POLITECNICO DI MILANO
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

Procedural generation of realistic building layouts from graphs

Relatore: Prof. Francesco Amigoni
Correlatore: Dott. Matteo Luperto

Tesi di Laurea Magistrale di:
Stefano Carideo, matricola 898536

Anno Accademico 2018-2019

Sommario

Nel campo della robotica mobile autonoma, testare piattaforme e algoritmi in scenari reali può essere difficile per varie ragioni. Una soluzione comune è quella di utilizzare simulazioni, le quali permettono un completo controllo dello scenario sperimentato. Tuttavia, i risultati ottenuti sono solidi e generalizzabili soltanto se ottenuti simulando ambienti realistici. Dato che ottenere planimetrie di edifici reali non è semplice, e in alcune applicazioni è necessario un elevato numero di ambienti, un'alternativa è quella di generare edifici in modo procedurale. Tuttavia, nonostante siano state proposte tante tecniche di generazione di edifici, nessuna è focalizzata sul ricreare le caratteristiche di veri edifici e allo stesso tempo scalare su ambienti di medie e grandi dimensioni.

L'obiettivo di questa tesi è di generare proceduralmente edifici realistici, che possono essere usati come ambienti simulati per ottenere prestazioni simili a quelle ottenute in ambienti reali. Per farlo, imponiamo che gli edifici generati abbiano la topologia (cioè lo stesso grafo rappresentante i collegamenti tra le stanze) di veri edifici. Formuliamo il problema di generazione dell'edificio come un problema di ottimizzazione MIQP, che minimizza gli scostamenti delle dimensioni delle stanze dai loro valori originali nell'edificio in ingresso, e al contempo impone come vincoli rigidi il rispetto della topologia delle stanze.

Per valutare gli edifici che generiamo, confrontiamo le prestazioni ottenute in due esperimenti con robot simulati. Il primo si basa sulla lunghezza dei percorsi pianificati tra alcune stanze, mentre il secondo considera il tempo e la distanza percorsa richiesti dal robot per completare l'esplorazione dell'ambiente. Entrambi gli esperimenti confermano che è possibile ottentere, in media, prestazioni molto simili tra gli edifici originali e quelli generati. Tuttavia, abbiamo riscontrato alcuni problemi nelle prestazioni di esplorazione quando l'edificio originale contiene un numero elevato di stanze non rettangolari, visto che sono individualmente più difficili da esplorare e non possono essere ricreate con il nostro metodo, che gestisce solo stanze rettangolari.

Abstract

In the field of autonomous mobile robotics, testing platforms and algorithms in real settings can be difficult for various reasons. A common solution is to utilize simulations, which allow complete control over the entire setting of the experiments. However, the results are truthful and can be generalized only if realistic environments are used. Since retrieving real world building floor plans is not an easy task, and in some applications a large number of environments is needed, an alternative is to generate layouts in a procedural way. However, while multiple layout generation techniques have been proposed, none of them focuses on recreating the features of real building layouts while also being able to scale well to medium and large environments.

The goal of this thesis is to procedurally generate realistic building layouts, which can be used as simulation environments to obtain performance similar to those obtained in the original layouts. To do so, we impose that the generated layouts share the same topology of the rooms of real world building layouts. We formulate the layout generation as a MIQP problem optimization, which minimizes the the deviation of the sizes of the rooms from their values in the original layout, while enforcing as hard constraints the same topology of the rooms as the original layout. We then show how the proposed method is able to generate multiple large-scale realistic buildings.

To evaluate the building layouts we generate, we compare the performance obtained by two experiments with simulated robots. The first is based on the length of the paths planned between some rooms, while the second considers the time and the length of the path traveled by the robot to complete the exploration of the environment. Both experiments confirm that it is possible to obtain, on average, very similar performance across the original layouts and the generated ones. However, we found some issues with exploration performance when the original layout contains a large number of non-rectangular rooms, since they are individually more difficult to explore and we can't recreate them because our method handles only rectangular rooms.

Contents

Chapter 1

Introduction

In the field of autonomous mobile robotics, testing platforms and algorithms in real settings can be difficult for various reasons, like the cost and the time required for setting up the robot and the complexity in recreating the desired controlled conditions of the environment. To overcome these issues, a common solution is to utilize simulations, which allow complete control over the entire setting of the experiments. However, in order to obtain results similar to those obtainable in real world situations, it is necessary to use simulated environments that represent well the real ones in which the robot is supposed to work.

As discussed by in [1], this is not usually the case, since most of the simulated environments employed in robot experiments are based on non-realistic layouts, like grid-based maps or labyrinths, which can be easily created in a consistent number. A possible solution could be using, for experiments, environments inspired from real world existing buildings. However, retrieving real world building floor plans is not an easy task, and in some applications a large number of environments is needed. An alternative to get real floor plans is to generate them in a procedural way. This can be a valid approach only if the structure of original building layouts is maintained. While multiple layout generation techniques have been proposed in different contexts, like some based on machine learning approaches [2, 3] or evolutionary strategies [4, 5], none of them focuses on recreating the features of real building layouts while also being able to scale well to medium and large environments.

The goal of this thesis is to procedurally generate realistic building layouts, which can be used as simulation environments, where a robot can obtain performance similar to those obtainable in the original layouts, when considering simulated robot experiments. Since we aim to generate realistic layouts with structural features similar to those of the real ones, we impose that the generated layouts have the same topology graph of the original environments, which means that we use as an input to our method a graph representing the connection between rooms. The generation method we utilize is inspired and adapted from the one proposed in [6], which formulates the layout generation as a MIQP problem optimization. In particular, we consider as input for each generation a single original layout, represented in a graph format. We then define the adjacencies between rooms, described in the topology graph, as hard constraints of the problem, and minimize the deviation of the sizes of the rooms from their original values in the input layout. We found that our generation method can generate successfully also large layouts, while its main limitation is that the rooms are restricted to a rectangular shape.

To evaluate the realistic building layouts we generate, we compare the performance obtained by two experiments with simulated robots performed in them. The first is based on the path planning between pairs of rooms, with an analysis of the lengths of the paths planned between them. As second experiment, we perform repeated exploration runs on various generated layouts and on their original counterpart. We utilize as performance metrics the time and the length of the path traveled by the robot to complete the exploration of each environment. Both experiments confirm that it is possible to obtain, on average, very similar performance across original layouts and generated ones. This confirms our hypothesis that procedurally generated buildings are realistic and are similar with their real counterpart. However, we found some issues with exploration performance when the original layout contains a large number of non-rectangular rooms. The reason is that these rooms are individually more difficult to explore and we can't recreate them in generated layouts since our method considers all rooms as rectangles.

The thesis is structured as follows.

In Chapter 2 we present an overview of the state of the art of experiments in robotics and of layout generation, starting with an brief introduction of the main concepts of autonomous mobile robotics and then describing the platforms utilized to perform simulated experiments. After that, we review some of the most significant layout generation techniques, focusing on their approach and their limits.

In Chapter 3 we introduce the context of the problem addressed in this work, followed by its formal formulation. We also define the main requirements of our generation method.

In Chapter 4 we present an overview of the method that we propose to perform the procedural generation of realistic layouts, focusing at the beginning on the various phases of the technique and then on the specific generation method we employ, presenting it also in its original formulation as proposed in [6].

In Chapter 5 we describe the implementation of our system in detail, focusing on each component and data structure utilized in the entire process.

In Chapter 6 we provide some considerations on our system, focusing on the main issues found and presenting some examples of generated layouts. We then introduce the setup utilized in the experiments, which are presented afterwards along with their results and final considerations.

In Chapter 7 we present a summary of our work, the conclusions, and some possible future developments.

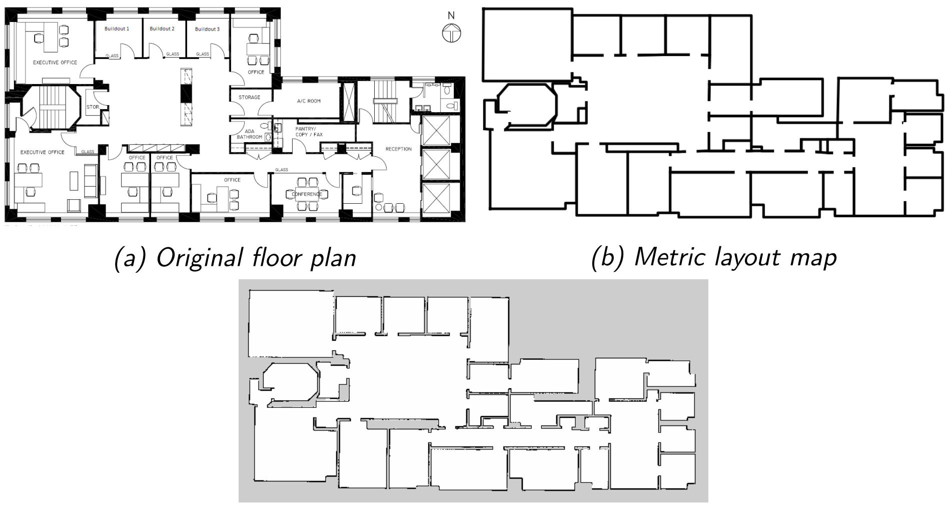
Chapter 2

State of the art

In this chapter, we present an overview of the state of the art of procedural layout generation, which is the task of creating the map of a building starting from structural and topological parameters, and in particular of its application to autonomous mobile robotics. The use of procedurally generated building layouts could provide numerous realistic simulated environments for testing and developing robots. Simulation tools in robotics are important because allow to run algorithms in a controlled setting without the need of a physical platform and environment. At first, we introduce autonomous mobile robots with some of their most relevant features. Next, we discuss experiments in robotics, identifying the important tools employed in this field, and the issues that restrict their effectiveness. Finally, we review some layout generation techniques, focusing on the approach they are based upon and their limits.

2.1 Autonomous mobile robotics

Autonomous mobile robotics is the field that studies robots that are supposed to move and act in a specific environment without the continuous intervention of a human operator. When a robot needs to navigate in an environment it requires a *map*, which is a spatial representation that includes obstacles and walls to define where the robot can actually travel and where it can't. There are different types of maps. Three examples are provided in Figure 2.1, in which the same building is represented with metric maps. In Figure 2.1a it is reported the original floor plan of the building. An example of metric map used as robot environment is pictured in Figure 2.1b. It is a perfect information layout map, that describes the structure of the building by defining the walls of the rooms. The example in Figure 2.1c is an *oc-*



(a) Original floor plan

(b) Metric layout map

(c) Occupancy grid map

Figure 2.1: Example of office metric maps

cupancy grid map. As discussed in [7], these kind of maps are fine-grained grids defined over the continuous space of locations. Each cell is associated with an occupancy value, which describes the probability of that location to be an obstacle. They are the common map representation used by robots to navigate the environment. Usually the occupancy grid maps are 2D, but the same concept can also be expanded to 3D worlds. However, one of their main problems is the number of cells, since an high resolution of the grid, while improving precision, also leads to significant computational expenses.

The map can be provided beforehand the *run*, or created directly while traveling. In the first case the robot only needs a localization system in addition to the navigation one, while in the second case a much more complicated system is required since this problem is *Simultaneous Localization and Mapping.* As discussed by Durrant-Whyte and Bailey in [8], this task is based on a continuous learning of the map (both creating and updating it) paired with a constant localization in it. The problem of navigating around an unknown environment in order to collect enough information to create a complete map is called *exploration*, and it is another task that is widely studied in this field. Some examples of works that study this problem are [9], [10], and [11].

2.2 Experiments in robotics

Both SLAM and exploration are some of the most complex and important problems in autonomous mobile robotics. Their study and development demand thorough testing. However, running a real robot in a real environment requires both space and time to setup the experiments. A common solution is the use of *simulations*, in which the algorithms can be tested on modeled robots in a virtual environment, with a complete control of the entire setting of the experiment.

A much more limited alternative to real and simulated runs (*online experiments*) is the use of *offline experiments*, that are based on the prerecorded data acquired by a robot in a specific environment. This approach excludes any kind of interaction between the robot and the world, including also the navigation itself, so it can't be used to test for example exploration algorithms, but only tasks that just require sensors readings like SLAM.

As discussed by Amigoni et al. in [1], offline experiments are used by 83.6% of the reviewed papers for SLAM evaluation. This can be explained by the fact that offline experiments require the acquisition of the data only once, reducing the overhead for the researchers, but limiting the choice of the entire setting and platforms (map, sensors, robot, exploration and navigation method, ...) that are made by the authors of the specific dataset employed. Some examples of such datasets are Radish [12] and RAWSEED [13, 14], which are both made by university researchers and are based on campus buildings (mainly indoor environments). Considering the online experiments, most of the publications employed simulations instead of real robot runs. However, it is important to note that only 21.5% of the simulated experiments used a real building as map, while in the other cases the layout was based on non-realistic maps, usually grid-based and handcrafted to test specific isolated features, like closed loops or labyrinths.

In fact, even if simulations have the advantage of allowing experimentation with almost any kind of environment, they can only be truthfully useful and hopefully generalizable if a realistic layout is used, as otherwise the features present in a certain custom map may be totally different from those of a real building.

There are some datasets that provide a very detailed 3D representation of a small amount of buildings, like AI2-THOR [15] and 2D-3D-S [16]. These systems are more indicated for applications that require high fidelity in both visualization and interaction between the world and the objects of the simulated world. However, the maps that they provide are very limited in number.

When testing SLAM or exploration algorithms the focus is typically on the map itself. In these cases, there are more suited frameworks like Stage [17], a lightweight 2D simulator. A bitmap image provided by the user is used to create the environment as input map, while both sensors and actuators are already implemented with various models. The user can also create plugins to expand and customize the capabilities of the engine and of the models provided. A more advanced simulator is Gazebo [18], which includes both high-quality graphics and a complex physic simulation based on Open Dynamics Engine[1]. There is also an improved sensors and actuators modeling, that allows accurate and extensive interactions between the robot and the environment. Both these systems can also handle multi-robot settings and are usually utilized in conjunction with ROS.

ROS (Robotic Operating System) [19] is a framework employed in robotics. It can be used seamlessly with simulated robots or with real ones, since it provides both hardware and software abstractions, organizing the single elements of the system in *nodes*, which operate individually and communicate between them with a message-passing protocol managed directly by ROS. There is a package manager with a large amount of already implemented algorithms and real hardware controllers. The distributed nature of ROS also allows the execution of a complete system on different devices. An example is a real robot with limited computational capabilities that relies on an external computer to perform SLAM and exploration tasks.

Stage and Gazebo can be used to test a large variety of maps and algorithms. To do so, a map layout is needed to be recreated in the simulated world. This can be done both by handcrafting a bitmap file or by using a dataset with specific buildings already encoded in a certain format (not necessarily in image files ready to be utilized directly in a simulator). These collections of floor plans are usually composed of layouts of the university campuses where the researchers who created the dataset work. Some examples are the MIT [20], the KTH [21] and the HouseExpo [22] datasets. In particular, the MIT and KTH datasets are based on the buildings of their respective university campuses and are encoded in XML files with the technique proposed in [23]. This format is based on the concept of graph with a floor as root of the XML, which contains the various spaces (rooms, the nodes of the graph) with a label to represent their type (e.g., corridor or classroom) and their geometrical structure, and the portals that consist of connections between two spaces (like doors, the arcs of the graph). Their main limitation is that even if they contain a relative large amount of floor plans, they are

[1]http://www.ode.org

very homogeneous as they include very similar layouts multiple times, and in particular they are all campus buildings, so not representative of the various classes of maps.

According to Amigoni et al. in [24], it is possible to use simulations in the evaluation of SLAM algorithms and obtain results comparable to real robot runs. In particular they propose a system to automate experimental runs in a simulated environment, overcoming in this way also the issue of variability of results (that is present even with real robots). This result is very important as some of the metrics used to evaluate SLAM algorithms require additional data to supplement those provided by the onboard sensors. In fact, the metric [25] that they use needs the *ground truth* information of the robot *pose*, which is the real position of the robot in the environment. The robot computes its estimated pose with *odometry*, which is the integration of the distance covered by wheels over time. However, the sensor readings from the wheels are not very precise, so the cumulative error can increase a lot over time. As discussed in [7], it is possible to improve the localization provided by the odometry in a probabilistic way, using data received from other sources, like laser sensors. Still, this corrected pose is still an estimation, so it can't be used as ground truth. In simulated settings it is easy to get the ground truth data of the robot position, while in a real setting with a physical robot it requires the use of complex and costly equipment, like the OptiTrack[2] system employed in [24] and the hardware utilized in [26].

However, an intrinsic limitation of their automated evaluation approach is that the simulated maps, as already reported in [1], need to be realistic in order to be actually significant in relation to real world environments. In particular, it is underlined the importance of the topological structure of the map, which is the adjacency relation between the various rooms of the layout. Another limitation is the quantity of realistic maps available, since we need a large amount of them to obtain consistent results. However, this is not always possible to do as it is difficult to retrieve a large dataset composed of maps that are both realistic and heterogeneous.

To solve this problem, we decided to increase the number of available realistic maps by generating them in a procedural way.

2.3 Layout generation

In this section we introduce the task of layout generation, and then present a general, not exhaustive, collection of methods that address this task divided

[2]https://optitrack.com/

by approach, focusing in particular on how they would behave when applied to generate layouts with specified features. The *layout* is a representation of the floor plan of a building. While the map is focused more on the structure of the walls and of other obstacles, like furniture, the layout includes information about the rooms, like their geometry, their function, and their connections.

The *procedural layout generation* is a technique that, starting from an initial set of input parameters, produces as output a layout that follows as much as possible the properties defined by the input data. The nature of the input parameters depends on the method itself and on the representation format used for the layout. Typically, the parameters are topological and structural. The *topological parameters* are the adjacency relations of the rooms, representing the connections between them, which are usually the doors. These parameters are commonly expressed in a graph format, in which the nodes represent rooms and the edges the connections between them. The *structural parameters* are the geometrical and functional properties of the rooms, like their sizes, their aspect ratio and their type.

As we want to generate realistic layouts, we need a generation technique that allows enough control on the result, since we want to impose the features that would make the resulting layout realistic.

As discussed in [27], the layout generation topic is not only considered in the field of Computer Science, but also in Architecture, so there is a large variety of methodologies. Some of them require a lot of interaction by the user as their main goal is more to assist an architect than to automatize the entire creation process.

Even in pure generative techniques there are different goals. In fact, some methods are focused towards generating layouts with a large amount of structural details, like rooms and facades forms, while others are aimed to keep a topological resemblance with the input data provided.

In addition to that, there is also a difference in the quantity of input data required by the various generation processes. In some cases a lot of information and parameters are provided to create a layout that adheres to most if not all of the initial constraints, which can be for example the requested adjacency of the rooms. In other cases just few basic parameters are needed by the system. This increases its degree of freedom but also allows for less control on the final solution by the user. In fact, in some works the explicit goal is to try to recreate a realistic building, by strictly following the parameters provided by an expert user or by another layout, while in other approaches the objective is to create quickly a lot of plausible buildings, for example to populate a large 3D scene, disregarding many details of the generated floor plans.

As last remark, most of the works are based on small buildings, usually residential houses, without focusing on the scalability of the method to medium and large layouts. It is also clear that some approaches can handle only problems with a very restrict number of rooms, at times even stating the limit themselves or showing the largest building produced.

In the following subsections we analyze some of the most interesting and studied approaches for the layout generation problem, dividing the works by methodologies.

2.3.1 Evolutionary computation

As presented in [28], *evolutionary computation* is a broad category of algorithms that are based on the idea of *population* composed of *individuals*, where each one contains a possible solution to the problem, and at each *generation* (iteration of the algorithm) the population is updated. This happens with two operators, *mutations* and *selection*. The first acts on each individual, changing in a random way their embedded solutions by a small margin, while the second operates on the population in its entirety, by removing the worst individuals after an evaluation with s *fitness function*, that is a cost or utility function employed to rate at each iteration the various solutions and to check the termination condition (if it is not just set as a certain number of generations to run).

These methods perform well especially with a very large *search space*, that is the set of possible solutions that can be formulated and from which the best one has to be found. There are multiple variants of evolutionary techniques, like genetic algorithms and evolutionary strategies. Some approaches that are based on evolutionary computation are reviewed in [29] and [30]. We present here some interesting works, divided by method.

Genetic algorithms

Genetic algorithms (GA) are a very popular optimization and search methodology that models a possible solution of the problem, called in this case *genotype*, with a particular low-level encoding format that is usually binary, so with a sequence of bits. As reviewed in [31], the genotype of each individual is modified by operators to improve the fitness function. In the general case genetic algorithms have three operators. The first, selection, chooses some individuals (typically the most fit) to be bred, while the second, *crossover*, creates new individuals by combining the two genotypes of the selected parents with a certain strategy. An example of crossover from [31] is pictured in

```
Parent one:        1 1 1 0 1 0 0 1 1 0
Parent two:        0 0 1 0 0 1 1 1 0 0
crossover point:          ↑
Child one:         1 1 1 0 0 1 1 1 0 0
Child two:         0 0 1 0 1 0 0 1 1 0
```

Figure 2.2: Example of crossover

Figure 2.2, where the genotype of the parents is transferred to the children by copying directly the genes until a certain point, after which the target of the copy is exchanged between the offspring. There are also more complex strategies to combine two original genotypes. The third operator is mutation. It operates changing the genotype locally in a certain random fashion by a small amount, acting directly on the genotype string. After the application of the operators, there will be new individuals that replace the original ones in the population. However, it is possible to adopt an elitism method that carries to the new generation the best individuals in any case, guaranteeing that the value of the fitness function won't decrease over time since at each generation the best solutions can't be lost.

An example that applies this technique is [32], that uses GA in different contexts. In fact, the goal is to generate a multi-level flat by producing initially the single floors and then, with the interaction of a user, combine them to form the entire building. This work states that the maximum number of rooms of a single floor that can be generated is 11.

A more advanced use of GA is in [33], where the layout representation is based on a grid and the rooms are considered spaces that occupy a certain amount of adjacent cells. A complex fitness function tries to enforce the input parameters, but it can't guarantee to always satisfy them all. In the corresponding thesis [34] it is presented the example in Figure 2.3, in which on the left it is pictured the generated office as retrieved from the genetic algorithm process, while on the right it is shown the same floor plan with some post correction steps. This work states that medium/large layouts are impractical to solve because of the exponential growth of the search space (the office in Figure 2.3 is considered one of the largest layouts that is possible to generate with this method) and even in smaller cases it is not guaranteed that all objectives are actually satisfied.

