DPLAN: Minimal Connectivity to Floorplan Generation

Rohit Lohani[1⋆] and Krishnendra Shekhawat[1]

BITS Pilani, Department of Mathematics, Pilani Campus, India

Abstract. Automated floor plan generation is an important problem in computational architectural design. The goal is to construct a floor plan from user-defined room numbers and door requirements. The user specifies which rooms must share a door and which rooms must not be adjacent. However, these requirements do not determine the exact placement or shape of the rooms. The task is therefore to arrange the rooms in a single floor plan so that all required door connections are satisfied and no rooms overlap. To address this problem, we propose *Door Connectivity to Floor Plan Generation (DPLAN)*, a graph-based prototype that generates floor plans from door and non-adjacency constraints. The framework operates in three stages.

1. **Connectivity adjustment:** The user-defined graph is examined. If it is disconnected, additional edges are added to connect its components.
2. **Graph construction:** From the connected graph, a bi-connected plane triangulation is constructed to ensure the existence of a floor plan without overlapping or empty spaces.
3. **Floor plan generation:** The triangulated graph is transformed into floor plans using two modes.
   - For **rectangular floor plans (RFPs)**, separating triangles are removed by modifying edges without adding new vertices, so that no extra rooms are created.
   - For **orthogonal floor plans (OFPs)**, separating triangles are removed by introducing additional vertices, which allows rectilinear room shapes.

By enforcing both door and non-adjacency requirements, the framework generates floor plans that satisfy the given constraints. Our proposed method is based on graph construction, plane triangulation, and systematic graph modification. The proposed method is implemented in Python and includes a prototype that allows interactive input of constraints and visualization of the generated floor plans. Currently, the framework supports rectangular plot boundaries. Future work includes support for non-rectangular plots, dimension-based scaling, and circulation modeling to further extend its practical applicability.

Keywords: Algorithm, Plane Graphs, Connectivity, Triangulation, Separating Triangle, Floor plan.

1 Introduction

Floor planning is a fundamental problem in architectural design because the arrangement of rooms determines access between spaces and the connections between rooms. A floor plan must satisfy several requirements, such as which rooms should be adjacent, which rooms should remain separate, and how the design fits within the given plot boundary. These requirements influence one another, which makes the design process difficult. As the number of rooms increases, the number of possible arrangements grows rapidly. For this reason, manual exploration becomes time-consuming, and simple computational search methods become inefficient.

Over the years, many automated floor-planning methods have been developed. These include optimization-based techniques, shape grammar approaches, graph-based models, and machine learning methods (discussed in section 3). Among them, graph-based approaches are effective because they represent rooms as vertices and spatial relationships as edges. This representation clearly shows the connections and separations between rooms. Graph-based models have also been used in areas such as VLSI floorplanning and computational architecture, where connectivity and planarity are important. However, many existing systems require users to specify complete adjacency information in advance or depend on heuristic optimization methods. This reduces flexibility and limits the range of floor plans that can be generated. In practical design settings, architects often specify only essential requirements, such as which rooms must be connected, which must remain separate, the shapes of rooms, and the boundary of the floor plan. Ensuring that both adjacency and non-adjacency requirements are satisfied in the final floor plan in a systematic way is challenging. Although modern computer-aided design tools and data-driven methods (see Section 3) can generate initial layouts, they provide limited control over the underlying structure. It is often unclear how properties such as bi-connectivity,

⋆ Corresponding author: Rohit Lohani, p20210045@pilani.bits-pilani.ac.in

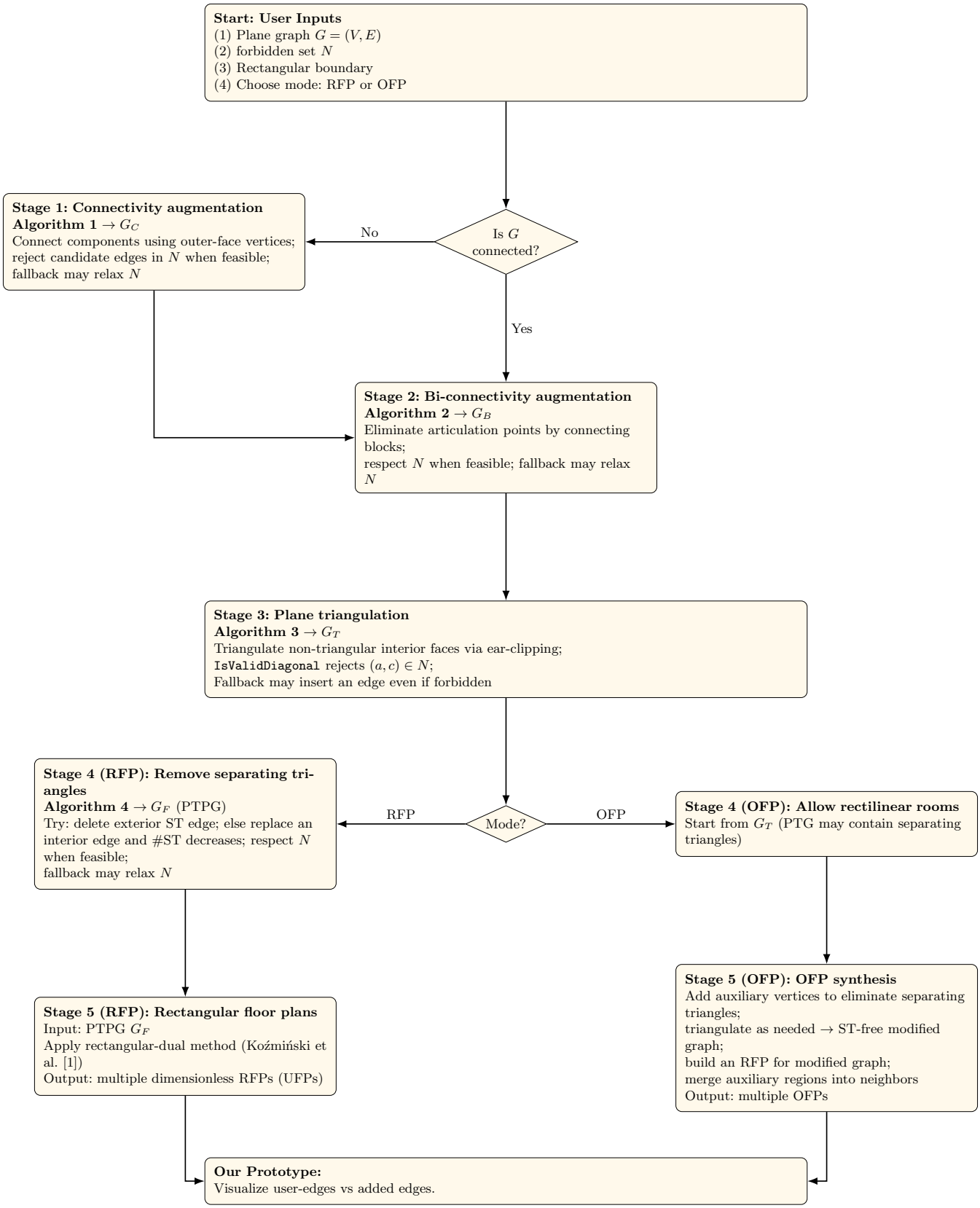

Fig. 1. Flowchart of the DPLAN pipeline from minimal door-connectivity input (with non-adjacency constraints) to RFP/OFP generation.

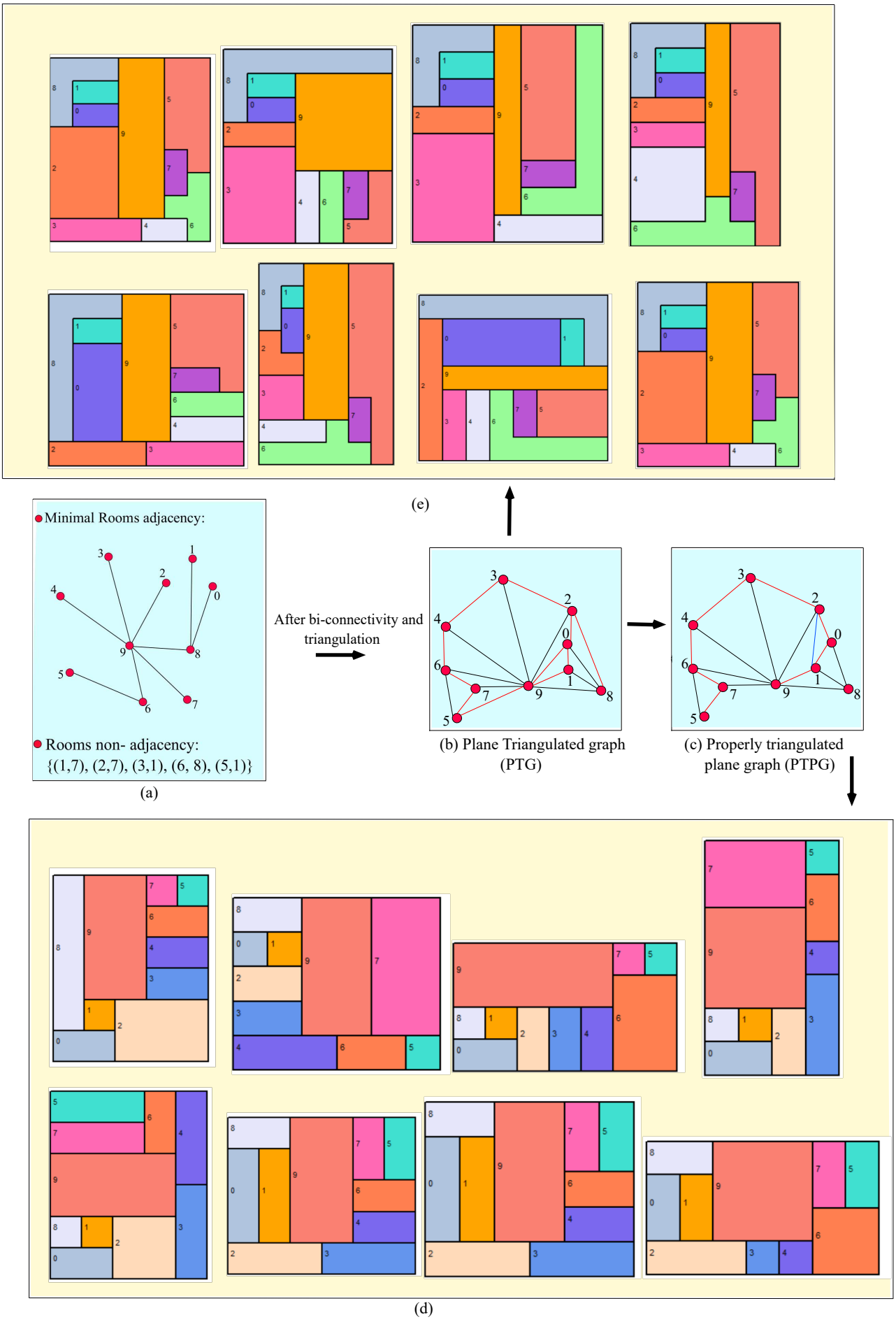

Fig. 2. (a) Minimal input graph with non-adjacency constraints. (b) A plane triangulated graph constructed using the proposed approach. (c) A properly triangulated plane graph obtained from the proposed approach. (d) Orthogonal floor plans generated from the resulting PTG. (e) Rectangular floor plans generated from the resulting PTPG.

triangulation, and the handling of separating triangles are ensured.
In this paper, we present *Door Connectivity to Floor Plan Generation* (DPLAN), a graph-based framework for automated floor plan generation (see Figure 1). The system builds floor plans from essential adjacency and non-adjacency requirements derived from door connections, without requiring complete room-to-room specifications (see Figure 2). The framework first ensures that the input graph is connected. It then converts the graph into a bi-connected plane-triangulated form. From this representation, multiple floor plans are generated that satisfy the given constraints. The resulting floor plans remain valid while allowing different room arrangements.
The current implementation supports dimensionless floor plans within rectangular plot boundaries. This provides a basis for studying the structural feasibility and variation in floor-plan design. Future work includes support for non-rectangular plots, room dimensions, and circulation modeling to extend the method toward practical architectural applications. By combining graph-based modeling with interactive software support, DPLAN offers a clear and extensible approach to automated floor plan generation.
The paper is organized as follows. Section 2 introduces the main definitions and notation used in this work. Section 3 reviews related literature. Sections 4 and 5 discuss existing approaches, identify research gaps, and position our contribution. Section 6 describes the proposed methodology, including the step-by-step process from graph construction to floor plan generation, along with the associated algorithms. Section 7 explains the use of the DPLAN interface. Sections 8 and 9 discuss limitations and directions for future work. Finally, Section 12 provides an appendix that includes runtime analysis, comparisons with existing methods, the scope of the approach, and computational complexity and correctness analyses of the implemented algorithms.

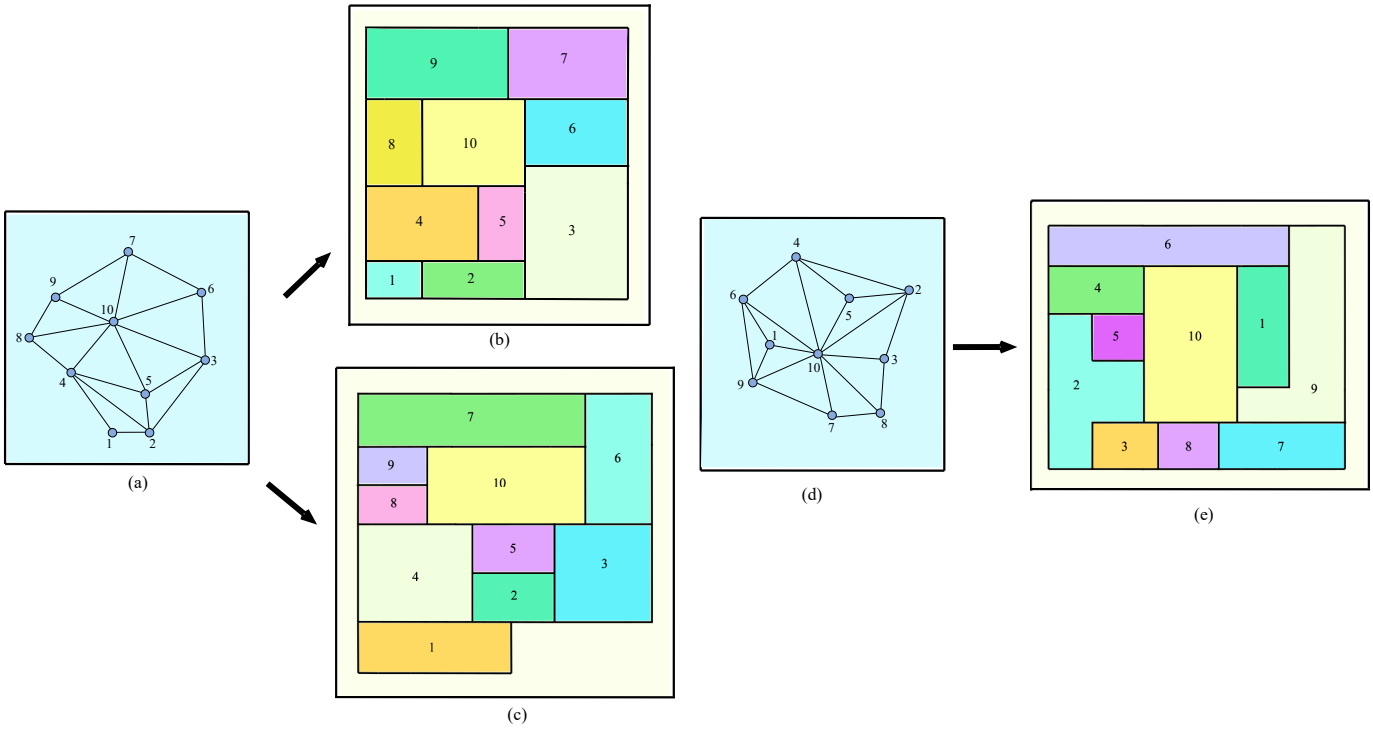

Fig. 3. (b) Rectangular floor plan based on the graph in (a); (c) non-rectangular floor plan derived from the same graph; (e) orthogonal floor plan generated from the graph in (d).

2 Terminologies

This section introduces the basic concepts and terminology used in this study. These definitions provide a clear foundation for representing and analyzing floor plans using graph structures and spatial relationships.

(i). **Floor Plan [2]:** A *floor plan* (F) consists of an outer polygonal region that is internally divided into n non-overlapping sub-regions, each representing an individual *room*. The arrangement of these rooms follows the connectivity prescribed by a graph on n vertices, where each vertex corresponds to a room. The polygon enclosing the entire subdivision is referred to as the *floor plan boundary*, and the straight line segments that separate neighboring rooms form the *walls*. Two rooms are regarded as adjacent precisely when they meet along a wall segment of positive length.
Floor plans can be classified according to the geometry of their outer boundary and the shapes of the rooms they contain. The most common categories are described below:

(a). **Rectangular Floor Plan (RFP):** A floor plan is termed a *rectangular floor plan* when both the enclosing plot and all interior spaces are bounded by rectangles. In such layouts, every room admits four orthogonal sides, and the overall configuration fits entirely within a rectangular boundary (see Figure 3a).

(b). **Orthogonal Floor Plan (OFP):** A floor plan is described as an *orthogonal floor plan* if its outer boundary is rectangular, while allowing some rooms to assume orthogonal shapes that are not strictly rectangular, including configurations such as L-, T-, or C-shaped regions. An illustration of such rooms is shown in Figure 3e (rooms 9 and 2).

(c). **Non-rectangular Floor Plan (NRFP):** A floor plan is categorized as a *non-rectangular floor plan* when the shape of its outer boundary is not rectangular, independent of whether the internal rooms themselves are rectangular or non-rectangular. An example is provided in Figure 3c.

(ii). **Graph Representations of Floor Plans:** A floor plan can be naturally modeled using graph-theoretic structures. In such representations, each vertex corresponds to a room, and an edge between two vertices indicates that the corresponding rooms share a common wall. Graphs constructed in this manner are known as *adjacency graphs*. Modeling floor plans as adjacency graphs enables the application of established graph algorithms for analysis, optimization, and floor plan synthesis.
In addition to adjacency graphs, floor plans are often studied using *dual graphs*. Given a planar embedding of a graph, its dual is obtained by associating each face of the original (primal) graph with a vertex in the dual graph. Two vertices in the dual graph are connected by an edge if and only if their corresponding faces in the primal graph share a common boundary edge.
Within the scope of this work, adjacency graphs for RFPs and NRFPs consist of vertices representing rectangular rooms and edges representing shared walls between them. In contrast, graphs derived from OFPs may include vertices associated with both non-rectangular and rectangular rooms with orthogonally shaped boundaries, while edges consistently represent adjacency through common walls.

(iii). **Plane Graphs and Their Variants [3]:** A *plane graph* is a graph together with a fixed drawing in the plane in which edges are represented as non-intersecting curves, meeting only at shared endpoints. This drawing induces a decomposition of the plane into connected regions known as *faces*. The unique unbounded region is called the *outer* (or exterior) face, while all bounded regions are referred to as *inner* (or interior) faces. A vertex that lies on the boundary of the outer face is termed an *exterior vertex*; all remaining vertices are classified as *interior vertices*.
Certain subclasses of plane graphs play a central role in floor plan generation and are described below:

(i). **Plane Triangulated Graph (PTG):** A plane graph is considered *triangulated* when every interior face is enclosed by exactly three edges, whereas the outer face may consist of three or more edges (see Figure 3d). A triangular cycle whose interior contains at least one vertex is referred to as a *separating triangle*, as it divides the graph into distinct regions (see Figure 3d, separating triangle $(6, 9, 10)$).

(ii). **Properly Triangulated Plane Graph (PTPG):** A *properly triangulated plane graph* is a connected plane graph in which all interior faces are triangular, and no separating triangles occur (see Figure 3b). The absence of separating triangles imposes a stronger structural condition, which is useful for algorithmic construction of floor plans.

3 Literature Survey

Automated floor plan generation has developed along two main research directions. The first direction is constraint-based. It uses graph models or rule-based procedures to explicitly represent requirements such as connectivity, planarity, and other structural conditions needed for floor plan construction. The second direction is data-driven. It

learns geometric patterns and design regularities from large collections of existing floor plans, often using generative models that convert a graph specification into a geometric layout.
In constraint-based methods, the floor plan is viewed as a partition of the building boundary into rooms, and the design requirements are encoded as an adjacency or connectivity graph. These methods are interpretable and can provide guarantees for the constraints they explicitly model. However, they often assume that a well-structured graph is already given. The process of converting sparse design requirements into such a structured graph is not always addressed directly.
In contrast, learning-based methods can generate visually diverse and realistic layouts. However, constraints such as strict non-adjacency, triangulation, or bi-connectivity are usually encouraged through training objectives rather than guaranteed by construction.

3.1 Graph-theoretic approaches

Table 1 presents representative constraint-based approaches that start from an explicit adjacency graph and generate candidate floor plans through graph transformations or optimization procedures. These methods work effectively when the input graph is already consistent and sufficiently detailed. However, they generally do not address the problem of minimally modifying an input graph while enforcing explicit non-adjacency requirements.

3.2 Learning-based and hybrid approaches

Table 2 presents representative learning-based approaches that generate room geometries from a given graph and boundary condition. These methods can produce realistic and diverse floor plans. However, structural constraints (such as adjacency, triangulation, connectivity) are usually encouraged through training objectives rather than guaranteed by construction, and the input graph is typically assumed to be internally consistent.

Table 1: Graph-theoretic and rule-based methods for floor plan generation.

