An efficient approach to address adjacency constraints in rectangular floor plan by using Monte-Carlo Tree Search

Feng Shi[a,b,c*], Ranjith K Soman[a], Jennifer Whyte [a,b], Ji Han[d]

[a] Centre for Systems Engineering and Innovation, Department of Civil and Environmental Engineering, Imperial College London, London

[b] The Alan Turing Institute, London

[c] Amazon Web Service EMEA SARL (UK Branch), London

[d] The University of Liverpool, Liverpool

*corresponding author: shi.feng.nwpu@gmail.com

Abstract

Manually laying out the floor plan for buildings with highly-dense adjacency constraints at the early design stage is a labour-intensive problem. In recent decades, computer-based conventional search algorithms and evolutionary methods have been successfully developed to automatically generate various types of floor plans. However, there is relatively limited work focusing on problems with highly-dense adjacency constraints common in large scale floor plans such as hospitals and schools. This paper proposes an algorithm to generate the early-stage design of floor plans with highly-dense adjacency and non-adjacency constraints using reinforcement learning based on off-policy Monte-Carlo Tree Search. The results show the advantages of the proposed algorithm for the targeted problem of highly-dense adjacency constrained floor plan generation, which is more time-efficient,

more lightweight to implement, and having a larger capacity than other approaches such as Evolution strategy and traditional on-policy search.

Keywords: Floor plan generation; highly-dense adjacency and non-adjacency constraint; algorithm; Off-policy Monte-Carlo tree search; reinforcement learning; generative design;

1. Introduction

Laying out a floor plan is one of the key tasks in architecture design. It involves making decisions on the design and layout of all the rooms usually in a 2D space to satisfy various geometric and topological constraints. Conventionally, this has been a manual trial and error drawing process, where different pieces are adjusted, rearranged and reconfigured repetitively until a suitable floor layout that satisfies the various requirements eventually emerges [1]. This iterative manual process requires a significant amount of human labour and time, and becomes ever less possible as the size and complexity of the design problem increases. Due to the iterative and repetitive nature of this problem, automated computational techniques have replaced the manual design process and become the main approach for generating floor plans [2].

Many computational algorithms including heuristic search, mixed-integer programming have been successfully developed to generate satisfactory floor plans [3]. Especially, the evolutionary methods which have dominated this field in the last decade can generate a variety of layouts. However, the adjacency constraints tackled by most of these approaches are small-in-scale, and more importantly sparse-in-density, where the number of rooms is within 10 and the number of constraints is usually equally around (or at least no more than twice) the number of rooms. For example, Camozzato et al. [4] proposed a procedural method to generate a floor plan of 8 rooms with only 1 adjacency constraint. In

[5], the authors illustrate a rectangular dissection method through an example of only 4 rooms with 3 adjacency constraints. Case study [6] tackles totally 9 adjacency constraints within 9 rooms, so the number of adjacency constraints is still no more than the number of rooms. Therefore, these approaches become inefficient with increased scale and density due to their limited scalability. For example, Rodrigues et al. [7] have applied the evolutionary methods to generate floor plans for a hotel up to 30 rooms, however the total number of adjacency constraints is only 34 and therefore still leads to a sparse adjacency matrix. Also, their case is not to generate a rectangular floor plan, therefore rooms can be placed in a more creative way with flexible boundaries. Finally, their algorithm had a runtime of 52 minutes on a 4GHz 8-core computer with multi-threading, which is not expensive when considering all kinds of granular constraints that were tackled in the original work. However, in case to address adjacency constraints only in initial floor plan, it may become not worth to apply the same approach. In addition to being limited to the small-scale and sparse-density of the adjacency constraints, this work hasn't considered the non-adjacency constraints. This paper tried to address the limitation of existing algorithms to handle high density topological adjacency and also non-adjacency constraints.

