POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione
Corso di Studi in Ingegneria Informatica

Automatic Generation of Maps in In Verbis Virtus

Relatore: **Prof. Pier Luca LANZI**

Correlatore: **Dott. Daniele LOIACONO**

Elaborato finale di: **Mattia FERRARI** **matr. 765998**

Anno accademico 2012/2013

Sommario

Negli ultimi anni, i continui miglioramenti hardware e software hanno permesso la creazione di videogiochi ambientati in mondi sempre più grandi e dettagliati. Questo ha provocato l'aumento delle aspettative dei giocatori riguardo alla quantità di contenuti, specialmente nei videogiochi AAA, portando all'aumento della quantità di lavoro richiesta dai designer. In particolare, sono necessari ambienti credibili per rendere i videogiochi attraenti. Creare simili ambienti in modo completamente manuale richiede però un team numeroso, e quindi un enorme budget. Di conseguenza, oggi la generazione procedurale dei contenuti (*procedural content generation*, PCG) è ampiamente usata dagli sviluppatori per creare contenuti automaticamente, in modo algoritmico. Le tecniche di PCG permettono la creazione di grandi quantità di contenuto impiegando meno designer, quindi sono spesso usate nei videogiochi per la creazione di ambienti esterni e interni.

Il videogioco *In Verbis Virtus*, che stiamo sviluppando alla Indomitus Games, è caratterizzato da ambienti interni sotterranei (dungeon) realizzati con una grafica all'avanguardia. L'introduzione di tecniche di PCG nel processo di sviluppo offre l'opportunità di creare livelli di gioco più grandi, mantenendo costi accettabili. In questa tesi, faremo una ricerca sugli approcci esistenti per la generazione di ambienti, per poi creare un algoritmo per la creazione automatica di mappe per *In Verbis Virtus*. Analizzeremo poi questo algoritmo per capire come controllare le mappe prodotte e per trovare i suoi punti di forza e i suoi limiti. Basandoci su questa analisi, estenderemo l'algoritmo per ottenere più controllo sul processo di generazione.

Organizzazione della Tesi

Questa tesi è organizzata come segue:

Nel Capitolo 2 discutiamo dell'uso di tecniche di PCG nei videogiochi. Presentiamo applicazioni rilevanti, dai primi videogiochi a quelli recenti, evidenziando i diversi problemi che la PCG ha aiutato a risolvere. Dopodiché descriviamo approcci per la generazione di ambienti, concentrandoci in particolare sugli ambienti interni.

Nel Capitolo 3 presentiamo il gioco *In Verbis Virtus*, su cui è focalizzata questa tesi, esponendo dettagli sull'ambientazione e il gameplay.

Nel Capitolo 4 descriviamo in dettaglio l'algoritmo che abbiamo creato per la generazione di dungeon in *In Verbis Virtus*. Per prima cosa facciamo una presentazione generale del nostro approccio. Poi descriviamo come abbiamo modellizzato gli elementi architettonici come stanze e corridoi. Infine mostriamo il modo in cui l'algoritmo colloca gli elementi architetturali per creare mappe.

Nel Capitolo 5 trattiamo la valutazione di mappe. Per prima cosa discutiamo applicazioni rilevanti della valutazione di mappe. Poi descriviamo le metriche che abbiamo scelto per valutare le mappe generate dal nostro algoritmo. Presentiamo un'analisi dei singoli parametri del nostro algoritmo, e infine un'analisi di combinazioni di parametri.

Nel Capitolo 6 descriviamo estensioni del nostro algoritmo che abbiamo introdotto per ottenere più controllo sul processo di generazione. Dopodiché presentiamo un'analisi di queste estensioni.

Contributi

Questa tesi contiene i seguenti contributi:

- Nel Capitolo 2 forniamo una panoramica sugli approcci per la generazione di ambienti, con particolare attenzione agli ambienti interni.
- Nel Capitolo 3 presentiamo il videogioco *In Verbis Virtus*, fornendo dettagli relativi all'ambientazione, al gameplay e agli strumenti usati per lo sviluppo.
- Nel Capitolo 4 descriviamo in dettaglio il nostro algoritmo per la generazione di dungeon. Mostriamo come l'algoritmo possa creare mappe

utilizzando elementi architetturali definiti dall'utente, evitando compenetrazioni tra gli asset.

- Nel Capitolo 5 mostriamo un approccio per la valutazione di mappe, usando metriche relative ai grafi. Forniamo indicazioni su come i parametri del nostro algoritmo influenzano le mappe generate.

- Nel Capitolo 6 descriviamo estensioni del nostro algoritmo che forniscono un maggiore controllo sul processo di generazione. Mostriamo come queste estensioni aiutino a risolvere i problemi di bias dell'algoritmo originale.

Ringraziamenti

Ringrazio il Prof. Pier Luca Lanzi e il Dott. Daniele Loiacono, per avermi guidato e supportato durante tutto il lavoro svolto per questa tesi.

Ringrazio la mia famiglia, per l'affetto e la comprensione che mi hanno dato in questi anni di studio.

Infine, ringrazio tutti i membri di Indomitus Games, per aver creduto nel progetto che ha portato alla realizzazione di questa tesi, e per il sostegno che continuano a dargli.

Mattia Ferrari

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent times, the continuous improvement of hardware and software allowed the creation of video games featuring increasingly large and detailed worlds. This have raised the expectations of players about the amount of available content, especially for AAA games, consequently increasing the amount of work required by designers. In particular, believable environments are necessary to make games attractive. Creating such environments completely by hand, however, requires a huge team and thus a huge budget. Accordingly, nowadays *procedural content generation* (PCG) is widely used among developers as the tool for the automatic creation of content, through algorithmic means. PCG allows the creation of large amount of content with fewer designers, thus it is often used in games to generate outdoor and indoor environments.

The game *In Verbis Virtus*, that we are developing at Indomitus Games, features dungeon-like indoor environments rendered in high-end graphics. The introduction of PCG techniques in the development process offers the opportunity to create larger game levels, while keeping costs acceptable. In this thesis, we will do a research on existing approaches for environment generation, and then create an algorithm for the automatic creation of maps for *In Verbis Virtus*. We will analyze the algorithm to find how to control its output and understand its strengths and limits. Basing on this analysis, we will then extend the algorithm to get more control on the generation process.

1.1 Outline

The thesis is organized as follows:

In Chapter 2, we discuss the use of PCG in games. We present relevant applications, from early games to recent ones, highlighting the different problems that PCG helped to solve. We then describe approaches for the generation of environments, focusing in particular on indoor environments.

In Chapter 3, we present the game *In Verbis Virtus*, on which we focus in this thesis. Several details of the setting and the gameplay are mentioned.

In Chapter 4, we describe in detail the algorithm we created for the generation of dungeon-like maps in *In Verbis Virtus*. We first give a general presentation of our approach. Then we describe how we modeled architectural elements such as rooms and corridors. Finally we show how the algorithm places architectural elements to create maps.

In Chapter 5, we deal with the evaluation of maps. We first discuss relevant applications of map evaluation. Then we describe the metrics we chose to evaluate maps generated by our algorithm. We present an analysis of single parameters of the algorithm, and finally an analysis of parameter combinations.

In Chapter 6, we describe extensions of our algorithm that we introduced to get more control on the generation process. We then present an analysis of the extensions.

1.2 Contributions

The thesis presents the following contributions:

- In Chapter 2, we give an overview of several approaches for the generation of environments, with particular attention on indoor environments.
- In Chapter 3, we present the game *In Verbis Virtus*, giving details about the setting, the gameplay and the tools used for the development.
- In Chapter 4, we describe in detail our algorithm for the generation of dungeon-like maps. We show how the algorithm can create maps with user-defined architectural elements, avoiding compenetrations of assets.
- In Chapter 5, we show an approach for the evaluation of maps, using graph metrics. We give indications on how the parameters of our algorithm affect generated maps.

- In Chapter 6, we describe extensions of our algorithm that give more control on the generation process. We show how the extensions help to solve bias problems of the original algorithm.

Chapter 2

Procedural Content Generation for Environment Creation

In this chapter, we first discuss the use of procedural content generation in games. We then give an overview of the algorithms for the generation of indoor and outdoor environments. At the end, we describe techniques for the generation of outdoor environments and techniques for the generation of indoor environments.

2.1 Procedural Content Generation

The earliest computer games were severely limited by memory constraints and had a little space to store content, accordingly large game levels had to be generated algorithmically on the fly. Procedural content generation (PCG) was thus introduced as the tool for the automatic creation of content, through algorithmic means.

Beneath Apple Manor[1], by Don Worth (1978), (Figure 2.1) was one of the first games to use procedural generation. It was a role-playing game (RPG), i.e. a genre where players control a character whose skills can be improved by solving quests and killing enemies. *Beneath Apple Manor* featured procedurally generated dungeons, i.e. underground environments consisting of rooms and tunnels. When a new game was started levels were randomly generated, so levels changed each game, greatly improving the replay value. The most

[1] `http://worth.bol.ucla.edu/`

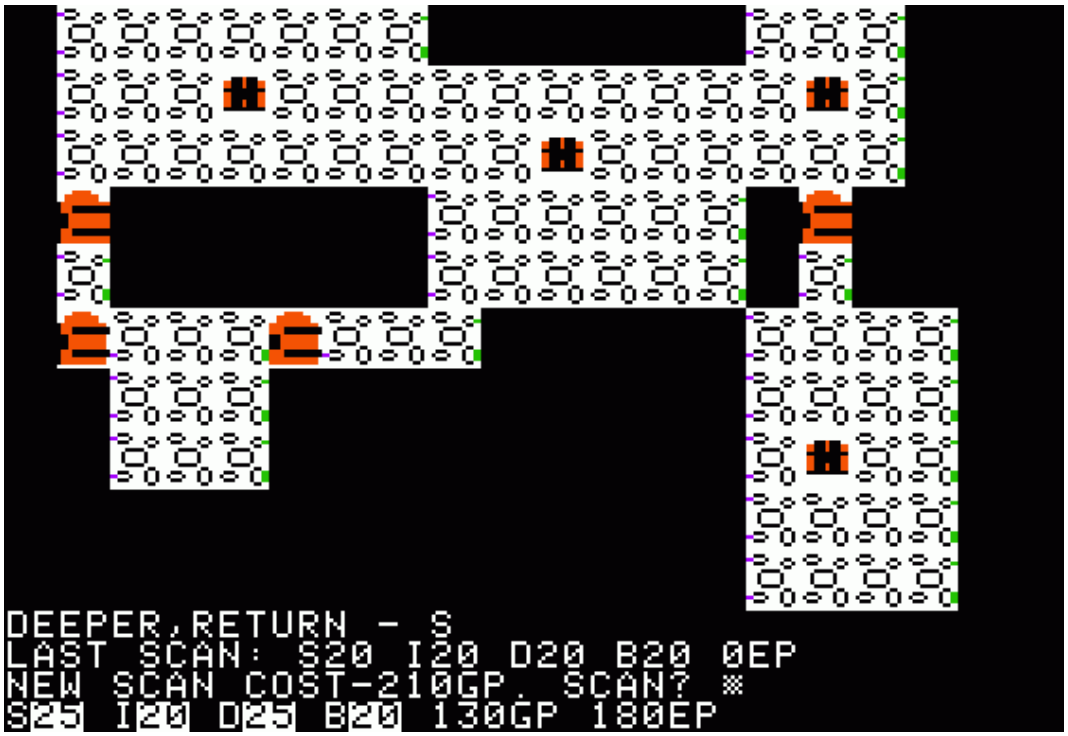


Figure 2.1: A screenshot of *Beneath Apple Manor*.

famous clone of *Beneath Apple Manor* was *Rogue*[2], by A.I. Design (1980), which originated the term *roguelike* to indicate this kind of games.

Elite[3], by Acornsoft (1984), was a space trading simulation game set in a universe containing 8 galaxies with 256 planets each. For each planet the position, the name, the description and the price of commodities were procedurally generated on the fly by an algorithm that used fixed parameter values. So, unlike *Beneath Apple Manor*, in *Elite* the game world was persistent among replays.

Other early games using PCG were *The Seven Cities of Gold*[4] (Ozark Softscape, 1985) (Figure 2.2), a strategy game in which the world map was randomly generated, and *Rescue on Fractalus!*[5] (Lucasfilm Games, 1984) (Figure 2.3), a sci-fi flight simulator that used fractals to create a mountainous environment.

Nowadays memory consumption is no longer a central issue. Hardware

[2] http://en.wikipedia.org/wiki/Rogue_(computer_game)
[3] http://www.iancgbell.clara.net/elite/
[4] http://en.wikipedia.org/wiki/The_Seven_Cities_of_Gold_(video_game)
[5] http://en.wikipedia.org/wiki/Rescue_on_Fractalus!

Figure 2.2: A screenshot of *Rescue on Fractalus!*.

Figure 2.3: A screenshot of *The Seven Cities of Gold*.

Figure 2.4: The city of *Grand Theft Auto IV*.

and software improvements, however, have raised the expectations of players about the amount of available content, especially for AAA games, increasing the amount of work required by designers. For example games of the *Grand Theft Auto*[6] series (Rockstar Games, 1996-2013) (Figure 2.4), feature large cities created by hand by artists, that require a huge team and thus a huge budget. Accordingly, nowadays PCG is widely used among developers to create large amount of content with fewer designers.

For instance the RPG *Diablo*[7], by Blizzard Entertainment (1996), has procedurally generated outdoor maps, dungeons and items, in a roguelike fashion. These features are also present in its sequels *Diablo II* (2000) and *Diablo III* (2012). *Just Cause*[8] (Avalanche Studios, 2006), an action-adventure game, has a large and varied group of tropical islands (Figure 2.5) created using procedural methods. *Spore*[9] (Maxis, 2008), an evolution simulation game, uses PCG extensively to create landscapes and creatures. Animations of the creatures are created on the fly according to their shape, and even background music is procedural. *Minecraft*[10] (Mojang, 2009) is a sandbox game, i.e. large freedom

[6] `www.rockstargames.com/grandtheftauto/`
[7] `http://us.blizzard.com/en-us/games/legacy/`
[8] `http://www.justcause.com/`
[9] `http://www.spore.com/`
[10] `https://minecraft.net/`

Figure 2.5: A screenshot of *Just Cause*.

is left to players and no explicit objective is given. Players can explore, modify the environment, fight enemies and craft items in a vast world randomly generated at each new game.

Today several middlewares for procedural generation exist, for instance *CityEngine*[11], for the generation of urban environments; *SpeedTree*[12], for the generation of large and detailed forests. *SpeedTree* is widely used in AAA games, for instance *The Elder Scrolls IV: Oblivion*[13] (Bethesda Softworks, 2006) (Figure 2.6) and *The Witcher 2*[14] (CDProjekt, 2011).

2.1.1 Indoor and outdoor environments

PCG algorithms for indoor environments are very different from those for outdoor environments. For this reason we present them in different sections. Indoor environments can be described in terms of walls delimiting viable spaces. Examples of indoor environments include the abstract maze of *Pac-Man*[15], by Namco (1980), (Figure 2.7); the natural caves of *The Elder Scrolls V: Skyrim*[16],

[11] `http://www.esri.com/software/cityengine/`
[12] `http://www.speedtree.com/`
[13] `http://www.elderscrolls.com/oblivion/`
[14] `http://www.thewitcher.com/witcher2/`
[15] `http://it.wikipedia.org/wiki/Pac-Man`
[16] `http://www.elderscrolls.com/skyrim`

Figure 2.6: A forest of *The Elder Scrolls IV: Oblivion*, created with the *SpeedTree* tools.

Figure 2.7: The maze of *Pac-Man*.

by Bethesda Softworks (2011), (Figure 2.8); the interior of modern buildings in the *Call of Duty*[17] series, by Infinity Ward (2003-2013) (Figure 2.9). Outdoor environments are open and unrestricted spaces. Examples of outdoor environments include the cities of the *Grand Theft Auto* series (Figure 2.4), the forests of *The Elder Scrolls IV: Oblivion* (Figure 2.6) and the galaxies of *Elite*.

2.2 Outdoor environments

In this section we discuss briefly relevant techniques for the generation of outdoor environments.

2.2.1 Cities

City generation consists in the creation of an urban area, and it deals mainly with the generation of a road network and the creation of buildings that can vary in function and architectural style.

[17] `http://www.callofduty.com/`

Figure 2.8: A cave of *The Elder Scrolls V: Skyrim*.

Figure 2.9: The interior of a building of *Call of Duty: Modern Warfare 3*.

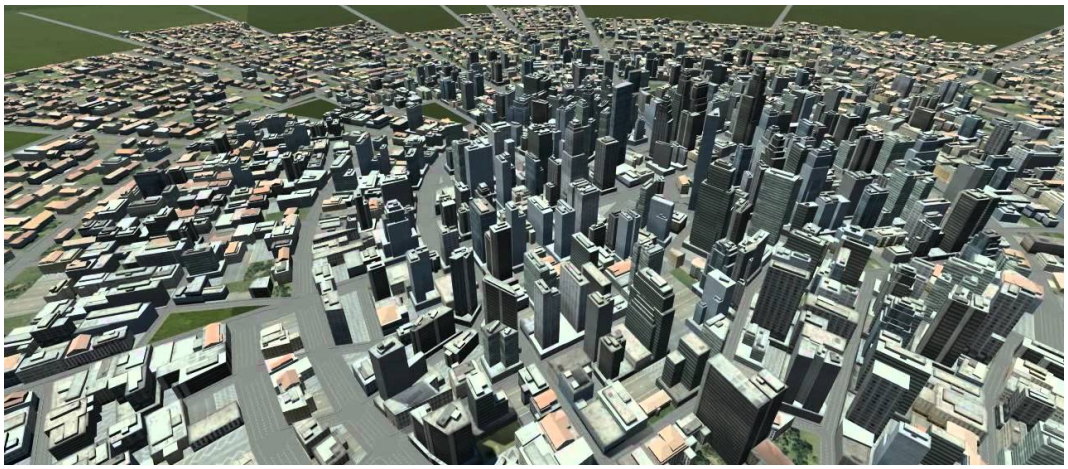


Figure 2.10: An example of city generated with *CityEngine*

CityEngine

CityEngine, described by Parish and Müller [12], is a system that generates a complete city starting from sociostatistical and geographical information. An example city is shown in Figure 2.10.

Most of the input data is provided to the system as bidimensional image maps, which can be drawn by hand or obtained by maps of existing urban areas. The input data can be categorized in two classes: geographical maps and sociostatistical maps. Geographical maps describe the elevation of the terrain and the distribution of water and vegetation. Sociostatistical maps contain information about population density, distribution of residential and commercial zones, street patterns and maximal height of the buildings. Using the input data, especially population density and road patterns, the system builds the road network, composed by highways and streets. After the creation of roads the city area results subdivided into small areas, called blocks. Each block is in turn subdivided into lots, and in each lot a building is created. The geometry of buildings is generated starting from an arbitrary ground plan and applying transformations including extrusion, scale, move and branching.

2.2.2 Forests

Forest generation mainly deals with the problems of creating plant models and placing them. Here we focus on this second problem. West [15] describes methods to arrange plants through random scattering, with the constraint that

there must be a minimum distance between plants. While these methods are effective in particular contexts, they tend to produce a uniform distribution that may look unnatural. Lane and Prusinkiewicz [8], on the other hand, present two approaches able to create clustering in the distribution: local-to-global and global-to-local.

Local-to-global

The first approach, named local-to-global, uses a set of rules to represent the interactions between plants. Plants of different species are initially scattered over the forest area. Then the simulation is started where the rules are applied. Plants grow and can spread seeds around themselves, they can die if overshadowed by other plants or when they are too old. Each species have different characteristics, such as growth speed and shade tolerance. Figure 2.11 shows three stages of this algorithm.

Global-to-local

The second approach, named global-to-local, places the plants on the surface according to a probability distribution. This distribution is initially uniform, and it is modified each time a new plant is added. Each plant, according to its species, deforms the probability distribution in a small surrounding area, thus affecting the next placements.

2.2.3 Terrain

Terrains are often represented through height-maps, i.e. bidimensional grids of elevation values. The following are methods for the generation of height-maps.

Mid-point displacement

This method, described by Miller [11], subdivides iteratively a randomly generated coarse height-map. The elevation of a new point is set to the average of its neighbor points, plus a random offset. The range of the offset is decreased at each iteration, according to a parameter that controls the roughness of the resulting terrain.

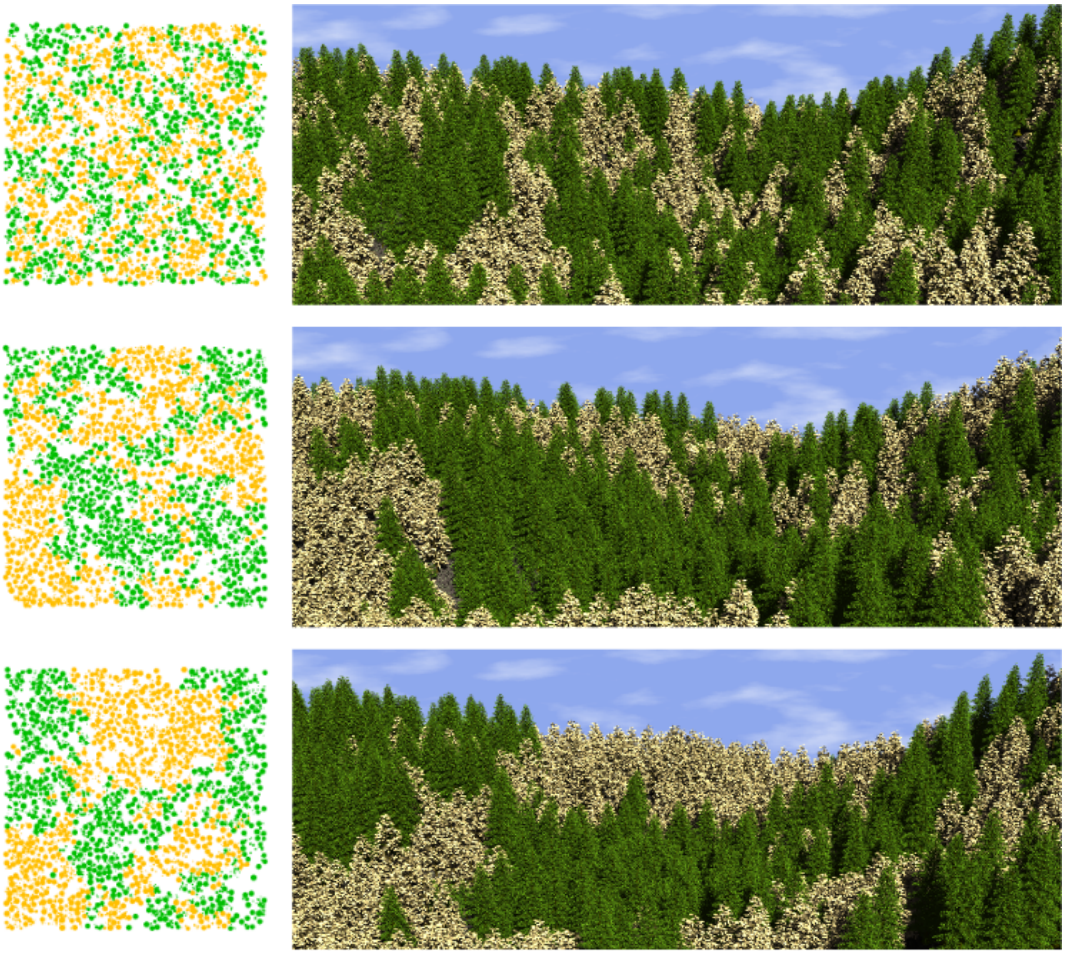


Figure 2.11: Three stages of the local-to-global forest generation algorithm, with two species of plants.

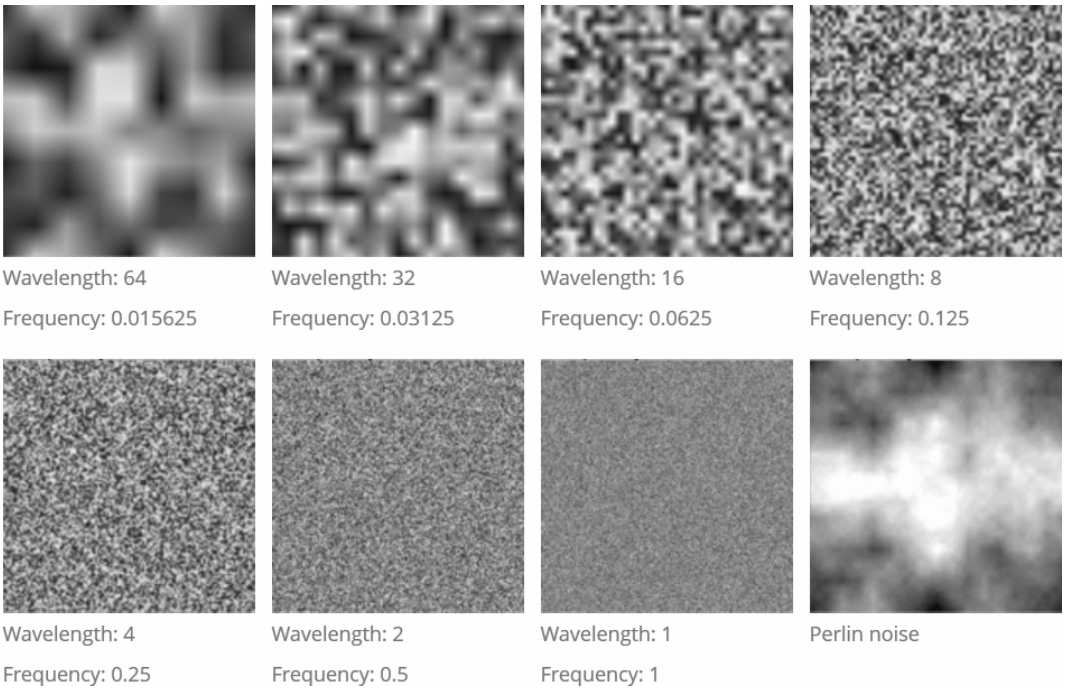


Figure 2.12: Seven octaves (noise functions) and the resulting Perlin noise.

Perlin noise

Perlin noise, originally described by Perlin [13], is a fractal noise generator. It generates noise by summing a succession of noise functions with different frequencies. An height map can be obtained using two-dimensional noise functions. Each of these noise functions is called octave. This is because each function have a frequency which is double the frequency of the previous one, a property that also have octaves in music. Each noise function has a weight, for terrain generation usually the higher the frequency, the less the weight. The number of octaves used depends on the resolution needed for the result. Figure 2.12 shows a set of octaves and the resulting Perlin noise.

Bitmap

This approach, proposed by Martin [9], takes as input a greyscale bitmap image. This image represents a rough height-map that describes the desired general features. The given height-map is smoothed using convolution[18], Gaus-

[18] `http://en.wikipedia.org/wiki/Convolution`

sian filtering[19] or B-splines[20], and the result is then converted to 3D terrain.

2.3 Indoor environments

We describe here algorithms for the generation of indoor environments, grouped according to the approach they adopt. We focus on bidimensional maps, but the same algorithms can be extended to generate tridimensional maps. The groups of algorithms we present are: subdivision-based generators, that partition a surface into distinct areas; graph-based generators, that represent maps using graphs; digger generators, that mimic the behavior of a digger creating tunnels and rooms; uniform generators, that first create rooms and then connect them with corridors; tile-based generators, that cover a surface with tiles containing parts of the map; cellular automaton generators, that create floor and walls from the cells of a cellular automaton; maze generators.

2.3.1 Subdivision-based generators

The techniques using a subdivision-based approach take as input a constrained surface and return the same surface partitioned into distinct areas. The game *Frozen Synapse*[21], by Mode 7 Games (2011), uses an algorithm of this type to generate levels.

Map generation in *Frozen Synapse*

Frozen Synapse is a turn-based strategy game, in which each player controls a small team and has to eliminate all enemies. Maps are procedurally generated to resemble floors of modern buildings. A screenshot of the game is show in Figure 2.13. The generating algorithm[22] subdivides the 2D floor area into rooms whose width and height is in a predefined range. The subdivision is rectilinear, that is all lines are perpendicular or parallel to each other. The fact that all the walls are placed in a orthogonal way, however, is not a big limitation for this game, being perfectly conceivable for modern buildings. The

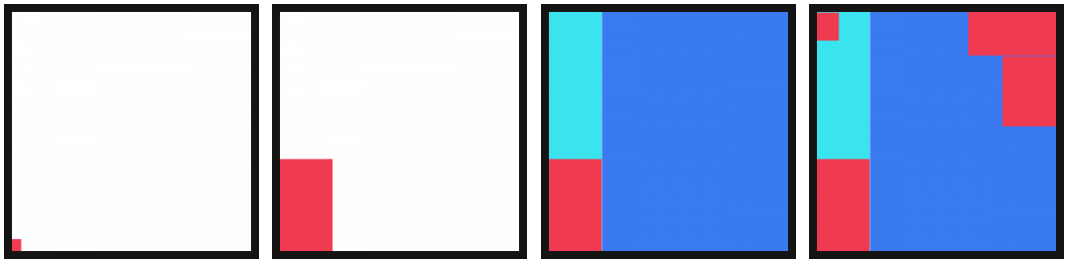
[19] `http://en.wikipedia.org/wiki/Gaussian_filter`

[20] `https://en.wikipedia.org/wiki/B-spline`

[21] `http://www.frozensynapse.com/`

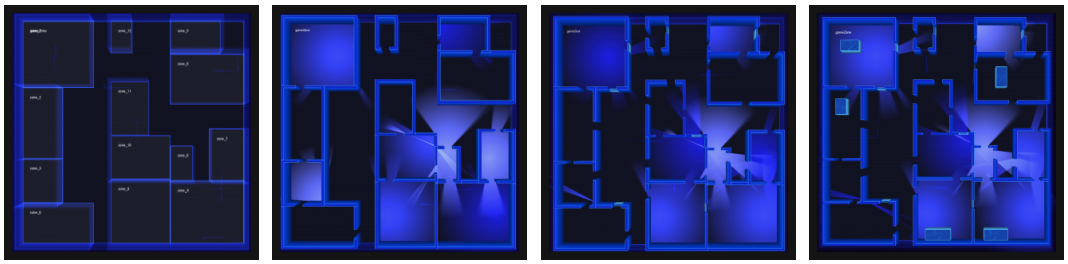
[22] `http://www.desura.com/games/frozen-synapse/news/frozen-synapse-procedural-level-generation`

Figure 2.13: A screenshot of *Frozen Synapse*.



(a) (b) (c) (d)

Figure 2.14: Steps for the generation of rooms in *Frozen Synapse*.



(a) (b) (c) (d)

Figure 2.15: Steps for the placement of doors, windows and objects in *Frozen Synapse*.

algorithm performs the following steps:

1. Starting from a rectangular area of the desired size, a random corner is selected (Figure 2.14a).
2. A randomly sized room-shaped block is created from that corner (Figure 2.14b).
3. The rest of the area is partitioned into two rectangles (the cyan and blue in Figure 2.14c).
4. The process is repeated recursively from step 1 for each rectangle obtained in step 3, if it is large enough to contain new rooms (Figure 2.14d).
5. The process stops when there are no more rectangles large enough.

After these steps are completed the resulting map consists of groups of rooms adjacent to each other. A possible arrangement of rooms can be seen in Figure 2.15a. The algorithm then adds doors, so that each room is accessible from every point in the map, windows, through which units can see and fire, and boxes, for cover. These are added performing the following steps:

1. For each group of rooms, starting from its top left room, a doorway is created to each adjacent room. The process is repeated recursively for each of these adjacent rooms, creating doorways to the rooms adjacent to them, and so on. For each group of rooms at least one doorway to the outside is also created. (Figure 2.15b)
2. More doors and windows are randomly added (Figure 2.15c).

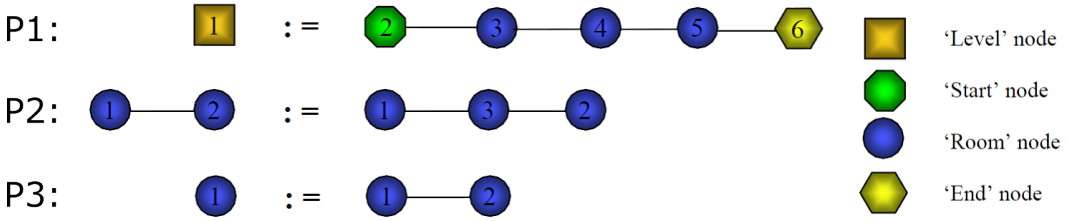


Figure 2.16: Example of production rules of a graph grammar for level generation.

3. Boxes for cover are randomly added. At this point the map is finished (Figure 2.15d).

2.3.2 Graph-based generators

Indoor maps can be described using graphs, with nodes representing different areas and edges representing the connections between them. However a major problem of this approach is the physical placement of nodes inside a constrained bidimensional area satisfying the connections defined by the graph.

The *Dungeon Generation System*

The *Dungeon Generation System*, described by Adams et al. [1], is an algorithm for the generation of dungeons using graphs. The graph is generated by applying graph grammar production rules [2]. An example of grammar graph production rules is shown in Figure 2.16. Each node represents a room: the *Start* node represents the room where the player spawns, while the *End* node represents the goal room that the player has to reach to finish the level. Each edge represents a doorway between the rooms it connects. The production rule *R1* creates a basic structure for the level from a *Level* node, with three rooms in a row between the *Start* and *End* rooms. The rule *R2* adds a new node between two existing nodes, so it prolongs an existing branch of the graph. The rule *R3* adds a new node connected to a single existing node, so it can create a new branch.

