Automation in Construction 119 (2020) 103288

Contents lists available at ScienceDirect

Automation in Construction

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

Generating layout designs from high-level specifications

Xiao-Yu Wang, Kang Zhang*

The University of Texas at Dallas, 800 W Campbell Rd, Richardson, TX, USA

ARTICLE INFO

Keywords:
Floor planning
Architecture
Computer-aided design
Reserved Graph Grammar
Graph manipulation

ABSTRACT

This paper presents a framework for the automatic generation of floor plans based on adjacency relations among rooms. The adjacency can be generated from user-specified design requirements using a graph grammar formalism. We propose a set of grammar rules to generate graphs that represent adjacency relationships. Our solution overcomes the limitation of previous approaches that generate only rectangular floor plans. We define a set of constraints, such as plan size, room orientation and aspect ratio, for specifying the desired floor plans; and present a set of algorithms for placing rectangular or non-rectangular rooms and for generating non-rectangular floor plan boundaries. We demonstrate that our method can generate varied floor plans from user-specified design requirements.

1. Introduction

Floor plan, as a fundamental architectural diagram, is one of the various representation tools used by architects to design buildings. Since architects may take days or even weeks to manually draw a floor plan, it is highly desired to automate the design process. It is, however, a challenging task to generate floor plans fulfilling design requirements. There are several existing approaches for automatic generation of floor plans. Stiny and Gips [1] first proposed a rule-based approach, called *Shape Grammar*, for generating designs from given initial shapes and transformation rules. Shape grammar is later used by researchers to generate floor plans in different styles, such as Palladian houses [2], Mughul gardens [3], Frank Lloyd houses [4], traditional Turkish houses [5] and Siza's houses [6]. Roth et al. [7] generated rectangular floor plans with non-convex envelopes. Kahng [8] proposed an approach to generate floor plans by placing components into a space. Merral et al. [9] generated building layouts based on constraints from a Bayesian network.

None of the above approaches is able to generate floor plans that meet users' specific requirements. For example, some rooms should have windows, while others may be of non-rectangular shapes due to their special functions. A common challenge for any floor plan generation approach is how to automate the interpretation of the user requirements and transform them into a generic data structure suitable for floor plan generation. Wang et al. [10] proposed a graph approach to floor plan generation (called *GADG*) based on graph structures. The downside of that approach is that an existing floor plan has to be provided to the system and rooms in generated plans need to be rectangular. Designers have limited control of the generated floor plans.

As one of the earliest approaches solving the space allocation problem by the division of rectangles, Flemming [11] generates dissection of rectangles into rectangular components while satisfying topological and dimensional constraints. This method describes a floor plan by summarizing topological relations of walls along with other constraints, e.g. required room adjacency and total area. A layout generation system [12] was developed based on this method and used in many applications, e.g. space navigation [13]. Similar to Flemming's approach, our framework extracts user-specified requirements and abstracts them as a graph data structure to generate floor plans. With the user-specified requirements, our framework uses a formal grammatical method to generate graphs satisfying such requirements. Instead of summarizing the topological relations of walls, we represent the topological relations of rooms as graphs. Comparing with Flemming's approach, our framework simplifies the representation format and clearly specifies the adjacency of rooms. We also overcome the limitation of only generating rectangular rooms.

Assume that the user has in mind a set of design requirements for his/her house as the following, with an example model shown in Fig. 1:

- Floor size of 11 m*13 m;
- 3 bedrooms and 2 bathrooms (2 bedrooms maybe of non-rectangular shape);
- Aspect ratio of all rooms except the hall way ≤ 2.5;
- Living room facing south.

This paper presents our approach capable of generating floor plans

* Corresponding author.
E-mail address: kzhang@utdallas.edu (K. Zhang).

https://doi.org/10.1016/j.autcon.2020.103288
Received 24 January 2020; Received in revised form 18 April 2020; Accepted 28 May 2020
0926-5805/ © 2020 Elsevier B.V. All rights reserved.

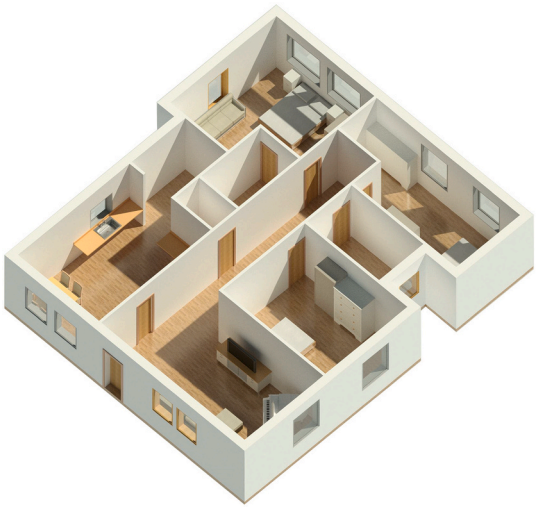

Fig. 1. A floor plan satisfying a set of design requirements.

with non-rectangular rooms from input requirements, such as those above.

Fig. 2 shows the process of generating floor plans, using the aforementioned design requirements as a specific example. The user's input requirements are given in the form presented in Fig. 2(a), from which the graph in Fig. 2(b) is automatically generated using a graph grammar formalism followed by a graph triangulation algorithm. Each graph vertex represents a room and each edge between two vertices represents that the corresponding two rooms are adjacent. To generate floor plans with non-rectangular rooms and boundary, we artificially insert additional vertices (blue ones for non-rectangular boundary and red ones for non-rectangular rooms) as in Fig. 2(c), to be identified by our generation algorithm, while retaining the rooms' adjacency relationships. Fig. 2(d) shows a generated floor plan. Although we manually place furniture in the generated plan to merely indicate each room's function, many automatic furniture placement approaches, e.g. [14], could be used instead.

This paper presents our graph grammar approach that extracts user-specified requirements and abstracts them as a graph data structure to represent the adjacency among rooms. We proceed with our floor plan generation algorithms that handle special requirements, such as non-rectangular boundaries and rooms.

This paper makes the following contributions:

- A formal mechanism specifying high-level constraints for generating floor plans.
- A generic approach for generating floor plans from user-specified requirements, via a graph data structure generated by a graph grammar formalism.
- A set of algorithms for placing rectangular or non-rectangular rooms from graphs and for generating non-rectangular boundaries.

The rest of this paper is organized as follows. Section 2 reviews the related works. Section 3 explains the definitions and concepts used in our approach, including the Reserved Graph Grammar (RGG) formalism, the Properly Triangulated Planar (PTP) graph, and the Rectangular Dual Graph (RDG). Section 4 introduces our RGG approach to generate adjacency graphs based on user-specified requirements. Section 5 describes the floor plan generation process and our room placing algorithms. Section 6 introduces constraints implemented in our approach to generate floor plans. Next, several experiments are demonstrated in Section 7. Section 8 concludes the paper and mentions future works.

2. Related work

A number of systems and approaches have been proposed to

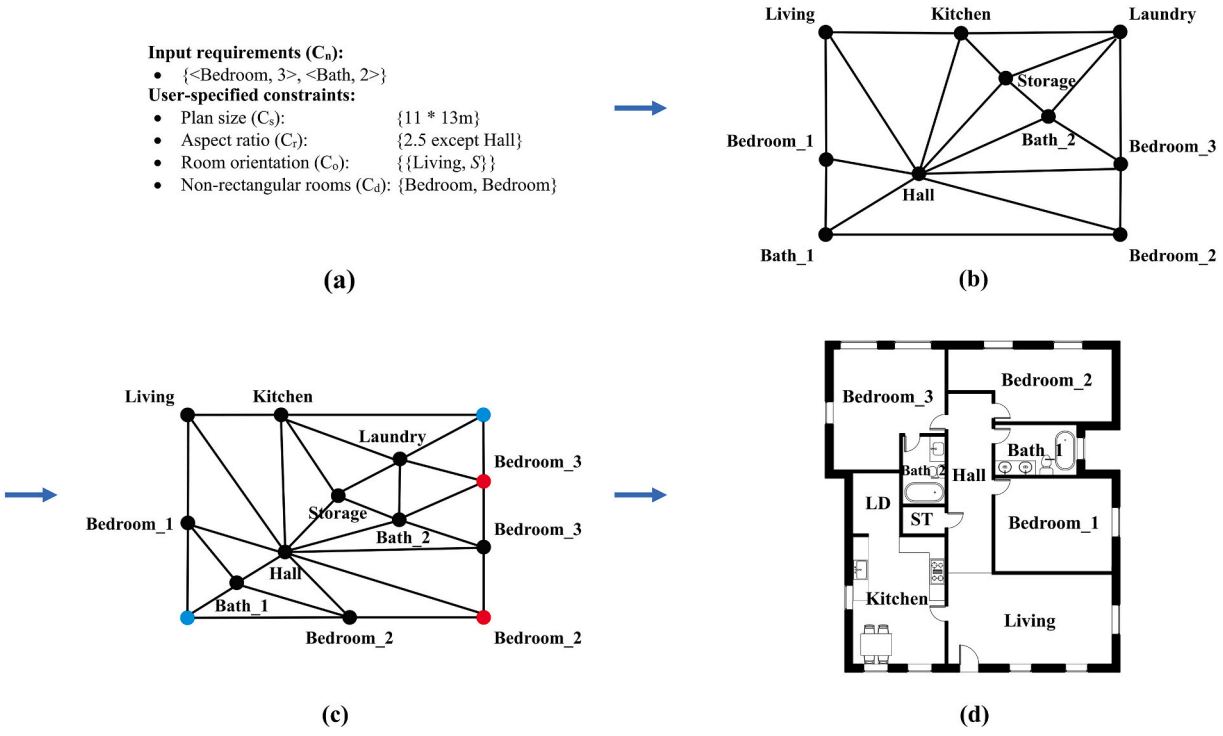

Fig. 2. The floor plan generation process. (a) User-specified design requirements with parametric constraints. (b) An automatically generated graph from the input requirements. (c) Two blue vertices and two red vertices (with duplicate labels) inserted to inform the generation algorithm about the non-rectangular boundary and non-rectangular rooms respectively. (d) a generated plan from the graph in (c), meeting the requirements in (a). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

generate building designs automatically. Some researchers place shapes into given planes, by solving the space allocation problem. Peng et al. [15] introduced a linear integer programming method to generate layouts. Shekhawat [16] proposed an explicit algorithm to generate rectangular floor plans inside given rectangles. Wu et al. [17] developed a system that automatically generates layout designs with high-level constraints based on a mixed integer quadratic programming (MIQP) formulation.

