Contents lists available at ScienceDirect

Automation in Construction

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

Residential complex design as a Constraint Satisfaction Problem

Shermin Sherkat [a,b,c,*], Ali Andaji Garmaroodi [b], Andreas Wortmann [c], Thomas Wortmann [a]

[a] Institute for Computational Design and Construction (ICD), Chair for Computing in Architecture (CA), University of Stuttgart, Keplerstrasse 11, Stuttgart, 70174, Germany
[b] Faculty of Architecture and Urban Design, University of Tehran, Tehran, 1417466191, Iran
[c] Institute for Control Engineering of Machine Tools and Manufacturing Units (ISW), University of Stuttgart, Seidenstrasse 36, Stuttgart, 70174, Germany

ARTICLE INFO

Dataset link: https://github.com/shshermin/CSP-solver-for-residential-building-design

Keywords:
CSP
Search strategy
Residential complex
Mass housing
Constraint Satisfaction Problem
Object-oriented data modeling

ABSTRACT

The design of mass housing projects, with their complex array of apartment types and constraints, can be challenging for architects. Automated-organizing programs can assist in exploring various design alternatives, but the computational cost of checking all possible building organizations grows exponentially. This paper describes a method that utilizes a variant of the Constraint Satisfaction Problem (CSP) to bound and direct the growth of search trees. The method allows designers to explore design alternatives using geometrical objects and incorporates constraints related to daylight and privacy evaluation. Two search tree strategies, breadth-first search (BFS) and depth-first search (DFS), are implemented using a custom solver, with DFS proving suitable for larger search trees and BFS being appropriate for searching through the entire tree. By clearly defining the problem and adjusting the constraints, designers can efficiently explore the design space and obtain valid building alternatives in a reasonable time.

1. Introduction

Housing is a complex and multifaceted subject with no single definition or definitive design solution. Therefore, it is necessary to develop tools and methods that reduce the design complexity. Automated organizing becomes challenging in residential complexes because, as the items (apartments) increase, the number of possible organizations increases exponentially [1]. This exponential growth in computational space and time poses a difficulty for computational design tools, as they need to allow designers to explore various results with different sets of variables.

Design tools should increase productivity and creativity by allowing designers to explore a wider variety of solutions than they could reach without computer systems [2]. This paper aims to provide a custom solver for residential building design that limits the search space and lets designers control these limits. Thus, designers can guide the search to achieve solutions with reduced computational requirements.

Constraint Satisfaction Problem (CSP) is a problem-solving paradigm. To formulate a CSP, we need (1) a set of variables and their corresponding domains and (2) a set of constraints on the values that the variables may take. The challenge of CSP is then to find valid values for each variable within their domain and satisfy the constraints [3].

Efficient generic CSP solvers exist [4,5], but these solvers only accept explicit mathematical expressions for defining the problem. In architecture software (e.g., Rhino 3D), the problems usually include complex geometric objects that are hard to define directly with mathematical expressions. Designers who use this software primarily deal with geometric objects and are rarely concerned with the underlying mathematical expressions. Thus, the paper presents the following contributions:

- Introducing a novel custom constraint solver that generates residential buildings, accepts parameters and constraints as geometrical objects, and utilizes the Grasshopper GUI. Notably, the solver offers designers more freedom in shape diversity by removing limitations on the number of inputs and design dimensions.
- Implementing complex geometrical constraints, including daylight availability and privacy evaluation, within the constraint solver to prove the concept.
- Testing the constraint solver with three different design examples to assess its effectiveness and performance.

In the remainder, Section 2 discusses the related work and where the contribution of this paper lies. Section 3 explains the methodology and how the solver was developed. Finally, Sections 4 and 5 discuss the program's results.

* Corresponding author at: Institute for Computational Design and Construction (ICD), Chair for Computing in Architecture (CA), University of Stuttgart, Keplerstrasse 11, Stuttgart, 70174, Germany.
E-mail address: shermin.sherkat@icd.uni-stuttgart.de (S. Sherkat).

https://doi.org/10.1016/j.autcon.2023.104995
Received 17 June 2022; Received in revised form 22 May 2023; Accepted 13 June 2023

2. Background

2.1. Constraint Satisfaction Problem (CSP)

To solve CSPs, solvers initially assign values to variables using search strategies. Subsequently, these strategies search to change the value of one variable at a time. Optimization problems can also include search strategies, especially local search methods [6]. Search algorithms rely on a data structure to track the search tree. For each node *n* in the tree, *n.parent* is the node in the search tree that generated it, *n.action* is the action applied to the parent to generate *n*, and *n. cost* is a function that assigns a numeric cost to each path from the initial node to the current node. Then, the solver will use the cost function to prioritize specific paths during the search process [6].

In architectural design, a constraint system is defined as the collection of geometric entities and their corresponding constraints, which describe the interactions between these entities. In the context of geometric CSPs, the objective is to determine geometric entities' positions, orientations, and dimensions to satisfy all the given constraints [7]. CSPs have been used for building layout design [5,8,9], building massing design [4], façade design [10], 3D object layout design [11,12], and schematic and layout design for services in a building's ceiling void [13]. Methods to solve CSPs in the design field include **numeric optimization, local search,** and **direct search** (branch and bound methods, and other heuristic procedures) [14].

Note that designers must convert the constraints and parameters into mathematical expressions to use the CSP solvers that leverage these three methods.

Numeric optimization: Numeric optimization algorithms optimize an objective function concerning some variables while constraints are applied to those variables [6]. Li et al. [8] use a commercial nonlinear optimization tool, LINGO by LINDO Systems as the solver for layout design. They use Successive Linear Programming (SLP) and Generalized Reduced Gradient (GRG) algorithms to optimize and find solutions. Shikder et al. [9] also use a gradient-based approach for layout design as the optimization method.

Local search: Local search algorithms begin from a single node and gradually move to neighboring nodes. These algorithms are inspired from statistical physics, such as simulated annealing, or from evolutionary biology, such as genetic algorithms [6]. Genetic algorithms have been used to guide the search process for facade design and 3D object layout design [10,11]. Larive et al. [12] use local search to randomly instantiate variables and explore their neighborhood based on a cost function to design a 3D object layout.

In this paper, we do not use these two CSP methods because the search space consists of discrete objects (apartments) rather than continuous and numerical variables. Additionally, we aim to decrease randomness and allow designers to have more control over the search process. Hence, we have chosen the direct search method.

Direct search: Direct search algorithms use systematic search strategies to determine the next variable replacement in a CSP. These search strategies all share basic tree search algorithms and differ primarily in their selection of the next node to expand(Section 3.2) [6]. The constraints and design parameters must be defined as mathematical models for the constraint solver engines to perform direct searches. However, as the complexity of the design elements and parameters increases, these mathematical models become harder to define, modify, and debug.

For instance, Donath and Bohme [4] employ two different tools for designing building massing using direct search. The first tool, OPL STUDIO 3.7.1 modeling environment, utilizes Depth-First Search (DFS). However, due to the lack of 3D visualization, they were only able to solve separate subdomains of the building massing. The second tool, MAXON's CINEMA4D XPRESSO visual scripting environment, offered a GUI but had limitations on shape diversity and the number of blocks. Furthermore, the constraint system had to be modeled as a directed graph. In another study, Medjdoub et al. [5] use ARCHIPLAN that utilizes DFS for layout planning. They optimize the search by minimizing a cost function: after it finds the first solution, it uses a cost to bound the search space further. However, their results are in 2D.

As mentioned above, these papers are confined to simple or 2D geometries primarily because the problems have to be explicitly defined as mathematical expressions. Additionally, the number of their input elements is limited.

2.2. Light and privacy constraints

In this section, we briefly discuss the background regarding the light and privacy constraints implemented in the constraint solver.

Regarding the privacy condition, we calculate the visible volume inside one apartment as viewed from the window of another apartment. Calculating the visible volume has also been used to quantify visual openness, which is the amount of open space (not built) visible to an individual at a specific point. Fisher et al. [15,16] have used visual openness for urban environments and interior design.

For light condition, we use inverse sun rays. Inverse sun rays are vectors emitted from a point towards the sun's direction (represented by the mean point or points corresponding to the sun's location) to determine whether a point is in shadow. Schwartz [17] uses inverse sun rays emitted from several points to assess the potential glare for pedestrians. Additionally, Miranda et al. [18] use inverse sun rays to identify how long a given location is in shadow over a given time period.

