Automation in Construction 123 (2021) 103491

Contents lists available at ScienceDirect

Automation in Construction

journal homepage: www.elsevier.com/locate/autcon

Review

Floor plan generation through a mixed constraint programming-genetic optimization approach

Graziella Laignel, Nicolas Pozin, Xavier Geffrier, Loukas Delevaux, Florian Brun, Bastien Dolla *

HABX, Ville pour tous, Paris, France

ARTICLE INFO

Keywords:
Automatic floor plan generation
Constraint programming
Genetic optimization
Space planning
Computational design

ABSTRACT

We present a novel approach for automatic apartment layout generation. Given a polygonal apartment envelope and a list of rooms with associated area, our so-called Optimizer algorithm generates several floor plans aiming at both architectural and functional constraints.

To do so, Optimizer discretizes the floor space into a grid according to architectural constraints and reduces the problem to a cell assignment which is solved through a coupled constraint programming - genetic optimization approach.

Obtained results demonstrate the feasibility of our approach, customized plans are architecturally and functionally valid, they are mostly generated in about 1 min.

1. Introduction

Designing an apartment layout can be a long and tedious task, especially when it comes to large surface areas or layouts subjected to complex constraints. As a result, in most real estate development projects, customers are proposed predefined floor plans designed mainly to comply with regulations that may not match their exact needs. Architect personalization would be costly and time-consuming.

Automatic floor plan generation is a key approach to address those limitations. This has been a research area since the 1970s [1]. Several mathematical techniques were employed to draw floor plans algorithmically [2]. Some authors used a constraint programming approach. In this paradigm, the plan is described by a set of variables that shall respect some imperative constraints. In [3–9] the apartment space is divided into rectangular rooms defined by a set of variables (e.g., positions and dimensions) to which various constraints are applied (e.g non-overlapping, adjacency and target area). Although those approaches may generate valid layouts, they are here limited to the over-simplistic case of rectangular rooms within rectangular apartment envelopes. In [10] authors propose and mixed integer quadratic approach leading to designs in more complex enveloppes, but only rectilinear polygonal room shapes areconsidered.

Some authors propose procedural techniques to draw floor plans. In [11], a growth-based algorithm is used to expand rooms in the plan but the apartment envelope can evolve which makes the problem much less constrained than ours. In [12] the plan is decomposed in a grid and a stochastic growth algorithm allocates cells to specified rooms, but since the grid is cartesian, only apartments with rectilinear walls can be obtained (mixed integer approach shall rather be linked to the constraint programming part, see above).

Shape grammars have also been used for procedural layout generation. The plan is formed through the composition of a set of hard-coded rules. Depending on the defined grammar, the technique can be limited to a given architectural style [1] or to limited types of envelope structures [13,14].

Some works treat the problem through an optimization formalism. In [15], floor plans are obtained as the result of a cell allocation problem: the plan is decomposed into a grid, and resulting cells are allocated through an optimization process that ensures the floor plan validity. In [16–18] genetic algorithms are employed to organize a floor space but again results are illustrated on simple rectangular geometries. Non-rectilinear envelopes are a challenge most approaches seem to forego. We can however notice that in [19] some floor plans with non-rectilinear enveloppes are obtained. But it is quite difficult to conclude about the reliability of the method because the large-scale dataset used in this work only contained floor plans with axis-aligned walls.

Some known works use machine learning techniques. [20] proposed a graph-constrained relational generative adversarial network method for house layout generation problem not compatible with a fixed apartment border given as an input. [21] proposes a deep learning

* Corresponding author.
E-mail address: bastien@habx.fr (B. Dolla).

https://doi.org/10.1016/j.autcon.2020.103491
Received 28 March 2020; Received in revised form 3 November 2020; Accepted 21 November 2020
Available online 15 December 2020

framework driven by a layout graph for floorplan generation. It enables user preferences and fixed apartment borders in inputs. However, these methods have two major limitations compared to the approach presented in this article: they are based on existing datasets which are not easy to collect and which may induce an architectural bias and they only deal with rectilinear plans.

Our work aims in particular to fill the gap in existing literature on automatic floor plan generation in non rectilinear enveloppes. Floor plans can contain a high amount of information (e.g., rooms may have various and complex non-standard polygonal shapes) and they may be structured by numerous imperative constraints as well as by aesthetic considerations that can hardly be formalized as imperative constraints. The problem dimensionality and the non-imperative nature of some design rules make the use of any of those previous techniques alone a challenging path.

We have developed an automatic floor plan generator called Optimizer. For a given apartment envelope and specifications on both the types of rooms it shall contain and their respective area, Optimizer generates various layouts that respect architectural and functional constraints along with the required specifications, without any human intervention. To do so we adopt a mixed approach. The envelope is first decomposed into a grid designed based on architectural considerations and adapted to any envelope shape. The grid cells are then allocated to the specified rooms in a constraint programming framework, an allocation is performed so as to respect architectural and functional constraints. Various layouts respecting the constraints can be obtained for a given couple (plan envelope, specifications). Circulations are then drawn in the layouts and a final cell shuffle is performed through a genetic optimization algorithm so as to obtain functional and aesthetic layouts. The process run by Optimizer is detailed in this article.

2. General principle

Optimizer is a novel algorithm for automatic floor plan generation. From an apartment envelope and a requested list of rooms with specified areas, it generates a set of layouts satisfying both architectural and functional constraints as well as the input specifications. See Fig. 1 for input/output illustration.

2.1. Input data

The apartment envelopes used as input for our algorithm are directly extracted from IFC[1] models built for residential new developments.

An apartment envelope is described from its so-called structural elements: the external limit of the apartment, the openings (window, French window and entrance door), technical ducts, load-bearing walls and pillars, external spaces (balcony, garden…) and stairs (Fig. 2).

In this paper, we assume that apartment envelopes are adequate with the input specification, i.e., the layout surface is approximately equal to the required area deduced from the room list, and there are enough ducts and windows to satisfy the specification. Area margins are added in the algorithm to account for circulations (see section *6*)

2.2. Description of the algorithm

A key design idea Optimizer is built upon lies in turning the plan generation into a space allocation problem. The creation of an apartment floorplan can be viewed as the partition of the apartment envelope polygon into several sub-polygons. Their assembly corresponding to a specific room or circulation space of the apartment and the assembly choice following various user-defined constraints: size, shape, adjacencies… The apartment is divided into grid cells that are assigned to the specified rooms. To do so, Optimizer operates sequentially as follows (see Fig. 3):

1. **Grid generation**: the apartment envelope is discretized into a grid. The grid structure is built according to elementary architectural rules. The walls of the final layout configurations will lie among the grid edges.
2. **Rooms placement through constraint programming:** To reduce the combinatorial size of the problem, some grid cells can be merged based on architectural and functional considerations. Each cell is assigned to a room. This creates a possible apartment layout. We look for every combination that satisfies defined architectural and functional constraints.
3. **Layouts selection**: At this stage, all cell distributions satisfying the constraints have been generated. We aim at keeping N layouts that are most relevant in terms of room distribution while being different from each other. To do so, the layouts are clustered, apartments are scored and the top N are kept.
4. **Circulation setting**: If it is not possible to circulate, corridors are added by allocating cells to this use.
5. **Final cell allocation optimization**: After the apartment has been divided into a functional layout and circulations have been set, a final cell shuffle is performed through a genetic optimization algorithm so as to improve several objectives such as room shape and room area.

For a given envelope and room list request, Optimizer generates from 0 to N layouts. In the following, N is set to $N = 3$.

Note that all plans shown in the illustrations and examples are referenced with their results in the appendix.

3. Grid generation

3.1. Objectives

In order to significantly reduce the search space for our problem, we create a generic grid inside the envelope polygon, thus transforming the partition problem into an assignment problem: which room of the floor plan should be assigned to each cell of the grid. The search space can therefore be reduced to the discrete parametrization of a boolean matrix M of shape N x S (N being the number of cells of the grid and S the number of rooms or circulation spaces of the apartment) with each element M[i, j] of M specifying whether the ith cell of the grid is assigned to the jth room of the apartement.

A simple approach to the grid creation would be to partition the floor plan via a regular cartesian grid (Fig. 4). However, this naïve method is not optimal as some parts of the floor plan might need more precision than others and because the grid should ideally take into account alignments to the outward corners of the apartment perimeter. To solve this problem, we created a specific ruled based method.

3.2. Grid principles

The grid is built inside the apartment's external perimeter, taking into account its openings such as the front door and the windows through the recursive application of a predefined set of shape rules, that we call "actions".

In order to facilitate the adjacency traversal of the grid, we represent it as a mesh using a half-edge data structure. This grants us fast access to topological information which will frequently be used in the grid operation and later on in the assignment problem. This structure also enables us to define atomic "actions" that each perform a specific transformation of the grid. These actions are the combination of a left-hand side query on the half-edge primitives and a right-hand and side transformation. The left-hand side query yields a specific half-edge of

[1] The Industry Foundation Classes (IFC) is an open data model intended to describe architectural, building and construction industry data. It's commonly available in most software used for Building Information Modeling (BIM)

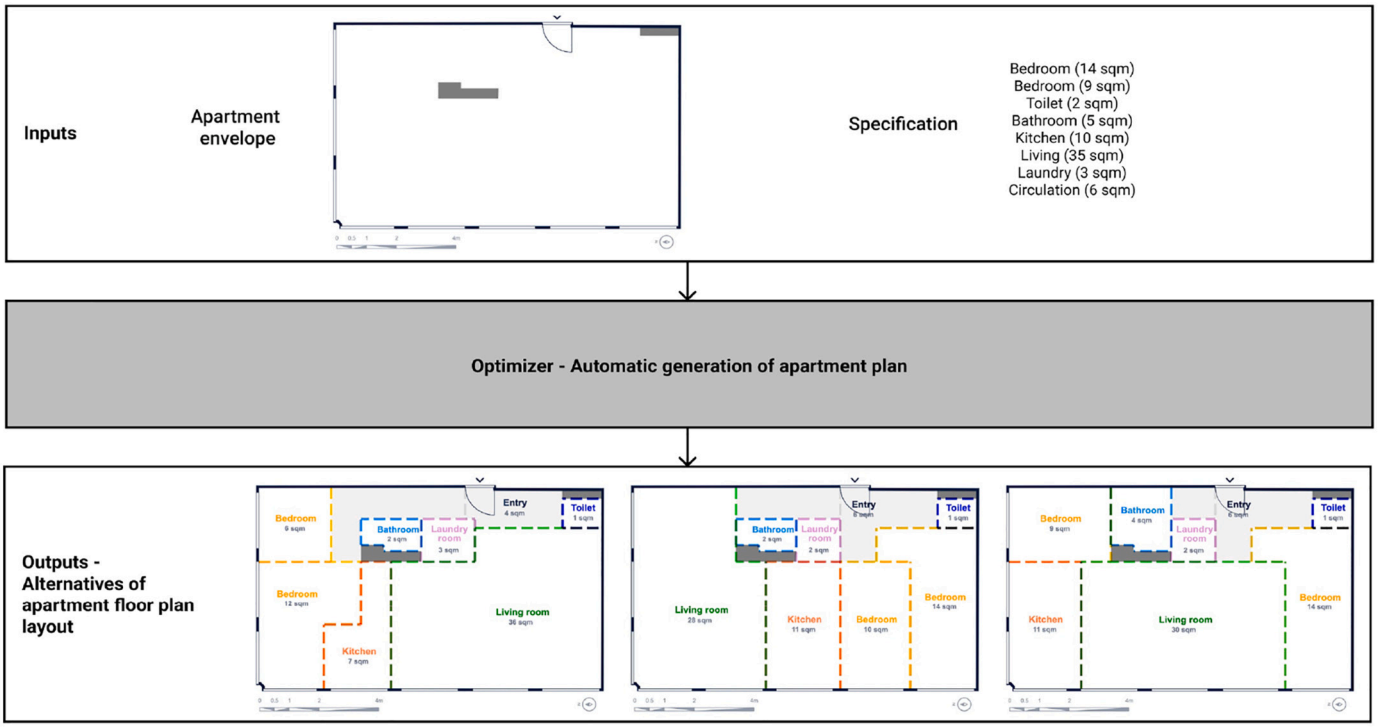


Fig. 1. Illustration of Optimizer inputs and outputs. Optimizer takes as input an apartment envelope along with specifications: a list of rooms with their objective area. It generates a list of layouts satisfying both those specifications and architectural and functional constraints.