Other researchers focus on generating floor plans from given graphs. Martin [18] developed a method to generate houses in three steps. This method first generates graphs representing the adjacencies among rooms. Then, rooms are placed and expanded to proper sizes using the Monte Carlo method. Marson and Musse [19] proposed a technique to generate floor plans from the hierarchical information specified by tree-maps. Rooms are generated by splitting parent spaces according to the given tree structures. Instead of representing adjacencies among rooms, edges on tree-maps present hierarchical relations.

Kozminski and Kinnen [20] created an $O(n^2)$ algorithm to construct rectangular floor plans from planar graphs. A validation algorithm is given to verify the existence of rectangular plans of given input graphs. Bhasker and Sahni [21] later proposed an O(n) algorithm to generate rectangular floor plans from triangulated planar graphs. Based on this algorithm, Wang et al. [10] introduced an approach to generate rectangular floor plans from room adjacencies obtained from existing plans. These approaches, however, only generate rectangular rooms based on existing graph structures.

Shape grammar was first proposed by Stiny and Gips [1] as a rule-based approach to modeling paintings and sculptures. It has since been extended to other areas, e.g. architecture designs [2], pattern designs [22] and industrial designs [23]. Researchers developed shape grammar rules to generate floor plans in different styles, e.g., Çağdaş [24] proposed shape grammar rules to generate English row-houses. These approaches, however, cannot generate plans from high-level specifications.

Wonka et al. [25] created a split grammar to generate building designs by recursively splitting existing shapes into sub-shapes. Their generation process is controlled by an attribute matching system and a separate control grammar. Merrell et al. [9] combined machine learning and optimization techniques with architectural methodologies to generate floor plans based on data sets trained from a Bayesian network. Shekhawat [26] recently proposed a generic approach to construct rectangular floor plans by enumerating all possible floor plans with the same adjacency.

Graph grammar is well studied for modeling concepts and structures in a 2D fashion. It is a rule-based approach which derives or verifies graphs following given graph transformation rules, called *productions*. A graph grammar could be of either context-free or context-sensitive type. The left graph of a context-free grammar, such as node label controlled grammar [27] and relational grammar [28], has only one non-terminal node. In contrast, a context-sensitive graph grammar has a stronger expressive power by allowing the left graph to be any node-edge graph. In recent 20 years or so, researchers have been focusing on the theoretical aspects and applications of context-sensitive graph grammars, such as, layered graph grammar (LGG) [29], Reserved Graph Grammar (RGG) [30], spatial graph grammar (SGG) [31] and edge-based graph grammar (EGG) [32].

With the enhanced expressive power, researchers proposed a large number of applications using context-sensitive graph grammars. Zhang et al. [33] proposed a visual approach to XML document design and transformation using RGG. Zhao et al. [34] presented a graph grammar approach to the discovery and verification of behaviors and functionalities of software. Roudaki et al. [35] introduced a framework, which discovers and validates web patterns using SGG. These applications use graph grammars' reduction capability by validating application graphs via user-specified productions. Researchers also combined graph grammar and shape grammar [36] [37] in generating floor plans by exhaustively applying graph productions. These approaches, however, cannot generate plans based on specific design requirements.

We [38] recently explored the derivation ability of graph grammar by generating Turkish houses. We also [39] proposed a set of productions for generating path graphs of floor plans under the style of Frank Lloyd Wright houses. The generated path graphs, however, cannot be used directly to generate floor plans. The research presented in this paper focuses on developing a generic framework for generating floor plans satisfying various user-specified requirements.

3. Concepts

Before explaining our method, this section introduces the terminologies first.

3.1. A graph grammar formalism

Graph grammar, extended from string grammar, is a 2D formal method capable of specifying various Visual Programming Languages (VPLs) and other graph-based applications [40]. Our approach uses the Reserved Graph Grammar formalism, or RGG [30], to generate graphs from user-specified requirements. RGG is defined based on grammatical rules, called *productions*. Each production consists of left-hand and right-hand graphs. For simplicity, we call them *left graph* and *right graph*. *Derivation* and *reduction* are two basic workflows in graph grammars. Derivation is a process of *L-application*, which uses a production's right graph to replace the sub-graph in the input graph that is isomorphic to the left graph of the production. This sub-graph is called the *redex* and the input graph is called the *host graph* in RGG. A redex is isomorphic to the right graph in a R-application or the left graph in an L-application. On the other hand, the reduction process is a *R-application* that replaces the redex of the right graph with the left graph. These two workflows of graph grammar provide the ability of both generating and parsing designs with given productions [38].

RGG introduces a marking mechanism to represent the context. Comparing with other context-sensitive graph grammars, it greatly simplifies the production format and reduces the number of productions by avoiding ambiguity in the generation and parsing process. Therefore, we choose RGG to generate adjacency graphs from user-specified design requirements. Fig. 3 shows a node labeled "*Floorplan*" has a small rectangle, labeled "*V*", called *vertex*, that connects to other nodes via edges. In this paper, we assign one vertex to each node.

To avoid dangling edges, RGG introduces a marking mechanism in productions. Each vertex in a production can be marked by assigning a unique integer, e.g. the vertex *v* of node *BA* is marked as "1" in Fig. 4. This mechanism allows edges of a vertex to be preserved after L/R-applications. Fig. 5 shows an L-application using the production in Fig. 4. The edge connecting to *v* of *BA* is preserved after the transformation.

3.2. Floor plans as graphs

A floor plan can be considered a planar graph, $G(V, E)$, consisting of a set of vertices V and a set of edges E. We consider each wall as an edge, e, and each intersection of walls as a vertex, v. A room in a floor plan is an area bounded by its edges, also called a *face*. The *dual graph* G_d of a floor plan is a graph that has a vertex corresponding to each room and an edge joining two neighboring rooms in the floor plan. To explain our method, we formally introduce several concepts, such as a

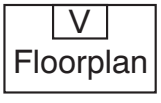

Fig. 3. A RGG node with a vertex that connects to other nodes.

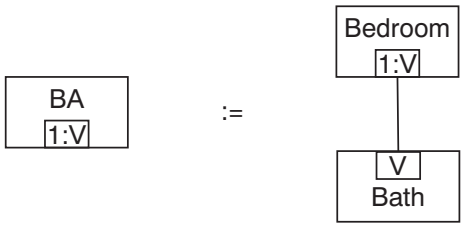

Fig. 4. A RGG production.

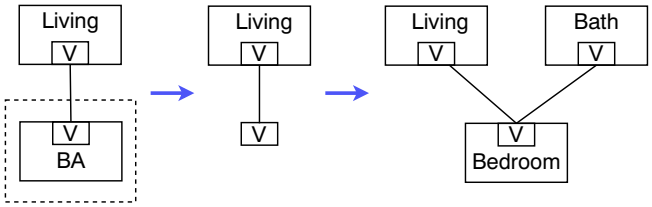

Fig. 5. An L-application, which replaces *BA* with *Bedroom* connecting with *Bath*.

planar graph, a properly triangulated planar graph and a rectangular dual graph.

Definition 1 (Planar Graph [41]). *A graph G is planar if and only if there exists a mapping of the vertices and edges of the graph into the plane such that*:

- *Each vertex is mapped into a distinct point*;
- *Each edge* (v_i, v_j) *is mapped onto a simple curve, with the vertices* v_i *and* v_j *mapped onto the endpoints of the curve*;
- *Mappings of distinct edges have only the mappings of their common endpoints in common.*

Definition 2 (Properly Triangulated Planar (PTP) Graph [21]). *A graph* G_p (V_p, E_p) *is a PTP graph if and only if it is a connected graph with a vertex set* V_p *and an edge set* E_p *such that*:

- *Every face in* G_p *is a triangle*;
- *All internal vertices have degree* ≥ 4;
- *All cycles that are not faces have length* ≥ 4.