The graph generation process starts with a *Level* node and, at each step, applies one of the production rules, thus modifying the graph. Examples of steps is shown in Figure 2.17. The algorithm, at each step, evaluates the graph with respect to several metrics. The considered metrics are the size of the level, the difficulty and the fun-value. For each metric the desired range is given as

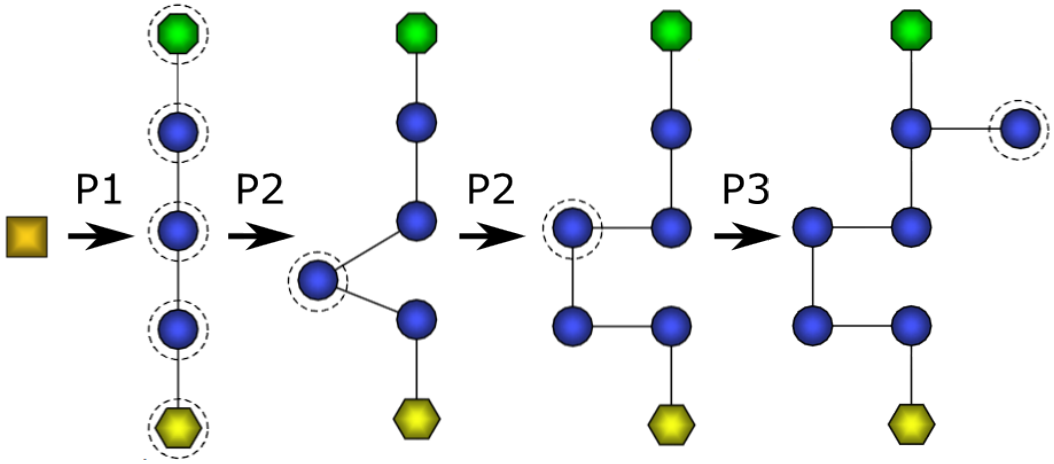


Figure 2.17: Example of derivation of a graph using the production rules of Figure 2.16

an input parameter. The algorithm is based on one known as hill climbing[23], which is a heuristic approach to explore a state space. The algorithm chooses the production rules to obtain a graph that satisfies the desired ranges. It is not guaranteed to succeed, and a maximum number of attempts is specified.

After the graph is created, the algorithm places gameplay objects in the pointless areas of the map. These are defined as areas that the player does not have to visit to complete the level. A pointless area is made useful by placing in its furthest away node a sufficient reward or a key. In the latter case the related door must be placed in an obligatory passage towards the exit of the level. Finally the algorithm adds more gameplay objects, taking into account the desired difficulty and fun-value. The *Dungeon Generation System* does not include methods to create a geometric description of the levels from the graphs, which remains a major challenge for graph-based algorithms.

2.3.3 Digger generators

These algorithms mimic the behaviour of a digger that creates a system of interconnected tunnels and rooms, starting from an area filled with solid material. This method[24] is used in the roguelike game *Tyrant*[25], by Mike An-

[23] `http://en.wikipedia.org/wiki/Hill_climbing`

[24] `http://www.roguebasin.roguelikedevelopment.org/index.php?title=Dungeon-Building_Algorithm`

[25] `http://games.hughesclan.com/tyrant/#`

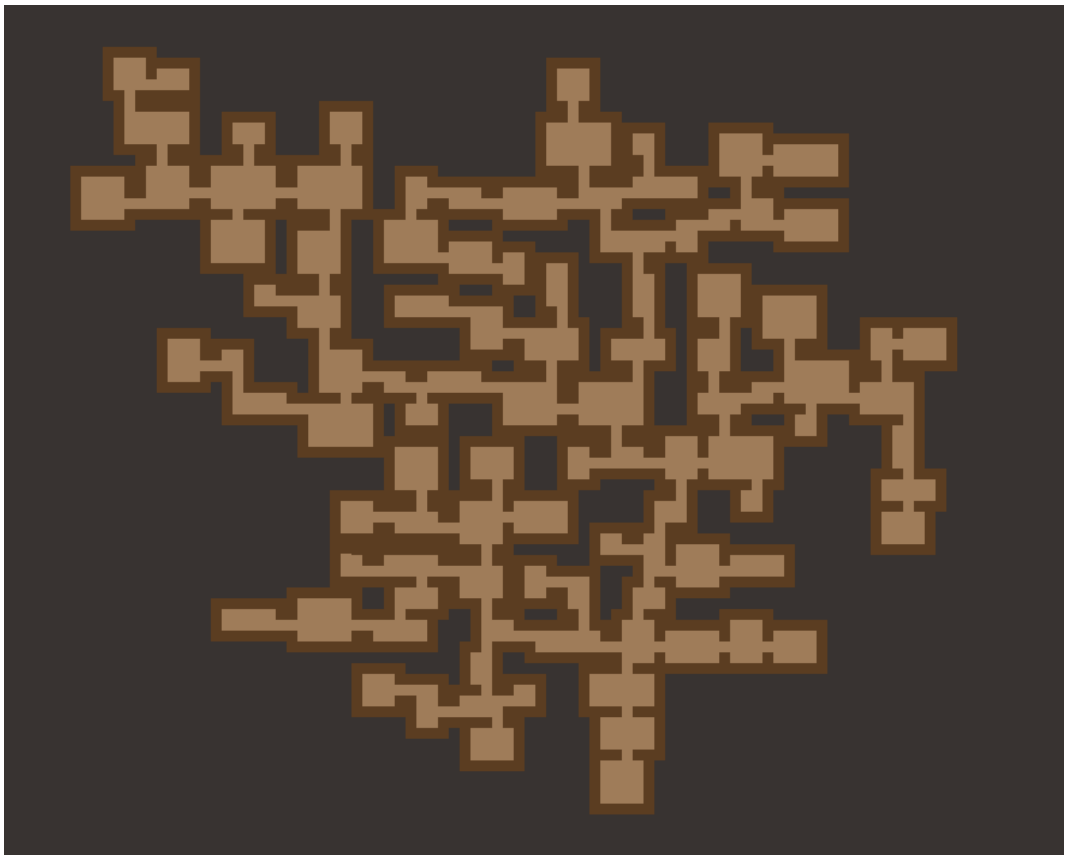


Figure 2.18: Example of a dungeon map from *Shadow Island*, generated using the digger generator algorithm.

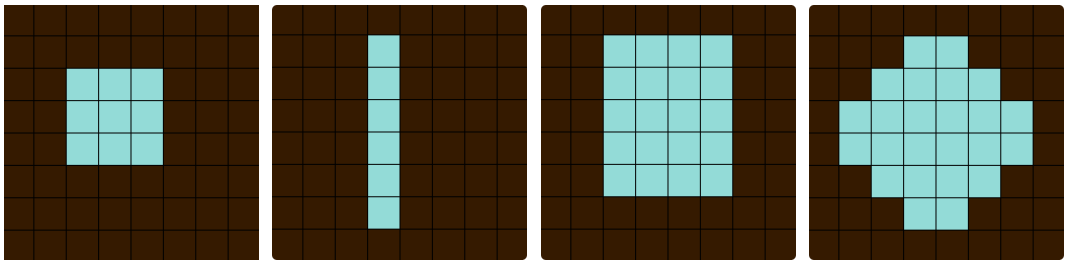
derson (2007). Figure 2.18 shows a map generated with this algorithm. The algorithm works on a discrete representation of the map: the available area is subdivided into square cells, called positions. Positions can contain floor, which is walkable, or solid material, which cannot be traversed.

In the algorithm the term *feature* indicates a component of the map, e.g. large room, small room, corridor, etc. A set of features must be passed as input to the algorithm. Examples of features are shown in Figure 2.19. The algorithm generates the map executing the following steps:

1. All the positions are filled with solid material.

2. A room, chosen from the defined features, is created in the center of the map (Figure 2.20a).

3. A random wall is chosen randomly from the map. A wall is a solid position adjacent to a floor position (not diagonally).



(a) Small room (b) Corridor (c) Large room (d) Diamond room

Figure 2.19: Examples of features of the digger generator algorithm. Dark positions represent solid material, while light positions represent walkable floor.

4. A new feature is chosen randomly from those given as input. Starting from the wall position previously selected, the neighboring positions are tested to check if the new feature can be placed without neither overlapping an already placed feature, nor exiting the border of the map (Figure 2.20b). The checked area is one position bigger on every side with respect to the new feature. This is because there must always be at least one solid position between the features, to separate them.

5. If there is enough space for the new feature, it is placed in the map. The previously selected wall is replaced by a door (Figure 2.20c).

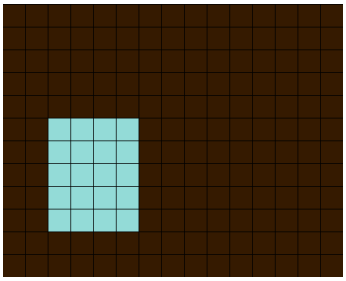
6. The steps are repeated from 3, in order to add more features. The process is stopped when a predefined end condition is met, e.g. when there is no more space for new features, or a certain number of rooms has been placed.

Features are always added linking them to a single floor position of maps. For this reason the maps cannot have loops, so only tree-like maps can be generated by this algorithm.

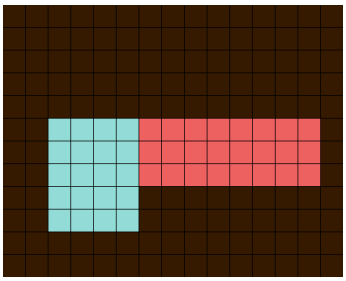
2.3.4 Uniform generators

These algorithms work in two phases: first, they place rooms; then they create corridors to connect the rooms. This approach is mainly adopted for dungeons in rougelike games, using a discrete representation of the map.

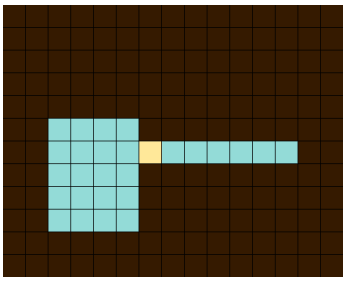
The rooms that can be placed are passed as input, like in the digger generator. The algorithm chooses one of them randomly and puts it in the map



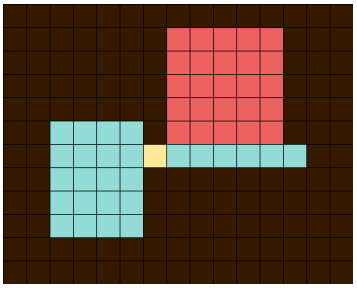
(a) The first room is placed.



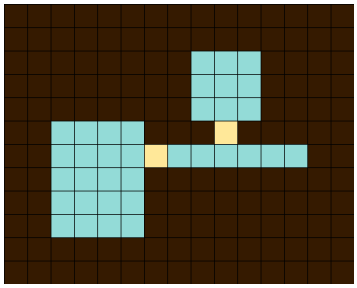
(b) The red area is scanned to check if a corridor can be placed.



(c) The corridor is added.



(d) The red area is scanned to check if a small room can be placed.



(e) The small room is added.

Figure 2.20: Steps of the digger generator algorithm. Red positions are tested to check if the new feature can be added without overlapping other features, yellow positions represent doorways.

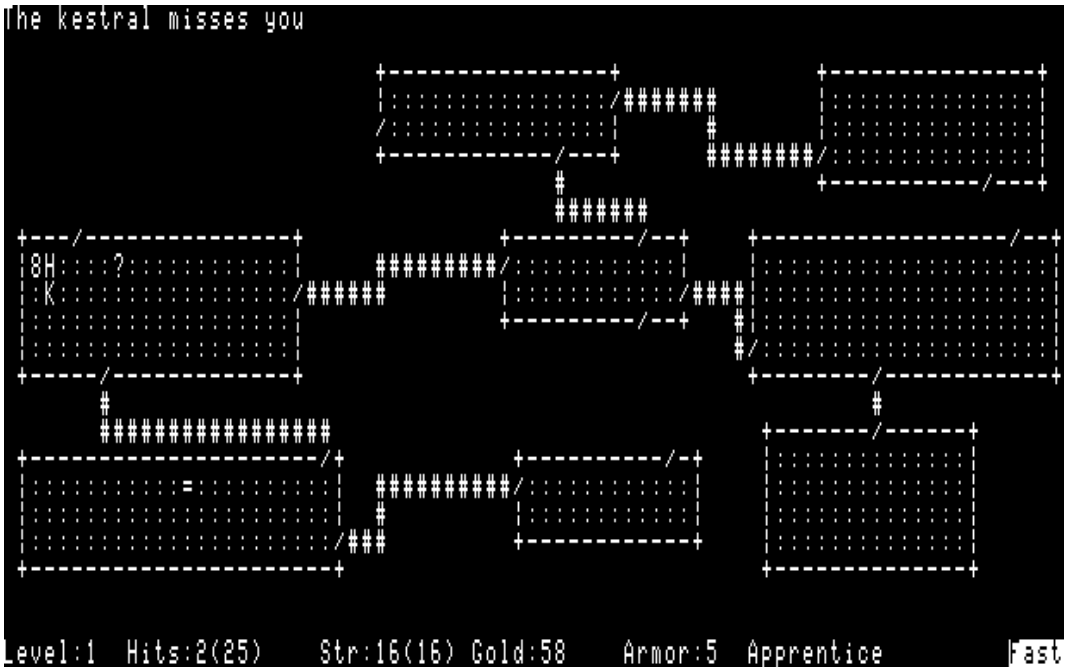


Figure 2.21: A screenshot of *Rogue*. Rooms are placed using a grid layout.

so that it does not overlap any other room; it keeps doing this until a certain number has been placed, or there is not enough space for more rooms. Four algorithms of this kind are described below.

Grid-based algorithm

The original *Rogue* used this algorithm[26], an example of map is shown in Figure 2.21. The positions of the map are partitioned in a grid layout, each position of the grid is thus a square area composed of positions of the map. Rooms are added inside empty positions of the grid. By doing this no overlap check is needed, however the resulting arrangement is strongly affected by the grid layout. Some of the grid positions may be left empty.

The corridors are made between rooms lying in neighboring positions of the grid. At least enough corridors to connect all rooms are created. The corridors are created with a Z shape, as can be seen in Figure 2.21, which can become an L or I shape in particular conditions.

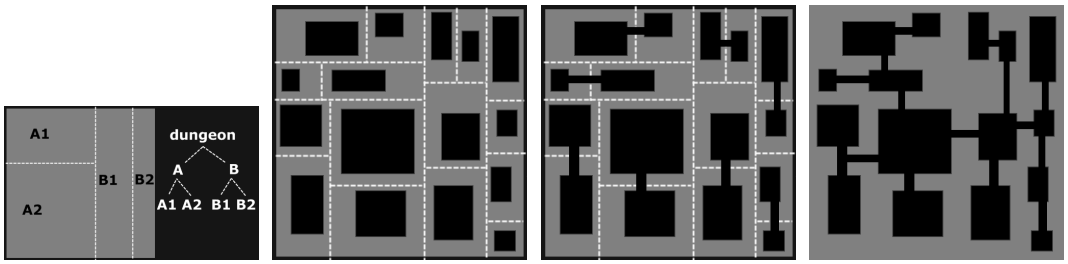
[26] `http://roguebasin.roguelikedevelopment.org/index.php?title=Simple_Rogue_levels`

Figure 2.22: A screenshot of *Angband*.

Random placement algorithm

This algorithm is used by the roguelike game *Angband*[27] (1990). A screenshot of the game is shown in Figure 2.22. When placing a new room a location on the map is randomly selected. It is checked whether the room can be put there, so that it does not cause any overlap. If there is enough space, the room is placed, otherwise other attempts are made until a suitable position is found.

For each corridor that has to be created, a path is generated from the origin room to the destination one. This is done through a random walk with an higher probability to move towards the destination room, with respect to other directions. The corridor is then built following this path. If two corridors overlap at some point, they are merged at that point. The corridors obtained can be winding, going in and out the same room multiple times and bend back to themselves.



(a) (b) (c) (d)

Figure 2.23: Examples of steps of the BSP-based dungeon generation process.

BSP-based algorithm

This method[28] uses binary space partitioning (BSP)[29] to recursively subdivide the map area into rectangles. Starting from the whole map area it performs a split choosing randomly the direction (horizontal or vertical) and the position. By doing this two rectangular areas are obtained, called sub-dungeons. For each of them a new random split is made and the process goes on recursively, thus creating a BSP tree in which each node is a rectangular area (Figure 2.23a). The process stops when all the rectangles corresponding to the leaves of the tree have approximately the size of the given rooms. At this point a room is placed inside each of these areas (Figure 2.23b). Due to the partitioning of the map no overlap check is needed.

To build corridors the algorithm loops through all the leaves of the tree, connecting each one to its sister (Figure 2.23c). If two rooms have face-to-face walls an I shaped corridor is used, otherwise a Z shaped corridor is used. The algorithm then goes up one level in the tree and repeats the process for the parent sub-dungeons. This is repeated until the first two sub-dungeons are connected (Figure 2.23d). Since connections are made following the BSP tree, there cannot be loops in the map.

[27] `http://rephial.org/`

[28] `http://doryen.eptalys.net/articles/bsp-dungeon-generation/`

[29] `http://en.wikipedia.org/w/index.php?title=Binary_space_partitioning&oldid=542008108`

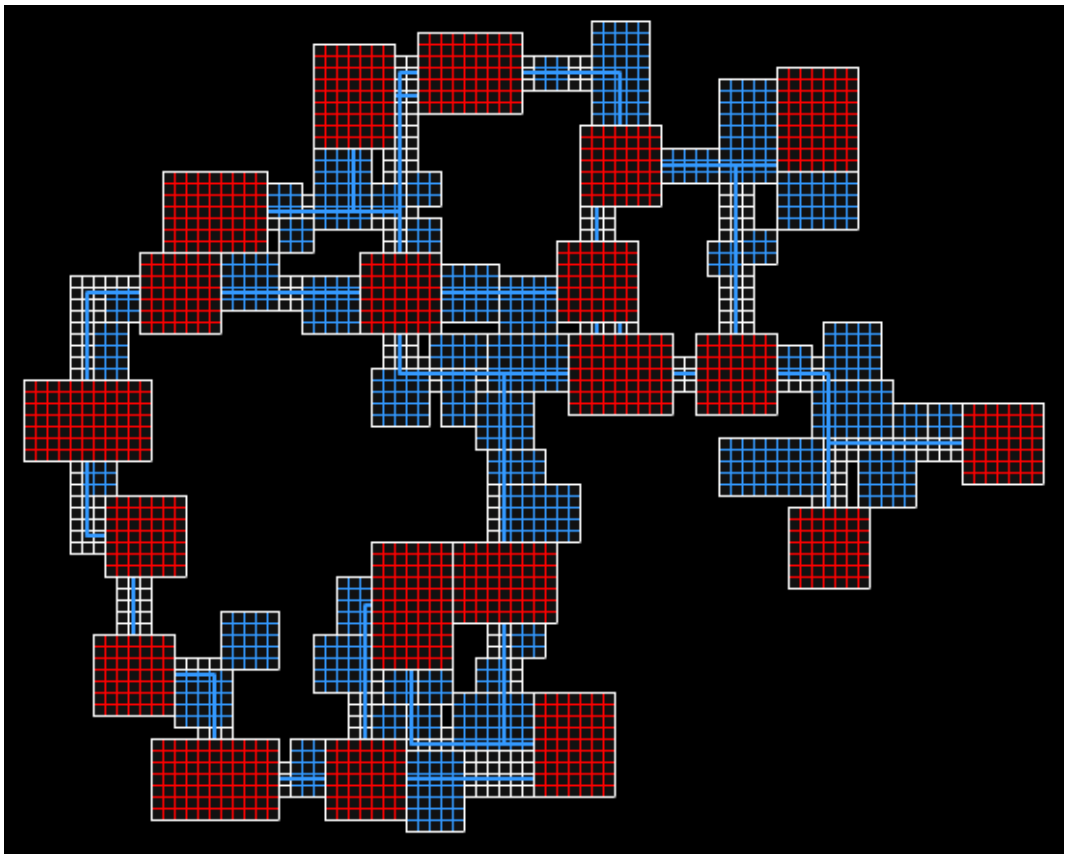


Figure 2.24: A map generated by the algorithm used in *Tiny Keep*. Rooms have red positions, corridors have blue and white positions.

Separation steering behaviour

This algorithm[30] is used by the 3D roguelike game *Tiny Keep*[31] by Phigames (2013). A map generated with this method is shown in Figure 2.24. It places many rectangles of different size randomly in a small area, without caring if they overlap. Then these rectangles are moved using a separation steering behaviour [14] so that at the end of the process none of them overlaps anymore. The gaps that remain between the rectangles are filled putting a *1x1* new rectangle on each of the uncovered positions. Then the rectangles with width and height above a specified threshold are marked as rooms, the other are used later to build corridors.

To decide how to make connections a graph of all rooms' center points is

[30] `http://www.reddit.com/r/gamedev/comments/1dlwc4/procedural_dungeon_generation_algorithm_explained/`

[31] `http://tinykeep.com/`

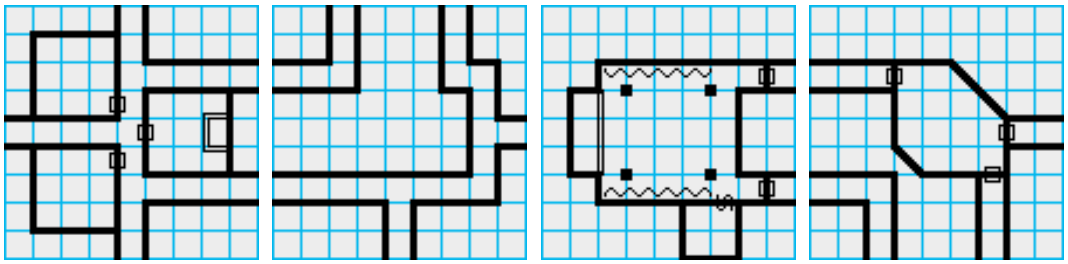


Figure 2.25: Tiles used by the *Inkwell Ideas Random Dungeon Generator*[34].

created using Delaunay triangulation[32], then the minimum spanning tree[33] of the graph is calculated. Then all the edges of the minimum spanning tree are selected, plus a certain percentage of the other edges, randomly chosen. For each of the selected edges a straight or L shaped path is generated between the two related rectangles. Any of the non-room rectangles that intersects the paths is marked as corridor, the remaining unused rectangles are discarded. By doing this the corridors obtained have twisty and uneven edges.

2.3.5 Tile-based approach

This approach uses Wang tiles theory, described by Cohen et al. [5], to compose predefined fragments of map. Wang tiles can be modeled by equal-sized squares with a color on each edge. These tiles can be arranged side by side on a regular square grid so that abutting edges of adjacent tiles have the same color. For map generation, such tiles can be defined as square areas containing a portion of a map, e.g. a room. The edge constraints, instead of being about colors, are about the map features that touch an edge. Each edge must match with an edge of the same type. It is necessary to meet these constraints while arranging the tiles to avoid discontinuities in the map. Figure 2.25 shows tiles that can be used to generate dungeons. There are two kinds of edges: those with a single corridor and those with two corridors.

[32] `http://en.wikipedia.org/wiki/Delaunay_triangulation`
[33] `http://en.wikipedia.org/wiki/Minimum_spanning_tree`
[34] `http://www.inkwellideas.com/roleplaying_tools/random_dungeon/`

Figure 2.26: A set of tiles used in *Diablo 3*.

Figure 2.27: Possible instances of the same dungeon in *Diablo 3*.

Map generation in *Diablo 3*

The RPG game *Diablo 3* uses a tile-based approach for the generation of maps, which are dynamically built each time a new game is started. Not all the areas of the game, however, are completely procedural: many of them mix static and procedural content. Generally, statically defined areas are those important for the plot, containing scripted events or ingame cinematics. Figure 2.26 displays one of the sets of tiles used in the game. Figure 2.27 shows different generated instances of the same dungeon; it can be noted that the map size can vary in a rather wide range. Figure 2.28 shows an area of the game which is statically defined, apart from the highlighted parts which are procedurally filled with tiles.

Herringbone Tiles

When generating maps with Wang tiles an artifact can arise from the grid layout. Edges are adjacent to each other in a straight line so, if they have particular features, that straight line may be visible when zoomed out. To solve this problem Barrett [3] proposes an approach which he calls *herringbone tiles*. Herringbone tiles use rectangles with a horizontal-to-vertical ratio of 1:2 and 2:1. Figure 2.29 shows the herringbone pattern.

Figure 2.28: A map of an area of *Diablo 3*. Blue regions are procedurally filled using tiles, while the rest is static.

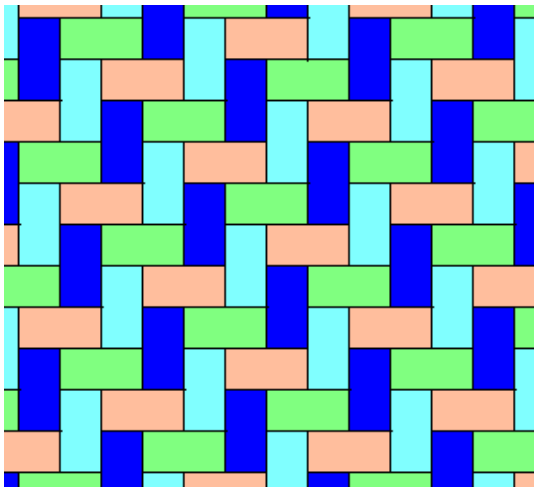


Figure 2.29: The *herringbone tile* pattern.

An advantage of herringbone tiling over square tiling is that, since the tile edges are broken up by the herringbone pattern, the occurrence of artifacts due to the alignment of edges is limited. Another possible solution to the problem is to use hexagonal tiles. However herringbone tiling is more suited, for example, for the generation of cities, where a rectangular layout is more appropriate.

2.3.6 Cellular Automata

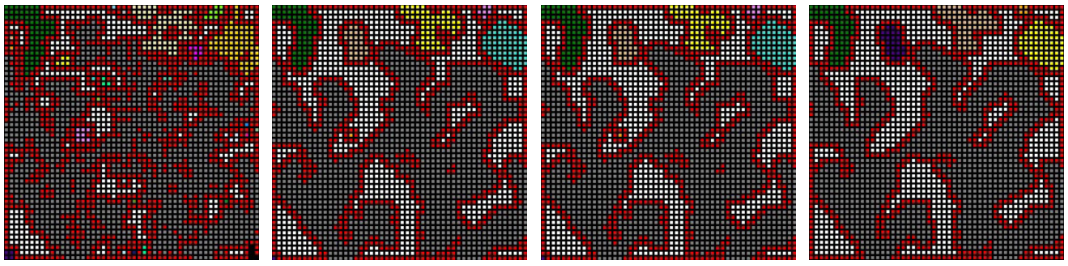
A cellular automaton[35] consists of a regular grid of cells, each in one of a finite number of states, such as *on* and *off*. At each iteration a fixed rule is applied, determining the new state of each cell in terms of its current state and of the current state of its neighbor cells. An application of cellular automata is to generate cave-like structures. In the following we present an algorithm suited to this purpose, described by Johnson et al. [7].

A cell-based representation of the map is used, each square cell can contain either floor or rock. Starting from a grid full of floor cells, the 50% of them, randomly selected, is filled with rock. The adopted rule is that, at each iteration of the algorithm, a cell becomes rock if at least 5 of its 8 neighbors are rock, otherwise the cell becomes floor. Intuitively, the rule is designed to prune the sparse areas of the grid, that are those with isolated rock cells, and to fill in the dense areas by patching up holes. The number of iterations has an impact on the average width of the caves generated; on average, the higher the number the wider the cave. Figure 2.30 shows the state of a grid after some iterations of the algorithm. The resulting map can be conceived as a system of caves.

One of the problems[36] of this approach is that it is prone to generate disconnected maps, i.e. maps with disconnected floor regions. One way to deal with this problem is to tweak the initial percentage of rock, the rule and the number of steps so that the size and amount of regions isolated from the biggest floor region is statistically low. These isolated areas can then be

[35] http://en.wikipedia.org/wiki/Cellular_automaton

[36] http://roguebasin.roguelikedevelopment.org/index.php?title=Cellular_Automata_Method_for_Generating_Random_Cave-Like_Levels



(a) $n = 1$ (b) $n = 2$ (c) $n = 3$ (d) $n = 4$

Figure 2.30: The state of a cellular automaton after n iterations. Rock cells are white, while each isolated floor region has a different color. Wall cells, which are rock cells with at least one floor neighbor, are red.

discarded. Another solution is to create floor paths to connect the disjoint regions, even though this may lead to unnaturally looking corridors.

2.3.7 Mazes

A maze is a tour puzzle in the form of a complex branching passage through which the solver must find a route. Mazes are a classic feature of roguelike and adventure games. Since the player can remember their solution, by generating them randomly at each new game the replay value is greatly improved.

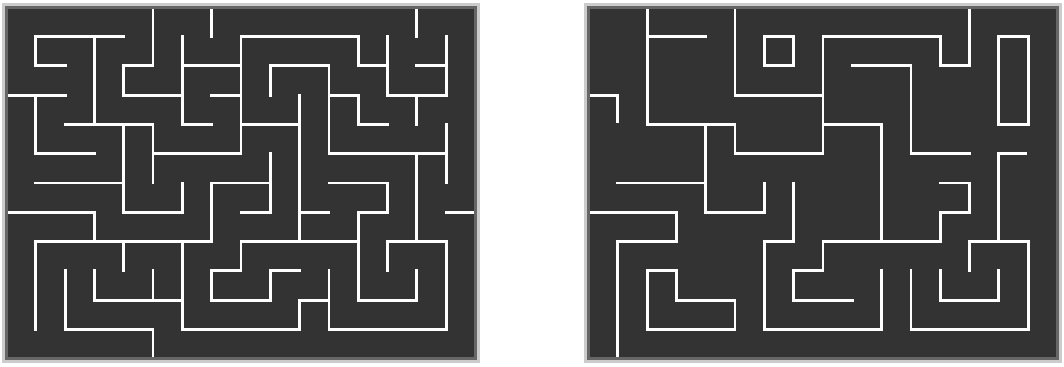
To represent mazes it is often used a particular discrete representation, in which the inside of positions is always walkable floor and walls are put on the edges of the positions. Most maze generation algorithms produce perfect mazes, i.e. mazes which have one and only one path from any point in the maze to any other point. Therefore perfect mazes have no inaccessible sections, no circular paths, no open areas. Figure 2.31 shows a perfect maze and a not-perfect maze. A perfect maze can be made not-perfect by removing walls.

Growing tree algorithm

The growing tree algorithm is one of the most popular for perfect maze generation. It performs the following steps:

1. Initially a wall is put on every edge of the positions.

2. Let C be a list of positions, initially empty. A position is chosen at random and it is added to C.



(a) A perfect maze.

(b) A not-perfect maze.

Figure 2.31: Two kinds of maze.

3. A position is chosen from C. One of the walls that are between the position and an unvisited neighbor of that position is removed, and that neighbor is added to C as well. If there are no unvisited neighbors, the position is removed from C.

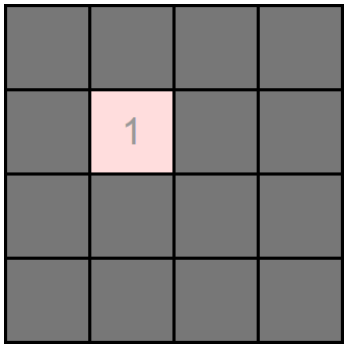
4. Step 3 is repeated until C is empty.

Figure 2.32 shows examples of steps of the algorithm. Results vary depending on which policy is used to choose the position at step 2. If it is chosen the most recently added position, the algorithm behaves similarly to the recursive backtracking maze generation algorithm[37]. Instead, if the position is randomly chosen, the algorithm becomes similar to Prim's maze generation algorithm[37].

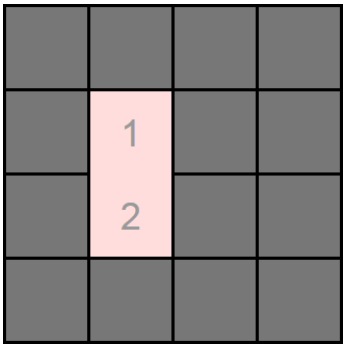
2.4 Summary

In this chapter we have dealt with procedural content generation in games, discussing how it have been used in past and how is used today. We have made a distinction between indoor and outdoor environments and we have described approaches for the generation of both.

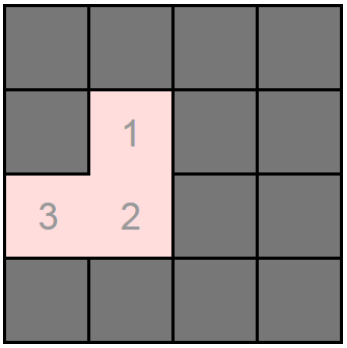
[37] http://en.wikipedia.org/wiki/Maze_generation_algorithm



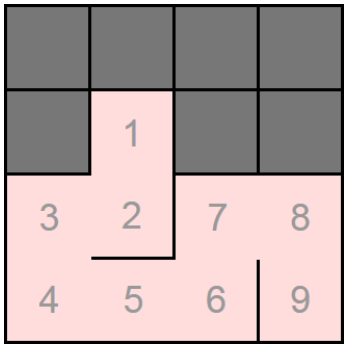
(a) A random position is added to C.



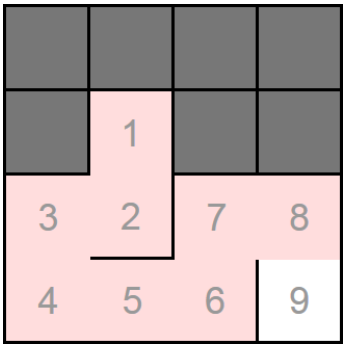
(b) An unvisited neighbor of position 1 is added to C, the adjacent wall is removed.



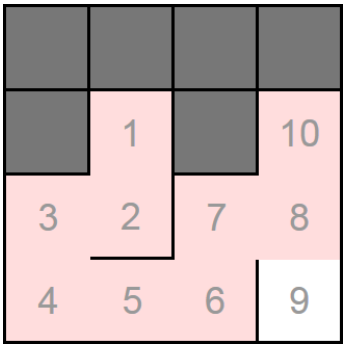
(c) An unvisited neighbor of position 2 is added to C, the adjacent wall is removed.



(d) After 6 iterations.



(e) Position 9 has no unvisited neighbors, it is removed from C.



(f) An unvisited neighbor of position 2 is added to C, the adjacent wall is removed.

Figure 2.32: Examples of steps of the growing tree maze generation algorithm. C is the list used internally by the algorithm. The *most recently added* policy is used when choosing a position from C.

Chapter 3

In Verbis Virtus

In this chapter, we describe *In Verbis Virtus*, a fantasy game in which players cast spells using their actual voice. We give details about the plot and the gameplay, discussing in particular spells, puzzles and monsters that are in the game.

3.1 The game

In Verbis Virtus[1] (IVV) is a first person game with a fantasy setting in which players impersonate a wizard. The game uses a speech recognition system that allows players to cast spells by pronouncing magic words with their own voice. We started the project during the *Videogame Design and Programming* course at the *Politecnico di Milano* and we are currently developing it at Indomitus Games. In IVV players must make their way through a misterious temple, from which they learn inceasingly powerful spells as they advance. The game includes puzzle and combat situations: the temple hides many obtacles and dangers, so players must use the spells to solve puzzles, overcome traps and defeat enemies. IVV is developed with the *Unreal Development Kit*[2] (UDK) by Epic Games, a version of the *Unreal Engine 3* made for indie game development.