Other evolutionary algorithms

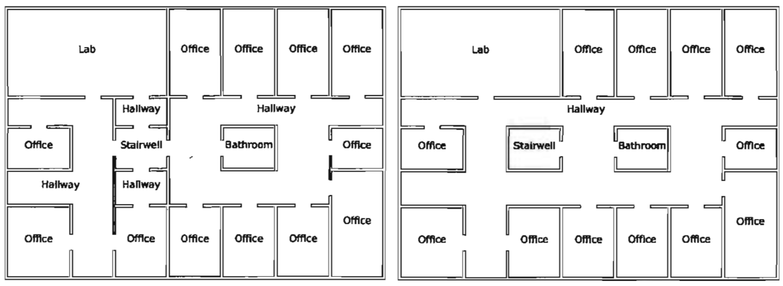
Evolutionary algorithms are the superclass of genetic algorithms, in which there are various techniques that are based on the same concept of population, but that handle the genotype representation and the associated operators in a different way. The reason why genetic algorithms are the most famous and common ones is their low-level encoding of the solution, which increases the generality of the method since don't require a complete understanding of the problem to be solved.

There are various works that utilize a particular kind of evolutionary algorithm, like in [35], where the goal is to create an interactive application that generates some unrefined layouts using generic parameters and an input target building. In particular, the system compares local neighborhoods similarity between the given floor plan and the one that is being generated, and utilizes this value as one of the multiple objectives of the fitness function. The operators employed are local transformations of the rooms, that allow movements, rotations, and swaps. The results are unrefined because they are meant to be starting drafts for architects.

Another system is [36]. The peculiarity of this work is that the fitness function also includes a similarity term based on the user ratings of previously generated layouts. One of the main issues of this method is the difficulty in the creation of a large dataset of rated environments, which must be provided in a considerable number while keeping the scores assigned in a consistent way during the entire process. Moreover, this strong dependence on user ratings can misguide the generation towards solutions that are not necessarily good but just contain particular features present in other highly rated layouts.

Evolutionary strategies

Evolutionary strategies (ES) differs from the previous evolutionary techniques because they are not based on the idea of genotype and breeding (with crossover), but more on the iterative application of selection and mutations. There is still a population of individuals, each one of them with a



(a) Original *(b) Corrected*

Figure 2.3: Example of office generated with a genetic algorithm

candidate solution evaluated with the fitness function, but in this case the individuals are not combined between them. Instead, they are only mutated with very problem-specific operators. The selection guarantees that particularly bad solutions are discarded (and replaced with freshly generated ones). The mutations can still perform modifications that decrease the value of a solution, so that it is possible to evade from *local maxima*, which are sub-optimal solutions that can't no longer be improved by moving locally.

There are two very promising layout generation systems that follow this approach. The first, called *Evolutionary Program for the Space Allocation Problem* (EPSAP), is proposed by Rodrigues et al. in [4,37], where an evolutionary strategy is coupled with a *stochastic hill climbing* (SHC) technique, which is a local optimization algorithm that tries to improve a given function with small movements, but not necessarily in the direction of the best local improvement. The evolutionary strategy guarantees the constant presence of multiple feasible solutions, one for each individual, and applies operations to the layouts on a global scale, aligning walls and assuring the feasibility of the layout. These operators are applied at each ES generation, but only after the SHC optimization, which acts on a local scale to manipulate geometrically the rooms. The system accepts as input both geometrical and topological parameters. A further improvement of this method has been proposed with enhanced capabilities, among which the generation of multi-level buildings, in [38], while a practical application of it, with also the purpose of optimizing thermal performance of the generated layouts, is [39]. An example of the enhanced method in [38] is reported in Figure 2.4, where a medium size building is successfully generated.

The second method is proposed by Guo et al. in [5]. Even in this case the procedure is divided in two steps, but here they are purely sequential. The first step is a *multi-agent system* finalized to find a feasible layout for the rooms according to the topology constraints. This system employs bubble agents that can be spheres or capsules (the system considers multi-level buildings) and, by the application of interaction rules (attraction, repulsion, ...) among them, provides in the end a certain topological layout mapped in space. The second part is the evolutionary strategy, that acts on the conversion of the previous layout to a grid-based representation and optimizes the entire building manipulating the geometrical features of the rooms with two specific operators. The first is the local movement of a boundary of a room, while the second is optional and is based on the swap of two rooms in the layout. The fitness function of the ES is calculated with a weighted sum of four criteria (topology, size, aspect ratio, and building shape). The main issues of this approach are scalability, the grid-based model, and that

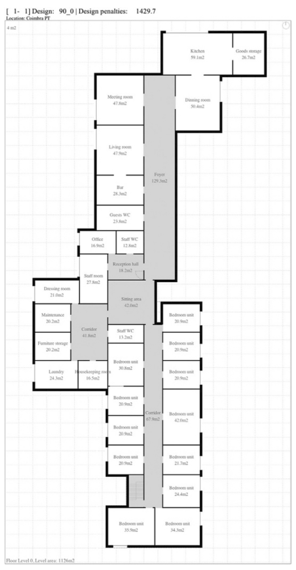


Figure 2.4: Example of building generated with the EPSAP method

the evolutionary strategy algorithm used is too simple and can get stuck in local maxima.

2.3.2 Treemaps

Treemapping is a method to subdivide a rectangle in a hierarchical way by nesting tiles (still rectangles) according to a certain derivation tree and it is typically used to visualize data in a clear and expressive way. *Squarified treemaps*, proposed in [40], try to solve an issue of the original concept, reported in Figure 2.5, which is the presence of tiles with a very large aspect ratio, by using a more complex recursive algorithm. These structures are used in [41] to represent the layout of a house, which is generated by subdividing the initial boundary in areas according to their use (private, social, and service area) and then subdividing again the regions to create the rooms. In this method there are input parameters to assure the sizes and the adjacency of rooms, which guide the subdivision in tiles and the allocation of the rooms to each one of them. Once the treemap has been generated, the system proceeds with the creation of doors and corridors, designed considering internal walls and the shortest path on them that connects the rooms as defined in the input data, even if the resulting corridor has a strange form. This method works only for (very) small layouts in which there is a clear subdivision of regions (that can be expanded over the three areas considered).

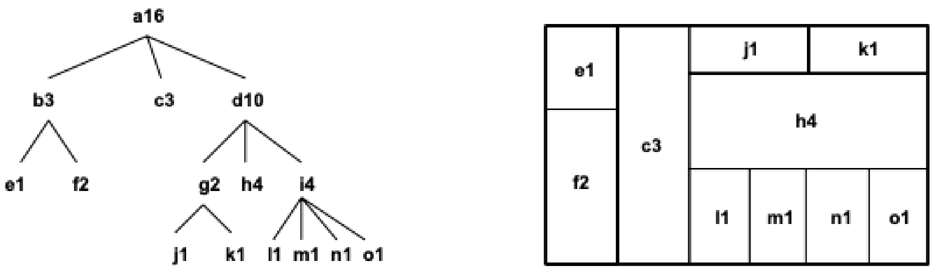


Figure 2.5: Example of derivation of a treemap from a tree

In fact, this framework is conceived to handle real-time house generations.

A much more advanced approach based on treemaps is [42], in which *rectified treemaps*, that are similar to squarified treemaps and guarantee customized aspect ratios and also area proportions among tiles, are used as basis for a genetic algorithm optimization. In this case the scalability is better, but there are still limits in the constraints definition and in the polishing required to get the final output. One example of layout generated with this method, taken from [42], is in Figure 2.6.

2.3.3 Grammars

Grammars in theoretical computer science are sets of defined rules used to form valid strings according to a certain defined syntax. They don't encode the meaning (semantics) but only the structure.

One of the first works to utilize grammars to generate layouts is [43]. The proposed approach is to capture a certain architectural style by creating a

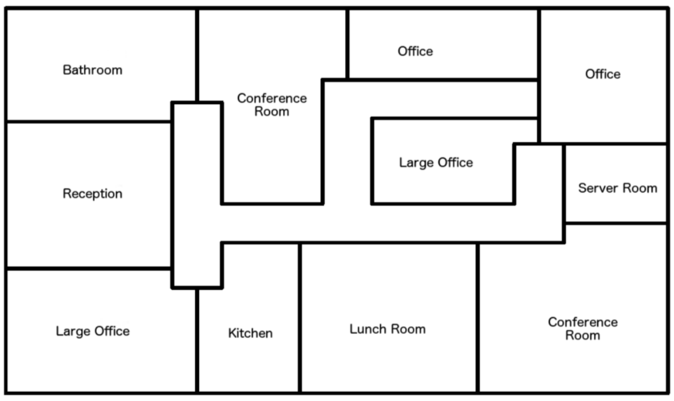


Figure 2.6: Example of office generated with the Evolutionary Treemap method

language based on basic shapes, which can be combined using grammar rules to create an entire building. The grammar rules are defined manually by an expert. In particular, it is presented as example the generation of a Palladian villa.

In general there are two main ways grammars can be employed in layout generation. The first is to define a grammar to describe the structure of a building and using it directly to generate some solutions (which are syntactically valid strings). A method proposed in [44] is based on an initial derivation of a certain number of layouts, from which the user can choose one to consider as final result or to further modify with the same set of rules. A similar system developed by Leblanc et al. in [45] is based on a programmatic way to apply derivations, where the input data is a program that defines the order and the conditions of the operations to be applied to the layout.

The second approach is to use as in the previous case the grammar to define the structure of the layout, but then to utilize a particular algorithm (usually a genetic one) to apply rules like in [46] or to represent the building itself as in [47], where the walls are directly encoded as genotype. The fitness functions are utilized to evaluate the quality of the structure of the layouts, like the respect of the adjacency and the sizes of the rooms. In both cases only small houses can be generated.

Some of the main issues of these methodologies are the difficult formulation of high level constraints in a clear and direct way, and the scalability with buildings of medium/large sizes (at least in an autonomous way).

2.3.4 Machine learning

Nowadays machine learning techniques are studied and employed in a large number of fields. *Machine learning* utilizes statistics to create models that can learn some specific features from provided data, and later perform predictions on similar data. There are examples of applications also to the layout generation problem.

Some of these approaches, like [2] and [3], are based on Bayesian networks to learn layouts, but with different scopes. *Bayesian networks* [48] are models that employ Bayesian inference to predict a certain event given the probability that another one happened. In the first case the predictive model is trained with a provided dataset in order to learn the topology and the basic geometrical features of the rooms and produce a so called *architectural program*, that is the layout specifications. The architectural program is later optimized with the *Metropolis-Hastings algorithm* [49], a statistical sampling method to estimate a certain distribution, using local operations that manipulates the layout by sliding walls and swapping rooms. A complex cost function tries to guide the optimization towards the desired architectural program. Differently, in [3] the Bayesian network is used to predict the rooms sizes and positions which, after a manual placement in the layout by the user, are further optimized with a simulated annealing approach in two steps. *Simulated annealing* [50] is a probabilistic technique used to approximate in a fast way the global optimum of a function. The first step involves local operations to improve the position and orientation of the rooms, while the second step refines the sizes. In both cases a relatively large dataset is required to train the model.

A particular approach has been proposed by Feng et al. in [51], in which the focus is optimizing large layouts according to some *crowd flow* metrics, that are based on the potential movements of people in the environment. A *regressor* [52], which is a predictor that learns a given function, is trained with the proper complete agent-based simulations to evaluate in a fast way an approximation of their costs instead of running dedicated simulations each time. This improvement is used during the optimization performed with a Metropolis-Hastings algorithm, that applies local operations and evaluates at each generation the cost of the layout using the regressor. This evaluation would be impracticable if the entire agent-based simulations had to be repeated at each iteration.

2.3.5 Operations research

A different and interesting approach to the layout generation problem has been proposed by Wu et al. in [6]. This method is based on a hierarchical framework that employs a Mixed Integer Quadratic Programming (MIQP) formulation, which is solved with an external solver (they used Gurobi[3]). The *Mixed Integer Quadratic Programming* is an operations research problem that contains both integer variables (in this case mainly binaries) and some particular quadratic formulations of the objective function. This system accepts as hard constraints the building boundary, the adjacency between rooms, and some geometrical values like min/max aspect ratio and sizes, while, as soft constraints, the target sizes of the rooms that the optimization tries to reach. This method handles only rectangular-shaped rooms in its basic formulation, but scales reasonably well with large layouts even if, for the generation of large buildings, a hierarchical framework is used. In this latter case the optimization is performed at different levels, at first

[3]http://www.gurobi.com

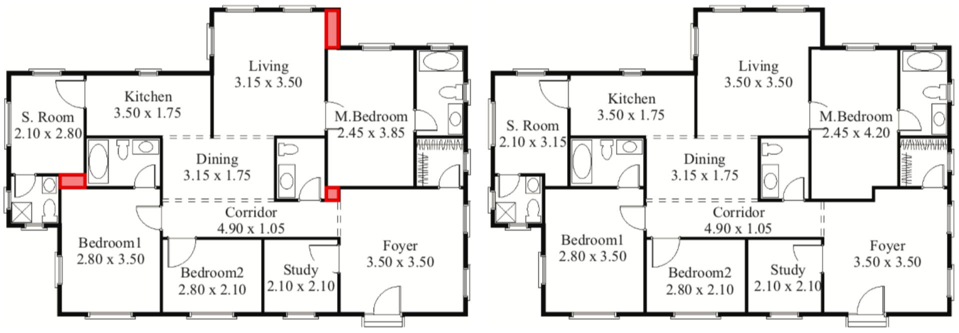


Figure 2.7: Example of building generated and later improved with the MIQP method

creating some high-level regions and then formulating other sub-problems in each space and so on until the desired degree of detail is reached. One downside of this hierarchical framework is that the problems have to be re-formulated each time in a dedicated way, even though this allows a complete customization of each level of generation. The MIQP problem is formulated by representing each room with its position (coordinates of the lower-left angle), size, and orientation (vertical or horizontal), while the constraints guarantee input parameters application and other structural features, like non-overlapping. A hierarchical approach is used also in refining the layout after a generation, formulating in an autonomous way the sub-problems required to close some holes in the building by updating the nearest rooms, which are decomposed in two rectangles in order to have a more complex structure that can be exploited. In Figure 2.7 it is reported an example taken from [6] of a building generated with this approach and further improved with the rooms decomposition and updating.

2.3.6 Other approaches

The layout generation problem has been studied also with other dedicated approaches. In [53] two algorithms are proposed to generate buildings in real-time. The first method is very basic and produces only simple houses, while the second generates at first a graph, and then places it in the space. A final procedure to create internal walls is not implemented but only described.

In [54] and [55] some systems are presented that are intended more as interactive guide for architects rather than generative frameworks. In the first case a knowledge base is created by discussing with experts, which will be later exploited by the system to automatically check if the inserted layout is feasible. In the second paper the system creates the starting layout and offers hints and information each time the user interact with the layout. To do so, the system considers various constraints and costs (the work is focused on the minimization of cuts of pre-cast concrete slabs required to realize the building).

Another different approach based almost entirely on graphs is [56], in which a planar dual graph is generated from the starting floor plan. Once the layout is represented in this way, it can be subjected to automatic or user-based transformations, like addition or deletion of single rooms. At the end of the entire process, the graph can be reconverted in a proper floor plan for final refining.

2.4 Summary

In this chapter we presented various approaches and techniques related to robotic experiments and to layout generation methods. In the first part we introduced autonomous mobile robots. In the second we highlighted the importance of simulations for robotic development and their limits, which are specifically related to the use of realistic buildings. At last, we discussed various layout generation techniques, underlining that just few of them can actually handle specific input parameters to create mid/large scale environments while allowing the user to control the presence of particular features in the generated buildings.

Chapter 3

Problem formulation

In this chapter, we present a formal definition of the problem addressed in this thesis and the approach proposed to solve it. We start by presenting the research context of this work, focusing in particular on the definition we use of realistic simulation of indoor buildings and why such realistic simulations are needed in robotics. We then set the goal of the thesis. Afterwards, we introduce the main characteristics required by a system that would be able to tackle the formulated problem.

3.1 Research motivation

In the field of autonomous mobile robotics, as discussed in Chapter 2, simulations are a fundamental tool for researchers and developers to test platforms and algorithms in a cheap and controlled way.

As reported in [1], simulations commonly employ non-realistic environments like grid-based closed loops and labyrinths. There are also competitions based on simulated rescue operations, in which the goal is to explore a certain unknown environment in order to find and rescue survivors. However, also in these cases the maps are handcrafted and not always with realistic buildings as target. One of these competitions is the RoboCup Rescue Simulation Virtual Robot League[1]. In Figure 3.1 are shown two examples of maps used in the 2019 edition. These layouts are clearly not realistic.

Simulations can be utilized to train and test robots. However, if the environments employed are too simple and don't contain real building features, then the robots can't reproduce the same performances in real world situations. On the contrary, using realistic layouts in simulations leads to comparable results as evidenced in [24].

[1]https://rescuesim.robocup.org/

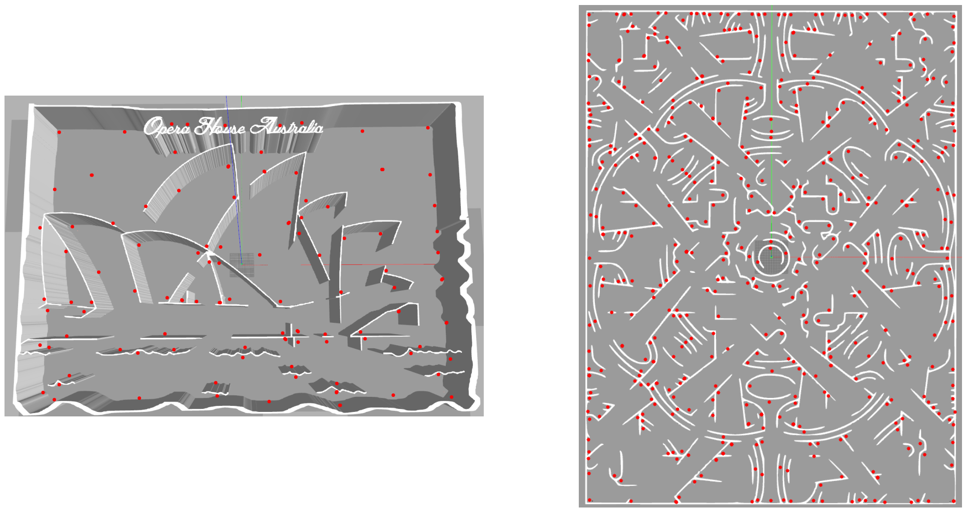


Figure 3.1: Example of maps used in the RC2019RVRL competition

The easiest possibility for using realistic layouts in simulations is to use floor plans of actually existing real world buildings. Unfortunately, even if converting real floor plans to layouts usable in the various robot simulators is not a particularly difficult task, retrieving a large amount of real building floor plans is not always easily possible in practice for their availability.

This is particularly true when hundreds, or even thousands of different buildings are required for testing a given methodology, as in deep reinforcement learning methods. One of the proposed solutions is to directly use floor plans of entire university campuses, which are available to researchers, as is done with the MIT campus dataset [20] and the KTH one [21], that overall contain 197 buildings and 940 floors. However, as discussed in [1], one of their main problems is the homogeneity of the included layouts, which in these particular cases is caused by the repetition of very similar building floor plans, since these datasets are both based on university campuses.

As previously seen in the state of the art of layout generation discussed in Chapter 2, there are different interpretations of what a *realistic* layout is. For example, the concept of realistic can be intended as representing in a good way the external shape of a building or following the rules specified by an expert. Another definition of realistic can be derived from [1], in which the structural realism of a map is discussed. We decided to adopt this concept, that is based on both structural and topological features that must be consistent between real buildings and the ones used in the simulations.

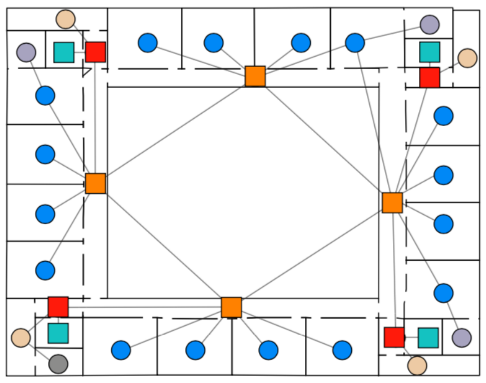


Figure 3.2: Layout with corresponding graph

3.2 Problem statement

The goal of this thesis is to generate in a procedural way realistic building layouts, so that it is possible to increase the number of available environments for simulated robotic experiments, while keeping the same performance obtained in real layouts. To do so, the generated layouts should be similar to real ones. Because of that, we consider as realism the correspondence of structural and topological properties between a real building and the one we generate.

As input data, we assume to already have a graph representation of the original floor plan. In particular, we utilize as input a graph representing the geometrical structure and the adjacency relations of the rooms. It is created from an original building floor plan and it still contains the position of every element (e.g., walls and doors) of each room. There are various techniques to create this kind of graph, like [23]. We utilize the same graphs used in [57], from which an example is reported in Figure 3.2. In this format there are also encoded the types of the rooms, represented in the figure as different node symbols. At the end of the process, we provide the generated layout in the same format of the input, which contains all the information about the structural elements. In this way, it is compatible with any method that already accepts the original graph format, while allowing it to be drawn directly in order to be used in robot simulations.

3.3 Method properties

In order to produce realistic buildings as output, our system needs to possess the following properties.

In contrast to other layout generation techniques, which either require a large dataset of floor plans to learn or architectural rules to be directly inserted by experts, our method should be usable by robotic researchers with just a small amount of real building plans available. These floor plans will constitute the foundation of the generative process. In this way, the availability of a few dozen floor plans could be used to generate hundreds or thousands of them. To do so, the objective is to be able to generate multiple different environments for simulations starting from just a single original building plan.

The generation method doesn't necessarily need to generate new environments in real-time, as is often the case within the field of procedural content generation. Our goal is to generate mid/large layouts that reflect in an accurate way features of real buildings, rather than to populate large scenarios with many simple ones. Still, since the main idea is to increase the number of available floor plans, the system should take a reasonable time to produce an output, possibly in the order of minutes.

The fundamental feature that we want to guarantee is the topological structure, that is the adjacency between rooms. The structural features, considered as the geometrical properties of the rooms, are also important but not as much as the topology. In fact, it is not strictly required to generate rooms with complex structure and even rectangles are fine. For our intended application, what is actually important is that the sizes of the original rooms are maintained to a certain degree in the generated layout.

Another important property of the system is the compatibility of both input and output formats with other common datasets, like the MIT and KTH ones, which follow the format presented in [23]. In this way, these datasets can be used as input to generate thousands of buildings.

In general, there is no need to generate multi-level buildings but only single-level floor plans.

3.4 Summary

In this chapter we introduced the problem of generating realistic layouts for simulation purposes. Next, we set the goal of the thesis and specified the input and output of our system. Finally, we defined the properties that must be respected by a generation system to be used for our purposes.

