this information is implicit. The existence of cycles imply that a given room R can have incoming connections from rooms 2 to R-1. In this order, the first subsection 01 means that room 3 does not connect to room 1 but it certainly does to room 2. The following subsections code this information for rooms 4, 5, and 6. The length of this section is directly proportional to the number of rooms in the area, information provided by the designer during the setup.
- The second section 1 : 2 : 6 : 4 implicitly indicates the name of the rooms that contain, in this order, the following items: the initial room (only present in area 1), the door to area 2, the door to area 4, and the key. The length of this section varies depending on the number of areas connected to this one. This information is coded in the high-level graph.
- The third section 5 : 6 ; m1 : m3 codes first the locations for every monster in the area, and then the subtype of each of them. This means that room 5 contains a type 1 monster, and room 6 contains a type 3 monster.
- Analogously, the last section 5 ; t2 codes the location and subtype for every treasure chest. The lengths of these last two sections depend on the amounts of monsters and enemies, respectively, defined by the designer and coded in the high-level graph.

For this example, a LL-CFG is automatically generated for creating the language of all possible cyclic graphs containing 6 rooms, two doors, one key, two monsters, and 1 treasure chest. Given the number of rooms, connections to other areas (doors), and a list with the different kinds of game objects existing in the area, the LL-CFG that generates the corresponding language is created using a generic template that is analogous to the one shown in Figure 5.

4 The Evaluation Functions

The proposed evolutionary system uses different fitness functions for the HL-EA and the LL-EA. In both cases, evaluation comes from an analysis performed over the graph structures coded in the individuals, without needing to test the encoded maps in a real game. This way the system saves computational time as it evolves general-purpose maps, with optimal features that are independent of any game where they can be played in.

There are four features evaluated by the HL-EA fitness function for a given high-level graph, and other two features in case of LL-EA fitness function for low-level graphs. The goal of the HL-EA fitness function is to evaluate the general

structure of the high-level graph and the expected game experience. The four evaluated features are the following:

- Branching: measures the average distance in nodes (calculated by an A* algorithm) between every pair of subsequent areas z_i and z_{i+1}, by applying the following equation:

$$\sum_{i=1}^{n-1} \frac{dist(z_i, z_{(i+1)})}{(i-1)} \tag{1}$$

  Maximizing this equation avoids the generation of corridor-like or star-like maps, where the proximity of subsequent areas does not encourage players to branch up their path to the exit. Figure 7, section B, shows an evolved high-level graph of five areas ($n = 5$) using the proposed high-level encoding scheme. In this case, the branching equation scores 9/4 out of 10/4, which nearly maximizes this feature. The real world example in Figure 4 achieves a score of only 7/4 for the same number of zones.
- Recoil: applies a penalty every time an area has to be visited more than once. This balances the branching effect, fostering alternative paths, but penalizing recurrent areas in excess. The example shown in Figure 7 has a reduced recoil level, focused mainly in area 1, which is expected to be visited four times. However, the game map lets the player peek the last levels before the player gains access to them, creating, this way, long-term objectives. The example in Figure 4 compensates its lower branching with a lower recoil penalty, with section B being the most visited at three times.
- Progression: scores whether the occurrence of game objects increases as the player progresses, i.e. the number of game objects is greater in the later areas than in the first ones. This is calculated by applying the following equation:

$$\sum_{j=1}^{m} \left(\sum_{i=1}^{n-1} \frac{count_{j,i}}{rooms_i} - \frac{count_{j,i+1}}{rooms_{i+1}} \right) \tag{2}$$

  where $count_{j,i}$ is the amount of instances of type j game objects in the area i, and $rooms_i$ is the number of rooms in the area i. Maximizing this equation results is a better game experience because the player will encounter a higher density of game objects as he or she progresses, providing a motivation to keep on playing to discover new objects. Some corner cases can lead to odd progression scores, for example area 4 in Figure 3 contains only one room, so it can only be completely filled or empty of content. This feature depends on the designer's input parameters.
- Designers input parameters: the designer manually sets the types and amount of game objects to be included in the map. A penalty applies for every asked object that is not present in the map. This parameter in the high-level fitness function establishes a limit in the maximization of the progression feature because the amount of game objects proposed by the designer may be insufficient to reach a high level of progression. Figure 7, section A, shows some of the objects that have been set by the designer: 5 monsters, 3 treasure chests and a ratio of 6 monsters for every 5 treasures.