Definition 3 (Rectangular Dual Graph (RDG) [21]). *A RDG* G_r (R_r) *of an n-vertex graph,* $G(V, E)$*, is comprised of n non-overlapping rectangles with the following properties*:

- *Each vertex* $v_i \in V$ *corresponds to a distinct rectangle* r_i *in the rectangular dual*;
- *If* (v_i, v_j) *is an edge in E, then rectangles* r_i *and* r_j *are adjacent in the rectangular dual.*

r_i and r_j are adjacent implying that they share an edge. Two rectangles sharing a node are non-adjacent. A planar graph has RDGs if and only if it is a PTP graph [20]. A RDG consists of a set of non-overlapping rectangles and preserves the adjacency from the corresponding PTP graph. Each rectangle in the RDG implies a room in a floor plan.

Definition 4 (Articulation Vertex). *A vertex v of a connected graph,* $G(V, E)$*, is an articulation vertex if its removal disconnects G.*

An articulation vertex represents a room occupying either an entire row or an entire column in a corresponding RDG. Such rooms are rare in real-world applications.

4. Graph generation using RGG

The first stage of our floor plan generation starts from generating graph structures representing the adjacency of rooms. Apart from extracting graphs from existing floor plans stored in semantic rich files as

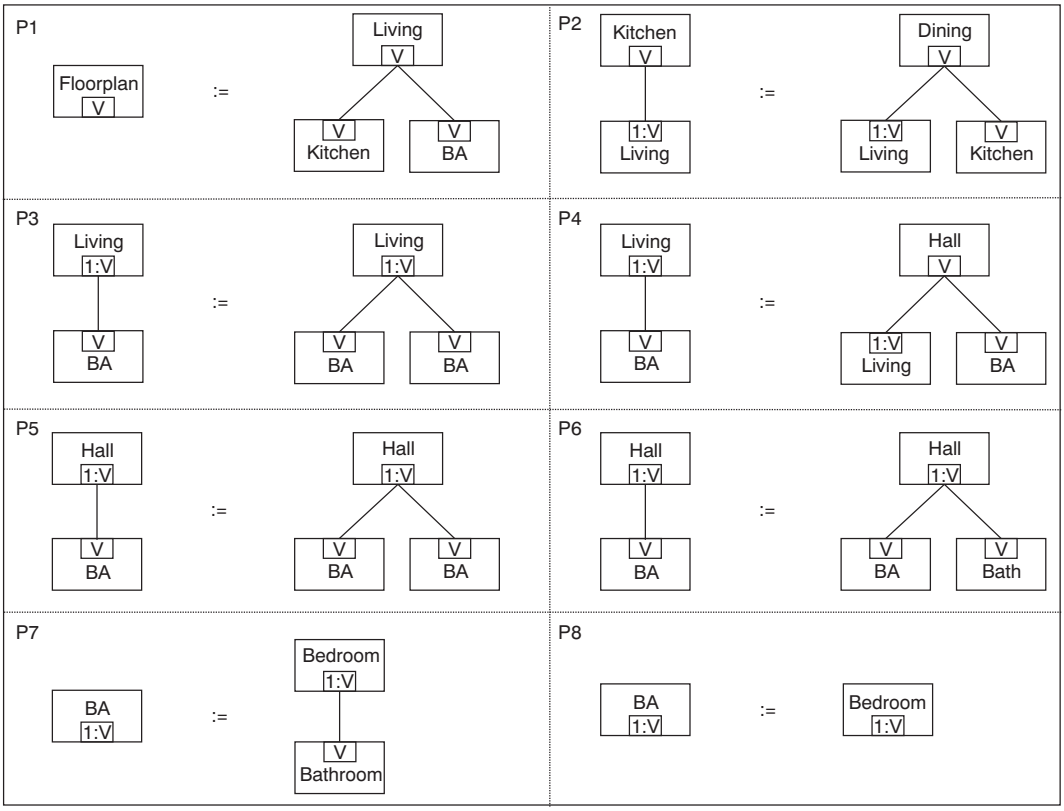

Fig. 6. RGG productions for generating floor plan adjacency graphs.

we did in Graph Approach to Design Generation (GADG) [10], we use RGG to generate graphs for floor plans according to user-specified design requirements.

RGG is a grammatical graph formalism for specifying various visual programming languages. Wang et al. [38] explored the derivation ability of RGG to generate tractable designs. Our approach uses RGG to generate floor plan path graphs based on user-specified design requirements. An edge connecting two nodes in a generated path graph represents the two corresponding rooms in a generated floor plan are connected by a door or an open area [39].

We design a set of productions, shown in Fig. 6, to generate an example type of floor plans' path graphs. Users can design their own graph grammars to generate floor plan structures based on their requirements. Designing a graph grammar is a one-time effort and can generate all the graphs meeting the requirements. In our specification in Fig. 6, a **Floorplan**, described in Production 1, represents a floor plan. It can be divided into three nodes, Living, Kitchen and BA (i.e. Bedroom area). A BA is composed of either a Bedroom or a Bedroom connect with a Bath, defined by Productions 7 and 8. A Dining may be positioned in the middle of a Living and a Kitchen as defined in Production 2. Production 4 defines a Hall connecting with a Living and a BA. Since a living room can connect to multiple bedrooms, Productions 3 and 5 specify a BA can split into two BAs. Production 6 enables a Bath to be connected with a Hall.

Designers can construct a graph grammar by analyzing and specifying the relations among rooms based on their demands. In our proposed graph grammar, we first design essential room connectivity of a floor plan. Assume that a floor plan should have a living room in the center connected with a kitchen and a Bedroom (BA, Production 1). In the BA, there should be a Hall (Production 4) connected with multiple BAs (Production 5). Adding a few specification requirements, e.g. a BA consists of a bedroom with a bathroom, we construct 8 productions for the grammar.

We previously developed a grammar induction engine capable of inferring production rules from a set of graph structures of existing floor plans [42]. This engine extracts common sub-structures from the given graph set using a machine learning based context sensitive induction algorithm. It facilitates designers to identify repeated sub-graphs from the given graph set, e.g. a bedroom connecting with a bedroom can be merged into a new area. Graph productions inferred from this engine are incomplete since some of them are incorrect and the productions connecting different sub-graphs may be missing. Designers then complete the graph grammar by selecting desired productions and creating productions that connect sub-graphs. Our previous work [39] shows that more than half productions in a graph grammar can be inferred using this engine from given graph structures of similar floor plans.

We introduce a simple parametric constraint, *node constraint*, C_n, for specifying requirements in the form $<a, n>$, where a is the name of the node and n is the number of occurrences of a in the generated graph. A set of constraints could be used to specify desired requirements. For example, to generate a floor plan with three Bedrooms and two Baths, one can use the constraint set $\{<Bedroom, 3>, <Bath, 2>\}$ to specify such a graph from a given graph grammar. We recently proposed a constraint validation algorithm to control generated graphs [39]. This algorithm guarantees that generated graphs fulfill all given constraints. But the generation may fail if the execution time over a pre-set threshold caused by an unsatisfiable constraint. For example, if a constraint $\{< Balcony, 1>\}$ is specified, but the node Balcony has no occurrence in the graph grammar.

Graphs generated by this graph grammar share the same properties, such as a Hall connecting a Living, Bedrooms and Baths. Input graphs need to be triangulated before being processed for floor plan generation. Since most of the graph grammar generated path graphs are non-triangulated, graphs need to be triangulated before generating floor plans. A graph is triangulated if every face of the graph is a triangle. There are many existing graph triangulation algorithms. Inspired by Cano and Moral [43], we propose an algorithm to triangulate graphs by adding edges. An overview of the algorithm is given in Algorithm 1. The algorithm creates an empty list, *newEdges*, in the beginning to store newly added edges, as shown in Line 1.

Given an input graph, the algorithm traverses the adjacency list of every vertex and adds an edge between every two neighboring vertices on the adjacency list to guarantee every face to be triangulated (Lines 2–7). For example, if a vertex v_i has an adjacency list $<v_1, v_2 \ldots v_n>$, the algorithm adds an edge between v_j and v_{j+1}, where j is a number between 1 to n-1. The algorithm removes v_i and all edges connecting with v_i since all faces containing v_i have been triangulated. It continues to triangulate faces containing other vertices until the whole graph is triangulated. Since the algorithm removes a vertex and all edges connecting to the vertex in every iteration, this traversal runs in linear time. Line 8 adds new edges into the original graph to create a triangulated graph. Duplicated edges are removed in this step.

Having triangulated the graph, the algorithm identifies articulation vertices using Tarjan's linear articulation finding algorithm [44] and then stores the vertices in a list, *articulationPoints* (Line 9). The articulation finding algorithm finds all articulation points of a graph by a single Depth First Search (DFS) traversal of the graph. Our algorithm then eliminates articulation vertices and vertices of degree 1 by adding additional edges (Lines 10–20). The worst time complexity of adding edges is O(VE), where V is the number of vertices and E is the number of edges in the given graph. At the end of the algorithm, a linear time planarity checking algorithm (Line 21) by Hopcroft and Tarjan [41] is applied to check whether the generated graph is planar. Their algorithm checks $E \leq 3V - 3$. It then checks the planarity of the graph starting from a cycle and adds one edge to it at a time. The overall time complexity of our graph triangulation algorithm is O(VE).

A graph can be triangulated differently by choosing vertices in different orders. Triangulated graphs will be used to generate corresponding floor plans which will be discussed in the next section. Fig. 7 presents two grammar-generated graphs using the same $C_n = \{<Bedroom, 3>, <Bath, 2>\}$ and result graphs after triangulations. The graphs can then be used as inputs to generate floor plans.