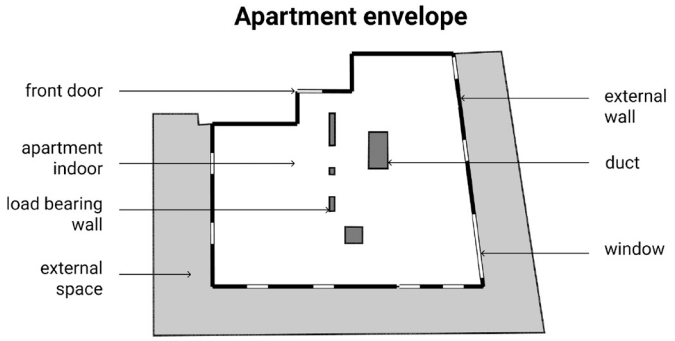


Fig. 2. Example of apartment envelope fed into Optimizer. The envelope contains an internal space (apartment indoor) as well as so-called external spaces (balconies, terrace…), apertures, ducts, load-bearing walls, pillars.

the grid that will then be fed to the transformation operator defined in the right-hand side. This transformation will in turn create new half-edges or delete existing ones. The action is recursively reapplied until no half-edges are returned by the query.

The grid is constructed through the application of a series of such actions on the initial perimeters (42 different actions were used for the example shown in Fig. 5, the Fig. 6 provides four of these actions as an example). This is useful to design different grid generation methods for different buildings, according to their specific architecture styles and floor plan shapes.

In order to tackle the very common case of a non-rectilinear polygonal envelope, a specific strategy was put in place. First, a primary orientation of the floor plan is determined by adding for each wall angle (modulo 90°) the length of the corresponding walls and keeping the angle with the highest count. The edges created by the actions transformations are then automatically aligned to the primary orientation or to its orthogonal counterpart. The method is applied recursively to each newly created face whose primary orientation might be different. The corresponding grid is then stitched at the frontiers between zones of different primary orientations via a specific action (Fig. 7).

4. Rooms placement through constraint programming

At this step, grid cells are distributed among the rooms the apartment shall contain. A given cell distribution corresponds to a solution layout. Here, we aim at generating all solutions that satisfy architectural and functional requirements specified as constraints.

4.1. Grouping of cells

For a grid containing N_c cells, if there are N_r rooms to position, there are $N_r^{N_c}$ possible cell allocation combinations in case no constraint is applied.

The problem size grows exponentially with the number of cells, and so does the computation time. Also, when the grid contains many cells, especially small ones, many solutions can just be a pixelated version of the same configuration.

To tackle this issue, when the grid contains a high number of cells some are grouped together according to the following process (Fig. 8):

1. Some seeds are planted in cells containing a window, a door entrance and some of those that are adjacent to a duct.
2. Those seeds are grown by merging adjacent grid cells iteratively. Growth is controlled so as to preserve proper shape ratios, maximum width, length, and area.
3. Once seeds are grown to their maximal expansion, the residual layout space is divided by extending every seed border.
4. Resulting units with a low area are merged to the neighbour that has the highest contact length. In case several neighbours have the same contact length, units are merged with the smallest one.

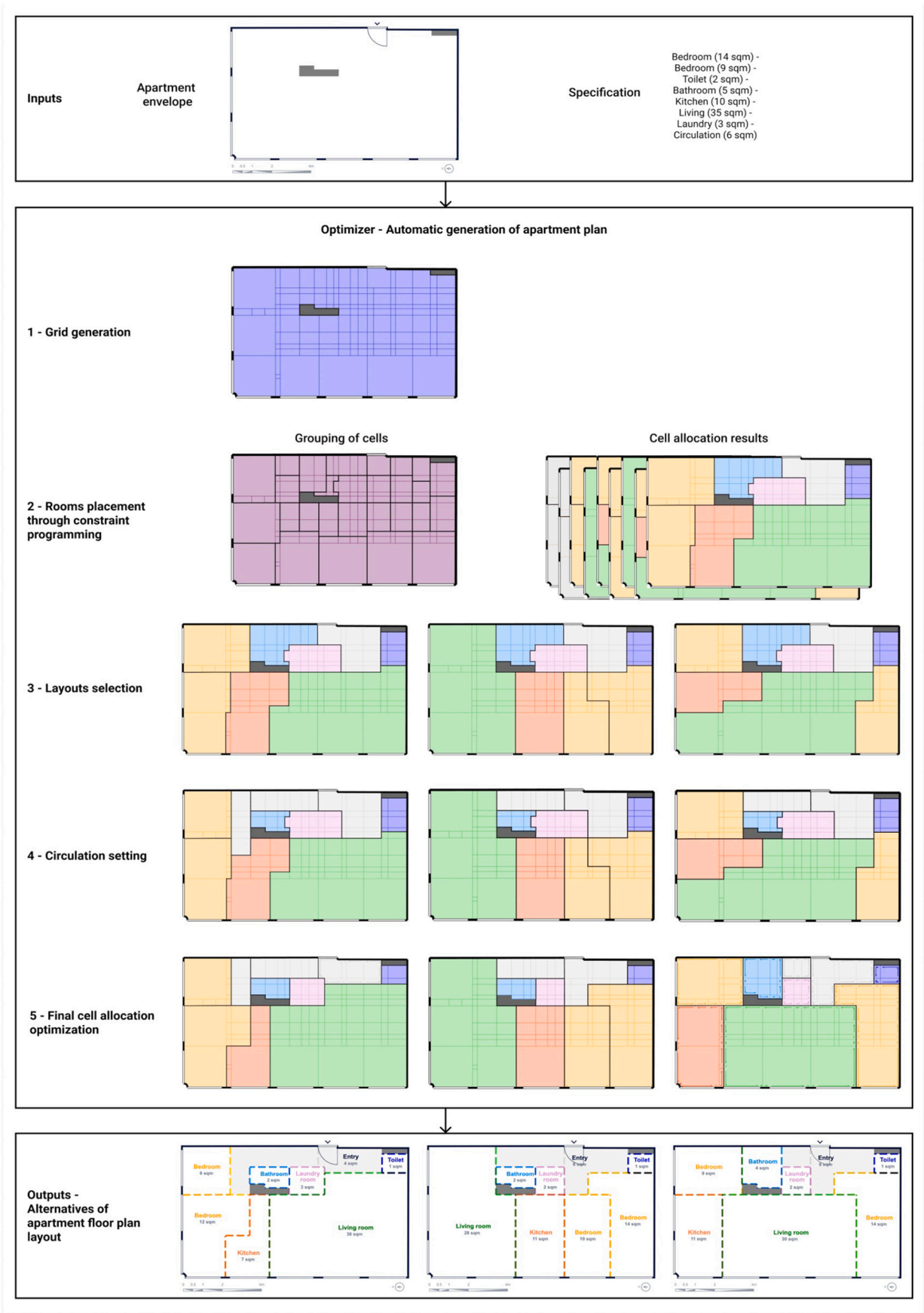


Fig. 3. General illustration of Optimizer flow. The results of each Optimizer step are illustrated in the diagram. Each room type is identified by a given colour.

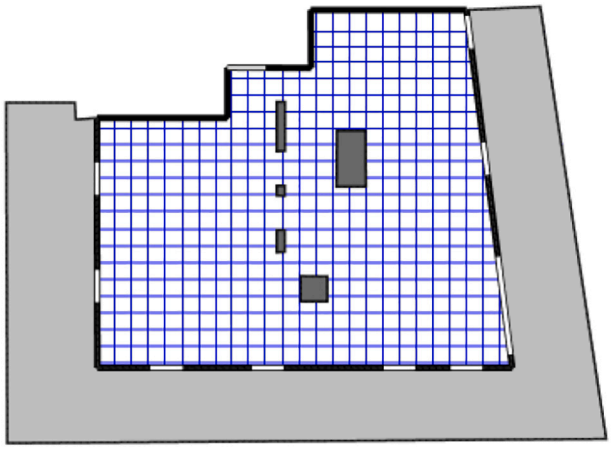


Fig. 4. Floor plan with a regular cartesian grid (~490 cells).

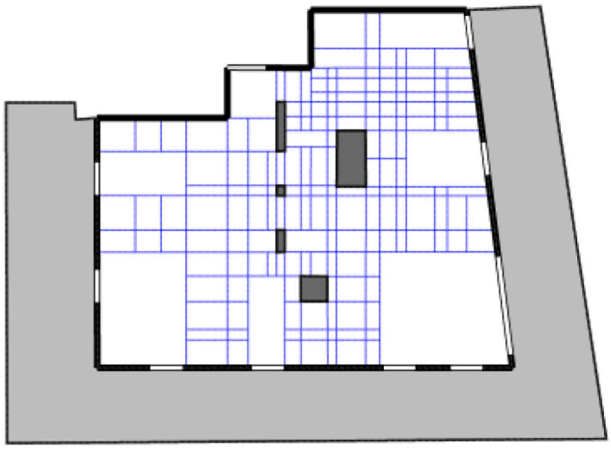


Fig. 5. Floor plan with an optimized grid (186 cells)

4.2. A cell allocation problem under constraints

The cell allocation problem is solved in those previously described macro-cells. Solving the cell allocation problem amounts to finding the cell distributions among specified rooms that satisfy the here-after specified constraints.

A non-exhaustive list of constraints, classified into various categories, is described below. Most of those constraints are imperative so as to make the plans technically constructible (for example the proximity between a bathroom and a duct) or to meet the specification (e.g.: all the rooms have to be positioned). However, we have introduced some subjective constraints at this stage to avoid generating plans that are not interesting from a purely functional point of view (e.g.: large external spaces must be accessible from a living room, two toilets shall not be positioned side by side…).

Geometric constraints:

- Each cell of the grid has to be allocated to a single room.
- Each room contained the input specification should have at least one grid allocated cell.
- All cells allocated to a given room shall be simply connected, i.e., connected by a path of adjacent cells.

Architectural constraints:

- The room area should be close to the area specified.
- Room shapes have to satisfy several criteria to be accepted. Notably, a consistent ratio between the square perimeter and the surface is imposed. This ratio depends on the room type.
- Some rooms need to contain some structural elements. For instance, the toilet and bathroom should be adjacent to a duct, the living shall contain at least one window…Those constraints are illustrated in the Table 1.

Functional constraints:

- So as to ensure proper lighting, the glass surface of rooms containing windows has to be bigger than a given percentage of the room surface area.
- In the input specification, two rooms can be open on each other (e.g., the kitchen and the living room). In case two rooms are required to be open on each other, a constraint is applied to impose that they share a common boundary of a length greater than a minimum value (the "open on" room is then authorized to have no window).
- The living room, kitchen and dining room should form a single functional block.
- No private room (bedroom, bathroom, laundry room, dressing room, WC) shall be isolated, i.e., any private room shall be adjacent to at least another private room. This ensures the creation of functional blocks.
- Two toilets should not be adjacent.
- In addition, access to large external spaces (garden, terrace, large balcony) from the living room is imposed.

Specific functional constraints have been defined for apartments with several levels, but are not exposed in this article. These constraints define for example room types allowed to get a staircase, but also the relative position of the rooms in relation to each other, or impose toilets on the ground floor.

Efficiency constraints:

In order to speed up the research process, we set up two additional constraints that prune the search domain.

- If the specifications contain two rooms with the same type and size, two of the solutions at least would be identical. To avoid redundancy, we introduce a symmetry breaker. Breaking symmetries of a model is a very effective technique to reduce the size of the search tree and consequently the search time [22].
- In a given room, two cells cannot be too far from one another. A constraint on the maximal distance between any two cells belonging to the same room is set depending on the room surface area.

4.3. A constraint programming problem

To attribute grid cells to the rooms while respecting the aforementioned constraints, we use a constraint programming solver [23]. This paradigm enables to get an exhaustive list of solutions that respect given imperative constraints.

Problem formalization:

For a grid containing N_c cells, if there are N_r rooms to position, the plan is represented by a vector $V = (v_i)_{i\in[1,Nc]}$ where v_i is the room index attributed to the cell i, $v_i \in [1, N_r]$. V is the set of variables of the constraint programming problem. Solving the cell allocation problem consists in finding all vectors that represent layouts satisfying our constraints.

To ease the expression of the constraints, we introduce an intermediary matrix of Booleans P, the so-called position matrix, defined by:

$$P(i,\ j)_{i\in[0,\ N_R-1],\ j\in[0,\ N_C-1]} = \begin{cases} 1 & if\ v_j = i \\ 0 & else \end{cases}$$

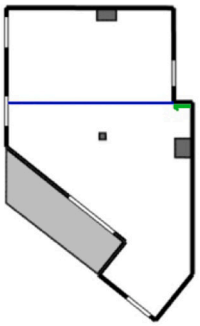
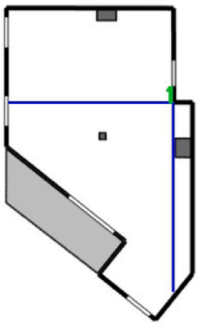
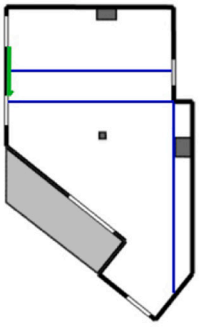
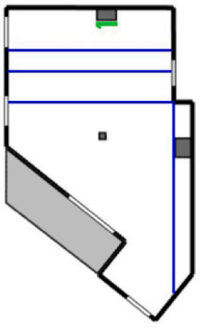
#	Left Hand Side (LHS)	Right Hand Side (RHS)	Example
1	Select a half-edge whose angle with the next half-edge is >= 270 deg.	Create a new cell by splitting the cell of the half-edge by drawing an edge starting at the end of the half-edge and perpendicular to the next half-edge	
2	Select a half-edge whose angle with the previous half-edge is >= 270 deg.	Create a new cell by splitting the cell of the half-edge by drawing an edge starting at the start of the half-edge and perpendicular to the previous half-edge	
3	Select a half-edge located between two windows	Create a new cell by splitting the cell of the half-edge by drawing an edge starting in the middle of the half-edge and orthogonal to it.	
4	Select a half-edge adjacent to a duct space located between two half-edge adjacent to the same duct	Create a new cell by drawing and edge parallel to the half-edge at a distance of 1.8 meter from it on its left-hand-side.	