All these features are linearly combined to obtain the HL-fitness function as follows: $HLfitness(individual) = Branching + Progression - Recoil - Designer$.

There is a low-level graph per area in the high-level graph. Figure 7, section C, shows the low-level graph for the area 1, composed by four rooms. So, there exists one LL-EA fitness function for each low-level graph that evaluates the following two different features to get an enjoyable game:

- Critical path: given a low-level graph, this is the shortest path from the entrance to the exit passing through the room with the key, needed to go to the next area. The critical path is calculated by means of an A* algorithm. This feature applies a penalty to every room that, being outside of the critical path, does not contain any game objects. All of these rooms outside the critical path are optional, and and they are worthy from an enjoyable point of view only if they include additional content, e.g. rewards or treasures after defeating a monster. Otherwise, these kind of rooms are not interesting to be visited. The critical path in the example case of Figure 7, section C, starts with the room marked with an asterisk, then go to the room on the top right, collect the key to area 2 and take the exit to this area throughout the door marked with the number two in a circle. There are two rooms outside the critical path, located on the left and at the bottom, however these rooms contain two different kinds of game objects: treasure and monsters; so, no penalty is applied in this case. An example of the value of this scoring is that in the real world example of Figure 4, the only room out of the critical path that doesn't contain a treasure or a monster is occupied by a one-time tutorial, and not just empty.
- Clustering: calculates the average distance in rooms (calculated by an A* algorithm) between every pair of game objects of the same kind. This average distance is calculated for each kind of game object j and then, all average distances are summed up:

$$\sum_{j=1}^{m} \left(\sum_{k=1}^{p-1} \frac{dist(n_{j,k}, n_{j,k+1})}{p-1} \right) \tag{3}$$

  where p is the number of rooms that contain objects of the kind j, and $n_{j,k}$, $n_{j,k+1}$ are two diffent rooms that contain objects of the same kind j. Maximizing this equation encourages the dispersion of game objects among the existing rooms in the same area, and avoids clusters of the same kind of objects. The example shown in Figure 7, section C, has an average distance of 2 for monsters and 1 for treasures, giving as a result a clustering level of 3. Taking into account the topology of the low-level graph and the number of different game objects present in the rooms, it is considered that the proposed evolutionary system has generated a good design for the low-level graph of area 1.

All these features are also linearly combined to obtain the following LL-fitness function for each low-level graph: $LLfitness(individual) = Clustering - Critical\_Path$.

5 Results

The proposed system is distributed as a software tool called GraphQuest [2]. This software displays an intuitive graphic interface that allows setting up and running the proposed evolutionary algorithm. Figure 7 displays an overview of

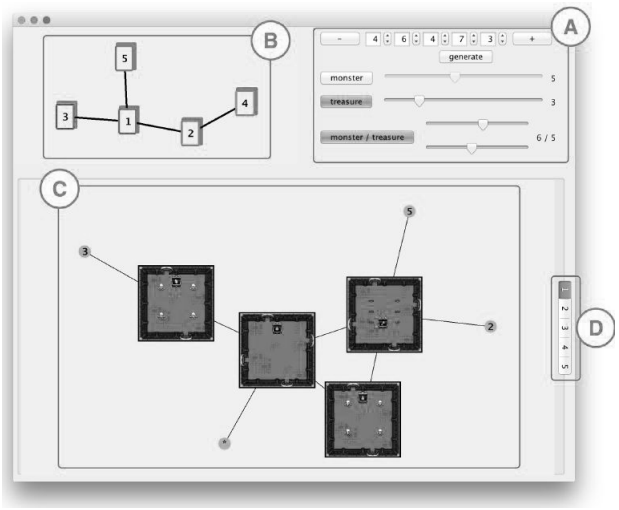

Fig. 7. Screenshot of the evolutionary world designer tool.