Topological adjacency constraint is one of the most important requirements during floor plan generation process, which defines the adjacency conditions between any pair of rooms. The complexity of topological adjacency constraints can be represented in terms of three factors: scale, density, and type of constraints. The first factor, the *scale of constraints* refers to the number of rooms n_{room} to place in a floor. Rooms can be any enclosed space. The larger number of the rooms we have, the larger scale of the adjacency constraints we need to tackle. The second factor, the *density of constraints* refers to the ratio between the number of constraints and the number of rooms $n_{constraints}/n_{rooms}$. For example, in residential floor plans, the adjacency constraints are often small-in-scale and sparse-in-density where there are only a limited of total rooms, and the number of constraints is roughly equal to or even less than the number of rooms. Whereas in other complex scenarios such as hotel and school planning, the problem usually have high-dimensional and dense adjacency constraints with a larger number of rooms to locate, and the constraint density may be much higher. The third factor, the *type of adjacency constraint* refers to adjacency constraints and non-adjacency constraints that need to be tackled while generating the floor layout. Adjacency constraint is very common in most types of floor plan design, which requires two rooms to be next to each other. Non-adjacency constraint which requires that two rooms must not be adjacent, though less common, is also necessary for some practical problems. For example, in a hospital floor plan, some rooms are not only required to be adjacent to other rooms for convenient circulating reasons, but also required to be non-adjacent to some other rooms for isolation and infection control.

This paper proposes an efficient and lightweight algorithm which focuses on tackling highly dense adjacency constraint matrix, and taking into consideration both the adjacency and non-adjacency constraints. It uses off-policy Monte-Carlo Tree Search (MCTS) based reinforcement learning algorithm to solve this problem. The rest of this paper is divided into four sections. Section 2 gives a brief review of the related computational approaches on floor plan layout design. Section 3 first introduces the MCTS method and the problem definition, and then presents the proposed off-policy MCTS for solving the floor plan problem with highly-dense adjacency and non-adjacency constraints. Section 4 demonstrates two practical case studies to evaluate the capabilities of the proposed algorithm. Limitation and future work are discussed in Section 5. Finally, conclusions are drawn in Section 6.

2. State of the art : Solving floor plan generation problem

Since the 1970s, researchers have developed computer-based approaches and algorithms for architecture design as detailed in the remaining part of this section. These approaches can be categorised into three main groups: conventional search methods, theoretical and mathematical proofs, and most recently the evolutionary approaches.

2.1 Conventional Search

The Conventional search methods are based on searches, enumerations and recursions by following predefined rules throughout the process. Bhasker and Sahni used a linear time algorithm to check if there are rectangular duals [8] and, if so, generates rectangular duals for any n-vertex planar triangulated graphs [9]. This is a remarkable work, however, it only applies when adjacency constraints represents a planar triangulated planar (PTP) graph. Other methods can vary from graph transformations [10], shape grammars [12] , rectangular dissections [5], placement and expansion [4] to exhaustive enumeration, heuristic search methods [14], and integer programming [16]. Veloso et al. [11] implements shape grammar into a design customization system based on Computer-Aided Architectural Design (CAAD) which includes both the algorithmic generation of designs and the detailed representation of the building. In [13], shape grammars are studied as a network structure of related designs that are visited consecutively in an exploration process. Heuristic search and integer programming are two other popular algorithms. Heuristic search highly depends on user's constraints and is often implemented differently from case to case, where single-stage approach and two-stage approach are two common heuristic search approaches used for multi-floor facility layout [15]. While on the other hand, integer programming is usually an effective method used to address geometric-dimensional constraints by solving a system of inequalities and equalities [17]. Recently, when given a rectangular floor plan layout, Upasani et al. [18] proposes an method based on linear-optimisation to adjust the geometric dimensional constraints of a given rectangular floor plan while keeping the topological adjacency relations unchanged. Most of these methods are rule-based, and can be implemented effectively. However, they are usually not scalable for large scale problem with highly-dense adjacency constraints, and more importantly, some of these algorithms may have blind cases that can never be achieved due to the limitations of the algorithm. For example, the floor plan in Figure 1 shows a floor plan design that is impossible to be generated by rectangular dissection algorithm proposed by Flemming [5].