Fig. 6. Examples of actions used in the grid generation method (the half-edges selected in green via the LHS condition and the half-edges created in blue by the RHS transformation). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Constraint formalization examples:

For instance, to express the constraint imposing that rooms surface area closed to the specified areas:

$$\forall i \in [0,\ N_r - 1],\ S_{Room_i} = \sum_{j=0}^{N_c - 1} S_{cell_j} * P(i,\ j)$$

$$MaxArea_i \geq S_{Room_i} \geq MinArea_i$$

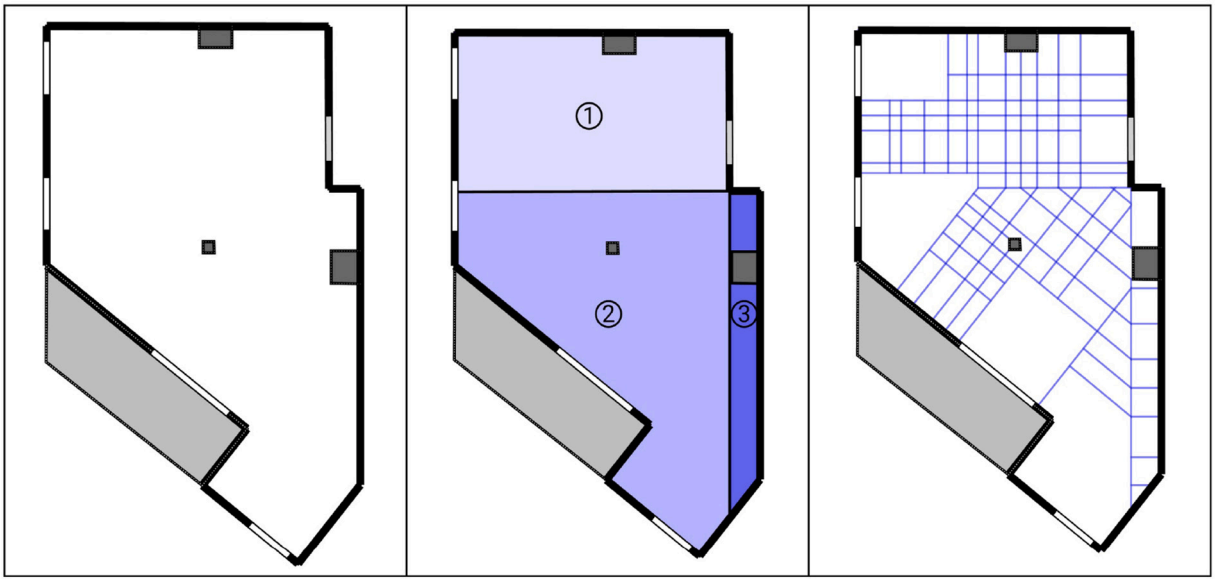


Fig. 7. Bi-directional floor plan grid steps - We observe the initial input envelope. Then the floor plan is split into three cells via a specific action. Each cell is then recursively split according to their primary directions.

where S_{Roomi} is the surface area of the room with index i, S_{cellj} is the surface area of the cell with index j and $MaxArea_i$ and $MinArea_i$ are the room i maximum and minimum surface areas deduced from the input specification.

As another example, let us express the connectivity constraint. As described in the previous section, we require all cells allocated to a room to be simply connected.

We define a so-called adjacency matrix A by:

$$A(j,\ k) = \begin{cases} 1 & \textit{if } cell_j \textit{ and } cell_k \textit{ are adjacent} \\ 0 & \textit{else} \end{cases}$$

Thus, let the matrix a_i of cells adjacency inside a room i defined by:

$\forall j,\ k \in [0,\ Nc\ -\ 1]\ ai(j,\ k)\ =\ A(j,\ k)\ *\ P(i,\ j)\ *\ P(i,\ k)$ where i is the current room index.

It can be shown using graph transitive closure that the connectivity constraint between cells j and k of room i is satisfied when $T(j,k) = 1$ where T is the matrix defined by:

$T = a_i + {a_i}^2 + {a_i}^3 + \ldots + {a_i}^{\sum_{j=0}^{N_c-1} P(i,\ j)-1}$ where $\sum_{j=0}^{N_c-1} P(i,\ j)$ is the number of cells in the room R_i.

This constraint has a high computational cost since whenever the solver tests a cell allocation configuration, many matrix products have to be computed and stored (each matrix itself contains programming constraints).

Instead, we propose a set of lighter constraints that are necessary to impose connectivity but not sufficient. Those constraints are described hereafter. Note that some of the obtained solutions might thus not be admissible and an a posteriori check enables us to remove them.

- The number of adjacencies between the cells belonging to the same room is greater than the number of cells attributed to this room minus one. The number $X_i$of cell adjacencies in room $R_i$is defined by:

$$X_i = \sum_{k=0}^{n_c-1} \left(\sum_{j>k}^{n_c-1} A(j,k)\ P(i,j)\ P(i,k) \right)$$

This constraint is defined by:

$$\forall i \in [0, N_r - 1]\ X_i \geq \sum_{j=0}^{N_c-1} P(i,j) - 1$$

- Each cell attributed to the room is adjacent to at least one other cell of this room.

Note that

$\forall j\ \in\ [0,\ N_c\ -\ 1],\ \sum_{j\neq k}^{N_c-1} A(j,\ k)\ P(i,\ j)\ \geq\ 1$ *if cell j is adjacent to at least* 1 *other cell in room* R_i.

We define the matrix Z by:

$$Z_{ik} = P(i,k) \left(\sum_{j\neq k}^{N_c-1} A(j,k)\ P(i,j) \right)$$

This constraint is defined by:

$$\forall i \in [0, N_r - 1] \forall k \in [0, N_c - 1]\ Z_{ik} \geq P(i,k_k)\ or\ \sum_{j=0}^{N_c-1} P(i,j) = 1$$

For the sake of concision, the whole list of constraints implemented in Optimizer is not presented here. The logic used in the formalization of the algorithm remains the same as for the previously described constraints.

As an output of the constraint solver, we obtain all the possible layouts that satisfy our constrained problem.

4.4. Compromise between solution diversity and computation time

We aim at generating a great variety of feasible and qualitative layouts as quickly as possible.

In some cases, the number of solutions is very high, which makes the search time consuming; we shall then strengthen the constraints and eventually stop the search arbitrarily. In other cases, we may not get any solution, we would then tend to relax the constraints. To get a good compromise between variety and computation time, we act both on the parameters relative to the solver and on the constraints themselves.

Solver parameters:

- a time limit is imposed on to the search (neither too small, nor too high, to satisfy our compromise)
- the solver is stopped whenever the number of generated solutions reaches an arbitrary limit

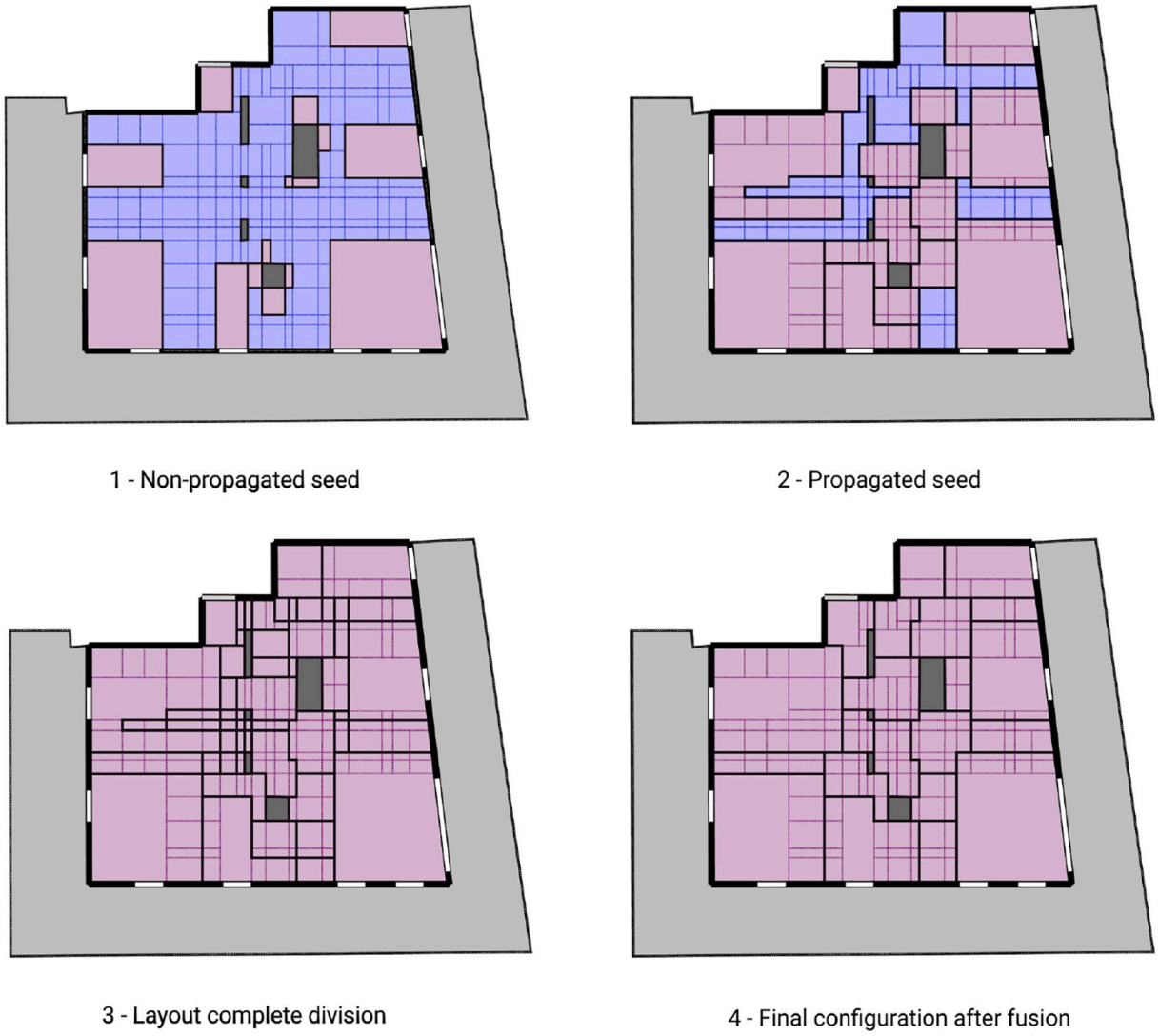


Fig. 8. Illustration of the cell grouping algorithm step

Table 1
Table showing the adjacency constraints between the structural elements of the plan and the rooms to be positioned.

	Front door	Window	Duct
Entrance	Required	Forbidden	Possible
Toilet	Forbidden	Forbidden	Required
Bathroom	Forbidden	Possible	Required
Living	Possible	Required	Possible
Living/kitchen	Possible	Required	Required
Dining	Forbidden	Required	Possible
Kitchen	Forbidden	Required	Required
Bedroom	Forbidden	Required	Possible
Office	Forbidden	Required	Possible
Laundry	Forbidden	Forbidden	Required
Dressing	Forbidden	Forbidden	Possible

Constraints parameters:

- admissible error on the target surface area
- ratio limit on the room shape constraint
- …

Both solver and constraint parameters were set through trial and error on a test database.

We illustrate the variability of the number of solution layouts and the computation time for couples (plan envelope, specification) with similar areas and close specifications (Table 2).

We make the following observations:

First of all, the number of grid cells and macro-cells is not simply correlated to the surface area or to the number of grid cells. It highly depends on the envelope structure (size, shape, and position of windows

Table 2
Diversity with regard to the number of solutions and the computation time illustrated on three couples (plan envelope, specification) with similar areas and specifications.

Plan envelope (see appendix)	n°4	n°9	n°1
Plan area	82.4 m2	75.5 m2	79.8 m2
Number of rooms required	7	7	8
Number of grid cells	190	125	143
Number of macro-cells	21	34	28
Total number of cell allocation configurations	5,58E+17	5,41E+28	1,93E+25
Number of layouts generated by the constraint programming solver	194	8660	21,218
Number of layouts filtered - those where cell connection is not satisfied (see part c).	0	5017	7065
Process time	1.4 s	57.15 s	139,76 s

and ducts).