[1] `http://www.inverbisvirtus.com/`

[2] `http://www.udk.com/`

Figure 3.1: The entrance of the temple of IVV.

3.2 Plot

IVV is set in a fantasy world. The protagonist of the game, Adam, is an archeologist conducting researches on buildings and artifacts left by an misterious ancient civilization. He has partecipated in many excavations, together with his lover, Leif. Magic is unknown in the world Adam lives in, however his studies lead him to think that the creators of those objects possessed supernatural powers. A newly discovered archeological site reveals a huge amount of objects and inscriptions, like Adam has never seen before. There is so much material that he even finds clues about the location of a temple, which appears to be the source of the extraordinary powers. Both him and Leif are overjoyed for the discovery, which can lead to the solution of the mysteries about the ancient civilization. However, after a short time, Leif dies for no apparent reasons. Adam is shocked and, since he can't find any other explanation, he is convinced that the newly discovered artifacts are involved with Leif's death. In confusion and dispair, he hastily goes on the road to the temple, hoping to find answers. His travel is extremely difficult, but in the end, in the midst of a desert, he reaches his destination (Figure 3.1).

In the temple, Adam finds inscriptions that teach him the mystic knowledge of its ancient constructors. He also comes in contact with Veritas (Figure 3.2), a supernatural entity that guides the visitors of the temple. Thanks to Veritas

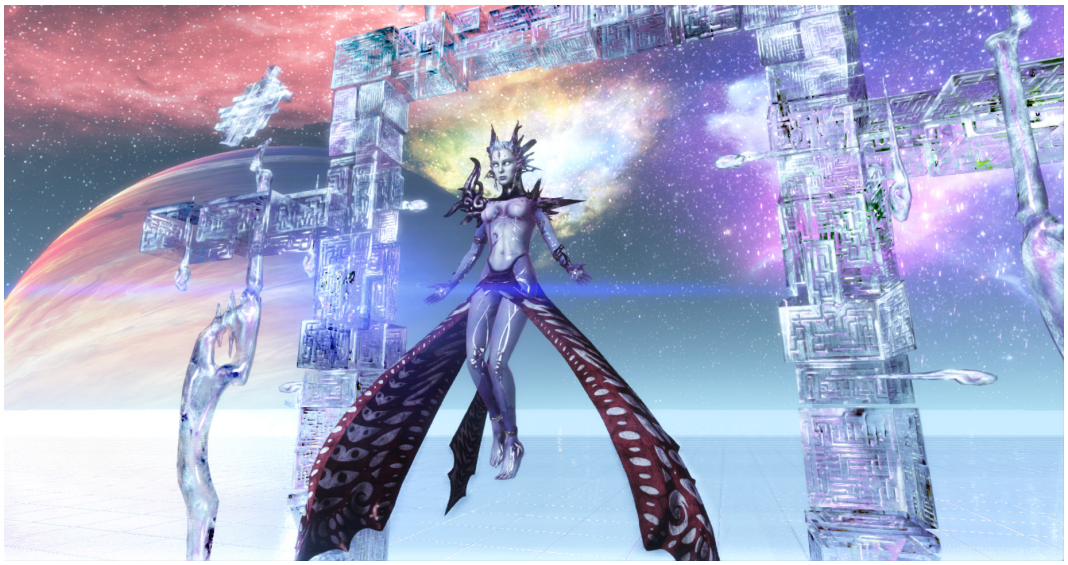


Figure 3.2: A screenshot of Veritas, the supernatural entity that guides the visitors of the temple.

and his studies as an archeologist, he understands very easily the language of the constructors, and masters very quickly the arcane arts. Adam soon finds that the temple also hides many obstacles and dangers, and he has to use his new powers at his best to overcome them. Learning more and more powerful spells, he makes his way into the depths, at the price of facing increasingly dreadful difficulties. This, however, does not affect his resolution to uncover the innermost secrets of the temple.

3.3 Gameplay

IVV takes place mainly in dungeon-like indoor environments. The protagonist is controlled with mouse and keyboard, plus a microphone for speech recognition. Gameplay is based on the use of magic abilities: almost all interactions with the environment are carried out using spells that the player must cast with his own voice. Players acquire new spells as they progress in the game. Challenges include puzzles and combats against monsters that try to kill the protagonist. In each situation, players have to use the spells they learned up to that point to get to a solution, or to defeat the enemies.

3.4 Spells

As the title suggests ("in verbis virtus" is the latin for "power is in words"), the game's main focus is on casting spells with voice. Speech recognition is achieved with the *CMU Sphinx*[3] open source toolkit. Spells are cast pronouncing magic formulas, which are revealed throughout the game. Players can choose between different languages for the magic formulas, including english and a fantasy language created specifically for the game. Spells have a formula to cast them, and can have additional formulas that trigger secondary effects. When spells are acquired, their descriptions are noted on character's journal. The journal also allows players to hear the right pronunciations of formulas. The game features several spells such as:

- The *light spell* creates a light sphere that illuminate the nearby area.
- The *light beam spell* produces a ray of white light that can be filtered through crystals to obtain different light colors (Figure 3.3). Light beams are used to activate special objects, called light beam receptors.
- The *command spell* activates objects marked with special runes, called *marks of command*.
- The *telekinesis spell* moves objects and can be used in combat to push enemies.
- The *teleport spell* takes the character instantly to the location he is aiming at.
- The *mark of fire spell* places fire runes on surfaces and monsters. An additional formula makes fire runes explode.
- The *shield spell* creates a barrier around the character that blocks incoming projectiles. An additional formula throws back blocked projectiles.

3.5 Puzzles

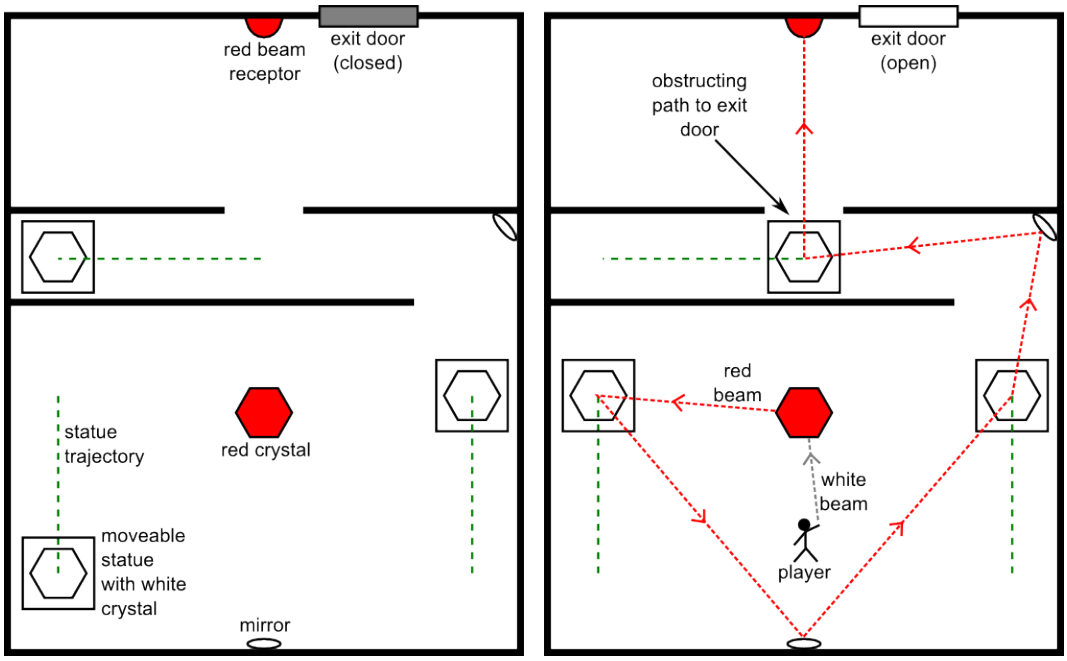
Puzzles in IVV are mainly situations where an obstacle, e.g. a closed door, prevents players from advancing in the level. Players have to figure out how

[3] `http://cmusphinx.sourceforge.net/`

Figure 3.3: The ray of the *light beam spell*, deflected by a crystal.

to use their spells to remove or bypass the obstacle. Very simple puzzles are placed immediately after locations where new spells are acquired. Such puzzles have the purpose to let players practice with the abilities they just learned. For instance, near the location where the *telekinesis spell* is obtained, there is a passage blocked by collapsed pillars; players have to use the telekinesis to remove the pillars before they can go through it.

More difficult puzzles require more complex actions and the use of multiple spells. For example, one of the featured puzzles requires to hit a light beam receptor with a red beam to open a door; schemes of the puzzle are shown in Figure 3.4. In this situation, there are three white crystals, which can be moved with the *command spell*, and a red crystal, which is fixed. Players have to position white crystals correctly so that, when they hit the red crystal with the *light beam spell*, the resulting red beam is deflected toward the receptor, opening the exit door. This puzzle tests thinking abilities of players, because they have to figure out how to open the door. Once open, the door starts to close slowly and players have to run to reach it before it closes completely. While running, players have to quickly move the white crystal obstructing the path to the door (Figure 3.4b) with the *command spell*, so spell casting skills of players are also tested.



(a) Initial situation (b) Correct positions of the crystals

Figure 3.4: Schemes of a puzzle. Figure (a) shows the initial situation. Figure (b) shows the correct positions of white crystals to take a red beam to the receptor, thus opening the exit door. White crystals are placed on statues bearing *marks of command*, which can be moved along the shown trajectories with the *command spell*.

Figure 3.5: Ingame screenshot of the situation shown in Figure 3.4b. White crystals are in the correct position and the red light beam reaches the receptor.

3.6 Monsters

The game includes several monsters, such as:

- The *savage* (Figure 3.6) is a small monster that uses as weapons wrist blades and tentacles that are on its back. It is weak when alone, so it is often found in groups.
- The *beast* (Figure 3.7) is a huge monster capable of deadly charges against the player. It very strong and hard to kill.
- The *winged glory* is a statue animated by magic that flies thanks to mechanical wings. It attacks throwing projectiles at the player.
- The *ignis fatuum* is a small ghost which resembles a ball of fire. It can pass through walls and cannot be damaged. When it approaches, players have to stand still and interrupt all spells, otherwise it deals magic attacks.

For the navigation of monsters that move on the ground we use the *navigation mesh*[4] system included in UDK, which automatically generates representations of levels that are used for pathfinding. This system can only manage planar representations of maps, so for the navigation of flying monsters we use manually placed waypoints[5].

3.7 Summary

We have described *In Verbis Virtus*, a fantasy game that uses a speech recognition system to let players cast spells with their voice. We have described the plot of the game and we have given details about the gameplay. Then, we have described spells included in the game and we have given examples of how they are used to solve puzzles. Finally, we have described monsters that players have to face in the game.

[4] `http://udn.epicgames.com/Three/NavigationMeshReference.html`
[5] `http://udn.epicgames.com/Three/UsingWaypoints.html`

Figure 3.6: A savage, a small monster that attacks the player with blades and tentacles.

Figure 3.7: A beast, a huge monster capable of deadly charges against the player.

Chapter 4

Floor plan generation

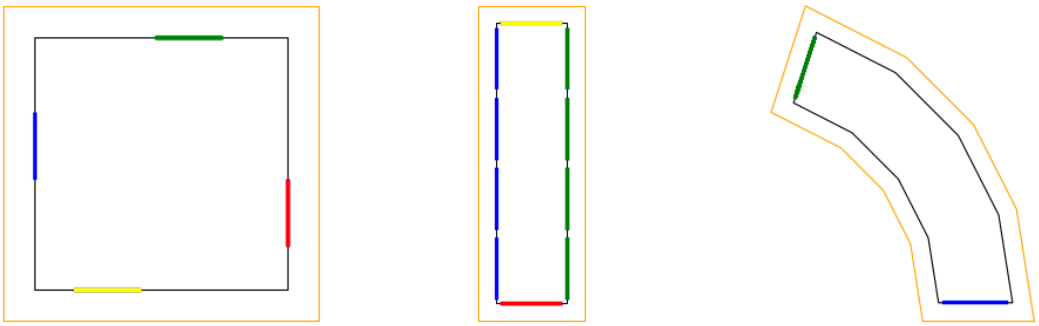
In this chapter, we describe our algorithm for the generation of dungeons floor plans in *In Verbis Virtus*, that are subsequently used to create the actual in-game maps. First, we outline our approach and describe its major features. Then we describe our algorithm in detail discussing its four main phases.

4.1 Our approach

Our algorithm generates dungeons for *In Verbis Virtus*. It combines the digger approach with the uniform approach and uses predefined architectural elements to generate a map. In particular, it extends the digger and uniform approaches into a continuous representation of the map. Our algorithm starts with one or more architectural elements given as input. Similarly to the digger approach, it repeatedly adds architectural elements making the map grow in a tree-like fashion. Then, similarly to the uniform approach, it adds corridors to transform the tree-like map into a more general topology.

4.2 Architectural element properties

Our algorithm employs several architectural elements to generate dungeon maps. Figure 4.1 shows three basic architectural elements (a room and two corridors) and highlights their properties. Each element is defined by the following properties:



(a) Rectangular room (b) Linear corridor (c) Curved corridor

Figure 4.1: Basic architectural elements. Floor perimeter is black, shell perimeter is orange, connectors of the same group have the same color.

The floor is a polygon that defines the walkable area of the architectural element. Walls and doors of the architectural element are placed on the perimeter of the floor. In Figure 4.1 floors are shown as black polygons.

The shell is a polygon that delimits the area covered by the assets of an architectural element. The shell is bigger than the floor and covers it completely. The assets are placed after the creation of the floor plan, but the area they occupy must be known when defining the shell. The assets can be partially or completely outside the floor area. The shell takes this into account and it is used by the algorithm to avoid that the assets of an element compenetrate another element. We do not calculate the shell so that it covers exactly the area taken by the assets, it is enough an approximation that covers at least that area. In Figure 4.1 shells are shown as orange polygons. The shell is calculated by applying a polygon offset operation[1] on the floor polygon.

Connectors are objects located on the perimeter of the floor where another architectural element can be attached. Connectors have several properties, shown in Table 4.1. The *base* property indicates the architectural element the connector belongs to. The *location* property indicates the position of the connector on the floor perimeter. The *linkedTo* property indicates the connector this connector is linked to. It is set to *None* when the connector is "unused", i.e. it is not linked to any connector. When two connectors are

[1] `http://www.cgal.org/Manual/latest/doc_html/cgal_manual/Straight_skeleton_2/Chapter_main.html`

Property	Type	Description
base	architectural element	The architectural element the connector belongs to.
location	2D vector	The position of the connector, must lie on the perimeter of its *base*.
linkedTo	connector or *None*	The connector this connector is linked to, or *None* if it is not linked.
requiredOffset	float	The distance by which the assets of the *base* protrude beyond the side of the floor the connector lies on.
normal	2D vector	The outward normal vector of the floor perimeter in the location of the connector.

Table 4.1: Connector properties.

linked together their *linkedTo* properties are set accordingly; the two connectors are thus "used". In the 3D map used connectors become doorways, while unused connectors are ignored and walls replace them. All the doorways have the same width. In Figure 4.1 connectors are represented as segments placed on the perimeter. When a connector is used, no asset is placed in front of it. The *requiredOffset* property must be set to the maximum distance by which the assets protrude beyond the side of the floor the connector lies on, as shown in Figure 4.2. *requiredOffset* is thus the minimum distance at which another architectural element can be placed to avoid compenetrations of the assets. It is usually set equal to the distance between the connector and the perimeter of the shell. Even if the floor of an architectural element can be a concave polygon, the connectors must be on the sides of the floor that lie on its convex hull. This is to simplify the overlap checks needed when the algorithm connects two architectural elements.

Groups of connectors form a partition of the set of connectors. Connectors are grouped depending on their position on the perimeter of the floor, i.e. connectors with similar positions are grouped together. In these architectural elements connectors that lie on the same side of the floor are in the same group. When the algorithm has to randomly choose a connector to attach a new architectural element, it chooses a group first, and then one of its connectors. This allows a more homogeneous selection of the connector than choosing

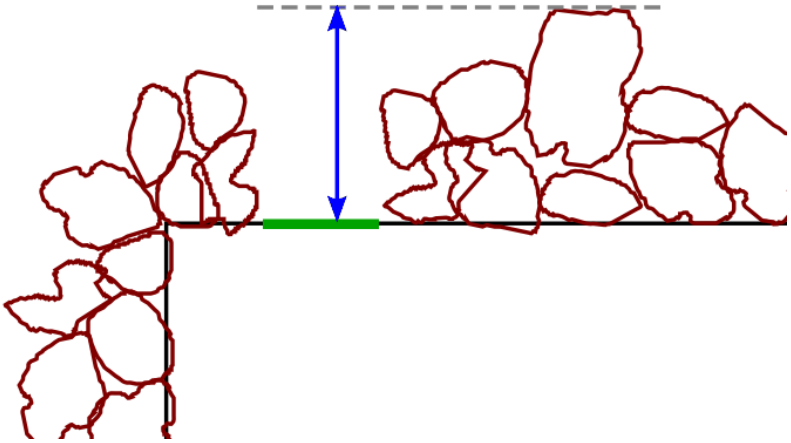


Figure 4.2: A used connector of a rectangular room decorated with rock assets. The double arrow indicates the *requiredOffset* distance of the connector.

uniformly between all connectors. If the random selection was made uniformly among connectors, for instance for the linear corridor (Figure 4.1b) it would be more likely to choose a lateral connector than a connector on one end.

4.2.1 Architectural element classes

We do not define architectural elements with entirely fixed characteristics; instead, we define generators of architectural elements. Each generator creates architectural elements of a class. Basic architectural element classes are:

The rectangular room is a room with rectangular shape, which has a connector on each side (Figure 4.1a). Width and height of the rectangle, and positions of the connectors on the sides are randomly chosen. It has four groups of connectors, one for each connector. All connectors have a *requiredOffset* equal to the distance between their location and the perimeter of the shell.

The linear corridor is a straight corridor, with a connector on each end and a row of connectors on each lateral wall (Figure 4.1b). The length is randomly determined. The lateral connectors have a *requiredOffset* equal to the distance between their location and the perimeter of the shell. Instead, the connectors on the two ends have a *requiredOffset* equal to zero, indicating that, if the connector is used, the assets of the architectural element are guaranteed not to protrude beyond the two ends of the floor. This fact proves useful when

the algorithm connects architectural elements together. The shell, however, goes beyond the two ends, because if an ending connector is not used a wall is created over it, with its assets protruding beyond the floor perimeter. The linear corridor has four groups of connectors, one for each side of the floor.

The curved corridor is an arc-shaped corridor with a connector on each end (Figure 4.1c). Angle and radius of the arc are randomly determined. The curved corridor has two groups of connectors, one for each connector. All connectors have a *requiredOffset* equal to the distance between their location and the perimeter of the shell.

4.3 Generating the plan

The algorithm starts with a set of architectural elements given as input and generates the plan in four phases: (i) in the *digging phase* it adds architectural elements in a way similar to the digger generator; (ii) in the *additional connection creation phase* it adds corridors to create more connections between the elements; (iii) in the *dead end pruning phase* it eliminates a portion of dead end corridors; (iv) in the *isolated parts pruning phase* it eliminates isolated parts of the map. The parameters that the algorithm takes as input are shown in Table 4.2.

4.3.1 *Digging phase*

The architectural elements in the *initialElements* parameter are placed in the map. Even if externally defined as parameters, these architectural elements must follow compenetrations and connections constraints. In this phase the algorithm repeatedly adds architectural elements to the map. For this purpose, it chooses a class from the $prob_{classes}$ parameter and generates an architectural element of that class. Then it chooses an unused connector of an architectural element of the map and finally tries to connect the new architectural element to this unused connector. The connection process can fail, which happens when the new architectural element, if placed, would overlap some element. For this reason a backtracking approach is used to find a suitable way to add architectural elements.

Name	Type	Description
$initialElements$	set of architectural elements	As first operation, the algorithm puts these architectural elements in the map. Cannot be empty.
$prob_{classes}$	dictionary (architectural element class, float)	Architectural element classes associated to probability values.
$prob_{degrees}$	dictionary (integer, float)	Degree values associated to probability values. The degree of an element is the number of its used connectors.
$maxRooms$	integer	Maximum number of rooms in the map.
$boundary$	polygon	Constrained area in which the map is built.
$connWnd_{start}$	float	Between 0 and 1.
$connWnd_{end}$	float	Between 0 and 1.
$connWnd_{use}$	float	Between 0 and 1.
$connMaxDist$	float	Maximum length of additional connections.
$deadEnds_{\%}$	float	Fraction of dead end corridors kept in the plan.

Table 4.2: Parameters of the plan generation algorithm.

Backtracking

The backtracking[2] methodology gets to a candidate solution of a problem through a sequence of choices. If the candidate solution is valid, then the problem is solved, otherwise the solution is discarded and for each choice all the other options are subsequently considered, from the last to the first choice. Even a partial solution is discarded if it cannot lead to a valid complete solution.

In our case a candidate solution is composed by a new architectural element and a connector to connect it to. The candidate solution is valid if the new architectural element can be successfully connected. Otherwise, if the connection process fails, the algorithm backtracks and tries other options of the choices it made, thus obtaining new candidate solutions. Our algorithm adopts the first valid solution it finds.

[2] `http://en.wikipedia.org/wiki/Backtracking`

Choices of our algorithm

The choices our algorithm makes to determine how to connect the new architectural element are the following:

1. The class of the new architectural element, randomly chosen according to the $prob_{classes}$ parameter, which is a dictionary that maps classes to probability values.

2. A degree D (the degree of an architectural element is the number of its used connectors). D is randomly chosen according to the $prob_{degrees}$ parameter, which is a dictionary that maps degree values (integer type) to probability values.

3. An architectural element E, randomly chosen among those that have degree D. Some elements can have a degree that does not appear in $prob_{degrees}$, in that case they are considered as if they have the nearest degree present in $prob_{degrees}$. The algorithm backtracks if there are not element with degree D.

4. A group of connectors, randomly chosen from the groups of connectors of E. Only the groups without any used connector are considered in this choice, the algorithm backtracks if the selected element has no such groups.

5. A connector C_E, randomly chosen from the selected group of connectors.

A new architectural element F of the selected class is generated. At this point all the choices are made, so the algorithm tries to connect F to C_E.

Linking connectors

We now describe the process to link two connectors, used to join a new architectural element. To link a given connector C_A to a connector C_B, let A and B be the bases respectively of C_A and C_B, A is placed in the map so that C_A overlaps C_B and with a rotation such that the normals of C_A and C_B face themselves, as shown in Figure 4.3. Then the following conditions are checked:

- The shell of A is completely inside the polygon specified by the *boundary* parameter.

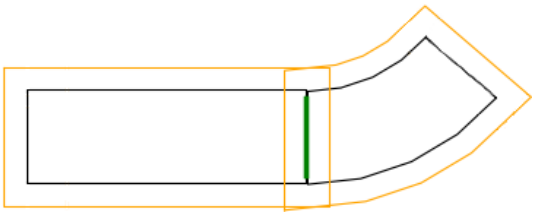


Figure 4.3: A linear corridor linked to a curved corridor. Only used connectors are shown.

- The shell of A does not overlap any other shell, apart from B's shell.

If both conditions are met the linking process succeeds and C_A and C_B are set as linked to each other. Otherwise the linking process fails and A is removed from the plan.

To link a given connector C_A to a connector C_B ignoring the architectural element Z, the same linking process just explained is used, except that in the overlap check, beside B's shell, Z's shell is ignored as well.

Connecting the new architectural element

The algorithm tries to connect a connector C_F of the new architectural element F to the selected connector C_E. If F is a corridor, C_F is randomly chosen among its ending connectors; otherwise it is randomly chosen among all the connectors of the architectural element. If at least one of the *requiredOffset*s of C_E and C_F is zero, then the algorithm tries to link C_E and C_F. If the linking process succeeds, F becomes an element of the plan. Instead, if both C_E and C_F have a non zero *requiredOffset*, they cannot be linked directly, because compenetrations may arise. A linear corridor L, with length equal to the sum of the two *requiredOffset*s, is created and one of its ending connectors is linked to C_E. C_F is then linked to the other ending connector of L. By doing this E and F are placed far enough from each other to avoid compenetrations (Figure 4.4). If both linking processes succeed, L and F become elements of the plan. Otherwise, if L fails to link, F is not placed; while if F fails to link, L is removed from the plan as well. If F is not added to the plan, the algorithm backtracks, changing the choices it made, until a new architectural element is successfully connected. The *digging phase* adds new architectural elements until at least one of the following conditions is met:

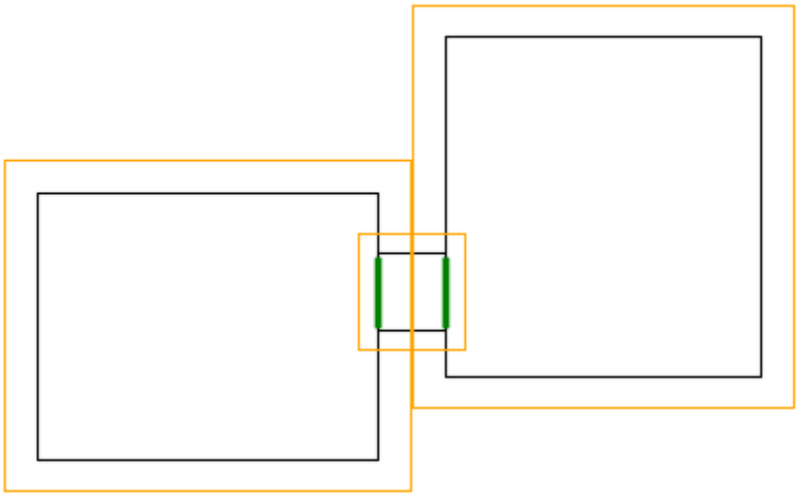


Figure 4.4: The result of connecting two rooms together: a linear corridor is placed between them to avoid compenetrations of the assets. Only used connectors are shown.

- The plan have a number of rooms equal to the *maxRooms* parameter.
- The backtracking process tries all options, but a new architectural element cannot not be added.

The latter condition is usually met when there is no more space in the *boundary* to add a new architectural element.

4.3.2 *Additional connections creation phase*

The *digging phase* adds architectural elements and connect them to a single element. For this reason, digging cannot create loops. Moreover if *initialElements* contains disconnected architectural elements, the *digging phase* cannot connect them. Unlike the *digging phase*, the *additional connection creation phase* adds corridors connecting two elements.

The *additional connection creation phase* first creates a list of all the unordered pairs of unused connectors of the elements. Then the list is sorted with respect to the shortest path distance between the connectors of each pair. For this purpose the algorithm builds a graph of the connectors (see Table 4.3), then calculates the shortest path distances using Dijkstra's algorithm[3]. A window of the ordered list (i.e. a contiguous portion of the list) is then selected

[3] `http://en.wikipedia.org/wiki/Dijkstra's_algorithm`

The graph of connectors G_{conn} of a map M is created performing the following steps:

1. Let G_{conn} be an empty graph.
2. For each architectural element E of M:
   (a) For each used connector C of E a node N_C is added to G_{conn}. Edges are created so that all these nodes are connected to each other. Each edge (N_A, N_B) is set a weight equal to the euclidean distance between connectors A and B.
3. For each pair (A, B) of linked connectors of M, an edge (N_A, N_B) with null weight is added to G_{conn}.

Table 4.3: Procedure to create the graph of connectors.

and some of the pairs in the window are randomly chosen. The parameters $connWnd_{start}$ and $connWnd_{end}$ indicate the start and the end of the window, with respect to the length of the list. The parameter $connWnd_{use}$ indicates the fraction of the pairs in the window to randomly choose. For each of the chosen pairs, if the euclidean distance between the two connectors is lower or equal than $connMaxDist$, then the algorithm tries to create a connection between them. The following steps are performed to join the two connectors (see Figure 4.5):

1. Let C_1 and C_2 be two connectors and F_1 and F_2 their *base*s.
2. If C_1 has a non zero *requiredOffset*, a linear corridor with length equal to the *requiredOffset* is linked to it, as shown in Figure 4.5b. In that case, let C_1' be the unused ending connector of the new linear corridor, otherwise let C_1' be equal to C_1. If the linking fails, the new corridor is removed and the process is aborted.
3. The same operations done for C_1 in the previous step are done for C_2, thus C_2' is defined similarly to C_1'.
4. Two curved corridors are linked respectively to C_1' and C_2', ignoring respectively L_1 and L_2. The arrangement of these two corridors is shown in Figure 4.5c. These two curved corridors have minimum radius and angle such that their unused connectors have normals that lie on the

same line, but have opposite verse. If the linking fails, the process is aborted and the corridors linked in the previous steps are removed from the plan. The process is aborted even if the angle of at least one of the two corridors is grater than 90 degrees. This is to simplify the checks needed to avoid overlap between architectural elements. Let C_1'' and C_2'' be the two unused connectors of the new curved corridors.

5. A linear corridor is linked to C_1'' and C_2'', as shown in Figure 4.5d. If the linking fails the process is aborted and the corridors linked in the previous steps are removed from the plan.

If the procedure terminates without a failure, the result is a sequence of corridors connecting C_1 to C_2 (Figure 4.5d).

4.3.3 *Dead end pruning phase*

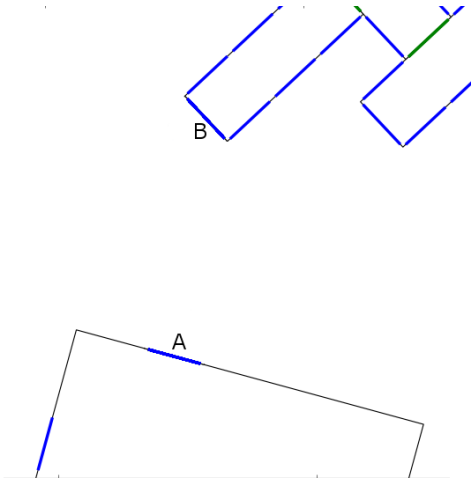
This phase removes a portion of dead ends from the plan, i.e. corridors with degree 1, by performing the following steps:

1. Given the number of dead ends N_D, we compute the number of dead ends to keep as $N_K = deadEnds_{\%} \times N_D$.
2. A set K is created randomly choosing N_K dead ends.
3. Dead ends that are not in K are removed from the plan.
4. Corridors that were linked to the removed dead ends may have become dead ends. Step 3 is repeated until there are no dead ends to remove.

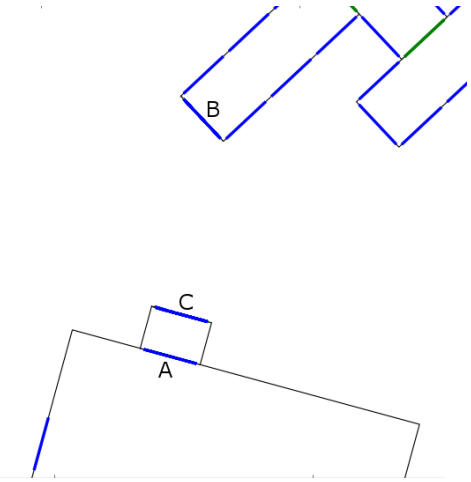
After the execution, only the dead ends in K are still in the plan. An example of the process is show in Figure 4.6.

4.3.4 *Isolated parts pruning phase*

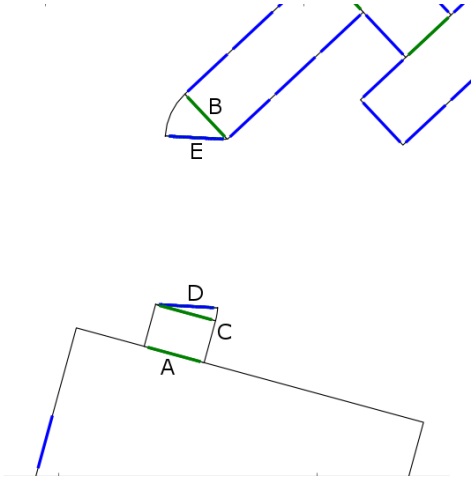
If the plan has disconnected parts, the algorithm keeps the biggest part (i.e. the one with the greatest area, calculated summing the areas of the floors of its elements) and removes the other parts.



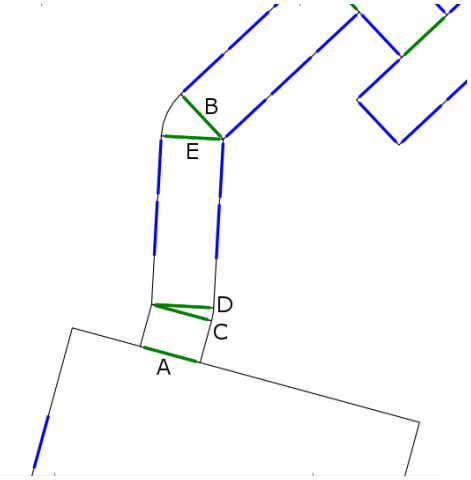
(a) A and B are the two connectors to be joined.



(b) A linear corridor is linked to A, which have a non zero *requiredOffset*. It is not needed for B, which have a *requiredOffset* equal to zero.