the software interface with a sample evolved game level. Notice that the evolutionary process is not deterministic, so that given a fixed set of input features, many different evolved levels can be obtained. The following sections are displayed by the software interface:

- Section A contains the set-up menu that designers use to specify content constraints for the generated maps. The list of features that can be constrained are: number of areas, number of rooms per area, number of monsters, number of treasure chests, and the desired monster/treasure ratio per area. As explained in the previous sections, these features shape the HL-CFG and the LL-CFG, as well as part of the HL evaluation. The system is scalable, accepting any number of additional features required by the designer.
- Section B shows the high-level graph depicting a general overview of the generated map and its interconnected areas. Five different areas named from 1 to 5 compose the level in this case. Area 1 contains the entrance room and Area 5 contains the final room. The evolutionary process evolved this map to a high branching level with a reduced recoil level, focused mainly in Area 1.
- Section C depicts the low-level graph for every area as a set of interconnected rooms. Each node displays a background image resembling a room

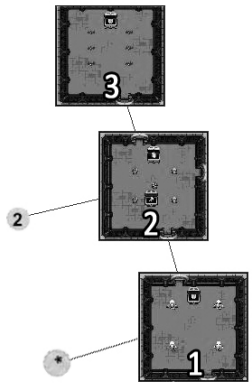

Fig. 8. Detail of an evolved low-level graph.

in a dungeon, with as many doors as the number of rooms connected to it. Numbers in circles represent the areas to which it is possible to go from a room when the corresponding key has been collected. Different monsters and treasure chests are displayed according to the content of every room: monsters (skeletons) and a treasure (jewel) on the left, a treasure chest in the middle, monsters (skeletons) and a treasure at the bottom, and, finally, a treasure and the key to area 2 on the top right. The start room is the one with an input connection marked with an asterisk. Clicking on each button in D changes the area displayed in C. In this example, there are five buttons to display each of the five areas of the game level.

Figure 8 shows the low-level graph for the first area from a different evolved map. In this case, the low-level EA was constrained to three rooms including three treasure chests and a five to one monster/treasure ratio. Room 1 is the entrance to the area. Room 2 contains the key and the door to Area 2, so the critical path is the following: enter Room 1, kill the monsters and get the treasure; then go to Room 2, kill the monsters, get the treasure and the key; then exit Area 1. Though Room 3 stays out of the critical path, it offers an optional challenge to the player comprised by six monsters protecting another treasure chest. This is achieved by analyzing the critical path as part of the low-level evaluation step. It is also important to notice that the treasure chests in rooms 1 and 2 belong to different kinds. Nevertheless, the chest in Room 3 is the same kind as that in Room 1. Evolution prevents object distribution from creating clusters of identical objects in adjacent rooms.

Test runs have been carried out for both HL-EA and LL-EA, in order to measure the average convergence speed in both cases. These tests were run for a map with 5 areas containing 5, 6, 7, 5, and 3 rooms each, 24 treasure chests, and a 3/4 treasure/monster ratio. A population of 30 individuals has been used, considering that it converges after 6 iterations without an improvement on the best fitness score. At each iteration, 18 new individuals were obtained by crossover, 20

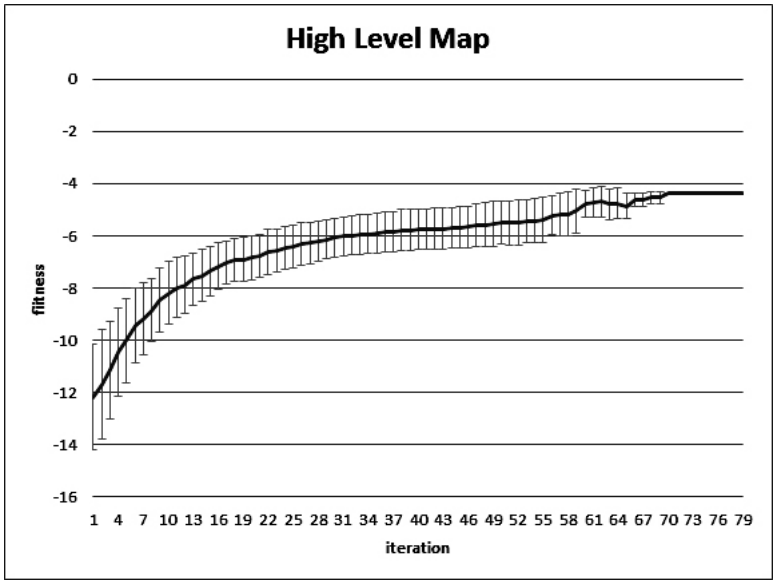

Fig. 9. Mean and standard deviation of the evolution of the best fitness score for high-level maps in 50 runs.