The size of the allocation problem can thus vary greatly from one plan to another leading to various number of solutions. Note also that the number of solutions is not simply correlated to the problem size. Constraints will prune the search tree with various intensities depending on the plan geometry, on the types and positions of structural elements and on the input specifications. The impact of the constraints on the tree pruning is complex and inextricable. The number of layouts generated and the computation time of the placement of room through constraint programming algorithms are not predictable.

When keeping only admissible solutions, we may exclude from zero to thousands of unconnected layouts and the final number of solutions has a high variability.

It is interesting to note that the processing time and the number of obtained solutions can be severely influenced by the types of rooms as well. Let us consider for example the case of two identical envelopes (plan n°9 - see appendix) where specifications differ: in one case a laundry room and in the other one a dressing room, both with the same area. The latter will lead to many more solutions because a dressing room is less constrained in its position within the floorplan since it does not require a water supply. See the process times obtained on a given plan in the Table 3:

5. Layouts selection

In this section, we describe how to keep, among all the solutions obtained through constraint programming, the three most relevant ones while being different from each other in terms of room distribution.

To do so, the layouts are clustered according to a custom metric that focuses on the distribution of the cells around structural elements. One element from each cluster is scored and the best three layouts are finally kept for the following steps.

5.1. Clustering

To propose only layouts with significant differences, we start by clustering similar solutions together.

To do so we use the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm from the sklearn library [24]. DBSCAN is chosen here because we do not have a priori on the number of clusters that should be obtained, we rather have intuition on which layout shall be assigned to the same cluster based on their relative distance.

A layout i is described by the matrix of booleans P_i as defined in part 4.c. The distance D_{ij} between two solutions i and j is computed using a custom distance that increases with the number of cells assigned to different rooms. This distance considers the size of each cell as well as the presence of structural elements around it, it is defined by:

$$\forall i,j \in [0, numberofsolutions]\ D_{ij} = \sum_{k=0}^{N_r-1}\sum_{l=0}^{N_c-1} Coeff(l)*S_{cell_l} \Big/ S_{plan}*abs(\,P(k,l) - P(k,l)\,)$$

$$where \begin{cases} S_{plan}\ is\ the\ plan\ surface\ area \\ S_{cell_l} is\ the\ cell\ l\ surface\ area \end{cases}$$

$$Coeff(l) = \begin{cases} 3\ \ if\ the\ cell\ l\ is\ adjacent\ to\ a\ window\ or\ a\ door\ window \\ 2\ \ if\ the\ cell\ l\ is\ adjacent\ to\ a\ duct\ or\ the\ frontdoor \\ 1\ \ else \end{cases}$$

Table 3
Impact of the room type on the number of solutions and the process time.

	Specification with laundry room	Specification with dressing room
Number of solutions	1261	8660
Processing time	6.57 s	57.15 s

At this step, only the global position of the rooms is relevant to us. To give an intuition, a cluster shall contain layouts where rooms have the same overall position, i.e., from one layout to the other a given room contains the same structural elements (windows, ducts…) but its walls positions may change. Two pixelated versions of the same configuration shall thus lie in the same cluster. Walls positions will be optimized afterward (see section 7).

In the Fig. 9, we illustrate how cell distribution may vary in a given cluster. Cells in red (those containing a structural element) are more likely to be distributed to the same rooms within the cluster while orange ones can have different allocations.

In Fig. 10 we illustrate how a given room (here the living room) is distributed among the cells in various clusters. As described before, cells containing structural elements are more likely to be allocated to the same room in a given cluster while other cells attributions might vary within the cluster. The overall room position changes from one cluster to another. A given cluster represents a layout type. One element per cluster is then arbitrarily selected and scored and we finally keep the three best ones among those selected.

5.2. Layout scoring

Our final aim is to keep the most relevant layouts. To do so, each selected layout has to be scored in some way.

As said before, Optimizer will finally output at most three relevant and various solutions. At this stage, there are as many layouts as there are clusters, i.e., N. In the following sections, we show how layouts obtained after the constraint programming part are redesigned so as to introduce circulations and to improve both the respect of specifications and architectural quality. We could redesign those N layouts and select the three best ones afterward. But for performance purposes, as the redesign is costly, we only keep three best layouts among those N and

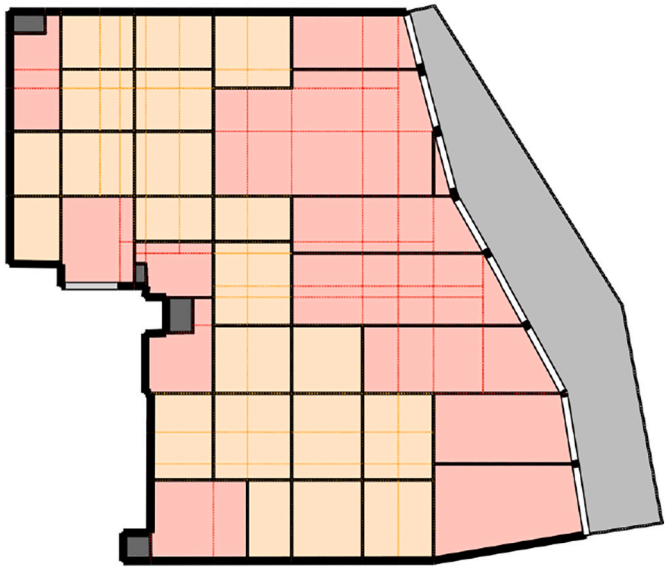


Fig. 9. Cells weighting within a cluster - Cells in red are likely to be attributed to the same rooms within a given cluster while yellow ones might have varying room allocations. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

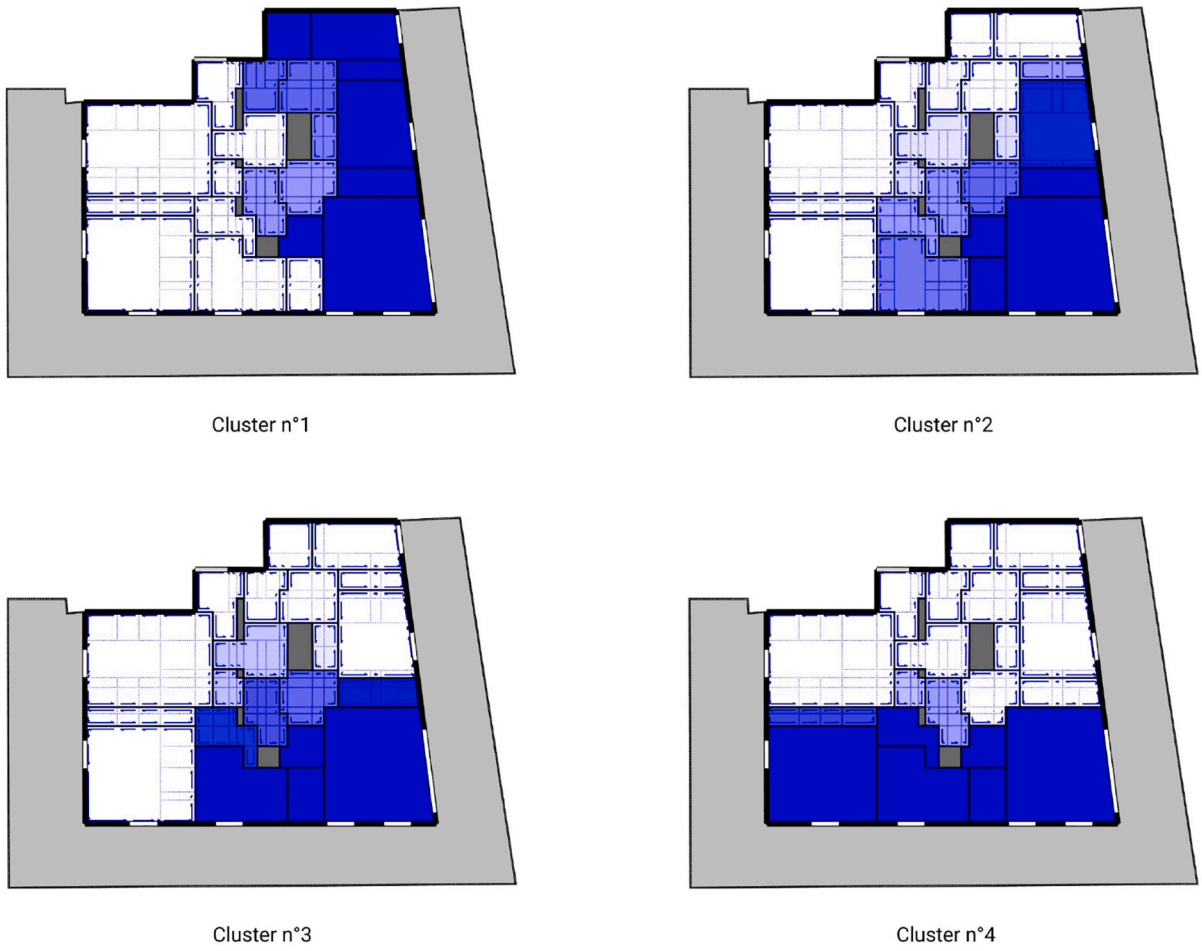


Fig. 10. Illustration of inter and intra-cluster variability- Each plot illustrates a cluster. The darker the cells, the more frequent the living room is assigned to these cells within the illustrated cluster. For instance, if a cell is constantly allocated to the living room, it will appear in dark blue while it will be white if it never belongs to it in the considered cluster. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

proceed to the next steps for those only.

To do so we designed scoring functions so as to evaluate, among other criteria:

- the respect of the specified objective area,
- the relevance of rooms positions in relation to one another,
- the room shape,
- the potential for circulating in the layout.

6. Circulation setting

At this stage of the algorithm, there is no guarantee that it is possible to circulate in the apartment since no corridor has been drawn. In this part, we expose how corridors are inserted in the layout.

In the following, we define as 'circulating rooms' the rooms belonging to the set {'living', 'living kitchen', 'entrance', 'dining room', 'corridor'}. Other rooms are referred to as 'non-circulating rooms'. A room is said to be 'isolated' if it is not possible to reach it when coming from the entrance door.

To draw circulations in the apartment we proceed into two steps. First, we find circulation paths in the apartment so that every isolated room is connected to the entrance. Those circulation paths are mono-dimensional sets of edges chosen in the grid. Then, based on those circulation paths, some grid cells are attributed to newly created corridors.

6.1. Determination of circulation paths

We call G_e the graph of edges of the grid. We can draw a path between two non-adjacent rooms R_i and R_j by picking a list of connected edges of G_e whose first element has a node belonging to R_i and whose last element has a node in R_j. There can be many paths between R_i and R_j. To draw appropriate corridors, we use the following rules:

- short corridors are preferred
- a corridor shall not contain a window, except if there is no other choice
- a corridor shall not separate a room that requires a duct from all the ducts it is adjacent to.

Each edge of G_e is given a weight that accounts for those rules and the circulation path between R_i and R_j is defined as the shortest path (i.e., the one with the lowest accumulated weight) linking those two rooms. This path is found using the Dijkstra algorithm.

In the following, connecting two rooms refers to finding the shortest path between those two rooms.

To determine the circulation paths in the layout we connect iteratively each isolated room R_i to the closest circulating room that is not isolated (i.e., the one that has the shortest path to R_i). Isolated rooms are treated in an arbitrary order but starting with circulating rooms. Once a room has been connected, it is no longer isolated. See Fig. 11 for an illustration of the process.

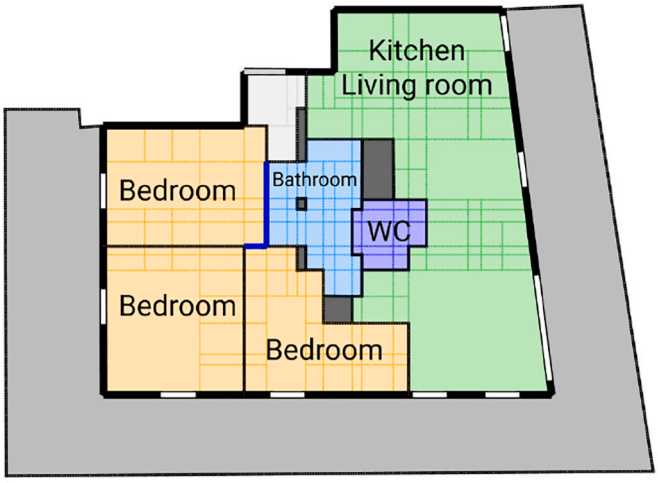


Fig. 11. Circulation path connected isolated rooms. The path is illustrated by the blue line. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

NB: when a circulation path has been formed, edges of this path have their weights reduced to zero so as to encourage further circulation paths to share parts with existing ones and hence reduce the total corridor length.