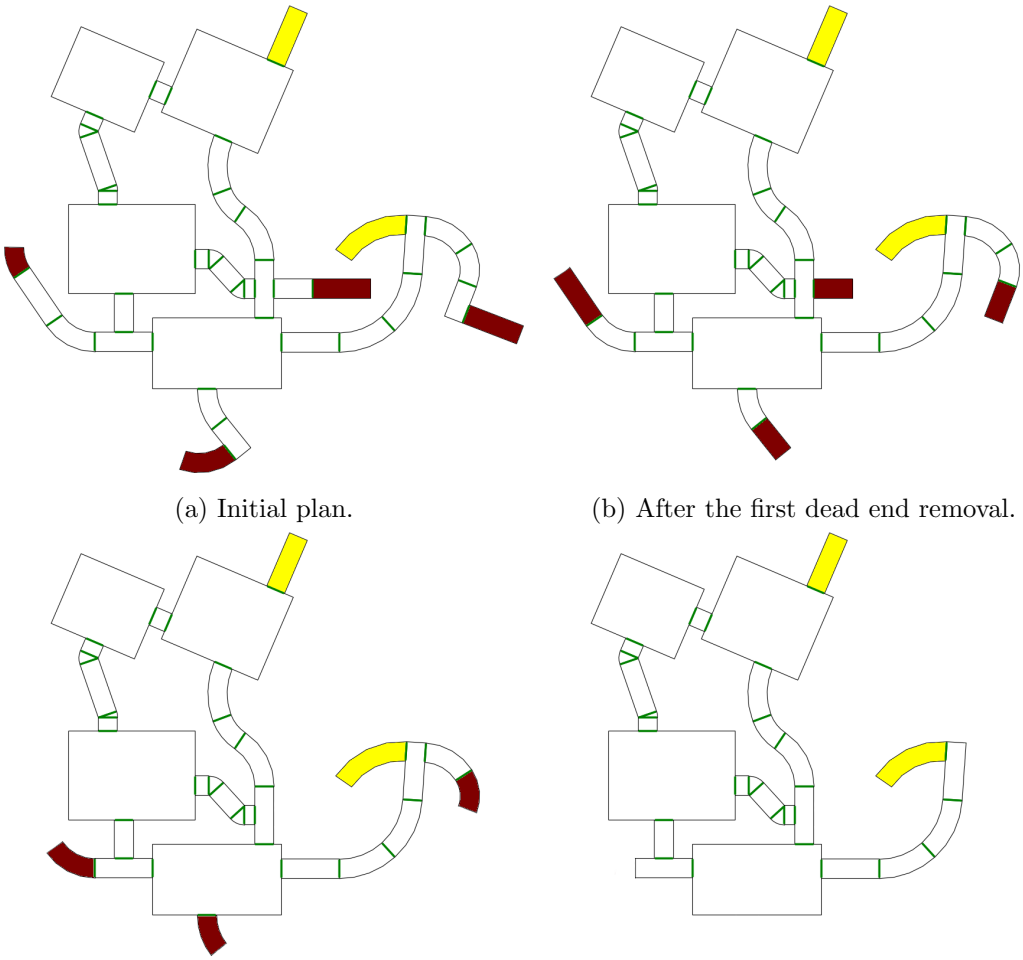


(c) Two curved corridor are linked to B and C so that their unused connectors are facing each other.



(d) A linear corridor is connected to D and E.

Figure 4.5: Steps of the creation of an additional connection.



(a) Initial plan.

(b) After the first dead end removal.

(c) After the second dead end removal.

(d) After the fourth dead end removal, there are no more dead ends to prune.

Figure 4.6: Steps of the *dead end pruning phase*. Dead ends to keep are yellow, dead ends to remove are red.

4.4 Summary

In this chapter, we have described our algorithm for the generation of dungeons floor plans in *In Verbis Virtus*. First we have outlined our approach, which combines the digger approach with the uniform approach and uses a continuous representation of the map. Then we have presented architectural elements, which are used by the algorithm to generate the map. Finally we have described our algorithm in detail discussing its four main phases.

Chapter 5

Evaluation of floor plans

In this chapter, we briefly discuss relevant applications of map evaluation. Then, we introduce the metrics we used to analyze the floor plans generated by our algorithm. Finally, we describe the evaluation procedure we applied to the generated maps, and we show the results.

5.1 Map evaluation

To evaluate maps with respect to desired features, it is crucial to choose suitable measures. McGuinness [10] defined several metrics to analyze maps created by different generators. He showed that representation choice in automatic map creation influences the features of generated maps. To compare algorithms using different representations, he collected data from generated maps, including the number of rooms, the average size of rooms, the number of dead ends, the average length of dead ends. He discovered that different representations are good for creating maps with specific characteristics.

Apart from McGuinness [10], another method to evaluate maps are heat maps[1], which are spatial visualizations of measures often used to represent player behaviour in maps. A common way to create heat maps is to subdivide the map space in a grid and to keep track of the value of the desired feature in each cell, cells are then colored depending on these values. Heat maps are widely used to design maps for multiplayer first person shooter games. Commonly used features include the number of kills and deaths (Figure 5.1),

[1] `http://blog.gameanalytics.com/blog/heatmapping.html`

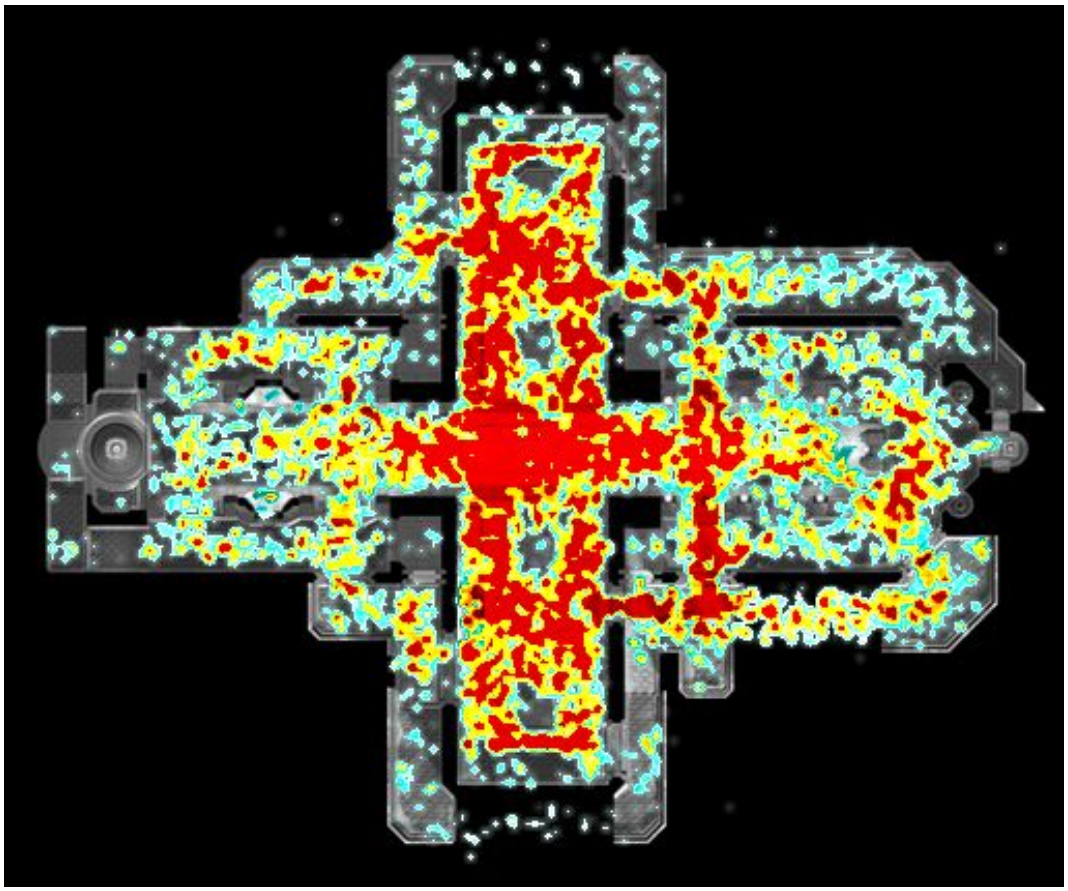


Figure 5.1: Heat map of a multiplayer map of *Halo 3* by Bungie Studios (2007), representing player death events. Red indicates a high death frequency, blue a small frequency.

players passage frequency, weapon usage. Heat maps are used to check if maps are played as intended by design, allowing to identify, for instance, bottlenecks and unused areas.

Güttler and Johansson [6] examined maps for first person team-based multiplayer games, proposing guidelines for the creation of maps that are entertaining and fair for all players. They focused on the paths available to reach objectives, which determine collision points, i.e. points in the map where teams are most likely to clash. The placement of collision points greatly influences tactical choices of players and the fairness of maps.

The complete graph G_C of a map M is created performing the following steps:

1. Let G_C be an empty graph.
2. For each architectural element E in M, a node N_E is added to G_C.
3. For each architectural element E in M:
   (a) For each used connector C of E:
      i. Let F be the architectural element linked to C. An edge from N_E to N_F is added to G_C, if not already present. The weight of the edge is set to the euclidean distance between the centers of the floors of E and F.

Table 5.1: Complete graph creation process.

5.2 Floor plan graph

To analyze the dungeon floor plans generated by our algorithm, we transform plans into graphs. Nodes correspond to architectural elements of the map, edges correspond to connections between architectural elements and edge weights correspond to distances between the centers of the architectural elements. Figure 5.2a shows a map and its complete graph, while Table 5.1 describes the creation of complete graphs.

From the complete graph we also generate a simplified graph by merging neighboring nodes with degree 1 and 2 into one node, plus setting all edge weights to 1. Thus, in the simplified graph a node can correspond to more elements. Figure 5.2b shows a map and its simplified graph, while Table 5.2 describes the creation of simplified graphs.

To evaluate the floor plans we generate, we tested several metrics, the most informative ones turned out to be: (i) degree, (ii) radius, (iii) closeness centrality, (iv) current flow closeness centrality, (v) random walk closeness centrality, (vi) betweenness centrality. The *degree* of a node is the number of its incident edges (i.e. the number of used connectors of the architectural element). The *radius* of a graph is calculated as the minimum eccentricity[2] of the nodes of the graph (the eccentricity of a node i is the greatest distance between i and any other node). The *closeness centrality*[3] (CC) of a node

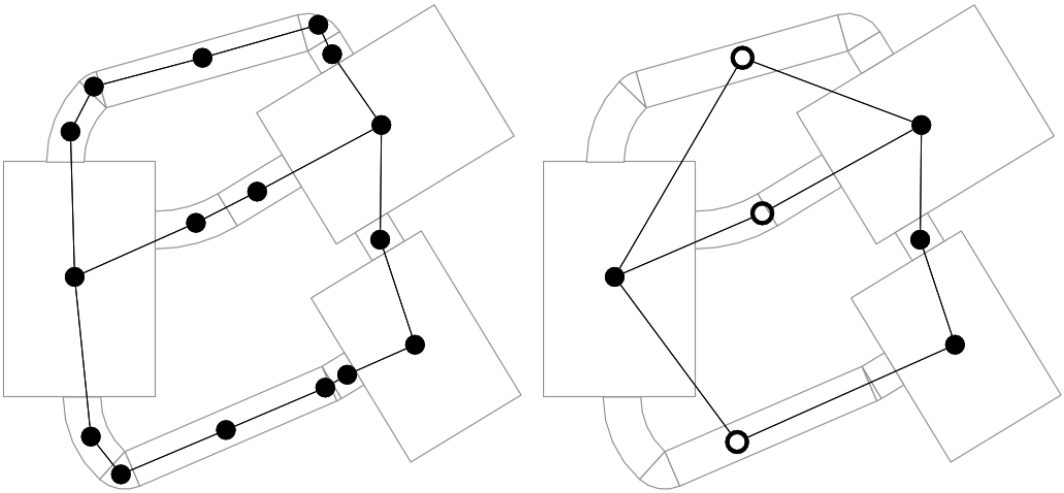
[2] `http://mathworld.wolfram.com/GraphEccentricity.html`

[3] `http://en.wikipedia.org/wiki/Centrality#Closeness_centrality`

The simplified graph G_S is created from the complete graph G_C, performing the following steps:

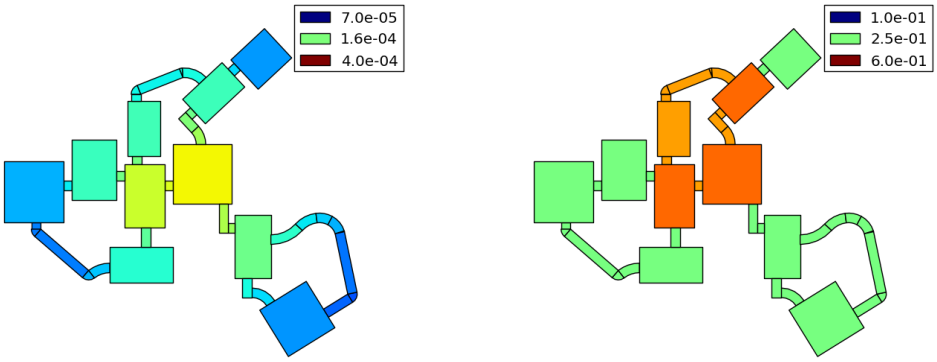
1. G_C is copied to G_S.
2. If G_S has a pair of neighboring nodes N_I and N_I with degree 1 or 2, the next step is executed; otherwise the process is terminated.
3. Let N_X and N_Y be the other neighbors respectively of N_I and N_J. N_I and N_J are removed from G_S.
4. A node N_K is added to G_S. Two edges (N_X, N_K) and (N_K, N_Y) are also added, with their weights both equal to 1.
5. The process is repeated from Step 2.

Table 5.2: Simplified graph creation process.



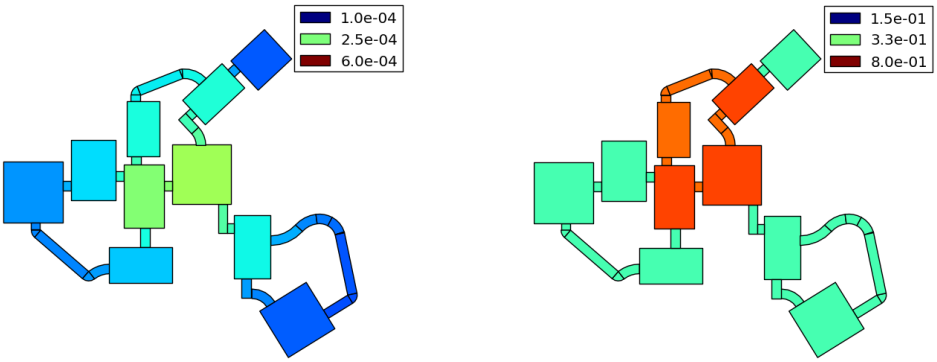
(a) Complete graph (b) Simplified graph

Figure 5.2: The complete graph and the simplified graph of a map. Solid dots are nodes corresponding to a single architectural element, hollow dots are nodes corresponding to more than one architectural element.



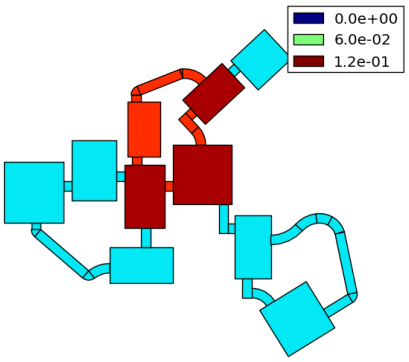
(a) complete graph (b) simplified graph

Figure 5.3: A map with colors representing the CC metric.



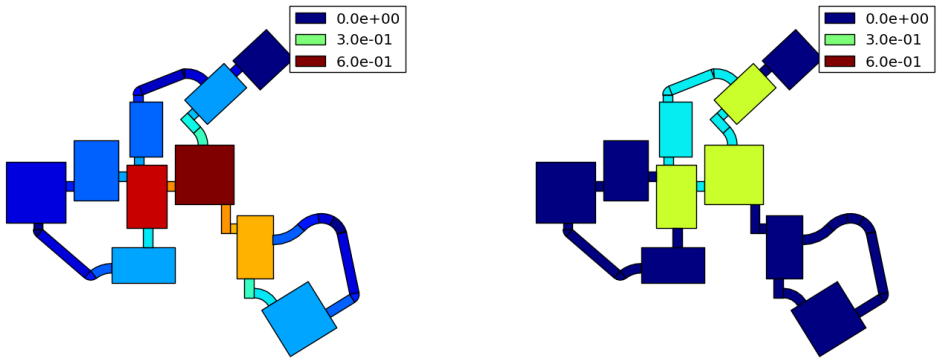
(a) complete graph (b) simplified graph

Figure 5.4: A map with colors representing the CFCC metric.



(a) simplified graph

Figure 5.5: A map with colors representing the RWCC metric.



(a) complete graph (b) simplified graph

Figure 5.6: A map with colors representing the BC metric.

i indicates how fast, on the average, a player in i can reach other nodes, assuming that the player always selects shortest paths when navigating from i to any other node. Figure 5.3 shows a map with colors representing CC. The CC of a node i in a graph G is the reciprocal of the average distance from all other nodes:

$$CC(i) = \frac{size(G) - 1}{\sum_{j \in G} d(i,j)}$$

where $size(G)$ is the number of nodes in G, and $d(i,j)$ is the distance between node i and node j (calculated as the shortest path distance). *Current flow closeness centrality* [4] (CFCC) is a variant of CC that considers nodes easier to reach if there are more paths to them. CFCC models the graph as an electrical network in which edges have a resistance equal to their weights. Figure 5.4 shows a map with colors representing CFCC. The CFCC of a node i is:

$$CFCC(i) = \frac{size(G) - 1}{\sum_{j \in G} r(i,j)}$$

where $r(i,j)$ is the *effective resistance* between node i and node j. Another variant of CC is *random walk closeness centrality*[4] (RWCC), that assumes that the player moves randomly when trying to reach a node. RWCC describes the average speed with which randomly walking processes reach a node from other nodes of the graph. Random walk processes start from a node and move from node to node along edges, choosing them randomly. We consider RWCC only for simplified graphs. Figure 5.5 shows a map with colors representing RWCC.

[4] `http://en.wikipedia.org/wiki/Random_walk_closeness_centrality`

Given a weighted graph, the transition matrix M of a random walk process is defined. The m_{ij} element of M is the probability that the random walk, from node i, proceedes directly to node j:

$$m_{ij} = \frac{w_{ij}}{\sum_{j \in G}^{n} w_{ij}}$$

where w_{ij} is the weight of the edge (i, j), if it exists; otherwise it is 0. The *mean first passage time* (MFPT) from a node i to a node j is the average number of steps it takes for a random walk process to reach node j from node i for the first time:

$$MFPT(i, j) = \sum_{k \neq j} ((I - M_{-j})^{-2})_{ik} m_{kj}$$

where M_{-j} is a transformation of M obtained by deleting its j^{th} row and column. The RWCC of a node i is the reciprocal of the average mean first passage time to that node:

$$RWCC(i, j) = \frac{size(G)}{\sum_{j \in G} MFPT(i, j)}$$

We consider RWCC only for simplified graphs, since in complete graphs edge weights represent distances between nodes, which are not related to transition probabilities. The *betweenness centrality*[5] (BC) of a node indicates how often a player crosses it while navigating between two other nodes. Similarly to CC, BC only considers shortest paths. BC is normalized by $\frac{2}{(size(G)-1)(size(G)-2)}$. Figure 5.6 shows a map with colors representing BC. The BC of a node i is:

$$BC(i) = \sum_{j \neq i \neq k \in G} \frac{p(j, k|i)}{p(j, k)} \frac{2}{(size(G) - 1)(size(G) - 2)}$$

where $p(j, k)$ is the number of shortest paths from node j to node k, and $p(j, k|i)$ is the number of these shortest paths that pass through i (without considering paths having i as an endpoint).

5.3 Evaluation of single parameters

As the very first step, we evaluated how the single parameters of our algorithm (see Table 4.2) affected the generated maps. For each parameter P under evaluation, we select a set of values S_V; then, for each value V in S_V, we generate

[5] `http://en.wikipedia.org/wiki/Centrality#Betweenness_centrality`

Property	Value
$initialElements$	A rectangular room.
$prob_{classes}$	Rectangular room: 25% Linear corridor: 37.5% Curved corridor: 37.5%
$prob_{degrees}$	1: 0.01% 2: 0.99% 3: 99%
$maxRooms$	10
$boundary$	$None$
$connWnd_{start}$	0
$connWnd_{end}$	1
$connWnd_{use}$	1
$maxConnDist$	10 times the width of connectors.
$deadEnds_{\%}$	0

Table 5.3: Default values of parameters of our map generation algorithm.

1000 maps setting P to V, while setting other parameters to the default values shown in Table 5.3. Default values are set as follows: the $initialElements$ parameter is set to start the generation with a rectangular room; $prob_{classes}$ gives a 25% probability to rectangular rooms and a 75% probability to corridors; $prob_{degrees}$ gives a 99% probability to architectural elements with degree 3, a 0.99% probability to degree 2 and a 0.01% probability to degree 1, thus there is an high chance that new elements are attached to elements with higher degree; $maxRooms$ is set to 10; $boundary$ in $None$, i.e. map growth is not constrained; $connWnd_{start}$ is 0, $connWnd_{end}$ is 1 and $connWnd_{use}$ is 1, so the *additional connection creation phase* adds the maximum number of connections; $deadEnds_{\%}$ is 0, i.e. all dead ends are removed from maps.

5.3.1 Evaluation of $prob_{classes}$

The $prob_{classes}$ parameter is a dictionary that associates architectural elements classes to probabilities. It is used in the *digging phase* to randomly choose classes of new architectural elements. To test the influence of $prob_{classes}$ we considered 5 $prob_{classes}$ values and we generated 5 sets of 1000 maps each. The first set ($S_{r10,c90}$) was generated giving a 10% probability to rooms and a 90% probability to corridors; probabilities used for each set are shown in

Set	Rectangular room probability	Linear corridor probability	Curved corridor probability
$S_{r10,c90}$	10%	45%	45%
$S_{r25,c75}$	25%	37.5%	37.5%
$S_{r50,c50}$	5%	25%	25%
$S_{r75,c50}$	75%	12.5%	12.5%
$S_{r90,c10}$	90%	5%	5%

Table 5.4: Architectural element class probabilities used to generate map sets for the evaluation of $prob_{classes}$.

Given an ordered set of values $x_1 \leq x_2 \leq \cdots \leq x_N$ with respective weights $w_1, w_2, \ldots, w_N$, the i^{th} weighted quartile q_i is equal to the value x_j such as

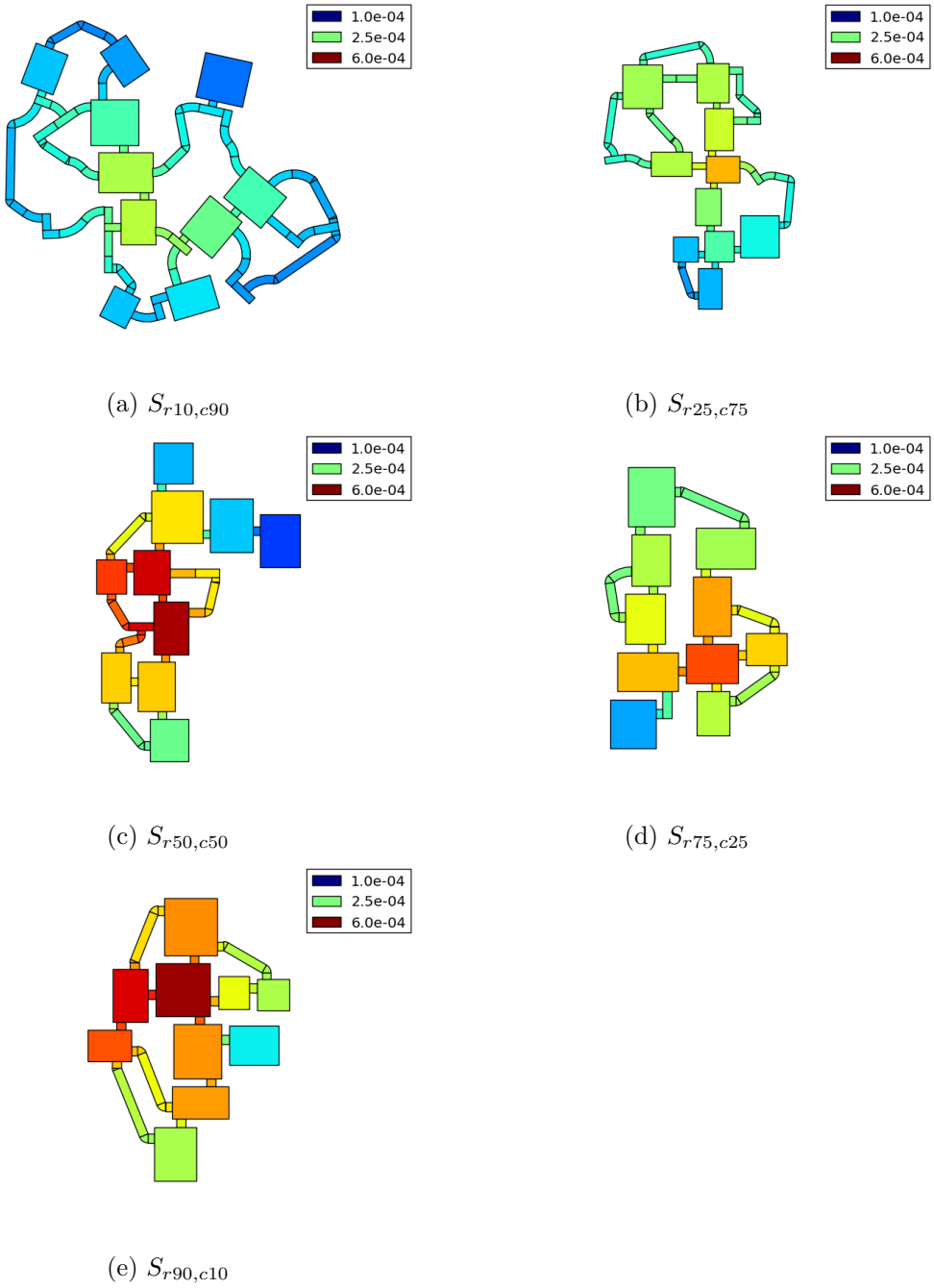
$$S_{j-1} < 0.25 \cdot i \cdot S_N \leq S_j$$

where $S_n = \sum_{k=1}^{n} w_k$

Table 5.5: Definition of weighted quartiles.

Table 5.4. Figure 5.7 shows maps of the generated sets. As the corridor probability decreases, there are less corridor placed by the *digging phase*; in the map from $S_{r90,c10}$ are present only corridors needed to separate rooms and corridors added by the *additional connection creation phase*. As a result, smaller corridor probabilities tend to produce smaller maps. Figures from 5.8 to 5.13 show boxplots[6] of the considered metrics for complete and simplified graphs. Boxplots of complete graphs show weighted quartiles (calculated as shown in Table 5.5), using as weights the floor area of architectural elements, while boxplots of simplified graphs show non-weighted quartiles.

[6] `http://en.wikipedia.org/wiki/Box_plot`



(a) $S_{r10,c90}$

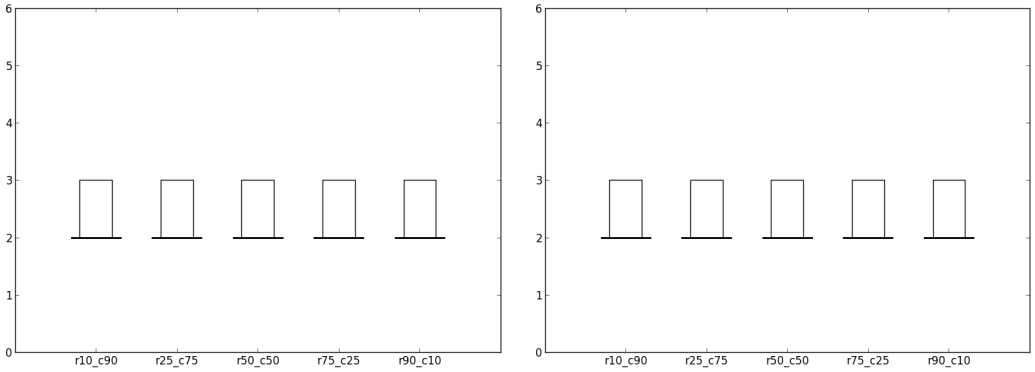
(b) $S_{r25,c75}$

(c) $S_{r50,c50}$

(d) $S_{r75,c25}$

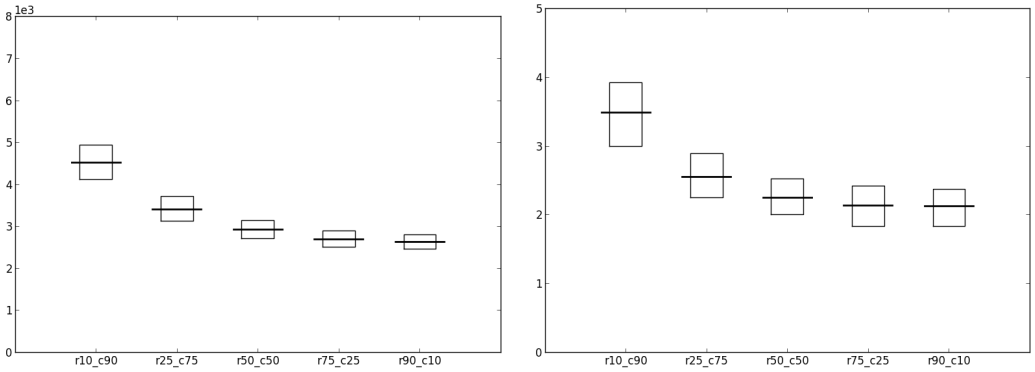
(e) $S_{r90,c10}$

Figure 5.7: Maps generated for the evaluation of $prob_{classes}$ without additional connections. Colors indicate CFCC of complete graphs.



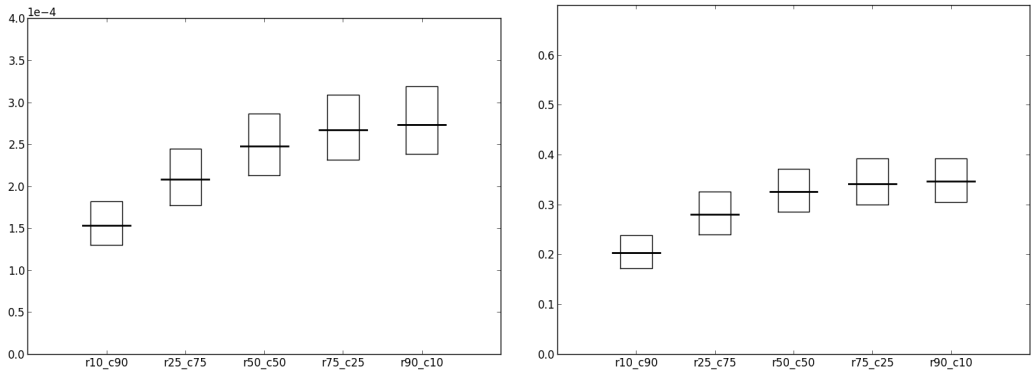
(a) degree, complete graph (b) degree, simplified graph

Figure 5.8: Boxplots of the degree metric for the evaluation of $prob_{classes}$.



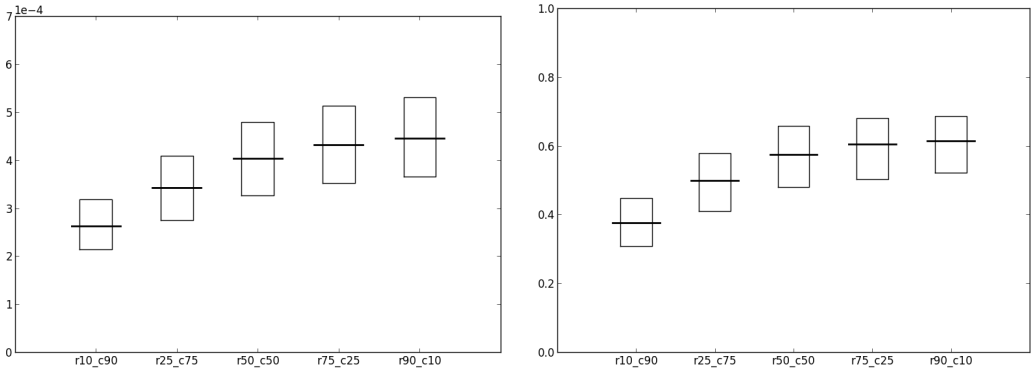
(a) radius, complete graph (b) radius, simplified graph

Figure 5.9: Boxplots of the radius metric for the evaluation of $prob_{classes}$.



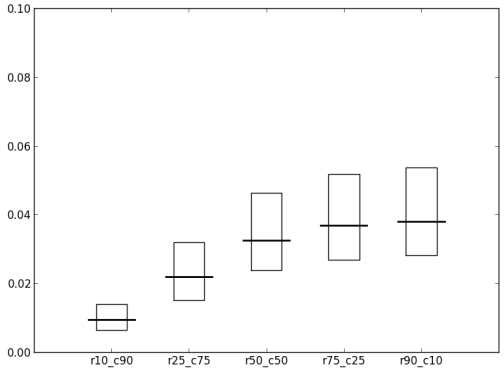
(a) CC, complete graph (b) CC, simplified graph

Figure 5.10: Boxplots of the CC metric for the evaluation of $prob_{classes}$.



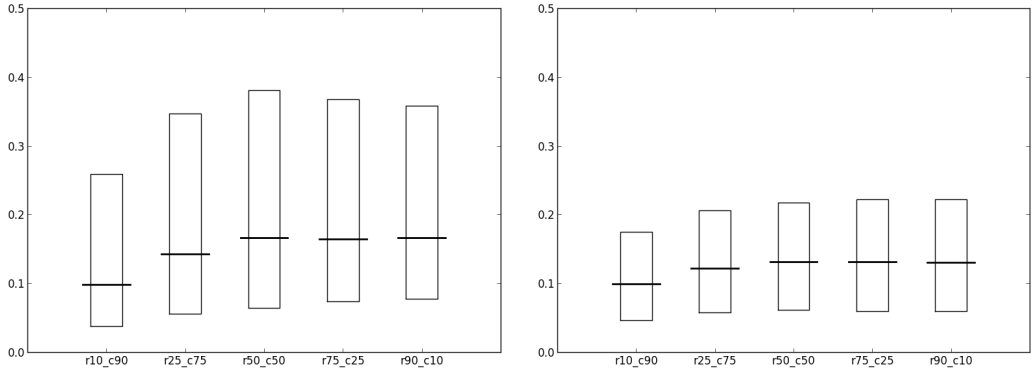
(a) CFCC, complete graph (b) CFCC, simplified graph

Figure 5.11: Boxplots of the CFCC metric for the evaluation of $prob_{classes}$.



(a) RWCC, simplified graph

Figure 5.12: Boxplots of the RWCC metric for the evaluation of $prob_{classes}$.



(a) BC, complete graph (b) BC, simplified graph

Figure 5.13: Boxplots of the BC metric for the evaluation of $prob_{classes}$.