3. Methodology

This section begins by providing an overview of the solver's function, the tools and languages utilized, the intended users of the solver, and the inputs required. Section 3.1 explains how using geometric objects in a CSP facilitates defining complex problems. Subsequently, detailed explanations regarding the search strategies (Section 3.2) and constraints (Section 3.3) will follow.

The solver arranges apartments to build a multi-story residential building on a regular 3D grid as a stack of 2D layers. Hence, all input apartments must consist of modules whose bounding box dimensions are identical to other modules and the grid. The solver searches through the apartment organization alternatives on each iteration and uses constraints to bound the search tree.

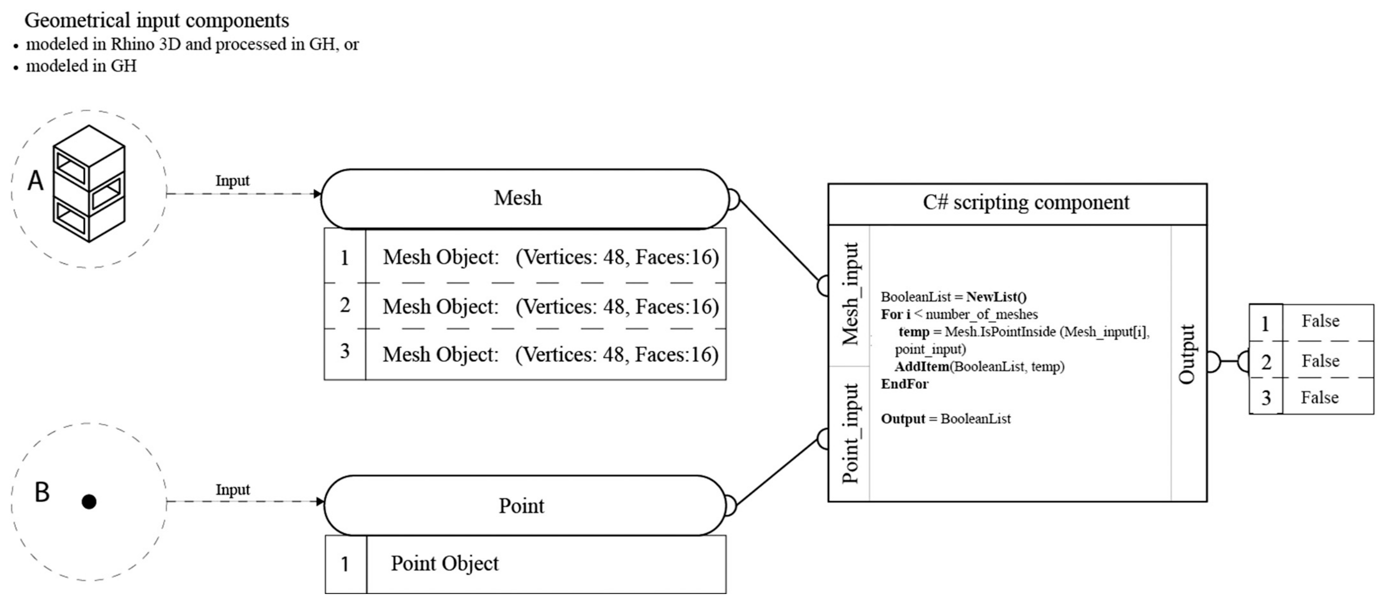
Regarding the tools and language, we use (1) Rhino 3D (a Computer-Aided Design (CAD) software) to create, modify, and document 3D models, and (2) Grasshopper (a visual programming plugin for Rhino3D) to create and modify 3D models by connecting building blocks called "components". The solver itself is developed as a new component within Grasshopper using C# language.

Rhino 3D and Grasshopper use Rhinocommon.dll and Rhino.dll libraries to create and modify geometries. Therefore, using the same libraries, we seamlessly input and manipulate geometric objects – built within Rhino 3D and Grasshopper – in our solver component. We can then develop the CSP solver and define constraints using these geometric objects. For instance, Fig. 1 illustrates three closed mesh objects built inside Rhino 3D. We input them into the solver's component in Grasshopper, where we can access them as a list and define a simple constraint using Rhinocommon library functionalities that checks if a point is inside the meshes.

As for the users, two types of users can benefit from this method:

1. Developers who define new constraints, input parameters, and search strategies and make modifications to the solver. Developers must be programming experts familiar with Rhino 3D, Grasshopper, and their geometric libraries (e.g., Rhinocommon).

2. Designers who use the solver to define their design problems. Designers only need to be familiar with Rhino 3D and Grasshopper interfaces. They adjust the constraints and solver inputs (already defined by developers) and explore design alternatives.


Geometrical input components
• modeled in Rhino 3D and processed in GH, or
• modeled in GH
A
Input
Mesh
1 Mesh Object: (Vertices: 48, Faces:16)
2 Mesh Object: (Vertices: 48, Faces:16)
3 Mesh Object: (Vertices: 48, Faces:16)
B
Input
Point
1 Point Object
C# scripting component
Mesh_input
Point_input
Output
1 False
2 False
3 False

```
BooleanList = NewList()
For i < number_of_meshes
    temp = Mesh.IsPointInside (Mesh_input[i],
    point_input)
    AddItem(BooleanList, temp)
EndFor

Output = BooleanList
```

Fig. 1. A geometric constraint for point inclusion in meshes.

Designers can adjust inputs falling into four categories: geometric, numeric, boolean, and string. Grasshopper provides features for representing and storing data related to these inputs. In the following, we give an example for each input type.

Geometric object inputs can include lists of meshes, surfaces, and other geometric object types. String inputs store data relevant to the constraints or the solver via text (e.g., specifying "south" for a module that requires light from the south, as described in Section 3.3.2). Boolean inputs provide true or false options through button components, allowing designers to control the application of specific constraints. Numeric inputs can be lists of numbers or single numerical values, such as specifying the number of floors. The Appendix contains a figure illustrating the inputs and outputs of the solver.

3.1. Object-oriented data modeling

Object-oriented data modeling describes approaches in which abstract models of software systems are created and systematically transformed into low-level implementations. This approach aims to reduce the gap between problems and software implementation, while also hiding the complexities of a system from its users and enabling them to tackle complex challenges [19].

In this paper, a geometric object is an abstract model of the mathematical implementation behind it. We use this approach to define the geometrically complex challenge of residential building organization as a CSP problem. Hence, designers can use constraint satisfaction methods without having to directly engage with the underlying mathematical formulas.

3.2. Search strategy

There are two classes of direct search for CSPs: Forward Search (FS) and Backward Search (BS) [20]. We have built a custom FS solver that employs two tree search strategies, namely Breadth-First Search (BFS) and Depth-First Search (DFS), to explore the solution space. In our search tree, each layer of depth corresponds to a floor, and the nodes represent different possible organizations of apartments on that floor. BFS expands the root node first and then expands all its successors. Consequently, all nodes at a given depth in the search tree are expanded before any node at the next level (Fig. 2-a). Conversely, DFS expands the deepest node in the current frontier of the search tree. It proceeds immediately to the deepest level of the search tree, where the nodes have no successors [6] (Fig. 2-b).

BFS: In our solver, BFS examines each floor's organizations (nodes) before proceeding to the next floor. Hence, a complete solution (building) is obtained only when all building alternatives have been explored until the last floor. BFS is suitable when (1) the number of apartments and floors is small, (2) the constraints effectively limit the branching, and (3) all path costs are equal.

Advantages: If there is any solution, it will find it (it is exhaustive) and yield all the solutions. Additionally, designers can manage the outputs by floor, enabling computations for specific subsets of floors.

Disadvantages: If the branching factor is not controlled, it can result in time and computer space limitations because the graph needs to store the previously expanded nodes. The time and space complexity of BFS with a depth of d and a branching factor of b would be $O(b^d)$[6].

DFS: DFS examines an entire building before moving on to the next alternative. Backtracking is used to reduce the total number of calculations. This method is suitable when (1) only a limited number of solutions are needed, rather than all possible solutions, (2) the constraints do not effectively bound the branching, and the search space is big.

Advantages: The computer space problem will reduce because once a node has been expanded, it can be removed from memory as soon as all its descendants have been fully explored. For a branching factor of b and a maximum depth of m, DFS requires the storage of $O(bm)$ nodes. We use a variant of DFS called backtracking search, in which only one successor is generated at a time, and each partially expanded node remembers the next successor to generate. . Hence, only $O(m)$ memory is needed [6].