Algorithm 1. Graph triangulation.

5. Floor plan generation

GADG is an effective method to generate floor plans based on an existing floor plan [10]. It is able to generate only rectangular rooms within a rectangular boundary. In this paper, we extend GADG by overcoming the limitation and generating floor plans with special requirements.

To generate floor plans, we proposed a PTP graph verification and a RDG finding algorithms to generate rectangular duals for a given graph [10]. We treated a RDG, a graph with n non-overlapping rectangles, as a floor plan and each rectangle in the RDG as a room. Given PTP graphs, GADG generates RDGs accordingly.

We assume that all input graphs discussed in this section are PTP graphs, it is therefore necessary to validate input PTP graphs before generating floor plans. This section first briefly introduces the PTP verification and the RDG generation process, and then presents our new approaches to the treatment of special floor plans.

PTP graph verification. The algorithm checks whether the given graph is a PTP graph satisfying all properties defined in Definition 3. Given a graph $G(V, E)$, the verification process can be divided into two steps:

- Confirming that the number of edges of each face in G is 3;
- Calculating the number of interior faces is equal to $|G.E| - |G.V| + 1$.

RDG generation. We proposed a linear time algorithm [10] to generate RDGs from PTP graphs. Assuming the input PTP graph G_p is shown in Fig. 8 Step 1, the algorithm randomly picks four vertices as

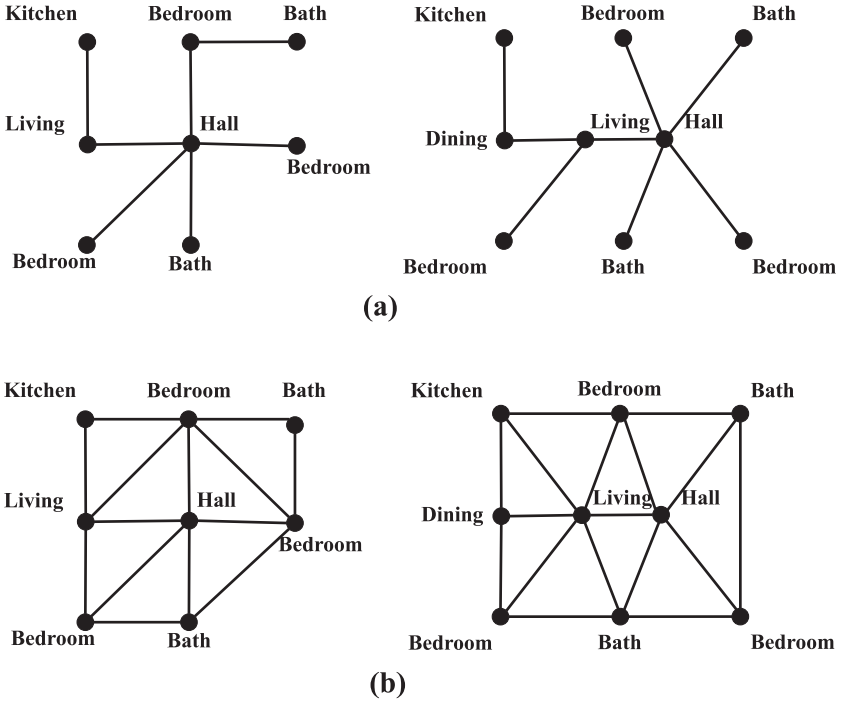

Fig. 7. (a) Two path graphs generated by RGG productions described in Fig. 6 with $C_n = \{<Bedroom, 3>, <Bath, 2>\}$. (b) triangulated graphs after applying the graph triangulation algorithm described in Algorithm 1.

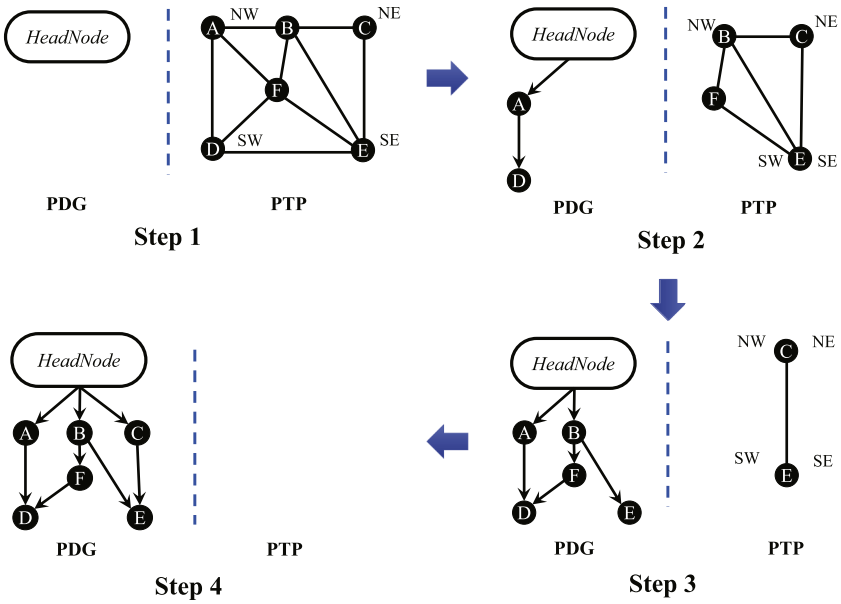

Fig. 8. Procedure of generating a PDG from a PTP graph.

corners, i.e. v_{NW}, v_{NE}, v_{SE}, v_{SW}, from boundary vertices. With G_p and four corner vertices, we can construct a directed graph, called *path digraph* (*PDG*) or G_d, as shown in Fig. 8. A directed edge from vertex v_1 to v_2 implies the corresponding r_1 is on top of r_2 in a generated RDG. Each path in G_d represents a column in generated RDGs. By recursively traversing the left boundary (from v_{NW} to v_{SW}) of G_p, we add vertices on the left boundary to G_d and remove these vertices from G_p. For example, v_A and v_D are removed from G_p and added to G_d, in Fig. 8. v_B and v_E become new v_{NW} and v_{SW}. Since there is only one vertex left on the bottom boundary after removing v_D, $v_{SW} = v_{SE} = v_E$. In step 2, we cannot directly incorporate the path $v_B \rightarrow v_F \rightarrow v_E$ into G_d since there is an edge (v_B, v_E) in G_p. The edge implies r_B and r_E are adjacent and the path $v_B \rightarrow v_F \rightarrow v_E$ implies r_B is on top of r_E with r_F in the middle. Such two conditions cannot be satisfied at the same time. We, therefore, create two paths $v_B \rightarrow v_F \rightarrow v_D$ and $v_B \rightarrow v_E$ instead. The algorithm stops when all vertices have been added to G_d. We then generate RDGs according to G_d from left to right starting from the HeadNode in accordance with the adjacency described in G_p. Fig. 9 shows a RDG generation process using the PTP and PDG graphs described in Fig. 8. The algorithm first places the first path $v_A \rightarrow v_D$, which generates r_A on top of r_D. In step 2, it traverses the next path, $v_B \rightarrow v_F \rightarrow v_D$, to create two rectangles r_B, r_F and expand r_D from the previous column. The next path $v_B \rightarrow v_E$ expands r_B and creates r_E below r_B. The final step places r_C and closes off all rectangles.

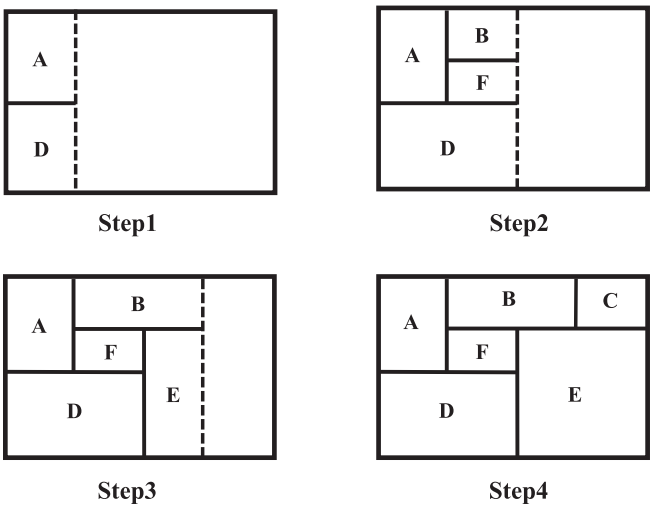

Fig. 9. A RDG generation process from the PTP and PDG graphs in Fig. 8.

A PDG generated by the above approach only contains rectangular rooms within a rectangular boundary [10]. In real-world applications, many floor plans have non-rectangular rooms and boundaries to cater for spaces such as porches and patios. In order to generate floor plans suiting various real-world applications, we overcome these limitations by modifying graphs without altering the room adjacency in generated plans. We introduce a type of vertex, marked as "Empty", in given graphs. The Empty vertex is capable of generating floor plans with non-rectangular boundaries without creating additional rooms. We also allow vertices with the same label to be duplicated to graphs. A set of room placing algorithms are proposed to generate rectangular rooms and merge rooms with the same label.

After 2-dimensional floor plans have been generated, 3-dimensional buildings/houses could be created automatically and viewed in Revit by converting coordinate information of these floor plans into Industry Foundation Classes (IFC) files. IFC is one of the commonly used standards for sharing Building Information Modeling (BIM) data among different applications/platforms. Similar to Bassier et al. [45], we could create an IfcWall object for each edge of the generated floor plan conforming to the IFC4 standard.