Work	**Year**	**Primary input**	**Main idea / representation**	**Output and remarks**
Koźmiński & Kinnen [1]	1985	Planar graph under rectangular-dual conditions	Characterizes when an adjacency graph admits a rectangular dual and gives a constructive method	Strong guarantees, but assumes a highly structured planar input instead of building one from sparse constraints.
Rinsma [4]	1988	Graph-theoretic floor plan existence setting	Provides existence-style results under stated combinatorial conditions	Mainly theoretical; not presented as a complete generation pipeline.
Lai & Leinwand [5]	1990	Plane graph (embedded)	Reduces rectangular-dual construction to a bipartite matching problem on a derived graph	Gives an algorithm when the input is already planar/embedded and satisfies the required conditions.
M. Kurowski [6]	2003	Plane near-triangulation / suitable planar graph	Constructs a dissection-like floor plan from a suitable planar graph using a simple procedure	Efficient construction, but does not target rich architectural constraints (e.g., explicit forbidden adjacencies).
Liao *et al.* [7]	2003	Planar embedding via orderly spanning tree	Uses planar encoding to build compact rectilinear drawings/dissections	Focused on planar encoding/drawing; not designed for constraint-heavy floor plan briefs.
Marson & Musse [8]	2010	Room list and rough size/grouping intent	Recursive partitioning (treemap-like subdivision) followed by adjustments	Fast and interactive, but strict adherence to the given adjacency/non-adjacency is not the main guarantee.
Zhang *et al.* [9]	2011	Plane triangulation	Studies transformations that improve triangulation structure for downstream constructions	Useful when a triangulation is already available; it does not focus on repairing sparse/disconnected briefs.

Continued on next page

Work	Year	Primary input	Main idea / representation	Output and remarks
Mirahmadi & Shami [10]	2012	Procedural rules + room program	Rule-based subdivision with corridor-driven heuristics and optimization	Produces plausible plans, but relies on heuristics rather than certified graph conditions.
Hao Hua [11]	2016	Template/shape cues + relational constraints	Matches a target template/topology and uses search/optimization for feasibility	Handles irregular regions well; feasibility is mainly achieved by optimization.
Wang *et al.* [12]	2018	Existing floor plan (or derived structure) + edits	Edits/repairs a plan by graph extraction and graph transformations	Effective when a coherent seed plan exists; less focused on generating from scratch under sparse constraints.
Upasani *et al.* [13]	2020	A valid dimensionless arrangement + geometric bounds	First fixes a valid arrangement, then assigns dimensions via optimization	Produces dimensioned plans, assuming a valid topology/arrangement is already given.
Shekhawat *et al.* [14]	2021	Adjacency graph (+ optional boundary/dimensions)	Combines graph construction with optimization/dimensioning for floor plan generation	Algorithmic workflow, but assumes the adjacency specification is already meaningful and complete enough.
Shekhawat *et al.* [15]	2023	Input graph + room-shape categories	Generates plans allowing specified orthogonal/non-rectangular room shapes within constraints	Supports richer room shapes; feasibility depends on the structural graph and the allowed shape family.
Bisht *et al.* [16]	2022	Adjacency graph (+ optional dimensions)	Enumerates topologically distinct floor plan/candidates consistent with a given graph, then dimensions them	Good for exploring alternatives once the adjacency graph is fixed; less focused on minimal graph repair/augmentation.
Shiksha *et al.* [17]	2025	Rule-level brief (required/optional adjacency, forbidden contacts, circulation cues)	Converts a brief into candidate design graphs using rules, then synthesizes plans	Expressive for brief-to-graph translation; emphasis is not on minimal augmentation to a certified backbone under all constraints.

Table 2: Learning-based (data-driven) methods for floor plan generation and reconstruction.

Work	Year	Primary input	Main idea / representation	Output and remarks
C. Liu [18]	2018	RGBD video / 3D scans	Multi-branch network predicts floor plan geometry/semantics, with an IP step for vector output	Targets floor plan reconstruction from scans rather than brief-based generation.
W. Wu *et al.* [19]	2019	Boundary + dataset priors (optionally, room program)	Two-stage learned synthesis: allocate rooms, then generate/refine walls within the boundary	Fast and plausible, but strict feasibility may require post-processing.
J. Chen *et al.* [20]	2019	RGBD evidence (top-view cues)	Room-wise optimization guided by learned cues and consistency terms	Focused on reconstruction quality, not free-form generation from constraints.
Hu *et al.* [21]	2020	Constraint graph + boundary	Graph2Plan: generates a floor plan from a graph and boundary using learned decoding stages	Can follow sparse constraints, but hard constraints often need explicit validation/repair.
Nauata *et al.* [22]	2020	Adjacency graph (rooms/types/contacts)	House-GAN: graph-conditioned GAN generates room boxes consistent with the input graph	Diverse generation; hard constraints may be violated without explicit checking.

Continued on next page

Work	Year	Primary input	Main idea / representation	Output and remarks
Nauata *et al.* [23]	2021	Graph + initial plan for refinement	House-GAN++: iterative refinement improves a generated plan under the same graph conditioning	Cleaner outputs, but strict feasibility still typically needs validation steps.
S. Wang *et al.* [24]	2021	Boundary + activity/-function priors	ActFloor-GAN: activity-guided synthesis for human-centric residential plans	Produces plausible designs; constraint correctness is mainly learned rather than guaranteed.
J. Sun *et al.* [25]	2022	Boundary + design constraints	WallPlan: predicts an intermediate wall-graph, then decodes to a structured plan	More structured than raster-only methods; still may require rule checks for CAD validity.
S. Shabani *et al.* [26]	2023	Constraint graph + vector corner/door representation	HouseDiffusion: conditional diffusion that directly generates vector floor plans	Stronger geometric control; performance depends on constraint encoding and training coverage.
Z. Han *et al.* [27]	2024	Adjacency graph (+ attributes)	Graph2pix: graph-conditioned generator produces floor plan images aligned to the graph	Usually followed by vectorization/-cleanup for CAD use.
Chen Ma Thi *et al.* [28]	2024	Boundary + user constraints	Learns boundary partitioning under constraints, then reconstructs a coherent plan	Practical synthesis, but formal guarantees are typically not provided.
Zhang *et al.* [29]	2024	Partial user specification (incomplete constraints/attributes)	MaskPLAN: masked generation completes missing attributes and produces a plan from partial input	Supports interactive partial briefs; still benefits from explicit feasibility checks.
S. Hong *et al.* [30]	2024	Graph, boundary, or both	Cons2Plan: conditional diffusion generates vector floor plans under various conditions	Vector output helps, but architectural constraints often still need validation.
P. Zeng [31]	2024	Multiple conditions (boundary + functional constraints)	Diffusion-based synthesis designed to improve controllability under multiple inputs	Good diversity/control; CAD-grade validity often requires post-processing.
M. Abouagour *et al.* [32]	2025	Boundary (and door/entry context)	Factorizes the task: first plan topology, then realize geometry with neural components	Explicitly separates topology from geometry; still not a formal graph-certification pipeline.

4 Gap in the Existing Research

In the early phase of architectural design, floor plan requirements are usually incomplete. Designers often specify only a few essential room relationships and outline a rough building boundary, while leaving many other adjacency decisions open. This reflects the exploratory nature of conceptual design, in which spatial arrangements are refined step by step rather than fixed at the outset. In contrast, many automated floorplanning methods assume a much stronger and more structured input. Typically, they require a planar, connected, triangulated adjacency graph that satisfies the conditions for constructing a rectangular/orthogonal floor plan. In these approaches, connectivity between spaces is often inferred from shared boundaries, and explicit non-adjacency constraints are rarely enforced during the initial graph construction. As a result, there is a gap between the flexible, partially specified inputs used in practice and the stricter assumptions made by many computational models.

A large body of graph-theoretic [14,16,8,10,12,11] research has demonstrated how adjacency graphs can be translated into families of feasible floor plans, particularly for rectangular or orthogonal floor plans. Several studies have shown that properly triangulated plane graphs (PTPGs), or PTGs, a closely related planar graph class, serve as a necessary backbone for generating rectangular floor plans. These methods are effective once such a graph is available, but they typically assume that the user already possesses the expertise to provide or construct a valid graph. From an architectural perspective, this places a substantial burden on the designer, who must understand graph planarity, connectivity, triangulation, and the presence or absence of separating triangles to provide a usable input. How to derive such graphs algorithmically from sparse adjacency and non-adjacency information remains largely unexplored.

Recent years have also seen significant progress in learning-based approaches for layout/floor plan generation, including systems such as Graph2Plan[21], House-GAN[22], House-GAN++[23], ActFloor[24], and WallPlan[25],

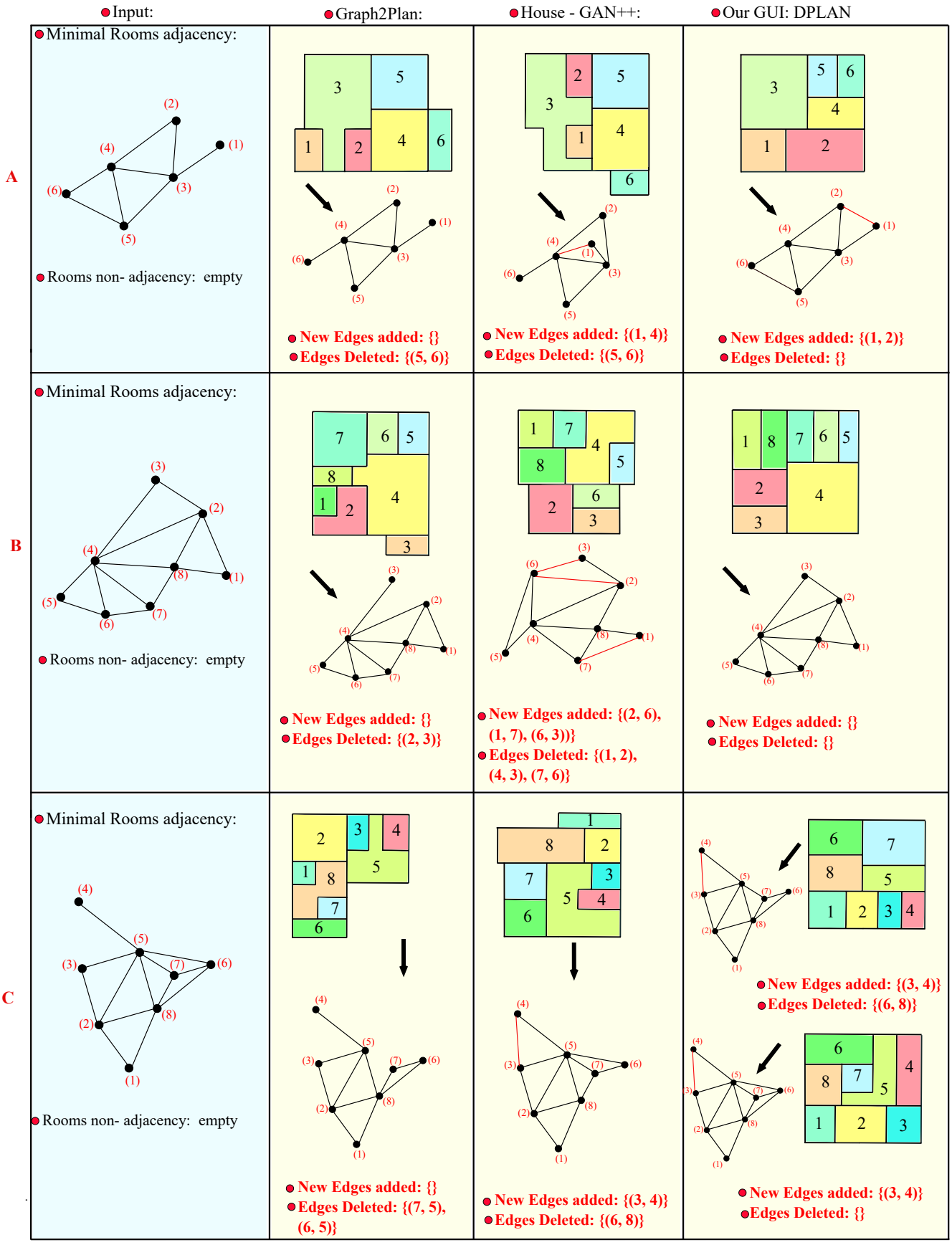

Fig. 4. Visual comparison of inputs and generated outputs using our method, HouseGAN++, and Graph2Plan.

Graph2pix[27], etc. These methods leverage large datasets to learn spatial patterns and generate visually plausible layouts under given boundary conditions. While powerful, their behaviour is fundamentally data-driven. Constraint satisfaction is guided by learned priors or soft loss functions rather than explicit combinatorial guarantees. As a result, adjacency requirements, non-adjacency requirements, and structural properties such as triangularity or bi-connectivity may fail to hold, particularly when the input specification is sparse, inconsistent, or deviates from the training distribution (see Figure 4). When such violations occur, these models offer limited repair mechanisms, since they do not reason directly about the underlying door-connectivity graph (see Table 3). These systems generate a floor plan conditioned on an input relationship graph, but they treat that graph as a prerequisite rather than an object to be constructed further. As a result, when the user specification is sparse, disconnected, or inconsistent, neither learning-based methods provides a mechanism to complete and certify the underlying door graph before geometric layout generation.

Taken together, the literature lacks a transparent, principled mechanism that bridges the gap between high-level architectural briefs and downstream floor-plan/layout generation. There is no standard pipeline that repairs incomplete or disconnected specifications, enforces connectivity and bi-connectivity under hard non-adjacency constraints, and upgrades the resulting structure into a properly triangulated plane graph or plane triangulated graph suitable for established rectangular or orthogonal floorplanning algorithms. The present work addresses this missing layer by treating the constraint graph itself as the primary design artifact. Starting from minimum adjacency and explicit non-adjacency information, we construct and certify a PTPG or PTG through a sequence of graph-theoretic augmentations, adding only the necessary edges. In doing so, door connectivity and separation requirements become explicit, verifiable, and controllable objects, rather than incidental by-products of geometric optimization or data-driven synthesis.

5 Our Proposed Work

We propose an interactive prototype that transforms an input door-connectivity specification into a graph suitable for floor-plan construction. From this backbone, the system generates floor plans that can be selected and further refined. Unlike approaches that assume the input constraint graph is fixed in advance, our method treats the evolving graph as the main design object. Each modification step is explicit, providing clear insight into how feasibility is established.

5.1 Inputs

The user specifies (i) a minimum door-adjacency graph representing only essential connections, (ii) a non-adjacency set representing forbidden connections, and (iii) a rectangular boundary template. This input format aligns with early-stage briefs while remaining precise enough to support correctness by construction processing.

5.2 Certified backbone construction

Our developed prototype implements a staged augmentation pipeline, where each stage has a focused structural goal and can be inspected and validated:

- **Connectivity augmentation:** if the input graph contains multiple connected components(blocks), the system adds the minimum number of edges needed to make it connected, rejecting any candidate edge that violates forbidden/non-adjacency edges.
- **Bi-connectivity augmentation:** the connected graph is augmented to eliminate articulation points. This ensures that the graph remains connected even after removing a single vertex, while still respecting the given non-adjacency constraints.
- **Constrained planar embedding and triangulation:** The bi-connected graph is then embedded in the plane and further refined by adding diagonals that satisfy the given constraints. This process produces a triangulated structure suitable for the subsequent synthesis stage.
- **Handling separating triangles (User mode-specific):** When the construction requires strictly rectangular modules, separating triangles are handled by carefully removing and adding edges so that a rectangular floor plan is possible. In contrast, if more flexible orthogonal floor plans/layouts are allowed, separating triangles are resolved by introducing additional nodes into the graph.

Table 3. Comparison between Graph2Plan, HouseGAN++, and DPLAN

Parameter	**Graph2Plan [21]**	**HouseGAN++ [23]**	**DPLAN (Proposed Method)**
Input specification	Room connectivity graph with room types and counts, together with a boundary extracted from the dataset.	Bubble diagram describing room relationships and a coarse program; boundary control is indirect.	Adjacency graph with an explicit non-adjacency set and a user-defined rectangular outer boundary.
Adjacency satisfaction	Attempts to realize the input graph, but generated layouts may introduce or miss certain adjacencies.	Encourages specified relationships, yet some edges may not be preserved due to generative sampling.	All required adjacencies are strictly enforced, and all non-adjacent pairs remain separated throughout graph construction and in the final layout.
Floorplan boundary	Follows dataset boundaries; minor deviations may occur in generated results.	Produces data-driven irregular boundaries; direct user control is limited.	Always fits the layout exactly within the specified rectangular boundary.
Topological guarantees	No formal guarantees on planarity, biconnectivity, or removal of separating triangles.	Focuses on visual realism; rectangular dual realizability is not ensured.	Explicitly constructs a separating-triangle-free plane triangulation, ensuring the existence of a valid rectangular dual consistent with the constraints.
Role of data	Requires large annotated datasets for training; performance depends on training distribution.	Strongly data-driven; generalization beyond trained building types may be limited.	Fully algorithmic and independent of training data; applicable without example floorplans.
Nature of output	Raster floorplans with extracted room boxes; dimensions reflect learned data patterns.	Vector layouts resembling professional designs; sizes and proportions follow learned distributions.	Families of dimensionless rectangular floor plans that can later be dimensioned using optimization or rule-based methods.
User control and interpretability	User specifies graph and boundary, but geometric realization is learned and less transparent.	Designer sketches relationships; internal generative decisions are not directly interpretable.	Each modification (connectivity, biconnectivity, triangulation, separating-triangle handling) corresponds to a clear algorithmic step, making the process transparent and verifiable.

5.3 Interaction and Downstream Synthesis

After the combinatorial backbone has been formally constructed and verified, i.e., PTPG or PTG, the prototype produces multiple dimensionless floor plan/layout alternatives and enables interactive refinement. The system visually distinguishes between edges defined by the user and those introduced during algorithmic augmentation. Whenever the user modifies adjacency requirements or non-adjacency/forbidden-contact edge pairs, the system immediately checks feasibility and reports whether the updated specification remains valid.

This framework addresses a practical limitation in existing approaches. Data-driven methods often generate visually convincing floor plans, but the underlying structural decisions that inform them are not directly transparent. Conversely, traditional graph-based methods provide formal correctness guarantees but typically operate in a fixed, non-interactive workflow. In contrast, the proposed prototype keeps the combinatorial structure editable, transparent, and formally validated before any geometric optimization or rendering is performed.

6 Methodology

This section presents the graph-theoretic framework underlying our approach to architectural floor plan generation, focusing on the algorithmic workflow and practical considerations. The method starts with user-provided inputs: a door-based adjacency graph and explicit non-adjacency constraints, which together describe essential spatial relationships. To obtain a connected, structurally valid representation suitable for floor-plan synthesis, the graph is augmented with additional edges corresponding to interior wall adjacencies. A key aspect of this stage is the treatment of separating triangles, as their presence directly affects the geometric realizability of the resulting floor plan.

When the objective is to generate strictly rectangular floor plans, the augmented graph must remain free of separating triangles while respecting all input constraints, since such configurations ensure compatibility with rectangular modules. In contrast, when non-rectangular room shapes are allowed, separating triangles are handled by introducing additional vertices that correspond to new spatial modules, enabling the synthesis of orthogonal floor plans.
The proposed framework is structured as a five-stage pipeline, where each stage performs a specific structural or geometric refinement. Only the stages necessary for the user's design objectives are executed. The process begins with the user-provided minimal constraint graph. The first step is to check whether the graph is connected. If it is disconnected, Algorithm 1 adds the minimum required edges to obtain a connected graph while strictly respecting all specified non-adjacency constraints. If the graph is already connected, the method proceeds to the next stage. In the second stage, Algorithm 2 augments the connected graph to make it biconnected. This ensures that the graph remains connected after the removal of any single vertex and that all required adjacency relations are preserved during this augmentation. The resulting biconnected graph is then converted into the intermediate representation needed for further processing. Subsequently, triangular configurations are treated according to the desired module geometry. When strictly rectangular modules are required, separating triangles are removed in a constraint-preserving manner so that all adjacency conditions remain satisfied.
After these steps, the output is either a plane triangulated graph (PTG) or a properly triangulated plane graph (PTPG), depending on the chosen layout model. This certified structure forms the input to the floorplan synthesis stage. The details of each stage are described in the following section

6.1 Minimal Connectivity to 1-connected Graph.

Connectivity is a basic requirement for floor-plan synthesis. If the constraint graph is disconnected, the rooms form separate components that cannot be realized as a single layout. Therefore, the graph must be at least connected (1-connected) so that all spaces belong to one coherent system. A connected graph provides the foundation for later steps such as biconnectivity and triangulation, which depend on a unified structure to produce a consistent geometric layout. Earlier work, including Roth et al. [33], also notes that valid floor plans are derived from connected and subsequently triangulated layout graphs. When users specify only door-based relations, the resulting graph may not be connected. In such cases, we introduce additional wall-adjacency edges to ensure connectivity before further processing. This augmentation is performed carefully so that no given non-adjacency constraints are violated. Algorithm 1 formalizes this step by inserting edges only when necessary.

Illustration of Algorithm 1 (see Figure 5):

1. Steps 1–6: begin by taking the input graph $G(V, E)$ together with the user-specified non-adjacency constraints shown in Figure 5(a). We then determine how the graph is partitioned by identifying all of its connected components. When this check reveals that the graph is already fully connected, meaning only a single component is present, the process stops, and the original graph is returned. This early termination prevents unnecessary augmentation and ensures that additional processing occurs only when the input actually contains multiple disconnected parts.