Chapter 4

Proposed solution

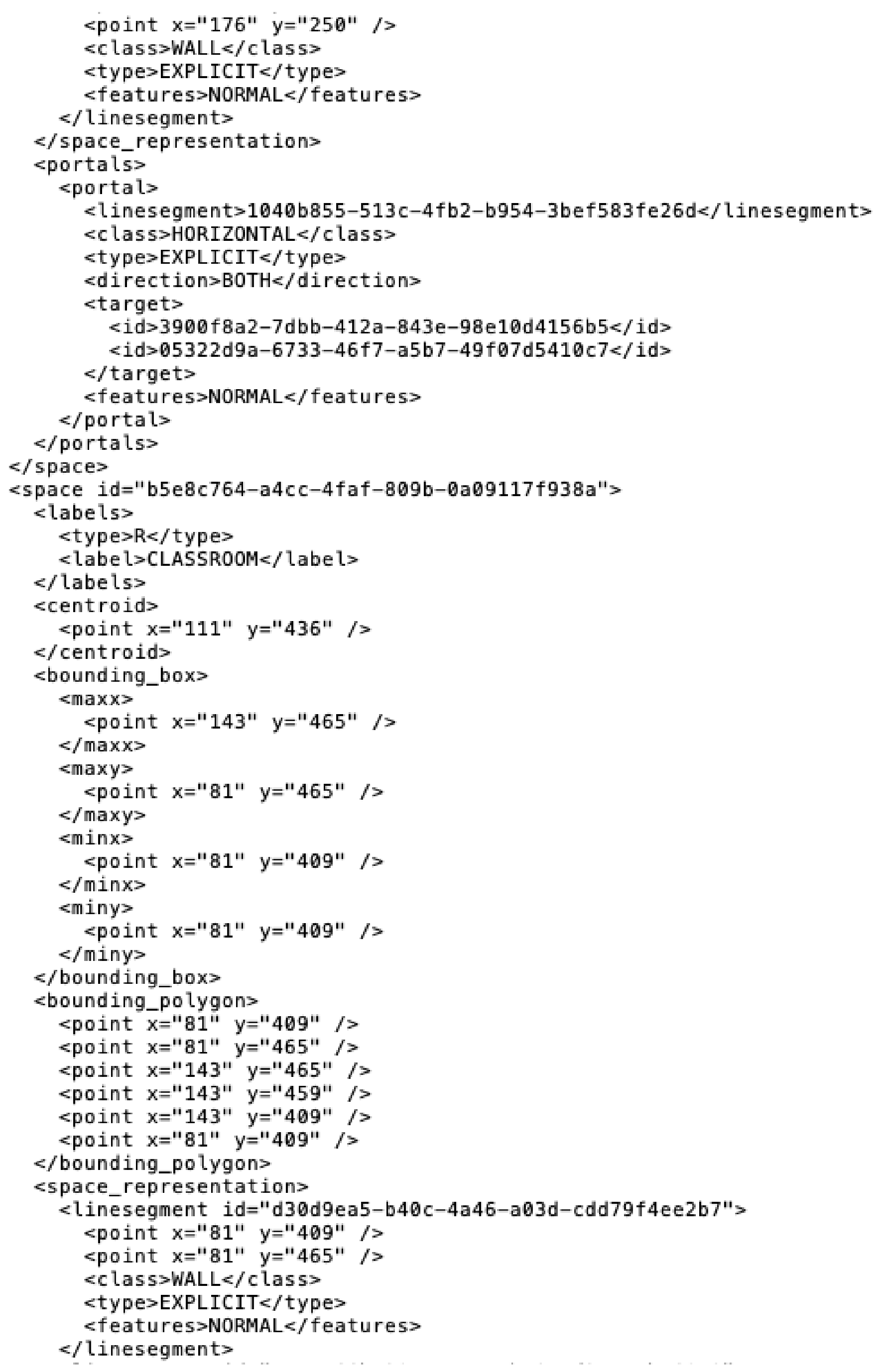
In this chapter, we propose a method for addressing the problem defined in Chapter 3 by providing an overview of the system we develop to perform the procedural generation of realistic layouts of indoor buildings.

4.1 Proposed methodology

Since the topology structure, namely the connection between different rooms, is the most important feature that we have to guarantee in the generated layouts, we base our approach on the graph representation of the original building.

As discussed in the last chapter in Section 3.2, we start from a graph representation of the original building floor plan. The problem of creating this graph from a building floor plan has already been addressed in previous works, like [23] and [58]. In particular, we consider a slightly modified version of the MIT format, which is still based on a XML representation of rooms as spaces connected by portals. In fact, we can also adapt our parser to use their datasets. In Figure 4.1 it is reported a fragment of the XML graph of the layout of a school, where it is possible to see the definition of a room (the *space* element) that still represents the original floor plan, since it includes the position of the room and its structural elements, like the walls.

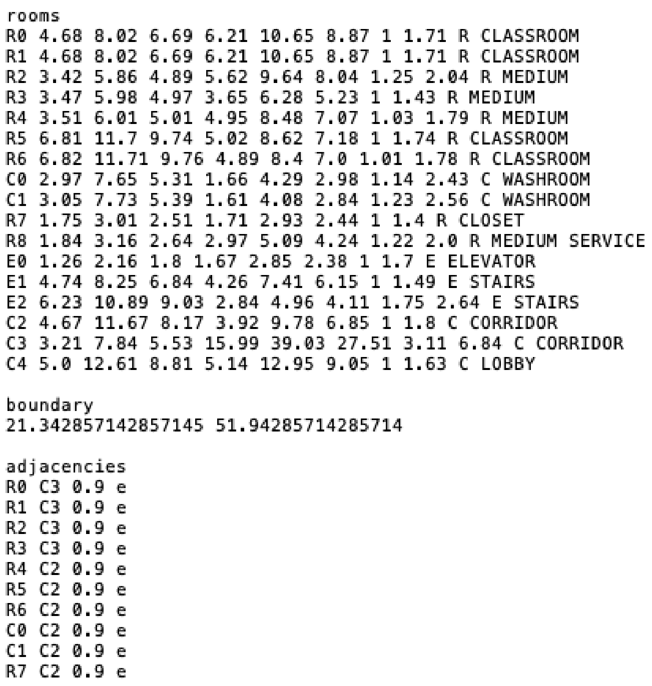
The input XML layout is processed into a plain-text file, which is the proper graph representation that we utilize as layout specification for the generation process. It includes the rooms sizes, their aspect ratio, and the adjacency between them. This format can be used to modify the layout specification or even to create a new one by hand. It is also possible to manually add specific constraints regarding the position of rooms in the generated layout. An example of an intermediate model file is in Figure 4.2.



```
      <point x="176" y="250" />
      <class>WALL</class>
      <type>EXPLICIT</type>
      <features>NORMAL</features>
    </linesegment>
  </space_representation>
  <portals>
    <portal>
      <linesegment>1040b855-513c-4fb2-b954-3bef583fe26d</linesegment>
      <class>HORIZONTAL</class>
      <type>EXPLICIT</type>
      <direction>BOTH</direction>
      <target>
        <id>3900f8a2-7dbb-412a-843e-98e10d4156b5</id>
        <id>05322d9a-6733-46f7-a5b7-49f07d5410c7</id>
      </target>
      <features>NORMAL</features>
    </portal>
  </portals>
</space>
<space id="b5e8c764-a4cc-4faf-809b-0a09117f938a">
  <labels>
    <type>R</type>
    <label>CLASSROOM</label>
  </labels>
  <centroid>
    <point x="111" y="436" />
  </centroid>
  <bounding_box>
    <maxx>
      <point x="143" y="465" />
    </maxx>
    <maxy>
      <point x="81" y="465" />
    </maxy>
    <minx>
      <point x="81" y="409" />
    </minx>
    <miny>
      <point x="81" y="409" />
    </miny>
  </bounding_box>
  <bounding_polygon>
    <point x="81" y="409" />
    <point x="81" y="465" />
    <point x="143" y="465" />
    <point x="143" y="459" />
    <point x="143" y="409" />
    <point x="81" y="409" />
  </bounding_polygon>
  <space_representation>
    <linesegment id="d30d9ea5-b40c-4a46-a03d-cdd79f4ee2b7">
      <point x="81" y="409" />
      <point x="81" y="465" />
      <class>WALL</class>
      <type>EXPLICIT</type>
      <features>NORMAL</features>
    </linesegment>
```

Figure 4.1: Fragment of the XML model of a layout

In this case it is possible to see the definition of the rooms with their sizes and their type. The *boundary* is the space available to generate the layout, and corresponds to a rectangle with the specified weight and height. At the end of the file there is the proper topology graph, represented as a list of adjacency between rooms, with additional information about the specific connection.



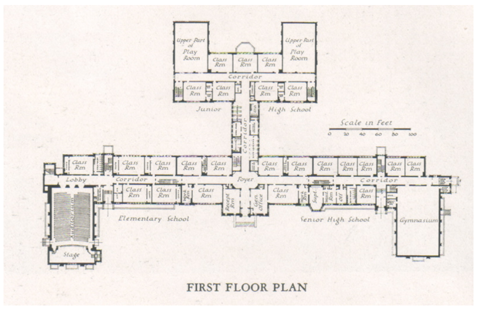
```
rooms
R0 4.68 8.02 6.69 6.21 10.65 8.87 1 1.71 R CLASSROOM
R1 4.68 8.02 6.69 6.21 10.65 8.87 1 1.71 R CLASSROOM
R2 3.42 5.86 4.89 5.62 9.64 8.04 1.25 2.04 R MEDIUM
R3 3.47 5.98 4.97 3.65 6.28 5.23 1 1.43 R MEDIUM
R4 3.51 6.01 5.01 4.95 8.48 7.07 1.03 1.79 R MEDIUM
R5 6.81 11.7 9.74 5.02 8.62 7.18 1 1.74 R CLASSROOM
R6 6.82 11.71 9.76 4.89 8.4 7.0 1.01 1.78 R CLASSROOM
C0 2.97 7.65 5.31 1.66 4.29 2.98 1.14 2.43 C WASHROOM
C1 3.05 7.73 5.39 1.61 4.08 2.84 1.23 2.56 C WASHROOM
R7 1.75 3.01 2.51 1.71 2.93 2.44 1 1.4 R CLOSET
R8 1.84 3.16 2.64 2.97 5.09 4.24 1.22 2.0 R MEDIUM SERVICE
E0 1.26 2.16 1.8 1.67 2.85 2.38 1 1.7 E ELEVATOR
E1 4.74 8.25 6.84 4.26 7.41 6.15 1 1.49 E STAIRS
E2 6.23 10.89 9.03 2.84 4.96 4.11 1.75 2.64 E STAIRS
C2 4.67 11.67 8.17 3.92 9.78 6.85 1 1.8 C CORRIDOR
C3 3.21 7.84 5.53 15.99 39.03 27.51 3.11 6.84 C CORRIDOR
C4 5.0 12.61 8.81 5.14 12.95 9.05 1 1.63 C LOBBY

boundary
21.342857142857145 51.94285714285714

adjacencies
R0 C3 0.9 e
R1 C3 0.9 e
R2 C3 0.9 e
R3 C3 0.9 e
R4 C2 0.9 e
R5 C2 0.9 e
R6 C2 0.9 e
C0 C2 0.9 e
C1 C2 0.9 e
R7 C2 0.9 e
```

Figure 4.2: Fragment of the intermediate model of a layout

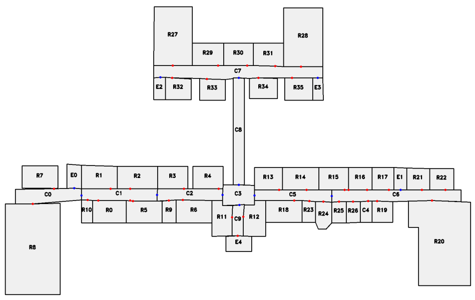
The intermediate model is used as input data for a layout generation technique inspired and adapted from the one proposed in [6], which is based on a MIQP formulation optimized with an external solver. The choice of starting from this method is motivated by the fact that such approach scales well even to medium and large layouts, and the definition of the constraints in a very explicit way can be used to guarantee that the generated building respects entirely the topology structure considered in input. Our formulation can handle only rectangles as rooms. This is not an issue for us since we don't care too much about particular shapes of the rooms, as stated previously in Section 3.3.

At the end of the generation process, the layout we obtain is drawn and converted to the same XML format utilized for the input.

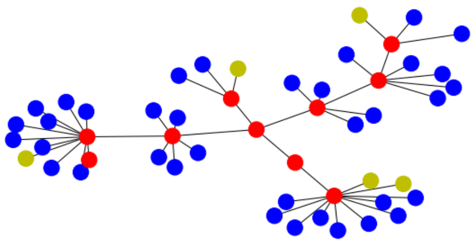
The various phases of our method are reported in Figure 4.3. In Figure



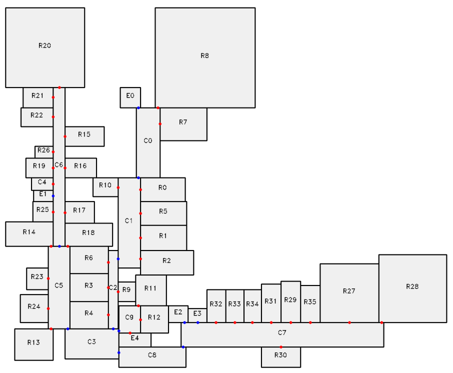
(a) Original floor plan



(b) Original layout in graph format



(c) Topology graph



(d) Generated output

Figure 4.3: Phases of our method

4.3a is shown the floor plan of the original building. Its conversion into the XML graph format is not directly considered in this thesis. In Figure 4.3b is pictured this graph model, which can be utilized to draw each room with the same location and structure of the original floor plan. Between the rooms, it is possible to see points that represent the *portals*, that are the connections between rooms (e.g., doors or passages). In Figure 4.3c is shown the topology graph of the layout, which is part of the intermediate model along with the sizes of the single rooms. In the figure it is possible to see the nodes of the graph colored according to the room type. The corridors are red, the entrances are yellow, and the generic rooms are blue. The result of our generation process, that is stored in the same graph format of the input layout, is displayed in Figure 4.3d.

In the following section we present the generation method, while in the next chapter we will discuss the complete system in detail.

4.2 Generation method

Considering the state of the art of layout generation techniques discussed in Section 2.3, we found that most of them are not adequate for our goal, since they can't handle mid or large layouts and enforce all the input constraints at the same time. The method we decided to employ is based on the one proposed by Wu et al. in [6], which is based on the formulation of the layout generation as a MIQP problem.

4.2.1 MIQP problem

In the operations research field, the *Mixed Integer Quadratic Programming* (MIQP) problem is a variant of the Mixed Integer Programming problem.

The formulation of an optimization problem is based on the definition of variables, which can be continuous or integer. The solving methods involve finding the value of the variables that either maximize or minimize a specified objective function while satisfying all the defined constraints, which are conditions that define the feasible combinations of values of the variables. The objective function is designed to reflect the goal of the specific situation for which the optimization problem is formulated. Since there are many complex methods to solve these problems, it is common to use solvers, which are mathematical software with various algorithms already implemented in a very efficient way.

The two main characteristics of the MIQP are the possibility to work with both continuous and integer variables, and that the objective function can be quadratic. An overview of quadratic problems can be found in [59] and [60]. Like in the linear case (MILP), the most common solving method of these problems is based on branch-and-bound [61].

4.2.2 Original method

As previously introduced in Section 2.3.5, the method proposed in [6] utilizes a MIQP optimization problem to generate the layout in a grid-based space. The input data required is the list of the rooms, considered as rectangles, with their target size (width and height), and their minimum and maximum aspect ratio. The *aspect ratio* is the ratio between the width and the height of the room, or viceversa, since the value is always considered ≥ 1. Another input that must be defined is the boundary of the layout, which is provided as a closed line on the grid. These are the mandatory information required. It is possible to specify also the adjacency between rooms, the absolute position of a room in the space, and the adjacency between a room and the boundary. In the basic formulation, each room is defined by five variables:

- x: integer variable that specifies the position of the bottom-left corner of the room along the x axis (horizontal).
- y: integer variable that specifies the position of the bottom-left corner of the room along the y axis (vertical).
- w: integer variable that specifies the width of the room, which is its size along the x axis.
- h: integer variable that specifies the height of the room, which is its size along the y axis. In the original paper it is called depth.
- o: binary variable that specifies the orientation of the room, which can be either vertical or horizontal.

The problem is based on finding the best allocation of rooms in the space defined by the boundary. This is performed by considering the objective function as the sum of two quadratic terms, which respectively control the total filling of the boundary and the closeness to the target sizes.

The validity of the generated layout is ensured by two constraints. The first imposes that every room is completely inside the boundary, while the second guarantees the non-overlapping of the rooms.

To satisfy the additional requirements, like the adjacency of rooms, there are other constraints. The most important are the aspect ratio constraints, which enforce the respect of the minimum and maximum values defined as input by considering the width, the height, and the orientation of the room, and the adjacency constraints, which impose the overlapping of the two specified rooms which, combined with the non-overlapping constraints, lead to the overlapping of just their borders.

The method can be expanded with a hierarchical framework, which allows to improve the details of single rooms and to generate large layouts efficiently.

The concept of improving rooms is utilized to close holes in the layout. It is based on the decomposition of the rooms around the hole into multiple rectangles. In particular, each room is considered as the union of two rectangles of the same size, obtained by dividing the original room along a random direction. A new optimization problem is formulated on this local sub-domain, using additional constraints to restrict the possible modifications of the new layout with respect to the initial situation. With this process, it is possible to obtain rooms with non-rectangular shapes.

The generation of large-scaled layouts is based on separating the complete generation into multiple problems. The first one is used to divide the initial boundary into macro-regions, which are later considered the boundaries for other problems. Utilizing this iterative process, it is possible to generate very large layouts with many simple optimization processes.

The formulated generation problem is very hard to solve, since there are many variables and the search space is huge. However, it is possible to accept sub-optimal solutions if they fulfill all the constraints and are obtained in a reasonable time.

4.2.3 Proposed method

The method we propose is based on the same mathematical formulation, but with continuous variables instead of integer ones. Since we need to guarantee the topological and structural properties of the original layout, we always employ the constraints to control the adjacency, the size, and the aspect ratio of the rooms. In fact, our input data contain all the adjacency relations and, for each room, also the minimum and maximum values for both the size and the aspect ratio. We consider the boundary only as a rectangle, while in the original formulation it could have complex shapes. The two reasons for this are that the layout specification is prepared in an autonomous way from an input layout, and that setting particular shapes of the boundary would restrict the generation process itself. However, it is still possible to manually add obstacles in the intermediate model in order to model the boundary in a specific shape.

Since the method already scales well on the number of rooms, we decided to not implement the hierarchical framework.

Instead, we found difficulties in finding solutions in reasonable time when cycles are present in the layout graph. To mitigate this issue, we develop a cycle decomposition technique that formulates in an autonomous way a local sub-problem around a cycle, and then utilizes the found solution to help the complete generation.

We also added parameters to account for the wall thickness and variable sizes of the doors between the rooms.

Since our formulation includes more constraints than the original one, the optimization process is typically harder to solve. For this reason, we added parameters to the objective function that allow the user to define weights for specific rooms, and in this way to control implicitly the optimization performed by the solver.

4.3 Summary

In this chapter we proposed a method to generate in a procedural way realistic layouts. Our technique is based on the extraction of the topological and structural features from the original plan. This process is performed by creating a graph representation of the original layout, which is saved in an intermediate file. This model, that can be manually modified or even created from scratch, is then used as input data for a generation method based on [6], which produces as output a new layout that is guaranteed to respect the topological structure defined in input.

Chapter 5

System architecture

In this chapter, we describe the implementation of the system proposed in Chapter 4. We start presenting a general overview of the system architecture, focusing on the pipeline structure and the layout format involved at each step. After that, we discuss each component of our software, highlighting its role in the generation process, the encountered problems, and how they have been addressed.

5.1 System pipeline

Our system can be described with the pipeline pictured in Figure 5.1. The circles represent the data structures employed for the layout, while the squares are the components that operate on them, for example with conversions or generations. Our system starts from an input layout in XML format that is used for generating new building layouts. As discussed in the previous chapter, we assume that we already have this file, which is derived from the original floor plan with methodologies like [23]. The XML model can already be considered a graph representation of the layout, since it is defined with spaces (the rooms) connected by portals. However, this format still contains all the locations of the rooms, including the coordinates of the walls. Since we just need the adjacency and the sizes of the rooms, we per-

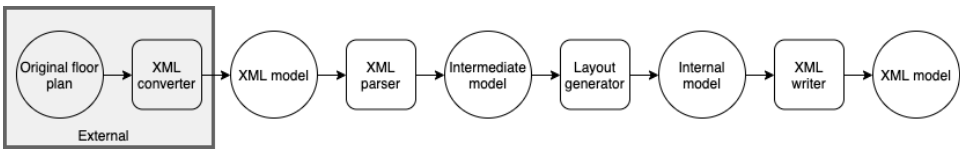


Figure 5.1: System pipeline

form a conversion that extracts this information from the XML format. As it can be seen in the pipeline, the result of this procedure is the intermediate model, which is encoded as a plain-text file. This new format allows the user to manually modify the layout structure and add new constraints which are later considered in the generation process. When the intermediate model is ready, another parsing is performed in order to prepare the MIQP formulation with the provided data. This procedure is straightforward, since the intermediate format contains the parameters used in the optimization problem. Once the generation is completed, the resulting layout is both drawn to file and saved in the same XML format utilized for the input layout.

Our system is entirely implemented in Python. In the rest of the chapter we describe each element of the pipeline in detail.

5.2 XML model

The first data structure of the pipeline corresponds to the input of our system. The original floor plan is considered already converted in XML format, which is also used to store the output at the end of the generation process. This format is based on the layout elements described in [23], but encoded differently in the actual file.

In both formats there is a complete description of the rooms and their adjacency. Each room is considered a *space* element, while connections between them are specified as *portals*. All the used coordinates are absolute in the floor space and are expressed as x and y pairs. A fragment of a XML model file was shown in Figure 4.1. The root of the XML tree in our format is the *building* element. Among its direct children there are different optional metadata fields. The only required element among them is *scale*, which is composed of *represented_ scale* and *real_ distance*. They are used to define the metric scale of the layout. In particular, the *real_ distance* element contains the size of the doors as a single value (it is assumed that all the doors in the layout have the same size, and the default is 90 cm), while *represented_ scale* measures how many units (which is the integer unit of measure used to express coordinates) correspond to the size of a door as defined in the other field. The layout itself is described in the *floor* element, that is a direct child of *building* and contains *spaces*, into which there is the list of *space* elements that represent the rooms. The format of *space* is:

- **id**: unique string that identifies the *space*. It is used in the definition of portals connections.