5.1. Generating non-rectangular boundaries

To generate designs with non-rectangular boundaries, we introduce a type of vertex marked "Empty", called *Empty vertex*, to be inserted into boundaries of input graphs controlled by a user-specified parameter. This type of vertex indicates an obstacle on the floor plan.

Empty vertices can only exist on the boundary since a floor plan with an obstacle in the middle is not considered by our approach. Given an input graph, our approach randomly adds "Empty" vertices, v_E, on the boundary. Whether to insert an Empty vertex is determined by the following binary function:

$$E_i(\lambda) = \begin{cases} 1 & if\ \lambda - v < 0 \\ 0 & otherwise \end{cases}$$

where v is a constant value range between 0 and 1, controlling the occurrence rate of empty vertices and λ is a random value between 0 and 1. In our implementation, we set v at 0.3. With the function, the algorithm traverses each boundary vertex to evaluate if Empty vertices need to be inserted. Note that users can increase the number of traversals to create more Empty vertices. There may be two or more Empty vertices connected in a generated graph.

Assuming an Empty vertex v_E connecting v_B needs to be inserted to a graph G, there are two scenarios, *Example* 1 and *Example* 2 in Fig. 10,

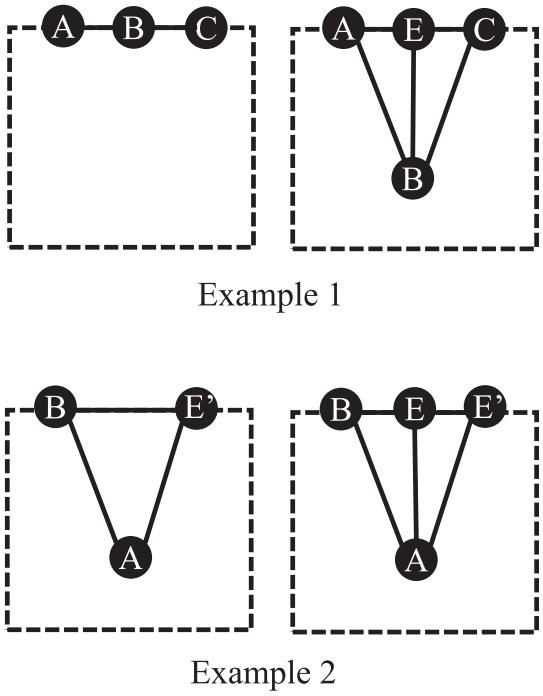

Fig. 10. Inserting Empty vertices into graphs.

that need to be discussed separately. For *Example* 1, v_B is connected with two "Non-Empty" vertices, v_A and v_C, on the boundary. We add v_E by creating three edges (v_A, v_E), (v_B, v_E) and (v_C, v_E). Based on Definition 3 that each internal vertex has degree ≥ 4, v_B needs to have at least three edges before inserting an Empty vertex to keep G a PTP graph. Although this approach moves v_B from the boundary to the inside in a generated plan, the corresponding room r_B is still on the boundary. Fig. 11 shows that adding an Empty vertex does not affect r_B as a room on the boundary since Empty vertices are ignored when generating RDGs. For *Example* 2, v_B is connected with at least an Empty vertex on the boundary. The algorithm selects one of the Empty vertices, $v_{E'}$, and a vertex v_A that connects with v_B and $v_{E'}$. Since G is a PTP graph and each face on a PTP graph has to be a triangle, we can always find such a v_A. The algorithm adds v_E by removing the edge (v_B, $v_{E'}$) and creating edges (v_A, v_E), (v_B, v_E) and ($v_{E'}$, v_E).

Theorem 1. A PTP graph remains triangulated after adding v_E using our approach.

Proof. To add v_E to G, *Example* 1 creates two faces $F(v_A, v_E, v_B)$ and $F(v_B, v_E, v_C)$ without affecting other faces of G. On the other hand, *Example* 2 adds v_E by removing the edge (v_B, $v_{E'}$) on $F(v_B, v_{E'}, v_A)$ and creating two faces $F(v_B, v_E, v_A)$ and $F(v_A, v_E, v_{E'})$. Other faces are unchanged. Therefore, G remains triangulated after adding v_E in either scenario.

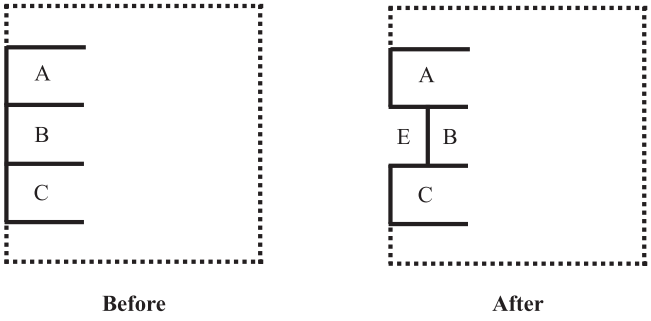

Fig. 11. Modifying boundary shape for room r_B.

5.2. Generating non-rectangular rooms

This sub-section discusses how to generate floor plans with non-rectangular rooms. We define a new type of vertex, called *duplicate vertex*, such that multiple duplicate vertices have the same label. Then, during the plan generation process, our algorithm identifies vertices with the same label and ignores the edges shared between the corresponding shapes.

5.2.1. Adding dummy rooms

Before generating designs, users can specify which rooms should be non-rectangular based on their demands. A constraint $C_d = \{L\}$, where L is a set of room labels, specifying a set of rooms requiring non-rectangular shapes. Our algorithm then duplicates the corresponding vertices to generate non-rectangular rooms. Moreover, users can also use the binary random function defined in Section 5.1 to randomly generate non-rectangular rooms.

To generate a non-rectangular room, our approach inserts a duplicated vertex, whose label is the same with the targeted vertex, into the PTP graph. Since our method generates floor plans with the adjacency preserved, adding additional vertices should not affect the adjacency. To add an additional vertex v_i to a graph, our method starts from a randomly chosen an edge connecting the original v_i. If the edge is a boundary edge, the method in Fig. 11 *Example* 2 (treating duplicate v_i as v_E and original v_i as $v_{E'}$) is performed to add v_i. The graph remains triangulated after the operation (proved in Theorem 1). Otherwise, the method finds two faces sharing the edge. Since all faces of a PTP graph are triangles, we can always find such two faces for a non-boundary edge. As shown in Fig. 12, to add vertex v_A, $C_d = \{A\}$, our method removes the edge (v_A, v_C) and adds the vertex by connecting four vertices of these two faces.

Theorem 2. A PTP graph remains triangulated after adding a new vertex by the method described in Fig. 12.

Proof. Since the original graph is triangulated, removing edge (v_A, v_C) only destroys faces $F(v_B, v_C, v_A)$ and $F(v_C, v_D, v_A)$. Other faces of the graph remain triangular. Then, our method adds v_A by yielding four new triangle faces. Therefore, the graph remains triangulated.

5.2.2. Merging duplicate rooms

Given a PTP graph, in which certain vertices are marked with the same label, our method recognizes these vertices and ignores the edges shared between two corresponding rooms in generated results. For example, Fig. 13 shows a plan with two shapes labeled *E*. The sharing edge (dashed line) between two shapes should be ignored in the final result.

We use three functions, *constructPlan*, *place* and *placeRow*, to generate plans from PTP and PDG graphs and merge rooms of the same label. In our algorithms, we follow several conventions, which are applicable to other variables.

- *v. left* The left X-coordinate of vertex *v*.
- **place**(*v*) Calling function *place* with the parameter *v*.
- (*x*, *y*) A point, whose location is (*x*, *y*).
- ((*v. left*, *v. top*), (*v. left*, *v. bottom*)) An undirected edge between two points.

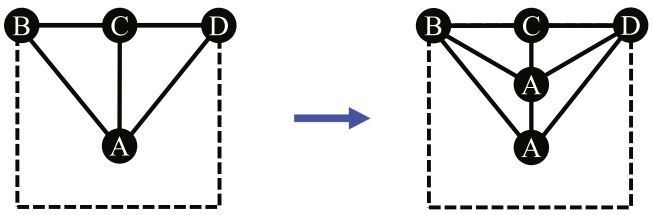

Fig. 12. Adding a vertex with the label *A* using the non-boundary edge (v_A, v_C).

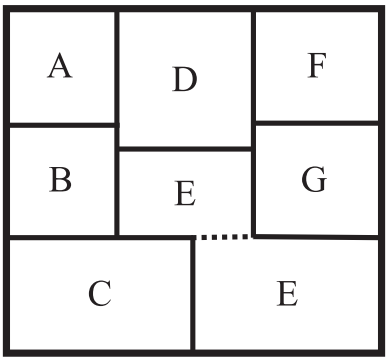

Fig. 13. A plan with two shapes labeled *E*.