2. Steps 7–11: are invoked when the input graph is divided into more than one connected component. In this stage, each component is examined individually, beginning with the extraction of the vertices that appear on its outer boundary, shown in Figure 5(b). For the example ($G(V, E)$) under consideration, these boundary sets are $\{2, 6, 8, 7, 0, 13, 1\}$, $\{9, 11, 12, 10\}$, and $\{3, 4, 5\}$. Using these outerface vertices, the algorithm then forms all possible cross-component pairs and removes any pair that conflicts with the specified non-adjacency constraints. The pairs that remain after this filtering step represent the viable edges through which disconnected components may be linked. For the graph $G(V, E)$ in the example, this procedure yields 51 such admissible connections/edges while satisfying non-adjacency constraints (see Figure 5c):
[21(outerface[1]–outerface[2]) + 18(outerface[1]–outerface[3]) + 12(outerface[2]-outerface[3])] = 51, which are stored in the *possible_edges* set.

3. Steps 12–16: instantiate a union–find data structure in which each connected component is initially represented as its own set (for example, all vertices of component[1] = $\{0, 2, 7, 8, 6, 1, 13, 14\}$ are assigned the same component index; see Figure 5(b)). The algorithm then processes the acceptable/candidate edges from *possible_edge_set* sequentially: for each edge (u, v), it checks whether u and v belong to distinct union–find sets; if so, it accepts (u, v)

Algorithm 1 : *Connectivity with Non-adjacency*

function *Connectivity*($G(V, E)$, *non_adj_set*)
  augmentation_edges $\leftarrow \{\}$.
  components $\leftarrow$ *Connected_Components*(G), $m = |components|$.
  outerfaces $\leftarrow \{\}$. ▷ *For each component i, store vertices lying on its outer face boundary*
  if *length*(*components*) == 1 **then**
    return $G_C(V_C, E_C) = G(V, E)$.
  else
    for $i \leftarrow 0$ **to** *length*(*components*) $- 1$ **do** ▷ *Compute outerface boundary of each component via planar embedding*
      $outerfaces[i] \leftarrow Extract\_Outerface(components[i])$.
    possible_edges $\leftarrow$ *Generate_Possible_Edges*($outerfaces$, *non_adj_set*).
    Initialize $root[i] \leftarrow i$ for $i = 0.....|components| - 1$.
    for $(u, v) \in$ *possible_edges* **do** ▷ *For each vertex u, component_ID(u) gives the ID of the component containing u.*
      $c_u \leftarrow Component\_ID(u)$, $c_v \leftarrow Component\_ID(v)$.
      if $Find\_Comp(c_u) \neq Find\_Comp(c_v)$ **then**
        $Union\_Comp(c_u, c_v)$.
        augmentation_edges $\leftarrow$ *augmentation_edges* $\cup \{(u, v)\}$, $m \leftarrow m - 1$.
    $E \leftarrow E \cup$ *augmentation_edges*.
    return $G_C(V_C, E_C) = G(V, E)$, m.
  function $Extract\_Outerface(component)$
    return $\{ v \in V_C \mid v$ lies on the boundary of *component* $\}$.
  function *Generate_Possible_Edges*($outerfaces$, *non_adj_set*)
    possible_edges $\leftarrow \{\}$.
    for $i \leftarrow 0$ **to** *length*($outerfaces$) $- 1$ **do**
      for $j \leftarrow i + 1$ **to** *length*($outerfaces$) $- 1$ **do**
        for $u \in outerfaces[i]$ **do**
          for $v \in outerfaces[j]$ **do**
            if $(u, v) \notin$ *non_adj_set* **then**
              possible_edges $\leftarrow$ *possible_edges* $\cup \{(u, v)\}$.
    return *possible_edges*.
  function $Find\_Comp(u)$
    if $root[u] \neq u$ **then**
      $root[u] \leftarrow Find\_Comp(root[u])$.
    return $root[u]$.
  function $Union\_Comp(u, v)$
    $root_u \leftarrow Find\_Comp(u), root_v \leftarrow Find\_Comp(v)$.
    if $root_u \neq root_v$ **then**
      $root[root_v] \leftarrow root_u$.
if $m > 1$ **then** ▷ *(fallback): ignore non-adj-constraints and retry replacements*
  $G_C(V_C, E_C) \leftarrow Connectivity(G_C(V_C, E_C)$, *non_adj_set* $= \phi)$.

as an augmentation edge and merges the two sets/components. For instance, $(2, 9)$ is accepted because vertex 2 belongs to component[1] and 9 to component[2], and the pair does not violate non-adjacency constraints; after this union, a subsequent candidate edge such as $(2, 11)$ is skipped because 11 now belongs to the same merged set as 2. This accept-and-merge process continues for the remaining candidate edges until all components are combined into a single connected graph. Only those edges that connect previously disjoint components are stored in the *augmentation_edges_set*.

4. Steps 17–18: conclude Algorithm 1 by incorporating the selected augmentation edges into the existing edge set E, thereby completing the connectivity of the graph. Once these edges have been inserted, the algorithm produces the final connected graph G_C, shown in Figures 5(c) and 5(d), which reflects the fully unified structure resulting from the earlier augmentation steps.

5. Steps 19–37: of Algorithm 1 rely on several helper/additional functions that perform the supporting work necessary for the main method to run smoothly. The $Extract\_Outerface$ function generates the vertices on the boundary cycle of each component in the input plane graph, as illustrated in Figure 5(b) (see outerface[1], outerface[2], outerface[3]). The *Generate_Possible_Edges* function then checks all cross-component pairs drawn from these boundary vertices and removes any pair that violates the non-adjacency constraints. The *union* function keeps track of the component to which each vertex belongs, using the operations $Find_Comp$ and $Union_Comp$; for example, once an edge in the *augmentation_edges* set merges components 1 and 2, any later edge whose endpoints lie in these two components is skipped, since they have already been joined. Every edge in the *augmentation_edges* set satisfies the non-adjacency constraints, and each chosen edge genuinely reduces the number of disconnected components. Through these checks, Algorithm 1 ultimately produces a valid connected graph G_C, as shown in Figure 5(a–d).

5. Steps 38–39: If the current constraints do not allow additional edges to be inserted to achieve full connectivity, a fallback step is applied. This step temporarily ignores non-adjacency constraints and re-executes the connectivity function to merge all components.

By applying Algorithm 1, we obtain a connected graph G_C derived from the user-specified input graph G together with the stated non-adjacency constraints (see Figure 5(a–d)). Since we are treating the input graph as a plane drawing, the outer face of each connected component is well-defined. In Algorithm 1, the vertices incident to the outer face are identified prior to any augmentation, and all additional edges are inserted only afterward; consequently, the extraction of outer-face vertices is independent of the augmentation step and does not require recomputation or planarity verification. Once the final connected graph G_C is obtained, a plane embedding of G_C serves as input to the subsequent algorithm, which further processes it to achieve bi-connectivity.

6.2 Connectivity graph to Bi-connected graph

Koźmiński et al. [1] showed that generating a rectangular or orthogonal floor plan requires the underlying layout graph to be both bi-connected and triangulated. Consequently, when deriving a rectangular floor plan from a user-specified minimal connectivity graph, the first step is to augment the previously obtained one-connected graph (from Algorithm 1) so that it becomes bi-connected, after which triangulation can be performed. Our proposed algorithm (Algorithm 2) provides a procedure for constructing a bi-connected graph while respecting user-defined constraints. Its objective is to introduce additional adjacencies only where necessary, ensuring that the graph remains connected even after the removal of any single vertex. Our proposed Algorithm 2 takes as input a plane-connected graph G_C and articulation points in G_C (using the approach proposed by Tarjan [34]), identifies the blocks associated with them, and then selectively adds edges within these blocks. Throughout this process, it adheres strictly to the supplied Non-Adjacency list, ensuring that no non-adjacency pairs are ever connected. To demonstrate the algorithm's operation, we illustrate Algorithm 2 step-by-step using a representative example with minimal connectivity.

Illustration of Algorithm 2 (see Figure 6):

1. Steps 1–7: We begin with the plane-connected graph $G_C(V_C, E_C)$ obtained from Algorithm 1, together with the list of articulation points and the set of user-specified non-adjacency constraints (see Figure 6(a)). The idea is to add a minimal extra edges so that the graph becomes bi-connected, while respecting the input constraints as far

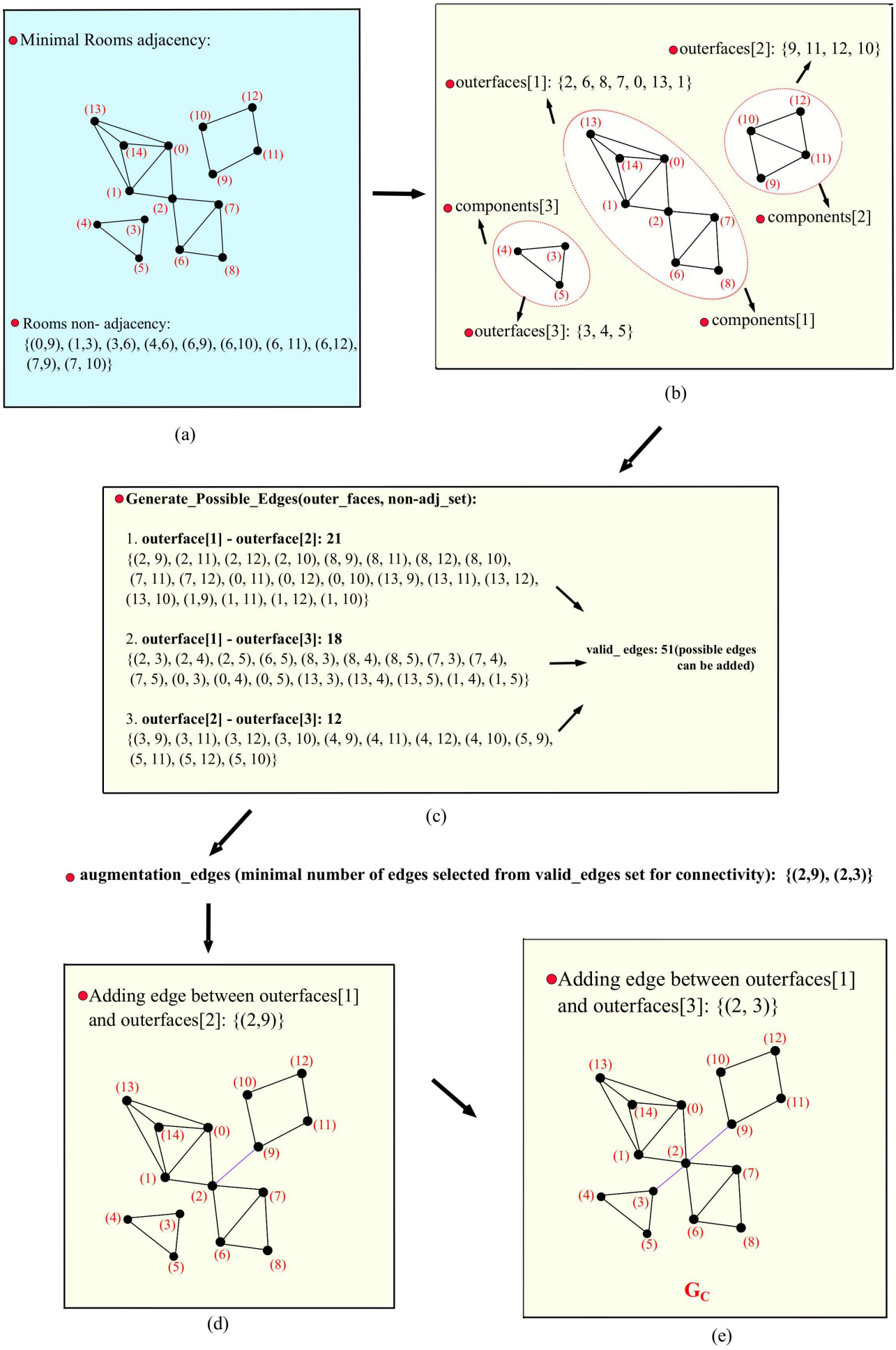

Fig. 5. (a-e) Construction of a connected graph G_C while preserving the specified input constraints.

Algorithm 2 : *Biconnectivity with non − adjacency*

function *Biconnectivity*($G_C(V_C, E_C)$,*articulation_points*, *non_adj_set*)
  selected_edges = {}.
  for $v \in$ *articulation_points* **do**
    blocks ← *Blocks*(G_C,v).
    valid_edges ← *Valid_Edges*(*blocks*, *non_adj_set*).
    bcn_edges ← *Connect_Blocks*(*blocks*,*valid_edges*).
    selected_edges ← *selected_edges* ∪ *bcn_edges*.
  $E \leftarrow E \cup$ *selected_edges*.
  return $G_B(V_B, E_B) = G_C(V_C, E_C)$.
  function *Blocks*(G,r)
    $G'(V', E') = G(V, E)$.
    $\{u_1, u_2, ..., u_k\} \leftarrow nbd(v)$, $V' \leftarrow V' - \{v\}$. ▷ *nbd(v) = vertices adjacent to v in G*
    for $i \leftarrow 1$ **to** k **do**
      $E' \leftarrow E' - \{(v, u_i)\}$.
    Components ← *Connected_components*(G'). ▷ *Connected_components*(G'): *Returns component: generator of sets.*
    for $i \leftarrow 0$ **to** *length*(*Components*) − 1 **do** ▷ *Compute outerface boundary of each component via planar embedding*
      blocks[i] ← *Extract_Outerface*(*Components*[i]).
    return *blocks*. ▷ $blocks = \{B_1, B_2, ..., B_k\}$.
  function *Valid_Edges*(*blocks*, *non_adj_set*)
    valid_edges = {}.
    for $i \leftarrow 0$ **to** *length*(*blocks*) − 1 **do**
      for $j \leftarrow i + 1$ **to** *length*(*blocks*) − 1 **do**
        $BL_1 \leftarrow$ *blocks*[i], $BL_2 \leftarrow$ *blocks*[j]
        for $u \in BL_1$ **do**
          for $v \in BL_2$ **do**
            if $(u, v) \notin$ *non_adj_set* **then**
              valid_edges ← *valid_edges* ∪ $\{(u, v)\}$.
    return *valid_edges*.
  function *Connect_Blocks*(*blocks*,*valid_edges*, *non_adj_set*)
    block_graph ← *Empty_Map*(). ▷ *Maps a block index to its corresponding set of adjacent blocks.*
    $b \leftarrow$ *length*(*blocks*).
    node_block ← *Empty_Map*(). ▷ *Associates each node with the block it belongs to.*
    for $i \leftarrow 0$ **to** $b - 1$ **do**
      for $u \in$ *blocks*[b] **do**
        node_block[u] ← b.
    parent ← *Array*[0, ..., $b - 1$].
    for $i \leftarrow 0$ **to** $b - 1$ **do**
      parent[i] ← i. ▷ *Every block initially serves as its own representative.*
    function *Find*[u] ▷ *To find the root of a block in connected components.*
      if *parent*[u] ≠ u **then**
        parent[u] = *Find*[*parent*[u]].
      return *parent*[u].
    function *Union*(u, v) ▷ *Union function to connect two blocks.*
      $root_u$ = *Find*[u], $root\_v$ = *Find*[v].
      if $root\_u \neq root\_v$ **then**
        parent[$root\_v$] = $root\_u$.
    valid_edges.sort(). ▷ *Sort valid edges by weight (default: unit weight).*
    selected_edges = {}.
    for $u, v \in$ *valid_edges* **do**
      block_u = *node_block*[u], *block_v* = *node_block*[v].
      if *Find*[*block_u*] ≠ *Find*[*block_u*] **then**
        Union(*block_u*, *block_v*).
        selected_edges ← *selected_edges* ∪ $\{(u, v)\}$.
        block_graph[*block_u*] ← *block_graph*[*block_u*] ∪ {*block_v*}.
        block_graph[*block_v*] ← *block_graph*[*block_v*] ∪ {*block_u*}.
    connected_blocks ← $|\{$ *Find*[u] $\mid u \in [0, b - 1]\}|$. ▷ *Check if all blocks are connected.*
    if *connected_blocks* > 1 **then** ▷ *fallback: Add non-adjacent edges if necessary to connect all blocks.*
      for $u, v \in$ *non_adj_set and* $u, v \in$ *blocks* **do**
        block_u = *node_block*[u], *block_v* = *node_block*[v].
        if *Find*[*block_u*] ≠ *Find*[*block_u*] **then**
          Union(*block_u*, *block_v*), *selected_edges* ← *selected_edges* ∪ $\{(u, v)\}$.
          block_graph[*block_u*] ← *block_graph*[*block_u*] ∪ {*block_v*}, *block_graph*[*block_v*] ← *block_graph*[*block_v*] ∪ {*block_u*}.
    return *selected_edges*.

as possible. An empty set *selected_edges* is created to store all final *bcn_edges*. The algorithm then scans the list of articulation points one by one. For a fixed articulation point v, the procedure `Blocks`(G_C, v) is called to see how the neighbourhood of v breaks into connected components when v is removed, and `Valid_Edges` lists all permitted candidate edges between those component vertices while satisfying non-adjacency constraints. Finally, `Connect_Blocks` chooses a suitable subset of these candidate edges (i.e., the minimal number of edges added to make the graph bi-connected), and the chosen edges are added to *selected_edges*.

2. Steps 8–9: After all articulation points have been processed, the set *selected_edges* contains every new edge that we intend to insert. These are now added to the original edge set E_C to form $E' = E_C \cup selected\_edges$, and the resulting graph is denoted by $G_B(V_B, E_B)$. In the running example, the red edges appearing in Figure 6(e) are exactly those stored in *selected_edges*. The graph G_B is the bi-connected graph that will be passed to the triangular phase construction step.

3. Steps 10–18: The helper function `Blocks`(G_C, v) determines how the graph around v is organized. Conceptually, we delete v and all edges incident to it from G_C, and then compute the connected components of the remaining graph G'. For each such component, we will compute its outerface vertices and record them as a block, so we obtain a collection

$$blocks = \{B_1, B_2, \ldots, B_\ell\}.$$

These blocks represent the regions that would separate if v were removed. For the example in Figure 6, when $v = 9$ we obtain two blocks

$$blocks[1] = \{10, 11, 12\}, \qquad blocks[2] = \{0, 13, 1, 2, 6, 8, 7, 3, 4, 5\},$$

as shown in Figure 6(b). The block decompositions for $v = 3$ and $v = 2$ are displayed in Figures 6(c) and 6(d).

4. Steps 19–28: Once the blocks are known, the function `Valid_Edges(blocks, non_adj_set)` builds a pool of valid edges that do not break the non-adjacency rules. For every unordered pair of distinct blocks B_i and B_j, it looks at all vertex pairs (u, w) with $u \in B_i$ and $w \in B_j$. If (u, w) appears in the non-adjacency set, that pair is discarded; otherwise, it is inserted into the set *valid_edges*. In this way, *valid_edges* collects every allowed way of joining different blocks around the same articulation point. The edges that are finally drawn in Figures 6(b)–(d) (such as $(0, 10)$ for vertex 9 or $(4, 2)$ for vertex 3) come from this pool.

5. Steps 29–63: The primary task of determining which edges should be retained is carried out by the function *Connect_Blocks*. Initially, each block is placed in its own set in a union-find data structure, and for each vertex, we store the index of the block it belongs to. The edges in *valid_edges* are then inspected one by one (since all edges have equal weight in our setting, any fixed ordering is sufficient). For each candidate edge (u, w), we identify the blocks *block_u* and *block_w* that contain u and w. If these blocks currently reside in distinct union find sets, the edge (u, w) is accepted: it is appended to *selected_edges*, the two sets are merged via a `Union` operation, and the auxiliary block graph is updated accordingly. This accept-then-merge process continues until no remaining candidate edge can reduce the number of disconnected block sets. If multiple block sets are still disconnected at this point, the fallback mechanism of our proposed algorithm makes another pass over all admissible pairs to connect the remaining components of the block graph, while ignoring the non-adjacency constraints.
In the Example 6, for articulation point 9 there are 27 $valid_edges$ set (It is generated in the same manner that we show during identifying $valid_edges$ in Algorithm 1: see Figure 5c). The edge $(0, 10)$ unifies as blocks *blocks*[1] and *blocks*[2] (Figure 6(b)). For instance, $(10, 0)$ (belongs to $valid_edges$) is admitted into *bcn_edges* because vertex 10 belongs to blocks[1] and vertex 0 belongs to blocks[2], and this pairing does not violate any non-adjacency rules defined by the user. After this merge, the next candidate edge, such as $(10, 13)$ (which belongs to *valid_edges*), is rejected because 10 and 13 now lie in the same merged set. This merging process continues across the remaining candidate edges in the *valid_edges* set until all components are combined into a single connected graph. Only the edges that connect different components are added to *bcn_edges*.
Similarly, for articulation point 3, the edge $(4, 2)$ performs the merging of its two adjacent blocks (see Figure 6(c)). A similar process for articulation point 2 yields a small collection of edges such as $(6, 1)$, $(6, 5)$, and $(7, 11)$ that ensure all blocks incident to vertex 2 ultimately merge into one merged block (Figure 6(d)). Each edge stored in *selected_edges* therefore satisfies the non-adjacency rules, joins vertices from different blocks of some articulation

point, and decreases the number of disconnected block sets. Combining all steps, Algorithm 2 converts the connected graph G_C generated by Algorithm 1 into the bi-connected graph G_B shown in Figure 6(e). The resulting graph G_B contains no articulation points and thus serves as the robust starting structure for the subsequent triangulation algorithm.