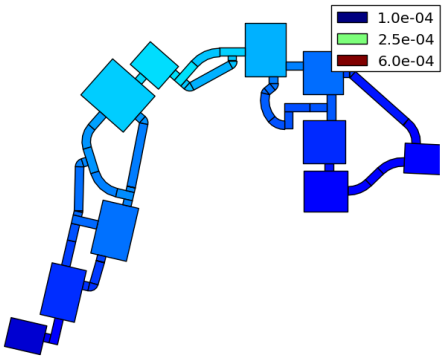
As regards the degree metric, it is shown no difference between the sets, for both complete and simplified graphs. The radius of graphs decreases along with the corridor probability, since the size of produced maps also decreases. CC, CFCC and RWCC increase as the map size decreases, because smaller maps require shorter paths to navigate between architectural elements. Little difference is shown between $S_{r50,c50}$, $S_{r75,c25}$ and $S_{r90,c10}$ in map plans and boxplots, so room probabilities above 50% can be avoided for map generation.

5.3.2 Evaluation of $prob_{degrees}$

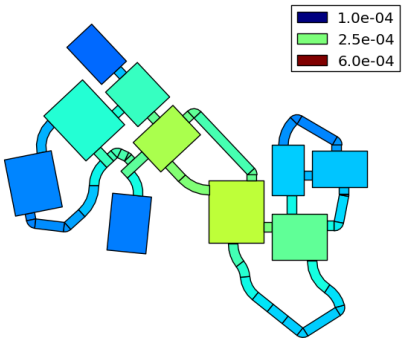
The $prob_{degrees}$ parameter is a dictionary that associates degree values to probabilities. It is used in the *digging phase* to choose randomly the degree (i.e. the number of used connectors) of the architectural elements to which attach new architectural elements. To test the influence of $prob_{degrees}$ we considered 5 $prob_{degrees}$ values and generated 5 sets of 1000 maps each. Maps of the first set (S_{low100}) were generated giving probabilities with a ratio of 100 to degrees from 1 to 3: a 99% probability to architectural elements with degree 1, a 0.99% probability to elements with degree 2 and 0.01% probability to elements with degree 3. Other sets were generated giving other ratios to these probabilities (see Table 5.6). Figure 5.14 shows maps of the generated sets. Favoring low degree values, the generated maps tend to be more linear, while favoring high degree values maps tend to be more branched. Figures from 5.15 to 5.20 show boxplots related to generated sets. The radius tends to be higher for S_{low100} than other sets, since in linear maps central architectural elements have high eccentricity. Also BC of complete graphs is higher for S_{low100}, because in linear maps there are few possible paths, so architectural elements have

Set	**Degree 1 probability**	**Degree 2 probability**	**Degree 3 probability**
S_{low100}	99%	0.99%	0.01%
S_{low4}	76%	19%	5%
$S_{uniform}$	33%	33%	34%
S_{high4}	5%	19%	76%
$S_{high100}$	0.01%	0.99%	99%

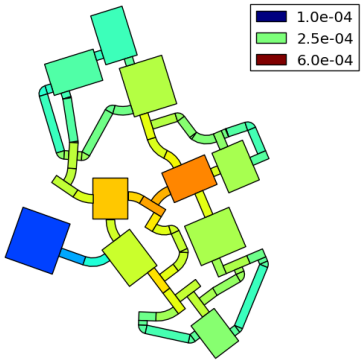
Table 5.6: Degree probabilities used to generate map sets for the evaluation of $prob_{degrees}$.



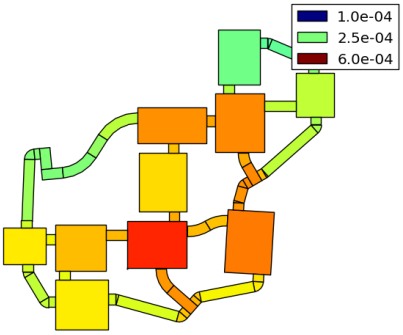
(a) S_{low100}



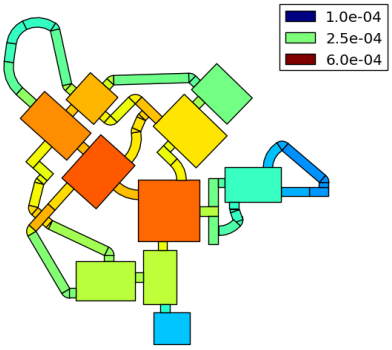
(b) S_{low4}



(c) $S_{uniform}$

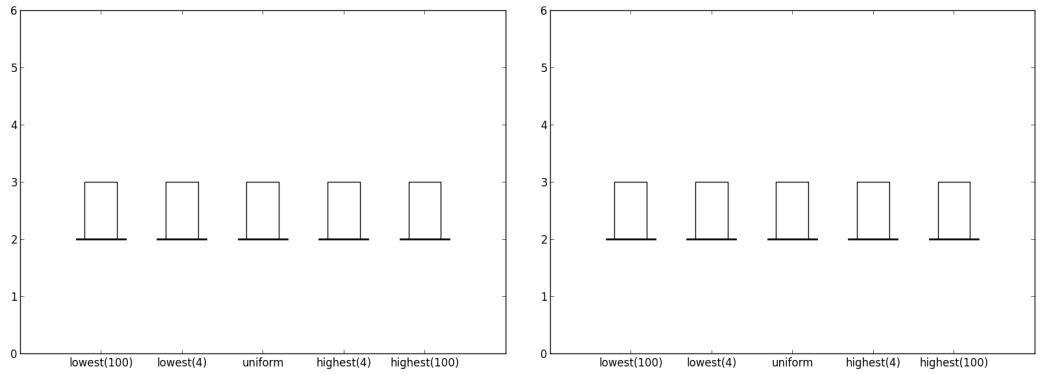


(d) S_{high4}



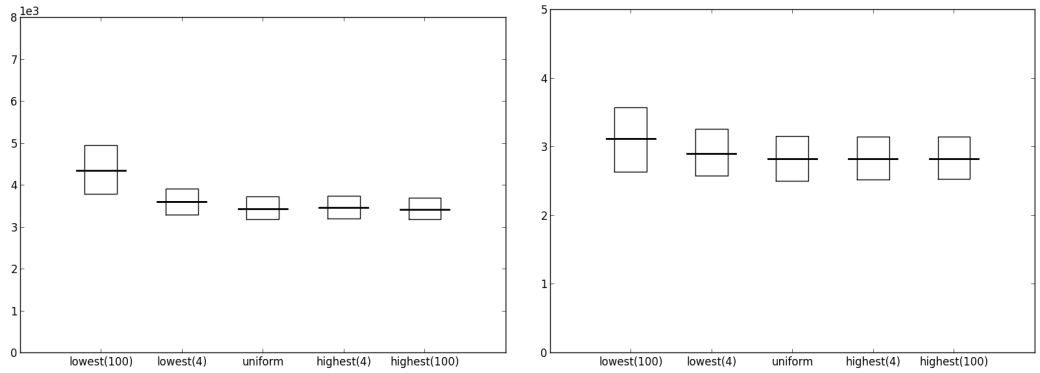
(e) $S_{high100}$

Figure 5.14: Maps generated for the evaluation of $prob_{degrees}$. Colors indicate CFCC of complete graphs.



(a) degree, complete graph (b) degree, simplified graph

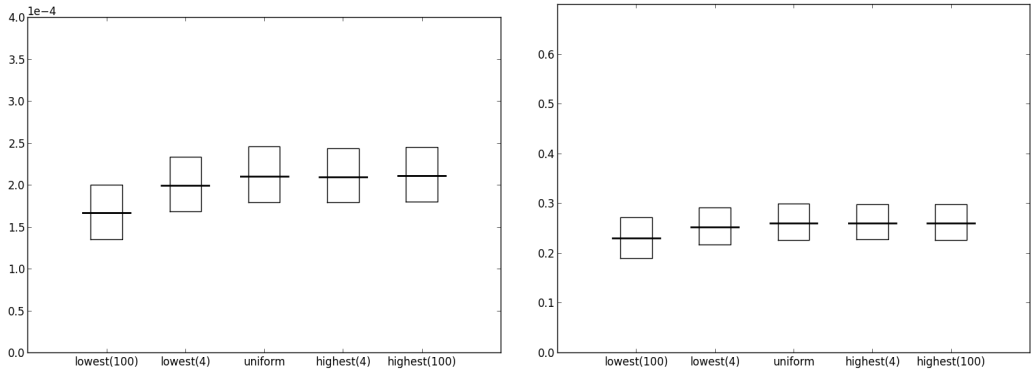
Figure 5.15: Boxplots of the degree metric for the evaluation of $prob_{degrees}$.



(a) radius, complete graph (b) radius, simplified graph

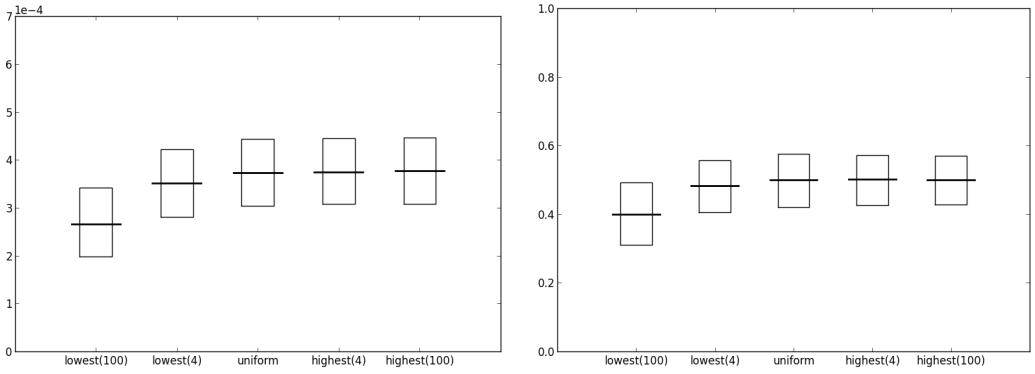
Figure 5.16: Boxplots of the radius metric for the evaluation of $prob_{degrees}$.

many shortest paths crossing them. CC and CFCC increase from S_{low100} to $S_{high100}$, since in branched maps elements are more packed together than in linear maps, so more additional connections are created, leading to shorter distances between elements. It can be noted that in the boxplots there is almost no difference between $S_{uniform}$, S_{high4} and $S_{high100}$, which means that, to control our algorithm's output, it is pointless to consider probability ratios above that of $S_{uniform}$.



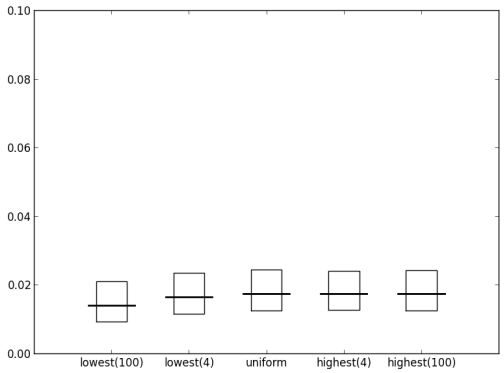
(a) CC, complete graph (b) CC, simplified graph

Figure 5.17: Boxplots of the CC metric for the evaluation of $prob_{degrees}$.



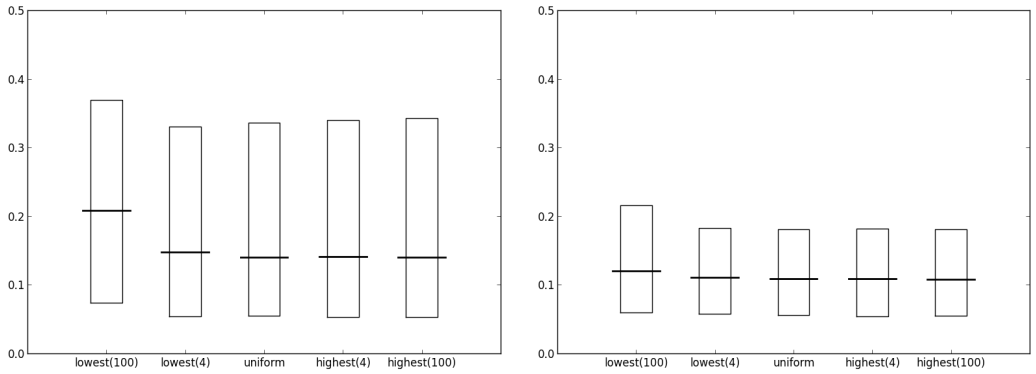
(a) CFCC, complete graph (b) CFCC, simplified graph

Figure 5.18: Boxplots of the CFCC metric for the evaluation of $prob_{degrees}$.



(a) RWCC, simplified graph

Figure 5.19: Boxplots of the RWCC metric for the evaluation of $prob_{degrees}$.

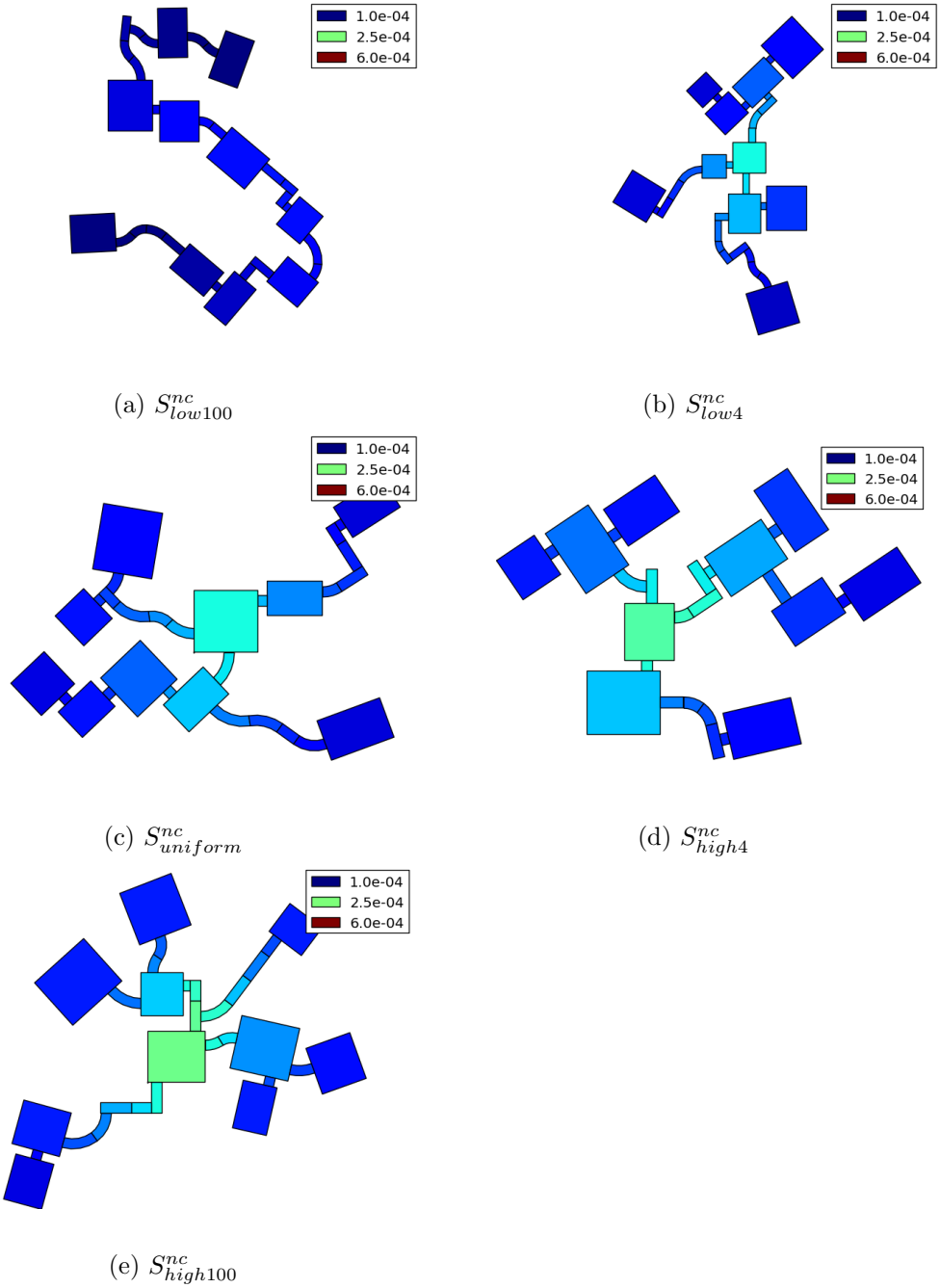


(a) BC, complete graph

(b) BC, simplified graph

Figure 5.20: Boxplots of the BC metric for the evaluation of $prob_{degrees}$.

We also tested the influence of $prob_{degrees}$ in the case that no connections are added in the *additional connection creation phase*, thus all generated maps have a tree-like structure. We considered 5 sets of maps ($S^{nc}_{low100}, S^{nc}_{low4}, S^{nc}_{uniform}, S^{nc}_{high4}, S^{nc}_{high100}$) generated using the same parameters as in the evaluation of $prob_{degrees}$ with default parameters (Table 5.6), except for the $connWnd_{use}$ parameter, which is set to 0. Figure 5.21 shows maps of the generated sets. Figures from 5.22 to 5.27 show boxplots related to the generated sets. Maps in S^{nc}_{low100} tend to be linear, so in complete graphs many nodes have degree 2. For the degree metric of simplified graphs of all sets, the first and second quartiles are 1; this is because many nodes in the tree-like complete graphs are merged into a single leaf node. For the radius metric, complete graphs of S^{nc}_{low100} have values higher than other sets, while simplified graphs of the same set have values lower than other sets. This is because simplified graphs of linear maps are very small compared to their complete graphs. For the same reason, CC, CFCC and RWCC metrics of complete graphs of S^{nc}_{low100} have values higher than other sets, while simplified graphs of the same set have values lower than other sets. The first two quartiles of BC of simplified graphs are 0 for all sets, because tree-like graphs have many leaf nodes, which have null BC since they are not crossed by any shortest path. Instead, non-leaf nodes are crossed by many shortest paths, so the third quartile of the same metric have high values. Since linear maps have very small simplified graphs, RWCC is very high for S^{nc}_{low100}. CFCC of complete graphs of all groups is lower than the case with additional connections (Figure 5.18a),

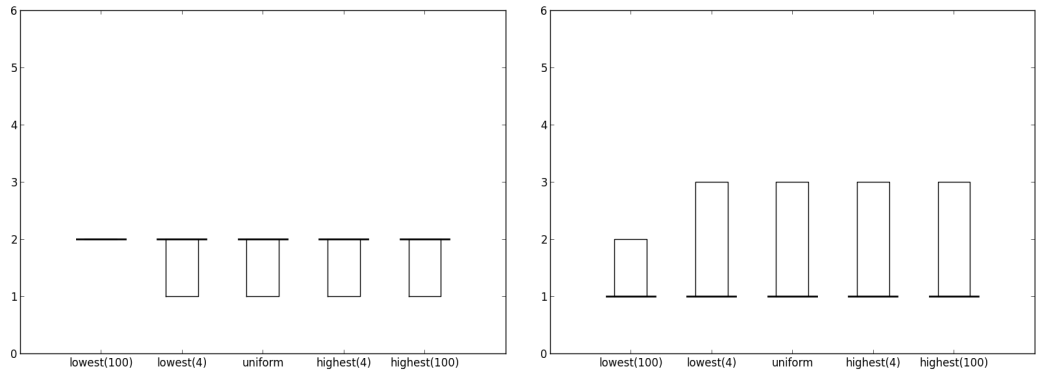


(a) S^{nc}_{low100} (b) S^{nc}_{low4}

(c) $S^{nc}_{uniform}$ (d) S^{nc}_{high4}

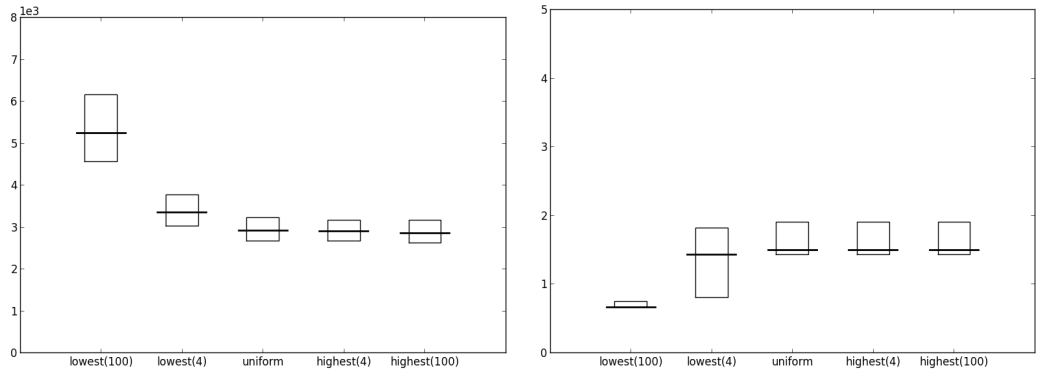
(e) $S^{nc}_{high100}$

Figure 5.21: Maps generated for the evaluation of $prob_{degrees}$ without additional connections. Colors indicate CFCC of complete graphs.



(a) degree, complete graph (b) degree, simplified graph

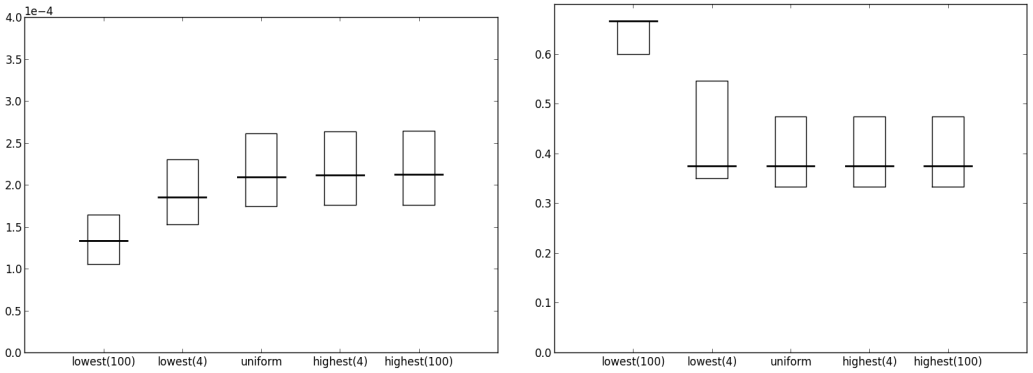
Figure 5.22: Boxplots of the degree metric for the evaluation of $prob_{degrees}$ without additional connections.



(a) radius, complete graph (b) radius, simplified graph

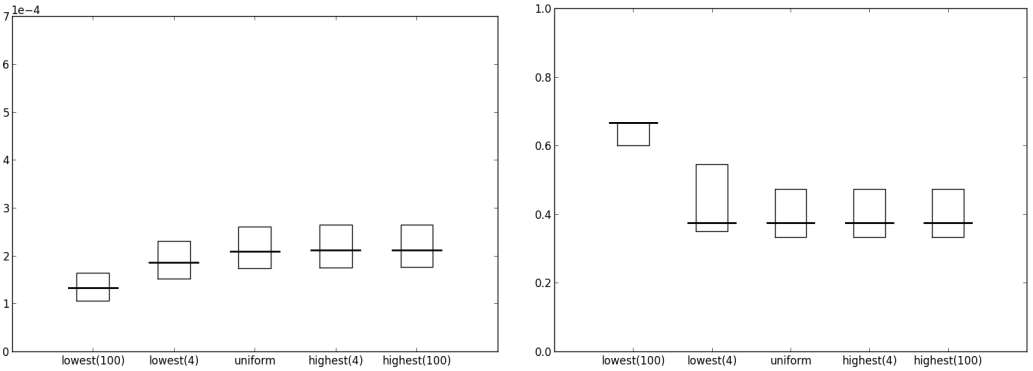
Figure 5.23: Boxplots of the radius metric for the evaluation of $prob_{degrees}$ without additional connections.

while CC is lower only for S^{nc}_{low100} and S^{nc}_{low4}. Thus, we suggest that shortest paths between architectural elements (which determine CC) are shortened by additional connections only in little branched maps.



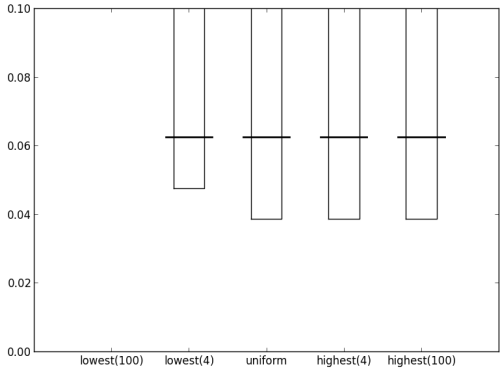
(a) CC, complete graph (b) CC, simplified graph

Figure 5.24: Boxplots of the CC metric for the evaluation of $prob_{degrees}$ without additional connections.



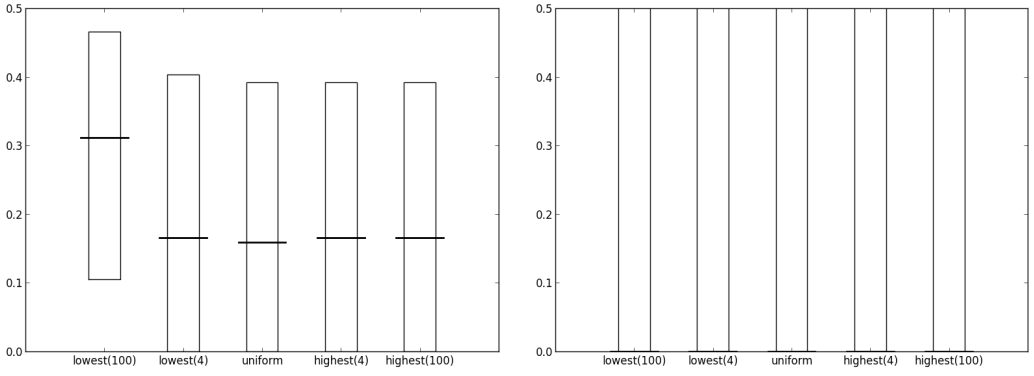
(a) CFCC, complete graph (b) CFCC, simplified graph

Figure 5.25: Boxplots of the CFCC metric for the evaluation of $prob_{degrees}$ without additional connections.



(a) RWCC, simplified graph

Figure 5.26: Boxplots of the RWCC metric for the evaluation of $prob_{degrees}$ without additional connections.



(a) BC, complete graph

(b) BC, simplified graph

Figure 5.27: Boxplots of the BC metric for the evaluation of $prob_{degrees}$ without additional connections.

Finally, we tested the influence of $prob_{degrees}$ in the case that the polygonal room architectural element (see Figure 5.28) is used, in addition to basic architectural elements. We generated 5 sets of maps ($S^p_{low100}, S^p_{low4}, S^p_{uniform}, S^p_{high4}, S^p_{high100}$) using the same parameters as in the evaluation of $prob_{degrees}$ with default parameters (Table 5.6), except for the $prob_{classes}$ parameter, which includes the polygonal room class (Table 5.7). Figure 5.29 shows maps of the generated sets. Figures from 5.30 to 5.35 show boxplots related to the generated sets. With respect to sets generated without polygonal rooms (Figure 5.20), the quartiles of the degree metric of complete graphs are higher for all sets apart from S^p_{low100}. This is due to the presence of polygonal rooms, which have more connectors than rectangular rooms (on average) and more area than corridors. This increase of degree values does not happen for simplified graphs, because the area of architectural elements is not considered. For other metrics no significant difference is shown with respect to sets generated without polygonal rooms. Thus, we suggest that the use of polygonal rooms, while affecting the appearance of the map, does not change the overall length and number of paths between architectural elements.

Element class	**Probability**
Rectangular room	12.5%
Polygonal room	12.5%
Linear corridor	37.5%
Curved corridor	37.5%

Table 5.7: Classes and probabilities set in the $prob_{classes}$ parameter, used to generate map sets for the evaluation of $prob_{degrees}$ with polygonal rooms.

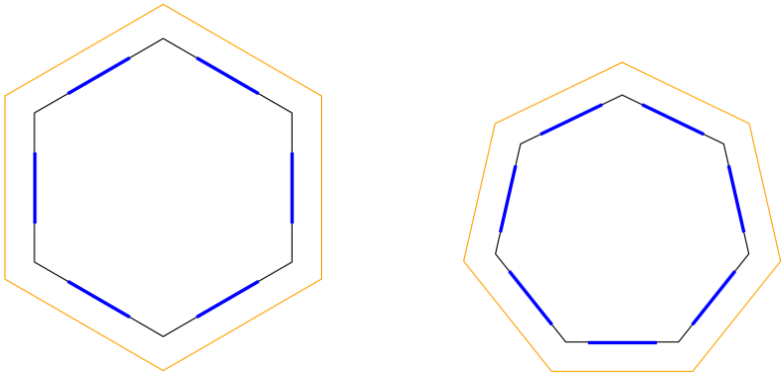
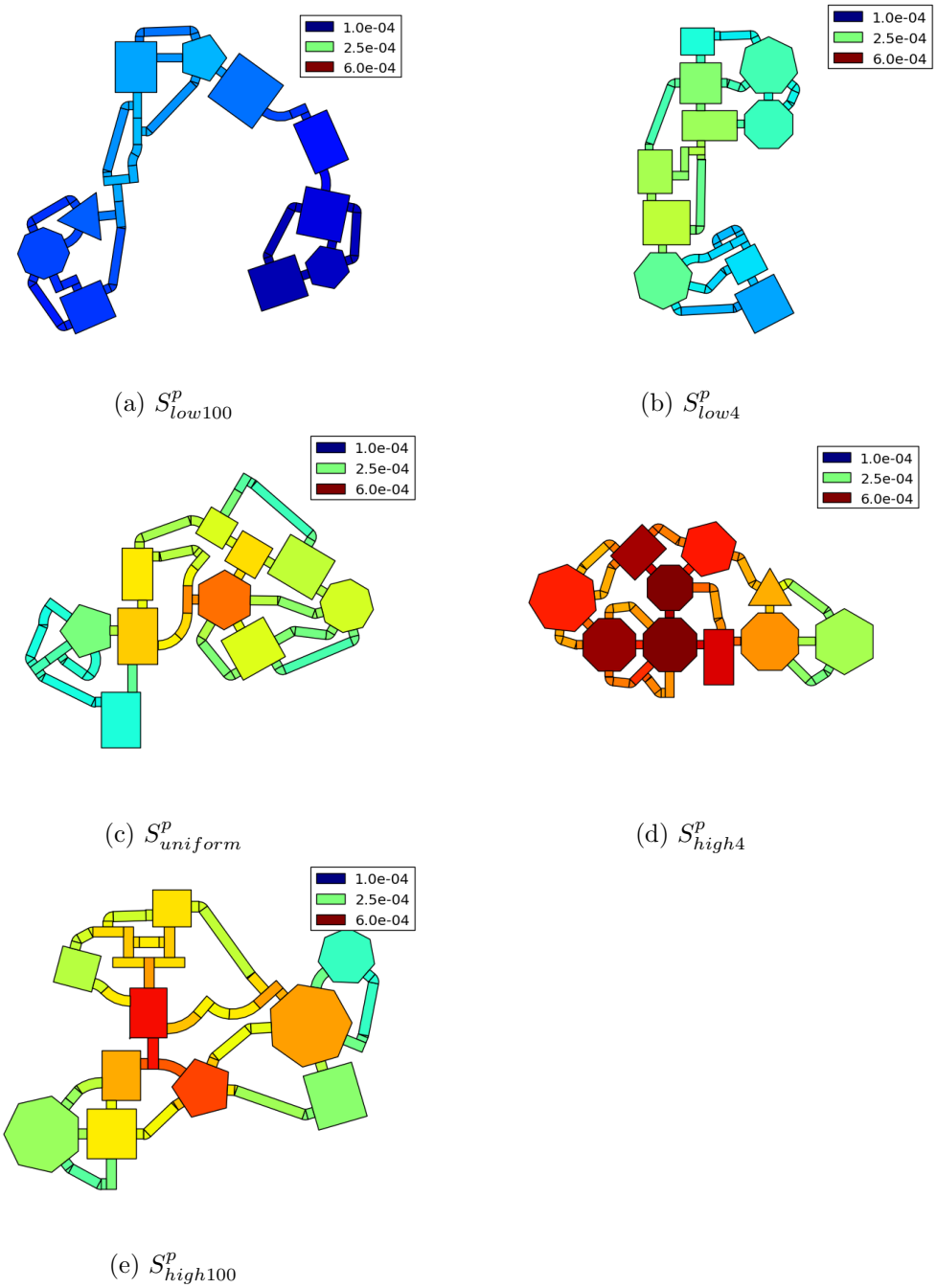


Figure 5.28: Examples of polygonal room architectural elements. Floors are black, shells are yellow, connectors are blue. Floors and shells of polygonal rooms are regular polygons, the number of sides is chosen randomly in the range $[3, 8]$. The length of the sides is also random. A connector is placed in the middle of each side.

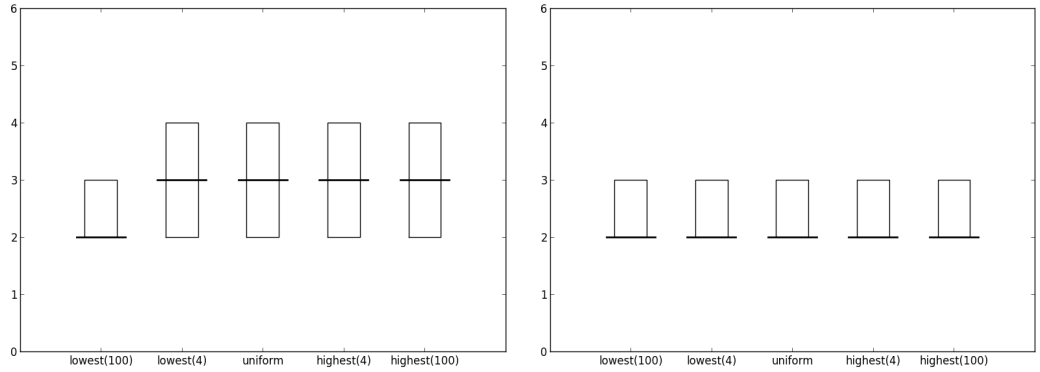


(a) S^p_{low100} (b) S^p_{low4}

(c) $S^p_{uniform}$ (d) S^p_{high4}

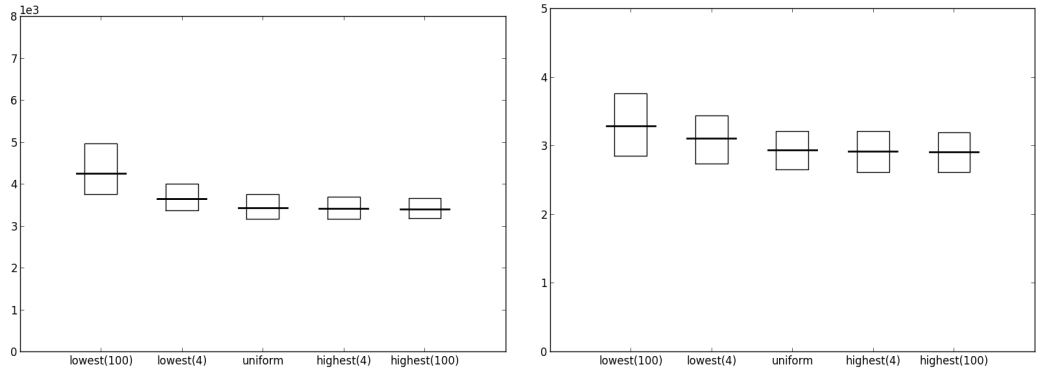
(e) $S^p_{high100}$

Figure 5.29: Maps generated for the evaluation of $prob_{degrees}$ with polygonal rooms. Colors indicate CFCC of complete graphs.