- **type**: the general type of the room, which is usually one between R - generic room, C - corridor, and E - entrance.
- **label**: a more detailed description of the type of the room, which identifies its main function (e.g., *classroom*, *bathroom*, *office*, ...).
- **centroid**: the position of the centroid of the room.
- **bounding_box**: the axis-oriented rectangle that circumscribes the room.
- **bounding_polygon**: the ordered list of points that define the boundary of the room as a polygon.
- **space_representation**: the ordered list of *linesegments* of the room. Each *linesegment* is composed of the following elements:
  - **id**: unique string that identifies the *linesegment*, used when representing a portal to define the connection in the portal list.
  - **points**: two points that define the starting and the ending of the entity considered by the linesegment (usually a wall, if it is a portal then the points share the same coordinates).
  - **class**: type of the linesegment entity. Can be *wall* or *portal.*
  - **type**: type that describes the portal, which can be *explicit* in case of a door or another generic passage with a barrier, and *implicit* if there is no physical barrier but the two spaces are still considered different. The case of *vertical* portals, included in the original MIT format, is not considered as we handle only single-level buildings.
  - **features**: not implemented, it should describe the type of door in case of portal.
- **portals**: the list of *portals* that are related to the current space. The *portal* element is formed by:
  - **linesegment**: id of the linesegment that this *portal* refers to. In case of explicit portal it represents the door.
  - **class**: used to define if the portal is *horizontal*, as in our case, or *vertical.*
  - **type**: the same type defined in the associated linesegment.
  - **direction**: used to represent the direction of the portal, that can be one-way or *both.* We only handle the latter case.

- **target**: the list of ids of the spaces that this portal connects. It defines the adjacency relations between the rooms.
- **features**: it is not implemented as in the linesegment case.

5.3 XML parser

The XML parser is the component that performs the conversion of the layout from the XML format to the intermediate one. The goal of this process is to extract the layout specification needed for the generation process. In fact, while the XML model contains the exact location of each structural element of the layout, like walls and doors, the intermediate model just represents the adjacency graph and the room sizes.

While the adjacencies between rooms are retrieved directly from the XML format since it is already based on a graph, the main problem is the definition of the size of all the rooms. In fact, the parser receives in input the rooms as polygons, while the result must be the size of a rectangle, since this is the only shape that is handled by our generation method. In particular, the intermediate model needs to contain the minimum, maximum, and target sizes (width and height), and the minimum and maximum aspect ratio.

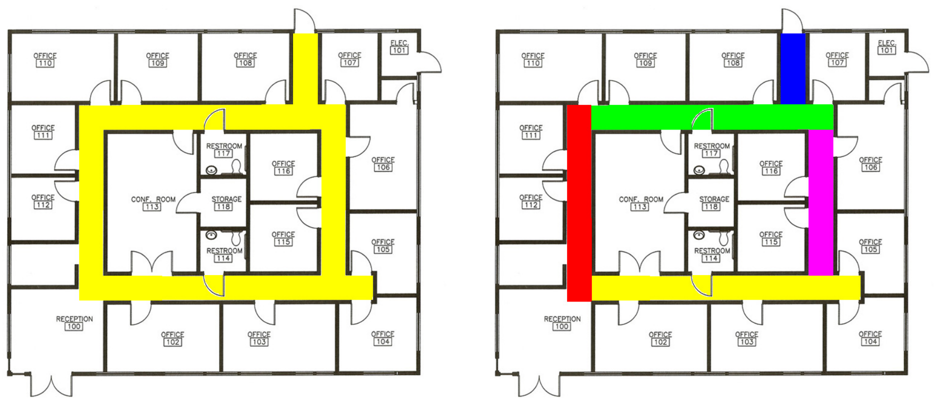
The boundary of the layout in the intermediate model is assumed to be a rectangle. To define its width and height (w_b, h_b), the parser considers the bounding box of the original layout.

$$w_b = x_{max} - x_{min}$$
$$h_b = y_{max} - y_{min}$$

There are two issues in the conversion of the rooms. The first is the presence of non-rectangular shaped spaces, while the second is the definition of the margin allowed as minimum and maximum for the sizes and the aspect ratio of the rooms. In the rest of the section, we discuss them in detail and present the solutions we devised to solve them.

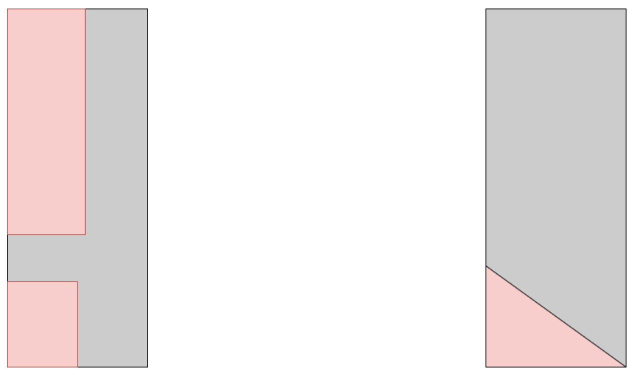
5.3.1 Room rectangularization

When converting the spaces into their intermediate model representation, it is possible that an input room is not rectangular. This is an issue that needs to be dealt with, since our generation method can only handle rectangular shaped rooms. However, our approach is focused on mid and large environments, like offices and schools, so we can usually consider the approximation of the shape of these rooms as a rectangle without relevant issues. An additional problem is how the original floor plan was subdivided into spaces.



(a) Single complex space *(b) Composition of simple spaces*

Figure 5.2: Example of different subdivisions of corridors in spaces



(a) *(b)*

Figure 5.3: Examples of problematic corridors

We assume that each space is more or less a rectangle, so that in case of complex structures, the space is actually considered as multiple spaces with implicit portals between them. For example, Figure 5.2 shows a floor plan with a corridor structure that is considered as with a single complex space in the first picture, and as multiple connected spaces in the second.

In the examples in Figure 5.3, the rooms in gray don't cover their entire bounding box, which is colored in red. These cases are common in real layouts and are usually considered as single spaces in the XML model. In Figure 5.3a, the bounding box still represents the main shape of the room, and considering the small lateral rectangle separately increases the topological complexity of the graph for no reason. Similarly, some rooms can contain oblique walls like in Figure 5.3b.

While we would prefer to handle just simple rectangular rooms, we develop a method to approximate problematic rooms based on their area. We consider the aspect ratio of the final room equal to that of the bounding box of the original space. To do so, we compute the area of the polygon that defines the original space. This value is by definition less than (or equal to) the area of the entire bounding box of the room, since the polygon shape is the part of surface of the bounding box that is effectively occupied. In particular, we consider the square root of their ratio, defined as:

$$ratio = \sqrt{\frac{area_{polygon}}{area_{bbox}}} = \sqrt{\frac{area_{polygon}}{(x_{max} - x_{min}) \cdot (y_{max} - y_{min})}}$$

This value is used to scale the sizes of the original bounding box to compute the target sizes of the room converted in rectangular shape. These sizes w^* and h^* are calculated as:

$$\begin{aligned} w^* &= (x_{max} - x_{min}) \cdot ratio \\ h^* &= (y_{max} - y_{min}) \cdot ratio \end{aligned}$$

5.3.2 Room variance

Once the target sizes of the converted room are defined as sizes of the rectangle, we need to consider how much they are allowed to vary in the generation process, since the MIQP formulation that we employ requires as hard constraints for each room a minimum and maximum value for the width, the height, and the aspect ratio. We decided to leave the definition of these bounds to the user, which has to insert three parameters. These parameters are expressed as percentage and are different between the corridors and the other rooms. In particular, the user defines the positive and negative margin for generic rooms and only the absolute value for corridors. The reason is that corridors usually require a much larger variance than the other rooms, which should be kept as close as possible to their target sizes. To prevent that the negative percentage could produce unrealistic room sizes, like corridors too narrow that don't allow the transit of people, we utilize an additional parameter *mindim* to enforce minimum guaranteed sizes of the rooms.

$$\begin{aligned} w_{min} &= \max(mindim, (x_{max} - x_{min}) \cdot (ratio - variance^-)) \\ h_{min} &= \max(mindim, (y_{max} - y_{min}) \cdot (ratio - variance^-)) \\ w_{max} &= (x_{max} - x_{min}) \cdot (ratio + variance^+) \\ h_{max} &= (y_{max} - y_{min}) \cdot (ratio + variance^+) \end{aligned}$$

The aspect ratio is computed with a more complex formula with a quadratic term. We still use the minimum and maximum variance provided, but in this case we don't multiply them directly since it would create extreme values for the rooms with an already high aspect ratio. The formula is based on the sum of two terms that depends on the average of the variances. In particular, the second term is quadratic in the aspect ratio, so that its effect is significant only with rooms with an high aspect ratio. Each term also includes a weighting parameter (α and β). The values we utilized are $\alpha = 1.5$ and $\beta = 0.2$. The minimum and maximum aspect ratio (ar_{min} and ar_{max}) are computed from the original one (ar^*) in this way:

$$variance_{avg} = \frac{variance^+ - variance^-}{2}$$
$$ar_{min} = \max(1,\ ar^* - \alpha \cdot variance_{avg} - \beta \cdot variance_{avg} \cdot (ar^* - 1)^2)$$
$$ar_{max} = ar^* + \alpha \cdot variance_{avg} + \beta \cdot variance_{avg} \cdot (ar^* - 1)^2$$

5.4 Intermediate model

The intermediate model is a text-based list of entities that compose the layout specification. It is created by the system by converting the XML file. This representation is later mapped directly to the MIQP problem.

Some information is lost during this conversion, as the initial XML format contains the rooms positions and structures while the intermediate model only their sizes and adjacency. This model also includes the size of the boundary into which the generation method will try to place the rooms. The intermediate model file can be easily checked and modified by hand, or even created completely from scratch. It is also possible to add here some additional constraints to the model, like the position of some rooms within the boundary.

As we seen in Figure 4.2, the model file is divided in different sections, delimited by a blank line and the name of the next one. In each line the delimiter between values is just a space. There is no specific end-of-line symbol. Each line that begins with "#" is commented. Not all sections are required, since some of them are used for optional constraints that are not employed by the converter.

If not specified otherwise, the sections are considered as lists, with each element on a new line after the initial section delimiter. All the values in square brackets represent optional elements. The sections are:

- **rooms**:
  - *id*: the unique name of the room. It is automatically provided by the converter as the combination of the letter corresponding

to the type of the room and a progressive number starting from 0 (e.g., R5 it is the sixth room with the type "R").

- *minw, maxw, minh, maxh*: the minimum and maximum sizes (width and height) of the room. They are hard constraints.
- *targetw, targeth*: the room target sizes that the optimization process tries to achieve. They represent the soft constraints of the problem.
- *minar, maxar*: the minimum and maximum aspect ratio of the room. They are hard constraints.
- *rtype*: the type of the room. In the layouts that we used there are just: *R* - generic room, *C* - corridor, and *E* - entrance.
- *label*: the label of the room. It usually describes the function of the room (e.g., *classroom*, *bathroom*, *office*, ...).
- *[onB]*: optional constraint that imposes that this room needs to directly touch at least a side of the external boundary.

• **boundary**:

- *w, h*: they represent the size of the boundary, which must be defined to formulate the MIQP problem. They are hard constraints.
- It is also possible to add in this section a list of obstacles. They are not used by the converter and can only be inserted manually. They are defined as:
  - *id*: the unique name of the obstacle.
  - *x, y*: the position of the bottom-left corner of the obstacle (which is a rectangle like the rooms) in the boundary space.
  - *w, h*: the (fixed) sizes of the obstacle.

• **adjacencies**:

- *id1, id2*: the ids of rooms to consider as connected. As only double-sided portals are considered, all the adjacency between rooms are assumed symmetric as well.
- *mina*: the minimum size of the adjacency. It is usually the door size. The conversion process automatically assign the door size value both to explicit and implicit portals. This is an hard constraint.
- *atype*: the type of adjacency. It is used to define explicit and implicit portals.

- **[positions]**:
  - *id*: the room to consider in a fixed position. At most one possible constraint of this type is allowed for each room.
  - *posx, posy*: the coordinates, within the boundary space, of the point that needs to be covered by the specified room.
- **[connections]**:
  - *id*: the room to consider in this constraint. Differently from the previous case, here the same room can have multiple entries, one for each time its id is repeated in the list.
  - *p1x, p1y, p2x, p2y*: p1 and p2 are two points that need to be both covered by the specified room. This constraint can be used to also control the orientation of the room (if there is a restriction of the aspect ratio), by specifying a pair of points that can only be covered with a certain orientation.
- **[clusters]**:
  - *clusterid*: the unique name of the cluster. After the cluster id, it must follow a list of rooms that belong to that cluster. Multiple clusters can be defined in this section leaving a blank line between the end of the list of rooms belonging to a cluster and the next cluster id. The list of rooms is defined as:
    - *id*: the room that belongs to this cluster.
    - *x, y*: the coordinates of the specified room.

5.5 Layout generator

The layout generation is performed through the solution of a MIQP problem. The input data of this formulation is retrieved from the intermediate model file. The layout specification, contained as plain-text format, is parsed and stored into a Layout object. This class is the foundation of the entire generation process, since it is used to store both the input parameters and the output layout.

5.5.1 Model

In this section we will present in detail our formulation of the MIQP problem.

As discussed in the previous chapter, the topology of the layout is an hard constraint of the optimization problem. There are hard constraints also to guarantee the minimum and maximum sizes of the rooms.

Another property we would like to enforce is that the area of the generated layout corresponds as much as possible to the original one. However, this can't be expressed in quadratic terms, since it would require a quadratic difference of the areas (already quadratic). Therefore, we decided to keep the objective function originally proposed in [6], while also introducing some parameters to control in a better way the generation process. As in the original formulation, the objective function is the weighted sum of two terms, both quadratic:

$$\min\ \lambda_{cover} \cdot E_{cover} + \lambda_{error} \cdot E_{error} \tag{5.1}$$

The first term controls the total area of the layout, guiding the optimization toward the filling of the entire boundary with the rooms. In particular, we added a parameter ρ_{cover} to control the importance of each room in this summation. The area term is:

$$E_{cover} = (w_b \cdot h_b) - \sum_{i}^{rooms} (w_i \cdot h_i) \cdot \rho_{cover_i} \tag{5.2}$$

The first product is the total area available in the boundary and it is used to keep the term positive. The summation operates on each room, computing the area of the single rooms weighted with ρ_{cover_i}, which is intended to decrease the importance of specific rooms in the entire term (usually the corridors). Weighting more this part of the objective function with λ_{cover} leads to fill the given boundary, expanding as much as possible every room.

The second term of the objective function tries to minimize the total divergence of the rooms sizes from the target ones given in input. As the error is already computed with a product, the term operates on the width and on the height independently. It also considers the orientation o of the room, that is a binary variable which represent the orientation of the room, to guarantee the right comparison between the target sizes. In particular, $o = 0$ means that the room is placed horizontally, with the width larger than the height, while with $o = 1$ the room has a larger height than width. The error term is:

$$E_{error} = \sum_{i}^{rooms} \rho_{error_i} \cdot ((h_i - o_i \cdot w_i^* - (1 - o_i) \cdot h_i^*)^2 + (w_i - (1 - o_i) \cdot w_i^* - o_i \cdot h_i^*)^2) \tag{5.3}$$

Also in this case there is a parameter ρ_{error_i} that allows a specific weight for certain rooms (typically to reduce the importance of errors of the corridors).

The rest of the summation is the sum of the error between the sizes of the room h_i and w_i, and their target values h_i^* and w_i^*, which are defined in the intermediate file, as seen in Section 5.4. In particular, the orientation variable o_i is used to consider the correct enforcement of the target size according to the alignment assumed by the room in a specific solution, selecting for example if the height h_i of the room should be compared with the target height or width, which are respectively h_i^* and w_i^*.

The constraints used by our formulation are linear and usually affects all the rooms. A list of the constraints and their explanation is provided below.

Minimum size constraint

This constraint is already present in the original formulation and is used to enforce the correct minimum sizes according to the room orientation. The only difference is that in our case both minimum sizes are enforced at the same time by using as *minD* the minimum between the minimum width and height, and their maximum as *maxD*.

$$\begin{cases} h \geq minD + (maxD - minD) \cdot o \\ w \geq minD + (maxD - minD) \cdot (1 - o) \end{cases} \quad \forall\, rooms \tag{5.4}$$

Maximum size constraint

This is a new constraints that we implemented. It is the dual of the previous constraint and its use is to constraint the maximum sizes of the rooms. In the original formulation in [6], the maximum size is typically restricted by the aspect ratio. We still utilize it, but we also added this constraint to control in particular the corridor width. In fact, using just the aspect ratio and the minimum sizes for the corridors is difficult to control their effective sizes, since they are usually very long but narrow, so with a high variance on the aspect ratio. With this constraint we can enforce both maximum sizes at the same time, producing very long corridors but with also complete control of the maximum width. As in the previous case, *minD* and *maxD* are respectively the minimum and the maximum among the maximum sizes inserted as hard constraints. In the implementation of [6], there is a similar constraint that sets the maximum size of the corridors according to their orientation. However, it operates only with a fixed size parameter, which is imposed on a single side of the corridors according to their orientation.

$$\begin{cases} w \leq maxD + (minD - maxD) \cdot o \\ h \leq maxD + (minD - maxD) \cdot (1 - o) \end{cases} \quad \forall\, rooms \tag{5.5}$$

Aspect ratio constraint

The aspect ratio constraint is the same of the one implemented in the original formulation. It is the first constraint that includes the constant M, which is a large number utilized in formulating logical operations through *binary variables*, which are integer variables restricted to the values 0 and 1. Here the constant M is used to control the enforcement of two alternative cases, expressed as different constraints, which are exclusively nullified by the addition or subtraction of M (according to the state of the binary variable) that guarantees that the constraint is valid for any allocated value of the other variables involved. We decided to actually enforce the aspect ratio constraint on every room except the corridors. The reason is that we already have the minimum and maximum size constraints and it is difficult, especially in converted layouts, to define good aspect ratio values for long corridors. Another important remark is that we only check the minimum and maximum aspect ratio, while the target value is not considered here but only in the error term of the objective function.

$$\begin{cases} minAR \cdot h \leq w + M \cdot o \\ maxAR \cdot h \geq w - M \cdot o \\ minAR \cdot w \leq h + M \cdot (1-o) \\ maxAR \cdot w \geq h - M \cdot (1-o) \end{cases} \quad \forall\, rooms \setminus corridors \tag{5.6}$$

Inside constraint

This constraint guarantees that each rooms remains in its entirety inside the boundary. It is implemented in the same way of the original formulation. w_b and h_b are respectively the width and height of the boundary.

$$\begin{cases} x + w \leq w_b \\ y + h \leq h_b \end{cases} \quad \forall\, rooms \tag{5.7}$$

Non-overlap constraint

Another fundamental constraint enforces the non-overlapping of different rooms. The differences between the original implementation and ours are the continuous nature of the variables (that doesn't affect the constraint formulation) and the addition in our case of the *walldim* parameter, which controls the wall thickness by leaving a certain gap between the rooms. In order to formulate this constraint, four additional binary variables are used for each pair of rooms, with the goal of controlling the simultaneous activation of the corresponding constraints. In fact, each auxiliary variable ω

is used to specify the relative direction (up, down, left, and right) of the second room with respect to the first one. This constraint operates also on the obstacles, if defined, that act as fixed rooms to not be overlapped.

$$\begin{cases} x_i \geq x_j + w_j + walldim - M\cdot(1-\omega_{right}) \\ x_i + w_i + walldim \leq x_j + M\cdot(1-\omega_{left}) \\ y_i \geq y_j + h_j + walldim - M\cdot(1-\omega_{up}) \\ y_i + h_i + walldim \leq y_j + M\cdot(1-\omega_{down}) \\ \omega_{right} + \omega_{left} + \omega_{up} + \omega_{down} \geq 1 \end{cases} \quad \forall\, i,j : rooms,\ i \neq j \tag{5.8}$$

Adjacency constraint

The adjacency constraint is used to guarantee the connection between two rooms by forcing them to share a common wall with at least a certain length. Our version is slightly more complex than the original one since we added the wall thickness and additional constraints to enforce that the room is at least larger as the connection on the side selected for the adjacency. The main idea is still the same: using an auxiliary binary variable α for each adjacency to select if the connection is horizontal or vertical, imposing in that direction also enough touching length between the rooms. This length is defined by the *minAdjacency* parameter, that usually represents the size of the door. However, the actual enforced value is computed by also considering the *doorUtilization* parameter, which is used to model the additional amount of space required for structural reasons.

This constraint forces an overlap of adjacent rooms, which will effectively just touch on the border because of the non-overlap constraint. In the original paper there is also a more advanced version of adjacency constraint, that is used to guarantee a connection between a certain room and two others, or at least one of them. We don't utilize this version of adjacency but only the described one.

$$\begin{cases} minA = \dfrac{minAdjacency}{doorUtilization} \\ x_{r1} \leq x_{r2} + w_{r2} + walldim - minA \cdot \alpha \\ x_{r1} + w_{r1} + walldim \geq x_{r2} + minA \cdot \alpha \\ y_{r1} \leq y_{r2} + h_{r2} + walldim - minA \cdot (1-\alpha) \\ y_{r1} + h_{r1} + walldim \geq y_{r2} + minA \cdot (1-\alpha) \\ w_{r1} \geq 2 \cdot walldim + minA \cdot \alpha \\ w_{r2} \geq 2 \cdot walldim + minA \cdot \alpha \\ h_{r1} \geq 2 \cdot walldim + minA \cdot (1-\alpha) \\ h_{r2} \geq 2 \cdot walldim + minA \cdot (1-\alpha) \end{cases} \tag{5.9}$$

$$\forall\, (r1, r2, minAdjacency) \in adjacency$$

Position constraint

This is an optional constraint that is used to force a specified room to pass over a certain fixed point within the boundary space. There is an analogous constraint in the original formulation that operates also on two points. It works by simply imposing that the point is inside the rectangle covered by the room. Only a single fixed point can be set for each room.

$$\begin{cases} x_i \leq x_{fixed} \\ x_i + w_i \geq x_{fixed} \\ y_i \leq y_{fixed} \\ y_i + h_i \geq y_{fixed} \end{cases} \quad \forall\, i : rooms,\ i \text{ is fixed} \tag{5.10}$$

Connection constraint

This is the two points version of the position constraint. The idea is the same, with both points that are constrained to be inside the room. The peculiarity of this constraint is that multiple pairs of points can be imposed to be inside the same room.

$$\begin{cases} x_i \leq x_{p1} \\ x_i + w_i \geq x_{p1} \\ y_i \leq y_{p1} \\ y_i + h_i \geq y_{p1} \\ x_i \leq x_{p2} \\ x_i + w_i \geq x_{p2} \\ y_i \leq y_{p2} \\ y_i + h_i \geq y_{p2} \end{cases} \quad \forall\, (i, p1, p2) : connections \tag{5.11}$$

Boundary constraint

This optional constraint is used to guarantee that a room touches at least one side of the boundary. It is a simplified version of the original one, since ours doesn't account for obstacles. We decided to ignore them because they are not used by the converter. Also in this case, four auxiliary binary variables β are used to select the boundary side to touch for each room. It is also possible to use global β variables to force all the rooms affected by this constraint to be on the same boundary side.

$$\begin{cases} x_i + w_i \geq w_b - M \cdot (1 - \beta_{right}) \\ x_i \leq M \cdot (1 - \beta_{left}) \\ y_i + h_i \geq h_b - M \cdot (1 - \beta_{up}) \\ y_i \leq M \cdot (1 - \beta_{down}) \\ \beta_{right} + \beta_{left} + \beta_{up} + \beta_{down} \geq 1 \end{cases} \quad \forall\, i : rooms, i \text{ fixed to boundary} \tag{5.12}$$

Cluster constraint

This is a new optional constraint that we use to fix relative positions among a certain group of rooms, which is called *cluster*. While the previous constraints enforce the location of a room within the boundary given one or two fixed points, in this case the specified coordinates are considered relative between the rooms of the same cluster. In fact, the cluster itself can be freely positioned within the boundary, as long as all the other constraints of the rooms allow that. The functioning of this constraint is similar to the position constraint defined previously, but with two differences. The first is that the enforced point is not fixed but is composed of two additional continuous variables, $x_{cluster}$ and $y_{cluster}$, that specify the position within the boundary of the entire cluster. In fact, this constraint forces that each room is located in a certain relative position to the cluster. The coordinates of the room in the cluster space are x_f and y_f, which are defined in the intermediate model in Section 5.4. The second difference is the addition of a *margin* parameter, which allows a certain deviation from the fixed coordinates by enlarging the room sizes when enforcing the position constraint. This leads to a larger degree of freedom in the positioning of the room by the solver. The *margin* parameter can be set zero (eliminating this additional offset) or even increased. It is possible to insert the same room in different clusters.