By applying Algorithm 2, we obtain a bi-connected graph G_B derived from the input graph G_C together with the stated non-adjacency constraints (see Figure 6(a–e)). Since we are treating the input graph as a plane drawing, the outer face of each connected component is well-defined. In Algorithm 2, the vertices incident to the outer face are identified prior to any augmentation, and all additional edges are inserted only afterward; consequently, the extraction of outer-face vertices is independent of the augmentation step and does not require recomputation or planarity verification. Once the final bi-connected graph G_B is obtained, a plane embedding of G_B serves as the input for the subsequent algorithm, which further processes it to achieve triangulation.

6.3 Bi-connected to Triangulation

After constructing a bi-connected graph from the user-specified door connectivity, the next stage performs triangulation to introduce interior wall adjacencies, eliminating unused space within the rectangular floor plan. Failure to triangulate, i.e., leaving any face non-triangular, can cause the floor plan/layout generation procedure to create voids or inefficiencies (see Figure 7). To address this, we proposed Algorithm 3 that triangulates a bi-connected graph while respecting user-defined non-adjacency constraints. The method compiles all non-triangular faces and non-adjacent edges, then iteratively decomposes each face into triangles. Using an ear-clipping strategy inspired by Mei et al. [35], it attempts to insert diagonals that neither violate user-imposed adjacency restrictions nor introduce disallowed/non-adjacency edges. When such valid diagonals exist, they are added to the graph, incrementally refining spatial relationships as illustrated in Figure 8, where six interior adjacencies are introduced. If a face admits no diagonal other than those explicitly prohibited non-adjacency edges, the algorithm temporarily relaxes these restrictions to ensure the graph becomes fully triangulated. This fallback mechanism guarantees a complete triangulation while preserving user constraints whenever feasible. To demonstrate the algorithm's operation, we illustrate Algorithm 3 step-by-step using a representative Bi-connectivity graph as input with respect to an example 8.

Illustration of Algorithm 3 (see Figure 8):

1. Steps 1–7: Algorithm 3 takes as input the bi-connected room adjacency graph $G_B(V_B, E_B)$ produced by Algorithm 2, together with the input non-adjacency constraints (see Figure 8(a)). The first task is to identify all faces of the input plane graph G_B that are not yet triangles. The procedure `Find_Faces`(G_B) is called, and its output is stored in the list `faces`. For each face, the ordered boundary vertices are obtained via `Vert(face)`. Whenever the boundary contains more than three vertices, and the face is not the outer boundary (checked by `Is_Exterior_Face`), the corresponding vertex list is appended to `non_tri_faces`. In the running example, this produces five non-triangular faces, collected as

$$\texttt{non\_tri\_faces[0]} = \{10, 9, 11, 12\},\ \texttt{non\_tri\_faces[1]} = \{0, 2, 9, 10\},\ \texttt{non\_tri\_faces[2]} = \{9, 2, 7, 11\},$$

$$\texttt{non\_tri\_faces[3]} = \{3, 5, 6, 2\},\ \texttt{non\_tri\_faces[4]} = \{6, 5, 4, 2, 1\},$$

as shown in Figure 8(b). These faces are exactly the regions that must be triangulated in the subsequent steps.

2. Steps 8–11: After the list of non-triangular faces has been obtained, an empty set `Triangulate_edges` is created to store all new identified edges that will be added further. The function `ClipTriangulation`(`non_tri_faces`, G_B, `non_adj_set`) is then invoked. It returns a set of interior new edges that triangulate every face in `non_tri_faces` while satisfying the given non-adjacency constraints. These diagonals are stored in `Triangulate_edges` and then added to the existing edge set, $E \leftarrow E \cup$ `Triangulate_edges`. The resulting triangulated graph, denoted G_T, is depicted in Figure 8(h), where the newly introduced edges are highlighted in green.

3. Steps 12–27: The function `Find_Faces`(G_B) computes all faces of a given plane graph G_B. It begins with a temporary edge set E', which initially contains all edges of G_B, and an empty list `faces`. While E' is not empty, the algorithm selects an edge $s \in E'$ and starts tracing the boundary of a face. A list P is created to store the edges

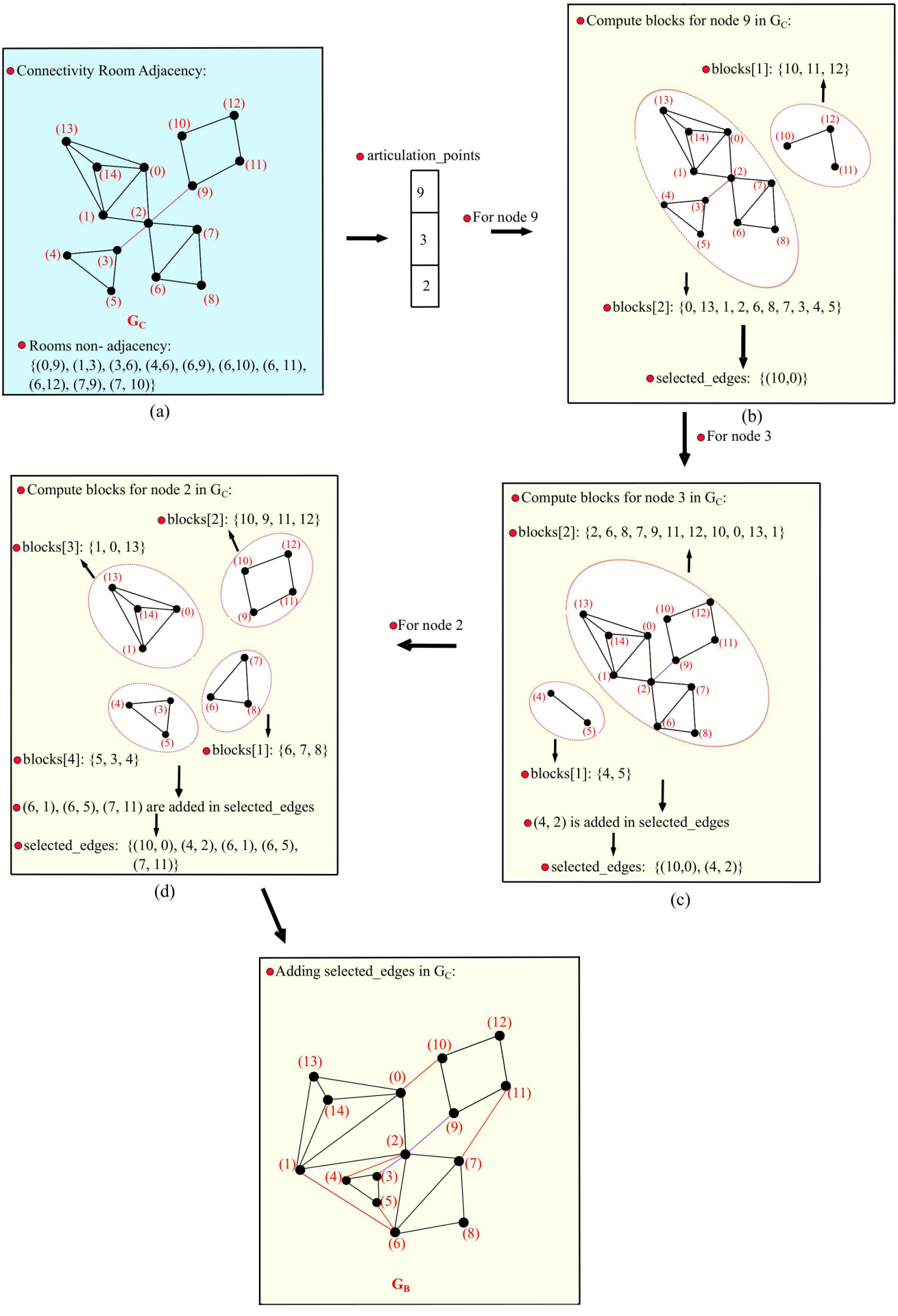

Fig. 6. (a-e) Construction of a Bi-connectivity graph G_B while preserving the specified input constraints.

Algorithm 3 : *Triangulation with non-adjacency*

```
function Triangulation(G_B(V_B, E_B), non_adj_set)
    faces ← Find_Faces(G_B).
    non_tri_faces ← [].
    for each face ∈ faces do                                   ▷ Vert(face): Returns the list of vertices forming the face.
        vertices ← Vert(face).    ▷ Is_Exterior_Face: Checks if the given vertices form the outer boundary of the graph.
        if vertices > 3 and Is_Exterior_Face(vertices, G_B) = false then
            non_tri_faces.append(vertices).
    Triangulate_edges ← [].
    Triangulate_edges ← Clip_Triangulation(non_tri_faces, G_B, non_adj_set).
    E ← E ∪ Triangulate_edges.
    return G_T(V_T, E_T) = G_B(V_B, E_B).
    function Find_Faces(G_B(V, E))
        faces ← [], E' = E.
        while E' ≠ ∅ do
            Choose s ∈ E'.
            P ← [s].
            c ← s.
            while true do          ▷ Get_Next_Edge: Given e = (u, v), returns the next edge (v, w) around v in G_B.
                n ← Get_Next_Edge(c, G_B).
                if n = s then
                    faces.append(P).
                    break
                else
                    P.append(n).
                    c ← n.
                    E'.remove(n).
        return faces.
    function Clip_Triangulation(non_tri_faces, G_B, non_adj_set)
        new_edges ← [].
        for each face ∈ non_tri_faces do
            vertices ← Vert(face).          ▷ Vert(face): Returns the ordered list of vertices forming the face.
            while |vertices| > 3 do
                clip_found ← false.                                  ▷ Clip_found is a boolean variable
                for i ← 0 to |vertices| − 1 do
                    (a, b, c) ← Consecutive(vertices, i).
                    if Is_Valid_Diagonal(a, c, vertices, face, non_adj_set) then
                        new_edges ← new_edges.append((a, c)).
                        vertices ← vertices.remove(b).
                        clip_found ← true.
                        break.
                if clip_found == false then          ▷ fallback: To connect all blocks ignore non-adjacency constraints.
                    (a, b, c) ← Consecutive(vertices, 0).
                    new_edges ← new_edges.append((a, c)).
                    vertices ← vertices.remove(b).
        return new_edges.
    function Consecutive(vertices, i)                ▷ vertices = (v_1, v_2, ..., v_l) is an ordered list of vertices
        a ← v_i.
        b ← v_{(i+1) mod l}.
        c ← v_{(i+2) mod l}.
        return (a, b, c).
    function Is_Valid_Diagonal(a, c, vertices, face, non_adj_set)
        if (a, c) ∈ non_adj_set then
            return false.
        for each edge (v_i, v_{i+1}) ⊆ vertices do          ▷ Check that diagonal does not intersect any face edges
            if (a, c) intersects (v_i, v_{i+1}) and (v_i, v_{i+1}) ≠ (a, b), (b, c) then
                return false.
        if (a, c) ∈ Diag(vertices)  and  (a, c) ⊆ int(vertices) then      ▷ Check that diagonal is inside the polygon face.
            return true.                          ▷ Diag(vertices) denotes the set of all possible diagonals of polygon face.
        else                                      ▷ int(vertices) denotes the interior region of polygon face.
            return false
```

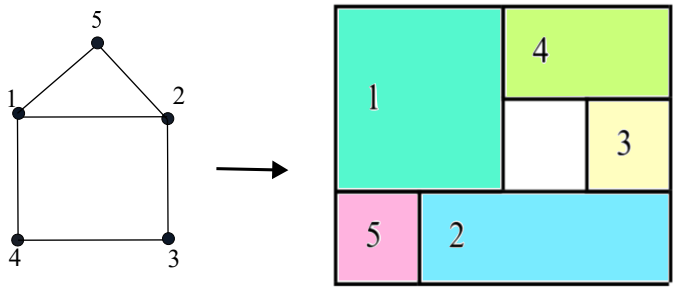

Fig. 7. Additional space is created in the floor plan corresponding to the graph with a non-triangular face (1,2,3,4).

of the current cycle, and a pointer e is set to s. At each step, the function `Get_Next_Edge`(e, G_B) returns the next edge in the cyclic order around the current vertex, according to the fixed planar embedding. If the returned edge equals the starting edge s, the traversal of the face is complete, and the cycle P is added to `faces`. Otherwise, the new edge is appended to P, the pointer e is updated, and the edge is removed from E'. This process continues until all edges have been assigned to boundary cycles. As a result, the algorithm produces one edge-cycle for each face of the planar embedding. In later steps, these cycles are filtered to obtain the set `non_tri_faces`.

4. Steps 28–45: The main triangulation part is performed by function `ClipTriangulation`, which applies an ear-clipping strategy to each non-triangular face while respecting the non-adjacency constraints provided by user. For every entry in `non_tri_faces`, the ordered boundary is stored in the list `vertices`. As long as the current face has more than three vertices, the algorithm attempts to clip an ear: a triple of consecutive vertices (a, b, c) for which the diagonal (a, c) is a valid internal diagonal. At the start of each outer iteration, the Boolean variable `clip_found` is set to `false`. The inner loop (Steps 34–38) scans all positions i along the boundary; for a given i, the function `Consecutive`(`vertices`, i) returns the triple (a, b, c) consisting of v_i, v_{i+1} and v_{i+2} (indices modulo the current boundary length). The candidate diagonal (a, c) is then checked by `Is_Valid_Diagonal`. If the test succeeds, (a, c) is appended to `new_edges`, the ear vertex b is removed from `vertices`, and `clip_found` is set to `true`; the algorithm then proceeds with the remaining smaller polygon. If no valid ear is found in a complete pass (`clip_found` remains `false`), the algorithm clips the first triple returned by `Consecutive`(`vertices`, 0) as a fallback (ignoring the non-adjacency constraints), adds the diagonal (a, c), and removes b. This process continues until each face is reduced to a single triangle. The diagonals accumulated in `new_edges` across all faces form `Triangulate_edges`.
Figures 8(c)–(g) illustrate this behaviour: for `non_tri_faces[0]` $= \{10, 9, 11, 12\}$ (Figure 8(c)), the triple $(10, 9, 11)$ yields the accepted diagonal $(10, 11)$ in the first iteration; for `non_tri_faces[1]`$=\{0, 2, 9, 10\}$ (Figure 8(d)), the candidate $(0, 9)$ is rejected in the first iteration (since $(0, 9)$ is part of non-adjacency constraints), whereas $(2, 10)$ is accepted in the second iteration; similar clipping steps on the remaining faces (Figures 8(e)–(g)) produce diagonals/edges such as $(0, 11)$, $(5, 2)$ and $(1, 5)$. For `non_tri_faces[1]`$=\{6, 5, 4, 2, 1\}$ (see Figure 8(g)), the possible edge $(5, 2)$ is explicitly rejected because it does not form an internal diagonal of the corresponding face, and therefore fails the `Is_Valid_Diagonal` test.

5. Steps 46–60: The final two functions describe how candidate edges/diagonals are selected and validated. The function `Consecutive`(`vertices`, i) returns the triple (a, b, c), where a is the vertex at position i, b is its successor, and c is its second successor along the current boundary.
Formally, $a = v_i$, $b = v_{(i+1) \bmod t}$, and $c = v_{(i+2) \bmod t}$ for a boundary of length t. The function `Is_Valid_Diagonal`$(a, c,$ `vertices`, f checks whether the diagonal (a, c) can be inserted. First, the diagonal is rejected if (a, c) appears in the non-adjacency set (see Figure 8(b), and similar cases in Figure 8(d)–(f)). Next, the algorithm tests whether (a, c) properly intersects any boundary edge (v_i, v_{i+1}), excluding the incident edges (a, b) and (b, c). If an intersection occurs, the diagonal is rejected. Finally, the algorithm verifies that (a, c) is a valid interior diagonal of the polygon and lies strictly inside the face (see Figure 8(g)). The function returns `true` only if all checks are satisfied.
As a result, every diagonal stored in `Triangulate_edges` is an interior edge that does not cross existing edges and respects all non-adjacency constraints. After inserting these diagonals into G_B, the triangulated graph G_T is obtained (Figure 8(h)), where every bounded face is a triangle. This graph is then ready for the subsequent separating-triangle removal step required for rectangular layout construction.

By applying Algorithm 3, we obtain a bi-connected and triangulated graph G_T derived from the input graph G_B together with the stated non-adjacency constraints (see Figure 8(a–h)). As the input graph is provided with a plane graph G_B, every new edge added within a non-triangular face is checked at the time of insertion to avoid any crossings with existing edges (Steps 51-60 of Algorithm 3). This guarantees that planarity is maintained throughout the construction. Therefore, after obtaining the final graph G_T, no separate planarity test is necessary, and the algorithm proceeds directly to the subsequent phase of graph generation.

6.4 Identification and Removal of Separating triangles

Once a plane-biconnected and triangulated graph G_T has been constructed, the next stage of floor-plan generation depends on the user-specified geometric constraints. If no restriction is imposed on module shape (that is, rectilinear shapes such as L-, T-, or C-forms are allowed), the general floor-planning procedure described in Subsection 6.5.2 can be applied directly. However, if each module is required to be rectangular, the graph G_T must first be examined for separating triangles. Since separating triangles are not compatible with the rectangular floor-plan topology, they must be removed before layout construction. In a rectangular layout, each region is represented by an axis-aligned rectangle with straight and contiguous boundaries. Under this representation, three regions cannot form a triangular cycle that encloses another region (see Figure 9). It is well established that graphs supporting rectangular floor plans must be free of separating triangles [36].
To detect these structures, we apply Johnson's algorithm [37] to enumerate all separating triangles, together with the user-defined non-adjacency constraints. For each detected triangle, one of its outer edges is selected for removal and replaced with a new diagonal edge. This replacement is accepted only if it satisfies the non-adjacency constraints and does not create additional separating triangles. If the condition is not satisfied, alternative acceptable edges are considered.

To eliminate a separating triangle S in a graph G, we begin by selecting an edge (a separating triangle S is bounded by three edges) incident to S and determining whether this edge lies on the boundary of the outer face of G; exterior edges may be removed directly. When all three edges of S are interior, additional structural checks become necessary. The acceptable/candidate edge must be examined to verify that its removal indeed resolves the separating triangle, does not introduce a new separating cycle of the same type, and does not violate any non-adjacency constraints. Algorithm 4 organizes the decision process into a structured set of cases, each paired with a verified modification. This approach eliminates the need to separate triangles while retaining the graph's structural integrity, which is required for the next stage of layout generation. As illustrated in Figure 10 (c-d), resolving the separating triangle (10, 2, 11) involved removing edge (10, 11) and inserting edge (9, 12), thereby preserving triangulation while eliminating the separating structure. Repeating this process yields a set of triangulated graphs free of separating triangles, i.e., PTPG (properly triangulated plane graph) graphs, which then serve as the basis for subsequent rectangular floor plan generation.

Illustration of Algorithm 4 (see Figure 10):

1. Steps 1–7: Algorithm 4 starts from the triangulated room adjacency graph $G_T(V_T, E_T)$ returned by Algorithm 3, together with the room non-adjacency set (Figure 10(a)). The main function `ST_Edge_Removal` first calls `ST`(G_T) to list all separating triangles of G_T; this list is stored in T, and its size $m = |T|$ is the current number of separating triangles. The edge set of G_T is copied to a new set E'_T, and a reference copy G_1 of the original triangulated graph G_T is created. In the running example, `ST`(G_T) returns the five separating triangles shown in Figure 10(b): $\{0, 13, 1\}$, $\{10, 2, 11\}$, $\{2, 4, 5\}$, $\{1, 5, 2\}$ and $\{2, 1, 6\}$. These are the cycles that must be broken in the subsequent steps so that the final graph has no separating triangles. The function `Remove_ST_Edge_Removal`$(T, G_T, m, E'_T, G_1, |V|,$ `non_adj_set`$)$ is then called once (and, if needed, a second time) to modify G_T until all these separating triangles are removed. The graph G_F returned by `ST_Edge_Removal` is the separating triangle-free graph used later for rectangular layout construction.