6.2. Corridor cell attribution

Now that circulation paths have been drawn, some grid cells are attributed to new rooms defined as corridors (Fig. 12). For each straight portion of a circulation path, one can decide to grow the corridor on one side, the other one, or both. The process is designed so that corridors grow preferentially on rooms that are larger than their target area objectives. Growth is stopped when the corridor width has reached a given arbitrary value. NB: a corridor cannot grow on a duct nor cut a room into pieces.

Limitation:

Since the order in which rooms are connected is arbitrary, the final corridor configuration might not be optimal. An improvement would be to find among all the corridor combinations the one with the lowest cost. Nevertheless, the processing time would be longer.

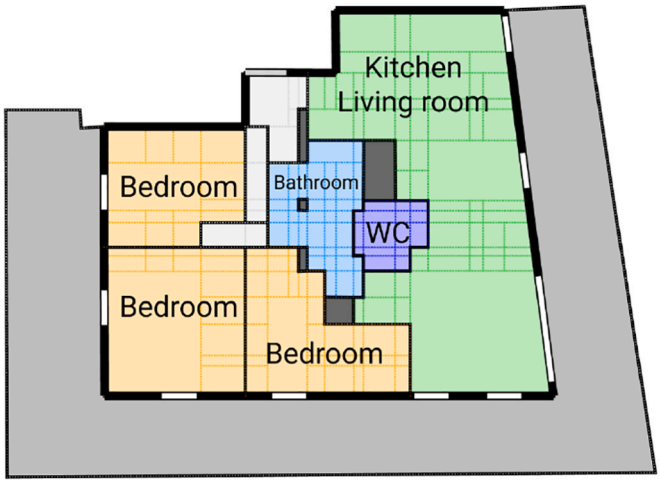


Fig. 12. Final corridor spaces - Corridor spaces are drawn in grey.

7. Final cell allocation optimization

7.1. Problem

In order to reduce the time spent by the constraint solver, its search space was reduced through the grouping of the cells of the grid in bigger cells (cf. part 4.a). This implies that the solutions returned are suboptimal with respect to the wider original search space. Moreover, the number and precision of the constraints given to the solver must be limited in order to reduce the processing time and to ensure that the problem has some solutions.

The solutions returned exhibit approximate shapes and areas that are insufficient for a satisfying architectural result (Fig. 13). In particular, architectural constraints such as the quality of the room shape (number of corners, concavity, alignments, symmetry, etc.) would be very hard to express in the context of a constraint programming algorithm. In addition, these architectural constraints are not imperative and a compromise must be found between all of them, making it more of an optimization problem than a constraint solving problem.

The solution returned by the constraint solver is thus used as a seeding point for an optimization algorithm that will try to find an optimal solution in the vicinity of it (Fig. 14). We chose to use a multiple-objective genetic algorithm for this subsequent optimization for multiple reasons: the freedom to define very complex, non-linear objectives, the capacity to achieve multiple objective searches and the possibility to parallelize the computation. Because of these advantages, genetic algorithms have been widely used in generative design with recent applications to real-life projects [25].

7.2. Algorithm

The algorithm used is based on NSGA-II [26] and its implementation inspired by the DEAP open-source project [27].

In the genetic representation of the optimization problem, an individual represents a floor plan and its genes represent the list of the cells of the grid assigned to each of its rooms.

$$Individual_1 = \{space_1 : [F_i, F_j, \ldots, F_k], space_2 : [F_l, F_m, \ldots, F_n], \ldots, space_n : [F_o, F_p, \ldots, F_q]\}$$

Main steps of the genetic algorithm used for the local optimization of the floor plan are described in Fig. 15.

7.2.1. Objectives and fitness

The goal of the optimization is to refine the raw solution found by the

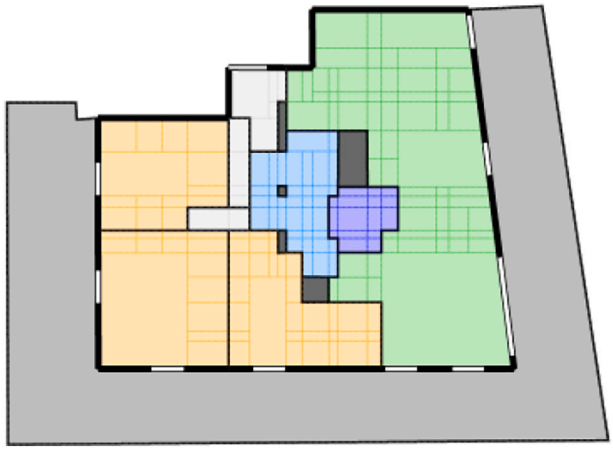


Fig. 13. Solution returned by the constraint solver with the created corridor. The rooms are "pixelated" with an excessive number of corners and their areas are imprecise.

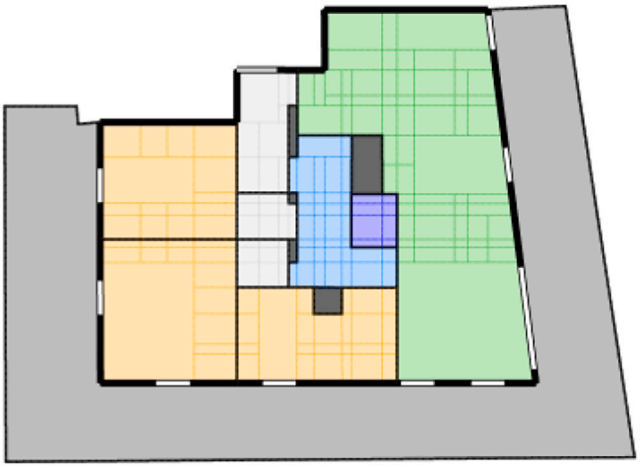


Fig. 14. Optimized floor plan via the refiner. The rooms are more aesthetically pleasing due to better alignment, more convex shapes, and fewer corners.

constraint solver into a more aesthetically pleasing and architecturally sound one. This is a highly subjective problem as the quality of a floor plan will be appreciated differently depending on the culture, the lifestyle and the personal taste of its inhabitants. Opinionated choices must be made in order to translate the perceived quality of a plan into a set of quantified variables: the characteristics that seem the most important must be selected, the way to model and quantify them must be determined.

The characteristics of a floor plan can be separated into two groups:

- individual qualities of each room of the plan (such as the shape of the room, its lighting, its area, etc.),
- global quality of the floor plan (such as the distribution, the symmetry, the ease of circulation, etc.).

For the sake of simplicity, we deliberately chose to focus narrowly on the individual qualities of each room, making the strong approximation that a floor plan composed of "good" rooms would make a "good" floor plan. We also made the assumption that we would pursue the same objectives for each room whatever its type (bathroom, bedroom, living, etc.), albeit with different weights per type for each objective.

This enabled us to structure a simple fitness evaluation as a two-dimensional matrix as Fig. 16: crossing the multiple objectives sought after and the multiple rooms composing the apartment.

Working with our architectural team, we selected six objectives (see Fig. 17) that we deemed both essential to a "good" architectural room and easily computable in a fast and efficient manner. It was not our intention to reduce architectural design to these six objectives. Many other choices would be equally or more relevant. These six choices were pertaining to our global ambition of generating a fast and simple apartment floor plan. Regardless, the proposed methodology was designed to be used with any set of arbitrary objectives.

7.2.2. Mutations

The main challenge presented when randomly modifying a floor plan is to avoid a mutation that would split a given room into several disconnected components or completely remove it. Therefore, the simplest way to incrementally modify a floor plan is to swap a cell between two adjacent rooms. For a given room, this creates two possible mutations: the removal of a cell or the addition of a cell. This mutation also ensures an exhaustive exploration of the search space.

In order to increase the convergence speed of our algorithm, we decided to use another mutation that would add or remove a whole row of aligned cells. The objective was to limit the number of mutations

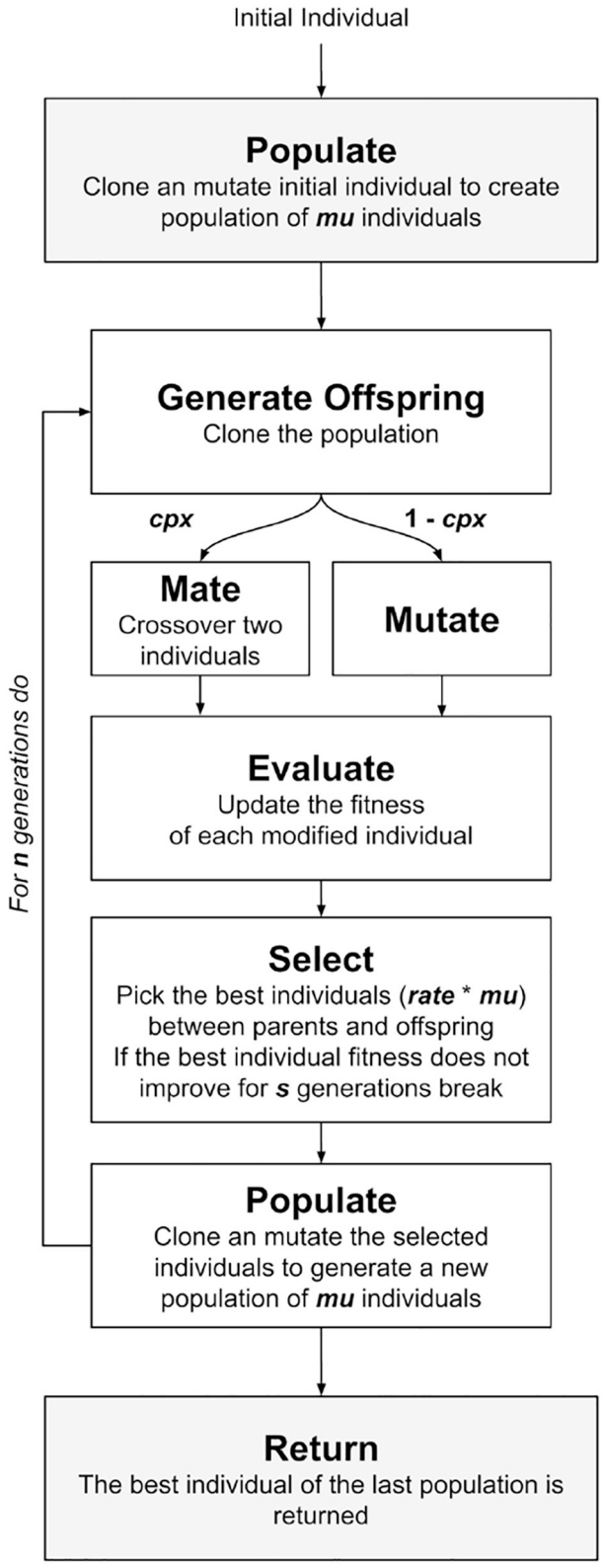


Fig. 15. The main steps of the genetic algorithm used for the local optimization of the floor plan.

needed to translate a wall between two rooms and to help the algorithm converge towards solutions with a maximum of aligned edges. However, this mutation does not ensure an exhaustive exploration of the search space and must be mixed with the simpler previously described mutation.

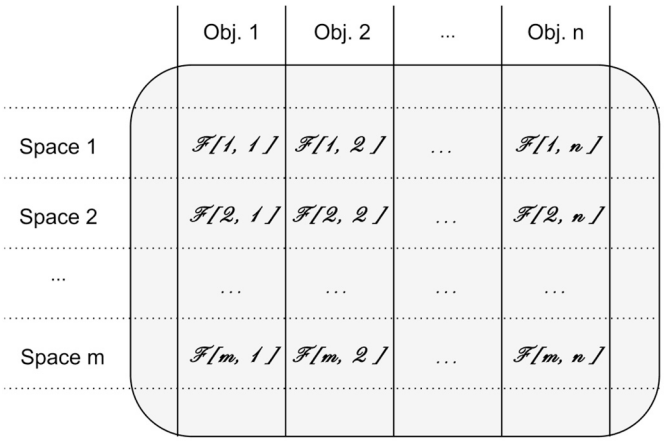


Fig. 16. Multi-dimensional Fitness represented as an m by n matrix, m = the number of rooms of the floor plan and n = the number of objectives sought-after objectives.

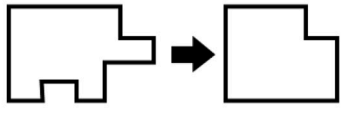
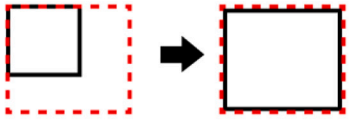
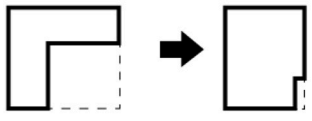
The mutation of a floor plan proceeds as follows:

1. a random room is picked (with a higher probability for the rooms with the worst fitness).
2. a random edge is picked among the edges of the border room whose edge pair points to another mutable room (e.g. not a null space, not a duct, not a load-bearing wall).
3. a mutation is selected from the four possible mutations described in Fig. 18 and applied to the floorplan. If the room has an area superior to the specification target, a higher probability is assigned to the removal of a cell or a row of cells. (See Figs. 19–21.)