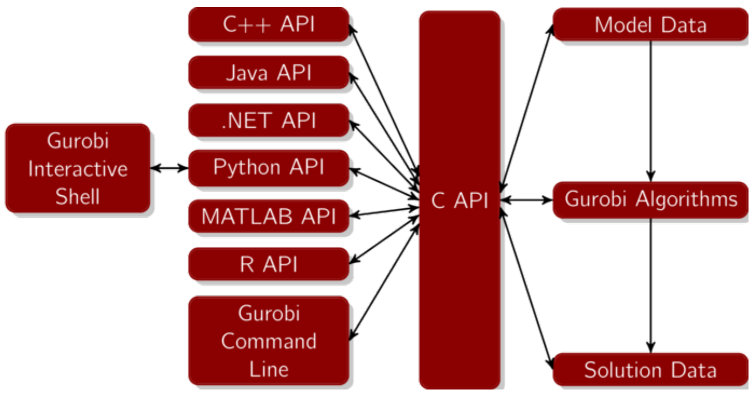


Figure 5.4: Overview of Gurobi API

In this way, the room is affected by both constraints at the same time.

$$\begin{cases} x_i - margin \leq x_{f_{ij}} + x_{cluster_j} \\ x_i + w_i + margin \geq x_{f_{ij}} + x_{cluster_j} \\ y_i - margin \leq y_{f_{ij}} + y_{cluster_j} \\ y_i + h_i + margin \geq y_{f_{ij}} + y_{cluster_j} \end{cases} \quad \forall\, i : rooms, j : clusters, i \in j \tag{5.13}$$

5.5.2 Implementation

The optimization is performed with Gurobi 8.1 [62], using the Python API to interface the solver with our system. Gurobi is a state of the art commercial solver, able to solve optimization problems of different nature, both linear and quadratic. The API architecture is described in the reference manual[1] with the structure represented in Figure 5.4. Gurobi allows to customize the optimization process through various parameters. We utilize the following:

- *TimeLimit*: allows to define a maximum time limit for the optimization. We need to set this parameter since we are only interested in a good solution and not in the optimal one (which could require a huge amount of time to be found because of the complexity of our problem). The typical time limit we use is 4 minutes.

- *Presolve*: controls the presolve level, which are preprocessing algorithms that try to reduce the search space of the problem and make the model easier to solve. We use $Presolve = 2$, which corresponds to the most aggressive level.

[1]https://www.gurobi.com/documentation/8.1/refman/index.html

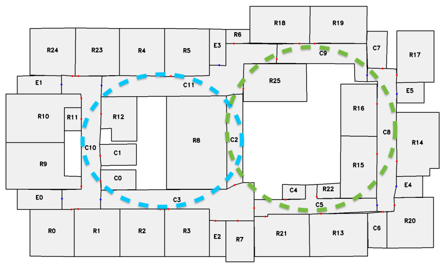
- *Heuristics*: it is the percentage of time allocated in MIP problems to the heuristics. The default value is 0.05, which means that the solver tries to find new solutions instead of improving the current bounds in only 5% of the runtime. We use a very high value of this parameter (usually between 0.6 and the maximum allowed, which is 1) since our model often requires a lot of efforts to even just find a feasible solution. This is one of the most important parameters that the user has to set. For smaller and simpler layouts, 0.6 is usually enough, while complex ones require a much higher value.

- *MIPFocus*: defines the high-level strategy utilized by the solver. There are three specific strategies and we found that different layouts perform better with different values of this parameter. We usually utilize the value 2 by default. With this strategy, the solver focuses on proving the optimality of the found solutions.

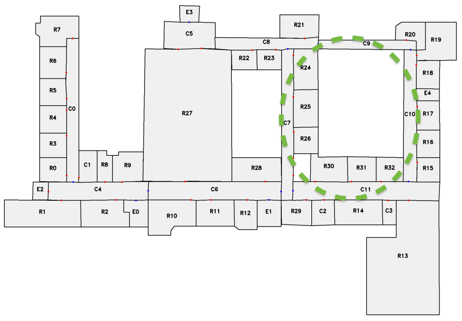
The generation workflow begins with the loading of the layout specification from an intermediate model file, usually created by the XML parser. This parsing procedure is very simple, contrary to the one described in Section 5.3, since the intermediate model embeds all the input parameters of the MIQP problem. A Layout object is instantiated and filled with all the input data. Once this procedure ends, the system prepares the Gurobi model by creating the objective function and all the constraints utilized in the specific problem. When the model is filled, the optimization process begins. After the defined time limit, the system stores the best solution found inside the Layout object itself. In the worst case, if no solution is found the program terminates here. If a solution is found, the Layout object is saved in a dump file using the pickle[2] library, which serializes and stores the class in a binary file, allowing its content to be reloaded in the future. After that, the system draws the generated layout in an image file, which can be used to quickly examine the result of the generation. At the end of the process, the system saves the generated layout in a XML file, which follows the same format of the initial input.

The optimization in some cases fail in identifying a feasible solution due to the computational complexity of the given instance of the problem. In particular, we found that this method works very well even with a large number of rooms, but can have problems in finding solutions when the layout contains cycles. To mitigate this issue, we develop a cycle decomposition technique that is based on solving a sub-problem with a lower number of

[2]https://docs.python.org/3/library/pickle.html



(a) Layout with two cycles



(b) Layout with a single cycle

Figure 5.5: Example of cycles in the layout graph

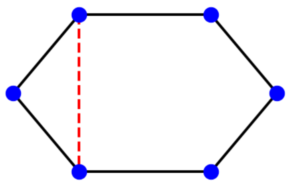
rooms and then applying some additional constraints when solving the global generation problem. We now present this method in detail.

5.5.3 Cycle decomposition

The cycle decomposition is a technique that helps finding a solution if the generation does not succeed. The first step is to find a cycle in the layout graph. Two example of layouts with cycles are shown in Figure 5.5. To operate on the graph, we employ NetworkX [63], which is a popular Python library to study and visualize graphs. In particular, since we work with planar graphs, we utilize the minimum cycle basis [64] to identify the minimum length cycles that don't contain internal chords. A chord is an edge that connects directly two non-consecutive nodes of the cycle, like the red edge in Figure 5.6a. We also want minimum cycles, so without internal shortest paths like the red one in Figure 5.6b. Internal connections like these can be problematic in two ways. The first is that the sub-problem could ignore these connections and impose positions for the cycle rooms that prevent the presence of other internal rooms between them. The second issue is that in case of a graph like the one in Figure 5.7, if the system selects the cycle composed of all the nodes connected by black edges, then the sub-problem would be equal to the global problem since all the original rooms are considered.

If there are multiple cycles with no inner connections, a parameter allows the user to choose the longest or the shortest one. The choice between the two depends on the specific layout, but usually selecting the longest cycle works better.

Once the target cycle is identified, a new graph is created with all the rooms that compose the cycle. After that, every room directly adjacent to


(a) Cycle with a chord

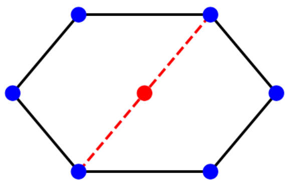

(b) Non-minimum cycle

Figure 5.6: Example of graph cycles with internal connections

them is also added, including the edges between such nodes.

When the new graph is ready, it is used to formulate a new problem with only these rooms. The result of this generation, which is performed with a lower time limit than the global one, typically 40 seconds compared to 4 minutes in the global generations, is utilized only to extract the center positions of the rooms that compose the cycle. These locations are stored in a new cluster, which structure has been previously defined in Section 5.4. Finally, the cluster is normalized. This procedure, as can be seen in Figure 5.8, is a translation of all the positions of the rooms to ensure that the coordinates are the lowest possible. The reason behind this is that we only need their relative position, while their absolute location is later determined in the global generation by adding to these coordinates the cluster offset, which is the same for the entire cluster. In this way, the generation process is free to place the cluster anywhere in the boundary, while keeping the relative distances between the rooms.

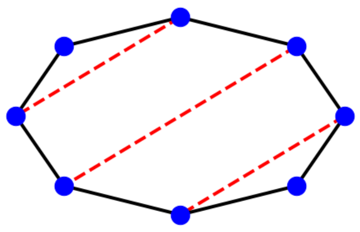


Figure 5.7: Example of large non-minimum cycle

Once this entire process ends, the system prepares the new global generation, adding the retrieved cluster to the layout specification. This problem corresponds to the one formulated initially, with the addition of the cluster

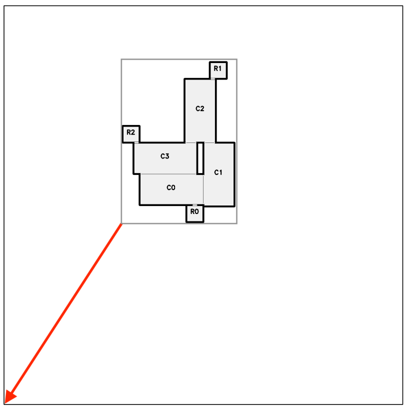


Figure 5.8: Normalization of the cluster

constraints, defined in Section 5.5.1, to impose the relative positions of the rooms of the cycle previously decomposed.

If no solution is found even in this case, the system performs another cycle decomposition. However, the rooms already in clusters are removed from the graph before the cycle retrieval. If there are no more cycles to decompose, the program terminates.

The goal of this method is to help the global generation by restricting the possibilities for some variables, specifically the most problematic ones, which are the positions of the rooms that compose cycles.

A issue of this technique is that it could create unfeasible clusters. In fact, this method doesn't account for rooms that are not immediate neighbors of the rooms that compose the cycle. To try to solve these issues, we utilize the *margin* parameter of the cluster constraints. This parameter, as defined in Section 5.5.1, increases the freedom of placement of the rooms of the clusters, allowing the global generation process to deviate by a small margin from the positions imposed by the sub-problem. The value of the *margin* parameter that we typically use is 5.

5.6 Drawer

After the generation process is complete and a layout is found, the system draws it to an image file. We don't consider furniture, windows, or other elements of the building, but just the walls and the connections between the rooms. An example of generated layout can be seen in Figure 5.9.

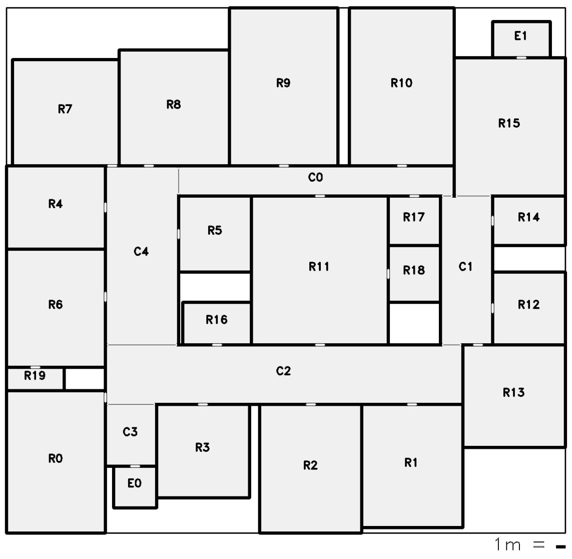


Figure 5.9: Example of generated layout

The represented elements in the image are the walls, the labels of the rooms, and the connections between them. In particular, the system utilizes the original portal type to define whether a portal should be a door or an implicit portal. The width of the door corresponds to the minimum adjacency size defined in the layout specification, while its position is always assumed to be at the center of the wall shared by the two adjacent rooms. An example of both kind of connections is shown in Figure 5.10.

5.7 XML Writer

The final operation performed by the system is to store the generated layout in an output file, utilizing the same XML format of the input. In this way it is possible to use both layouts with the same tools, with no need to implement a new parser for another format.

The id of each space in the output doesn't correspond to their original value in the input file, but rather to the id that we assigned to the room in the intermediate model, presented in Section 5.4. This makes them easily recognizable in the XML file. Moreover, since the ids we define correspond to the type of the room joined by the order of the room in the list of all spaces (e.g., R2 is the third room in the file with type "R"), it can also be

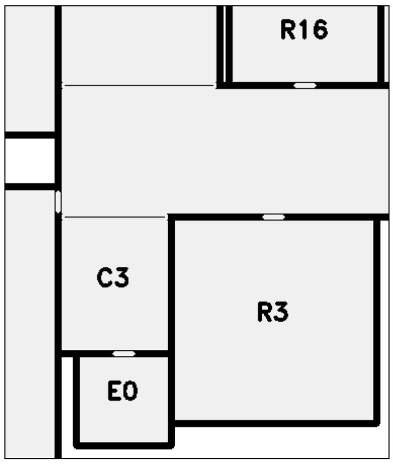


Figure 5.10: Example of explicit and implicit portal representations

derived in an easy way from the original file with a simple parsing procedure.

5.8 Summary

In this chapter we described the implementation of our system. We started illustrating the pipeline of the entire process, which operates at each stage with a different data structure for the layout. We then discussed each component in detail, focusing in particular on the issues they face and how they solve them.

Chapter 6

Experimental results

In this chapter, we present some building layouts generated with our method and the experiments we performed to validate them. We start showing some examples of generations, making considerations on the parameters used and how they affect the results. Next, we introduce the infrastructure that we utilize to perform the experiments. After that, we present the two experiments we decided to use to evaluate the realism the generated layouts. The first one involves static path planning between rooms, while the second one involves exploration of the environments by a simulated robot. For both the experiments, we present and comment their results.

6.1 Generated layouts

In this section we make some considerations about the generation process. At first, we cover how to control the area of the generated layouts. We then discuss some issues we found when running the method. Finally, we present some examples of generated layouts.

6.1.1 Area control

As our method starts from the layout of a building and generates from it new different layouts with the same structure, we can compare the main features of the generated layouts against the initial one. But while every layout generated with our method is guaranteed to respect the topology graph defined in input, it is not possible to impose the equivalence of areas between the input layout and the generated one. However, the user can utilize the λ_{cover} parameter of the objective function defined in Section 5.5.1 to guide the optimization process and control in an implicit way the area of the generated layout.

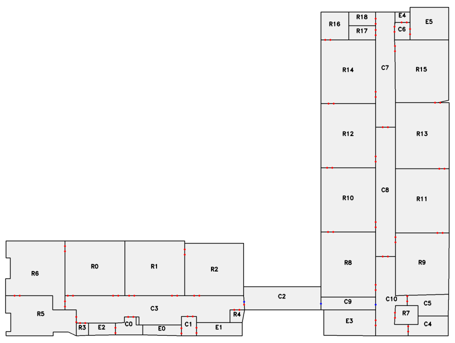


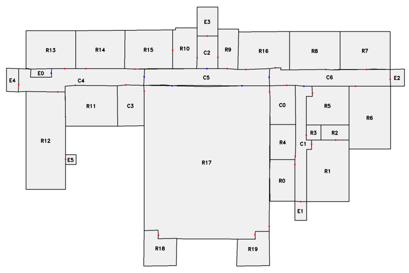
Figure 6.1: Original layout of a school

This parameter weights the sum of the two terms E_{cover} and E_{error} of the objective function of the MIQP problem. In general, using a value $\lambda_{cover} = 1$, like in the original formulation in [6], leads to layouts with a much larger area than the original one, since the boundary size is automatically defined as the bounding box of the original layout, which can be an overapproximation of the original boundary for layouts like the one shown in Figure 6.1.

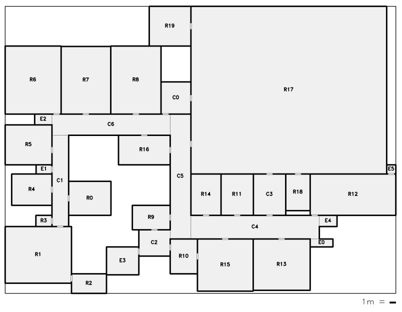
The area of the building is influenced directly by three factors in the optimization. The first is the cover term E_{cover} of the objective function, that is an explicit maximization of the total sum of the areas of each room. The second factor is the other term of the objective function, E_{area}, which minimizes the displacement of the sizes of the rooms from the target ones. The third factor is the minimum and maximum sizes of the rooms, which are hard constraints and guarantee that the width and height of each generated room don't deviate much from the original values defined in the layout specification.

We found that a λ_{cover} value between 0.1 and 0.2 allows to generate layouts with areas similar to the original ones, even when the bounding box used as boundary doesn't fit very well the original shape of the building, like the school in Figure 6.1. Lower values like 0.001 or less can be used, but the resulting area will be much smaller than the original. In addition to the λ_{cover}, it is also possible to tune the optimization process using the ρ_{cover} and ρ_{error} parameters, allowing a finer control of the importance of the two terms of the objective function with specific values for each room. Their default values are 1 for each room.

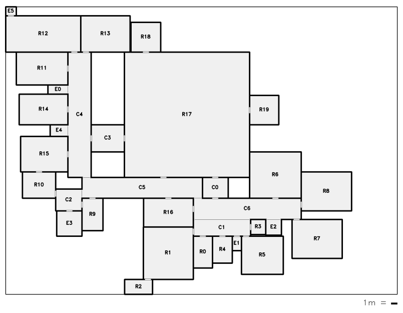
In Figure 6.2 we show an example of generated layouts with different λ_{cover}. The generated layout with the area closer to the original in Figure 6.2a is the one with $\lambda_{cover} = 0.1$, which is shown in Figure 6.2c. The gener-



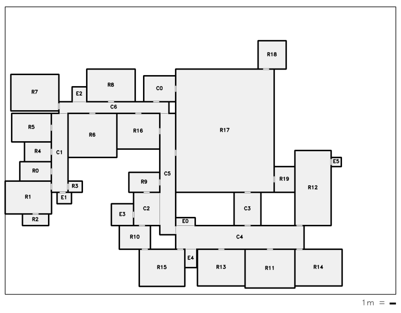
(a) Original - 1530 m^2



(b) Generated - 2278 m^2 - $\lambda_{cover} = 1$



(c) Generated - 1448 m^2 - $\lambda_{cover} = 0.1$



(d) Generated - 1268 m^2 - $\lambda_{cover} = 0.0001$

Figure 6.2: Comparison of generated layouts of a school with different values of λ_{cover}

ated version with $\lambda_{cover} = 0.0001$ is smaller but still relative close, while the one with $\lambda_{cover} = 1$ is completely different considering the area.

6.1.2 Generation issues

We accept any solution found by the optimization process, since the hard constraints of the MIQP formulation assure that every feasible solution respects the topology graph and the rooms sizes defined in input. However, there is no guarantee that the solver can actually find a solution in a certain amount of time.

In these cases it is possible to help Gurobi by changing some of its parameters, defined in Section 5.5.2. In particular, increasing the *TimeLimit* and the *Heuristics* values can provide a massive help to the solver. In addition, it is also possible to enlarge the boundary size in the layout specification to facilitate the search of a feasible solution.

Here, we present the main issues that we found with our method.

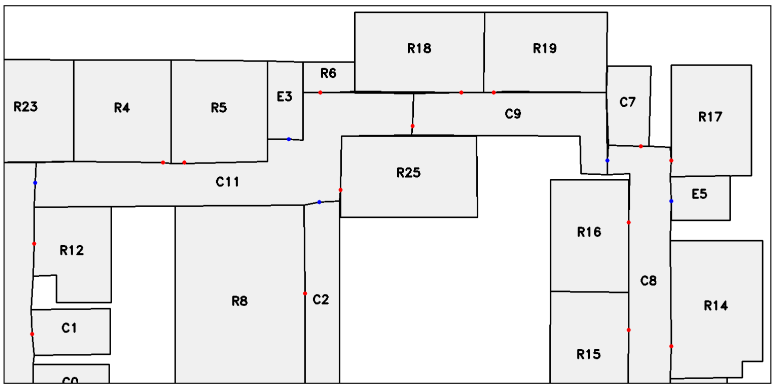
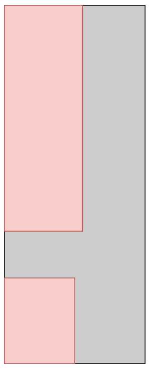
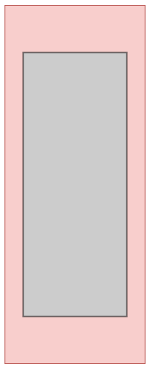


Figure 6.3: Example of non-rectangular rooms in the original layout



(a) Original



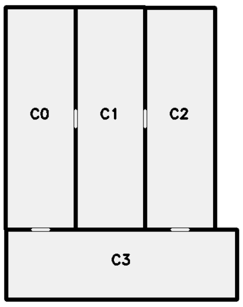
(b) Approximated

Figure 6.4: Examples of approximation of a non-rectangular room

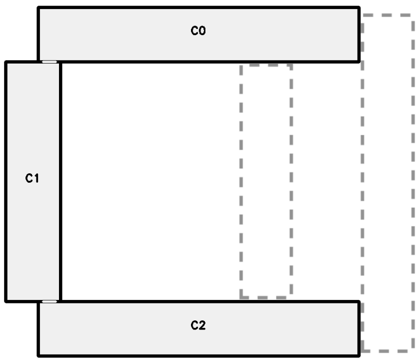
Non-rectangular rooms

The first issue is caused by particular room structures in the input layouts. An example is shown in Figure 6.3, in which the corridors C8, C9, and C11 have non-rectangular shapes. The XML parser approximates these rooms with the technique presented in Section 5.3.1. However, utilizing these approximations can lead to an unfeasible layout specification. In fact, the conversion procedure is based on keeping the original area for the rectangular room, but since the final aspect ratio is the one of the initial bounding box, the resulting room will be shorter and wider in case of a corridor like the one reported in Figure 6.4, where in red is represented the original bounding box and in gray the room.