2. Steps 8–22: Inside `Remove_ST_Edge_Removal`, the algorithm processes the separating triangles one by one. For a fixed triangle $T \in T$, the flag `removed` is set to `false` and the local set E'_T is initialised with the three edges of T. The function first attempts a strategy (Step 1 in the comments): delete an exterior edge of T without adding any new edges. To do this, it scans all edges $e = (x, y)$ of G_T and only considers those belonging to E'_T. For each such edge, it computes the common neighbours N_{xy} of x and y in G_T. If $|N_{xy}| = 2$, the edge (x, y) is incident to

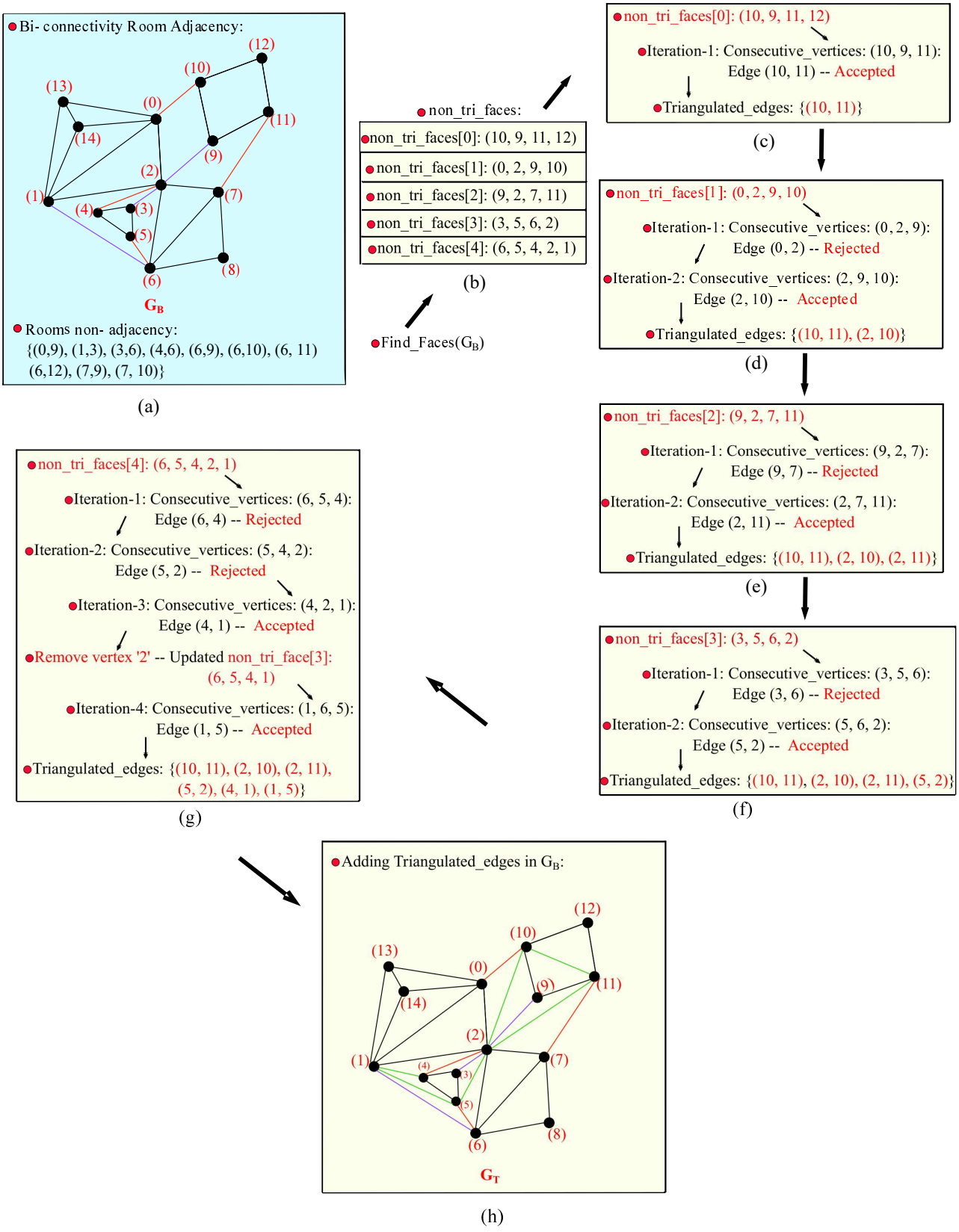

Fig. 8. (a-h) Construction of a bi-connected triangulation graph G_T while preserving the specified input constraints.

Algorithm 4 Separating triangles removal with non-adjacency

function ST_Edge_Removal(G_T, non_adj_set)
  $\mathcal{T} \leftarrow \mathrm{ST}(G_T)$. ▷ *$ST(G_T)$: return all separating Triangles in G_T*
  $m \leftarrow |\mathcal{T}|$, $E' = E_T$, $G_1(V_1, E_1) = G_T(V_T, E_T)$. ▷ *number of separating triangles*
  $(m, G_F) \leftarrow Remove\_ST\_Edge\_Removal(\mathcal{T}, G_T, m, E', G_1, |V|, non\_adj\_set)$.
  if $m > 0$ **then**
    $E' = \phi$, $(m, G_F) \leftarrow Remove\_ST\_Edge\_Removal(\mathcal{T}, G_T, m, E', G_1, |V|, non\_adj\_set)$.
  return G_F *(Free from separating triangle)*.
  function $Remove\_ST\_Edge\_Removal(\mathcal{T}, G_T, m, E', G_1, |V|, non\_adj\_set)$
    for all $T \in \mathcal{T}$ **do**
      removed $\leftarrow$ false. ▷ *removed is a boolean variable.*
      $E_T \leftarrow \mathrm{Edges}(T)$. ▷ *return edges in separating triangle T.*
      for all edge $e = (x, y) \in E_T$ **do** ▷ *Step 1: try removing an exterior edge*
        if $(x, y) \in E'$ **then**
          continue
        $N_{xy} \leftarrow \mathrm{Common\_Neighbors}(G_T, x, y)$. ▷ *returns common neighbours of x and y*
        if $|N_{xy}| = 2$ **then** ▷ *edge $\{x, y\}$ is an exterior edge.*
          $E \leftarrow E$ - $\{(x, y)\}$, $\mathcal{T} \leftarrow \mathrm{ST}(G_T)$, $m \leftarrow m - 1$.
          removed $\leftarrow$ true.
          break
      if *removed* **then**
        continue ▷ *Step 2: try edge replacement $(x, y) \mapsto (r, i)$ respecting non_adj_set*
      edgeFound $\leftarrow$ false.
      for all edge $e = (x, y) \in E_T$ **do**
        if $(x, y) \in E'$ **then**
          continue ▷ *T: separating triangle formed by cycle of vertices x, y, z with interior vertices $v_1, ...v_n$.*
        for all $r \in \mathrm{Common\_Neighbors}(G_T, x, y)$ **do**
          $d \in \mathrm{nbd}(x) \cap \mathrm{nbd}(y) \cap \mathrm{nbd}(z)$.
          if $r \neq d$ **and** $r \notin T$ **and** $(r, d) \notin E_T$ **then**
            if $(r, d) \notin$ non_adj_set **then**
              $G'_T \leftarrow G_T \cup \{(r, d)\} \setminus \{(x, y)\}$.
              $\mathcal{T}' \leftarrow \mathrm{ST}(G'_T)$, $m' \leftarrow |\mathcal{T}'|$.
              if $m' \leq m - 1$ **and** `is_planar`(G'_T) **then**
                $G_T \leftarrow G'_T$, $\mathcal{T} \leftarrow \mathcal{T}'$, $m \leftarrow m'$, *edgeFound* $\leftarrow$ true.
                break.
              else
                $G_T \cup \{(x, y)\} \setminus \{(r, d)\}$.
        if *edgeFound* **then**
          break
      if *edgeFound* **then**
        continue ▷ *Step 3 (fallback): ignore non-adj-constraints and retry replacements*
      for all edge $e = (x, y) \in E_T$ **do**
        if $(x, y) \in E'$ **then**
          continue. ▷ *T: separating triangle formed by cycle of vertices x, y, z with interior vertices $v_1, ...v_n$.*
        for all $r \in \mathrm{Common\_Neighbors}(G_T, x, y)$ **do**
          $d \in \mathrm{nbd}(x) \cap \mathrm{nbd}(y) \cap \mathrm{nbd}(z)$.
          if $r \neq d$ **and** $r \notin T$ **and** $(r, d) \notin E_T$ **then**
            $G'_T \leftarrow G_T \cup \{(r, d)\} \setminus \{(x, y)\}$.
            $\mathcal{T}' \leftarrow \mathrm{ST}(G'_T)$, $m' \leftarrow |\mathcal{T}'|$.
            if $m' \leq m - 1$ **and** `is_planar`(G'_T) **then**
              $G_T \leftarrow G'_T$, $\mathcal{T} \leftarrow \mathcal{T}'$, $m \leftarrow m'$, *edgeFound* $\leftarrow$ true.
              break.
            else
              $G_T \cup \{(x, y)\} \setminus \{(r, d)\}$.
    return (m, G).

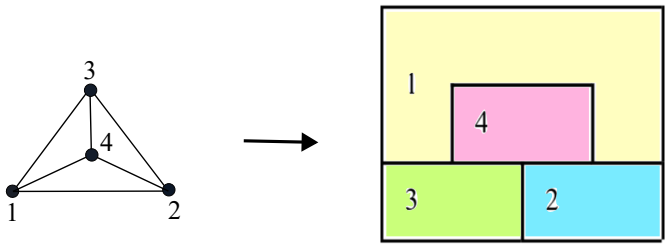

Fig. 9. A non-rectangular room (Room 1) appears in the floor plan due to the presence of a separating triangle (1,2,3) in the input graph.

exactly two triangular faces and is therefore an exterior edge of T; in that case the edge is removed from E_T, the list of separating triangles is recomputed as $T \leftarrow \texttt{ST}(G_T)$, the counter m is replaced by the new value $m' = |T|$, and `removed` is set to `true`. The algorithm then goes directly to the next triangle. Figure 10(c) illustrates this case for `Sep_Tri[0]` $= \{0, 13, 1\}$: the edge $(0, 13)$ is recognised as an exterior edge and deleted, which removes the separating triangle without introducing any new one.

3. Steps 23–41: If no exterior edge of a separating triangle T can be safely removed (that is, the flag `removed` remains `false`), the algorithm proceeds to a more general resolution strategy (Step 2), in which an edge of T is replaced by a new edge while still respecting the given non-adjacency constraints. A boolean flag `edgeFound` is initialized to `false`. The algorithm then considers each edge $e = (x, y)$ of T as a potential candidate edge for replacement. For a selected edge (x, y), it first identifies vertices r that are common neighbours of x and y, and then identifies vertices d that are adjacent to x, y, and the third vertex z of T; such vertices d lie in the interior region enclosed by T. Whenever $r \neq d$, r is not a vertex of T, and the edge (r, d) is not already present in E_T, the pair (r, d) is considered as a possible replacement for (x, y).
Each candidate edge (r, d) is first checked against the non-adjacency constraints and discarded if it violates any user-specified restriction. Otherwise, a temporary graph G'_T is constructed by inserting (r, d) and removing (x, y). The set of separating triangles in G'_T is then recomputed as $T' = \texttt{ST}(G'_T)$, with $m' = |T'|$. If this modification strictly reduces the number of separating triangles ($m' \leq m - 1$), the replacement is accepted: G_T is updated to G'_T, the current triangle set is replaced by T', m is updated to m', and `edgeFound` is set to `true`, allowing the algorithm to continue with the next separating triangle.
Figure 10(d) illustrates such a successful constrained replacement for `Sep_Tri`$[1] = \{10, 2, 11\}$. In this example, several candidate replacements involving edges incident to $(10, 2)$ and $(2, 11)$ are rejected until the algorithm removes the edge $(11, 10)$ and inserts $(9, 12)$. This modification respects the non-adjacency constraints and eliminates the separating triangle without introducing new ones.

4. Steps 42–54: The final stage of the procedure introduces a fallback strategy (Step 3), which is invoked only when Step 2 fails to identify any admissible edge replacement that satisfies the non-adjacency constraints, that is, when `edgeFound` remains `false` after the constrained search. In this stage, the algorithm repeats the examination of triangle edges and candidate vertex pairs, but the non-adjacency check $(r, d) \in$ `non_adj_set` is deliberately omitted. For each admissible pair (r, d) with $r \neq d$, $r \notin T$, and $(r, d) \notin E_T$, a temporary graph G'_T is formed by inserting (r, d) and removing (x, y). The separating triangles of G'_T are then recomputed as $T' = \texttt{ST}(G'_T)$, with $m' = |T'|$. If this modification strictly reduces the number of separating triangles ($m' \leq m - 1$), the replacement is accepted following the same update procedure used in Step 2.
This fallback mechanism guarantees progress even when all admissible replacements are excluded by non-adjacency constraints. Its effect is illustrated in Figures 10(e) and 10(f). For the separating triangles $\{2, 4, 5\}$ and $\{1, 5, 2\}$, every constraint-respecting candidate is rejected, prompting Step 3 to bypass the non-adjacency set and replace the edge $(5, 2)$ with $(3, 6)$, thereby eliminating both triangles simultaneously (Figure 10(e)). For the remaining triangle $\{2, 1, 6\}$ (Figure 10(f)), the algorithm identifies $(1, 6)$ as an exterior edge and removes it, eliminating the final separating triangle. After all triangles in T have been processed, the graph G_T evolves into the final graph G_F shown in Figure 10(f), which contains no separating triangles. Algorithm 4 therefore produces a triangulated, non-separating graph that respects non-adjacency constraints whenever possible and serves as a valid combinatorial foundation for

the subsequent rectangular layout construction.

Applying Algorithm 4 transforms the triangulated input graph G_T, together with the specified non-adjacency constraints, into a properly triangulated plane graph G_F (see Figure 10(a–f)). Since G_T is given with a fixed plane embedding, any edge introduced during the elimination of separating triangles is inserted only within non-triangular faces created by earlier edge removals. Each such insertion is verified at the time of addition to ensure that it does not intersect existing edges, thereby preserving planarity throughout the process. After construction terminates, the resulting graph G_F admits a planar embedding, which is used directly as the combinatorial input for the subsequent floor-plan generation phase.

6.5 Graph to Floorplan Generation

6.5.1 RFP Generation: After removing all separating triangles from the triangulated graph G_T (see Subsection 6.4), the resulting graph G_F is a properly triangulated plane graph (PTPG) that satisfies the user input constraints. Such graphs meet the structural conditions required for rectangular dual construction, as discussed by Koźmiński et al. [1]. These PTPGs therefore serve as input for rectangular floor-plan generation.
A *dimensionless floor plan* (UFP) is a layout in which each room is modeled as an axis-aligned rectangle, and only adjacency relations between rectangles are fixed; exact dimensions and areas are not assigned. This abstraction allows designers to examine circulation patterns, relative room placement, and orientation without committing to precise measurements. It also supports comparing alternative layouts and detecting adjacency conflicts early, before geometric refinement.
Graphs provide a clear way to represent which rooms must be adjacent and which rooms must remain separate. Therefore, graph-based approaches are well-suited for generating families of dimensionless floor plans. In our implementation, the PTPG constructed in Section 6.5 is used as input to the algorithm of Kant et al. [38], which produces a rectangular representation where each rectangle corresponds to a room.
The resulting UFP serves as a structural blueprint that can later be scaled and detailed. For the example PTPG shown in Figure 10f, the corresponding UFPs are presented in Figure 11. Although these layouts share the same underlying adjacency graph, they differ in geometric arrangement. For instance, rooms 4 and 1 share a vertical wall segment in one layout and a horizontal wall segment in another. More generally, the alternatives vary in relative room placement and shared wall segments while preserving the same adjacency structure.

6.5.2 OFP Generation: In our proposed prototype, designers and architects can generate either rectangular floor plans (RFPs) or orthogonal floor plans (OFPs). When the OFP option is selected, modules can take general rectilinear shapes rather than being restricted to rectangles. See Figure 12: Starting from the generated graph G_T, the construction follows the methodology described by Krishnendra et al. [14]. As established by the theorem of Kant and He [38], a bi-connected planar triangulated graph containing a separating triangle cannot admit an RFP. Accordingly, in such cases, a generated PTPG G_T is augmented by introducing auxiliary vertices and further triangulated so that the resulting modified graph is free of separating triangles. An RFP is then constructed for this modified graph, and the regions corresponding to the auxiliary vertices are subsequently merged with suitable neighboring rooms to obtain an OFP for the original G_T. This sequence of transformations, illustrated in Figure 12 (a-d), systematically removes separating triangles while preserving planarity and adjacency constraints. Furthermore, since a single modified graph may yield multiple valid RFP realizations, selectively merging auxiliary modules enables the generation of multiple distinct OFPs, as shown in Figure 12 (e-m).

7 Practical Use of the DPLAN Interface

This section illustrates the practical use of the proposed DPLAN prototype by highlighting how automatically generated residential floor plans/layouts can be refined through user-driven customization. After the system produces an initial rectangular layout, users can modify the layout boundaries and room geometries to accommodate individual design preferences and functional constraints, including creating non-rectangular spaces that extend beyond standard configurations. Figures 13 and 14 present representative examples of two- and three-bedroom residential layouts, respectively, demonstrating how variations in input constraints, such as adjacency and non-adjacency relations between rooms, as well as room shape specifications, yield distinct customized outcomes. These examples

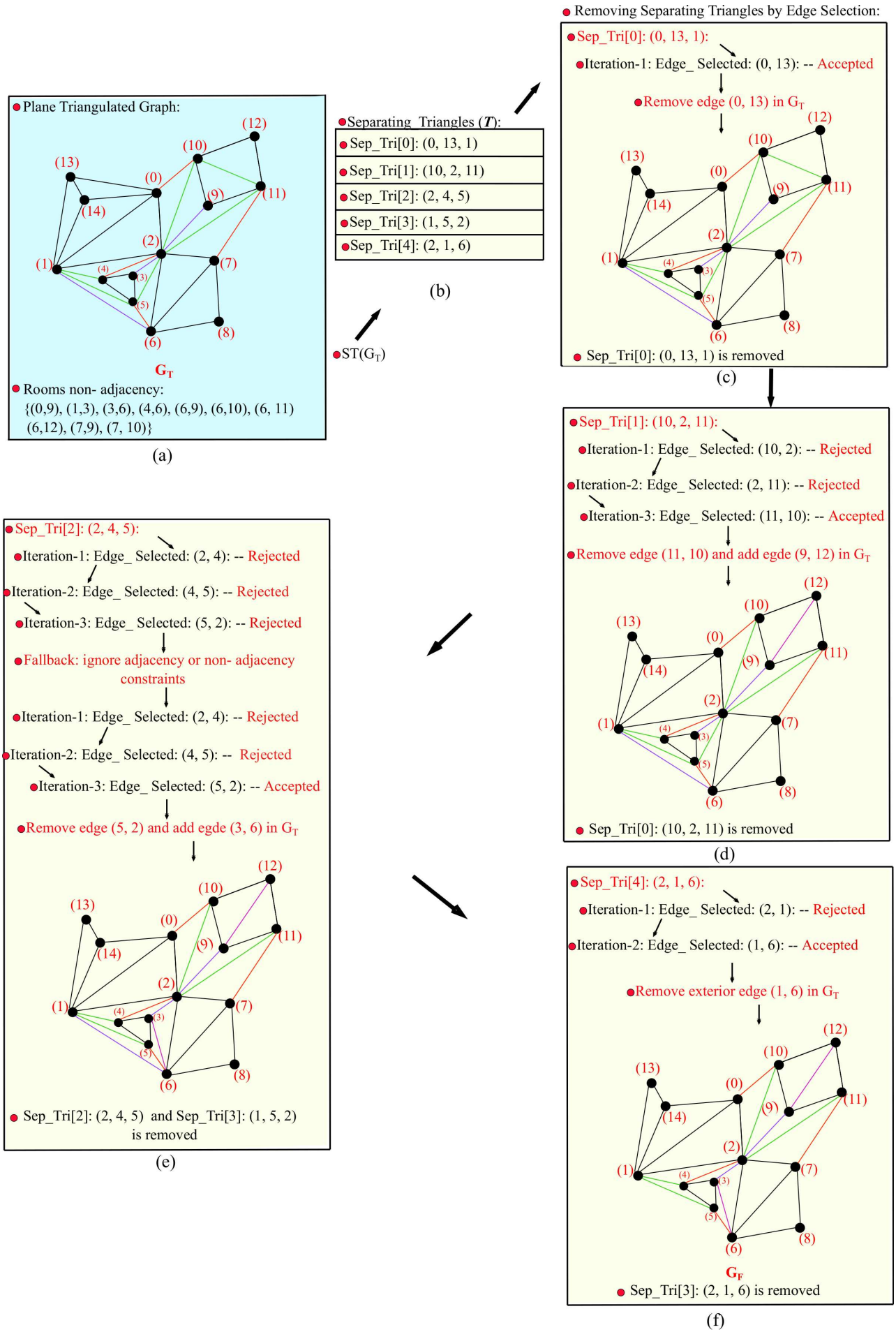

Fig. 10. (a-f) Breaking of separating triangles.

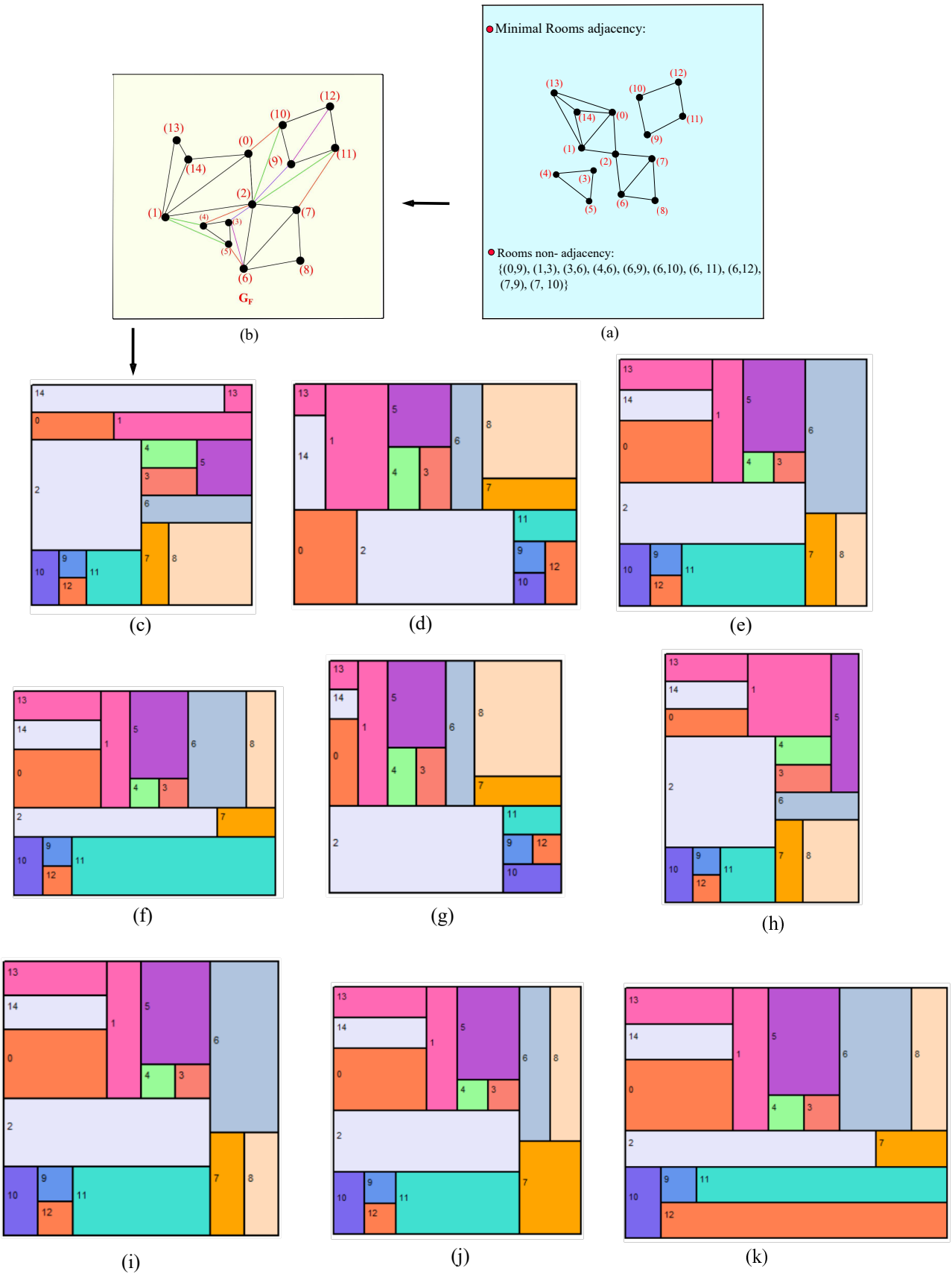

Fig. 11. (a-k) Multiple Rectangular Floor plans generated based on the previously generated graph G_F.