Here, *x*, *y*, *firstPath* are global variables, representing the current X value, Y value of the position and whether the current vertex is on the first path of the PDG. Each vertex *v*, representing a rectangle *r* in generated PDGs, has its own variables:

- *visit* A Boolean value indicating whether the vertex has been reached during the traversal.
- *left* The left X-coordinate of *r*.
- *right* The right X-coordinate of *r*.
- *top* The top Y-coordinate of *r*.
- *bottom* The bottom Y-coordinate of *r*.
- *leftY* The Y-coordinate of the currently placed left vertical edge. In the beginning, *leftY* is set to the same as *top*, since no left vertical edge has been placed yet.
- *edges* A collection of edges, which form the rectangle *r* in a generated plan.

Algorithm 2 takes the *HeadNode* as the starting point to generate plans. *x*, *y*, and *firstPath* are initialized. The algorithm calls *place* function to calculate *left*, *right*, *top*, *bottom* values and places left vertical edges for each vertex. The vertical edge is ignored if two connected vertices share the same label. All vertical edges are placed after *place* function except the right boundaries of vertices on the rightmost path. Lines 5–13 in Algorithm 2 traverse the rightmost path of the PDG and place right boundary edges for each vertex on that path. Then, *placeRow* function is called to place horizontal edges of generated plans. The algorithm traverses the PDG and draws a horizontal line if two connected vertices having different labels.

Algorithm 2. *constructPlan* function.

We further explain our algorithm using the PTP graph and its PDG introduced in Fig. 8. To cover the case with two vertices of the same label, we change the label of v_F to *E*, as shown in Fig. 14. The algorithm takes the *headNode* as the input to traverse the PDG. It starts traversing the leftmost path $v_A \rightarrow v_D$. Since this is the first path, the algorithm generates the leftmost vertical line in Fig. 15(a) based on Algorithm 3 (lines 3–8). Then, the second path $v_B \rightarrow v_F \rightarrow v_D$ is traversed. Lines between 12 and 33 run repeatedly for all neighboring vertices on the

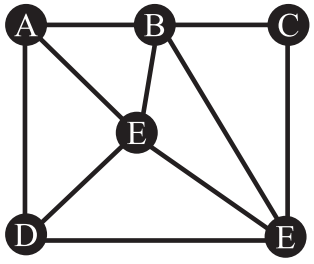

Fig. 14. A PTP graph with two nodes labeled *E*.

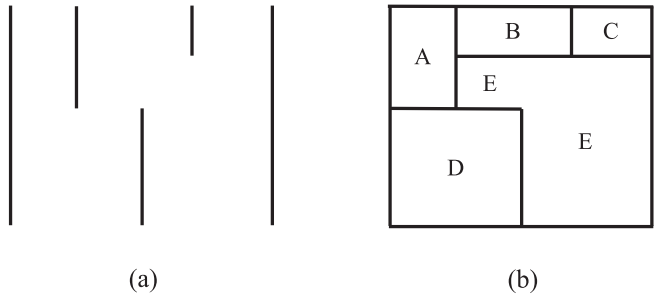

Fig. 15. Generating plans using the PTP graph in Fig. 14. (a) A generated result after executing *place* function. (b) A generated plan after executing *placeRow* function.

PTP graph to determine the bottom of the current vertex. The bottom of vertex v_B is calculated as y_1 based on v_A. Then, the bottom of vertex v_F is determined by v_A and v_D. The algorithm calculates the bottom of v_F as y_2 by line 23 and creates a vertical edge from y_1 to y_2. Since v_D is visited and is a child of v_F, line 17 creates an edge between y_2 and y_3, where y_3 is the top Y-coordinate of v_D. The algorithm returns to the path $v_B \rightarrow v_E$. Since vertex v_F and v_E are both labeled *E*, line 16 prevents the vertical edge between two vertices. The final path $v_C \rightarrow v_E$ creates the vertical left edge of v_C. Up to this step, all vertical edges are created except the rightmost boundary. The algorithm traverses the rightmost path of the PDG to generate boundary edges for each vertex (lines 5–13 of Algorithm 2). Fig. 15(a) shows a result after these steps.

Algorithm 3. *place* function.

placeRow function is called (line 14 of Algorithm 2) to generate horizontal lines column by column based on the PDG. As shown in Algorithm 5, the function checks the labels of connected vertices on the PDG. If two connected vertices have different labels, a horizontal line is placed between the boundary of two vertices. Fig. 15(b) shows a generated plan after executing the *placeRow* function.

Algorithm 4. Contd. *place* function.

Algorithm 5. *placeRow* function.

The capability of generating non-rectangular rooms provide more possibilities for generating plans. The resulting designs are also closer to real-world applications. Fig. 17 shows one design with and another without non-rectangular rooms and boundaries while sharing the same room adjacency in Fig. 16. The floor plan in Fig. 17(b) is generated with the constraints C_d = {*Bedroom*_1, *Bedroom*_2} and an empty vertex connecting with *Bath*.

6. Design constraints

Our method is extensible for supporting various constraints in generating floor plans. Plans generated without proper constraints may be far from reality or design requirements. For example, the size of a plan is too large or a room is too narrow. To generate well controlled floor plans, we define several high-level constraints. Users can specify these constraints based on their own requirements.

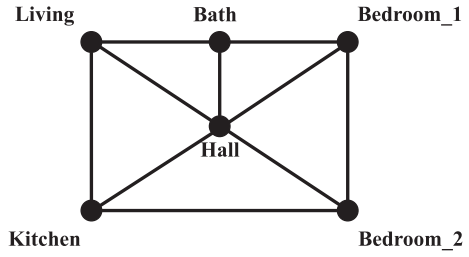

Fig. 16. A PTP graph with two bedrooms and one bath.

Plan size: C_s. One can specify the overall size of generated plans using two parameters l_s and w_s. l_s is the length and w_s is the width of generated plans. Our algorithm then generates results under the user-specified $C_s = \{w_s * h_s\}$.

Aspect ratio: C_r. We allow users to set the maximum aspect ratio, $C_r = \{r\}$, for all the rooms in generated plans. r is a constant value specifying the maximum ratio. The aspect ratio of a rectangular room is the ratio of its longer side to its shorter side. Aspect ratios of all generated rooms should be less than or equal to r, which means:

$$r \geq r_i \geq 1$$

The size of each vertex v_i is controlled by four boundary values, $v_i.left$, $v_i.right$, $v_i.top$, and $v_i.bottom$. Since vertices with the same label represent a room, we calculate the width and height of the room as:

$$w_i = Max_{v_i.label=\sigma}(v_i.right) - Min_{v_i.label=\sigma}(v_i.left)$$

$$h_i = Max_{v_i.label=\sigma}(v_i.top) - Min_{v_i.label=\sigma}(v_i.bottom)$$

Then, r_i is calculated as:

$$r_i = \frac{Max(w_i,\ h_i)}{Min(w_i,\ h_i)}$$

This equation compares the ratio for vertices with the same label. For rooms generated by combining several vertices with the same label, it calculates the aspect ratio of the bounding box for each room in generated plans.

Room orientation: C_o. Considering that users can require a room facing a particular direction, such as Living room facing south to receive maximum sunlight, we provide C_o allowing users to specify the directions of rooms. We use the parameter set $\{r_i, o_i\}$ to represent the direction of room r_i. Four possible values, *N*, *W*, *S*, *E*, for o represent four directions of generated plans. Since the direction of an internal room is meaningless, we encourage users to only set direction constraints for rooms on the boundary. If the user requires an internal room in a certain direction to have a window, our previous research [10] provides a graph modification method, which may alter the adjacency relation of the input graph, to move the internal room to the boundary.

7. Experiments

We implement our algorithm in JAVA using Eclipse SWT Widget on a desktop with dual 2.93 GHz Intel Core i7 processes and 8 GB RAM, running Windows 7 Enterprise.

We apply the graph grammar productions introduced in Fig. 6 to generate a graph, as shown in Fig. 18(a), by given C_n = {<*Bedroom*, 3>, <*Bath*, 2>}. After adding Empty vertices and duplicating existing vertices, the adjacency of rooms in the generated plans remains the same. Fig. 19 shows two floor plans generated from the given graph. The room orientation constraint C_o is specified as {{*Living*, *W*}}. Aspect ratio constraint is set C_r as {*2.5 except Hall*}.

For comparison, we also generate two floor plans with the same adjacency described in Fig. 18(b). One uses this approach and the other uses GADG with the size constraint C_s = {*15 * 10m*} and the maximum aspect ratio C_r as {*2.5 except Hall*}. Fig. 20(a) shows a floor plan generated by GADG. All rooms in the plan are rectangular. To make it realistic, designers may continue to refine. Moreover, since the northwest Bath locates in the same column with Living room, i.e. sharing the same length with Living room, the generated room is either too large or too narrow. In contrast, this approach can solve this issue by introducing an Empty vertex to reduce the size of such a room. Non-rectangular rooms for certain requirements, e.g., the corners of the two southeast Bedrooms in Fig. 20(b) can be used as Closets, could be generated from the requirements that two bedrooms and the hall need to be non-rectangular.