If the original room has a relative small area, but a large bounding box caused by an highly irregular profile, like the corridor C11 in Figure 6.3,



(a)



(b)

Figure 6.5: Examples of cycles

then it is possible that the approximated room is too short to connect all the required rooms.

To solve this issue, a possible solution is to increase the $variance^+$ and $variance^-$ values, which are used, like discussed in Section 5.3.2, by the XML parser to specify larger bounds for the sizes of the rooms in the layout specification. However, utilizing high values for $variance^+$ and $variance^-$ leads to a much larger search space of the MIQP problem, which increases the difficulty of the solver in finding feasible solutions.

Presence of cycles of rooms in the topology graph

Another issue, already discussed in Section 5.5.2, is the presence of cycles of rooms in the topology graph. Imposing the adjacency between the rooms of a cycle is harder than the normal case, since it is not just a problem of finding the right locations of the rooms but also the right sizes. In particular, the main difficulty is the correlation that exists between the sizes of the rooms of a cycle. In fact, while a trivial solution like the one reported in Figure 6.5a can be found if there are no additional constraints beyond to form a cycle, in a more complex scenario like in Figure 6.5b, the final room to close the cycle is constrained in both position and size.

In small and medium layouts, the solver can manage this additional complexity and find a feasible solution without too much difficulties. However, this become a noticeable issue if the layout contains a large number of rooms. Moreover, this problematic is not addressed in the methods presented in the state of the art in Section 2.3, since most of them don't consider large layouts, completely avoiding this issue.

The cycle decomposition technique, described in Section 5.5.3, helps the solver in these situations by isolating a cycle, solving a sub-problem limited to those rooms, and then keeping their relative positions in the global generation problem.

The use of this method is optional, since it doesn't guarantee to actually find a solution and it can even prevent the generation of the layout. In fact, the relative positions fixed in the sub-problems could be unfeasible in the global formulation of the problem, since they don't account for the presence of rooms that aren't directly part of the cycle or immediate neighbors. In addition, another downside of this technique is that it requires an additional global generation each time it is applied, until a solution is found or until there are no more cycles to decompose. Even if the *TimeLimit* of the sub-problem is typically chosen lower than the original one, it is usually better to try multiple Gurobi parameters choices before applying this method.

6.2 Generation examples

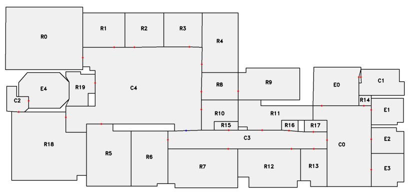
In this section we present some layouts generated with our system. The dataset we utilized is composed of schools and offices of medium and large scale.

For each instance of the problem, defined as the combination of the layout specification and all the used parameters, Gurobi always finds the same solution. However, it is possible to obtain multiple different layouts from the same layout specification in input by introducing small perturbations in it. To do so, the most simple and effective way we found is fixing a location of a certain room with the position constraint. This allows to generate a large number of layouts from the same input, as originally specified in the requirements in Section 3.3.

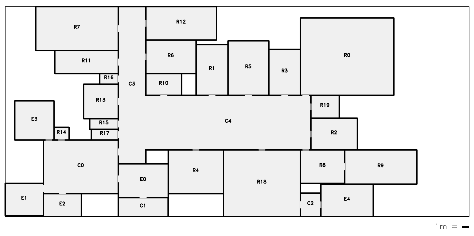
All the generations are performed on a computer with a 1.7 GHz Intel Core i7 processor and 8GB of DDR3 RAM. The main computational bound was the dual core CPU, while the memory usage was relatively low.

Office 1

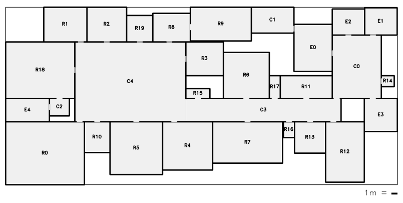
In Figure 6.6 is shown the original layout and three generated versions of a medium sized office. The layout in Figure 6.6b is the layout generated directly from the input layout. To generate the other two, we fixed the position of the room E1 within the boundary by manually editing the intermediate model file. In this way, we obtained two generated layouts that are different



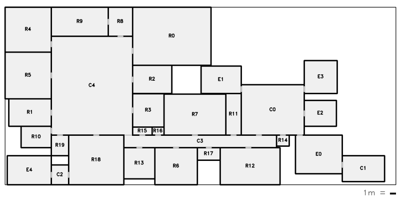
(a) Original - 1590 m^2



(b) Generated - 1325 m^2 - $\lambda_{cover} = 0.1$



(c) Generated - 1618 m^2 - $\lambda_{cover} = 0.1$



(d) Generated - 1379 m^2 - $\lambda_{cover} = 0.1$

Figure 6.6: Office 1

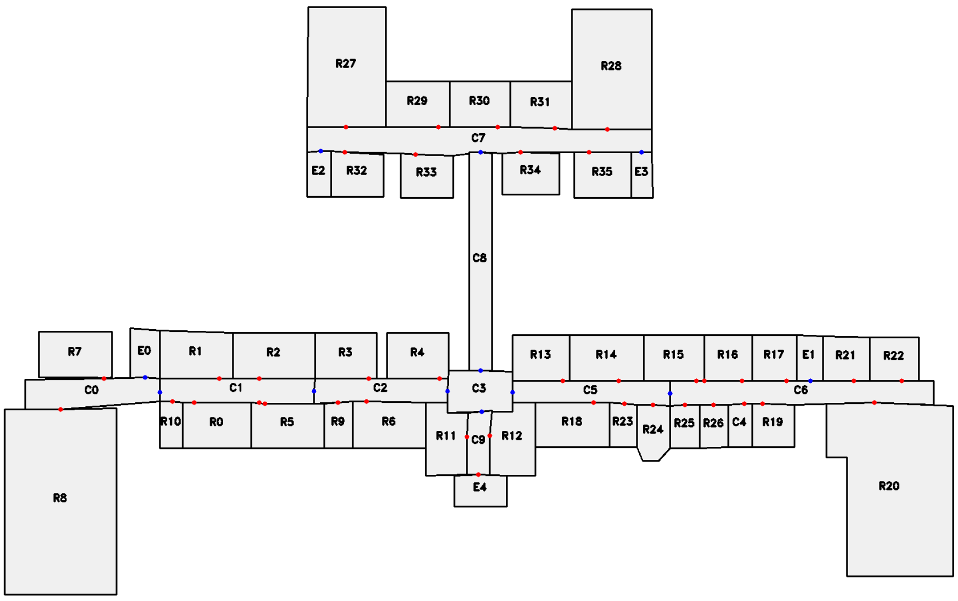
from the first one. We can see that the general structure is similar between all the layouts, and in particular the largest rooms in the original layout are still the largest in the generated versions.

School 1

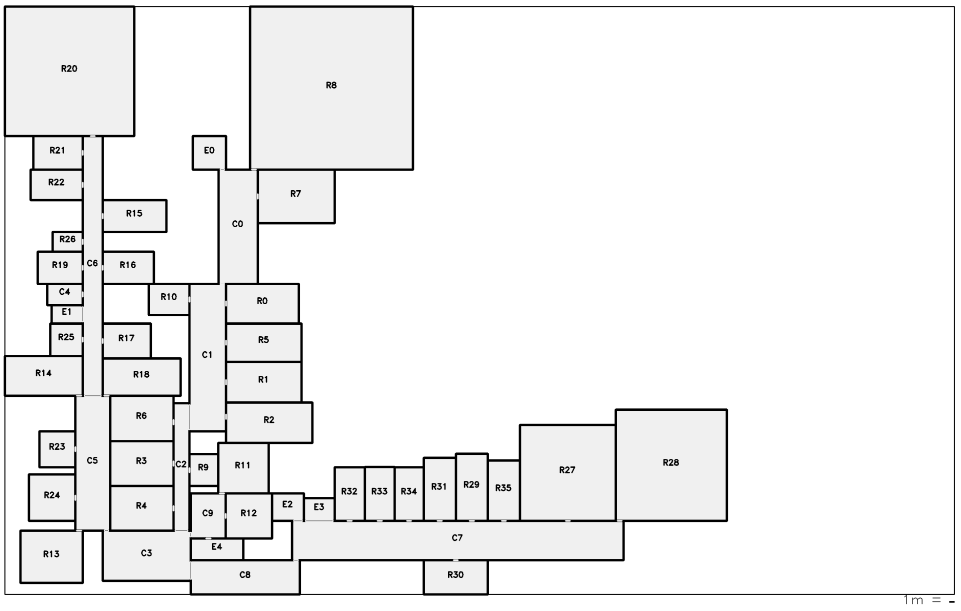
In Figure 6.7 is shown the generation of a large scale school. In this example, we can see that our method handled well the generation of a large layout, since despite the bounding box is not fit to the original building shape, the areas of the two layouts are very similar.

School 2

The generation of this layout, pictured in Figure 6.8, is more difficult than the previous examples because of the presence of a cycle. To find a feasible solution, we employ the cycle decomposition technique illustrated in Section 5.5.3. We report in Figure 6.9a the layout generated for the sub-problem, which contains less rooms than the global one. The complete result of the generation is shown in Figure 6.9b.



(a) Original - 7166 m^2



(b) Generated - 7124 m^2 - $\lambda_{cover} = 0.2$

Figure 6.7: School 1

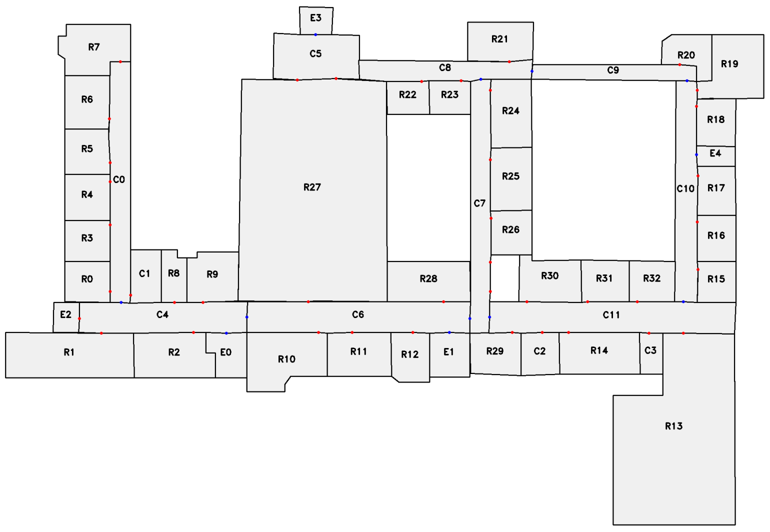


Figure 6.8: Original - 2764 m^2

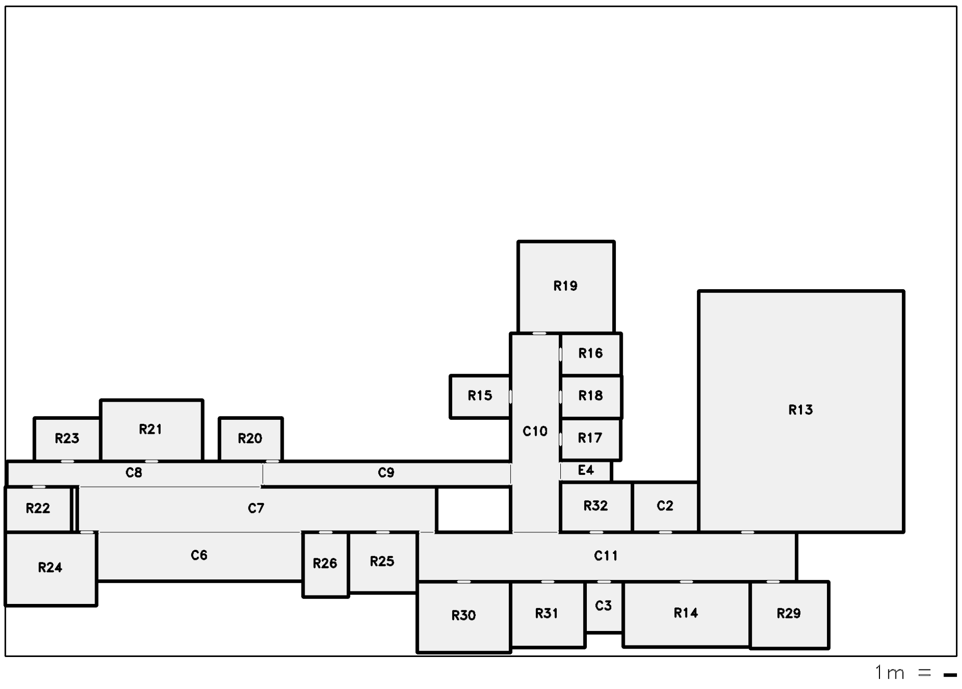
6.3 Simulated robot setup

As stated in Section 3.2, we want to generate realistic building layouts to perform experiments with simulated robots on a large number of environments with consistent results. To evaluate if the layouts that we generate are similar to the original ones, we utilize two approaches. The first is to consider the length of the paths planned between some rooms, while the second is the time and the traveled distance required to explore the environment. If these values are similar in the generated layouts and in the original environments, then we can say that the former ones are "functionally" similar to the latter ones, meaning that the effort of performing tasks in them is comparable.

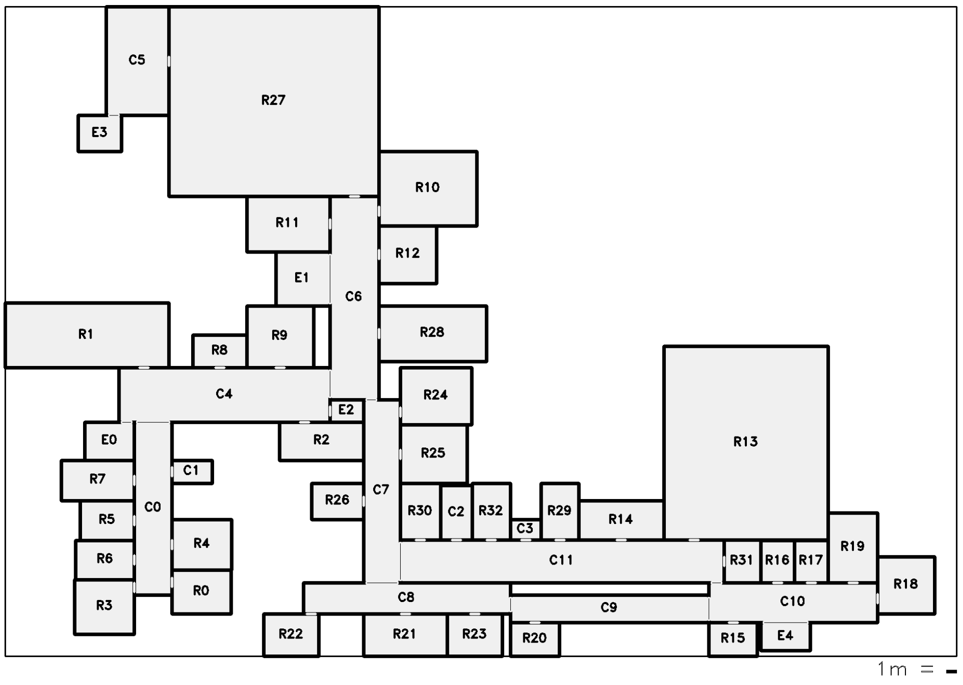
We perform these experiments with a simulated robot, utilizing a setup based on ROS and Stage. In this section we are going to describe this setup.

6.3.1 ROS

As introduced in the state of the art in Section 2.2, ROS (Robotic Operating System) [19] is an open-source framework employed in robotics that provides services like hardware abstraction, low-level device control, and standard implementations of commonly-used functionalities. ROS is based on packages that can be either written and built with the help of the provided tools and libraries, or directly retrieved through a dedicated package management system. ROS acts as an operative system, organizing the executions and



(a) Layout generated for the sub-problem



(b) Generated - 2473 m^2 - $\lambda_{cover} = 0.1$

Figure 6.9: School 2

the communications of the software components in use, even across different machines.

The execution of programs is modular and based on the concept of *node*, which is a process that performs computation. Each node implements a particular functionality, that can be both hardware or software related (e.g., controlling a sensor, localizing the robot, ...).

The exchange of information between nodes is handled directly by ROS. In particular, the nodes communicate utilizing *messages*, which are data structures that contain typed fields. These fields can be primitive types (e.g., integer, floating point, Boolean, ...), arrays, or nested structures.

These messages are routed between nodes using a publisher/subscriber protocol based on topics. A *topic* is the name reserved by a node to publish messages. In this way, the messages are not exchanged directly from a node to another, but are rather published on a topic and available to anyone that is subscribed to it. This allows multiple nodes to send or to receive the same messages, while keeping a decentralized architecture since the nodes aren't directly aware of the execution of other nodes.

An alternative communication paradigm offered by ROS is the *service*. Services are based on a request/reply interaction between two nodes. In particular, the format of the communication is based on a pair of different data structures, one for the request and one for the response. With this system, the communication is directly between a single provider of the service and an individual requester, which sends a request and waits for the response.

6.3.2 Stage

Since the focus of our experiments is the planning and the exploration of the environment, we decided to utilize Stage since we don't require high fidelity simulations. Stage [17], as already introduced in Section 2.2, is a lightweight 2D simulator that model a single or multiple robots in a virtual world. It models the uncertainty affecting the dynamics and the perception of robots, but not their physical interaction with the environment. We use it to simulate a generic differential drive robot equipped with a frontal laser range scanner sensor, with a field of view of 270° and a range of 30 m.

Stage accepts the description of the setting, which includes the bitmap image representing the environment and the parameters required to model the robot, as a *world* file. The input floor plan is represented as a black and white image, specifying with white pixels the free space and with black ones the obstacles, which in our case are only the walls of the building layout.

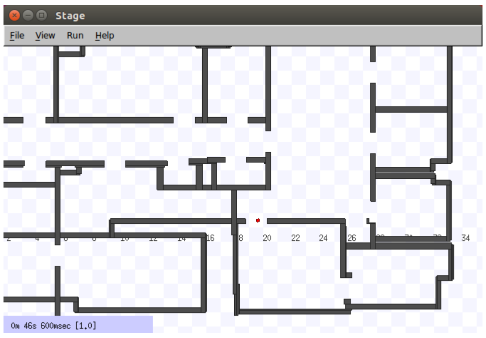


Figure 6.10: Snapshot of a Stage simulation

Stage, as specified in its documentation[1], models the odometry error in a simplistic way, by continuously adding a constant error defined randomly at the start of the simulation.

A snapshot of a Stage simulation is shown in Figure 6.10.

6.3.3 Exploration package

As stated in Section 2.1, the exploration task is based on navigating around the environment, collecting information in order to create a complete map. We utilize the standard explorer package [65] of ROS, which is based on coordinated multi-robot explorations but can also be used with a single robot. This package provides multiple frontier-based algorithms, with different exploration strategies available. A *frontier* is the boundary between the known and unknown regions of the map. An example of map partially explored is shown in Figure 6.11, with some frontiers indicated in red. The strategy that we employ sets as new goals points on the nearest frontiers, using as distance the length of the expected travel path of the robot. Once a goal is reached, a new one is generated until there are no frontiers left to explore.

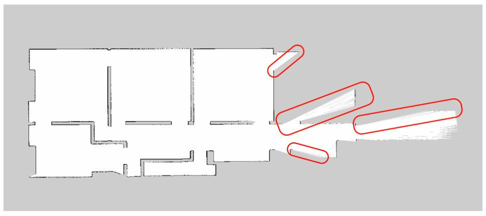


Figure 6.11: Example of frontiers between explored and unexplored regions of a map

[1]http://rtv.github.io/Stage/group__model__position.html

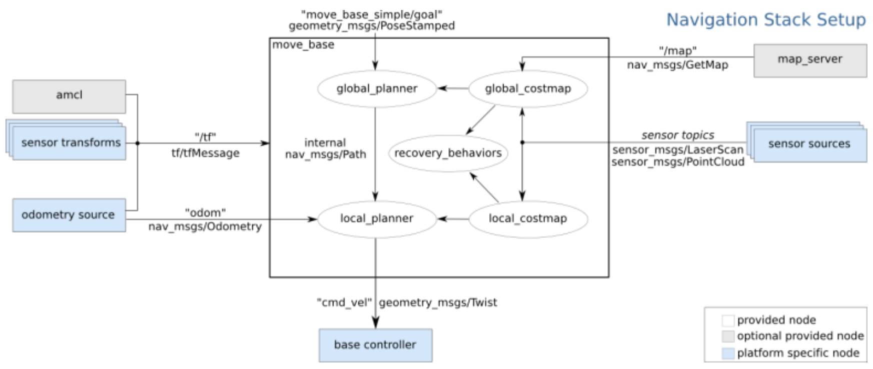


Figure 6.12: Architecture of the ROS Navigation stack

6.3.4 Navigation stack

To allow the simulated robot to reach the goals created by the explorer package, we utilize the ROS navigation stack [66]. Its structure is shown in Figure 6.12. The navigation stack covers all the functionalities required by the robot to navigate an environment, receiving as inputs the sensor data and the known map. In particular, the navigation stack is based around the move_base module, that coordinates the operations of the global and the local planners.

The *global planner* is the component that considering as input the position of the robot, the goal location, and the map, decides which path should be taken by the robot to reach the goal. This decision is based on the *costmap*, which is an occupancy grid map that assigns at each cell of the grid a cost. The costmap is utilized to enlarge the obstacles of the map with the size of the robot, so that the robot won't collide with them.

The computed path is then utilized by the *local planner*, which is similar to the global planner but operates on a local costmap, which is a restriction of the global costmap that considers only the immediate surroundings of the robot. The typical size of the local costmap is 6 m for both width and height. The goal of the local planner is to directly control the motors of the robot to follow the defined path calculated by the global planner.

6.3.5 SLAM package

Both the navigation and the explorer packages require a map to operate. As discussed in Section 2.1, the task of creating a map while keeping the robot localized is SLAM (Simultaneous Localization and Mapping). We use the ROS wrapper for the OpenSlam's GMapping implementation, which is based on [67] and [68].

GMapping is a particle filter SLAM algorithm that utilizes information from odometry and laser sensors to learn the grid map. As discussed in [7], a particle filter considers a finite number of hypotheses (the *particles*), which are utilized to store the candidate maps of the environment (as in the case of GMapping), the estimated pose, or both.

The odometry and the raw laser range scanner data are provided to GMapping directly by Stage.