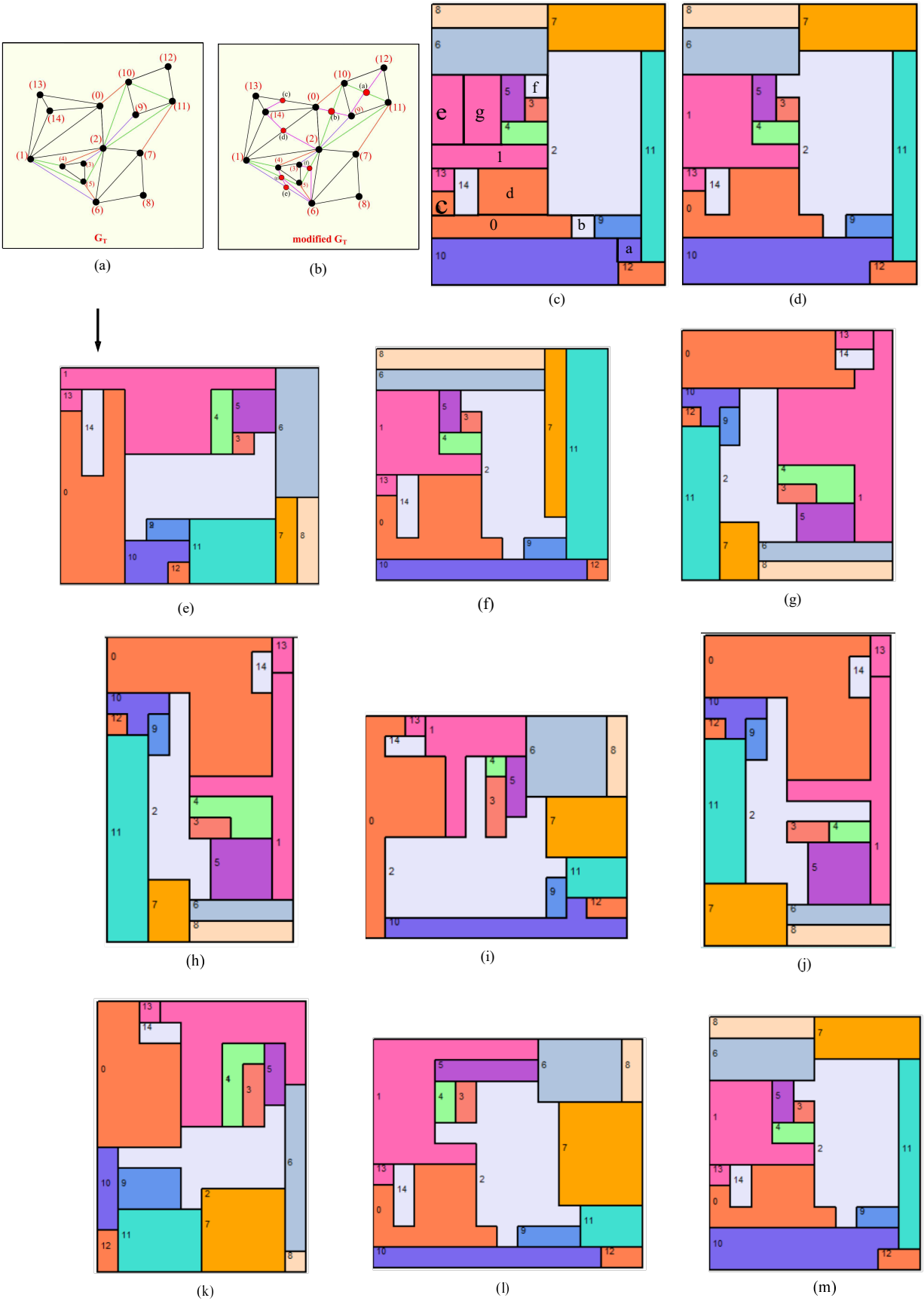

Fig. 12. (a-m) Multiple orthogonal Floor plans generated based on the previously generated graph G_T.

highlight the precision enabled by manual refinement and demonstrate how the graph-generation design framework embedded in the software facilitates the generation of realistic, adaptable floor plans. Together, they demonstrate the effectiveness of the proposed approach in producing residential layouts that closely align with user requirements and real-world design considerations.

8 Conclusion

This work introduces a DPLAN prototype that applies a sequence of graph-based algorithms to generate families of floor plans, either rectangular floor plans (RFPs) or orthogonal floor plans (OFPs), from user-defined constraints. The user provides high-level spatial requirements, including which rooms must be adjacent (for possible door placement) and which room pairs must remain non-adjacent and not share a wall. Based on these inputs, the system constructs a biconnected, plane-triangulated graph that satisfies both adjacency and non-adjacency constraints. This properly triangulated plane graph (PTPG) acts as the combinatorial foundation for producing multiple, distinct, yet constraint-consistent layout alternatives. At its current stage, the prototype generates layouts with a rectangular outer boundary and produces dimensionless (topological) floor plans suitable for early design exploration and structural analysis.

9 Future Enhancement

Several extensions are planned to bring the prototype closer to practical architectural use. On the geometric side, future work will support non-rectangular outer boundaries and assign explicit dimensions to rooms and corridors. This will allow the system to produce scale-accurate floor plans rather than purely combinatorial layouts. Circulation modelling will also be improved by enabling users to define corridor requirements and preferred access patterns, which can be incorporated directly into the graph constraints. From a usability perspective, the interface will be enhanced to simplify constraint editing, comparison of alternative layouts, and export to standard CAD or BIM platforms. Additional developments include automatic door placement, basic three-dimensional visualization of generated layouts, and integration with evaluation modules such as daylight analysis or area-efficiency assessment. Together, these extensions aim to develop the prototype into a more complete decision-support tool for architects and space planners.

10 Supplementary Material

A video illustrating the execution of the proposed Python-based framework, which generates floor plans under different user-defined constraints, is available at the following link:
Demonstration
A brief demonstration of our updated prototype, reflecting ongoing developments, is provided below. This video highlights the revised editing workflow along with recent usability enhancements:
Future Work

11 Declaration of Interest

The authors confirm that no financial or personal relationships exist that could be perceived as influencing the results reported in this paper.

References

1. Krzysztof Koźmiński and Edwin Kinnen. Rectangular duals of planar graphs. *Networks*, 15(2):145–157, 1985.
2. Ingrid Rinsma. Existence theorems for floorplans. *Bulletin of the Australian Mathematical Society*, 37(3):473–475, 1988.
3. Jayaram Bhasker and Sartaj Sahni. A linear time algorithm to check for the existence of a rectangular dual of a planar triangulated graph. *Networks*, 17(3):307–317, 1987.
4. Ingrid Rinsma. Rectangular and orthogonal floorplans with required room areas and tree adjacency. *Environment and Planning B: Planning and Design*, 15(1):111–118, 1988.

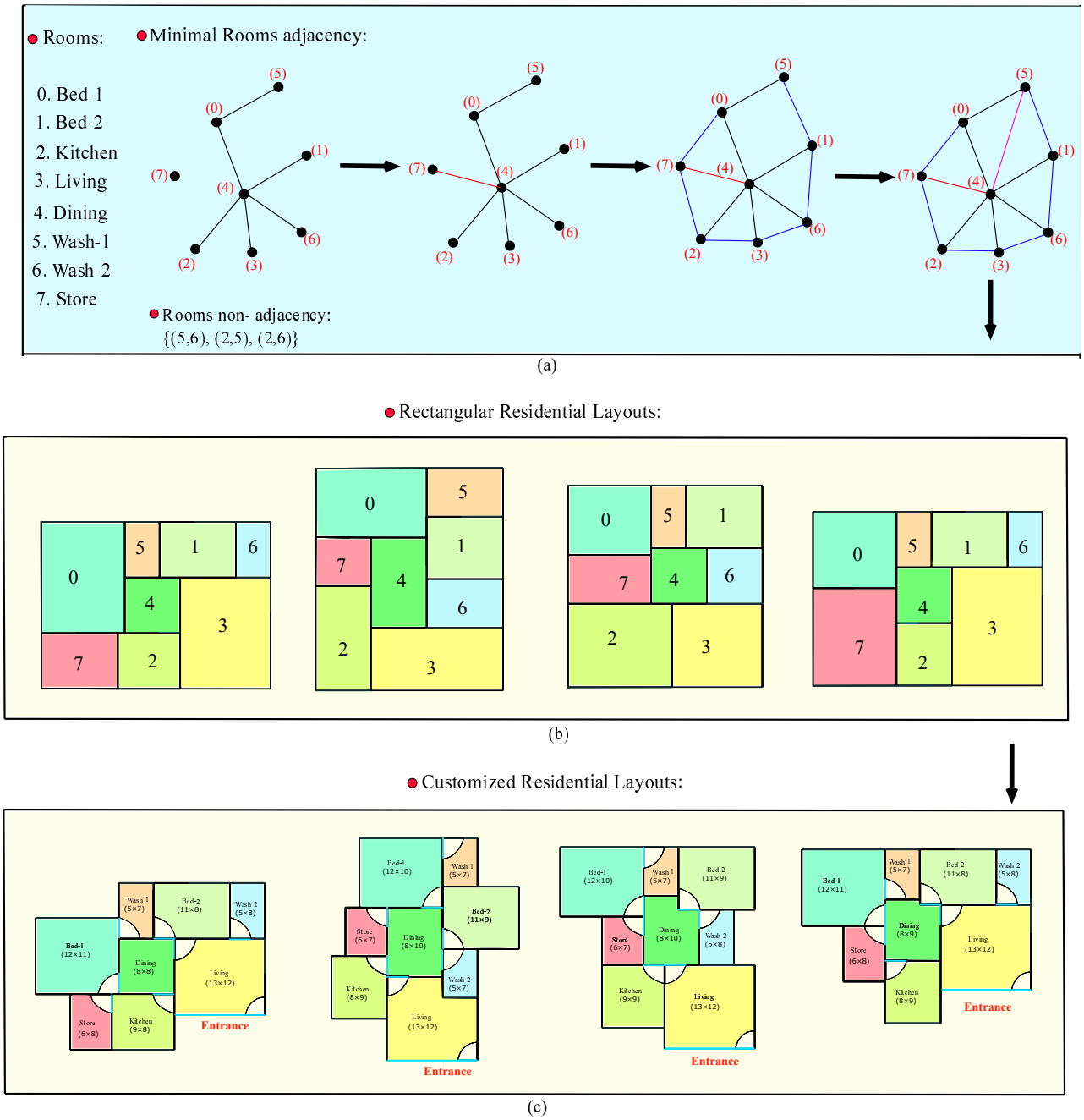

Fig. 13. (a–c) Customized two-bedroom residential floor plans obtained through user-guided refinement of the generated layouts.

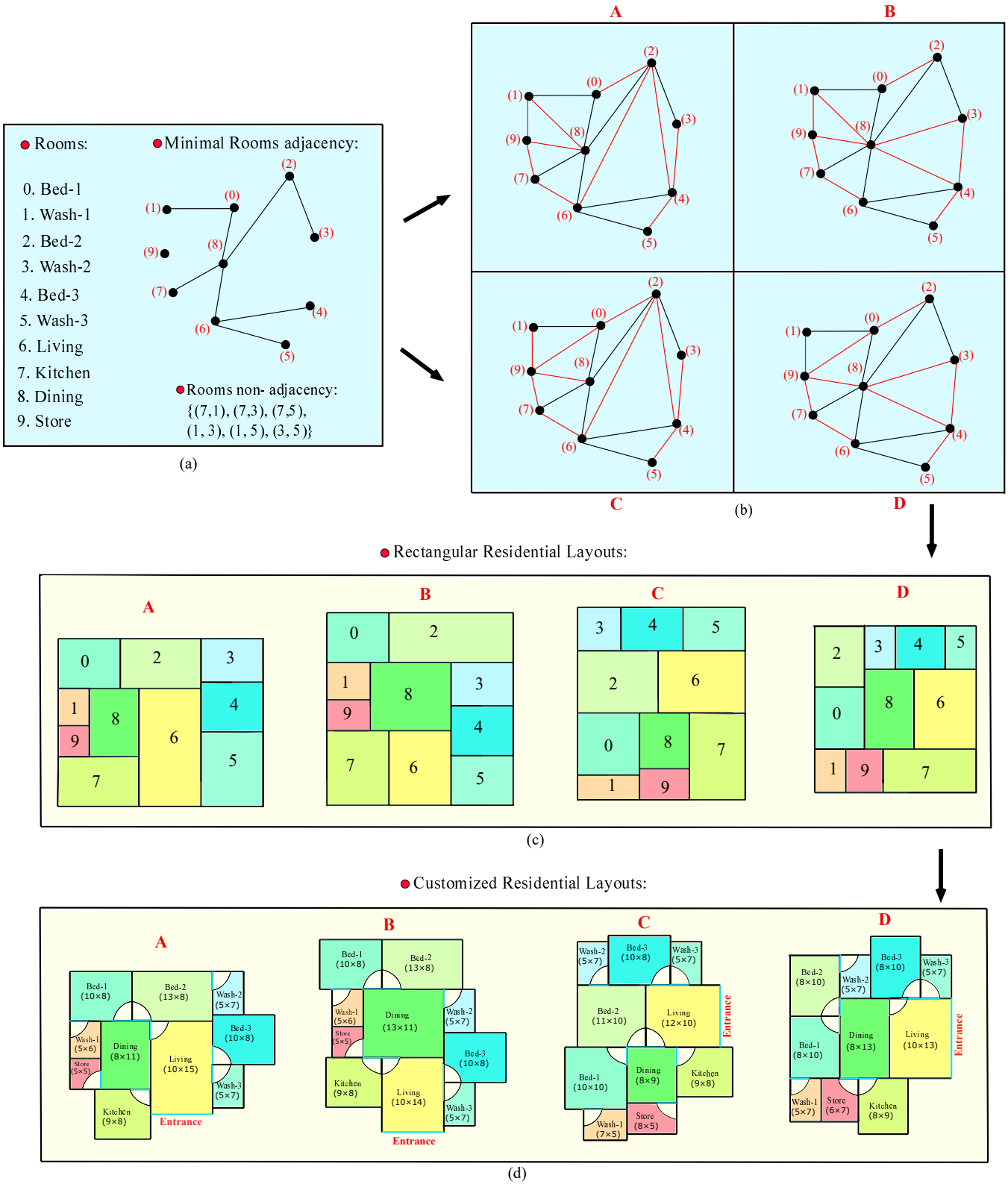

Fig. 14. (a–d) Customized three-bedroom residential floor plans obtained through user-guided refinement of the generated layouts.

5. Yen-Tai Lai and Sany M Leinwand. Algorithms for floorplan design via rectangular dualization. *IEEE transactions on computer-aided design of integrated circuits and systems*, 7(12):1278–1289, 1988.
6. Maciej Kurowski. Simple and efficient floor-planning. *Information processing letters*, 86(3):113–119, 2003.
7. Chien-Chih Liao, Hsueh-I Lu, and Hsu-Chun Yen. Compact floor-planning via orderly spanning trees. *Journal of Algorithms*, 48(2):441–451, 2003.
8. Fernando Marson and Soraia Raupp Musse. Automatic real-time generation of floor plans based on squarified treemaps algorithm. *International Journal of Computer Games Technology*, 2010(1):624817, 2010.
9. Huaming Zhang and Sadish Sadasivam. Improved floor-planning of graphs via adjacency-preserving transformations. *Journal of combinatorial optimization*, 22(4):726–746, 2011.
10. Maysam Mirahmadi and Abdallah Shami. A novel algorithm for real-time procedural generation of building floor plans. *arXiv preprint arXiv:1211.5842*, 2012.
11. Hao Hua. Irregular architectural layout synthesis with graphical inputs. *Automation in construction*, 72:388–396, 2016.
12. Xiao-Yu Wang, Yin Yang, and Kang Zhang. Customization and generation of floor plans based on graph transformations. *Automation in Construction*, 94:405–416, 2018.
13. Nitant Upasani, Krishnendra Shekhawat, and Garv Sachdeva. Automated generation of dimensioned rectangular floorplans. *Automation in Construction*, 113:103149, 2020.
14. Krishnendra Shekhawat, Nitant Upasani, Sumit Bisht, and Rahil N Jain. A tool for computer-generated dimensioned floorplans based on given adjacencies. *Automation in Construction*, 127:103718, 2021.
15. Krishnendra Shekhawat, Rohit Lohani, Chirag Dasannacharya, Sumit Bisht, and Sujay Rastogi. Automated generation of floorplans with non-rectangular rooms. *Graphical Models*, 127:101175, 2023.
16. Sumit Bisht, Krishnendra Shekhawat, Nitant Upasani, Rahil N. Jain, Riddhesh Jayesh Tiwaskar, and Chinmay Hebbar. Transforming an adjacency graph into dimensioned floorplan layouts. *Computer Graphics Forum*, 41(6):5–22, 2022.
17. Shiksha, Rohit Lohani, Krishnendra Shekhawat, Arsh Singh, and Karan Agrawal. Automated generation of housing layouts using graph-rules. *Computers & Graphics*, 134:104506, 2026.
18. Chen Liu, Jiaye Wu, and Yasutaka Furukawa. Floornet: A unified framework for floorplan reconstruction from 3d scans. In *Proceedings of the European conference on computer vision (ECCV)*, pages 201–217, 2018.
19. Wenming Wu, Xiao-Ming Fu, Rui Tang, Yuhan Wang, Yu-Hao Qi, and Ligang Liu. Data-driven interior plan generation for residential buildings. *ACM Transactions on Graphics (TOG)*, 38(6):1–12, 2019.
20. Jiacheng Chen, Chen Liu, Jiaye Wu, and Yasutaka Furukawa. Floor-sp: Inverse cad for floorplans by sequential room-wise shortest path. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.
21. Ruizhen Hu, Zeyu Huang, Yuhan Tang, Oliver Van Kaick, Hao Zhang, and Hui Huang. Graph2plan: learning floorplan generation from layout graphs. *ACM Trans. Graph.*, 39(4), August 2020.
22. Nelson Nauata, Kai-Hung Chang, Chin-Yi Cheng, Greg Mori, and Yasutaka Furukawa. House-gan: Relational generative adversarial networks for graph-constrained house layout generation. In *European Conference on Computer Vision*, pages 162–177. Springer, 2020.
23. Nelson Nauata, Sepidehsadat Hosseini, Kai-Hung Chang, Hang Chu, Chin-Yi Cheng, and Yasutaka Furukawa. House-gan++: Generative adversarial layout refinement network towards intelligent computational agent for professional architects. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13632–13641, 2021.
24. Shidong Wang, Wei Zeng, Xi Chen, Yu Ye, Yu Qiao, and Chi-Wing Fu. Actfloor-gan: activity-guided adversarial networks for human-centric floorplan design. *IEEE Transactions on Visualization and Computer Graphics*, 29(3):1610–1624, 2021.
25. Jiahui Sun, Wenming Wu, Ligang Liu, Wenjie Min, Gaofeng Zhang, and Liping Zheng. Wallplan: synthesizing floorplans by learning to generate wall graphs. *ACM Transactions on Graphics (TOG)*, 41(4):1–14, 2022.
26. Safa Shabani, Mohammad Fekri, Mohammad Ali Sadeghi, and Peter Wonka. Housediffusion: Vector floorplan generation via a diffusion model with discrete and continuous denoising. *Computer Graphics Forum*, 42(7):e14916, 2023.
27. ZHEN HAN, XIAOQIAN LI, YE YUAN, and RUDI STOUFFS. Graph2pix: A generative model for converting room adjacency relationships into layout images. In *Proceedings of the 29th International Conference of the Association for Computer-Aided Architectural Design Research in Asia (CAADRIA) 2024*, pages 139–148, 2024.
28. Chau Ma Thi. Deep learning for automated 3d floor plan generation. *VNU JOURNAL OF SCIENCE: COMPUTER SCIENCE AND COMMUNICATION ENGINEERING: Vietnam National University Journal of Science*, 2024.
29. Hang Zhang, Anton Savov, and Benjamin Dillenburger. Maskplan: Masked generative layout planning from partial input. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8964–8973, 2024.
30. Shibo Hong, Xuhong Zhang, Tianyu Du, Sheng Cheng, Xun Wang, and Jianwei Yin. Cons2plan: Vector floorplan generation from various conditions via a learning framework based on conditional diffusion models. In *Proceedings of the 32nd ACM International Conference on Multimedia*, pages 3248–3256, 2024.
31. Pengyu Zeng, Wen Gao, Jun Yin, Pengjian Xu, and Shuai Lu. Residential floor plans: Multi-conditional automatic generation using diffusion models. *Automation in Construction*, 162:105374, 2024.
32. Mohamed Abouagour and Eleftherios Garyfallidis. Gflan: Generative functional layouts. *arXiv preprint arXiv:2512.16275*, 2025.