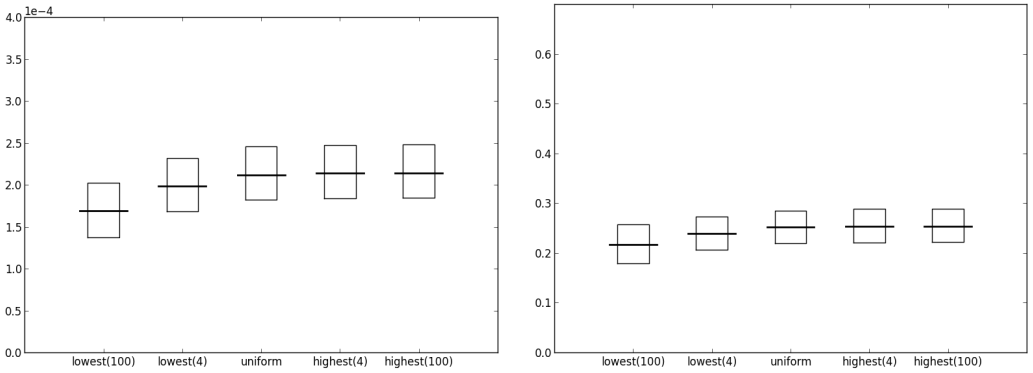
(a) degree, complete graph (b) degree, simplified graph

Figure 5.30: Boxplots of the degree metric for the evaluation of $prob_{degrees}$ with polygonal rooms.



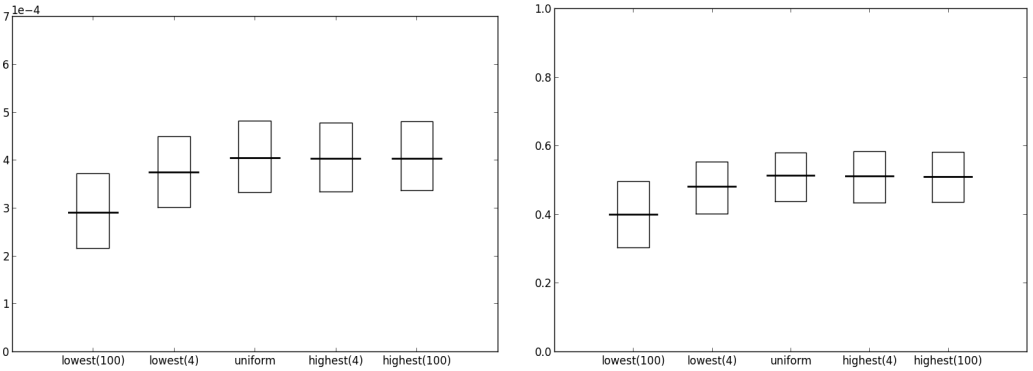
(a) radius, complete graph (b) radius, simplified graph

Figure 5.31: Boxplots of the radius metric for the evaluation of $prob_{degrees}$ with polygonal rooms.



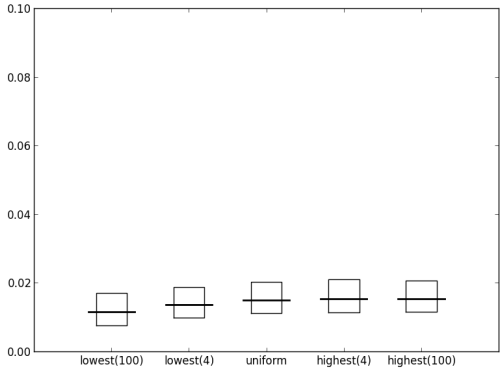
(a) CC, complete graph (b) CC, simplified graph

Figure 5.32: Boxplots of the CC metric for the evaluation of $prob_{degrees}$ with polygonal rooms.



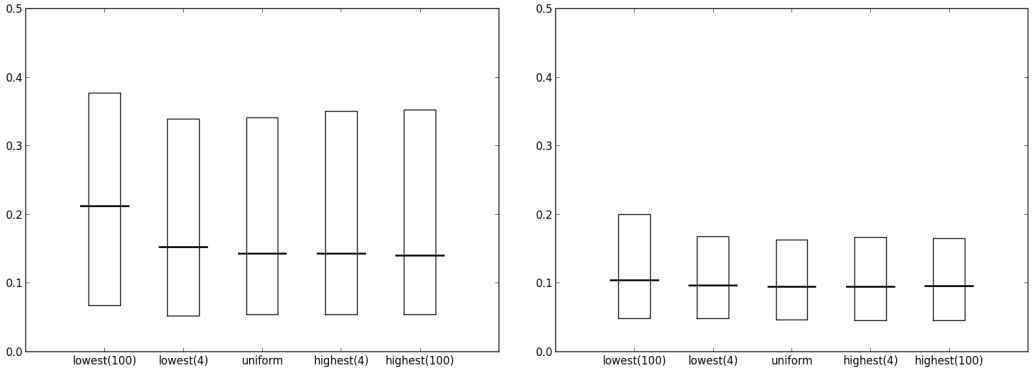
(a) CFCC, complete graph (b) CFCC, simplified graph

Figure 5.33: Boxplots of the CFCC metric for the evaluation of $prob_{degrees}$ with polygonal rooms.



(a) RWCC, simplified graph

Figure 5.34: Boxplots of the RWCC metric for the evaluation of $prob_{degrees}$ with polygonal rooms.



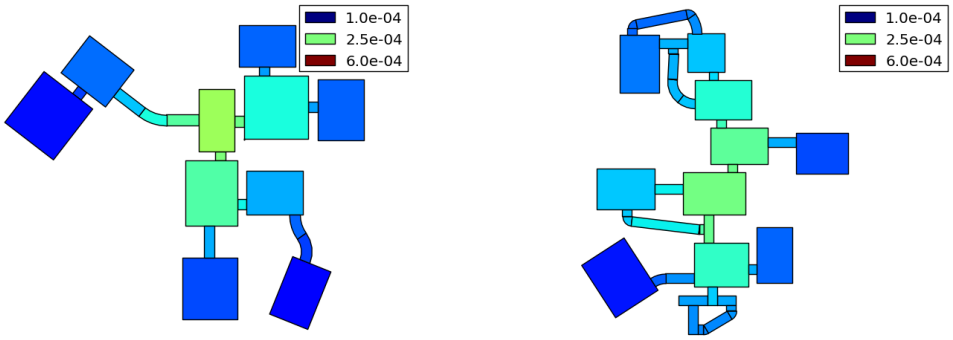
(a) BC, complete graph

(b) BC, simplified graph

Figure 5.35: Boxplots of the BC metric for the evaluation of $prob_{degrees}$ with polygonal rooms.

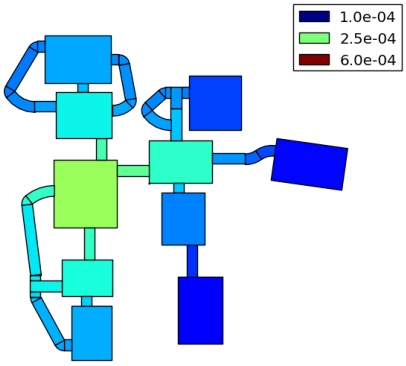
5.3.3 Evaluation of $connWnd_{start}$

To test the influence of $connWnd_{start}$ we created 5 sets of 1000 maps each. The first set S_{near0} was generated setting $connWnd_{start}$ to 1, so no connections are added in the *additional connection creation phase*. The second set S_{near25} was generated setting $connWnd_{start}$ to 0.75, so the algorithm tries to add the 25% of nearest connections (i.e. it joins only close connectors, so it creates small loops). Sets S_{near50} and S_{near75} were generated setting $connWnd_{start}$ respectively to 0.5 and 0.25, thus trying to create the 50% and 75% of nearest connections. Set $S_{near100}$ was generated setting $connWnd_{start}$ to 0, so the algorithm tries to create the maximum number of connections. Figure 5.36 shows maps of the generated sets. Maps of S_{near0} have no additional connections, so they have a tree-like structure. Maps of S_{near25} have few connections, created between unused connector that are close in the map. Boxplots of the considered metrics are shown in figures from 5.37 to 5.42. The first two quartiles of BC of S_{near0} and S_{near25} are 0, since their maps of S_{near0} are tree-like and maps of S_{near25} only have small loops, so there is a relevant number of nodes which are not crossed by any path. CC of simplified graphs is greater for S_{near0} than other sets. This is because maps in S_{near0} have no additional connection, so their simplified graphs have fewer nodes than other sets; since in simplified graphs distances between neighboring nodes are all equal, in small graphs paths between nodes are shorter than in large graphs, so CC is bigger for nodes in smaller graphs. Instead, CC of complete graphs is similar for all sets; we suggest that this is because the creation of additional connections, while shortening distances between architectural elements, also increase the size of the map, thus the two effects tend to compensate. CFCC of complete graphs is similar for S_{near0} and S_{near25}, because in S_{near25} the additional connections created between close connectors does not produce new paths between far architectural elements; S_{near50}, S_{near75} and $S_{near100}$ have increasingly higher values because of the increasing number of additional connections they create. CFCC of simplified graphs has a similar trend to that of complete graphs, except for S_{near0}, which has greater values than S_{near25}, because S_{near0} has smaller simplified graphs. According to the trends of CC and CFCC we suggest that, for these metrics, simplified graphs reflect the increase of complexity of the map due to additional connections, without considering the



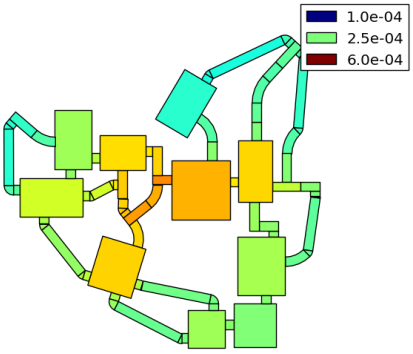
(a) S_{near0}

(b) S_{near25}



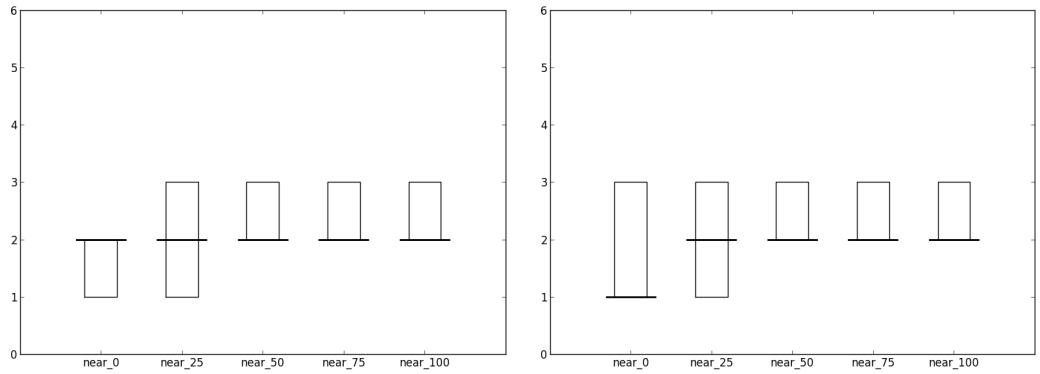
(c) S_{near50}

(d) S_{near75}



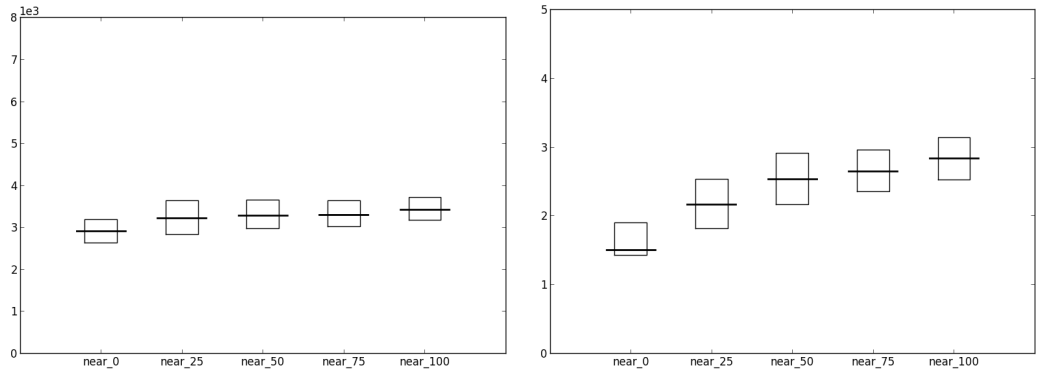
(e) $S_{near100}$

Figure 5.36: Maps generated for the evaluation of $connWnd_{start}$. Colors indicate CFCC of complete graphs.



(a) degree, complete graph (b) degree, simplified graph

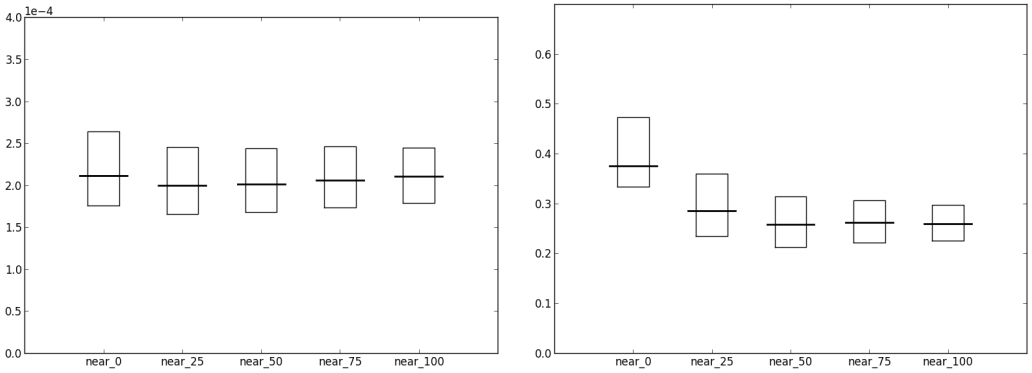
Figure 5.37: Boxplots of the degree metric for the evaluation of $connWnd_{start}$.



(a) radius, complete graph (b) radius, simplified graph

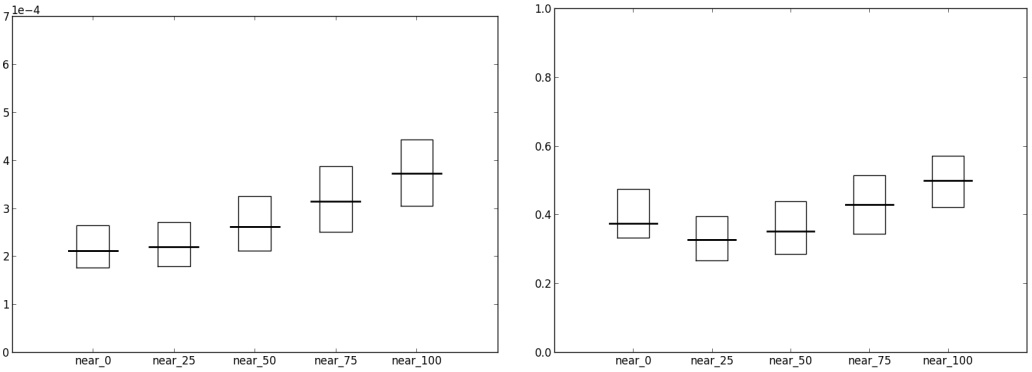
Figure 5.38: Boxplots of the radius metric for the evaluation of $connWnd_{start}$.

absolute distances between architectural elements. Instead, complete graphs reflect both complexity and absolute distances (which affect travelling times of the players).



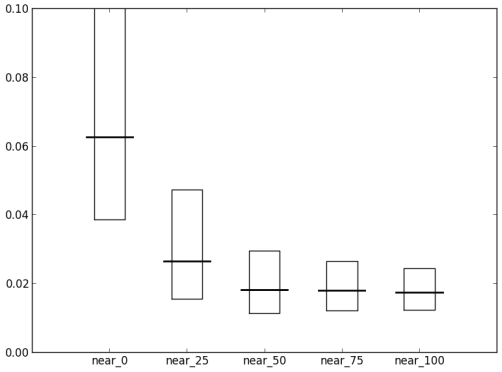
(a) CC, complete graph (b) CC, simplified graph

Figure 5.39: Boxplots of the CC metric for the evaluation of $connWnd_{start}$.



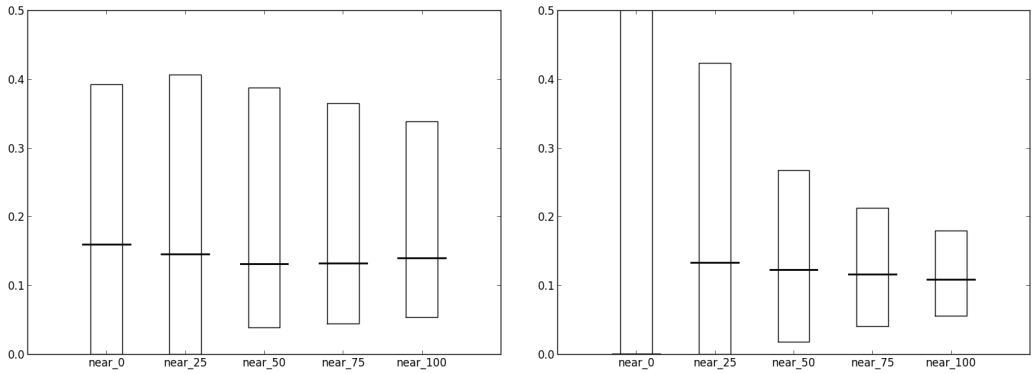
(a) CFCC, complete graph (b) CFCC, simplified graph

Figure 5.40: Boxplots of the CFCC metric for the evaluation of $connWnd_{start}$.



(a) RWCC, simplified graph

Figure 5.41: Boxplots of the RWCC metric for the evaluation of $connWnd_{start}$.



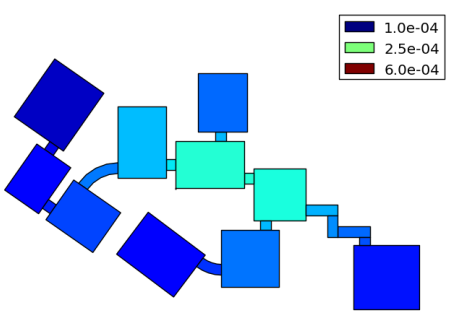
(a) BC, complete graph

(b) BC, simplified graph

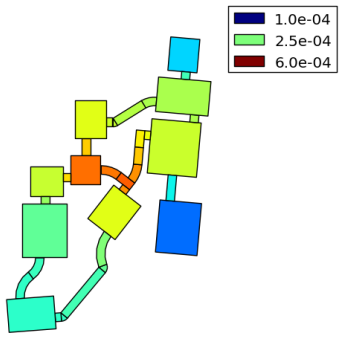
Figure 5.42: Boxplots of the BC metric for the evaluation of $connWnd_{start}$.

5.3.4 Evaluation of $connWnd_{end}$

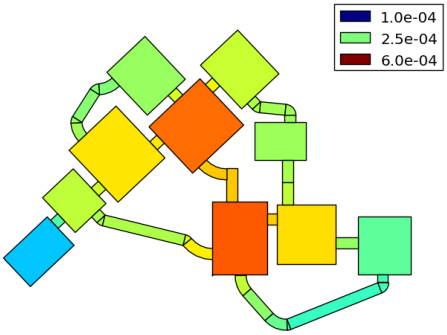
To test the influence of the $connWnd_{end}$ parameter we created 5 sets of 1000 maps each. The first set S_{far0} was generated setting $connWnd_{end}$ to 0, so that no connections are added in the *additional connection creation phase*. The second set S_{far25} was generated setting $connWnd_{end}$ to 0.25, so the algorithm tries to add the 25% of farthest connections. Sets S_{far50} and S_{far75} were generated setting $connWnd_{end}$ respectively to 0.5 and 0.75. S_{far100} was generated setting $connWnd_{end}$ to 1, so that the maximum number of connections are created. Figure 5.43 shows maps of the generated sets. Boxplots of the considered metrics are shown in figures from 5.44 to 5.49. Similarly to sets generated for the evaluation of $connWnd_{start}$, CC of complete graphs very similar for all sets. CC of simplified graphs decreases progressively when the 50% of connections and more are added. CFCC of complete graphs increases from S_{far0} to S_{far75}, with the most relevant difference between S_{far0} and S_{far_25}, while there is almost no difference between S_{far75} and S_{far100}. Therefore, the addition of the 25% of nearest connections in S_{far100}, with respect to S_{far75}, has very little effect on CFCC of complete graphs; similarly, S_{near0} and S_{near25} have very little difference for the same metric (Figure 5.40a). The results of the evaluations of $connWnd_{start}$ and $connWnd_{end}$ suggest that the algorithm is effective in creating additional connections as indicated by these two parameters.



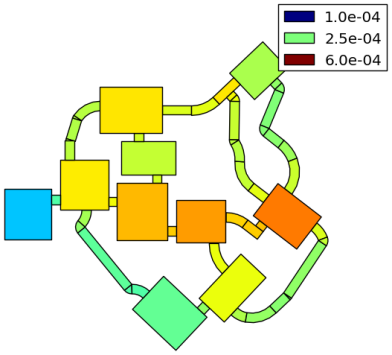
(a) S_{far0}



(b) S_{far25}



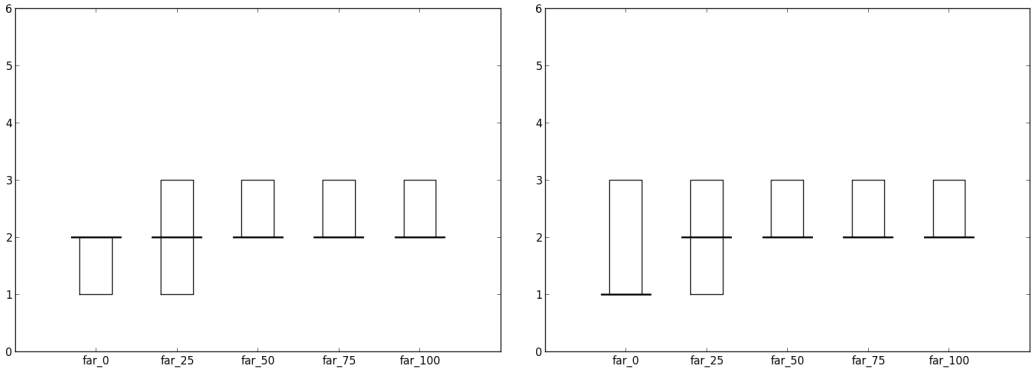
(c) S_{far50}



(d) S_{far75}

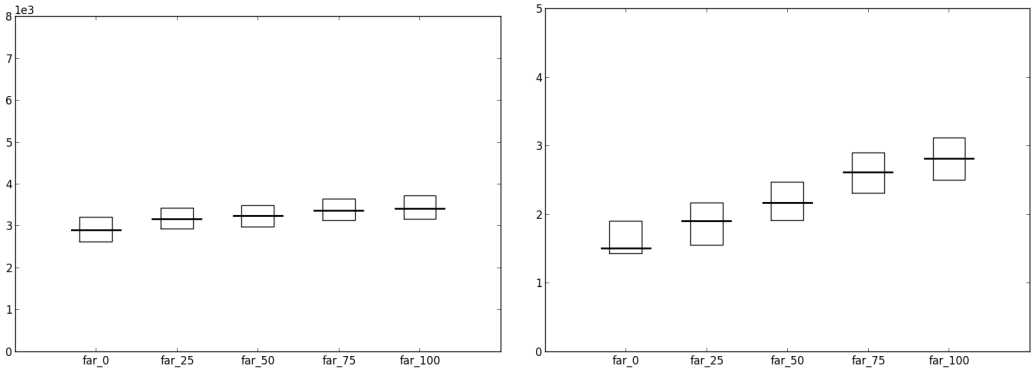
(e) S_{far100}

Figure 5.43: Maps generated for the evaluation of $connWnd_{end}$. Colors indicate CFCC of complete graphs.



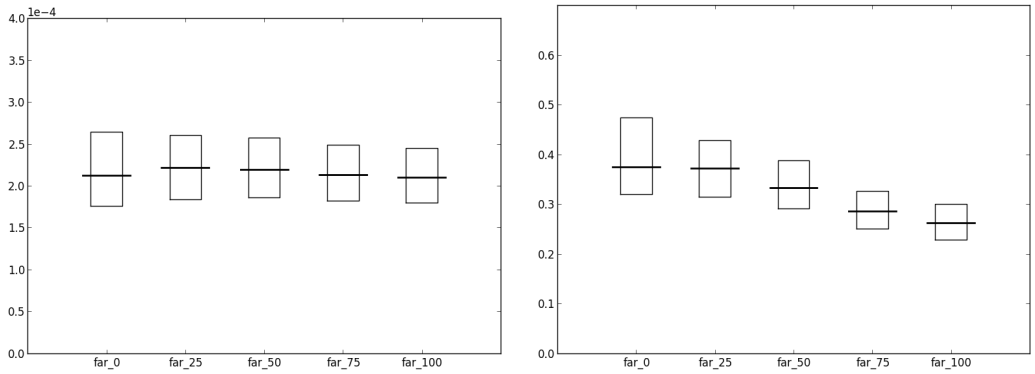
(a) degree, complete graph (b) degree, simplified graph

Figure 5.44: Boxplots of the degree metric for the evaluation of $connWnd_{end}$.



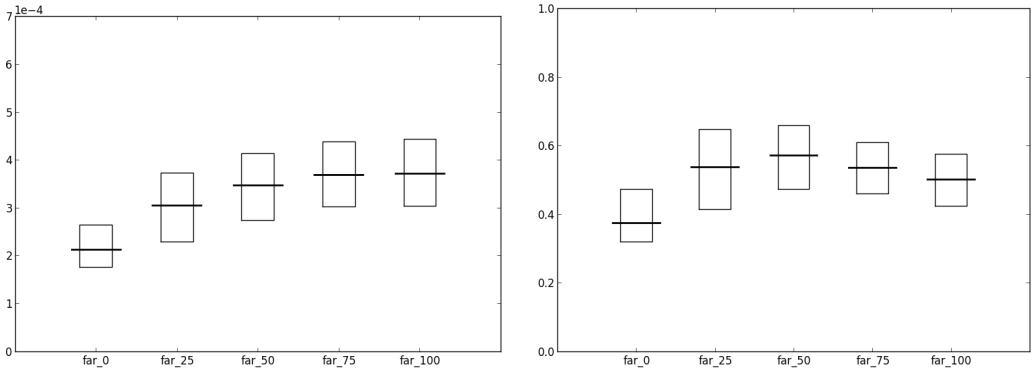
(a) radius, complete graph (b) radius, simplified graph

Figure 5.45: Boxplots of the radius metric for the evaluation of $connWnd_{end}$.



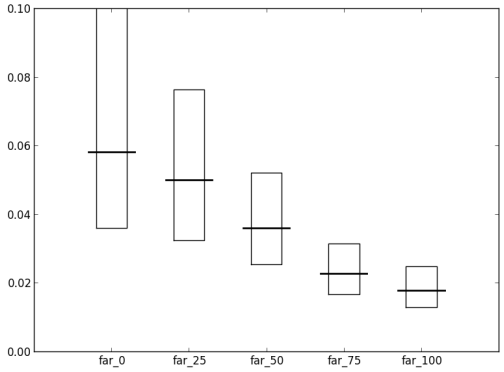
(a) CC, complete graph (b) CC, simplified graph

Figure 5.46: Boxplots of the CC metric for the evaluation of $connWnd_{end}$.



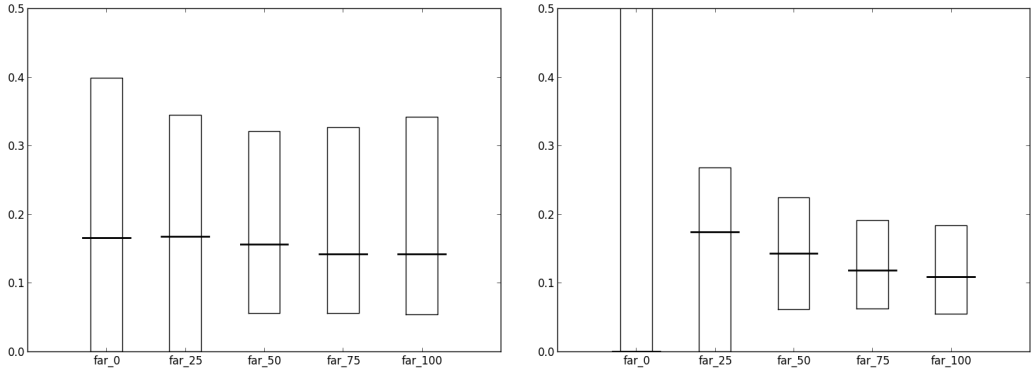
(a) CFCC, complete graph (b) CFCC, simplified graph

Figure 5.47: Boxplots of the CFCC metric for the evaluation of $connWnd_{end}$.



(a) RWCC, simplified graph

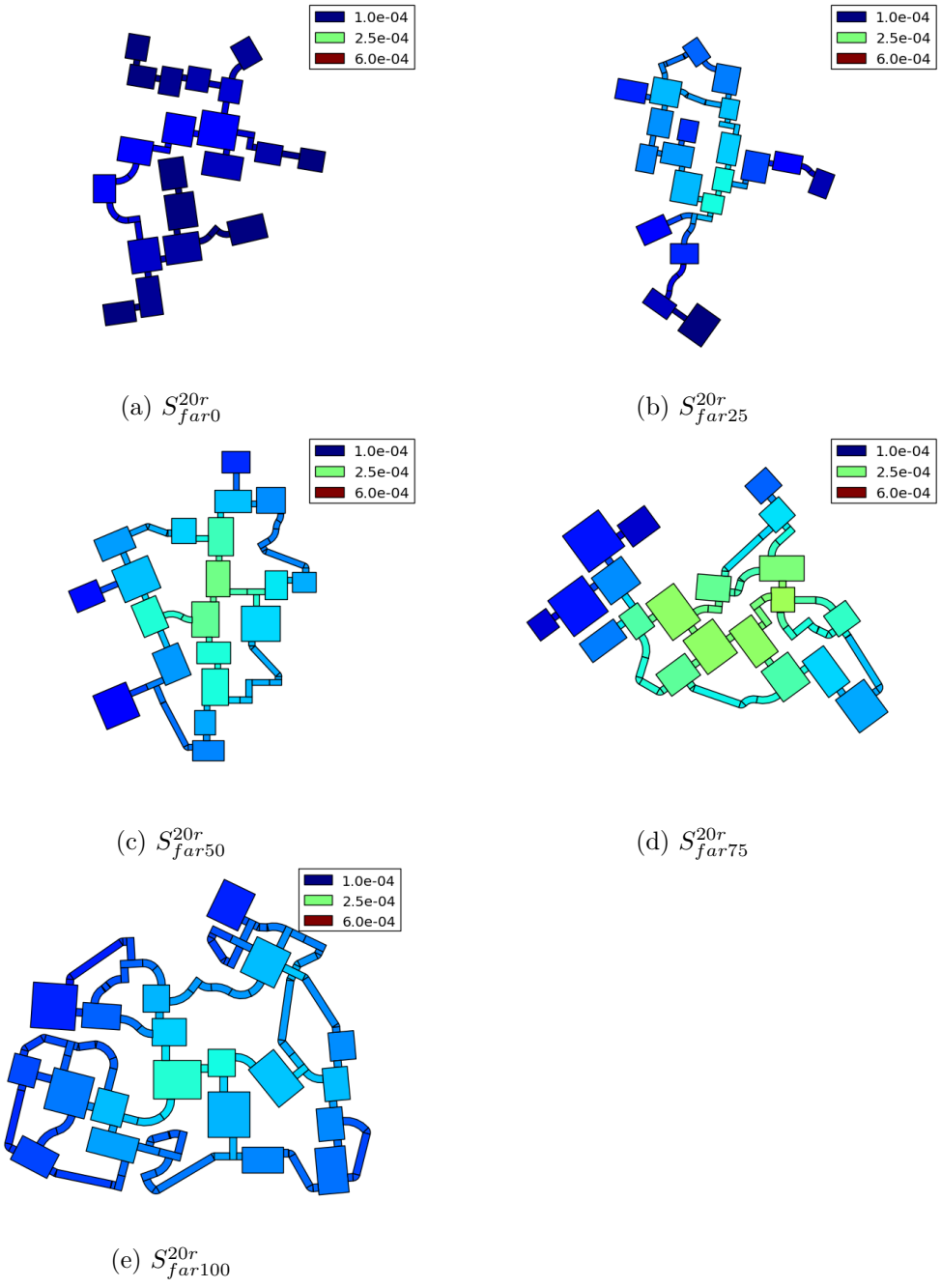
Figure 5.48: Boxplots of the RWCC metric for the evaluation of $connWnd_{end}$.



(a) BC, complete graph (b) BC, simplified graph

Figure 5.49: Boxplots of the BC metric for the evaluation of $connWnd_{end}$.