7.2.3. Crossover

The crossover of two individuals is an essential part of the genetic algorithm as it allows to explore faster new regions of the search space by combining different parts of two floor plans. Intuitively, it can be thought of as the process by which we take the best rooms of two floor plans and combine them together to hopefully create a better solution. It is especially useful when optimizing larger apartments where rooms distant from another are more decoupled. In this context, some individuals might be close to the optimum in different partial areas of the floor plan. Combining them in an efficient manner brings the search closer to the global optimum.

However, implementing such a crossover presents the difficult

Corners	Areas	Convexity
The number of corners of the room	The absolute difference between the area of the room and the desired area established in the specification	The difference between the area of the room and its rectangular bounding box
		
F(s, 1) = Abs(Cardinal[angles between consecutive edges > 35] - 4)	F(s, 2) = max(min_specified - Area(s), Area(s) - max_specified, 0) / Area(s)	F(s, 3) = Area(s) - Areas(BoundingBox(s)) / Area(s)

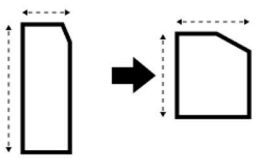
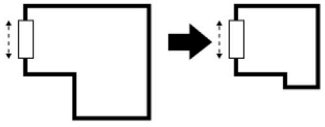
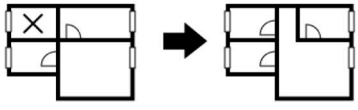
Depth Ratio	Window Area Ratio	Connectivity
The ratio between the width and the depth of the rectangular bounding box of the room.	The ratio between the area of the windows openings and the area of the room floor	A penalty per each room that is not connected to the entrance of the floor plan through a corridor or a circulation space (cf. part 0)
		
F(s, 4) = bounding_box_depth / bounding_box_width	F(s, 5) = Sum(Area(windows)) / Area(room)	F(s, 6) = # of disconnected rooms * penalty

Fig. 17. The six objectives sought-after in the optimization.

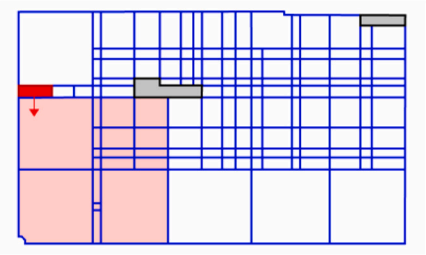
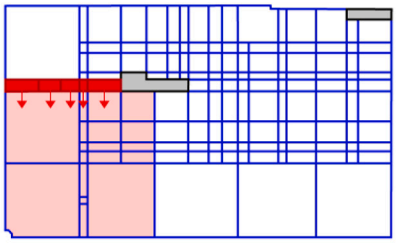
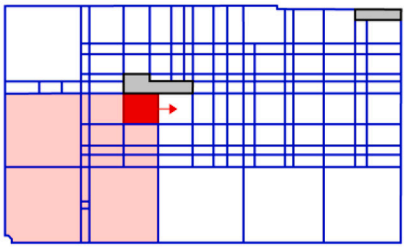
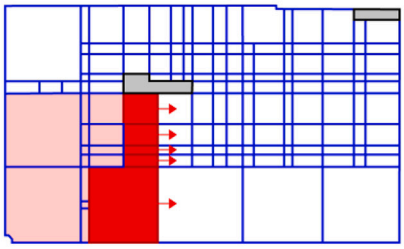
Add a cell	Add a row of cells
	
A random edge on the border of the room is picked. The cell of its edge pair is removed from the adjacent room and is assigned to the room.	A random edge on the border of the room is picked with all the adjacent edges aligned with it. The cells of the edges pair are removed from their corresponding room and assigned to the room.
Remove a cell	Remove a row of cells
	
A random edge of the border of the room is picked, its corresponding cell is removed from the room and assigned to the adjacent room of the edge pair.	A random edge is picked with all its adjacent and aligned edges. The corresponding cells are removed from the room and assigned to the adjacent room of the edge pair.

Fig. 18. The four mutations applied to the floor plan.

challenge of mixing two floor plans without "corrupting" the solution by breaking a room into disconnected components, completely removing a room or breaking the interior circulation between rooms. We defined a crossover method that preserves by design the first two constraints and we treat the third constraint via a corresponding death penalty in the fitness calculation (cf. objective 6 of Fig. 17). Our crossover technique consists in selecting in each parent a specific room and copying over each of these two rooms to the other individual. The rooms are selected randomly with weighted probabilities proportional to their fitness difference with their room counterparts in the other parent. A room that has a worse fitness than its counterpart is given a weight of zero. If the crossover corrupts the children by splitting or removing a room, the operation is aborted and the children are computed by applying a mutation to their parents.

Room A is selected from Parent 1 and is copied into Parent 2 to create Child 2. Room B is selected from Parent 2 and is copied into Parent 1 to create Child 1(cf. Fig. 19).

7.2.4. Selection

The goal of the selection phase is to select the best individuals while maintaining a sufficient diversity in the genetic pool. As we are trying to optimize multiple objectives as described in 7.2.1. At each generation, among the population comprised of parents and children, only the individuals belonging to the Pareto optimal front are kept.[2]

To maintain as much diversity as possible, all identical individuals are removed and replaced by mutations of other selected individuals. If the size of the Pareto front exceeds the population size, which happens frequently due to the high number of dimensions, an elitist strategy is used and the individuals of the pareto front with the worst weighted fitness are removed. The algorithm differs from NSGA-II in this regard and the crowding distance is only used in tournament selection before the crossover phase.[3] The domination relationship between individuals is based on the multi-dimensional array comprised of the six objectives

[2] Our algorithm is based on a pareto domination comparison in a similar fashion as NSGA-II (20). An individual is said to be pareto dominated if another individual has a fitness equal or better on every or better on every dimension and strictly better on at least one dimension. The set of all the non-dominated individuals form the Pareto optimal front.

[3] We used binary tournament selection based on dominance to select individuals for crossover, as implemented in the DEAP library (22). The population is sorted according to the crowding-distance and split in four equal groups. Each individual is randomly matched against an individual of another group, and the winner is selected for crossover. Each individual participates in a tournament exactly twice, the number of individuals selected for the crossover is therefore equal to the length of the initial population.

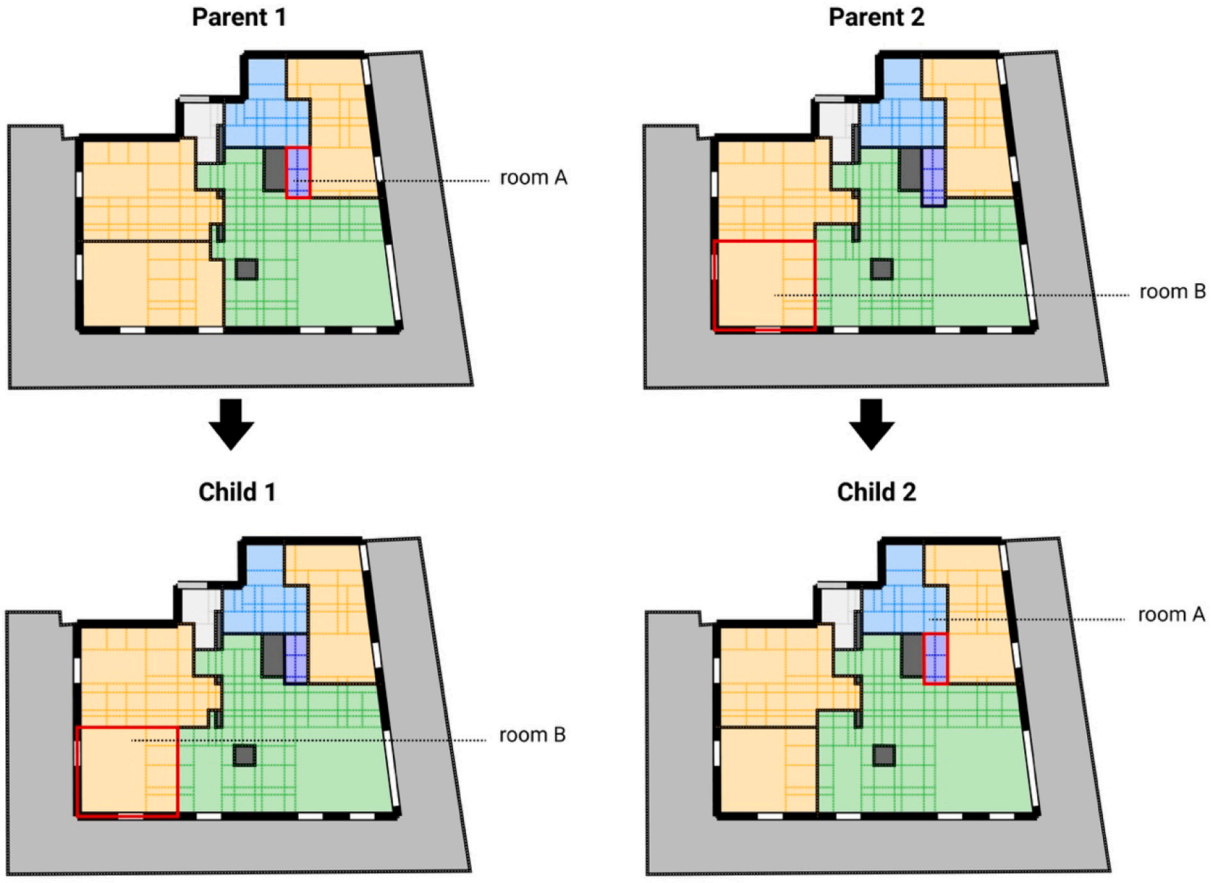


Fig. 19. Illustration of the crossover of two individuals (Parent 1 and Parent 2) leading to the creation of two new individuals (Child 1 and Child 2).

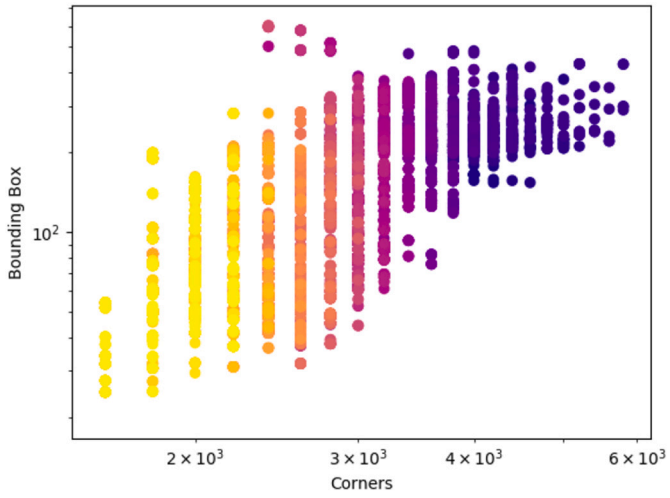


Fig. 20. Fitness values for a set of 120 individuals representing 68 generations. The colour indicates the generation number, blue for the first generations and yellow for the last generations (plan 4 - appendix). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

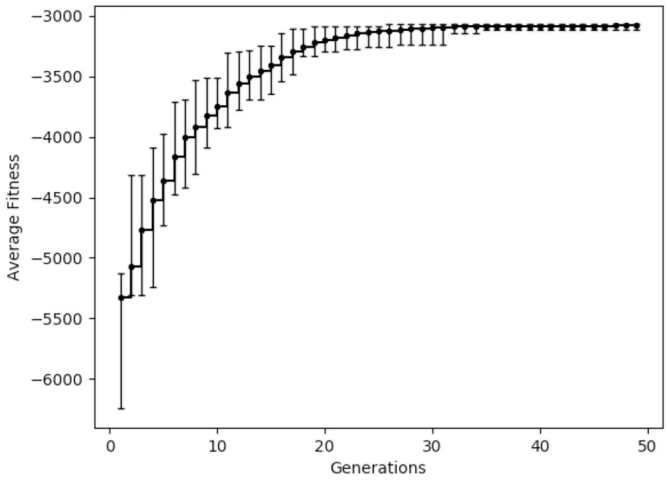


Fig. 21. Fitness average, minimum and maximum values for 50 generations and 25 runs (plan 4 - appendix).

described in 7.2.1 summed over all the rooms.

The final output is selected by returning the individual with the highest global fitness, defined as the weighted sum of the fitness values over all rooms and objectives. The weights given to each objective have therefore a significant influence on the output, both for the final selection of the best individual and for the elite selection at each generation (Fig. 20 - 21).

8. Results

Optimizer has been designed and improved through the use of a training database of sixty-two couples (plan envelope, specifications). To assess the quality of the results and ensure that we did not overlearn on this first database, we tested the algorithm on another database of nineteen couples (plan envelope, specifications), referred to as the

validation database. For each element of the test database, we also have the floor plan proposed by an architect, referred to as the reference plan. A quality criterion would be that among the plans automatically generated by Optimizer for a given couple (plan envelope, specifications), one is similar to the plan designed by the architect. But this observation is not sufficient to evaluate the results of our algorithm, so we have developed a scoring of the plans obtained.