Disadvantages: Although DFS can be exhaustive in searching the entire tree, in our case, designers must set a limit for the number of outputs, making it non-exhaustive.

Designers can improve the DFS by assigning a cost for apartment attributes. The node with the lowest cost on each floor will be explored first.

This paper focuses solely on BFS, DFS, and a cost function. Heuristic search methods can be applied to these search strategies to improve the search. However, applying them and analyzing how they will affect the design exploration process is outside the scope of this paper. In the following paragraphs, we discuss other search strategies, their practicality, and why we excluded them from solving our problem.

(1) Depth-limited search and (2) iterative deepening depth-first search: DFS fails in problems with infinite nodes to explore due to infinite depth. To address this, (1) depth-limited search supplies DFS with a

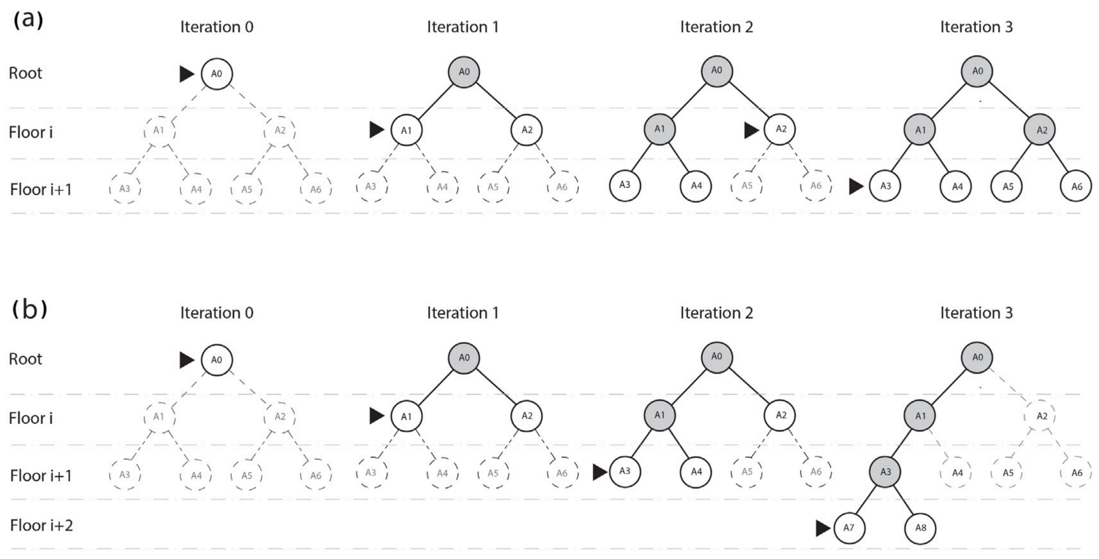


Fig. 2. (a) The BFS logic for search (b) The DFS logic for search.

predetermined depth limit *l*. That is, nodes at depth *l* will have no successors. (2) iterative deepening DFS gradually increases the depth limit, starting from 0 and incrementing by 1, until it reaches the shallowest goal [6]. In our search tree, the depth corresponds to the finite number of floors in a building. Hence, these two strategies will have little benefit to our problem.

(3) *Iterative lengthening search*: this strategy is like iterative deepening but uses increasing path-cost limit instead of increasing depth limit [6]. The cost we have introduced in this work is also finite as it corresponds to the finite number of modules on each floor. Therefore, this search strategy will have little benefit to our problem.

(4) Bidirectional search: Bidirectional search runs two simultaneous searches — forward search and backward search. The two searches should meet in the middle [6]. This strategy would be valuable if we already knew the buildings' final organizations and sought only the sequence of actions that led to these organizations, which is not the case in our problem.

3.2.1. *Creating a grid*

In the first step, designers create a 3D grid to define the building's boundary and specify four types of points (Fig. 3): (1) a vertically aligned series of points extending from the first to the last floor that builds the connection shaft (numeric input set by designers), (2) entrance points adjacent to the connection shaft points (numeric input set by designers), (3) invalid points to which apartments should not move (geometric input set by designers) (4) valid points to which apartments can move (automatically processed using previous data).

3.2.2. *Data structure*

We require a structured approach to facilitate the search process, incorporate costs and constraints, and provide organized data for designers to adjust the buildings. The primary classes are Space, Apartment, and Building: (Fig. 4)

Space: The "Space" class represents an individual module and includes the following attributes:

1. Type (numeric input set by designers): A whole number (0 to 50) that specifies the module's type. Type zero always belongs to the entrance module, but designers set the other types according to their preferences. For example, type one could represent a room module.
2. Light (numeric input set by designers): A whole number (0 to 50) that determines the amount of daylight the module needs. Light zero is always for modules that do not need any light. Designers define additional codes based on their needs and specifications (further details in Section 3.3.2).
3. Mesh (geometric input set by designers): A joined single mesh that sets the module's geometry.
4. Cost (numeric input set by designers): A whole number (0 to 50) that prioritizes the nodes for expanding in DFS. The higher the cost function, the lower the chance of being expanded in the search.

Other Space attributes will be set automatically during the code execution.

Apartment: The "Apartment" class represents an individual apartment and includes the following attributes:

1. Space list: a list of the Apartment's Space class objects
2. Cost: the average cost of the Apartment's Space objects
3. Rotation angles (numeric input set by designers): rotation angles specified for the Apartment objects

Building: The "Building" class represents an individual building alternative and includes the following attributes:

1. Apartment list: a list of all Apartment class objects in an alternative classified by floor
2. Point list: a list of invalid and occupied points of an alternative

3.2.3. *Creating the nodes*

On each iteration, a selection of apartments will be put around the shaft in a certain order, and we call each of these alternatives an apartment organization. To calculate all of the apartment organizations (nodes of the search tree), designers must set the following inputs: (Fig. 5 illustrates an example for one floor)

- a list of all the Apartment objects, e.g., {A, B} (Apartment objects built as explained in Section 3.2.2)
- a list of the number of apartments on each floor, e.g., {2} (numeric input set by designers)
- a list of apartment indexes to be used on each floor, e.g., {(0,1)} (numeric input set by designers)

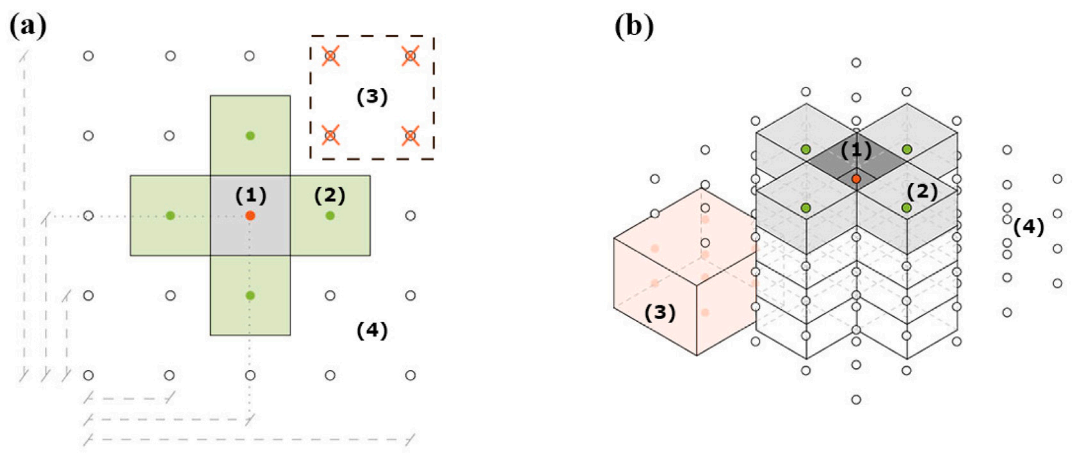


Fig. 3. (a): Different types of points in the building grid representation: (1) red points as the vertical shaft, (2) green points as entrance, (3) crossed points as invalid points, and (4) white points as valid points. (b): 3D view of the same points in the building grid.