We also tested the influence of $connWnd_{end}$ in maps with 20 rooms. We generated 5 sets of maps ($S^{20r}_{far0}, S^{20r}_{far25}, S^{20r}_{far50}, S^{20r}_{far75}, S^{20r}_{far100}$) using the same percentages of farthest connections as in the evaluation of $connWnd_{end}$ with default parameters. The $maxRooms$ parameter was set to 20. Figure 5.50 shows maps of the generated sets. Boxplots of the considered metrics are shown in figures from 5.51 to 5.56. CC and CFCC of the generated sets have trends similar to sets generated for the evaluation of $connWnd_{end}$ with 10 rooms (figures 5.46 and 5.47); however, maps with 20 rooms have lower values than maps with 10 rooms because maps with 20 rooms are bigger, so their architectural elements are generally farther from each other. Sets of maps with 20 rooms have generally less sparse CC, CFCC, RWCC and BC values than corresponding sets with 10 rooms (figures from 5.46 to 5.49). For this reason, we suggest that increasing the number of rooms, and consequently the size of maps, tends to produce architectural elements with less sparse values of centrality metrics.

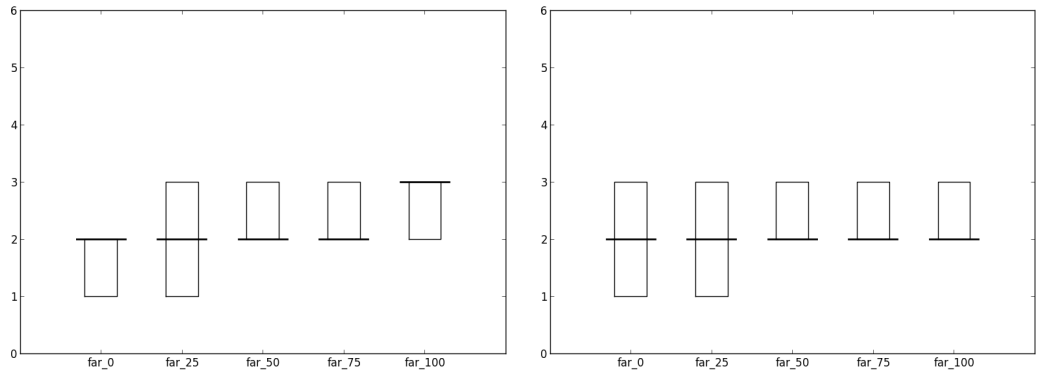


(a) S^{20r}_{far0} (b) S^{20r}_{far25}

(c) S^{20r}_{far50} (d) S^{20r}_{far75}

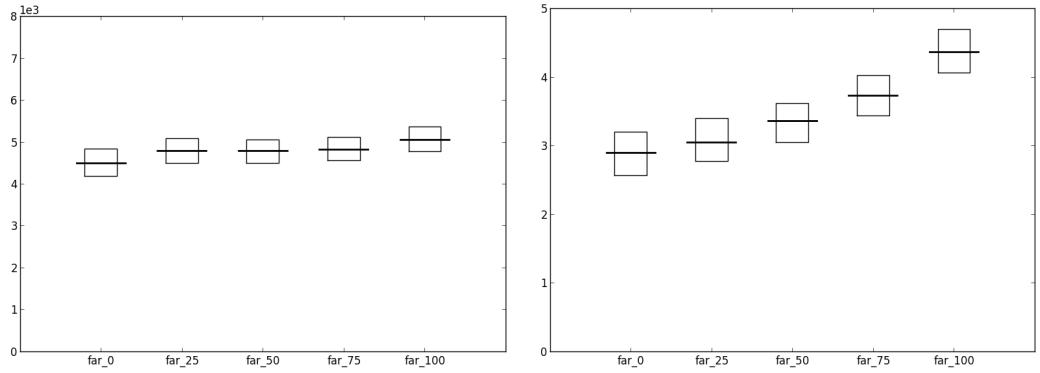
(e) S^{20r}_{far100}

Figure 5.50: Maps generated for the evaluation of $connWnd_{end}$ in maps with 20 rooms. Colors indicate CFCC of complete graphs.



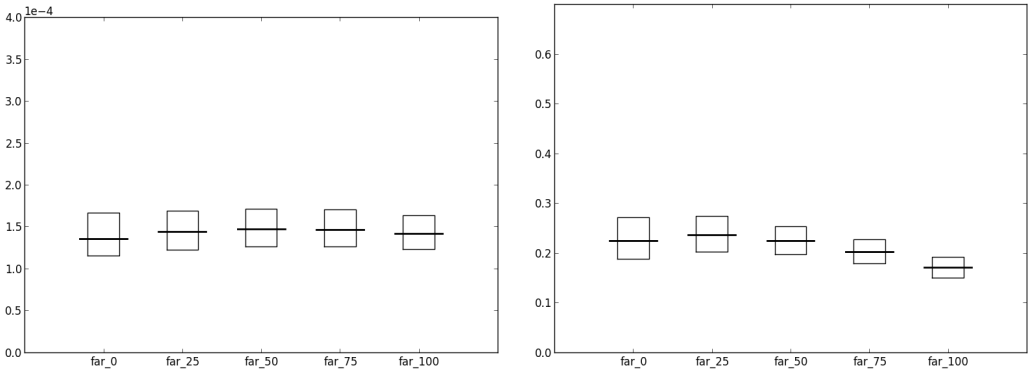
(a) degree, complete graph (b) degree, simplified graph

Figure 5.51: Boxplots of the degree metric for the evaluation of $connWnd_{end}$ in maps with 20 rooms.



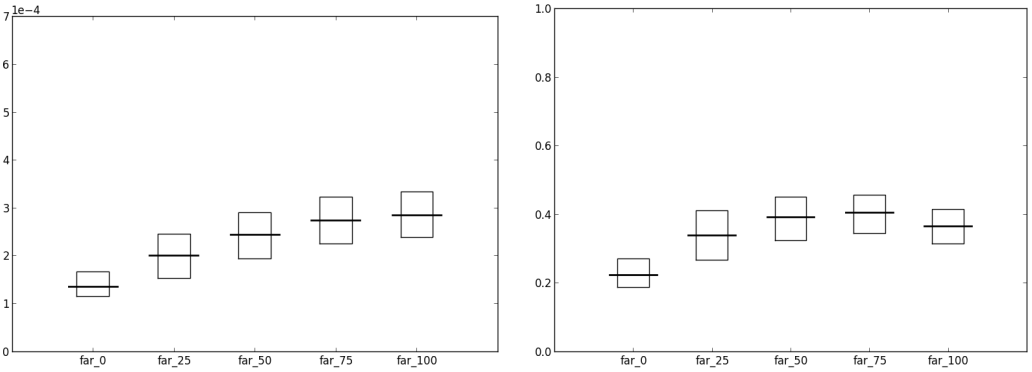
(a) radius, complete graph (b) radius, simplified graph

Figure 5.52: Boxplots of the radius metric for the evaluation of $connWnd_{end}$ in maps with 20 rooms.



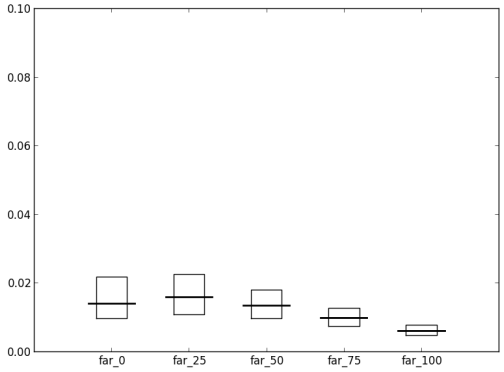
(a) CC, complete graph (b) CC, simplified graph

Figure 5.53: Boxplots of the CC metric for the evaluation of $connWnd_{end}$ in maps with 20 rooms.



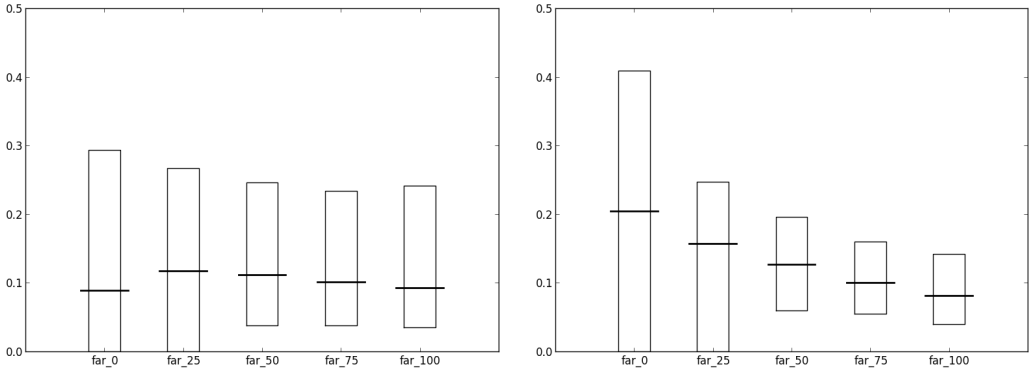
(a) CFCC, complete graph (b) CFCC, simplified graph

Figure 5.54: Boxplots of the CFCC metric for the evaluation of $connWnd_{end}$ in maps with 20 rooms.



(a) RWCC, simplified graph

Figure 5.55: Boxplots of the RWCC metric for the evaluation of $connWnd_{end}$ in maps with 20 rooms.

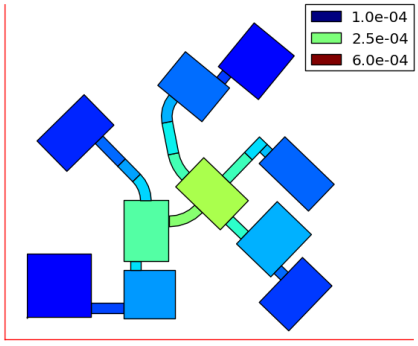


(a) BC, complete graph

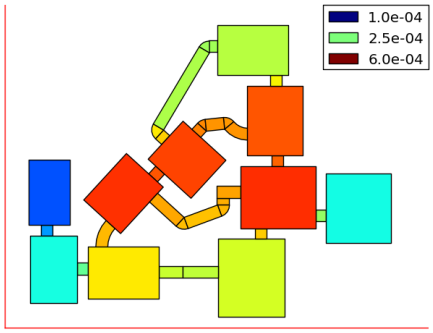
(b) BC, simplified graph

Figure 5.56: Boxplots of the BC metric for the evaluation of $connWnd_{end}$ in maps with 20 rooms.

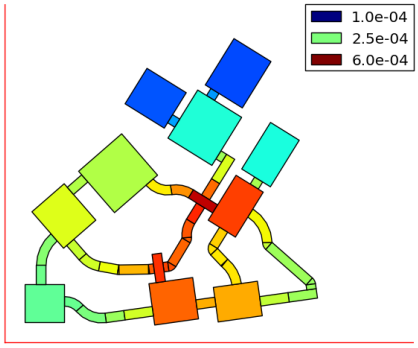
Finally, we tested the influence of $connWnd_{end}$ in maps generated using a rectangular *boundary*. The rectangular room in $initialElements$ was placed near the lower left corner of the boundary, so that maps could only grow upwards and to the right. We generated 5 sets of maps ($S^{co}_{far0}, S^{co}_{far25}, S^{co}_{far50}, S^{co}_{far75}, S^{co}_{far100}$) using the same percentages of farthest connections as in the evaluation of $connWnd_{end}$ with default parameters. Figure 5.57 shows maps of the generated sets. Despite the presence of the boundary, the maps do not show notable differences in the shape with respect to maps generated without constraints (Figure 5.43). Boxplots of the considered metrics are shown in figures from 5.58 to 5.63. The first quartile of BC of S^{co}_{far25} is not zero, while it is zero for the set S_{far25} generated without boundary (Figure 5.49). Other metrics in general does not show differences with respect to sets generated without boundary (figures from 5.44 to 5.48). These results suggest that placing the initial room in the corner of a rectangular boundary does not affect significantly the appearance and the metric values of the generated maps.



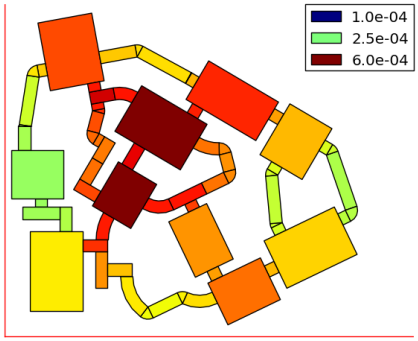
(a) S^{co}_{far0}



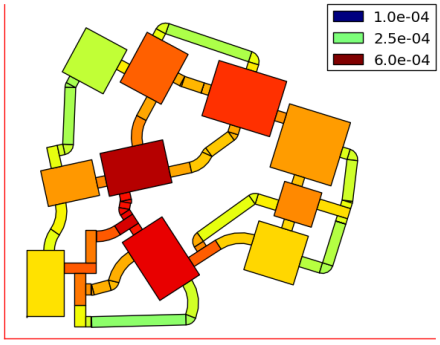
(b) S^{co}_{far25}



(c) S^{co}_{far50}

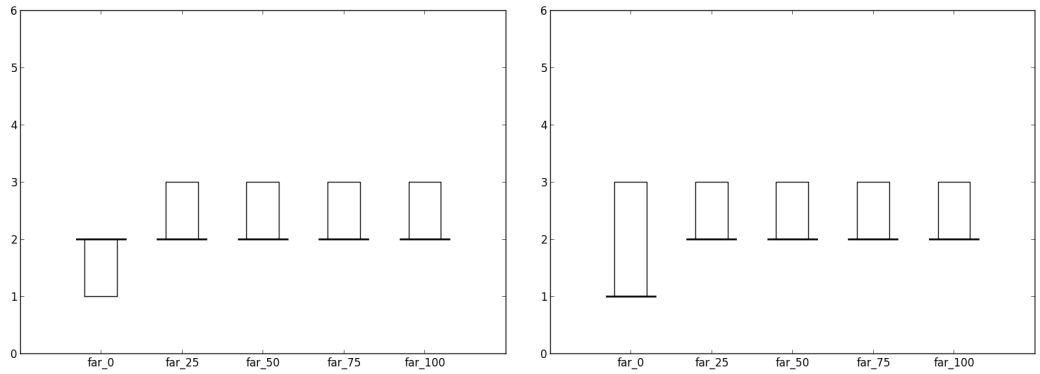


(d) S^{co}_{far75}



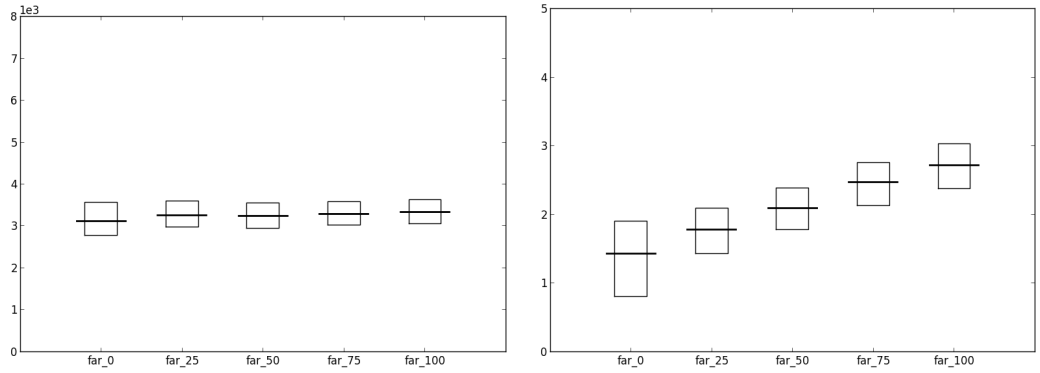
(e) S^{co}_{far100}

Figure 5.57: Maps generated for the evaluation of $connWnd_{end}$ in maps with the initial element in a corner of a rectangular boundary (drawn in red). Colors indicate CFCC of complete graphs.



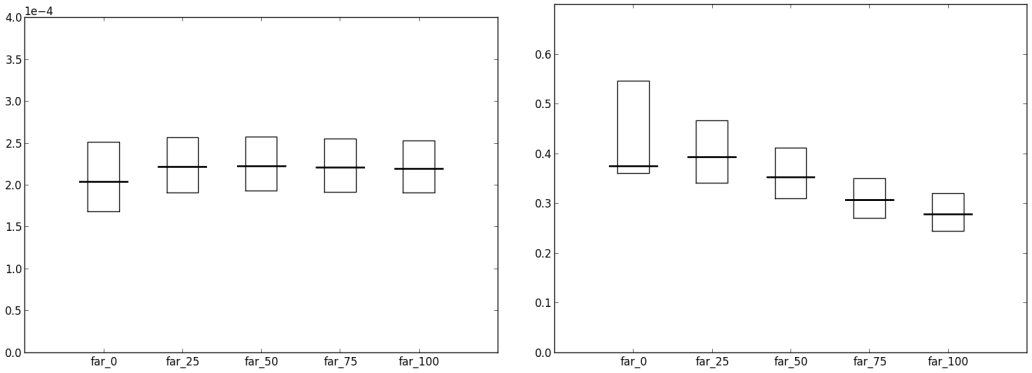
(a) degree, complete graph (b) degree, simplified graph

Figure 5.58: Boxplots of the degree metric for the evaluation of $connWnd_{end}$ in maps with the initial element in a corner of a rectangular boundary.



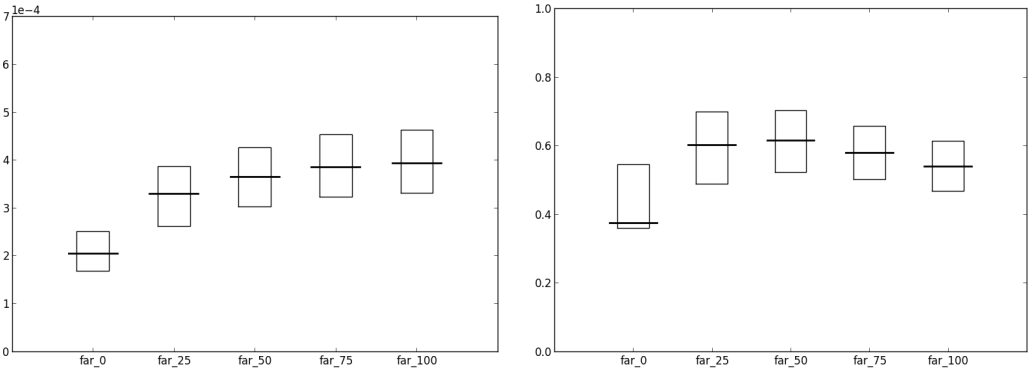
(a) radius, complete graph (b) radius, simplified graph

Figure 5.59: Boxplots of the radius metric for the evaluation of $connWnd_{end}$ in maps with the initial element in a corner of a rectangular boundary.



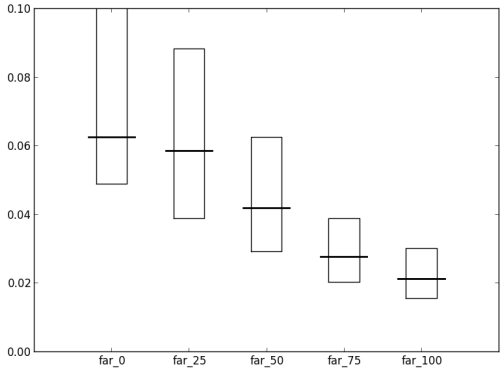
(a) CC, complete graph (b) CC, simplified graph

Figure 5.60: Boxplots of the CC metric for the evaluation of $connWnd_{end}$ in maps with the initial element in a corner of a rectangular boundary.



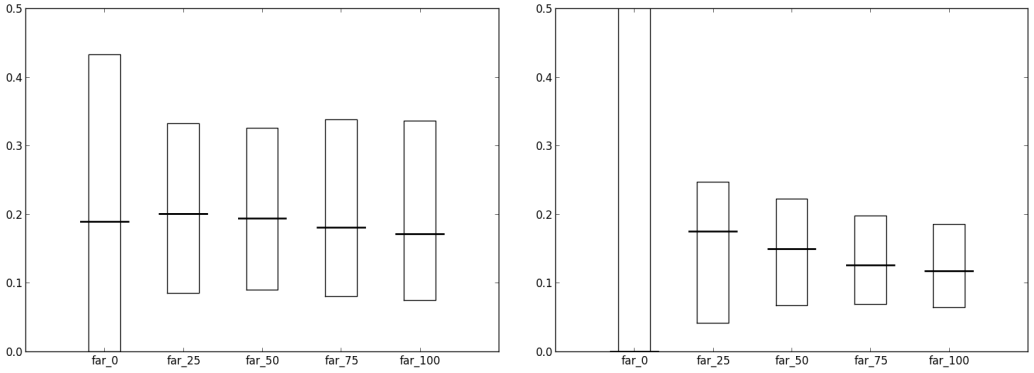
(a) CFCC, complete graph (b) CFCC, simplified graph

Figure 5.61: Boxplots of the CFCC metric for the evaluation of $connWnd_{end}$ in maps with the initial element in a corner of a rectangular boundary.



(a) RWCC, simplified graph

Figure 5.62: Boxplots of the RWCC metric for the evaluation of $connWnd_{end}$ in maps with the initial element in a corner of a rectangular boundary.



(a) BC, complete graph

(b) BC, simplified graph

Figure 5.63: Boxplots of the BC metric for the evaluation of $connWnd_{end}$ in maps with the initial element in a corner of a rectangular boundary.

5.4 Evaluation of parameter combinations

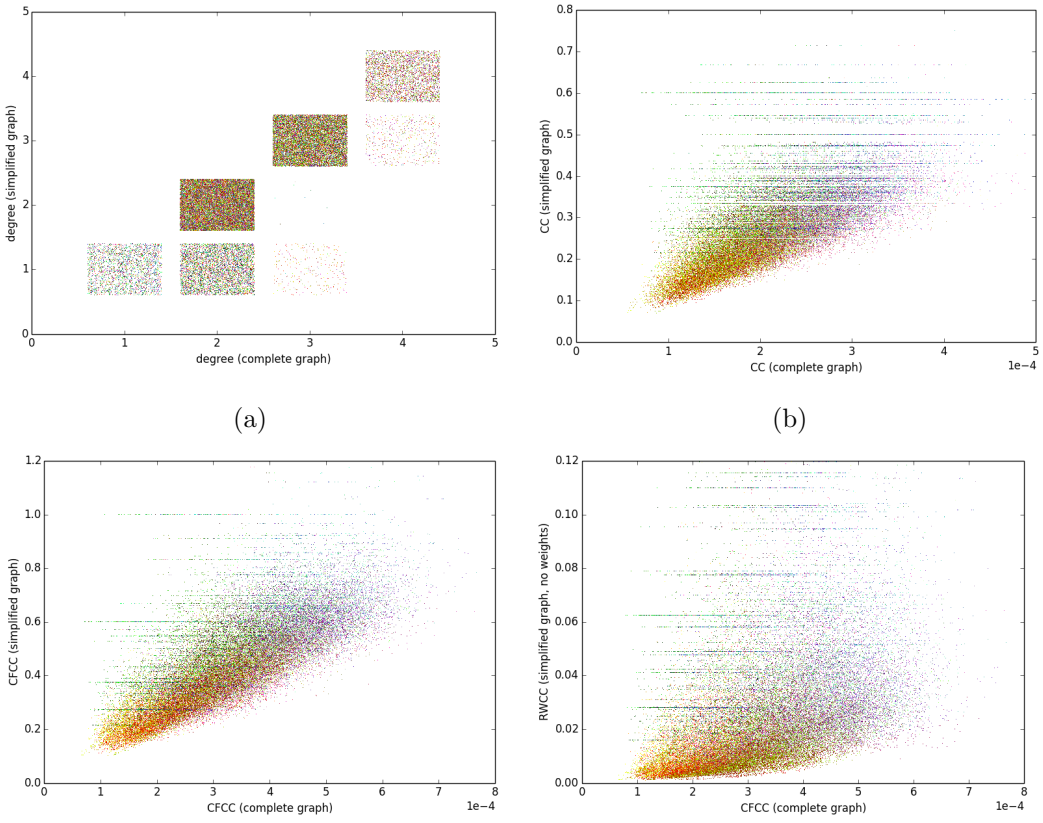
We created configurations of parameters combining different values of $prob_{classes}$, $connWnd_{start}$, $connWnd_{end}$, $prob_{degrees}$, $boundary$; other parameters were set to their default value (Table 5.3). The values we used for each parameter are shown in Table 5.8. We chose 3 values for $prob_{classes}$: the first gives a 10% probability to rooms and a 90% probability to corridors, the second 25% and 75%, the third 50% and 50%. For $connWnd_{start}$ and $connWnd_{end}$ we chose 8 pairs of values, which create respectively: no additional connections; 25%, 50%, 75% of farthest connections; all connections; 75%, 50%, 25% of nearest connections. We chose 3 values for $prob_{degrees}$: the first gives to degrees from 3 to 1 probabilities with a ratio of 100, the second probabilities with a ratio of 4, the third probabilities with a ratio of 1. For $boundary$ we chose 2 values: the first assigns no boundary, the second assigns a rectangular boundary that has the initial room placed by the algorithm in its lower left corner. In total, there are $3 \times 8 \times 3 \times 2 = 144$ combinations of parameters; for each combination, we generated a set of 20 maps. We created scatterplots[7] representing metric values of these maps, shown in Figure 5.64. Scatterplots have on the x axis values related to complete graphs, and on the y axis values related to simplified graphs (except for the RWCC metric which is shown together with CFCC). Points correspond to architectural elements, except for the radius metric, for which points correspond to maps. Colors of points depend on the combinations of parameters used to generate the related maps (see Table 5.9). For the degree metric it is shown that most architectural elements are concentrated at degrees 2 and 3. The majority of architectural elements have the same degree value for both graphs, which means that they do not change degree value when transforming complete graphs in simplified graphs. Instead, most of architectural elements with degree 2 in the complete graph have degree 1 in the simplified graph. These are architectural elements in the middle of branches of maps, which become leaf nodes in simplified graphs. For CC and CFCC, it is shown a linear dependence between values of complete graphs and values of simplified graphs.

We also generated scatterplots with colors representing single parameters. Figure 5.65 shows scatterplots with colors denoting only the $prob_{classes}$ param-

[7] `http://en.wikipedia.org/wiki/Scatter_plot`

Property	V_{index}	**Values**
$prob_{classes}$	1	Rectangular room: 10% Linear corridor: 45% Curved corridor: 45%
	2	Rectangular room: 25% Linear corridor: 37.5% Curved corridor: 37.5%
	3	Rectangular room: 50% Linear corridor: 25% Curved corridor: 25%
$connWnd_{start}, connWnd_{end}$	1	0, 0
	2	0, 0.25
	3	0, 0.5
	4	0, 0.75
	5	0, 1
	6	0.25, 1
	7	0.5, 1
	8	0.75, 1
$prob_{degrees}$	1	1: 0.01% 2: 0.99% 3: 99%
	2	1: 5% 2: 19% 3: 76%
	3	1: 33% 2: 33% 3: 34%
boundary	1	*None*
	2	Rectangular boundary, the initial room is placed in the lower left corner of the boundary

Table 5.8: Parameter values used for the evaluation of parameter combinations.



(a) (b)

(c) (d)

Figure 5.64: Scatterplots for the evaluation of parameter combinations.

Given a combination C of the parameter values shown in Table 5.8, the color of the corresponding points in scatterplots is calculated as follows:

1. Let $C(p)$ be the value of the parameter p in the combination C. Referring to Table 5.8, let $V_{index}(v)$ be the index of the parameter value v; let $V_{maxIndex}(p)$ be the maximum index of the values of the parameter p.
2. We define the $v_{index}(p)$ function to map indices values to the range $[0, 1]$:
$$v_{index}(p) = \frac{V_{index}(C(p)) - 1}{V_{maxIndex}(p) - 1}$$
3. The RGB components of the color associated to C are
$$R = v_{index}((connWnd_{start}, connWnd_{end}))$$
$$G = 0.4 \times v_{index}(prob_{degrees}) + 0.6 \times v_{index}(boundary)$$
$$B = 0.75 \times v_{index}(prob_{classes})$$

Table 5.9: Procedure to calculate the colors used in scatterplots for combinations of parameter values.

eter. For the radius metric colors show that, along the complete graph axis, the radius increases along with the corridor probability. Instead, along the simplified graph axis, colors are more sparse. Similarly, the scatterplot of the median CC value of each map shows that colors are more separated along the complete graph axis than the simplified graph axis. Instead, for CFCC colors are equally sparse on both axes. We suggest that radius and CC of complete graphs depend on corridor probability more strictly than simplified graphs.

Figure 5.66 shows scatterplots with colors denoting only the $prob_{degrees}$ parameter. For all of the considered metrics colors are evenly distributed on the same area and no grouping can be observed.

Figure 5.67 shows scatterplots with colors indicating whether maps were generated without constraints or placing the initial room near the corner of a rectangular *boundary*. Also in this case, no color grouping is visible. This suggests that placing the initial room in the corner of a rectangular boundary does not affect significantly the metric values, as shown in the evaluation of $connWnd_{end}$.

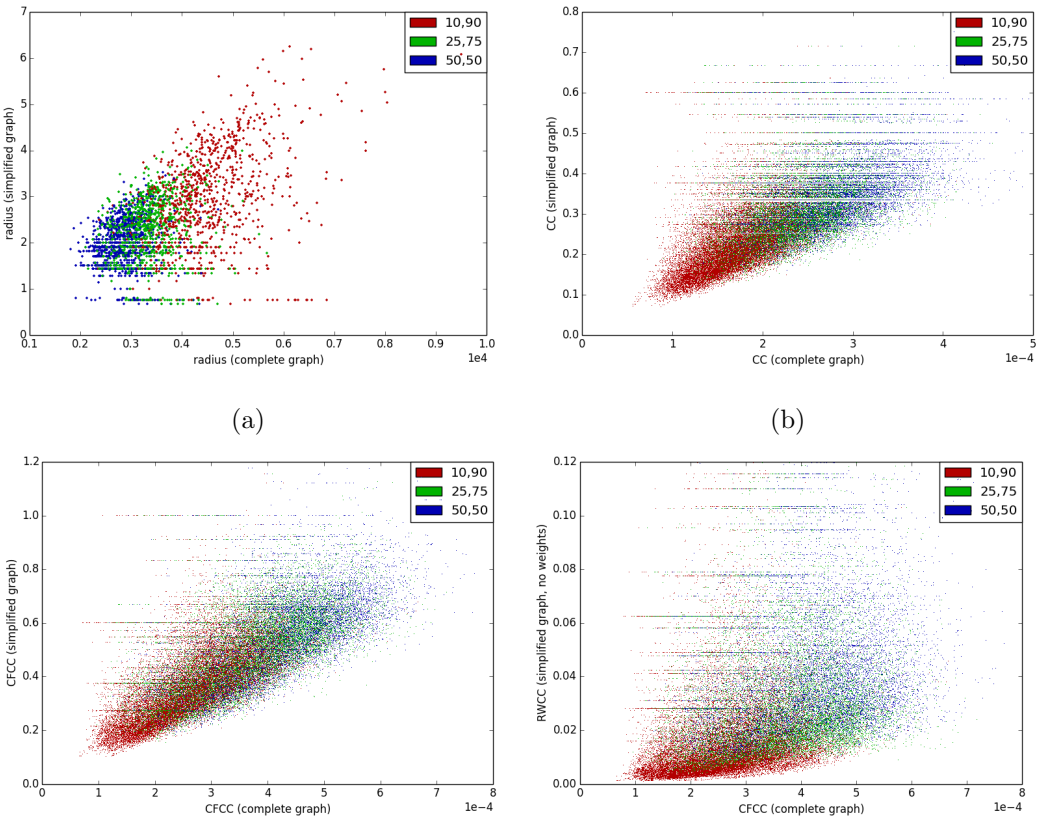


Figure 5.65: Scatterplots for the evaluation of parameter combinations. Colors depend only on the $prob_{classes}$ parameter. The legend indicates the probabilities given to rooms and corridors (in the order).

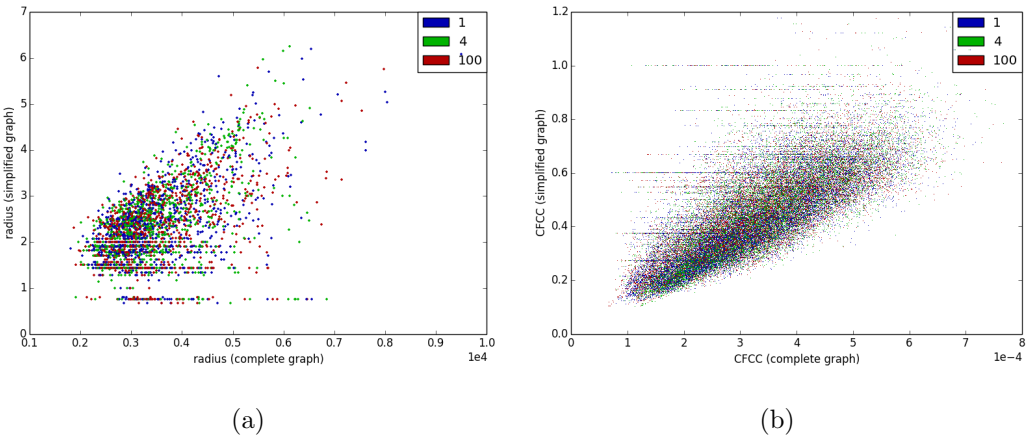
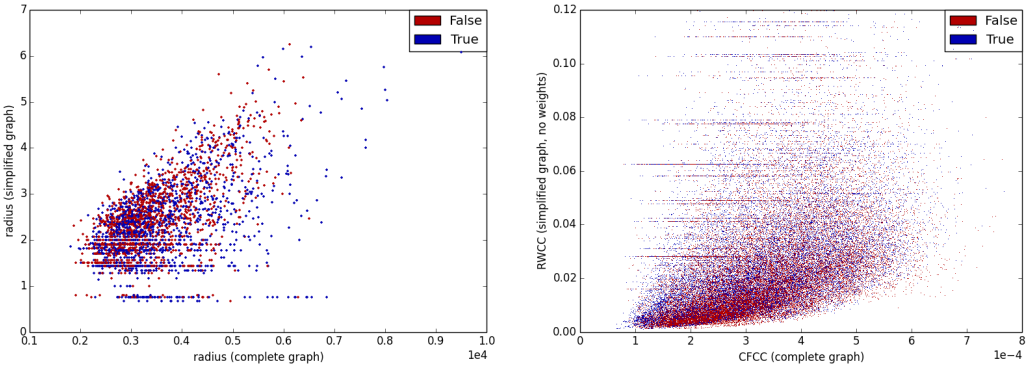


Figure 5.66: Scatterplots for the evaluation of parameter combinations. Colors depend only on the $prob_{degrees}$ parameter. The legend indicates the probabilities given to rooms and corridors (in the order).