We evaluated the execution time of our approach by generating floor plans with different numbers of rooms. By generating 20 different

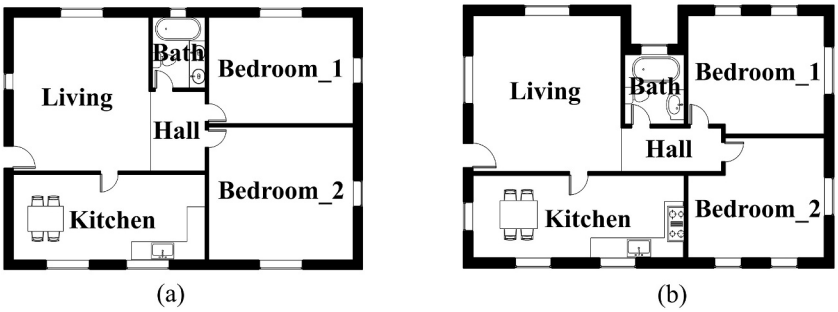

Fig. 17. Floor plans generated by our method. (a) A plan generated from the graph in Fig. 16. (b) A plan generated from the same graph after adding an Empty vertex and the constraints C_d = {*Bedroom*_1, *Bedroom*_2}.

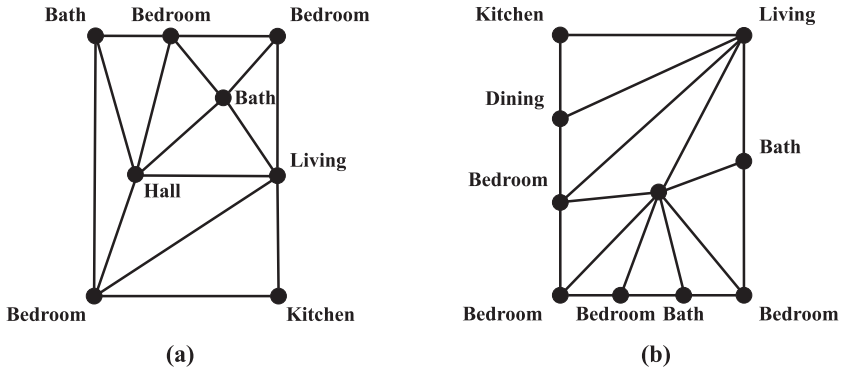

Fig. 18. Two RGG generated graphs: (a) a graph generated with node constraints C_n = {<*Bedroom*, 3>, <*Bath*, 2>}. (b) a graph generated with node constraints C_n = {<*Bedroom*, 4>, <*Dining*, 1>}.

Fig. 19. Two generated plans with the same room adjacency described by the graph in Fig. 18(a).

floor plans for each user input, we calculate the average generation time. Fig. 21 shows that the average execution time increases slightly with the increasing number of non-rectangular rooms, represented as duplicate vertices. Since most real-world floor plans have less than 20 rooms, our approach generates non-rectangular floor plans in a reasonable time.

8. Conclusion and future works

In this paper, we propose a framework for automatic generation of floor plan designs. The framework overcomes the limitations of only generating rectangular rooms and highly relying on existing floor plans introduced by GADG. It uses a RGG formalism to generate graphs representing room adjacencies based on user-specified constraints. A triangulation algorithm is followed to triangulate generated graphs. Two types of vertices, empty and duplicate vertices, are introduced to enhance the generation power. With these vertices, the framework is capable of generating different non-rectangular plans, whose room adjacency relationships are consistent with the original user requirements. In addition, we introduce a set of algorithms for placing non-rectangular shapes from graph structures. With user-specified high-level constraints, e.g. plan size and aspect ratio, floor plans are

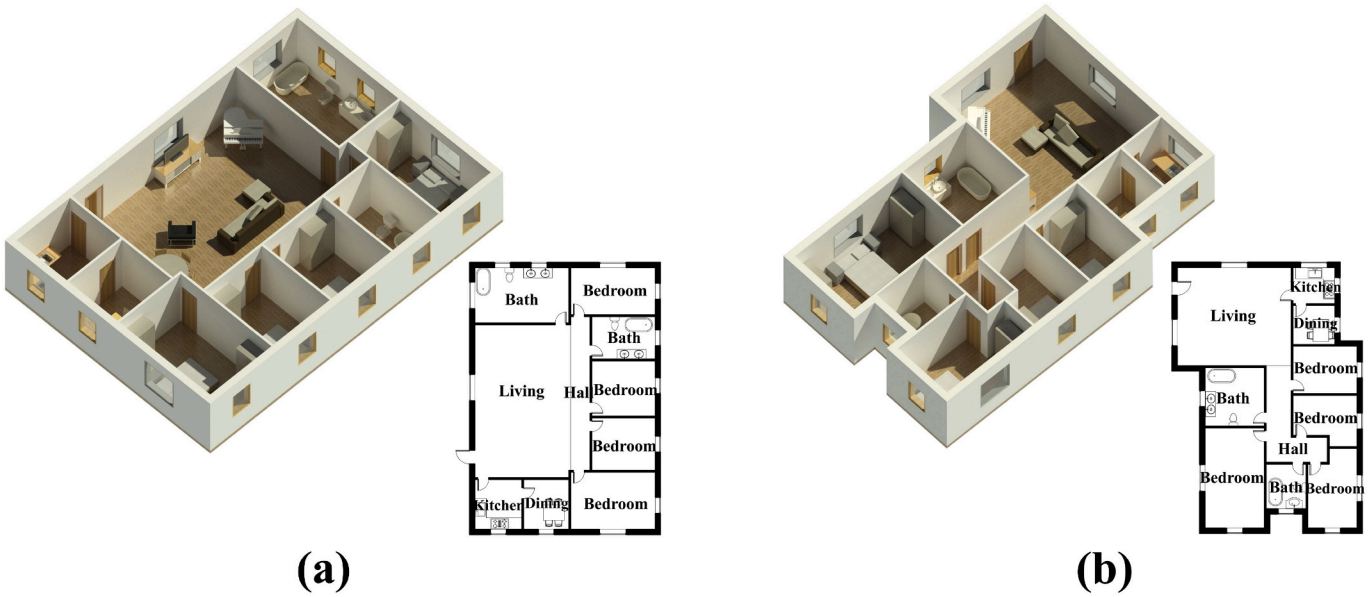

Fig. 20. Two generated plans using the adjacency graph described in Fig. 18(b). (a) The plan generated by GADG. (b) The plan generated by this approach.

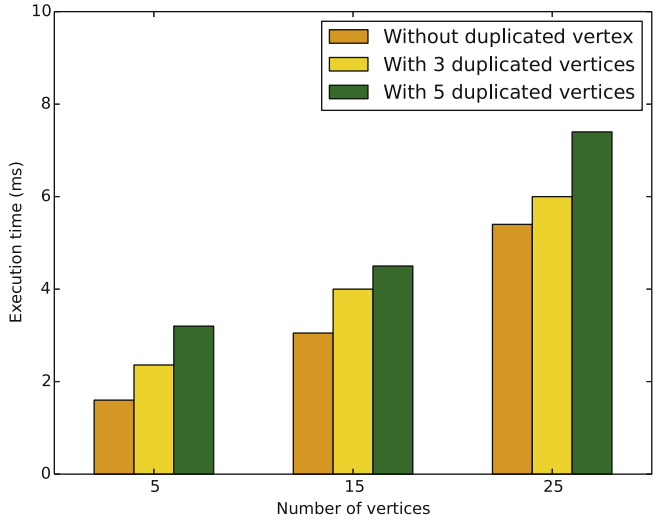

Fig. 21. The comparison of execution time with or without non-rectangular rooms.

generated suiting various design requirements.

Experiments have been made to testify the generation power of our approach. We demonstrate that our method can generate different plans consisting of varied room shapes based on user-specified constraints within a reasonable time. Experiments also show that designers can control generated floor plans by giving parametrized design requirements, e.g. the number of bedrooms, the size of the plan and the aspect ratio of rooms.

Our framework of course has limitations. For instance, our current approach generates floor plans only from PTP graphs. Although a graph triangulation algorithm, which triangulates graphs by adding edges, is introduced, the adjacency relation in an input graph may be altered after triangulation. Floor plans generated from PTP graphs should not contain two intersecting partitions as a 4-way cross that separate four rooms, yet this type of plans may exist in real-world applications. Moreover, our current implementation only supports specifying which rooms should be adjacent instead of which rooms should not be adjacent. As future works, we plan to extend our framework to other applications, e.g. buildings, schools, game designs. Moreover, doors and windows are randomly generated for floor plans in this approach, we plan to use graph grammar to formally define a rule-based approach for generating house façades. The generated façades can then be used to control the locations of rooms, e.g. a window cannot be located in the middle of two rooms. To automatically generate both interior and exterior of multistory buildings, we also plan to introduce constraints which limit the location and size of stairs/elevator, since they should be consistent among different floors. We could adapt our approach so that it would be used to generate vertical building designs.

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.

Appendix A. Supplementary data

Supplementary data to this article can be found online at https://doi.org/10.1016/j.autcon.2020.103288.