were generated by mutation, and 20 were randomly generated. Each experiment was run 50 times. Figures 9 and 10 show the mean best fitness scores obtained by the HL-EA and the LL-EA, respectively. Error bars show the standard deviations. Convergence is reached after an acceptable number of iterations: 79 and 14, respectively. All tests were carried out using a standard personal computer, taking no longer than 10 seconds for the combined HL-EA and LL-EA to finish.

6 Conclusions and Future Work

This paper presented a novel method and system for generating dungeon-type levels for games. The system described is the first to use a two-step method where a high-level graph is evolved, which is later expanded into a low-level graph. This distinction between two different levels of generation carries several benefits. Importantly, the high-level graph does not include all the details of the low-level graph, and therefore defines a smaller search space than if a low-level graph would have been searched for directly. The fitness evaluation can also be streamlined, and in the current implementation is computationally lightweight. The system further guarantees solvability of the levels by construction rather than by generate-and-test, and allows the designer a relatively high level of control.

Computational experiments show that the method reliably and quickly generates what to the authors of the current paper are apparently good levels. However, in future work we plan to carry out extensive testing with human player

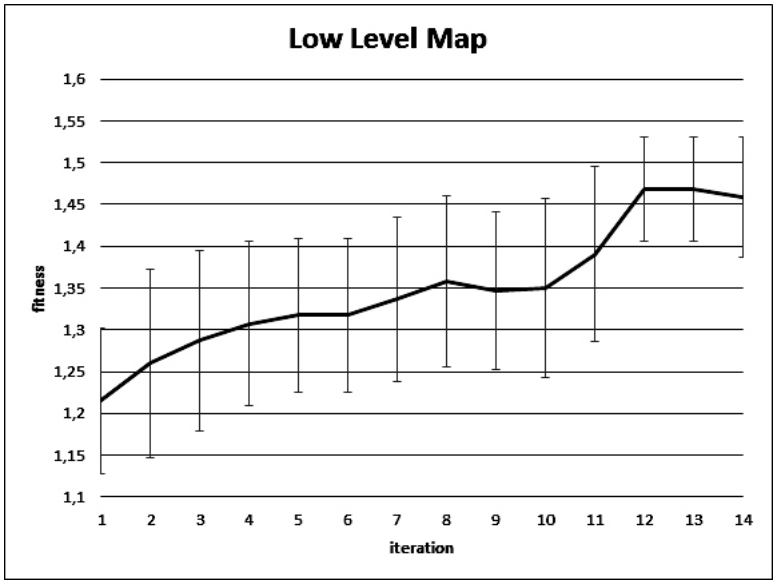

Fig. 10. Mean and standard deviation of the evolution of the best fitness score for low-level maps in 50 runs.

and designers, to verify that the generated levels are perceived as well-designed, and investigate to what extent the existing control options answer to game designers' need to parametrize their level generators. At this point, it would be also desirable that the combination weights of fitness evaluation components for both, HL-EA and LL-EA, were left to the designer's choice.

It would also be desirable to extend the current generator so that it can be embedded in a mixed initiative game design tool, such as Sentient Sketchbook [9]. Here, the designer would be able to make changes to the design at any time, and the system would give feedback about the changes from different angles (using different "computational critics"). Suggestions for changes could be generated using evolution from the current high-level level description; these could be accepted or ignored by the user in a form of optional interactive evolution.