33. J Roth, R Hashimshony, and A Wachman. Turning a graph into a rectangular floor plan. *Building and Environment*, 17(3):163–173, 1982.
34. Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.
35. Gang Mei, John C Tipper, and Nengxiong Xu. Ear-clipping based algorithms of generating high-quality polygon triangulation. In *Proceedings of the 2012 International Conference on Information Technology and Software Engineering: Software Engineering & Digital Media Technology*, pages 979–988. Springer, 2012.
36. I Rinsma. Existence theorems for floor-plans. *Bulletin of the Australian Mathematical Society*, 37(3):473–475, 1988.
37. Donald B Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.
38. Goos Kant and Xin He. Regular edge labeling of 4-connected plane graphs and its applications in graph drawing problems. *Theoretical Computer Science*, 172(1-2):175–193, 1997.
39. Ephraim Korach and Zvi Ostfeld. Dfs tree construction: Algorithms and characterizations: Preliminary version. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 87–106. Springer, 1988.
40. Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, 1975.
41. Xin He. On floor-plan of plane graphs. *SIAM Journal on Computing*, 28(6):2150–2167, 1999.
42. Feng Shi, Ranjith K Soman, Ji Han, and Jennifer K Whyte. Addressing adjacency constraints in rectangular floor plans using monte-carlo tree search. *Automation in Construction*, 115:103187, 2020.

12 Appendix

12.1 Analysis of Performance

This section explains the computational performance of our proposed pipeline and compares it with existing graph-based and learning-based methods for floorplan generation. We analyze each step of the algorithm separately and describe the type of structural operations performed at that stage. Using the notation and helper functions defined earlier in Section 6, we derive clear worst-case time complexity bounds for each step.
We also discuss how these computational costs relate to the properties of typical architectural room adjacency graphs, such as their size and structural constraints. Finally, we compare our method with established approaches to show where our contribution fits within existing architectural layout generation techniques.

12.1.1 Overview of the Computational Structure The pipeline described in Section 6 converts a user-defined door-adjacency graph $G(V, E)$ into a properly triangulated plane graph G_F. This transformation is carried out in four structured stages, each enforcing a specific graph-theoretic property required for the final layout/floor plan.

1. Algorithm 1 ensures connectivity while respecting non-adjacency constraints, producing the graph G_C.
2. Algorithm 2 augments the graph to achieve bi-connectivity, producing G_B.
3. Algorithm 3 performs constrained triangulation to obtain G_T.
4. Algorithm 4 removes separating triangles (when required), resulting in the final graph G_F, particularly when each module must correspond to a rectangular region.

All four procedures operate directly on a fixed plane embedding of the graph. At every step, adjacency relations are modified only when necessary and in a clearly defined manner. The pipeline does not explore or enumerate multiple alternative adjacency configurations, unlike learning-based or search-based approaches. Instead, it follows a deterministic sequence of structural modifications, where each step serves a single, well-defined purpose: enforcing connectivity, bi-connectivity, triangulation, or eliminating separating triangles.
Therefore, the computational cost of the pipeline depends only on the size and structural properties of the input graph. It does not depend on exploring multiple combinatorial possibilities, which keeps the behavior predictable and easier to analyze.

12.1.2 Run-time Analysis of the Proposed Algorithms We now analyze the worst-case time complexity of Algorithms 1–4. Let $n = |V|$ denote the number of rooms (vertices), and let $m = |E|$ denote the number of edges in the current adjacency graph. Since the input graph is planar, we have $m = O(n)$ at every stage of the pipeline.
The non-adjacency constraints are provided as part of the input and stored in a data structure that allows constant-time access for any specified pair of rooms. This assumption applies only to retrieving the constraint information

itself. During the execution of Algorithms 1–4, these non-adjacency constraints are checked repeatedly whenever new possible/candidate edges are generated and evaluated. Although each individual lookup takes constant time, the total cost of enforcing non-adjacency constraints depends on the number of candidate edge combinations examined. Therefore, the overall contribution of constraint checking is naturally included in the total time complexity of the corresponding algorithmic steps.

12.1.3 Algorithm 1: Connectivity with Non-Adjacency: Algorithm 1 ensures that every room/node lies in a single connected component while satisfying the non-adjacency constraints. Its complexity arises from three primary operations:

- **Connected components:** The call CONNECTED_COMPONENTS(G) runs in $O(n+m)$ time using the standard Depth First Search (DFS) algorithm discussed in the paper [39], which takes linear time complexity.

- **Extraction of outerface boundaries:** For each disconnected component formed during connectivity augmentation, EXTRACT_OUTERFACE() function traces the boundary of the outer face in the plane embedding to identify its incident vertices. As outlined in Algorithm 1, this traversal follows the cyclic edge order around boundary vertices and marks visited edges and vertices to prevent repetition. Since each vertex and edge is encountered at most once across all components, the total cost of outerface extraction is linear in the size of the graph. Combined with the connected component computation and union-find operations, this step contributes $O(n+m)$ time to the overall complexity, where n and m denote the number of vertices and edges, respectively.

- **Candidate edge generation:** After computing the outer-face vertex sets for all connected components, the function GENERATE_POSSIBLE_EDGES systematically constructs admissible/possible edges that may be used to connect distinct components without violating non-adjacency constraints. Let the input graph consist of k connected components, and let b_i denote the number of vertices incident to the outer face of component i. Since only boundary vertices can be connected without destroying planarity, the algorithm restricts its search to cross-component pairs drawn from these outer-face sets.
  For each unordered pair of components (i, j) with $i < j$, the algorithm examines all vertex pairs (u, v) such that $u \in$ outerfaces[i] and $v \in$ outerfaces[j]. The total number of candidate pairs considered is therefore

$$\sum_{i<j} b_i b_j \;=\; \frac{1}{2}\left(\left(\sum_{i=1}^{k} b_i\right)^2 - \sum_{i=1}^{k} b_i^2\right).$$

  Because every outer-face vertex belongs to exactly one component and the union of all outer faces is a subset of the vertex set V, we have $\sum_i b_i \le n$. Consequently, the total number of vertex pairs examined is bounded above by $O(n^2)$ in the worst case.
  Each candidate pair is cross-checked against the Room Non-Adjacency list. This test determines whether adding the corresponding edge would violate a user-specified separation constraint. The non-adjacency set is stored in a hash-based data structure, so each lookup runs in constant expected time. Importantly, this constant-time check applies only to verifying whether a specific pair (u, v) is forbidden; it does not imply that the overall edge-generation process is constant time, as the algorithm must still enumerate all admissible cross-component pairs.
  Combining the quadratic bound on the number of candidate pairs with the constant-time feasibility check for each pair, the overall time complexity of GENERATE_POSSIBLE_EDGES is $O(n^2)$ in the worst case.

- **Union find merging:** Once the set of admissible candidate edges has been generated, Algorithm 1 iterates through this set to progressively merge disconnected components. Each candidate edge (u, v) connects vertices belonging to two distinct components, as determined by their representative elements in the disjoint-set structure. Before performing a merge, the algorithm queries the union-find data structure to check whether u and v already belong to the same component; if not, it performs a union operation to merge the corresponding components.
  The number of such union–find operations is bounded by the number of candidate edges examined. As established in the candidate edge generation phase, the total number of admissible cross-component vertex pairs is $O(n^2)$ in the worst case. As a result, the total number of find and union operations executed by the augmentation loop throughout Algorithm 1 is bounded by $O(n^2)$.

Each connectivity test and component merge in Algorithm 1 is performed using a union-find data structure augmented with path compression and union-by-rank (or size). With these heuristics, the total cost of performing a sequence of union and find operations on n elements is bounded by $O(p\,\alpha(n))$, where p is the number of operations and $\alpha(\cdot)$ is the inverse Ackermann function [40]. This function grows extremely slowly and remains bounded by a small constant for all input sizes relevant to architectural layout graphs. As a result, although Algorithm 1 may invoke union find operations for up to $O(n^2)$ candidate edges, the cost of each operation is negligible compared to the cost of enumerating the candidates themselves. Consequently, the union–find merging phase does not increase the overall asymptotic complexity beyond $O(n^2)$.
Combining the quadratic bound on the number of candidate edges with the near-constant amortized cost of each union-find operation, the total time spent in the merging phase of Algorithm 1 is bounded by $O(n^2)$. This bound reflects the algorithm's deliberate strategy of exhaustively testing all planarity-preserving, non-adjacent cross-component connections while maintaining efficient component tracking through a disjoint-set data structure.

Thus, the total running time of Algorithm 1 is

$$T_{\text{conn}} = O(n^2),$$

with the quadratic term reflecting the necessity of comparing boundary vertices across disconnected components.

12.1.4 Algorithm 2: Bi-Connectivity with Non-Adjacency: Algorithm 2 upgrades the connected graph G_C to a bi-connected graph G_B by eliminating articulation vertices, that is, vertices whose removal disconnects the current door-connectivity structure. The algorithm operates by locally repairing articulation-induced separations while respecting the Room Non-Adjacency constraints, and continues until no articulation vertex remains.

- **Block decomposition around an articulation vertex:** For a fixed articulation vertex v, the procedure Blocks(G_C, v) constructs the graph obtained by removing v and its incident edges, and then computes the connected components of the resulting graph. Using a standard DFS/BFS traversal [39], this step requires $O(n+m)$ time on the current graph. Since the graph remains planar throughout the pipeline, we have $m = O(n)$, and hence block decomposition is linear in n.

- **Cross-block candidate enumeration under non-adjacency:** Let $\{B_1, \ldots, B_b\}$ denote the blocks incident to v. The routine Valid_Edges enumerates all vertex pairs (u, w) with $u \in B_i$ and $w \in B_j$ for all unordered block pairs (B_i, B_j), and filters them by testing whether $(u, w) \notin$ `non_adj_set`. The number of candidate pairs examined is

$$\sum_{i<j} |B_i|\,|B_j| \;=\; \tfrac{1}{2}\left(\left(\sum_i |B_i|\right)^2 - \sum_i |B_i|^2\right) \;\le\; O(n^2),$$

  since the blocks form a partition of $V \setminus \{v\}$. When the Room Non-Adjacency relation is stored in a representation supporting constant-time membership queries, the total cost of this enumeration step is $O(n^2)$ in the worst case.

- **Merging blocks via union–find:** From the admissible candidate edges, the procedure Connect_Blocks selects a minimal subset that merges all blocks incident to v into a single bi-connected structure. This is implemented using a disjoint-set data structure over block indices. Each candidate edge induces a constant number of Find and Union operations. With path compression and union-by-rank, a sequence of p such operations runs in $O(p\,\alpha(n))$ amortised time. Here, $p = O(n^2)$ in the worst case, so this step is asymptotically dominated by the candidate enumeration cost.

Remark on edge insertions and global progress: Although Algorithm 2 inserts edges during execution, the graph remains planar at all times; therefore, the total number of edges satisfies $m \le 3n - 6$, and all subroutines of cost $O(n+m)$ remain linear in n. Moreover, each accepted edge insertion strictly reduces the articulation structure of the graph by merging blocks incident to the processed articulation vertex. Consequently, the total number of accepted insertions over the entire execution is $O(n)$.

While the above steps are described locally for a single articulation vertex, the algorithm does not incur the worst-case $O(n^2)$ cost independently for each such vertex. Because each insertion reduces the number of articulation-induced blocks and no articulation vertex is processed indefinitely, the total number of cross-block vertex pairs examined across the full execution is bounded by $O(n^2)$.
Therefore, the overall worst-case running time of Algorithm 2 is

$$T_{\text{biconn}} = O(n^2),$$

with the quadratic term arising from explicit cross-block candidate enumeration under non-adjacency constraints. In typical architectural door graphs, the number of articulation vertices and the sizes of block partitions are small, and the observed running time is substantially lower than this worst-case bound.

12.1.5 Algorithm 3: Constrained Triangulation: Algorithm 3 transforms the bi-connected planar graph G_B into a plane triangulation G_T while respecting the Room Non-Adjacency constraints whenever feasible. The procedure operates on the planar embedding of G_B and consists of enumerating all faces and triangulating each non-triangular interior face by inserting admissible diagonals.

- **Face enumeration:** The routine FIND_FACES(G_B) extracts all facial cycles of the planar embedding by traversing directed edges according to their cyclic order around vertices. Each directed edge is marked once it has been assigned to a face, and since each edge is incident to at most two faces, the total cost of enumerating all faces is $O(n + m)$. As the graph remains planar throughout, $m = O(n)$ and face enumeration is linear in n.

- **Amortised ear-clipping triangulation [35]:** Let f be an interior face whose boundary is a simple polygon of length $\ell > 3$. The procedure CLIP_TRIANGULATION repeatedly removes boundary vertices by inserting diagonals until f is decomposed into triangles, performing exactly $\ell - 3$ successful clipping steps.
  In each step, the algorithm scans the current boundary to identify a triple of consecutive vertices (a, b, c) such that the diagonal (a, c) satisfies the predicate IS_VALID_DIAGONAL. This predicate includes (i) a membership query against the Room Non-Adjacency set, (ii) a geometric intersection test against existing boundary edges, and (iii) a check that the diagonal lies in the interior of the face. In the straightforward implementation, the geometric intersection test dominates and requires $O(\ell)$ time.
  Importantly, each diagonal candidate is tested at most a constant number of times before either being accepted or becoming irrelevant due to boundary reduction. As a result, the total cost of all diagonal validity tests over the full triangulation of a face of length ℓ is $O(\ell^2)$.

To obtain a graph-level bound, observe that the sum of boundary lengths over all faces in a planar embedding is $O(m)$, and since $m = O(n)$, the total triangulation cost over all faces is

$$T_{\text{tri}} = \sum_{\text{faces}} O(\ell^2) = O(n^2).$$

In practice, the non-triangular faces arising from architectural door-connectivity graphs are typically small, and admissible diagonals are identified quickly, leading to substantially faster observed runtimes.

12.1.6 Algorithm 4: Separating-Triangle Removal Algorithm 4 eliminates separating triangles from the triangulated graph G_T in order to obtain a properly triangulated plane graph G_F (PTPG), which forms the required input class for rectangular-dual based layout construction [41]. The algorithm processes separating triangles iteratively and applies local graph modifications that strictly reduce their number, while preserving planarity and triangulation at every step.

- **Detection of separating triangles [37]:** The procedure ST(G_T) identifies all separating triangles in the current graph. Since G_T is a plane triangulation, all triangular faces can be enumerated by adjacency-based traversal, and each candidate triangle can be tested for separability by checking whether it encloses interior vertices. This detection step runs in $O(n+m)$ time, and because planarity is maintained throughout the algorithm, $m = O(n)$, yielding linear-time detection.

– **Exterior-edge deletion test:** For a separating triangle $T = (x, y, z)$, Algorithm 4 first attempts to delete an exterior edge. For an edge $(x, y) \in E(T)$, the algorithm computes the common neighbours of x and y. In a plane triangulation, each edge is incident to exactly two triangular faces, so the number of common neighbours is constant. Consequently, testing whether an edge is exterior and deletable requires only local adjacency checks and incurs constant time per edge. Since each separating triangle has exactly three edges, this stage incurs a cost of $O(1)$ per triangle.

– **Constrained edge replacement.** If no exterior edge can be deleted, the algorithm attempts to replace an edge of the separating triangle while preserving triangulation and respecting non-adjacency constraints. Let $T = (x, y, z)$ be a separating triangle. For each boundary edge (x, y), the algorithm selects a vertex r from CommonNeighbors(x, y) and a vertex d from $\text{nbd}(x) \cap \text{nbd}(y) \cap \text{nbd}(z)$. In a plane triangulation, each edge has exactly two face-completing common neighbours, and a triangle admits at most one vertex adjacent to all three of its corners. Hence, the number of candidate (r, d) pairs examined per separating triangle is bounded by a constant.

  For each candidate replacement, the algorithm performs: (i) a membership query against the Room Non-Adjacency set, (ii) a constant-time local edge update that preserves planarity, and (iii) a recomputation of the separating triangles in the modified graph, denoted by $\text{ST}(G'_T)$. The recomputation step dominates and requires $O(n)$ time.

– **Fallback replacement without non-adjacency filtering.** If no admissible replacement satisfying the non-adjacency constraints is found, the algorithm repeats the same constant-size candidate search without enforcing the non-adjacency filter. This fallback guarantees progress by ensuring that at least one separating triangle is eliminated, without altering the asymptotic cost.

Progress and global complexity: Each accepted deletion or replacement is applied only if it strictly reduces the total number of separating triangles. Let s denote the number of separating triangles present during execution. Since each accepted modification reduces this count, Algorithm 4 performs at most s successful iterations. As $s = O(n)$ for planar triangulated graphs, and each iteration incurs a cost of $O(n)$ due to recomputation of $\text{ST}(\cdot)$, the overall worst-case running time of Algorithm 4 is

$$T_{\text{sep}} = O(s\, n) = O(n^2).$$

In practical architectural door-connectivity instances, the number of separating triangles is typically small and decreases rapidly, resulting in observed running times well below this worst-case bound.

12.1.6.1 Total worst-case complexity of Algorithms 1–4: Combining Algorithms 1, 2, 3, and 4, and using the fact that each stage runs in $O(n^2)$ time under the stated assumptions, the end-to-end worst-case time complexity of the proposed pipeline is

$$T_{\text{total}}(n) = O(n^2).$$

12.1.7 Practical Behaviour on Architectural Inputs Although the theoretical analysis yields a quadratic worst-case bound, the graphs arising from real architectural door-connectivity specifications have additional structure. This structure significantly reduces the running time in practice.

– The outer-face boundaries of connected components are usually small. Therefore, the number of cross-component vertex pairs examined during candidate edge generation is much lower than the worst-case $O(n^2)$ bound.

– Articulation vertices are rare in typical room-adjacency graphs. When they occur, they usually connect only a small number of blocks, so the bi-connectivity augmentation step remains limited in scope.

– Faces created during the intermediate stages are generally small, most often quadrilaterals or pentagons. As a result, the constrained triangulation step requires only a few edge insertions.

– Separating triangles are uncommon in practical inputs and are removed quickly. Hence, Algorithm 4 typically completes after only a small number of iterations.

Consequently, the observed running time on all tested architectural instances remained well below the theoretical upper bound. Empirically, the growth in running time followed a near-quadratic trend but with small constants. This allowed the system to provide interactive performance for layouts containing approximately 20 to 30 rooms.

12.2 Comparison with Existing Methods

Table 4 compares our method with existing approaches for floorplan and graph generation. The main difference lies in the starting point. Many existing methods assume that the input adjacency graph is already connected, triangulated, and suitable for floorplan construction. In contrast, our method is designed for early design stages, where the user may provide only a small set of door adjacencies together with a list of non-adjacency constraints. If the input graph is disconnected or incomplete, our system first repairs it. It makes the graph connected and bi-connected while respecting all non-adjacency constraints. The final result is a planar and triangulated graph that can be directly used by standard floorplan algorithms. Learning-based systems mainly focus on generating floor plans that look realistic. They usually do not strictly guarantee that all adjacency and non-adjacency constraints are satisfied. Graph-theoretic methods provide stronger guarantees, but they require the input graph to already satisfy strict structural conditions. Our method combines the advantages of both directions. Starting from minimal and possibly incomplete input, we construct a graph that is structurally valid, easy to understand, and ready for rectangular or orthogonal floorplan generation.

12.2.1 Rule-based Graph Enumeration [17] Rule-based systems generate floor plans by applying rewrite rules and testing many possible room arrangements. As the number of rooms increases, the number of possible configurations grows very quickly. In the worst case, this growth is exponential. Our method does not explore multiple alternative graphs. It works on a single graph and modifies it step by step in a fixed manner. Therefore, its running time remains polynomial rather than exponential.

12.2.2 Classical Graph-Theoretic Floorplanning [14,16,8,10,12,11] Classical rectangular-dual algorithms assume that the input graph is already bi-connected and triangulated. Under these assumptions, they run efficiently. However, they do not address how to construct such a graph from sparse design constraints. Our method fills this gap. It transforms a partially defined adjacency graph into a properly triangulated plane graph in polynomial time, while preserving all required adjacencies and non-adjacency constraints.

12.2.3 Learning-based Floorplan Generation [21,22,23,24,25,27] Recent learning-based models can generate realistic floor plans. However, adjacency relations are often treated as soft constraints, which means violations can occur. Additional correction steps are usually required. Our approach can serve as a reliable backbone for such systems. It produces a planar graph that satisfies all structural conditions, which can then be used as input to geometric or learning-based floorplan generators.

12.2.4 Summary The proposed pipeline constructs a valid floorplan graph from minimal door-adjacency information without assuming that the input is already well-formed.

- **Polynomial-time performance:** the worst-case running time is quadratic under planarity assumptions.
- **Deterministic process:** each modification has a clear structural purpose.
- **Constraint preservation:** all adjacency and non-adjacency constraints are respected throughout the process.
- **Ready for floorplan generation:** the final graph satisfies the conditions required by rectangular-dual algorithms.

This makes the proposed approach suitable for applications where structural correctness and strict constraint satisfaction are important.

12.3 Scope and Extensions

The proposed prototype is designed as a constraint-first framework for constructing planar and triangulated graphs from sparse door-adjacency specifications. The current version focuses on correctness, clarity, and ease of use during early design stages. At the same time, it allows several natural extensions.