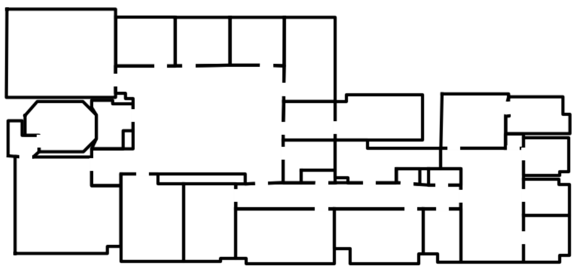


Figure 6.13: Map of Office 1

6.3.6 Map preparation

To use both the original and the generated layouts in Stage, we need to convert the XML file to a bitmap image. The system we employ draws each linesegment, which structure has been defined in Section 5.2, representing a wall as a line with a predefined width. The portals are considered all explicit and with the same size, which is taken from the *real_ distance* element. To draw them, the system just adds a white segment, corresponding to the size of the door, over the location of the portal linesegment. An example of bitmap image prepared with this method is shown in Figure 6.13.

6.4 Planning experiments

The first experiment we perform is to check if the travel distance between two rooms doesn't change in a significant way in the generated layouts with respect to the original environment. In particular, we consider for the paths only the rooms marked as *entrances* (E) in the input layout.

The methodology we adopt is to utilize the *makePlan* service of the global planner to retrieve the plan between each pair of rooms. This service allows to insert manually both the starting and the goal pose.

All the generated layouts utilized in this experiments are generated with $\lambda_{cover} = 0.1$

6.4.1 Office 1

The first layout we analyze is the same office presented in Section 6.2. This is a medium size building with five entrances. We utilized the three generated layout shown in Figure 6.6 and two other generated layouts, obtained by fixing the location of E1 in different points of the space.

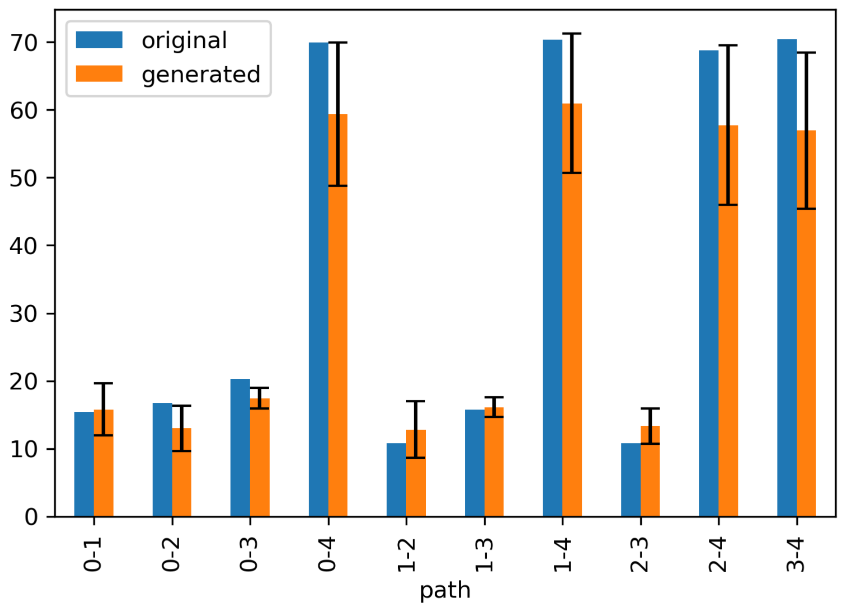
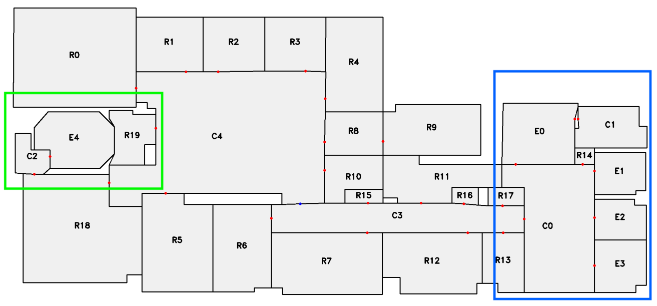


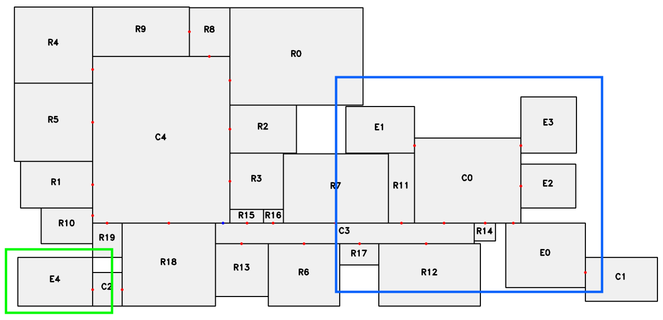
Figure 6.14: Path planning results for the Office 1 layout. The numbers indicate the path between two entrances (e.g., 0-1 indicates the path between the entrance E0 and E1)

In Figure 6.14 it is shown the data retrieved from the path planning. On the horizontal axis there are the various paths between the rooms, which are symmetric and so taken just once for each pair of rooms. On the vertical axis it is represented the path distance. In particular, the blue column corresponds to the original layout, while the orange one is the average between the five generated layouts, with the respective standard deviation.

It is possible to notice from both the data in Figure 6.14 and the layouts in Figure 6.15 that the first four entrances (E0, E1, E2, and E3) are in the same region, while the remaining one (E4) is on the other side of the building. This feature is present in both the original layout and in the generated ones.



(a) Original layout



(b) Generated layout

Figure 6.15: Comparison of entrances between the original Office 1 layout and a generated one

6.4.2 School 2

The second layout we consider is the second school presented in Section 6.2. This is a large scale layout. As in the previous case, we utilize the technique of fixing rooms to generate multiple variants with the same input layout. Like the last example, also in this layout there are five entrances.

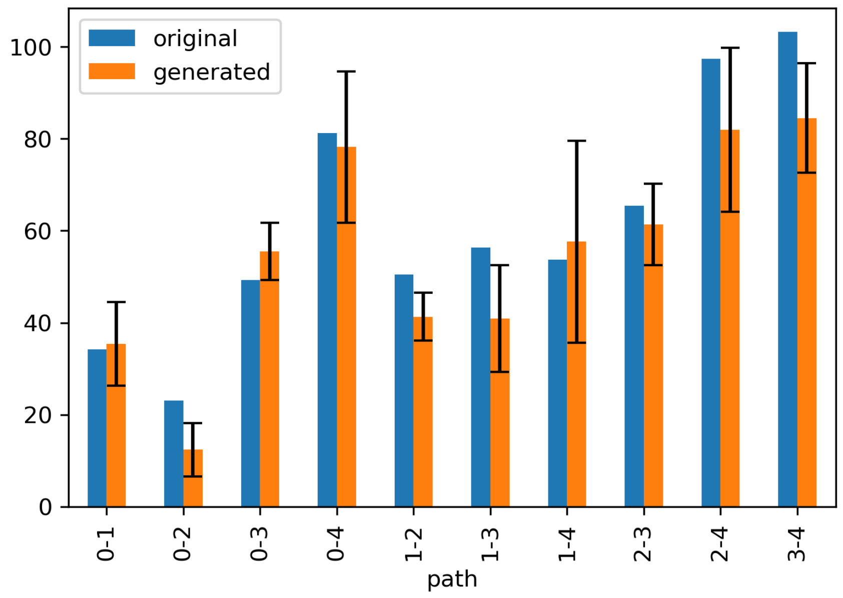
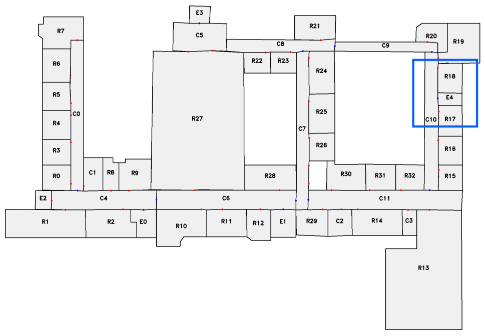


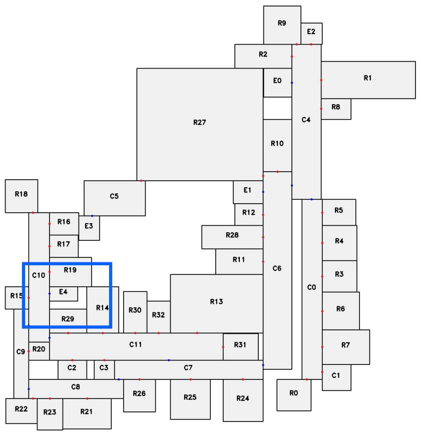
Figure 6.16: Path planning results for the School 2 layout

In Figure 6.16 it is shown the data retrieved for this environment. It emerges, from both the graph and the layouts in Figure 6.17, that the entrances are more uniformly distributed in this environment than in the last one. However, there are large deviations in values of the generated layouts, especially in the paths that connect E4 to the other entrances. In Figure 6.17a it is possible to see that the entrance E4 is on the right side of the original building, while in two reported generated layouts there are two opposite situations. In fact, in Figure 6.17b the room is very far from all the other entrances, while in the other example in Figure 6.17c, E4 is much closer to them.

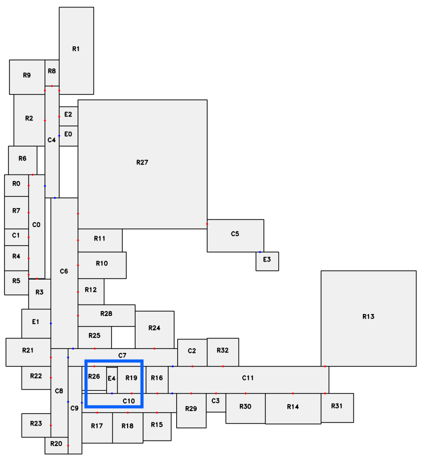
Still, on average we can see that the generated layouts are similar to the original one, despite the large scale of the specific layout that could have caused much larger oscillations in the values.



(a) Original layout

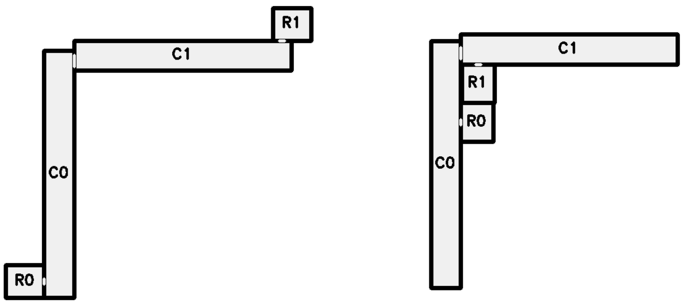


(b) First generated layout



(c) Second generated layout

Figure 6.17: Comparison of E4 between the original School 2 layout and two generated ones



(a) (b)

Figure 6.18: Comparison between two situations from the same topology structure

6.4.3 Planning results considerations

Given the data obtained above and from other similar layouts, that we omitted for brevity, we can say that the generated layouts maintain, on average, a similarity in the distance between rooms. This is a positive result, since in the generation method we only account for the topology graph, disregarding the problem of keeping the rooms evenly distanced in the resulting layouts.

In particular, we expect some degree of deviation in the generated layouts with this metric, since the adjacencies we consider in the generation allows swaps of rooms. An extreme example of this is shown in Figure 6.18, where the same topology graph allows both cases. This is a limit of utilizing the adjacency relations between the rooms as a graph. However, in practical scenarios, we expect on average a certain degree of consistency in the rooms distances.

6.5 Exploration experiments

The second experiment we perform to validate the similarity of the original layouts and the generated ones is based on the exploration performance in such environments.

The metrics we consider are the exploration time and the traveled distance of the robot required to complete the exploration process. To obtain this data, we perform ten exploration runs on each environment. If the robot gets stuck during the exploration, we repeat that run.

For this experiment, we consider a total of twelve environments, five offices and seven schools. For each one of them, we run an exploration in the original building layout and in two generated layouts obtained with different parameters. In particular, we utilize a layout generated with $\lambda_{cover} = 0.1$ and one with $\lambda_{cover} = 0.2$ for each environment.

In each exploration run, the starting point is always the same, and corresponds to the same room (an entrance) across all the three variants.

6.5.1 Exploration data overview

We summarize the results we obtained from each run of exploration with a t-SNE, which is a machine learning technique proposed by Maaten et al. in [69] to reduce the dimensionality of data for visualization purposes. We utilize this method to represent multiple variables in a single two-dimensional graphic. In addition, with this technique it is possible to evidence the presence of clusters in the data. To produce the graph in Figure 6.19, we combined the exploration time, the traveled path, and the area of the layout.

In the graph, each marker color represents a different layout, while the marker shape defines the variant in the following way: the crosses are the original layouts, while the circles and the squares are the layouts generated with the values of λ_{cover} as reported in the legend (indicated respectively as `cover01` and `cover02` for $\lambda_{cover} = 0.1$ and $\lambda_{cover} = 0.2$).

Most environments are well clustered. This means that there are no noticeable differences between the exploration performance in the original layouts and in their generated versions. If a single run of a layout is far from the cluster of the same environment, like for example the isolated point of School 3 that is positioned near the Office 3 cluster, it is reasonable to conclude that there was an issue during the exploration of that specific run. In fact, this School 3 run presented anomalies.

However, the layout Office 5, which is colored in black in the figure, is divided in two different clusters. One for the runs of the original layout and the other for the explorations of the generated layouts. We now analyze in detail this particular layout, comparing it to others in order to identify the reason of this strange behavior.

6.5.2 Exploration data analysis

Since Office 5, reported in Figure 6.20, is a rather small office, we decided to compare it to other similar sized layouts. In this case, we consider Office 4 and School 5, which can be seen in Figure 6.21

From these pictures, it is possible to notice that Office 5 is composed of many non-rectangular rooms, especially regarding the corridors structure.

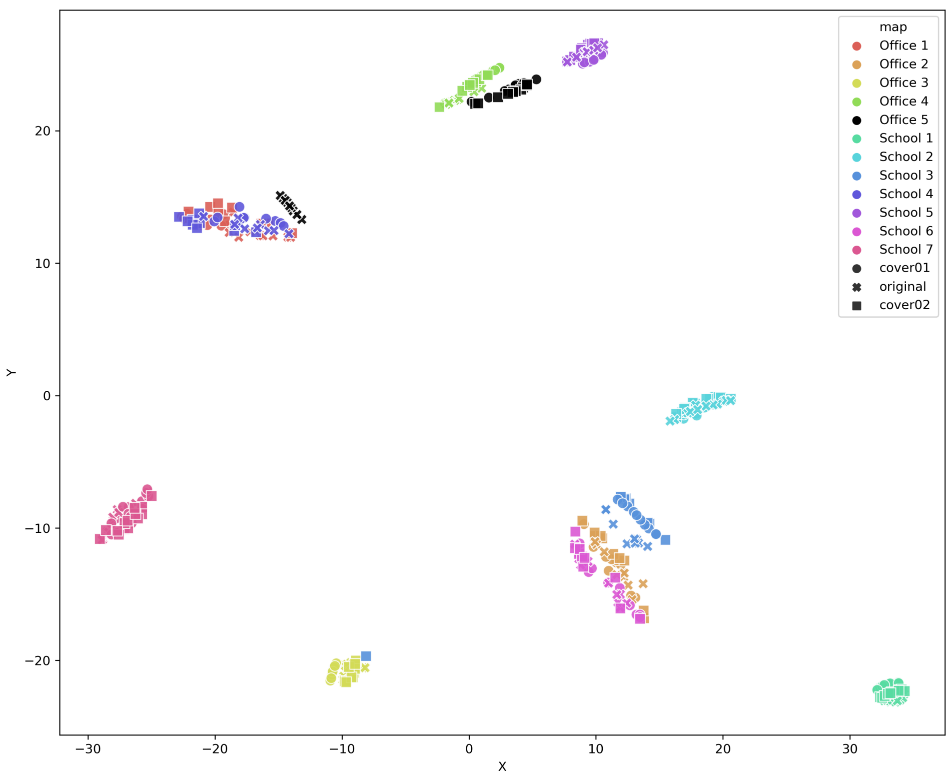


Figure 6.19: t-SNE that summarize the exploration data results

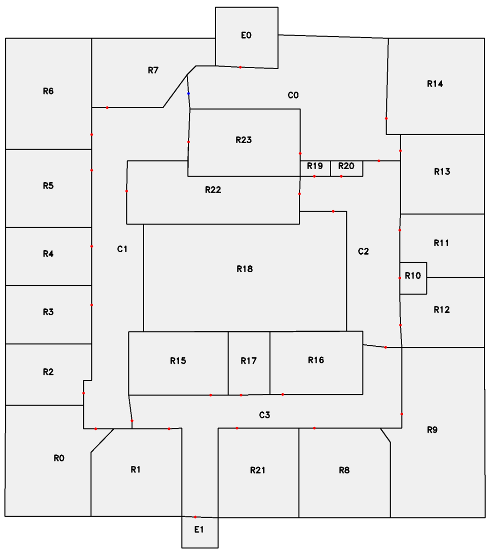
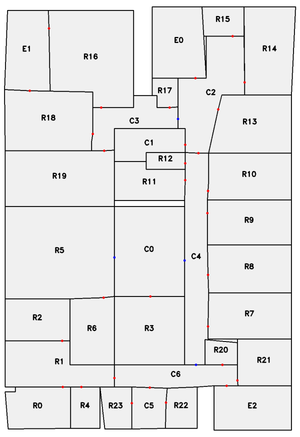
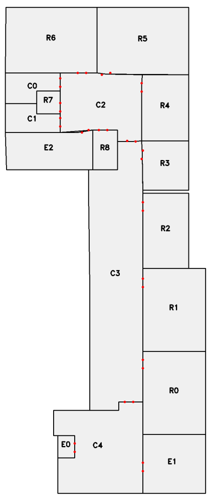


Figure 6.20: Original layout of Office 5

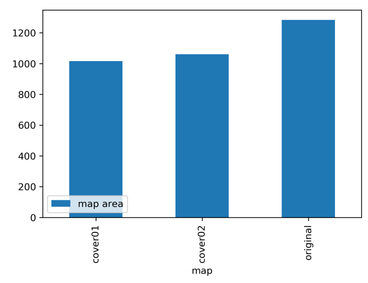


(a) Office 4

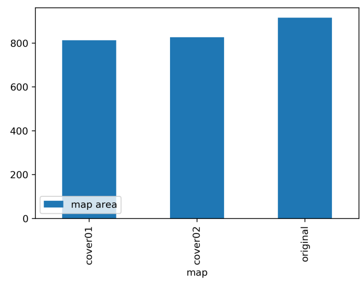


(b) School 5

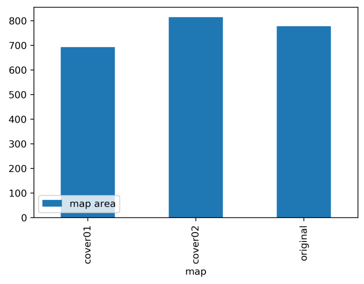
Figure 6.21: Original layouts of Office 4 and School 5



(a) Office 5



(b) Office 4



(c) School 5

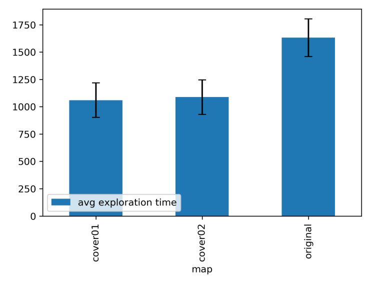
Figure 6.22: Comparison of map areas of Office 5, Office 4, and School 5

On the contrary, Office 4 and School 5 are mainly composed of rectangular rooms.

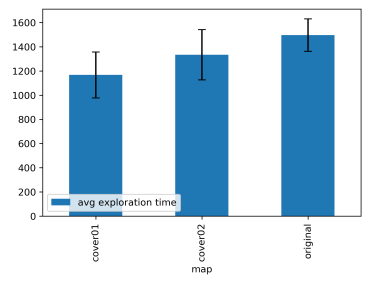
In Figure 6.22 the areas of the original and generated versions of these layouts are reported. Except for School 5, where for $\lambda_{cover} = 0.2$ a layout larger than the original has been created, the areas are similar across the layouts.

In Figure 6.23 the exploration times of these three layouts in all their variants are shown. In Figure 6.23c, the exploration times of School 5 are extremely similar between the original layout and the generated ones. Office 4 is slightly worse but still consistent, while in Office 5 the exploration performance is significantly different.

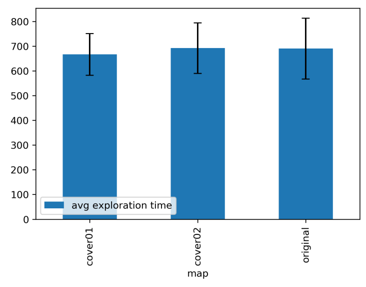
This pattern is present also considering the traveled path of the robot reported in Figure 6.24. In Figure 6.24c, it is interesting that the average traveled path for the $\lambda_{cover} = 0.1$ variant is higher than the others, despite being the smaller layout and the one with the fastest exploration. This can be caused by a particular initial position of the robot in this generated layout, since all the exploration runs on the same layout starts in the same room, which can be in favorable or unfavorable location with respect to the exploration strategy employed. As discussed previously in Section 6.4.3, it is not possible with our topology graph to provide the specific position of a



(a) Office 5

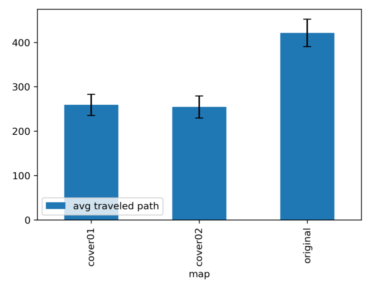


(b) Office 4

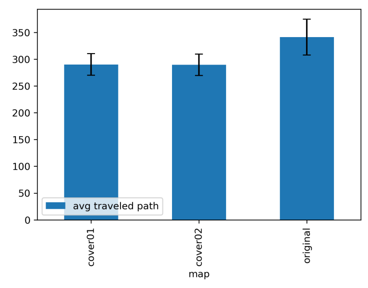


(c) School 5

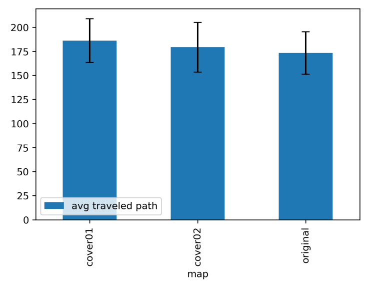
Figure 6.23: Comparison of exploration times of Office 5, Office 4, and School 5



(a) Office 5

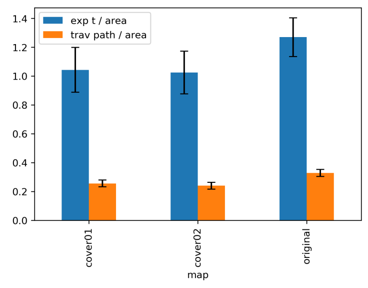


(b) Office 4

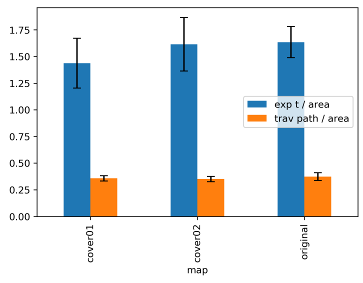


(c) School 5

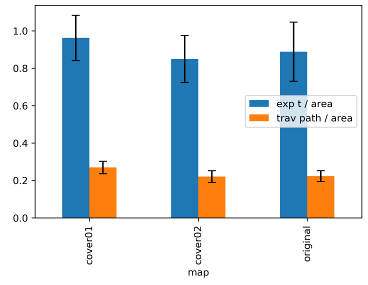
Figure 6.24: Comparison of traveled distances of the explorations of Office 5, Office 4, and School 5



(a) Office 5



(b) Office 4



(c) School 5

Figure 6.25: Comparison of exploration times and traveled distances divided by the area of the layout for Office 5, Office 4, and School 5

room in respect to the others.