References

1. Adams, E., and Dormans, J.: Game Mechanics: Advanced Game Design, (2012)
2. Izquierdo, R.: GraphQuest. http://robertoia.github.io/GraphQuest/ (2015)
3. Couchet, J., Manrique, D., and Porras, L.: Grammar-Guided Neural Architecture Evolution. In Bio-inspired Modeling of Cognitive Tasks, 437-446. Berlin, Heidelberg: Springer, (2007)
4. Dormans, J: Adventures in level design: generating missions and spaces for action adventure games. Proceedings of the 2010 Workshop on Procedural Content Generation in Games, (2010)

5. Font, J. M., Manrique, D., and Pascua ,E.: Grammar-Guided Evolutionary Construction of Bayesian Networks. 4th International Work-Conference on the Interplay Between Natural and Artificial Computation, 60–69, IWINAC 2011. La Palma: Springer Berlin Heidelberg, (2011)
6. Karavolos, D., Anders B., Bidarra, R.. Mixed-Initiative Design of Game Levels: Integrating Mission and Space into Level Generation. In Proceedings of the 10th International Conference on the Foundations of Digital Games. 2015.
7. Kerssemakers, M., Tuxen, J., Togelius, J., and Yannakakis, G. N.: A procedural procedural level generator generator. IEEE Conference on Computational Intelligence and Games, 335-341, CIG 2012. Granada. (2012)
8. Koster, R: Theory of fun for game design. O'Reilly Media, Inc. (2013)
9. Liapis, A., Yannakakis, G. N., and Togelius, J.:Sentient sketchbook: Computer-aided game level authoring. Proceedings of ACM Conference on Foundations of Digital Games, (2013)
10. Linden, R. van der, Lopes, R., and Bidarra, R. Designing Procedurally Generated Levels. In Ninth Artificial Intelligence and Interactive Digital Entertainment Conference. (2013)
11. Myerson, R.: Game Theory: Analysis of Conflict. Harvard University Press.(1991)
12. Nintendo: The Legend of Zelda: Oracle of Seasons. The Legend of Zelda: Oracle of Seasons. (2001)
13. Ochoa, G.: On genetic algorithms and Lindenmayer systems. In Parallel Problem Solving from NaturePPSN V, 335-344. Springer Berlin Heidelberg. (1998)
14. Preuss, M., Liapis, A., Togelius, J.: Searching for good and diverse game levels. 2014 IEEE Conference on Computational Intelligence and Games, 1-8, CIG 2014. Dortmund (2014)
15. Prusinkiewicz, P., Lindenmayer, A.: The algorithmic beauty of plants. Springer Science and Business Media (1990) Chicago
16. Shaker, N., Togelius, J., Nelson, M. J.: Procedural Content Generation in Games: A Textbook and an Overview of Current Research. Springer (2015).
17. Shaker, N., Nicolau, M., Yannakakis, G. N., Togelius, J, and O' Neill, M.: Evolving levels for super mario bros using grammatical evolution. In IEEE Conference Computational Intelligence and Games (CIG), 304-311, (2012).
18. Smith, A. M., and Mateas, M.: Answer Set Programming for Procedural Content Generation: A Design Space Approach. IEEE Transactions on Computational Intelligence and AI in Games, 3, 187-200 (2011)
19. Sorenson, N., and Pasquier, P.: Towards a generic framework for automated video game level creation. Applications of Evolutionary Computation, 131-140. Springer: Berlin Heidelberg (2010)
20. Sorenson, N., Pasquier , P., and DiPaola, S.: A generic approach to challenge modeling for the procedural creation of video game levels. IEEE Transactions on Computational Intelligence and AI in Games, 3, 229-244 (2011)
21. Togelius, J., Yannakakis, G. N., Stanley, K. O., and Browne, C.: Search-Based Procedural Content Generation: A Taxonomy and Survey. IEEE Transactions on Computational Intelligence and AI in Games, 1, 172-186 (2011)
22. Togelius, J., De Nardi, R., and Lucas, S. M.: Towards automatic personalised content creation for racing games. IEEE Symposium on Computational Intelligence and Games, 252-259, CIG 2007. Honolulu (2007)
23. Whigham, P. A.: Grammatically-based genetic programming. Proceedings of the workshop on genetic programming: from theory to real-world applications, 33-41 (1995)