- **Boundary model:** The present implementation assumes a rectangular outer boundary. This matches common architectural practice and works well with rectangular dual floorplan methods. The same framework can be extended to support irregular boundaries, courtyards, or more complex site shapes by using more general planar embedding techniques.
- **Strict handling of non-adjacency constraints:** Non-adjacency requirements are enforced as hard constraints throughout the pipeline. If the given constraints are inconsistent, the system reports that no feasible graph exists. In future work, this can be extended to include controlled constraint relaxation or analysis tools that help users identify minimal changes needed to restore feasibility.

- **Scalability for early design:** By maintaining planarity and performing explicit structural checks, the framework achieves polynomial-time performance suitable for small and medium-sized floor plans, which are typical in conceptual design. Extending these guarantees to very large programs motivates future work on incremental updates and improved pruning strategies.

- **Separation of topology and geometry:** The pipeline focuses only on constructing a valid floorplan graph. Geometric aspects such as room areas, proportions, circulation, and regulatory constraints are handled separately by downstream optimization or geometric solvers. This separation allows the method to integrate with different geometric floorplan generation tools.

Overall, the framework provides a reliable and extendable foundation for floorplan construction. Future work will aim to support more general boundary shapes, improve scalability, and incorporate higher-level optimization objectives while preserving structural correctness.

12.4 Correctness

In this section, we will prove that our Proposed Algorithms 1–4 are guaranteed to construct a graph belonging to the targeted class for every admissible input. The specified non-adjacency constraints for the specified room are preserved whenever they are explicitly enforced by the algorithmic steps. In addition, each proposed algorithm incorporates a fallback mechanism that is invoked only when all otherwise admissible edge choices are excluded due to non-adjacency constraints, thereby ensuring progress while deviating from the constraints only when strictly necessary.

Throughout, let $G = (V, E)$ denote the input door-connectivity graph together with a fixed *non-adjacency set* $\mathcal{N} \subseteq \binom{V}{2}$ of forbidden pairs. Algorithm 1 constructs a connected augmentation G_C, Algorithm 2 constructs a biconnected augmentation G_B, Algorithm 3 produces a plane triangulation G_T while rejecting diagonals in $\mathcal{N}$ and any diagonal that violates planarity, and Algorithm 4 removes separating triangles to obtain a separating-triangle free graph G_F.

12.4.1 Correctness of Algorithm 1 (Connectivity with Non-adjacency)

Theorem 1. *Let $G = (V, E)$ be a plane graph whose connected components are $C_1, \ldots, C_k$ with $k \geq 1$, and let $\mathcal{N}$ be a set of forbidden pairs. If Algorithm 1 returns $G_C = (V, E \cup A)$, then:*

1. *G_C is connected.*
2. *The number of added edges satisfies $|A| \leq k - 1$, and if Algorithm 1 succeeds in merging all components, then $|A| = k - 1$ (hence the augmentation is minimum in cardinality among all augmentations that connect these k components).*

Proof. **Case 1**: Suppose there exists a set of edges whose insertion into G produces the graph G_C, and none of these added edges belong to the non-adjacency set $\mathcal{N}$.
Now, If $k = 1$, Algorithm 1 returns G immediately, so the claims hold trivially with $A = \emptyset$.

Table 4. Qualitative comparison of representative methods. Unlike most approaches that assume a well-formed adjacency graph, our method builds a valid floorplan graph from sparse door-adjacency and non-adjacency constraints, including automatic connectivity repair.

Method	Year	Typical input assumption	How constraints are handled	Graph feasibility and interaction
Marson & Musse [8]	2010	High-level room program and area data; not based on an explicit door-adjacency graph	Heuristic constraint handling; non-adjacency is not explicitly modeled	Real-time floorplan generation; no explicit certified graph stage
Mirahmadi et al. [10]	2012	Procedural rules; adjacency not represented as an explicit planar graph	Rule-based handling; limited support for explicit non-adjacency constraints	Fast generation; no certified planar or bi-connected graph construction
Wang et al. [12]	2018	Starts from an already usable adjacency structure	Graph transformations mainly refine adjacency; non-adjacency is not central	Efficient rectangular floorplan generation under assumed-valid input
Chen et al. (Floor-SP) [20]	2019	As-built data (RGBD/scan); not based on user-defined door graph	Constraints arise from optimization terms; no hard non-adjacency model	Strong for reconstruction; not focused on sparse input repair
Wu et al. [19]	2019	Boundary-driven and dataset-trained generation	Constraints encouraged through learning objectives	Produces realistic floorplans; structural feasibility not explicitly certified
House-GAN [22] / House-GAN++ [23]	2020–21	Requires a coherent adjacency graph as input	Compatibility learned from data; no hard planarity or bi-connectivity guarantees	High realism; limited transparency in structural enforcement
Graph2Plan [21]	2020	Uses valid floorplan graphs retrieved from a database	Constraints guided by retrieval and learning; limited explicit non-adjacency enforcement	Effective graph-conditioned synthesis; assumes usable input graphs
Fen Shi et al. [42]	2020	Dense adjacency specifications for large facilities	Search- or learning-based constraint satisfaction	Scales to dense inputs; not focused on minimal structural repair
GPLAN [14]	2021	Requires a planar and connected input graph	Hard constraints for modeled requirements; non-adjacency not primary	Efficient floorplan generation given a suitable graph
G2PLAN [16]	2022	Requires a prescribed adjacency graph	Graph-theoretic and optimization-based processing; assumes structurally valid input	Generates dimensioned floorplans; does not repair sparse or disconnected inputs
DPLAN: Proposed prototype	–	*Sparse door-adjacency edges with explicit non-adjacency constraints; input may be disconnected*	*Strict non-adjacency enforcement at every step*	*Constructs a valid graph through connectivity repair, bi-connectivity augmentation, planar triangulation, and optional separating-triangle removal, with interactive feasibility feedback*

Assume $k > 1$. Algorithm 1 first computes, for each component C_i, the set $\mathsf{outerfaces}[i]$ consisting of vertices on the outer-face boundary of C_i (Steps 8–9 and function ExtractOuterface). It then forms the candidate set

$$\mathsf{possible\_edges} = \{(u, v) \mid u \in \mathsf{outerfaces}[i],\ v \in \mathsf{outerfaces}[j],\ i \neq j,\ \text{and } (u, v) \notin \mathcal{N}\},$$

exactly as implemented in GeneratePossibleEdges (Steps 21–29). Hence every candidate edge explicitly satisfies $(u, v) \notin \mathcal{N}$.
Next, Algorithm 1 initializes a union-find structure whose elements are the component indices $\{1, \ldots, k\}$ (Step 11). During the scan of $\mathsf{possible\_edges}$, an edge (u, v) is appended to A *only if* its endpoints lie in different current union–find sets, i.e., $\mathsf{FindComp}(c(u)) \neq \mathsf{FindComp}(c(v))$ (Steps 12–16). Therefore:

- **(Forbidden edges never added)** Since (u, v) is drawn from $\mathsf{possible\_edges}$, we have $(u, v) \notin \mathcal{N}$ for every added edge.

- **(Each accepted edge strictly reduces the number of component-sets)** At the moment an edge is accepted, it merges two previously distinct union–find sets (Step 15). Thus, the number of union–find sets decreases by exactly one per accepted edge.

Let s_t denote the number of union–find sets after t accepted edges. Initially $s_0 = k$. Each accepted edge decreases s_t by 1, so $s_t = k - t$. In particular, after $k - 1$ accepted edges, we have $s_{k-1} = 1$, meaning all original components are in one set. Hence, whenever the scan finds enough cross-component candidates to merge all components, the algorithm produces a connected graph using exactly $k - 1$ new edges; this proves claim (2).

Case 2: Suppose no set of edges exists whose insertion into G yields the graph G_C without violating the non-adjacency constraints, that is, every admissible edge belongs to the non-adjacency set $\mathcal{N}$. In this situation, the algorithm activates a fallback mechanism (Steps 38–39), in which the non-adjacency set is temporarily cleared, and the main procedure is invoked again to ensure the construction of G_C, and now the candidate edge will be:

$$\mathsf{possible\_edges} = \{(u, v) \mid u \in \mathsf{outerfaces}[i],\ v \in \mathsf{outerfaces}[j],\ i \neq j\}$$

and proof will be the same as discussed in Case-1.

Now, we will conclude by establishing claim (1), namely that the output graph is connected. Let $C_1, \ldots, C_k$ denote the connected components of the input graph G, and define the *component graph* H as follows: each vertex of H corresponds to one component C_i, and an edge is added between C_i and C_j whenever Algorithm 1 introduces an augmentation edge (u, v) with $u \in C_i$ and $v \in C_j$.

By construction, each augmentation edge connects two components that were previously distinct according to the union-find structure maintained by the algorithm. Consequently, no augmentation edge can create a cycle in H, and the resulting component graph is a forest. At termination, the algorithm inserts exactly $k - 1$ augmentation edges, so H contains k vertices and $k - 1$ edges. A forest with these parameters must be a tree, and therefore H is connected. This implies that for any pair of components C_i and C_j, there exists a path between them in H. Replacing each edge along this path by its corresponding augmentation edge in the constructed graph G_C yields a walk connecting C_i and C_j in G_C. Since each component C_i is internally connected in G, it follows that G_C is connected.
We now argue minimality. Any connected supergraph (any graph formed by augmenting G with additional edges such that the resulting graph is connected) of G must merge the original k components into one, and the insertion of a single edge can reduce the number of components by at most one. Therefore, at least $k - 1$ new edges are required to achieve connectivity. Since Algorithm 1 terminates after adding exactly $k - 1$ edges, the resulting augmentation is minimal with respect to the number of added edges.

□

12.4.2 Correctness of Algorithm 2 (Biconnectivity with Non-adjacency) Algorithm 2 processes each articulation point v of the connected graph G_C and connects the blocks that arise when v is removed.

Lemma 1 (Correctness of Blocks). *For an articulation point v in G_C, the procedure* Blocks(G_C, v) *returns exactly the subset of boundary vertices (its outerface boundary under the inherited embedding) of the connected components of $G_C - \{v\}$.*

Proof. BLOCKS forms $G' = (V \setminus \{v\}, E \setminus \{(v,u) : u \in \text{nbd}(v)\})$ by deleting v and all incident edges (Steps 11–15), and then calls CONNECTEDCOMPONENTS(G') (Step 15). By definition, these are precisely the connected components of $G_C - \{v\}$. The returned collection is labeled as $\mathsf{blocks} = \{B_1, \ldots, B_\ell\}$ (Step 18). □

Lemma 2 (Validity of candidate edges). *For a fixed articulation point v with blocks $\{B_1, \ldots, B_\ell\}$, the procedure* VALIDEDGES($\mathsf{blocks}, \mathcal{N}$) *returns a set of edges each of which joins vertices from two distinct blocks and does not belong to $\mathcal{N}$.*

Proof. The code enumerates unordered pairs of distinct blocks (B_i, B_j) and then all pairs (u, w) with $u \in B_i$ and $w \in B_j$ (Steps 21–26). It inserts (u, w) into $\mathsf{valid\_edges}$ only if $(u, w) \notin \mathcal{N}$ (Steps 26–27). Thus, every returned edge is cross-block and non-forbidden. □

Lemma 3 (Connecting blocks removes articulation at v). *Let v be an articulation point in a connected graph G. Let $B_1, \ldots, B_\ell$ be the connected components of $G - \{v\}$. If we add edges so that in the augmented graph G' the subgraph induced by $V \setminus \{v\}$ is connected (equivalently, the* block-incidence graph *on $\{B_1, \ldots, B_\ell\}$ becomes connected), then v is not an articulation point of G'.*

Proof. In G', remove v. By assumption, the remaining graph $G' - \{v\}$ is connected. Hence, removal of v does not disconnect G', so v is not an articulation point in G'. □

Theorem 2. *Assume Algorithm 2 is run on a connected graph G_C with articulation-point set $A(G_C)$ and forbidden set $\mathcal{N}$. Let G_B be the returned graph. Then:*

1. *G_B is connected.*
2. *Every edge added during the constrained phase satisfies the non-adjacency constraints, i.e., is not in $\mathcal{N}$.*
3. *If for every articulation point v there exists a set of allowed edges (not in $\mathcal{N}$) that connects the blocks of $G_C - \{v\}$, then the output G_B is biconnected.*

Proof. **(1) Connectivity.** Algorithm 2 only adds edges to G_C (Steps 7–9) and never deletes vertices; adding edges cannot destroy connectivity, hence G_B remains connected.

(2) Non-adjacency preservation (constrained phase). For each articulation point v, candidate edges are drawn from VALIDEDGES, which by Lemma 2 excludes $\mathcal{N}$. CONNECTBLOCKS selects a subset of these candidates (Steps 47–55 and Step 53), so every selected edge from this phase satisfies $(u, w) \notin \mathcal{N}$.

(3) Biconnectivity under feasibility. Fix an articulation point $v \in A(G_C)$. By Lemma 1, blocks are the components of $G_C - \{v\}$. CONNECTBLOCKS is designed to connect these blocks by selecting cross-block edges that merge block-sets under a union–find structure (Steps 36–56); whenever it succeeds in making the block-sets connected, Lemma 3 implies v is no longer an articulation point in the augmented graph. Algorithm 2 repeats this for every $v \in A(G_C)$ (Steps 3–7), so after processing all articulation points, the resulting graph has no articulation points. A connected graph with no articulation points is biconnected. □

12.4.2.1 Remark (about the fallback). The pseudocode contains an explicit fallback that may ignore non-adjacency constraints to connect the remaining blocks (Steps 57–62). Therefore, the strongest unconditional guarantee is: *Algorithm 2 returns a biconnected graph whenever it can complete the block-connection step for each articulation point; it preserves non-adjacency constraints for all edges selected from* VALIDEDGES*, and only the fallback may violate $\mathcal{N}$.*

12.4.3 Correctness of Algorithm 3 (Triangulation with Non-adjacency) Algorithm 3 triangulates each non-triangular face by adding diagonals that pass ISVALIDDIAGONAL checks, which explicitly reject forbidden diagonals and any diagonal that would cross an existing boundary edge or lie outside the face.

Lemma 4 (Safety of accepted diagonals). *Every diagonal (a, c) accepted by* ISVALIDDIAGONAL *is (i) not in $\mathcal{N}$, (ii) lies strictly inside the current face polygon, and (iii) does not properly intersect any existing edge of the current embedding. Hence, adding (a, c) preserves planarity of the embedding and respects the non-adjacency constraints.*

Proof. The validation function first rejects (a, c) if it is present in the non-adjacency set (explicitly stated in the description of IsValidDiagonal). It then checks for proper intersections against the face's boundary edges (excluding incident edges) and rejects any such intersection. Finally, it verifies that (a, c) is a polygon diagonal that lies strictly inside the face region. Therefore, any accepted diagonal simultaneously satisfies (i)–(iii), and adding it keeps the embedding planar and does not introduce a forbidden adjacency. □

Lemma 5 (Progress on a single face). *Let f be a bounded face whose boundary is a simple cycle of length $t \geq 4$. Each time Algorithm 3 adds an accepted diagonal inside f, the face is split into two bounded faces whose boundary lengths sum to $t+2$. In particular, the number of non-triangular faces decreases after finitely many accepted diagonals, and after $t-3$ accepted diagonals, the region originally corresponding to f is fully triangulated.*

Proof. In a simple polygonal face, a non-crossing diagonal between two non-adjacent boundary vertices partitions the polygon into two smaller polygons. This is exactly the geometric meaning of adding a diagonal that lies strictly inside the face and does not intersect the boundary (Lemma 4). Every such split reduces the maximum face size, and a t-gon requires exactly $t-3$ diagonals to triangulate. □

Theorem 3. *Assume Algorithm 3 is applied to a planar embedding of a biconnected graph G_B and uses* IsValidDiagonal *for acceptance. Let G_T denote the output. Then:*

1. *G_T is planar and contains G_B as a spanning subgraph.*
2. *Every diagonal added by Algorithm 3 respects the non-adjacency constraints (i.e., is not in $\mathcal{N}$).*
3. *Every bounded face of G_T is a triangle; hence G_T is a plane triangulated graph.*

Proof. Algorithm 3 only adds edges, so G_B remains a spanning subgraph of the result. By Lemma 4, every accepted diagonal is non-crossing and lies inside the target face, so planarity is preserved throughout; this proves (1). The same lemma gives (2).

For (3), Algorithm 3 iterates over all non-triangular faces and repeatedly attempts diagonals until the face is decomposed; by Lemma 5, each face of length t becomes triangulated after at most $t-3$ accepted diagonals. Applying this to every bounded face yields that all bounded faces are triangles in the final graph. □

12.4.3.1 Remark (about the fallback). The pseudocode contains an explicit fallback that may ignore non-adjacency constraints to connect the remaining blocks (Steps 41–44). Therefore, the strongest unconditional guarantee is: *Algorithm 3 returns a triangulated graph whenever it can complete the ear clipping step for each non-triangulation face; it preserves non-adjacency constraints for all edges selected from* ValidEdges*, and only the fallback may violate $\mathcal{N}$.*

12.4.4 Correctness of Algorithm 4 (Separating-triangle removal) Algorithm 4 takes the triangulated graph G_T and repeatedly modifies it to eliminate separating triangles, returning a graph G_F claimed to be free of separating triangles. The procedure explicitly maintains a counter $m = |T|$, where $T = \mathrm{ST}(G)$ denotes the current set of separating triangles.

Lemma 6 (Monotone progress measure). *Within* RemoveSTEdgeRemoval*, a graph modification is accepted only if the new separating-triangle count m' satisfies $m' \leq m-1$ (Step 32). Therefore, each accepted modification strictly decreases m.*

Proof. In the edge-replacement stage, the algorithm forms a candidate $G'_T = G_T \cup \{(r, d)\} \setminus \{(x, y)\}$ and recomputes $T' = \mathrm{ST}(G'_T)$ and $m' = |T'|$ (Steps 30–31). It accepts the replacement only if $m' \leq m-1$ (Step 32) and then sets $G_T \leftarrow G'_T$ and $m \leftarrow m'$ (Step 33). In the deletion stage, when an eligible edge is deleted, the code updates $m \leftarrow m-1$ after recomputing separating triangles (Steps 16–18). Hence, any accepted action strictly decreases m. □

Lemma 7 (Termination). *Algorithm 4 terminates after a finite number of accepted modifications.*

Proof. At every acceptance, m decreases by at least 1 (Lemma 6). Since $m \geq 0$ always, there can be at most m_0 accepted modifications where $m_0 = |\mathrm{ST}(G_T)|$ is the initial number of separating triangles. All loops are over finite sets (triangles, edges, neighbors), and no step increases m while being accepted. Therefore, the algorithm cannot accept modifications indefinitely and must terminate. □

Theorem 4. *Let G_T be the triangulated graph produced by Algorithm 3 and let $\mathcal{N}$ be the non-adjacency set. If Algorithm 4 returns G_F, then:*

1. *G_F contains no separating triangles, i.e., $\mathrm{ST}(G_F) = \emptyset$.*
2. *During the constrained replacement stage, every added edge (r, d) satisfies $(r, d) \notin \mathcal{N}$.*
3. *If the constrained stage cannot reduce m further, the fallback stage (which drops the non-adjacency filter) guarantees that the algorithm can still reduce m and thus reach $\mathrm{ST}(G) = \emptyset$.*

Proof. Algorithm 4 begins by computing $T = \mathrm{ST}(G_T)$ and $m = |T|$ (Steps 1–4), and then calls RemoveSTEdgeRemoval to iteratively modify G_T until separating triangles disappear, with a second call permitted if $m > 0$ remains (Steps 5–7).

(1) Elimination of separating triangles. By Lemma 6, every accepted modification strictly reduces the number m of separating triangles. Lemma 7 guarantees that only finitely many such reductions can occur. The procedure terminates only after repeated calls to RemoveSTEdgeRemoval fail to produce further reductions, at which point the outer routine returns the graph G_F (Step 7). Since the algorithm is explicitly designed to accept only modifications that decrease $|T|$, and is re-invoked with an empty working edge set $E' = \varnothing$ whenever necessary (Step 6), the process converges to the unique fixed point $m = 0$. Consequently, the output graph satisfies $\mathrm{ST}(G_F) = \emptyset$, and contains no separating triangles.

(2) Non-adjacency preservation in the constrained stage. In the replacement stage, an edge (r, d) is considered only if $(r, d) \notin \mathcal{N}$ (Step 29). Hence, every accepted replacement edge in this stage respects the non-adjacency constraints.

(3) Guaranteed progress via fallback. If no deletion or constrained replacement is possible for the current triangle, the pseudocode explicitly enters a fallback stage labeled "ignoring non-adjacency constraints and retrying replacements." This strictly enlarges the candidate search space (it includes all candidates previously considered, plus those previously filtered out by $\mathcal{N}$), while still enforcing the acceptance condition $m' \leq m-1$ before committing to an update. Therefore, whenever there exists *any* edge replacement that decreases m, the fallback can realize such a decrease, ensuring eventual reachability of $m = 0$.
This completes the correctness argument. □

12.4.4.1 Remark (Role of Algorithm 4 under RFP and OFP modes). The requirement handled by Algorithm 4 depends on the selected user mode. When the user demands that all rooms be rectangular (RFP mode), separating triangles must be eliminated without introducing additional vertices. In this case, Algorithm 4 transforms the given triangulated graph into a separating-triangle-free graph, which is necessary for constructing a rectangular dual.
On the other hand, if non-rectangular modules are permitted (OFP mode), separating triangles can be resolved by inserting new vertices. In this setting, the correctness of Algorithms 1–3 is already sufficient to ensure a valid plane triangulated backbone, and no further structural restriction is required at this stage.