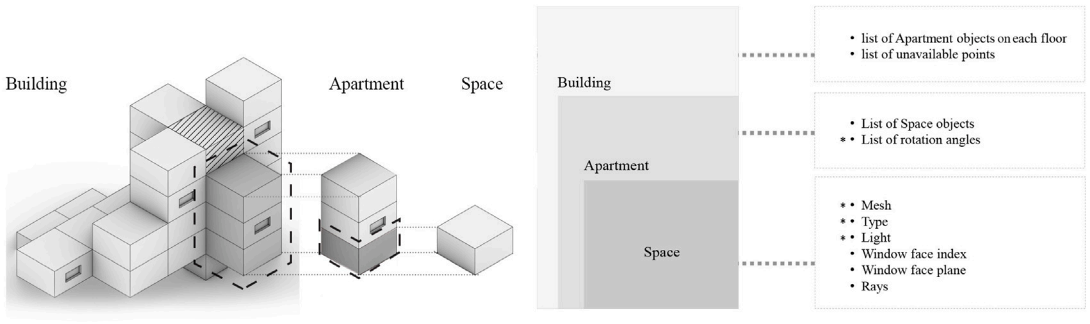


Fig. 4. The hierarchy of classes and their attributes (designers set the attributes marked with (*)).

- a list of available entrance points on each floor, e.g., {0,1,2,3} (numeric input set by designers)

First, we calculate the “combination with repetition” for apartments on each floor separately. This example considers two kinds of apartments (A, B) to choose from, and we want two apartments on the floor. Consequently, the number of possibilities for choosing apartments would be $C(n,r)$ (Fig. 5-b).

$$C = \binom{n+r-1}{r} \tag{1}$$

Second, we calculate all possible positions to which these apartment combinations can move. We use “permutations without repetition” for entrance point indexes. The number of possible positions would be $P(n,r)$ (Fig. 5-a).

$$P = \frac{n!}{(n-r)!} \tag{2}$$

Now, the total number of possible organizations for this floor would be:

$$J = C \times P \tag{3}$$

In some cases, repetitive organizations (K) can occur. For example, if the chosen apartment selection is (A, A), then the positions (0,1) and (1,0) would be identical. K is calculated with a function that substitute each position on a floor with the apartment indexes and deletes the identical organizations; the number of unique organizations for each floor would be:

$$I = J - K \tag{4}$$

Given that we have all the positions for each floor, the solver searches through them and checks the constraints to find the solutions.

3.2.4. Breadth-First search

We show the unique organizations of each floor n as I_n (Eq. (4)), and the unique organizations that have satisfied all the constraints as I_n^*. The solver first applies the constraints to all the selections of apartments and positions for the first floor (I_1). Then, only the organizations that satisfy all the constraints, I_1^*, are outputted for the next floor. Hence, the number of outputs that need to be evaluated for the second floor is as follows:

$$N = \left(I^*{}_1\right) \times \left(I_2\right) \tag{5}$$

3.2.5. Depth-First search

The solver goes from the first to the last floor for each building on each iteration. So, the maximum number of unique organizations stored and checked for each iteration is equal to the number of floors (one unique organization per floor). If a node fails to satisfy all the constraints, the iteration backtracks to the previous node, and none of the nodes after the invalid one will be expanded. Additionally, designers can guide the search by assigning costs to the Space objects. In this case, the branch with the least cost will be selected first.

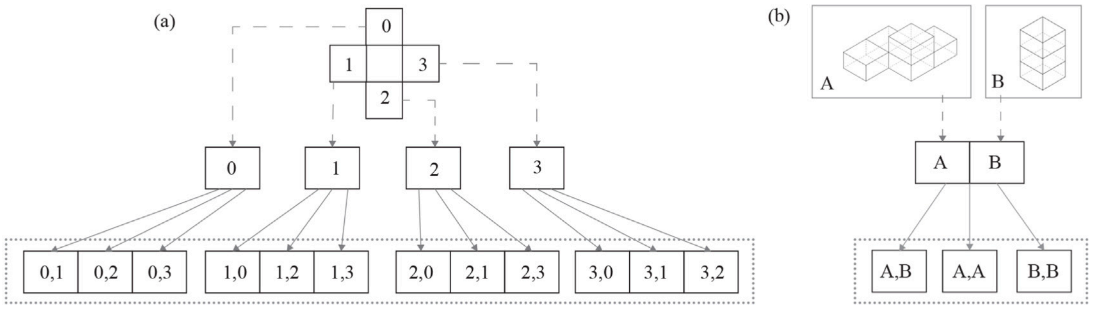


Fig. 5. (a) Permutation without repetition for entrance point indexes of the floor (b) Combination with repetition for apartment indexes.

3.3. Constraints

Definition of constraints directly affects the performance of the solver and the quality of its outputs. Therefore, it is essential to define constraints that effectively limit the search space and are computationally easy to evaluate. We have implemented three examples of geometrical constraints to test the method: overlapping, light, and privacy. We choose the light and privacy constraints for two reasons: (1) they require complex geometrical calculations, and (2) they are common in designing residential complexes. These constraints use geometric objects rather than direct mathematical inputs. For instance, we represent a circle as an object rather than defining it using the equation $(x-h)^2+(y-k)^2 = r^2$, where (h, k) represents the center and r represents the radius of the circle (Section 3.1).

3.3.1. Overlapping

The overlapping constraint checks if any module center falls within the invalid points list and detects module overlap.

3.3.2. Light

The Light constraint checks if the modules of an Apartment object receive enough light. This constraint optimizes the calculation process through two approaches: (1) emitting rays from the module's faces rather than the sky (inverse sun rays) and (2) specifying the faces of the apartment that require evaluation. The Space class (Section 3.2.2) includes information about which faces can receive light. Designers specify this information using string inputs. For instance, if a module can receive light from the south or east, designers input "south, east". These designated faces are referred to as valid faces. Other inputs include:

1. *Ray vectors* (geometric input set by designers): Designers input a list of vector objects. These vectors represent the rays emitted from module faces, which can be the negative vector of average sun rays or custom rays.
2. *Light receiving surfaces* (geometric input set by designers): Designers input a list of surface objects surrounding the building. Light rays travel from the points located on the valid faces of a module towards the direction of ray vectors. Then, if these rays intersect with the light-receiving surfaces, the rays would be considered successful — light can reach that point on the module's face (Fig. 6-a).
3. *Accuracy (numeric input set by designers)*: Designers input a number for accuracy, indicating the number of rays emitted from each face.
4. *Light obstacles (geometric input set by designers)*: Designers input a list of solid objects (cubes, cuboids, etc.), representing obstacles that may obstruct light.
5. *The number of reflections (numeric input set by designers)*: Designers input a number that represents how many times an emitted ray can be reflected. This parameter is adjustable if designers want to consider indirect light. However, for simplicity, indirect rays reflected from other blocks will not be calculated (Fig. 6-c).

Based on each module's light setting (valid faces), the solver checks the light constraint module by module. For each face, a list of rays, equal in number to the specified *accuracy* (a) will be emitted in the direction of the *ray vectors* (with the constraint that the angle between the ray vectors and the face's normal vector does not exceed 90 degrees). The rays will succeed if they reach the *light receiving surfaces* (Fig. 6-a). The solver then calculates the number of successful rays (s) for that face. Based on the ratio (b) (set by default to 0.5, but is adjustable), the constraint $s > a \times b$ is evaluated as either true or false (Fig. 6-b). If any module fails to satisfy the light condition, the corresponding alternative is discarded.

Once the alternatives are generated, designers can access all objects attributes. For example, designers can access the successful ray origins, which can help them in determining optimal window positioning.

3.3.3. Privacy

The privacy constraint checks the privacy between two modules with windows from different apartments. It minimizes calculations by checking only two numbers:

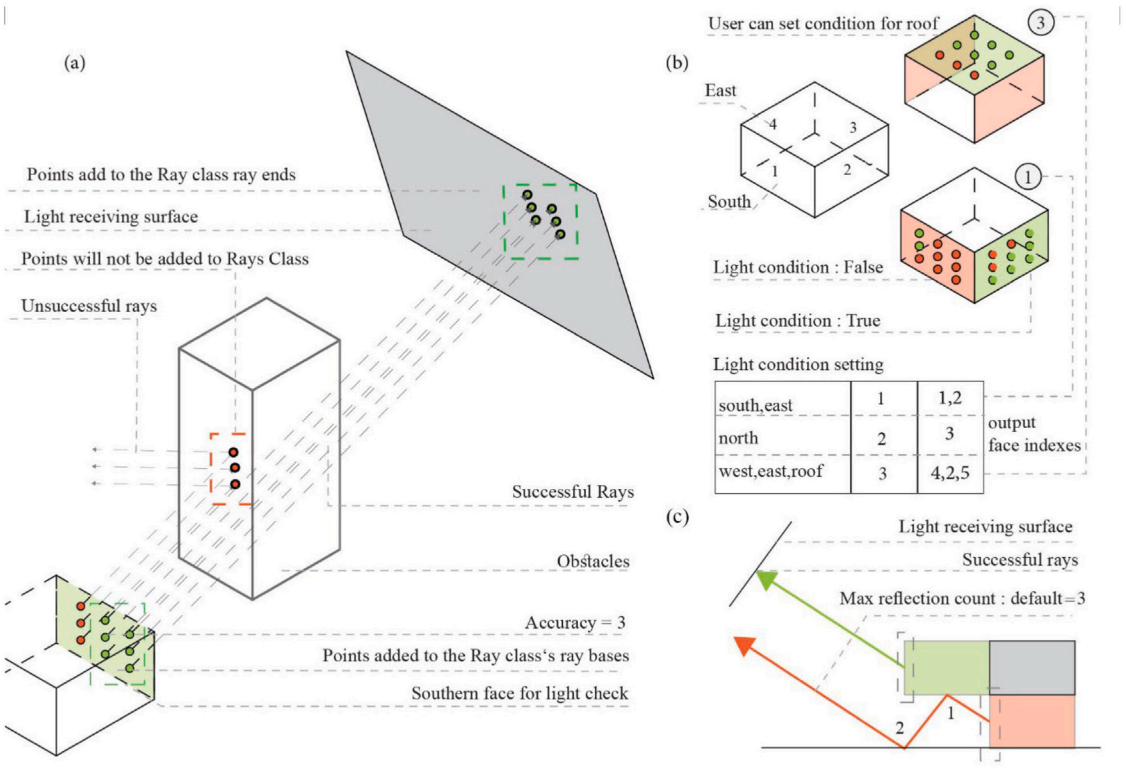
- Distance (numeric input set by designers): the distance between the centers of the windows in two different apartment modules (Fig. 7-b).
- Angle (numeric input set by designers): the angle between the normal vector of the face and the line connecting the centers of the two faces. The angle is denoted as α for one face and β for the other face (Fig. 7-b).

Designers must input two limit values:

(1) Maximum distance value: Beyond this distance, privacy checking becomes unnecessary. The distance value will be checked when the privacy constraint executes between two modules. If the distance is lower than the input limit, the angle α is checked.

(2) Maximum α value: This angle represents the threshold beyond which the visible volume is considered unacceptable. The maximum α value is denoted as α^*, and the ratio of the visible volume $\left(\frac{\text{visible volume}}{\text{modules' volume}} \times 100\right)$ as r. Designers use a separate algorithm to find the α^* value based on the input r value. This algorithm (1) calculates the visible volume from another module's window while changing α (with adjustable window locations), and (2) uses the Galapagos[1] component to calculate α^* parameter with a fixed y value and variable x values (Fig. 7-a). Finally, the solver checks the constraint $((\alpha \mid \beta) > \alpha^*)$.

[1] The Galapagos component is an optimization tool within Grasshopper plugin that utilizes evolutionary and annealing algorithms.



south,east	1	1,2
north	2	3
west,east,roof	3	4,2,5

Fig. 6. (a) The process of checking light conditions for each module (b) Module face indexes and designers' light condition settings (c) Maximum reflection count for a face (higher values impact search speed).

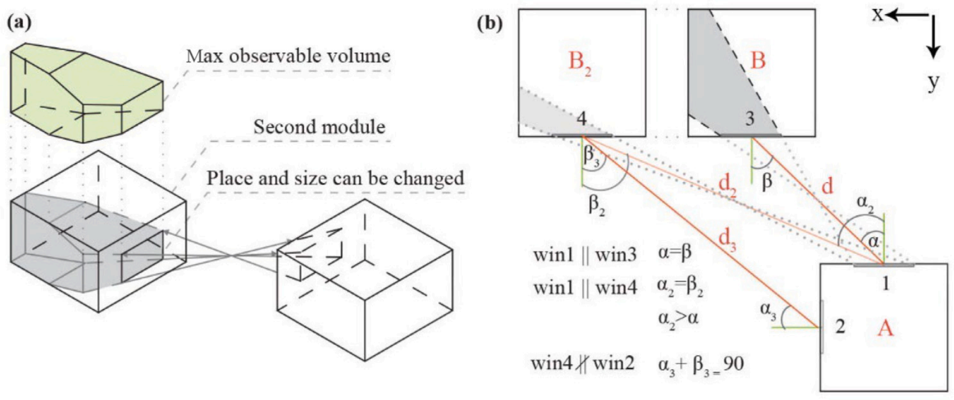


Fig. 7. (a) The visible volume from another apartment (b) Calculation of the α value (changing the x value impacts α and (d).

3.4. Constraint propagation

Constraint propagation communicates the domain reduction of a variable to all of the constraints applied to this variable. Hence, constraint propagation reduces the variables' domain. This process continues until no more variable domain can be reduced or when a domain becomes empty, resulting in a failure [6].

For instance, if the number of apartments to be organized on a floor exceeds the number of valid entrance points on that floor, the search will not start. Similarly, if a module is only set to receive light from one face, and that face is connected to another module within the same apartment, the light constraint would not be checked for the entire apartment. Hence, the domain of the apartments reduces.

Designers can adjust the constraints and the search process. So, the more "strict" the constraints are set, the more variable domains will be reduced. To illustrate this, we provide examples for each constraint to clarify what we mean by "strict". (1) In the light constraint, each module can receive light from four faces (south, east, north, and west). Therefore, a module that can receive light from only one face (e.g., south) has a stricter constraint compared to a module that can receive light from three faces (e.g., south OR east OR north). (2) In the privacy constraint, a ratio of the visible area (Section 3.3.3) between 5

Table 1
The setting for E1.

Integer and Boolean inputs			String inputs			Apartment data				
Module dimensions	5 * 5 * 3 (m)		Light conditions	North	1		Module Id	Type	Light	Cost
Floor count	4			West		A	0	0	0	0
Light constraint	On			North	2		1	2	3	0
Privacy constraint	On			East			2	1	0	0
Apartment count on each floor	Floor	Apartment count		South	3		3	3	1	0
	1	2		East			4	1	0	0
	2	2	Privacy data	Distance	15	B	0	3	1	0
	3	2		Angle	60		1	1	0	0
	4	1					2	0	0	0

to 15% is more strict than a ratio exceeding 70%. (3) In the overlapping constraint, limiting the valid entrance points from the beginning is more strict than allowing all the entrance points to be available.

3.5. *After outputting the alternatives*

Designers must input geometries with only the necessary details because this solver is particularly suitable for the design exploration stage. Further details can be added after the alternatives are outputted. Designers can still explore the design, add details such as windows and structure, and access the data of each building alternative through the object-oriented data model we have created. For example, a designer can select certain types of apartment modules (Section 3.2.2) to add detail to them.

3.6. *Examples*

We introduce three different settings for testing our method. Each of the modules of the apartments has an ID (Fig. 8), based on which Type and Light codes have been assigned (Section 3.2.2).

E1: Only two apartments are used, and the number of floors is set to four. We use both light and privacy constraints for this example. We have defined three light conditions, each can receive light from two sides. For privacy, we set the visible volume to 10 percent, resulting in an angle of 60 (Table 1).

E2: Four different apartments have been used, and the number of floors is set to three. Both light and privacy constraints are active, and we have defined four light conditions. Additionally, we changed the visible volume for privacy to 15 percent, which resulted in an angle of 50 (Table 2).

E3: Eight different apartments have been used, and the number of floors is increased to eight. Again, both light and privacy constraints are active. We have defined five light conditions, and privacy constraint details are the same as in E2 (Tables 3 and 4).

4. Results

There is no specific building alternative solution for this method, and the results depend on designers and their inputs. We show some of the valid alternatives created by our solver using the three examples E1, E2, and E3.

For E1 and E2, we first use the BFS algorithm and assign no cost. In these examples, BFS is suitable because we have inputted a few apartments (two for E1 and four for E2) and activated both light and privacy constraints. Therefore, we can search for all the valid alternatives. As a result, the solver outputs seven valid alternatives in 40 s[2] for E1 (Fig. 9) and forty-three valid alternatives in 5.1 min for E2. Fig. 10 illustrates four randomly selected outputs from these forty-three alternatives.

To compare the two search strategies, we conduct a DFS for E2 with the same setting as Table 2 without using cost. We set the limit for the number of outputs to four. Hence, the solver computed the four valid alternatives in 1.4 min (Fig. 11).