(a) (b)

Figure 5.67: Scatterplots for the evaluation of parameter combinations. Colors depend only on the *boundary* parameter. In the legend, *False* indicates that the map was generated with no boundary, *True* indicates that the initial room was placed near the corner of a rectangular boundary.

5.5 Summary

We have briefly discussed works that deal with the evaluation of maps with respect to particular metrics. Then, we introduced the metrics we used to analyze the maps generated by our algorithm. We have described the procedure we used for the evaluation of single parameters of our algorithm. For each of the evaluated parameters, we showed examples of generated maps and boxplots of the considered metrics, commenting them. Finally, we have described the evaluation of combinations of parameters, showing and discussing scatterplots of the considered metrics.

Chapter 6

Extensions of the algorithm and analysis

Our algorithm proved to be effective in generating useful maps, however the analysis showed that most of the parameters give limited control on the map generation process. Accordingly, we finally explored two extensions of our original algorithm, which could provide better controls to guide the map generation.

6.1 Extensions of our algorithm

We introduced two extensions to our algorithm to gain more control on generated maps: the *root eliminating procedure* and the *grid placement procedure*. In the previous analysis, we provided a rectangular room in the $initialElements$ parameter, which was used as the root of the tree built in the *digging phase*. Root elements are likely to be in the middle of the map and to have many connections, thus inducing high centrality values for many architectural elements. To generate a new variety of maps, we extended our original algorithm introducing the *root eliminating procedure*, that removes from the map the initial room. The *root eliminating procedure* is executed between the *additional connection creation phase* and the *dead end pruning phase*.

The *grid placement procedure* is capable of occupying the space more uniformly than our original algorithm. The *grid placement procedure* is a grid-based uniform generator, i.e. it first places rooms in a grid layout and then

creates corridors to connect them. It takes as input a number of rooms R: first, it subdivides the plane in a grid layout and places R rectangular rooms in neighboring cells of the grid; then it executes the *additional connection creation phase* of our original algorithm to create the maximum number of connections; finally, it removes disconnected parts of the map executing the *isolated parts pruning phase* of our original algorithm.

6.2 Evaluation of the extensions

We evaluated the extensions using the same procedure as in the previous analysis. We generated 3 sets of 1000 maps each: the first (S_{orig}) using our original algorithm with the parameters shown in Table 5.3, the second (S_{rootEl}) using the *root eliminating procedure* with the same parameters, the third (S_{grid}) using the *grid placement procedure*. All the maps of the three sets have 10 rooms.

Figure 6.1 shows maps of S_{rootEl}, showing also how these maps would be if the root element was not eliminated. In the first two maps (figures from 6.1a to 6.1d) the remotion of the root element produces an overall decrease of the CFCC of the architectural elements. Instead, in the third map (figures 6.1e and 6.1f) the root element is in a peripheral position, and its remotion produces an increase of the CFCC of most architectural elements.

Figure 6.2 shows maps generated with the *grid placement procedure*. It can be noted that these maps have generally lower CFCC values than maps generated by our original algorithm, even if in both cases the maximum number of connections is created. This is due to the more uniform distribution of the rooms performed by the *grid placement procedure*. It can also generate maps with large linear parts, as shown in figures 6.2e and 6.2f.

Figures from 6.3 to 6.8 show boxplots of the three sets of maps. Complete graphs of S_{grid} have all the quartiles of the degree metric at 2. This indicates that the *grid placement procedure* is likely to produce linear parts in maps. The radius metric of complete graphs increases along sets in the order S_{orig}, S_{rootEl}, S_{grid}, with little difference between S_{rootEl} and S_{grid}. The CC metric of complete graphs decreases along sets in the same order, again with little difference between S_{rootEl} and S_{grid}. For CFCC the trend is similar to CC, but with greater differences among the sets. Instead, for simplified graphs, S_{grid} has low radius and high CC, CFCC, and RWCC. This is because of the linear parts

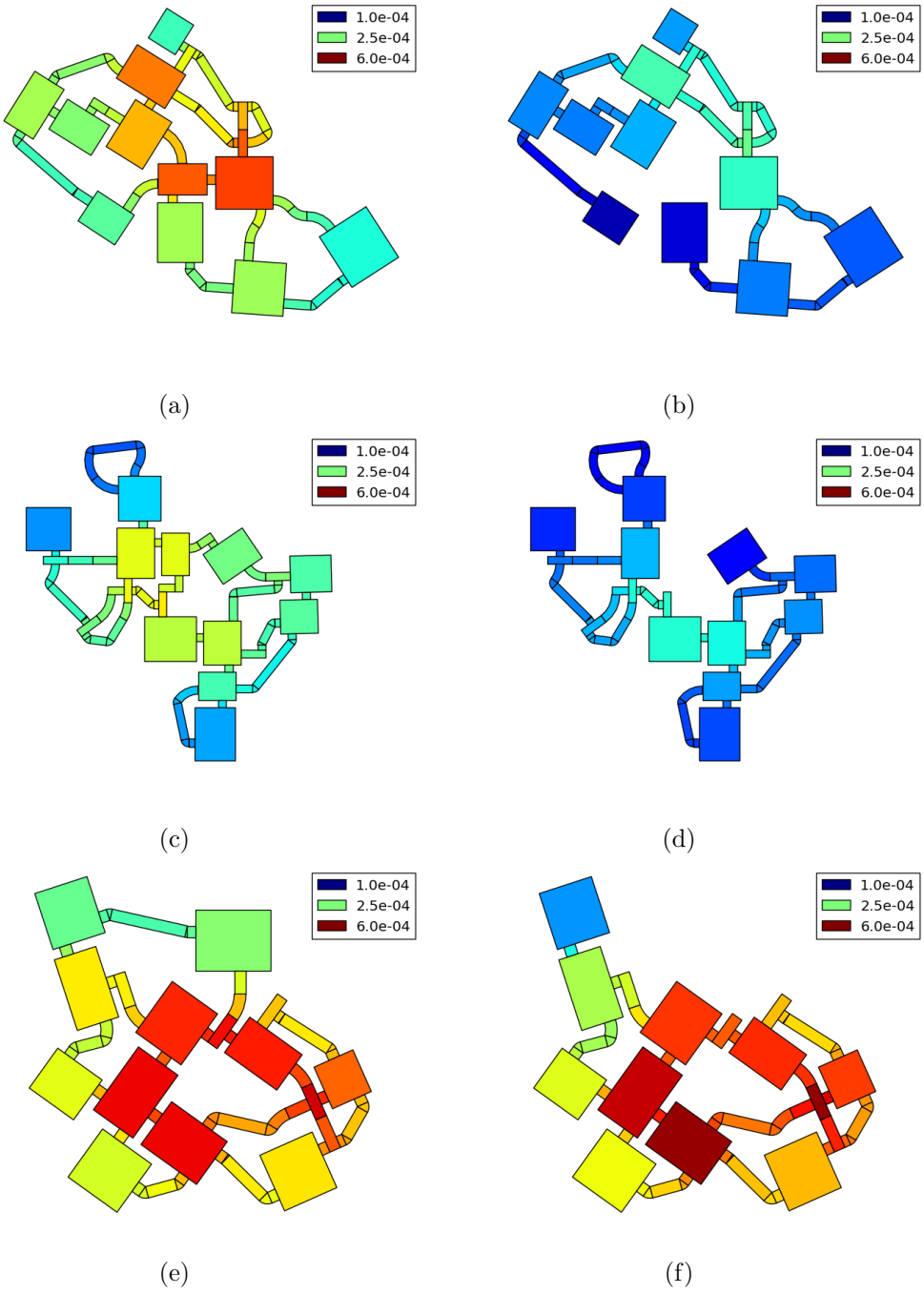


Figure 6.1: On the left: maps in which the root element was not removed. On the right: the same maps generated with the *root eliminating procedure*. Colors represent CFCC of complete graphs.

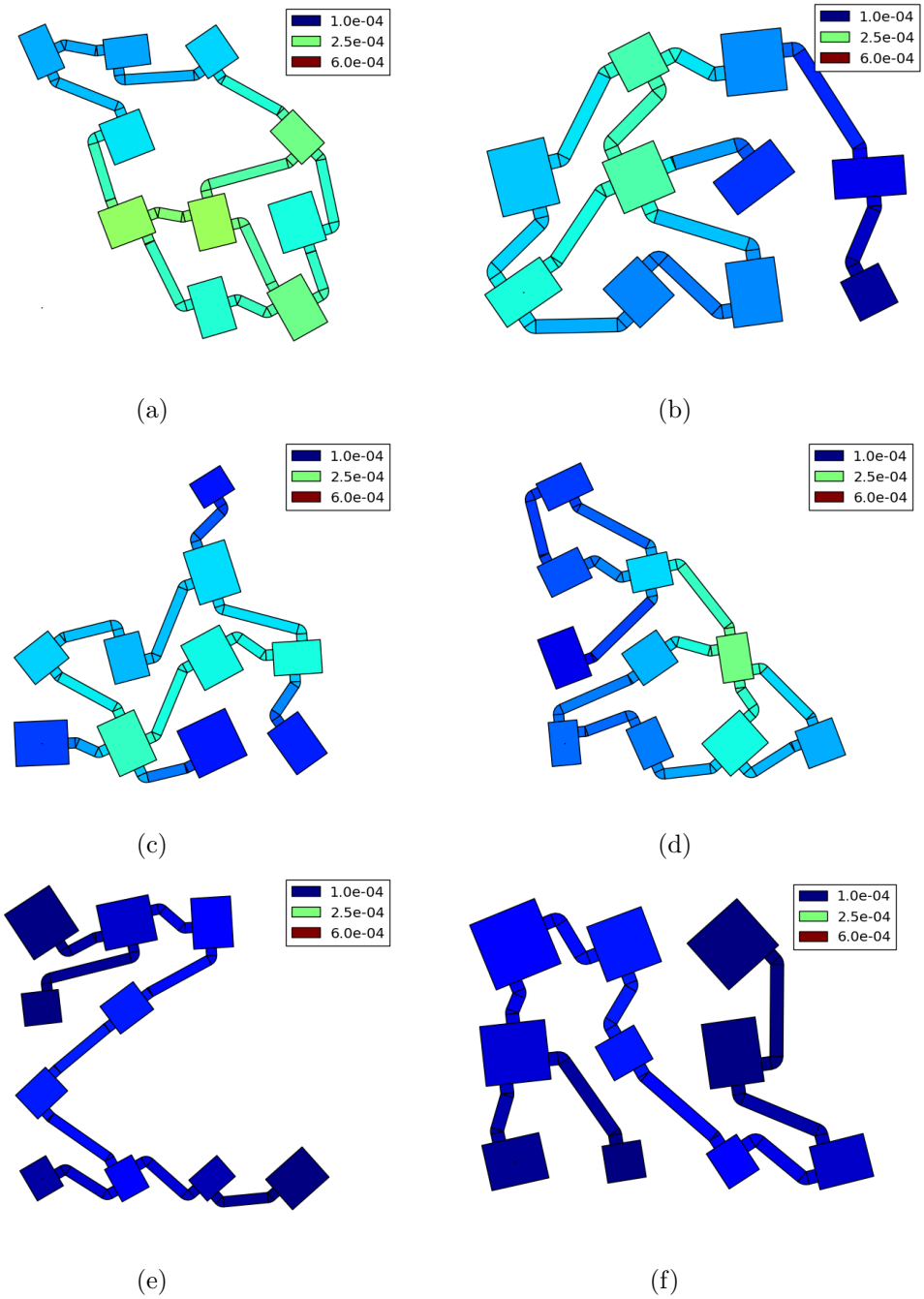
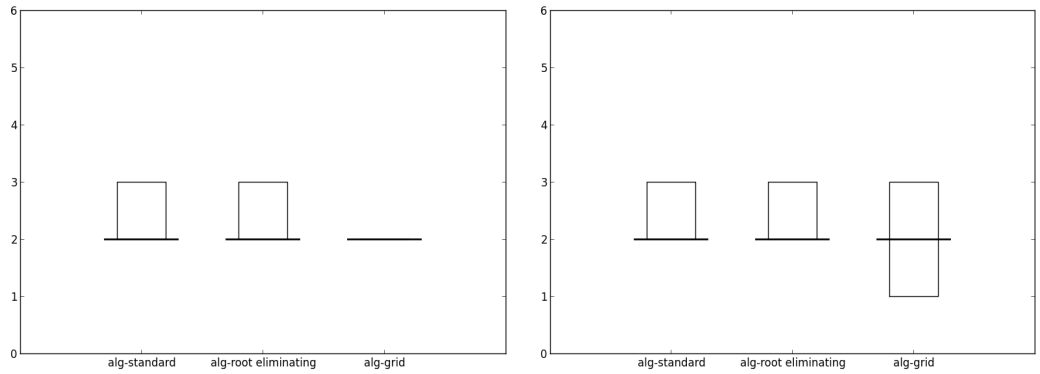
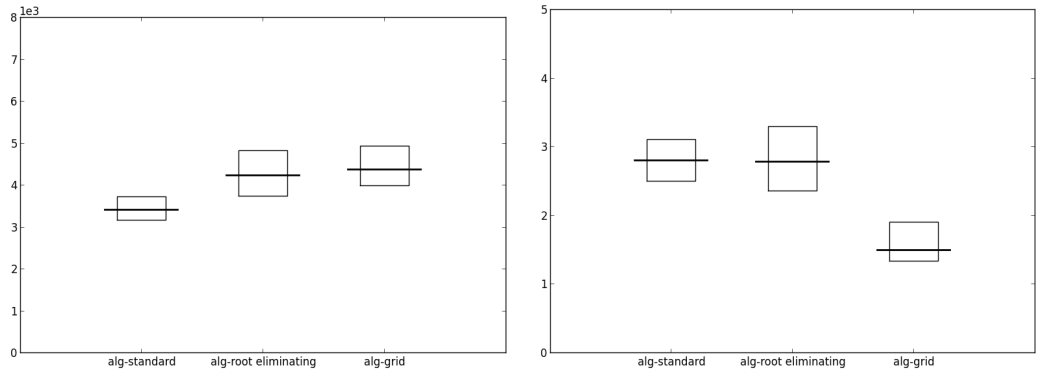


Figure 6.2: Maps generated with the *grid placement procedure*. Colors represent CFCC of complete graphs.



(a) degree, complete graph (b) degree, simplified graph

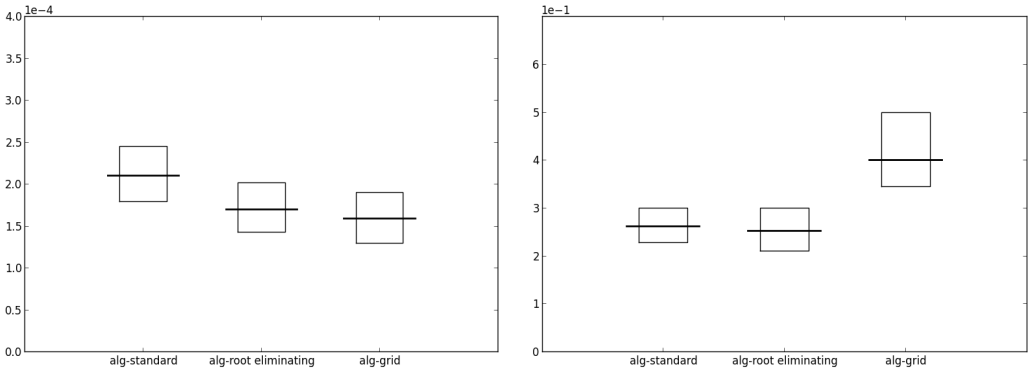
Figure 6.3: Boxplots of the degree metric for the evaluation of the extensions of the map generation algorithm.



(a) radius, complete graph (b) radius, simplified graph

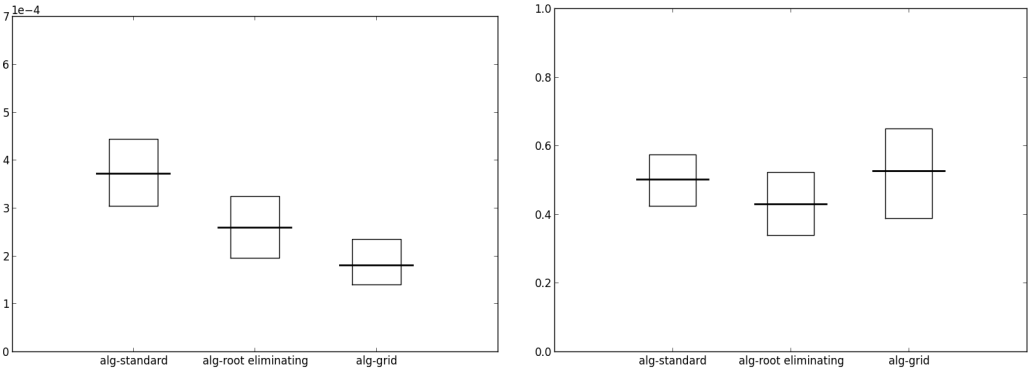
Figure 6.4: Boxplots of the radius metric for the evaluation of the extensions of the map generation algorithm.

of maps generated by the *grid placement procedure*, which are transformed into single nodes in simplified graphs. Thus, these simplified graphs are small, so their nodes have high CC, CFCC and RWCC values. For the same reason, the first quartile of the BC metric of S_{grid} is 0, since small simplified graphs tend to have a high percentage of leaf nodes, which are not crossed by any path.



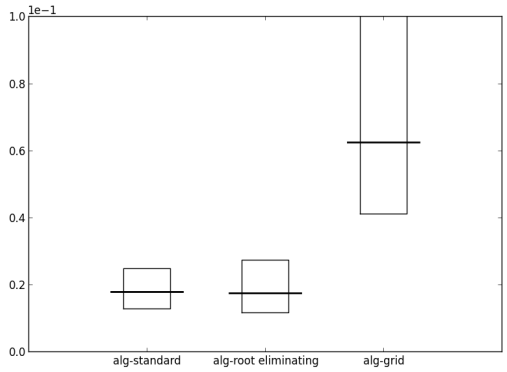
(a) CC, complete graph (b) CC, simplified graph

Figure 6.5: Boxplots of the CC metric for the evaluation of the extensions of the map generation algorithm.



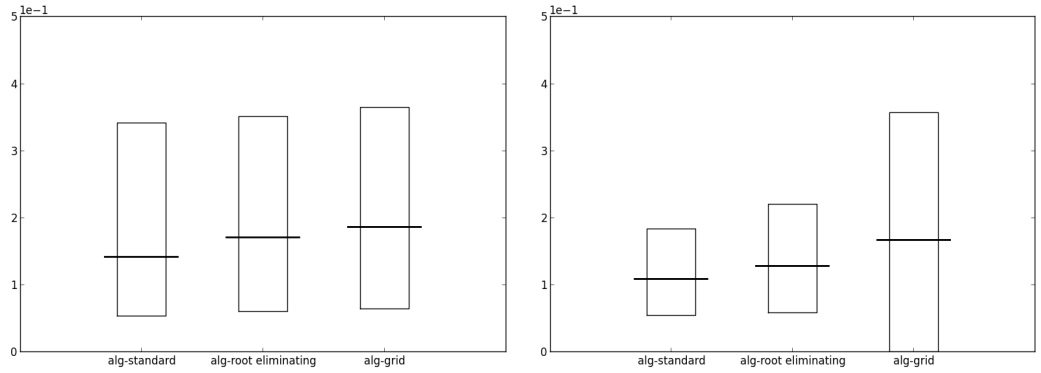
(a) CFCC, complete graph (b) CFCC, simplified graph

Figure 6.6: Boxplots of the CFCC metric for the evaluation of the extensions of the map generation algorithm.



(a) RWCC, simplified graph

Figure 6.7: Boxplots of the RWCC metric for the evaluation of the extensions of the map generation algorithm.



(a) BC, complete graph

(b) BC, simplified graph

Figure 6.8: Boxplots of the BC metric for the evaluation of the extensions of the map generation algorithm.

Figure 6.9 shows scatterplots of the three sets of maps. Colors of points indicate the set they are related to. For the radius metric, points of S_{rootEl} occupy a wider area than points of S_{orig}. Points of the S_{grid} extend to very low radius values along the simplified graph axis, because the related maps contain many linear parts. In scatterplots of CC, CFCC, S_{orig} and S_{rootEl} occupy areas of similar size and show a linear dependence between values of complete graphs and values of simplified graphs; points of S_{grid} do not show a linear dependence. In scatterplots of CC, CFCC and RWCC, the area taken by S_{rootEl} is shifted toward lower values than the area taken by S_{orig}.

Basing on the results, we suggest that *root eliminating procedure* and the *grid placement procedure* actually introduce two different kinds of variations with respect the original algorithm. By eliminating the root element, generated maps tend to have greater and more sparse radius values. The overall CC and CFCC of the maps are effectively reduced; however, it was shown that when the root is a peripheral element, its remotion can have the opposite effect, increasing the CC and CFCC of most the architectural elements of the map. A more efficient way to reduce CC and CFCC could be to remove the element with the highest centrality, instead of the root element. The *grid placement procedure*, by placing rooms in a grid layout, generates maps with shape different from maps created from a tree-like structure; moreover, CC and CFCC tend to be even lower than with the *root eliminating procedure*. The *grid placement procedure* produces both highly connected maps and linear maps; it would be useful to extend it introducing parameters to control the type of the generated maps.

6.3 Summary

We have described the *root eliminating procedure* and the *grid placement procedure*, that extend our map generation algorithm giving more control on the generation process. Then, we have analyzed maps generated by the two extensions, presenting boxplots and scatterplots of the considered metrics. Our results showed that the two extensions actually introduce two different kinds of variations with respect to the original algorithm.

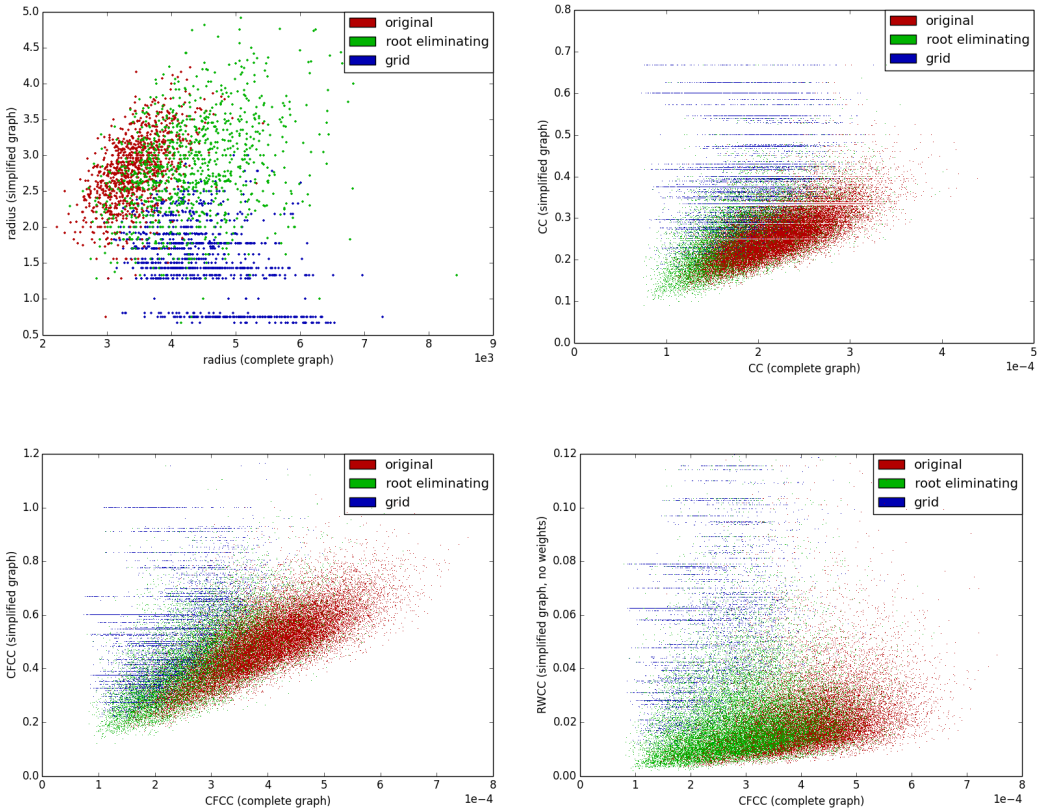


Figure 6.9: Scatterplots for the evaluation of the extensions of the map generating algorithm.

Chapter 7

Conclusions and Future Works

In this thesis, we have dealt with procedural content generation (PCG) in games. We have discussed its applications, from early games that used PCG to deal with strict memory constraints, to recent games that use PCG to reduce costs of creating large amount of content. Then, we have described techniques for the generation of outdoor environments and, more extensively, of indoor environments. We have focused on the automatic generation of large amount of content for an high end game, *In Verbis Virtus*. In particular, we have described an algorithm to generate dungeon-like maps. Starting from an initial architectural element, the algorithm first creates a tree-like map, then it adds more connections between the architectural elements to obtain a more general topology. We have presented an extensive analysis of the algorithm that showed how the parameters affect the generated maps. Then, we have introduced two extensions of the algorithm to get more control on the generation process. In particular, we wanted to avoid the bias due to the high centrality of the initial architectural element. We have presented the results of the analysis of the two extensions, showing that both help to avoid the bias and effectively introduce variations with respect to the original algorithm.

We created a graphical interface to easily generate maps with the algorithm (Figure 7.1); for each parameter, we allow to select values only in the range that was proved to be useful in the evaluation process. We added new types of architectural elements: oval rooms and irregular rooms (Figure 7.2). Figure 7.3 shows a floor plan generated with the graphical interface configured as in Figure 7.1. To test the generated maps in the game, we also created an exporter that converts the floor plan generated by the algorithm into a 3D map, in the

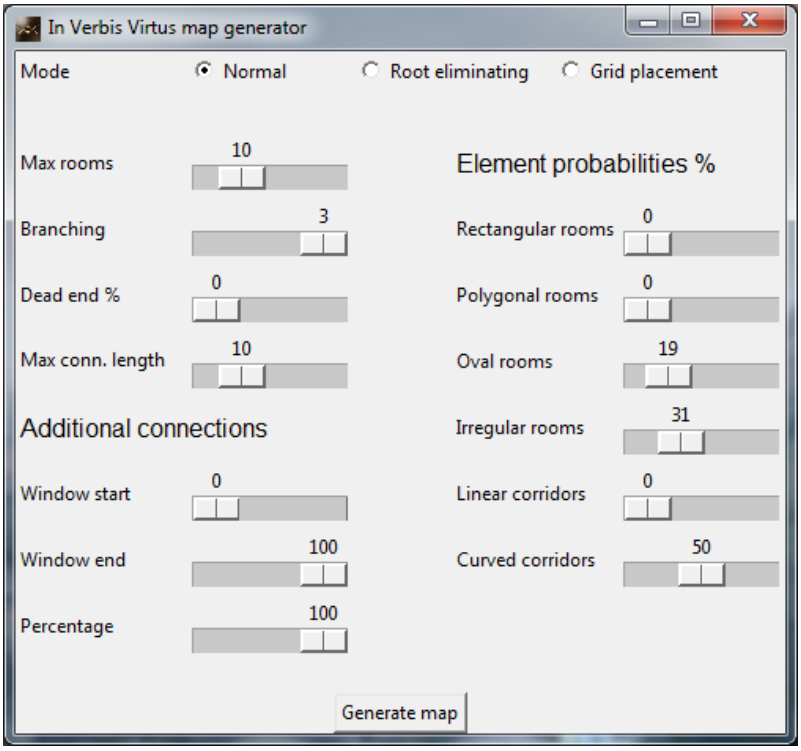


Figure 7.1: Graphical interface to control our map generation algorithm.

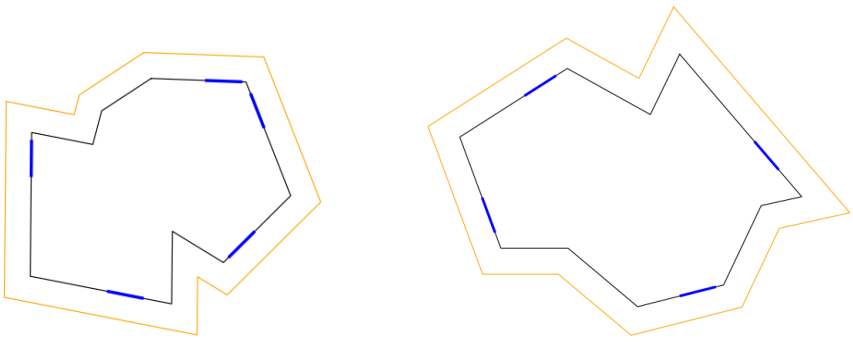


Figure 7.2: Two examples of irregular rooms. Floors are black, shells are yellow, connectors are blue. Irregular rooms are randomly generated joining regular polygons modified through translation, rotation and scale operations.

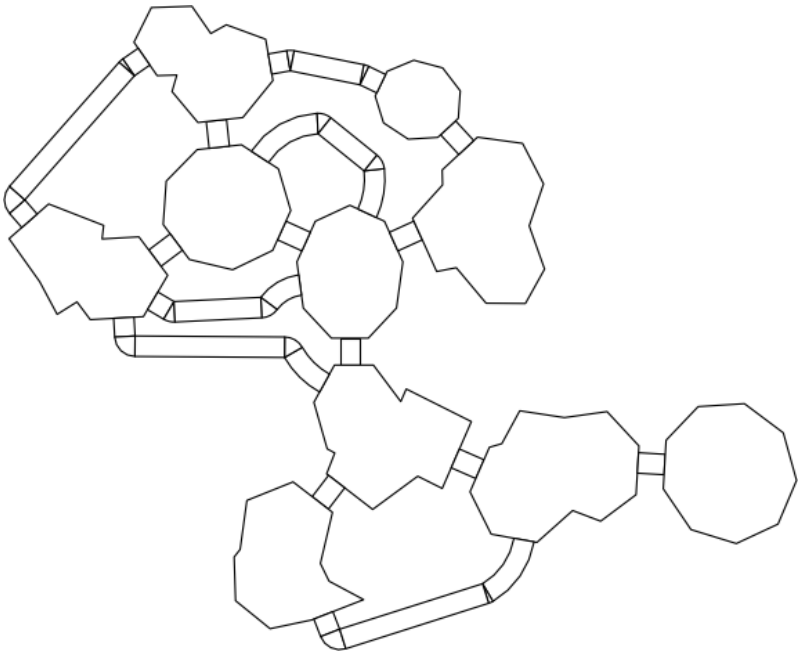


Figure 7.3: Floor plan generated by our algorithm, containing oval rooms and irregular rooms.

format used by the Unreal Development Kit. The exporter creates walls, floors and ceiling using rock assets, so that the map looks like a cave. It also adds light sources and decorations like columns, statues and plants. Figure 7.4 shows an ingame view of a map created completely automatically: the floor plan was generated by our algorithm and the assets were placed by our exporter.

7.1 Future works

We plan to extend this work in two ways.

First, we plan to evolve our map exporter into a full featured **asset placement toolkit**, that users can configure to control various aspects of map decoration (e.g. lighting and architectural style). Procedural techniques can be used to automatically combine given assets to obtain new styles of decoration.

In addition, we plan to extend our algorithm to use more complex indications on how to place architectural elements, in order to **guide the generation with formal grammars**. We intend to use sets of rules of L-systems[1].

[1] `http://en.wikipedia.org/wiki/L-system`

Figure 7.4: Ingame view of a map generated by our algorithm, with rocks and decorations automatically placed by our exporter.

Bibliography

[1] David Adams et al. Automatic generation of dungeons for computer games. *Bachelor thesis, University of Sheffield, UK.*, 2002.

[2] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schrr, and Gabriele Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34:1–54, 1996.

[3] Sean Barrett. Herringbone wang tiles. `http://nothings.org/gamedev/herringbone/`.

[4] Ulrik Brandes and Daniel Fleischer. Centrality measures based on current flow. In *Proceedings of the 22nd annual conference on Theoretical Aspects of Computer Science*, STACS'05, pages 533–544, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-24998-2, 978-3-540-24998-6. doi: 10.1007/978-3-540-31856-9_44. URL `http://dx.doi.org/10.1007/978-3-540-31856-9_44`.

[5] Michael F. Cohen, Jonathan Shade, Stefan Hiller, and Oliver Deussen. Wang tiles for image and texture generation. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 287–294, New York, NY, USA, 2003. ACM. ISBN 1-58113-709-5. doi: 10.1145/1201775.882265. URL `http://doi.acm.org/10.1145/1201775.882265`.

[6] Christian Güttler and Troels Degn Johansson. Spatial principles of level-design in multi-player first-person shooters. In *Proceedings of the 2nd workshop on Network and system support for games*, NetGames '03, pages 158–170, New York, NY, USA, 2003. ACM. ISBN 1-58113-734-6. doi: 10.1145/963900.963915. URL `http://doi.acm.org/10.1145/963900.963915`.

[7] Lawrence Johnson, Georgios N. Yannakakis, and Julian Togelius. Cellular automata for real-time generation of infinite cave levels. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10, pages 10:1–10:4, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0023-0. doi: 10.1145/1814256.1814266. URL http://doi.acm.org/10.1145/1814256.1814266.

[8] Brendan Lane and Przemyslaw Prusinkiewicz. Generating spatial distributions for multilevel models of plant communities. In *Proceedings of Graphics Interface*, pages 69–80, 2002.

[9] Kai Martin. Using bitmaps for automatic generation of large-scale terrain models. http://www.gamasutra.com/view/feature/3175/using_bitmaps_for_automatic_.php, 2000.

[10] Cameron McGuinness. Statistical analyses of representation choice in level generation. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pages 312–319, 2012. doi: 10.1109/CIG.2012.6374171.

[11] Gavin S P Miller. The definition and rendering of terrain maps. *SIGGRAPH Comput. Graph.*, 20(4):39–48, August 1986. ISSN 0097-8930. doi: 10.1145/15886.15890. URL http://doi.acm.org/10.1145/15886.15890.

[12] Yoav I. H. Parish and Pascal Müller. Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 301–308, New York, NY, USA, 2001. ACM. ISBN 1-58113-374-X. doi: 10.1145/383259.383292. URL http://doi.acm.org/10.1145/383259.383292.

[13] Ken Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3): 287–296, July 1985. ISSN 0097-8930. doi: 10.1145/325165.325247. URL http://doi.acm.org/10.1145/325165.325247.

[14] C. W. Reynolds. Steering behaviors for autonomous characters. 1999.

[15] Mick West. Random scattering: Creating realistic landscapes. http://www.gamasutra.com/view/feature/1648/random_scattering_creating_.php, 2008.