One of the problems with the two metrics we utilized is that they depend on the total area of the specific layout. In fact, if a layout is larger, the robot would take a longer time to navigate it completely, with also an increased total traveled distance.

While we utilized the same values of the λ_{cover} parameter for all the layouts, it is possible to control the area of the generated layout by changing the weights of the objective function, as discussed in Section 6.1.1. This allows to generate layouts with a much similar area than the ones we considered in this experiment.

However, this process must be performed manually and can require a long time to find the right value for all the involved parameters. Instead, to check the dependency of the exploration time and of the traveled distance on the layout area, we just perform a division to normalize the two metrics on the area. These results can be seen in Figure 6.25.

It is interesting to notice in particular the values of Office 4, pictured in Figure 6.25b. With this normalization, the exploration time of the $\lambda_{cover} = 0.2$ variant becomes very similar to the one of the original layout. This means that this generated layout is similar to the original one from an exploration point of view, and that the difference in the exploration performance in Figure 6.23 and Figure 6.24 depends on the area of the layout. We expect to obtain similar performance results in this environment with an extensive tuning of the cover parameters to increase the area of these generated layouts.

6.5.3 Exploration results considerations

Considering the data acquired in this experiment, we state that our generation method can provide layouts with exploration performances similar to the original ones. This means that we can successfully generate multiple realistic building layouts that behave like the real world ones that we utilize as input in our generation process. However, we identified two issues that can lead to a significant difference in exploration time and length of traveled path.

The first is the presence of non-rectangular rooms in the original layouts. This can be an issue both for the generation itself, as discussed in Section 6.1.2, and for the exploration. In fact, we are limited with our method to generate rectangular rooms, which are easier to explore by the robot. The reason is that a non-rectangular room can require multiple observations (and thus multiple exploration goals) to be completely mapped, since a complex room shape can limit the line of sight of the laser range scanner of the robot. An example of this situation is shown in Figure 6.26, where the the robot represented by the blue arrow has just entered the room. However, the onboard laser range scanner is limited to view the green portion of the room. The red dashed line is the line of sight limit caused by the structure of the room. This leads to the creation of additional frontiers for the room on the boundary between the green and the red regions. In this way, a single room can take much longer to be fully explored, extending the entire exploration process in both time and distance required.

The second issue is the variability of the connections between the rooms. In fact, the same problem of the swap of the rooms introduced in Section 6.4.3 can affect both the starting position of the explorations and the corridors configurations of the generated layout, reducing, or increasing, the original distance and time required to reach another part of the building. Since we utilized an exploration strategy based on the nearest frontiers, this issue was probably minimized in our exploration runs. However, like in the previous case of path planning, we expect these differences to be, on a global scale, on average negligible.

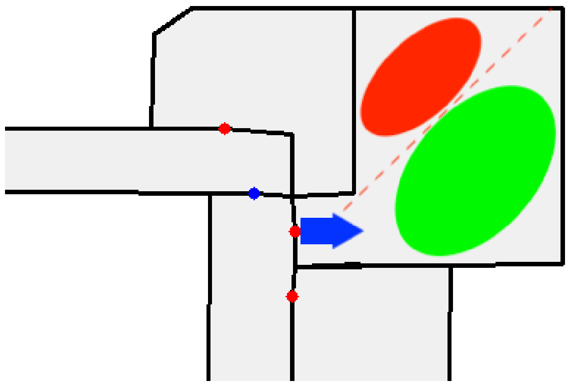


Figure 6.26: Example of line of sight limitation of the onboard laser range scanner in a non-rectangular room

6.6 Summary

In this chapter, we discussed at first the issues of our generation method. After that, we presented some examples of generated layouts. Next, we described the setup of the experiments we perform, their results, and some observations on them, concluding that the major issue in both the generation and the explorations experiment is the presence of non-rectangular shaped rooms in the original layout.

Chapter 7

Conclusions and future developments

In this thesis, we proposed a procedural generation method to produce a large quantity of realistic layouts of buildings. In this way, we can utilize environments in robot simulations that are representative of real world ones.

We started reviewing the state of the art of procedural layout generation techniques. In particular, we focused on the scalability of the methods for procedural generation to handle medium and large scale environments and how on well they perform while respecting the features specified in input.

We also discussed the concept of realism when applied to layouts. The definition we use is based on the similarity of the structure of the original layouts to the generated ones, focusing in particular on the topology graph and the sizes of the rooms.

We utilized these concepts to propose a procedural layout generation method that, given in input a building layout, preserves the original topology graph while minimizing the deviation of the sizes of the rooms from their original values in the original layout. To perform this generation, we based our method on a MIQP optimization problem. This technique allows the definition of specific hard constraints, which can be utilized to enforce both the adjacencies defined in the original topology graph and that the sizes of the rooms don't deviate too much from the original ones, while guaranteeing scalability to medium and large layouts.

To validate the realism of the layouts generated with our method, we performed two types of experiments, where we compared the performance of a robot performing a task in a generated environment with the performance obtained by the same robot in the real world building counterpart. We first considered the path planning between pairs of rooms, focusing on how the length of the paths planned vary across the original layout and the generated ones. We found that the lengths of planned paths are similar on average (over multiple generated layouts) to those measured in the original layout, even if we don't consider the distances between rooms in the generation method but only the topology graph. The second experiment we performed is the exploration of the environments with a simulated robot. In this case, we focused on evaluating if the exploration time and the length of the traveled path required to complete the exploration process is similar between real world and simulated environments. Across the tested layouts, we found that most of them share similar performance with the original layout, meaning that our method successfully generated realistic building layouts. However, in some isolated cases, our method generates layouts that don't perform like their original versions. We identified the cause of this problem, which is caused by the complex structure of some rooms of the original layouts. In fact, our method can only generate rectangular rooms, which are easier to explore by the robot than non-rectangular shaped ones.

In conclusion, our procedural generation method performs well in generating realistic layouts, handling also medium and large scale ones, even if there are some limitations that can affect both the generation process and the experiments performed on the generated layouts. In fact, as we discussed in Section 6.1.2, non-rectangular shaped rooms and cycles in the input graph can increase the computational complexity of the generation, and even preventing it in some extreme cases.

We now list some possible future developments of our work.

- **Combined rooms**: to allow the generation of non-rectangular shaped rooms, we can introduce a technique similar to the one proposed in [6] to decompose the rooms in multiple rectangles. While this approach is theoretically possible, it could increase significantly the computation required to find a solution. Since we consider mainly medium and large scale layouts, the effect of few non-rectangular rooms is typically negligible on average, but it can be a major issue with smaller layouts.

- **Modular regions**: to reduce the complexity of the optimization problem, we can divide the layout in regions, which will be generated independently and joined afterwards. An approach like this can be seen as an enhancement of the method utilized to generate large environments with the hierarchical framework in [6], since in our case the sub-problems would be automatically generated. In addition, the connection of the various regions can be done utilizing constraints like the connection constraint, defined in Section 5.5.1, to impose the positions of the rooms that connect them.

- **Detailed portals**: at the moment, the portals are only characterized by their type (explicit or implicit). However, it is possible to improve their modeling in various ways. For example, the generation method already handles variable door sizes for each portal, but this value is not used in the XML parser. Another possible addition is to consider the presence of multiple doors between two rooms, as with large rooms that need to allow the flow of large crowds (e.g., conference rooms, cafeterias, ...).

- **Boundary regularization**: it is possible to introduce a post-processing phase to regularize the external shape of the layout, which now can be irregular like we saw in Figure 6.7b. This procedure can consist on the alignment of the walls of the external rooms. In this case, we also need to limit the maximum deformation allowed, so that the resulting sizes of the rooms wouldn't deviate too much from their original specifications.

- **3D generation**: it is possible to extend our system to produce a 3D model of the generated environments. In addition, for more realistic simulations, we can also integrate other generation methods to generate furniture and textures. In this way, it is possible to use our 2D layouts as starting point to recreate some photorealistic 3D environments.

Bibliography

[1] F. Amigoni, M. Luperto, and V. Schiaffonati, "Toward generalization of experimental results for autonomous robots," *Robotics and Autonomous Systems*, vol. 90, pp. 4–14, 2017.

[2] P. Merrell, E. Schkufza, and V. Koltun, "Computer-generated residential building layouts," *ACM Transactions on Graphics (TOG)*, vol. 29, no. 6, p. 181, 2010.

[3] J. F. Rosser, G. Smith, and J. G. Morley, "Data-driven estimation of building interior plans," *International Journal of Geographical Information Science*, vol. 31, no. 8, pp. 1652–1674, 2017.

[4] E. Rodrigues, A. R. Gaspar, and Á. Gomes, "An evolutionary strategy enhanced with a local search technique for the space allocation problem in architecture, part 1: Methodology," *Computer-Aided Design*, vol. 45, no. 5, pp. 887–897, 2013.

[5] Z. Guo and B. Li, "Evolutionary approach for spatial architecture layout design enhanced by an agent-based topology finding system," *Frontiers of Architectural Research*, vol. 6, no. 1, pp. 53–62, 2017.

[6] W. Wu, L. Fan, L. Liu, and P. Wonka, "MIQP-based layout design for building interiors," in *Proceedings of Computer Graphics Forum*, vol. 37, pp. 511–521, Wiley Online Library, 2018.

[7] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics.* The MIT Press, 2005.

[8] H. Durrant-Whyte and T. Bailey, "Simultaneous localization and mapping: part I," *IEEE robotics & automation magazine*, vol. 13, no. 2, pp. 99–110, 2006.

[9] B. Yamauchi, "A frontier-based approach for autonomous exploration," in *Proceedings of the 1997 IEEE International Symposium on Compu-*

tational Intelligence in Robotics and Automation, pp. 146–151, IEEE Computer Society, 1997.

[10] D. Holz, N. Basilico, F. Amigoni, and S. Behnke, “Evaluating the efficiency of frontier-based exploration strategies,” in *Proceedings of ISR 2010 (41st International Symposium on Robotics) and ROBOTIK 2010 (6th German Conference on Robotics)*, pp. 1–8, VDE, 2010.

[11] F. Amigoni and V. Caglioti, “An information-based exploration strategy for environment mapping with mobile robots,” *Robotics and Autonomous Systems*, vol. 58, pp. 684–699, 2010.

[12] A. Howard and N. Roy, “The robotics data set repository (radish),” 2003. http://radish.sourceforge.net/.

[13] A. Bonarini, W. Burgard, G. Fontana, M. Matteucci, D. G. Sorrenti, and J. D. Tardos, “Rawseeds: Robotics advancement through web-publishing of sensorial and elaborated extensive data sets,” in *Proceedings of IROS’06 Workshop on Benchmarks in Robotics Research*, 2006.

[14] S. Ceriani, G. Fontana, A. Giusti, D. Marzorati, M. Matteucci, D. Migliore, D. Rizzi, D. G. Sorrenti, and P. Taddei, “Rawseeds ground truth collection systems for indoor self-localization and mapping,” *Autonomous Robots*, vol. 27, no. 4, pp. 353–371, 2009.

[15] E. Kolve, R. Mottaghi, D. Gordon, Y. Zhu, A. Gupta, and A. Farhadi, “AI2-THOR: An interactive 3D environment for visual AI,” *ArXiv*, vol. abs/1712.05474, 2017.

[16] I. Armeni, S. Sax, A. R. Zamir, and S. Savarese, “Joint 2D-3D-semantic data for indoor scene understanding,” *arXiv preprint arXiv:1702.01105*, 2017.

[17] R. Vaughan, “Massively multi-robot simulation in stage,” *Swarm Intelligence*, vol. 2, pp. 189–208, 12 2008.

[18] N. Koenig and A. Howard, “Design and use paradigms for Gazebo, an open-source multi-robot simulator,” in *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, (Sendai, Japan), pp. 2149–2154, Sep 2004.

[19] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, “ROS: an open-source Robot Operating System,” in *Proceedings of ICRA Workshop on Open Source Software*, 2009.

[20] E. Whiting, J. Battat, and S. Teller, "Generating a topological model of multi-building environments from floorplans," in *Proceedings of CAAD (Computer-Aided Architectural Design) Futures 2007*, pp. 115–28, July 2007.

[21] A. Aydemir, P. Jensfelt, and J. Folkesson, "What can we learn from 38,000 rooms? Reasoning about unexplored space in indoor environments," in *Proceedings of 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2012.

[22] T. Li, D. Ho, C. Li, D. Zhu, C. Wang, and M. Q.-H. Meng, "HouseExpo: A large-scale 2D indoor layout dataset for learning-based algorithms on mobile robots," *arXiv preprint arXiv:1903.09845*, 2019.

[23] E. Whiting, J. Battat, and S. Teller, "Topology of urban environments," in *Proceedings to Computer-Aided Architectural Design Futures (CAAD-Futures) 2007*, pp. 114–128, Springer, 2007.

[24] F. Amigoni, V. Castelli, and M. Luperto, "Improving repeatability of experiments by automatic evaluation of SLAM algorithms," in *Proceedings of 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 7237–7243, IEEE, 2018.

[25] R. Kümmerle, B. Steder, C. Dornhege, M. Ruhnke, G. Grisetti, C. Stachniss, and A. Kleiner, "On measuring the accuracy of SLAM algorithms," *Autonomous Robots*, vol. 27, no. 4, p. 387, 2009.

[26] H. Chen, X. Zhao, J. Luo, Z. Yang, Z. Zhao, H. Wan, X. Ye, G. Weng, Z. He, T. Dong, and S. Schwertfeger, "Towards generation and evaluation of comprehensive mapping robot datasets," *arXiv preprint arXiv:1905.09483*, 2019.

[27] D. Lobos and D. Donath, "The problem of space layout in architecture: A survey and reflections," *Arquiteturarevista*, vol. 6, no. 2, pp. 136–161, 2010.

[28] A. Eiben and M. Schoenauer, "Evolutionary computing," *Information Processing Letters*, vol. 82, no. 1, pp. 1 – 6, 2002.

[29] K. Dutta and S. Sarthak, "Architectural space planning using evolutionary computing approaches: a review," *Artificial Intelligence Review*, vol. 36, no. 4, p. 311, 2011.

[30] V. Calixto and G. Celani, "A literature review for space planning optimization using an evolutionary algorithm approach: 1992-2014," in *Proceedings of XIX Congresso da Sociedade Ibero-americana de Gráfica Digital 2015*, CUMINCAD, 2015.

[31] J. McCall, "Genetic algorithms for modelling and optimisation," *Journal of Computational and Applied Mathematics*, vol. 184, no. 1, pp. 205 – 222, 2005. Special Issue on Mathematics Applied to Immunology.

[32] M. Verma and M. K. Thakur, "Architectural space planning using genetic algorithms," in *Proceedings of The 2nd International Conference on Computer and Automation Engineering (ICCAE)*, vol. 2, pp. 268–275, IEEE, 2010.

[33] R. W. Flack and B. J. Ross, "Evolution of architectural floor plans," in *Proceedings of European Conference on the Applications of Evolutionary Computation*, pp. 313–322, Springer, 2011.

[34] W. F. Robert, *Evolution of Architectural Floor Plans.* PhD thesis, Faculty of Computer Science, Brock University, 2010.

[35] N. Sönmez, "Architectural layout evolution through similarity-based evaluation," *International Journal of Architectural Computing*, vol. 13, no. 3-4, pp. 271–297, 2015.

[36] A. Bahrehmand, T. Batard, R. Marques, A. Evans, and J. Blat, "Optimizing layout using spatial quality metrics and user preferences," *Graphical Models*, vol. 93, pp. 25–38, 2017.

[37] E. Rodrigues, A. R. Gaspar, and Á. Gomes, "An evolutionary strategy enhanced with a local search technique for the space allocation problem in architecture, part 2: Validation and performance tests," *Computer-Aided Design*, vol. 45, no. 5, pp. 898–910, 2013.

[38] E. Rodrigues, A. R. Gaspar, and Á. Gomes, "An approach to the multi-level space allocation problem in architecture using a hybrid evolutionary technique," *Automation in Construction*, vol. 35, pp. 482–498, 2013.

[39] E. Rodrigues, *Automated floor plan design: generation, simulation, and optimization.* PhD thesis, 2014.

[40] M. Bruls, K. Huizing, and J. J. Van Wijk, "Squarified treemaps," in *Data Visualization 2000*, pp. 33–42, Springer, 2000.

[41] F. Marson and S. Musse, "Automatic real-time generation of floor plans based on squarified treemaps algorithm," *International Journal of Computer Games Technology*, vol. 2010, 2010.

[42] B. J. De Vlugt, "Modern optimization algorithms and applications: Architectural layout generation and parallel linear programming," 2015.

[43] W. J. Mitchell, *The Logic of Architecture: Design, Computation, and Cognition.* Cambridge, MA, USA: MIT Press, 1st ed., 1990.

[44] H. Hua, "A bi-directional procedural model for architectural design," in *Proceedings of Computer graphics forum*, vol. 36, pp. 219–231, Wiley Online Library, 2017.

[45] L. Leblanc, J. Houle, and P. Poulin, "Component-based modeling of complete buildings," in *Proceedings of Graphics Interface 2011*, GI '11, pp. 87–94, Canadian Human-Computer Communications Society, 2011.

[46] M. A. Rosenman, "The generation of form using an evolutionary approach," in *Evolutionary Algorithms in Engineering Applications*, pp. 69–85, Springer, 1997.

[47] T. Schnier and J. S. Gero, "Learning genetic representations as alternative to hand-coded shape grammars," in *Artificial Intelligence in Design'96*, pp. 39–57, Springer, 1996.

[48] C. M. Bishop, *Pattern recognition and machine learning.* Springer Science+ Business Media, 2006.

[49] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery, *Numerical Recipes 3rd Edition: The Art of Scientific Computing.* Cambridge University Press, 2007.

[50] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing.," *Science*, vol. 220 4598, pp. 671–80, 1983.

[51] T. Feng, L.-F. Yu, S.-K. Yeung, K. Yin, and K. Zhou, "Crowd-driven mid-scale layout design.," *ACM Transactions on Graphics (TOG)*, vol. 35, no. 4, pp. 132–1, 2016.

[52] A. Liaw and M. Wiener, "Classification and regression by randomforest," *R news*, vol. 2, no. 3, pp. 18–22, 2002.

[53] J. Martin, "Algorithmic beauty of buildings methods for procedural building generation," *Computer Science Honors Theses*, 2005.

[54] R. Rajapaksha, K. Jayawardena, and S. Fernando, "A knowledge base approach for efficient home floor plan generation applying polygon based representation," in *Proceedings of 8th International Conference on Ubi-Media Computing (UMEDIA)*, pp. 324–329, IEEE, 2015.

[55] H. Liu, Y.-L. Yang, S. Alhalawani, and N. J. Mitra, "Constraint-aware interior layout exploration for pre-cast concrete-based buildings," *The Visual Computer*, vol. 29, no. 6-8, pp. 663–673, 2013.

[56] X.-Y. Wang, Y. Yang, and K. Zhang, "Customization and generation of floor plans based on graph transformations," *Automation in Construction*, vol. 94, pp. 405–416, 2018.

[57] M. Luperto and F. Amigoni, "Predicting the global structure of indoor environments: A constructive machine learning approach," *Autonomous Robots*, vol. 43, no. 4, pp. 813–835, 2019.

[58] M. Luperto and F. Amigoni, "Extracting structure of buildings using layout reconstruction," in *Proceedings of International Conference on Intelligent Autonomous Systems*, pp. 652–667, Springer, 2018.

[59] B. C. Eaves, "On quadratic programming," *Management Science*, vol. 17, no. 11, pp. 698–711, 1971.

[60] R. Fletcher, *Practical Methods of Optimization; (2nd Ed.).* New York, NY, USA: Wiley-Interscience, 1987.

[61] R. Fletcher and S. Leyffer, "Numerical experience with lower bounds for MIQP branch-and-bound," *SIAM Journal on Optimization*, vol. 8, no. 2, pp. 604–616, 1998.

[62] L. Gurobi Optimization, "Gurobi optimizer reference manual," http://www.gurobi.com.

[63] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using NetworkX," in *Proceedings of the 7th Python in Science Conference* (G. Varoquaux, T. Vaught, and J. Millman, eds.), (Pasadena, CA USA), pp. 11 – 15, 2008.

[64] T. Kavitha, K. Mehlhorn, D. Michail, and K. E. Paluch, "An O(m^2n) algorithm for minimum cycle basis of graphs," *Algorithmica*, vol. 52, no. 3, pp. 333–349, 2008.

[65] T. Andre, D. Neuhold, and C. Bettstetter, "Coordinated multi-robot exploration: Out of the box packages for ROS," in *Proceedings of IEEE GLOBECOM WiUAV Workshop*, Dec. 2014.

[66] E. Marder-Eppstein, E. Berger, T. Foote, B. Gerkey, and K. Konolige, "The office marathon: Robust navigation in an indoor office environment," in *Proceedings of International Conference on Robotics and Automation*, 2010.

[67] G. Grisetti, C. Stachniss, and W. Burgard, "Improved techniques for grid mapping with rao-blackwellized particle filters," *IEEE transactions on Robotics*, vol. 23, no. 1, p. 34, 2007.

[68] G. Grisetti, C. Stachniss, and W. Burgard, "Improving grid-based SLAM with rao-blackwellized particle filters by adaptive proposals and selective resampling," in *Proceedings of the 2005 IEEE international conference on robotics and automation*, pp. 2432–2437, IEEE, 2005.

[69] L. v. d. Maaten and G. Hinton, "Visualizing data using t-SNE," *Journal of machine learning research*, vol. 9, no. Nov, pp. 2579–2605, 2008.