The time needed for a DFS depends on where the solutions lie in the search tree. In the worst case, this time will be the same as BFS. However, the required computer space makes DFS more suitable for larger search trees. Note that these four alternatives would be identical to alternatives 0 to 3 produced by BFS because we assigned no cost.

The four outputs of DFS are illustrated in Fig. 12 when we assign a cost value of 2 to all modules with type 2 (Table 2-the empty frame), and a cost value of 1 to the remaining modules. This approach allows the search to prioritize alternatives with lower path costs.

Finally, E3 tests a more complex design with eight floors and eight apartments to evaluate the solver on larger-scale designs. Since this setting will have many solutions, we only use DFS to solve it and set the limit for the number of outputs to 50. This setting will provide us with the first 50 valid alternatives in the search tree. Fig. 13 shows four of these 50 outputs. It took 40.7 min for the solver to output the first 50 solutions, which proves that the computer will not run out of space while calculating the solutions, but it takes more time to find and output 50 valid alternatives. Additionally, because of the way DFS performs the search, we can see that the first 50 solutions only differ on the top floor (deeper layers of the search tree).

5. Discussion and conclusion

Conducting a direct search in the solutions domain is computationally expensive for organizing a residential complex because the search tree (building alternatives) grows exponentially as the number of apartments and floors increases.

We have utilized object-oriented data modeling to build a CSP solver within the Grasshopper plugin in Rhino 3D. The solver conducts BFS and DFS without requiring strict mathematical models for defining the problems and constraints.

Our CSP solver has two types of users: developers who edit the solver and create its inputs and constraints, and designers who utilize the solver to define their design problem. Developers can define objectives and conditions via geometrical objects and the features of Rhino 3D/Grasshopper. Conversely, designers can define their design problem as a CSP using the Rhino 3D/Grasshopper interfaces and control the search process by setting cost functions and adjusting the constraints (Section 3.4).

Regarding the search methods, designers can use each search method for a different purpose:

DFS is more suitable when dealing with large search trees where computing all the answers is unnecessary. However, in large-scale problems with numerous solutions, designers should adjust the number and types of apartments for each floor separately to have more variety on lower floors.

BFS provides all the valid solutions exhaustively but is practical only when the search tree is small and the constraints effectively limit the growth of the search tree.

[2] We used Asus G551jw, Intel Core i7-4720HQ, NVIDIA GeForce GTX 960M (4 GB GDDR5), and 16 GB DDR3.

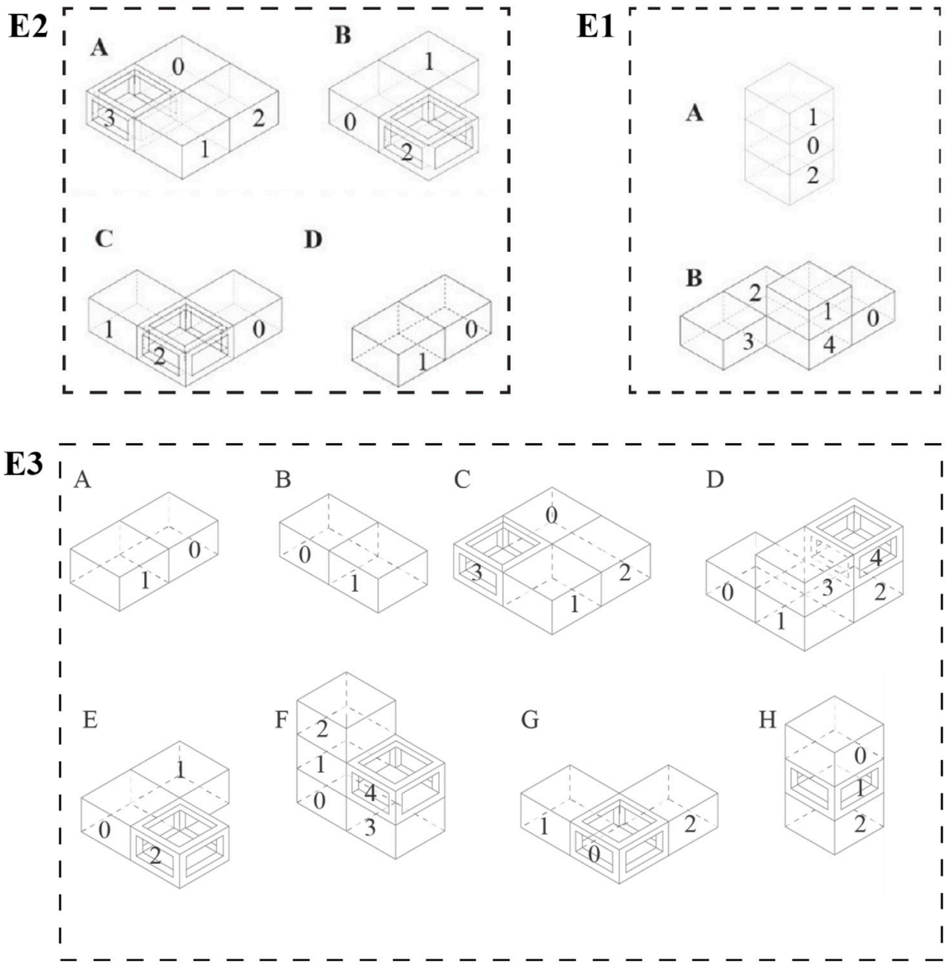


Fig. 8. Apartments used in the examples one,two, and three.

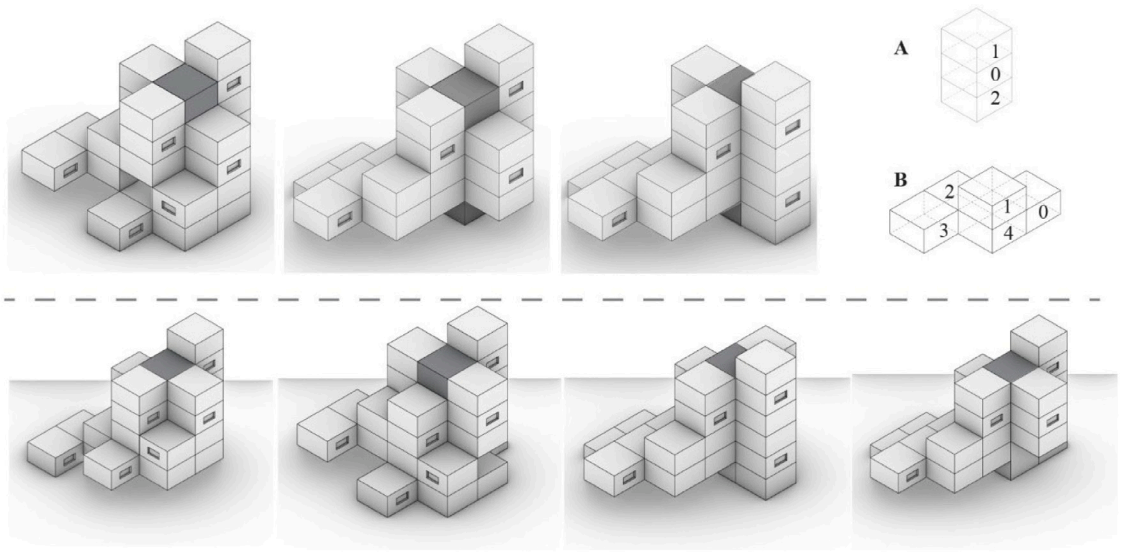


Fig. 9. All the seven alternatives from BFS (Table 1).

Table 2
The setting for E2.

Integer and Boolean inputs		
Module dimensions	5 ∗ 5 ∗ 3 (m)	
Floor count	3	
Light constraint	On	
Privacy constraint	On	
	Floor	Apartment count
Apartment count on each floor	1	3
	2	3
	3	3

String inputs		
Light conditions	West	1
	North East	2
	South East	3
	South	4
Privacy data	Distance	12
	Angle	50

Apartment data	Module Id	Type	Light	Cost
A	0	0	0	0
	1	2	2	0
	2	3	3	0
	3	3	3	0
B	0	1	1	0
	1	0	0	0
	2	2	4	0
C	0	1	2	0
	1	0	0	0
	2	2	3	0
D	0	0	0	0
	1	1	3	0

Table 3
The setting for E3.

Integer and Boolean inputs			String Inputs		
Module dimensions	5.5.3 (m)		Light condition	North	1
Floor count	8			North East	2
Light condition	On				
Privacy condition	On			South East	3
	Floor	Apartment count		South North	4
Apartment count on each floor	1,2	3,3			
	3,4	3,2		South	5
	5,6	2,2	Privacy condition	Distance	15
	7,8	3,2		Angle	60

Table 4
The apartments input setting for E3.