In the following, we illustrate the results obtained on a few envelopes, propose a metric to evaluate the generated plans, compare them with those proposed by an architect, and investigate the time computation of the algorithm.

Note that Optimizer is used daily on plan envelopes from the actual building program. This allows us to evaluate its output relevance and to perform continuous improvement thanks to the feedback of our architects.

8.1. Quality of results assessment

In this section, we illustrate how two plans can be quantitatively compared and we use this metric to assess the quality of Optimizer with regard to layouts designed by our architects.

8.1.1. Scoring method

Here we propose a scoring function to evaluate a plan.

This score is a compilation of several components that we designed with an architect:

- the respect of the specified areas,
- the number of corners per room,
- the convexity of each room,
- the minimum room dimensions,
- the relative position of the rooms,
- the natural lighting of the apartment.

These 7 criteria were chosen in order to evaluate the results obtained by our algorithm in the most possible objective way. The first three components are already used and described in the final cell allocation optimization section (see section 7.2.1) while the last three ones are detailed in Fig. 22.

The final score is an average of these components.

8.1.2. Validation database results

Example of Plan n°4 scoring results :

Our algorithm has not succeeded in finding exactly the reference plan, but Solution 1 is very close to it (Fig. 23). We observe that the three solutions proposed by our algorithm are very different from each other, which offers an interesting variety of solutions. Note that, as expected, the reference plan has the highest score. Among the generated solutions, the one that is more akin to the reference plan has the best score.

To perform a deeper comparison between those plans, we can distinguish the performance obtained on all scored criteria, as done in the following radar chart.

As expected, we can observe (Fig. 24) that the reference apartment plan has the highest score on every criterion except for the one on the relative position of the rooms. The reason is that there is more proximity between the bedrooms and the bathroom and between the living room and the entrance door in Solution 2 than in the reference plan. This although the toilets are all the more appropriate in the reference plan as they are closer to the entrance. The significant differences in the quality of the plans especially lie in the number of corners in the rooms, the position of the rooms within the plan and the minimum recommended dimensions.

Some examples of results on the validation database are given in the appendix, architect plan scores and Optimizer best solution scores can be compared.

The layout proposed by the architect is not always obtained among the solutions proposed by Optimizer. Among the nineteen architect plans of the validation database, four are proposed by Optimizer. This is not an imperative objective as long as Optimizer provides architecturally valid alternatives. However, it is interesting to consider why this is so.

Several characteristics of our algorithm can account for it:

- As explained in Part 4., the constraint solver may not find every solution because the time limit has been reached. This is for instance what happened with Plan n°9.
- In the constraint programming step, we impose some subjective constraints in order to limit the number of non-relevant solutions. See for example the rules on adjacency between rooms which have a functional meaning but which can limit space optimization in some cases.

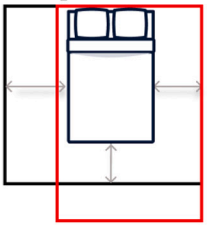
Minimum room dimensions	Relative position of the rooms	Natural lighting of the apartment
To be furnished, rooms of a given category shall have minimum dimensions. Each room is given a binary score (0 if minimal dimensions are not respected, 100 else) and the resulting average score is attributed to the layout. Example of master bedroom: 	Basic rules for positioning the rooms are checked, for example, the proximity of one of the toilets to the entrance door, the proximity of the bedrooms with a bathroom, easy access to the living room from the entrance... Each room is given a score and the resulting average score is attributed to the layout.	We assess the natural lighting of the apartment. To do so, each cell of the grid is said to be enlightened if a straight line can be drawn from this cell to a window without intersecting a wall and if the distance from the cell to the window is shorter than a given distance. The layout score is then computed based on the area proportion that receives natural light.

Fig. 22. The three new scoring components of the final scoring.

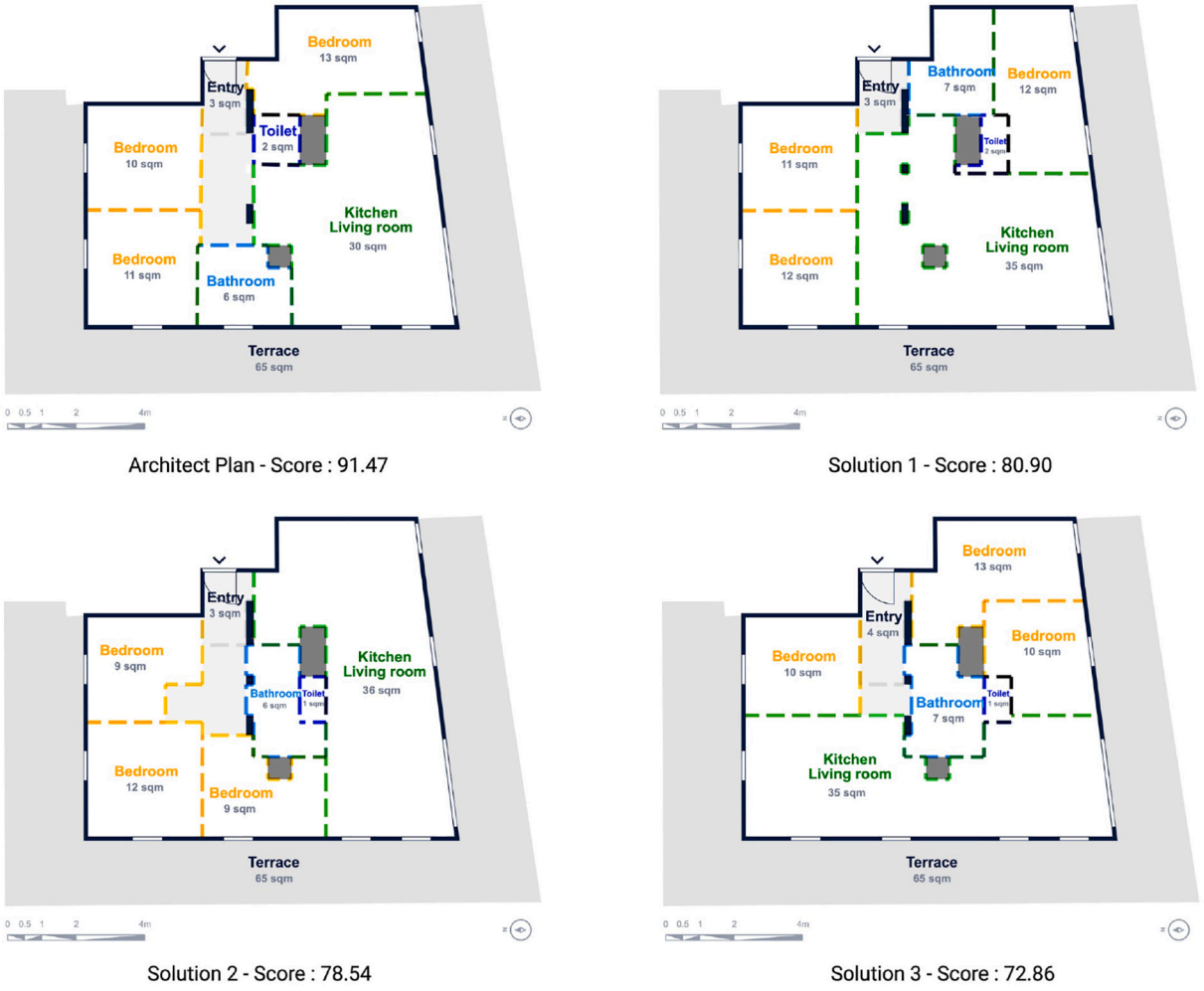


Fig. 23. P lan proposed by an architect for a given couple (plan envelope, specifications) and the three final solutions proposed by Optimizer along with their score.

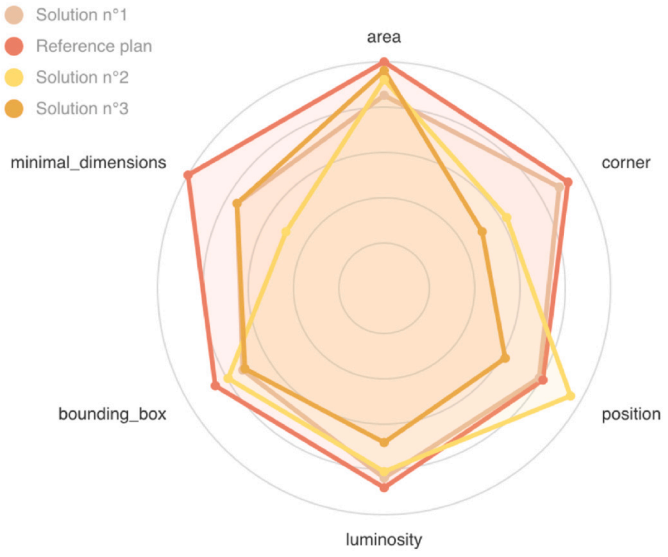


Fig. 24. Scoring of the reference plan and optimizer solutions (Fig. 23) with detail on each component. The score is a compilation of the six criteria. Scores obtained by the architect plan are illustrated by the red line. Others are Optimizer results. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

- In the constraint programming part, we imposed that water rooms touch a duct (see part 4). Architects, though, regularly place water rooms a few meters away from the duct (see Plan n°1, where, in the layout proposed by an architect, the laundry does not have direct contact with the duct).
- The walls of the rooms drawn by the architect may not correspond to lie among the grid edges. This mainly happens for non-rectilinear envelopes. Let us, for instance, consider Plan n°3 (see appendix). The layout proposed by optimizer has two significant limitations: the circulation area has a pillar in its center, and bedroom shapes are not optimized since non-rectilinear angles make them difficult to furnish.

8.2. *Computation time*

We aim at reaching real-time generation of solutions. To do so, we made some sharp choices in the design of Optimizer. Our aim is not to obtain every possible plan for a given envelope but rather to propose a variety of relevant alternatives in less than one minute.

The Table 4 below presents statistics on Optimizer process times for our test and validation databases.

Statistics on Optimizer computation time as a function of the plan area and number of rooms specified are shown Figs. 25 and 26. They are computed on five hundred layouts generated on for real use cases. To make the comparison relevant, we only considered cases for which Optimizer generates one solution only.

As expected, the average and variance of the computation time increases with the apartment surface area and the number of required

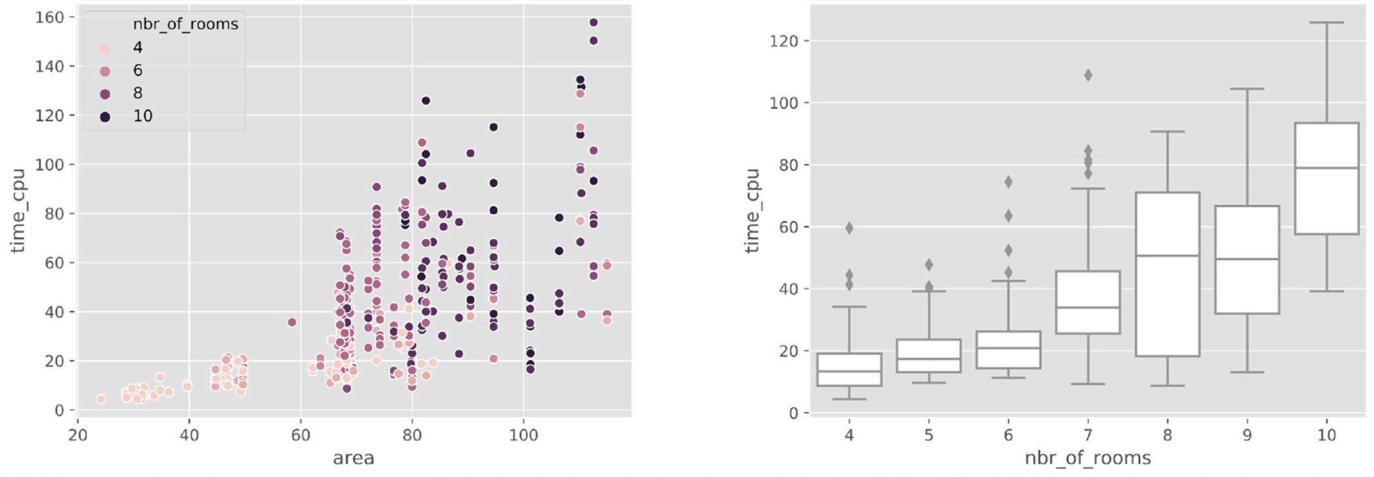


Fig. 25. Computation time in relation to both the apartment surface area and the specified number of rooms.

Fig. 26. Processing time repartition: Average computation time for each part of optimizer considering the input specification number of rooms.

Table 4
Databases running times.

	Training plan database	Validation plan database
Average processing time	57 s	1 min 21 s
Minimum processing time	2 s	32 s
Maximum processing time	4 min 36 s	3 min 37 s

rooms. High variances make computation times unpredictable. Moreover, on all the plans the most time-consuming part is the final optimization even if constraint programming takes a significant part of the computation time on the large plans.