References

[1] G. Stiny, J. Gips, Shape grammars and the generative specification of painting and sculpture, Proceedings of the Workshop on Generalisation and Multiple Representation, Leicester, 1971 (ISBN: 0-7204-2063-6).
[2] G. Stiny, W.J. Mitchell, The palladian grammar, Environment and Planning B: Planning and Design 5 (1) (1978) 5–18, https://doi.org/10.1068/b050005.
[3] G. Stiny, W.J. Mitchell, The grammar of paradise: on the generation of mughul gardens, Environment and Planning B: Planning and Design 7 (2) (1980) 209–226, https://doi.org/10.1068/b070209.
[4] H. Koning, J. Eizenberg, The language of the prairie: Frank Lloyd Wright's prairie houses, Environment and Planning B: Planning and Design 8 (3) (1981) 295–323, https://doi.org/10.1068/b080295.
[5] G. Çağdaş, A shape grammar: the language of traditional Turkish houses, Environment and Planning B: Planning and Design 23 (4) (1996) 443–464, https://doi.org/10.1068/b230443.
[6] J.P. Duarte, Towards the mass customization of housing: the grammar of Siza's houses at Malagueira, Environment and Planning B: Planning and Design 32 (3) (2005) 347–380, https://doi.org/10.1068/b31124.
[7] J. Roth, R. Hashimshony, A. Wachman, Generating layouts with non-convex envelopes, Build. Environ. 20 (4) (1985) 211–219, https://doi.org/10.1016/0360-1323(85)90036-8.
[8] A.B. Kahng, Classical floorplanning harmful? Proceedings of the 2000 International Symposium on Physical Design, ISPD '00, ACM, New York, NY, USA, 2000, pp. 207–213, , https://doi.org/10.1145/332357.332401.
[9] P. Merrell, E. Schkufza, V. Koltun, Computer-generated residential building layouts,

ACM Trans. Graph. 29 (6) (2010) 181:1–181:12, https://doi.org/10.1145/1882261.1866203.
[10] X.-Y. Wang, Y. Yang, K. 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.
[11] U. Flemming, Wall representations of rectangular dissections and their use in automated space allocation, Environment and Planning B: Planning and Design 5 (2) (1978) 215–232, https://doi.org/10.1068/b050215.
[12] U. Flemming, R. Woodbury, Software environment to support early phases in building design (seed): overview, J. Archit. Eng. 1 (4) (1995) 147–152, https://doi.org/10.1061/(ASCE)1076-0431(1995)1:4(147).
[13] S.-F. Chien, U. Flemming, Design space navigation in generative design systems, Autom. Constr. 11 (1) (2002) 1–22, https://doi.org/10.1016/S0926-5805(00)00084-4.
[14] P. Merrell, E. Schkufza, Z. Li, M. Agrawala, V. Koltun, Interactive furniture layout using interior design guidelines, ACM Trans. Graph. 30 (4) (2011) 87:1–87:10, https://doi.org/10.1145/2010324.1964982.
[15] C.-H. Peng, Y.-L. Yang, P. Wonka, Computing layouts with deformable templates, ACM Trans. Graph. 33 (4) (2014) 99:1–99:11, https://doi.org/10.1145/2601097.2601164.
[16] K. Shekhawat, Algorithm for constructing an optimally connected rectangular floor plan, Frontiers of Architectural Research 3 (3) (2014) 324–330, https://doi.org/10.1016/j.foar.2013.12.003.
[17] W. Wu, L. Fan, L. Liu, P. Wonka, Miqp-based layout design for building interiors, Computer Graphics Forum 37 (2) (2018) 511–521, https://doi.org/10.1111/cgf.13380.
[18] J. Martin, Procedural house generation: a method for dynamically generating floor plans, Symposium on Interactive 3D Graphics and Games, 2 2006 doi:10.1.1.97.4544.
[19] F. Marson, S.R. Musse, Automatic real-time generation of floor plans based on squarified treemaps algorithm, International Journal of Computer Games Technology (2010), https://doi.org/10.1155/2010/624817.
[20] K. Kozminski, E. Kinnen, An algorithm for finding a rectangular dual of a planar graph for use in area planning for vlsi integrated circuits, Proceedings of the 21st Design Automation Conference, DAC '84, IEEE Press, Piscataway, NJ, USA, 1984, pp. 655–656, , https://doi.org/10.1109/DAC.1984.1585872.
[21] J. Bhasker, S. Sahni, A linear algorithm to find a rectangular dual of a planar triangulated graph, Proceedings of the 23rd ACM/IEEE Design Automation Conference, DAC '86, IEEE Press, Piscataway, NJ, USA, 1986, pp. 108–114, , https://doi.org/10.1145/318013.318031.
[22] Y.-N. Li, K. Zhang, D.-J. Li, Rule-based automatic generation of logo designs, Leonardo 50 (2) (2017) 177–181, https://doi.org/10.1162/LEON_a_00961.
[23] M. Agarwal, J. Cagan, A blend of different tastes: the language of coffeemakers, Environment and Planning B: Planning and Design 25 (2) (1998) 205–226, https://doi.org/10.1068/b250205.
[24] G. Çağdaş, A shape grammar model for designing row-houses, Des. Stud. 17 (1) (1996) 35–51, https://doi.org/10.1016/0142-694X(95)00005-C.
[25] P. Wonka, M. Wimmer, F. Sillion, W. Ribarsky, Instant architecture, ACM Trans. Graph. 22 (3) (2003) 669–677, https://doi.org/10.1145/882262.882324.
[26] K. Shekhawat, Enumerating generic rectangular floor plans, Autom. Constr. 92 (2018) 151–165, https://doi.org/10.1016/j.autcon.2018.03.037.
[27] G. Rozenberg, E. Welzl, Boundary nlc graph grammars — basic definitions, normal forms, and complexity, Inf. Control. 69 (1) (1986) 136–167, https://doi.org/10.1016/S0019-9958(86)80045-6.
[28] K. Wittenburg, Earley-style parsing for relational grammars, Proceedings IEEE Workshop on Visual Languages, 1992, pp. 192–199, , https://doi.org/10.1109/WVL.1992.275765.
[29] J. Rekers, A. Schürr, Defining and parsing visual languages with layered graph grammars, J. Vis. Lang. Comput. 8 (1) (1997) 27–55, https://doi.org/10.1006/jvlc.1996.0027.
[30] D. Zhang, K. Zhang, J. Cao, A context-sensitive graph grammar formalism for the specification of visual languages, Comput. J. 44 (3) (2001) 186–200, https://doi.org/10.1093/comjnl/44.3.186.
[31] J. Kong, K. Zhang, X. Zeng, Spatial graph grammars for graphical user interfaces, ACM Transactions on Computer-Human Interaction 13 (2) (2006) 268–307, https://doi.org/10.1145/1165734.1165739.
[32] X.-Q. Zeng, X.-Q. Han, Y. Zou, An edge-based context-sensitive graph grammar formalism, Journal of Software 19 (8) (2008) 1893–1901, https://doi.org/10.3724/SP.J.1001.2008.01893.
[33] K. Zhang, D.-Q. Zhang, Y. Deng, Graphical transformation of multimedia xml documents, Ann. Softw. Eng. 12 (1) (2001) 119–137, https://doi.org/10.1023/A:1013310705258.
[34] C. Zhao, J. Kong, K. Zhang, Program behavior discovery and verification: a graph grammar approach, IEEE Trans. Softw. Eng. 36 (3) (2010) 431–448, https://doi.org/10.1109/TSE.2010.3.
[35] A. Roudaki, J. Kong, K. Zhang, Specification and discovery of web patterns: a graph grammar approach, Inf. Sci. 328 (2016) 528–545, https://doi.org/10.1016/j.ins.2015.08.052.
[36] T. Grasl, A. Economou, From topologies to shapes: parametric shape grammars implemented by graphs, Environment and Planning B: Planning and Design 40 (5) (2013) 905–922, https://doi.org/10.1068/b38156.
[37] T. Strobbe, P. Pauwels, R. Verstraeten, R. De Meyer, J. Van Campenhout, Toward a visual approach in the exploration of shape grammars, Artificial Intelligence for Engineering Design, Analysis and Manufacturing 29 (4) (2015) 503–521, https://doi.org/10.1017/S0890060415000475.
[38] X.-Y. Wang, Y.-F. Liu, J. Li, K. Zhang, Generating tractable designs by transforming shape grammars to graph grammars, Proceedings of the 11th International Symposium on Visual Information Communication and Interaction, VINCI '18, ACM, New York, NY, USA, 2018, pp. 41–48, , https://doi.org/10.1145/3231622.3231637.
[39] X.-Y. Wang, Y.-F. Liu, K. Zhang, A graph grammar approach to the design and validation of floor plans, Comput. J. 63 (1) (2019) 137–150, https://doi.org/10.1093/comjnl/bxz002.
[40] G. Rozenberg, Handbook of Graph Grammars and Computing by Graph Transformation, World Scientific, 1997 (ISBN: 978-981-02-2884-2).
[41] J. Hopcroft, R. Tarjan, Efficient planarity testing, J. ACM 21 (4) (1974) 549–568, https://doi.org/10.1145/321850.321852.
[42] K. Ates, K. Zhang, Constructing veggie: machine learning for context-sensitive graph grammars, 19th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2007), vol. 2, 2007, pp. 456–463, , https://doi.org/10.1109/ICTAI.2007.59.
[43] A. Cano, S. Moral, Heuristic algorithms for the triangulation of graphs, Advances in Intelligent Computing, Springer, Berlin, Heidelberg, 1995, pp. 98–107, , https://doi.org/10.1007/BFb003594.
[44] R. Tarjan, Depth-first search and linear graph algorithms, SIAM J. Comput. 1 (2) (1972) 146–160, https://doi.org/10.1137/0201010.
[45] M. Bassier, R. Klein, B. Van Genechten, M. Vergauwen, Ifc wall reconstruction from unstructured point clouds, Annals of the Photogrammetry Remote Sensing and Spatial Information sciences 4 (2) (2018) 33–39, https://doi.org/10.5194/isprs-annals-IV-2-33-2018.