Apartment	A		B		C				D					E			F					G			H		
Module ID	0	1	0	1	0	1	2	3	0	1	2	3	4	0	1	2	0	1	2	3	4	0	1	2	0	1	2
Type	0	1	0	1	0	1	1	2	1	0	1	1	2	0	1	2	0	1	1	1	2	2	0	1	1	2	0
Light	2	3	4	3	0	3	2	5	0	3	2	3	2	0	2	5	0	0	4	3	3	3	0	2	2	2	3

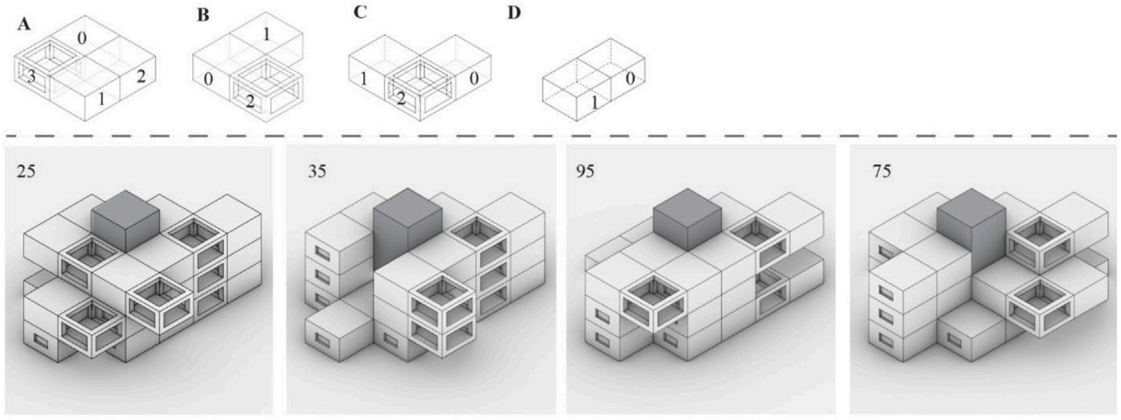


Fig. 10. Four randomly selected alternatives out of 43 from BFS (Table 2).

As for constraints, we have implemented three geometrical constraints in the solver to prove the concept: overlapping, light, and privacy. In the light constraint, light rays originate from each face and will get calculated if caught by the light catchers. For the privacy constraint, each pair of modules is evaluated based on the distance between windows and an angle corresponding to the observable volume.

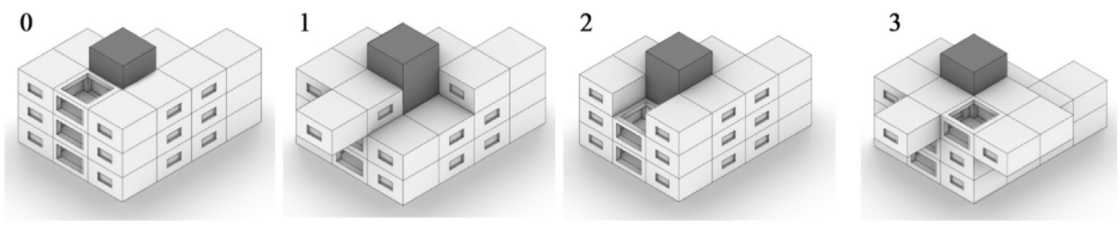


Fig. 11. Four valid alternatives from DFS (no cost assigned, Table 2).

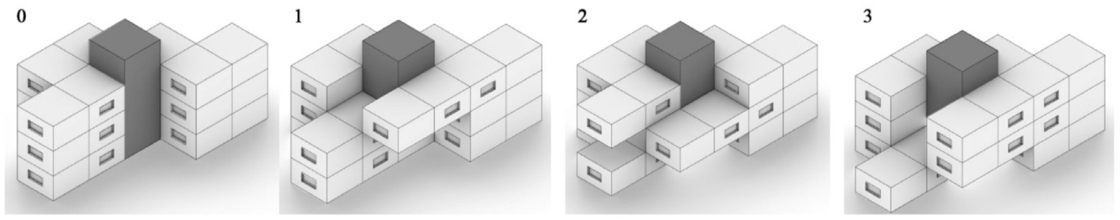


Fig. 12. Four valid alternatives from DFS (cost assigned to the framed module).

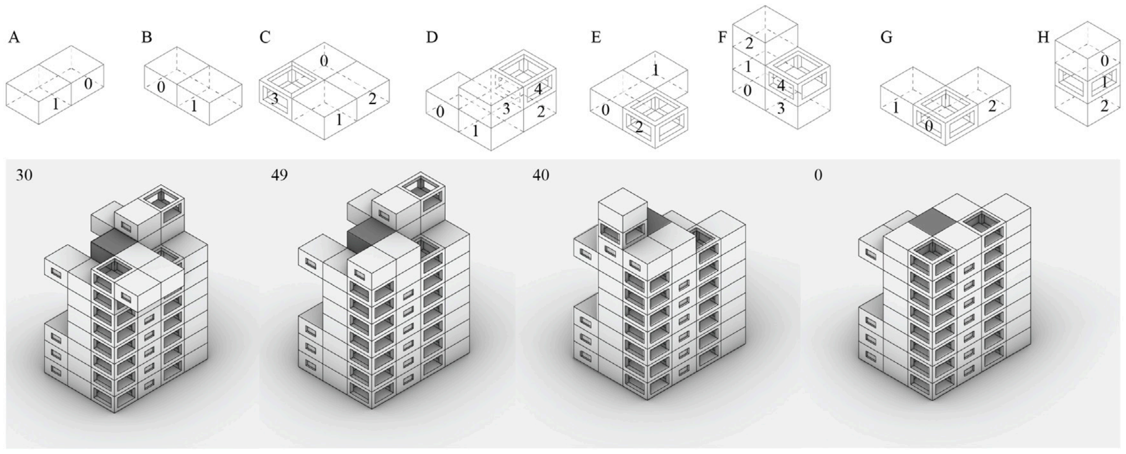


Fig. 13. Four valid alternatives from DFS (eight floors and apartments, Tables 3 and 4).

Future improvements for our solver include: (1) Developers can add new constraints. These constraints can be existing simulators implemented as black boxes within the solver. However, the efficiency of these simulators may vary due to differences in data modeling structure. Future research can explore adapting their logic to a similar data model as our solver. (2) Developers can add heuristic search methods that allow designers to perform A* search, greedy best-first search, and explore the design space more effectively in large-scale problems. (3) A survey can be conducted to compare the process of designing using our solver with other solvers that rely on purely mathematical inputs. (4) Finally, the solver can be made available as a plugin for Grasshopper.

Declaration of competing interest

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

Data availability

We have shared a link to our code: https://github.com/shshermin/CSP-solver-for-residential-building-design.

Acknowledgments

This research was supported by:
the Ministry of Science, Research, and the Arts (MWK) Baden-Württemberg, Germany, within the Artificial Intelligence Software Academy (AISA) - 66034654 and the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2120/1 - 390831618

Appendix

See Fig. A.1.

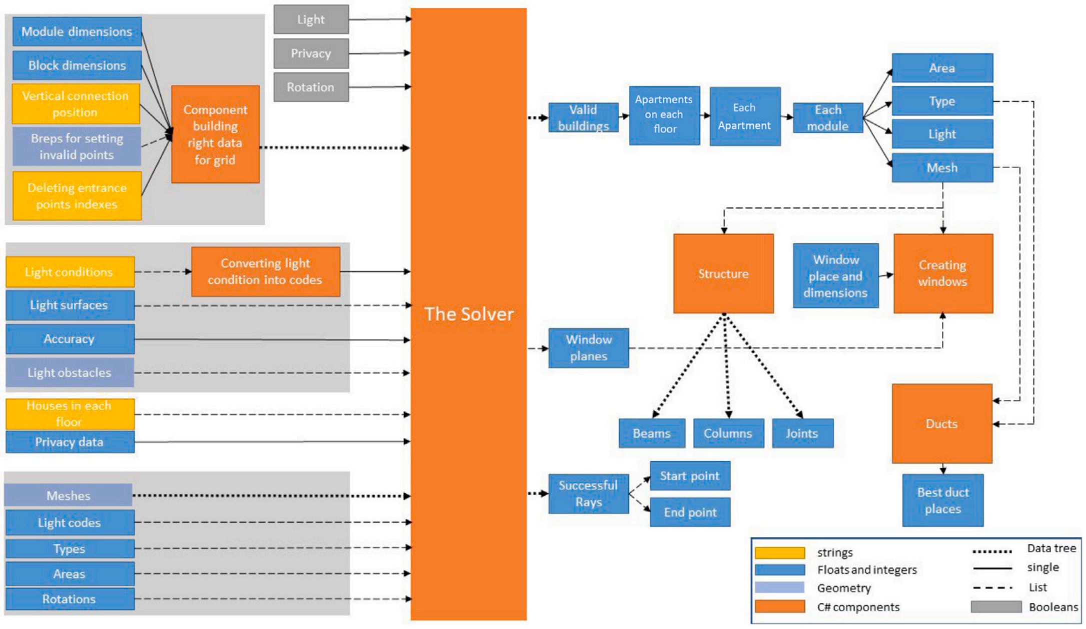


Fig. A.1.