We designed this algorithm so as to generate floor plans for apartments sold in French residential new developments, those rarely contain more than ten rooms. Therefore, we did not investigate the behavior of the algorithm for a higher number of rooms. The computation time is indeed greatly affected by the number of rooms, in case it is too high new positioning constraints may have to be introduced to prevent the search space from growing exponentially.

Several options could be explored to reduce computation time:

- If we limit ourselves to a single valid solution, we can stop at the first solution proposed by constraint programming. We would then reduce the constraint programming part and get rid of the selection part.
- The grid generation and grouping of cell parts can be pre-computed offline. They are associated with the plan envelope only and will not change with the required specifications.
- After the selection step, layouts can be processed in parallel.
- Our algorithm was developed in python; another language such as C# might be more adequate when considering the processing time.

9. Conclusion

In conclusion, we proposed in this note an algorithm for automatic apartment plan generation. Given a plan envelope and specifications, it provides various layout alternatives within a short computing time. We validated our approach by comparing the results obtained with plans proposed by architects; this leads to promising perspectives.

Optimizer proposes layout options to our customers every day, based on their needs. It allows going one step further in the customization process. At present, generated layouts are not accurate enough to be proposed as sales plans, but they allow clients to project themselves and they can be used as an advanced step towards the final plan.

Future developments include several improvements. We are considering some improvements on the grid in non-rectilinear cases. Another point we should tackle is the positioning of corridors, as the refiner may propose plans where the walking path is not optimal in terms of shape and area. Also, the boundary between the living room and the circulation spaces is not always well managed.

The processing of large apartments by our algorithm may be too time-consuming and should be improved as well. We could do so by including restrictions on room positioning according to the apartment orientation, or by adding an intermediate notion of a group of rooms to tighten the constraint programming model.

We are currently working on the placement of the doors and on the automatic positioning of equipment inside the rooms (beds, bathtub or kitchen elements).

Declaration of Competing Interest

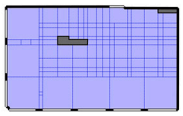
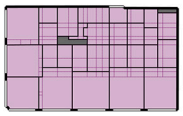
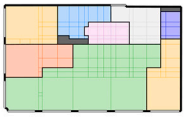
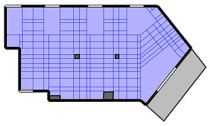
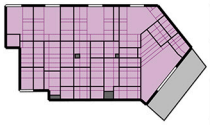
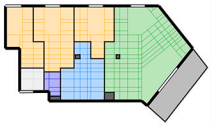
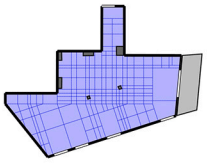
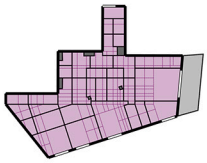
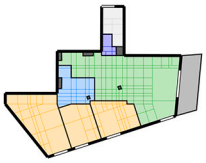
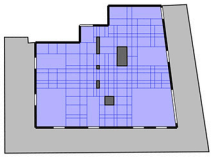
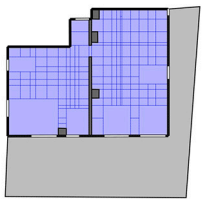
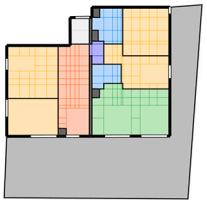
The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The authors are grateful to all HABX team for their support, especially Florent Clairambault and Jean Clairambault for their advices to improve the manuscript. This work was partially financed by a grant of Bpifrance(France) obtained during a digital innovation competition (Concours d'Innovation Numérique). The illustration plans were inspired by our collaborations with AAVP Architecture, Antonio Virga Architecte, Alternative Architecture, Architectonia, Armand Nouvet Architecture et Urbanisme, DATA Architectes, Ateliers 2/3/4/, AZC Architectes, DGM & Associés, Cynthia Voisin Architecte, Jard & Brychcy Architecture, PetitdidierPrioux Architectes, Wilmotte & Associés Architectes.

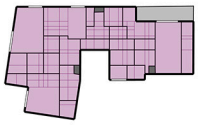
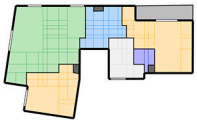
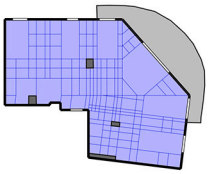
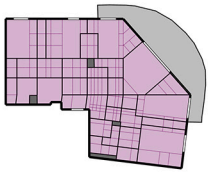
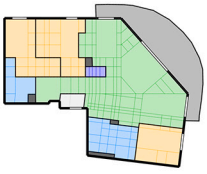
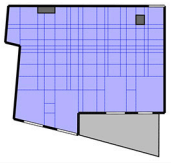
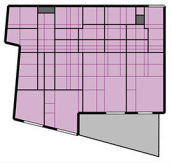
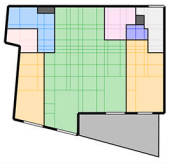
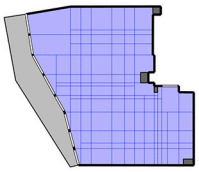
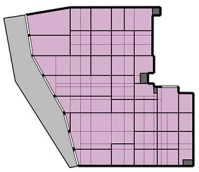
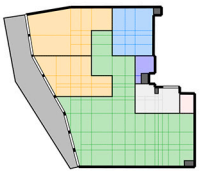
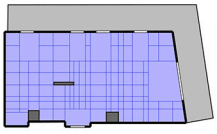
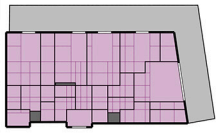
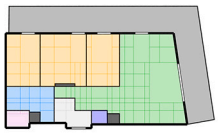
Appendix A

Sample of results obtained by optimizer on the validation database
Part 1

	Grid	Grouping of cells	Best solution	Best solution	Architect Plan	Results
1						Architect plan score : 68.46 Best solution score : 87.61 Computation time : 1 min 26s
2						Architect plan score : 87.69 Best solution score : 82.47 Computation time : 1 min 14s
3						Architect plan score : 90.54 Best solution score : 62.30 Computation time : 1 min 35s
4						Architect plan score : 91.47 Best solution score : 80.90 Computation time : 2 min 36s
5						Architect plan score : 83.78 Best solution score : 74.56 Computation time : 1 min 04s

Appendix B

Sample of results obtained by optimizer on the validation database
Part 2

	Grid	Grouping of cells	Constraint programming best solution	Best solution	Architect Plan	Results
6						Architect plan score : 85.23 Best solution score : 79.22 Computation time : 1 min 18s
7						Architect plan score : 86.36 Best solution score : 84.44 Computation time : 2 min 12s
8						Architect plan score : 82.75 Best solution score : 70.30 Computation time : 1 min 06s
9						Architect plan score : 82.75 Best solution score : 70.30 Computation time : 2 min 03s
10						Architect plan score : 83.34 Best solution score : 77.22 Computation time : 54s

References

[1] Mitchell Stiny, The Palladian grammar, Environ. Plan. 5 (1978) 5–18, https://doi.org/10.1068/b050005.

[2] Calixto, Celani, A Literature Review for Space Planning Optimization Using an Evolutionary Algorithm Approach: 1992-2014, 2015, pp. 662–671, https://doi.org/10.5151/despro-sigradi2015-110166.

[3] Medjdoub, Yannou, Separating topology and geometry in space planning, Comput. Aided Des. 32 (2000) 39–61, https://doi.org/10.1016/S0010-4485(99)00084-6.

[4] Charman, A constraint-based approach for the generation of floor plans, in: Proceedings Sixth International Conference on Tools with Artificial Intelligence, 1994, pp. 555–561, https://doi.org/10.1109/TAI.1994.346443.

[5] Li, Frazer, Tang, A Constraint Based Generative System for Floor Layouts, the Fifth Conference on Computer Aided Architectural Design Research in Asia / ISBN 981-04-2491-4 Singapore 18-19 May 2000, 2000, pp. 441–450. http://papers.cumincad.org/data/works/att/5b5d.content.pdf (Accessed date: 24 october 2020).

[6] Puret, Slimane, Projet HM2PH, Génération Automatique de Plans et Visite Virtuelle D'habitats Adaptés pour Personnes Handicapées (HM2PH Project, Automatic Generation of Plans and Virtual Tour of Adapted Housing for Disabled People). https://www.theses.fr/2007TOUR4024, 2007.

[7] Wang, Yang, Zhang, Customization and generation of floor plans based on graph transformations, Autom. Constr. 94 (2018) 405–416, https://doi.org/10.1016/j.autcon.2018.07.017.

[8] Upasani, Shekhawat, Sachdeva, Automated generation of dimensioned rectangular floorplans, Autom. Constr. 113 (2020) 103–149, https://doi.org/10.1016/j.autcon.2020.103149.

[9] Shi, Soman, Han, Whyte, Addressing adjacency constraints in rectangular floor plans using Monte-Carlo tree search, Autom. Constr. 115 (2020) 103–187, https://doi.org/10.1016/j.autcon.2020.103187.

[10] Wu, Fan, Liu, Wonka, MIQP-based layout Design for Building Interiors, Comp. Graphics Forum 37 (2018) 511–521, https://doi.org/10.1111/cgf.13380.

[11] Merrell, Schkufza, Koltun, Computer-Generated Residential Building Layouts. Association for Computing Machinery Transactions on Graphics. Article No.: 181, 2010, https://doi.org/10.1145/1866158.1866203.

[12] Camozzato, Dihl, Silveira, et al., Procedural floor plan generation from building sketches, Vis. Comput. 31 (2015) 753–763, https://doi.org/10.1007/s00371-015-1102-2.

[13] Wu, Fu, Tang, Wang, Qi, Liu, Data-driven interior plan generation for residential buildings, in: Association for Computing Machinery Transactions on Graphics Article No.: 234, 2019, https://doi.org/10.1145/3355089.3356556.

[14] Duarte, Jose, A discursive grammar for customizing mass housing: the case of Siza's houses at Malagueira, Autom. Constr. 14 (2) (2005) 265–275, https://doi.org/10.1016/j.autcon.2004.07.013.

[15] Veloso, Celani, Scheeren, From the generation of layouts to the production of construction documents: an application in the customization of apartment plans, Autom. Constr. 96 (2018) 224–235, https://doi.org/10.1016/j.autcon.2018.09.013.

[16] Michalek, Choudhary, Papalambros, Architectural layout design optimization, Eng. Optim. 34 (2002) 461–484, https://doi.org/10.1080/03052150214016.

[17] Elezkurtaj, Franck, Algorithmic Support of Creative Architectural Design. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.121.3050, 2002 (Accessed date: 24 october 2020).

[18] Verma, Thakur, Architectural Space Planning Using Genetic Algorithms, The 2nd International Conference on Computer and Automation Engineering (ICCAE), Singapore, 2010, pp. 268–275, https://doi.org/10.1109/ICCAE.2010.5451497.

[19] Doulgerakis, Genetic and Embryology in Layout Planning, University College London, 2007. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.135.2409&rep=rep1&type=pdf (Accessed date: 24 october 2020).

[20] Nauata, Chang, Cheng, Mori, Furukawa, House-GAN: Relational Generative Adversarial Networks for Graph-constrained House Layout Generation. https://arxiv.org/abs/2003.06988, 2020.

[21] Hu, Huang, Tang, van Kaick, Zhang, Huang, Graph2Plan : Learning Floorplan Generation from Layout Graphs. Association for Computing Machinery Transaction on Graphics Article 118, 2020, https://doi.org/10.1145/3386569.3392391.

[22] Or-Tools User's Manual - Breaking symmetries with Symmetry Breakers. https://acrogenesis.com/or-tools/documentation/user_manual/manual/search_primitives/breaking_symmetry.html, 2014 (Accessed date: 24 october 2020).

[23] Perron, Furnon OR-Tools 7.2. https://developers.google.com/optimization, 2020 (Accessed date: 24 october 2020).

[24] Pedregosa, et al., Scikit-learn: machine learning in python, J. Mach. Learn. Res. 12 (2011) 2825–2830. http://jmlr.org/papers/v12/pedregosa11a.html (Accessed date: 24 october 2020).
[25] Nagy, Lau, Locke, Stoddart, Villaggi, Wang, Zhao, Benjamin, Project discover: an application of generative design for architectural space planning, in: Proceedings of the Symposium on Simulation for Architecture and Urban Design Article 7, 2017, pp. 1–8, https://doi.org/10.22360/simaud.2017.simaud.007.
[26] Deb, Pratap, Agarwal, Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, IEEE Trans. Evol. Comput. 6 (2002) 182–197, https://doi.org/10.1109/4235.996017.
[27] A. Fortin, M. De Rainville, Gardner, Parizeau, Gagné, DEAP: Evolutionary algorithms made easy, J. Mach. Learn. Res. 13 (2012) 2171–2175, https://doi.org/10.5555/2503308.2503311 (Accessed date: 24 october 2020).