References

[1] Sven Schneider, Jan-Ruben Fischer, Reinhard König, Rethinking automated layout design: Developing a creative evolutionary design method for the layout problems in architecture and urban design, in: John S. Gero (Ed.), Design Computing and Cognition '10, Springer Netherlands, Dordrecht, 2011, pp. 367–386, http://dx.doi.org/10.1007/978-94-007-0510-4_20, ISBN 9789400705098 9789400705104, URL http://link.springer.com/10.1007/978-94-007-0510-4_20.

[2] Peter J. Bentley, David W. Corne, An introduction to creative evolutionary systems, in: Creative Evolutionary Systems, Elsevier, ISBN: 9781558606739, 2002, pp. 1–75, http://dx.doi.org/10.1016/B978-155860673-9/50035-5, URL https://linkinghub.elsevier.com/retrieve/pii/B9781558606739500355.

[3] Malik Ghallab, Dana S. Nau, Paolo Traverso, Automated Planning: Theory and Practice, Elsevier/Morgan Kaufmann, Amsterdam ; Boston, ISBN: 9781558608566, 2004.

[4] Dirk Donath, Luis Felipe González Böhme, Constraint-based design in participatory housing planning, Int. J. Archit. Comput. 6 (1) (2008) 97–117, http://dx.doi.org/10.1260/147807708784640081, ISSN 1478-0771, 2048-3988, URL http://journals.sagepub.com/doi/10.1260/147807708784640081.

[5] B. Medjdoub, B. Yannou, Dynamic space ordering at a topological level in space planning, Artif. Intell. Eng. (ISSN: 09541810) 15 (1) (2001) 47–60, http://dx.doi.org/10.1016/S0954-1810(00)00027-3, URL https://linkinghub.elsevier.com/retrieve/pii/S0954181000000273.

[6] Stuart J. Russell, Peter Norvig, Ernest Davis, Artificial Intelligence: A Modern Approach, third ed., in: Prentice Hall series in artificial intelligence, Prentice Hall, Upper Saddle River, ISBN: 9780136042594, 2010.

[7] Glenn A. Kramer, A geometric constraint engine, Artificial Intelligence (ISSN: 00043702) 58 (1–3) (1992) 327–360, http://dx.doi.org/10.1016/0004-3702(92)90012-M, URL https://linkinghub.elsevier.com/retrieve/pii/000437029290012M.

[8] S.-P. Li, John H. Frazer, M.-X. Tang, A Constraint Based Generative System for Floor Layouts, Singapore, 2000, pp. 441–450, http://dx.doi.org/10.52842/conf.caadria.2000.441, URL http://papers.cumincad.org/cgi-bin/works/paper/5b5d.

[9] S. Shikder, A. Price, M. Mourshed, Interactive constraint-based space layout planning, 2010, http://dx.doi.org/10.13140/RG.2.1.2003.4963, URL http://rgdoi.net/10.13140/RG.2.1.2003.4963.

[10] Ming-xian Lee, Ji-Hyun Lee, Form, style and function - A constraint-based generative system for apartment façade design, in: Communicating Space(S) [24th ECAADe Conference Proceedings], CUMINCAD, ISBN: 0-9541183-5-9, 2006, pp. 874–883, Volos (Greece) 6-9 September 2006, URL http://papers.cumincad.org/cgi-bin/works/paper/2006_874.

[11] Stéphane Sanchez, Olivier Le Roux, Hervé Luga, Véronique Gaildrat, Constraint-based 3D-object layout using a genetic algorithm, in: Proceedings of International Conference on Computer Graphics and Artificial Intelligence. Limo., 2003, URL https://www.researchgate.net/publication/245776169_Constraint-based_3d-object_layout_using_a_genetic_algorithm/references (19/05/2023).

[12] Mathieu Larive, Olivier Le Roux, Veronique Gaildrat, Using meta-heuristics for constraint-based 3D objects layout, in: Proceedings of International Conference on Computer Graphics and Artificial Intelligence, 2004, URL https://www.researchgate.net/publication/245776169_Constraint-based_3d-object_layout_using_a_genetic_algorithm/references, (19/05/2023).

[13] G. Bi, B. Medjdoub, A Hybrid Approach to Solve Space Planning Problems, Rome, Italy, p. 12, http://dx.doi.org/10.4203/ccp.82.12, URL http://www.ctresources.info/ccp/paper.html?id=538.

[14] M.H. Imam, M. Mir, Nonlinear programming approach to automated topology optimization, Comput. Aided Des. (ISSN: 00104485) 21 (2) (1989) 107–115, http://dx.doi.org/10.1016/0010-4485(89)90146-2, URL https://linkinghub.elsevier.com/retrieve/pii/0010448589901462.

[15] Dafna Fisher-Gewirtzman, Nir Polak, A learning automated 3D architecture synthesis model: demonstrating a computer governed design of minimal apartment units based on human perceptual and physical needs, Archit. Sci. Rev. 62 (4) (2019) 301–312, http://dx.doi.org/10.1080/00038628.2019.1611537, ISSN 0003-8628, 1758-9622, URL https://www.tandfonline.com/doi/full/10.1080/00038628.2019.1611537.

[16] Dafna Fisher-Gewirtzman, Dalit Shach Pinsly, Israel A Wagner, Michael Burt, View-oriented three-dimensional visual analysis models for the urban environment, Urban Des. Int. 10 (1) (2005) 23–37, http://dx.doi.org/10.1057/palgrave.udi.9000133, ISSN 1357-5317, 1468-4519, URL http://link.springer.com/10.1057/palgrave.udi.9000133.

[17] Mathew Schwartz, Evaluating jogging routes in mass models, in: SimAUD '20: Proceedings of the 11th Annual Symposium on Simulation for Architecture and Urban Design, Society for Computer Simulation International, Virtual Event, Austria, 2020, pp. 93–100, ISBN 1-56555-371-3 (ISBN-10) 978-1565553712 (ISBN-13), URL https://www.simaud.org/proceedings/.

[18] Fabio Miranda, Harish Doraiswamy, Marcos Lage, Luc Wilson, Mondrian Hsieh, Claudio T. Silva, Shadow accrual maps: Efficient accumulation of city-scale shadows over time, IEEE Trans. Vis. Comput. Graphics 25 (3) (2019) 1559–1574, http://dx.doi.org/10.1109/TVCG.2018.2802945, ISSN 1077-2626, 1941-0506, 2160-9306, URL https://ieeexplore.ieee.org/document/8283638/.

[19] Robert France, Bernhard Rumpe, Model-driven development of complex software: A research roadmap, in: Future of Software Engineering (FOSE '07), IEEE, Minneapolis, MN, USA, ISBN: 978-0-7695-2829-8, 2007, pp. 37–54, http://dx.doi.org/10.1109/FOSE.2007.14, URL http://ieeexplore.ieee.org/document/4221611/.

[20] Ian P. Gent, Ewan MacIntyre, Patrick Presser, Barbara M. Smith, Toby Walsh, An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem, in: Gerhard Goos, Juris Hartmanis, Jan Leeuwen, Eugene C. Freuder (Eds.), Principles and Practice of Constraint Programming — CP96, Vol. 1118, Springer Berlin Heidelberg, Berlin, Heidelberg, 1996, pp. 179–193, http://dx.doi.org/10.1007/3-540-61551-2_74, ISBN 9783540615514 9783540706205, URL http://link.springer.com/10.1007/3-540-61551-2_74.