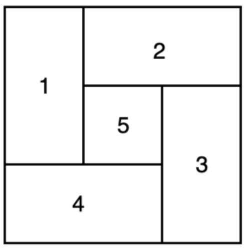

Figure 1 Blind case for rectangular dissection algorithm

2.2 Theoretical and Mathematical Proof

At the meantime, there is a second group of research works that are making remarkable contributions to provide the theoretical and mathematical proof of the layout problem. The works tried to formulate theorems on the conditions and boundedness of the solvability of layout problems to further support the computer algorithms. Koźmiński and Kinnen [19] proposed that a planar graph G has a corresponding rectangular floor plan with four rooms on the boundary if and only if every interior face is a triangle and the exterior face is a quadrangle and G has no separating triangles (a separating triangle is a triangle whose removal separates the graph). Shekhawat [20] created definitions on a generic rectangular floor plan and maximal rectangular floor plan and stated that a design solution can be identified if the target dual graph is the subgraph of the dual graph of one of the maximal rectangular floor plans. The same authors made contribution from a mathematical theory perspective on evaluating the feasibility of providing a generic floor plan solution given the topological constraints [21]. They also proposed an approach based on graph theoretic tools to produce rectangular and orthogonal floor plan where they innovatively introduce circulations in the floor plan when a desired solution does not exist for the given adjacency constraints [22]. These theorems and knowledge can be considered as guidelines to help improve efficiency when designing a computer algorithm.

2.3 Evolutionary methods

In most recent ten years with the rapid development of computational strength, Evolutionary Methods as the main group have dominated this field of automated floor plan generation. According to Kalay [23], Evolutionary Methods ''have proved their ability to generate surprisingly novel solutions'' and ''the innovative abilities of GAs (Genetic Algorithms) have been demonstrated in part through their application to art and to the generation of floor plans''. Basically, they mimic biological evolution through natural selection towards the optimal solution. By starting with an initial population of random individuals (floor plans in this case), the algorithms repeatedly modify the population by applying three main types of operators (selection, crossover, mutation) at each iteration to generate offspring layouts until a certain set of criteria are met. Many efforts have already been carried out in this field for generative floor plan design. Wong and Chan [24] proposed an Evolutionary Algorithm called EvoArch which encodes topological configuration in the adjacency matrices of the graphs and applies operators on these adjacency matrices, where the nodes of the graphs can be swapped and mutated. In [25], the authors proposed an extension to the standard genetic algorithm, which optimally groups some activities together in the first stage of the computation, and then optimally places activities within these groups at a second stage. More interestingly, Quiroz et al. [26] proposed a collaborative interactive genetic algorithm for floor planning, based both subject criteria and object criteria. Subject criteria allows the designers to make active decisions on selecting offspring from population, while object criteria corresponds with the codified user constraints.

More recently, Rodrigues et al. [27] developed a hybrid evolutionary approach involving Evolutionary Strategy (ES) and Stochastic Hill Climbing (SHC) to generate floor plans for complex requirements including highly detailed geometric and topological user constraints. Multiple evaluators had to be hand-crafted to measure the fitness of the individual (floor plan) against each kind of constraint. In every iteration, operators resulting in improved fitness of an individual will be preserved and otherwise discarded to eventually obtain a set of feasible design solutions that minimize the penalties due to not fulfilling the geometric and topological objectives. The same authors have also used this similar Evolution strategy to solve multi-level space allocation problem [7], and conducted clustering algorithms on generated floorplans based on feature vectors yielded from different shape representation methods [6]. Dino [28] applied the evolutionary approach for 3D space layout design problem: given an exact predefined 3D building boundary, the aim is to find solutions that allocate multiple 3D spaces to fully occupy the building boundary without overflow as well as satisfying other user constraints.

All these works have indicated the advance of Evolutionary Methods in the generative design of the floor plan, which outperforms previous conventional approaches mainly in two aspects: the scale of the problem and the complexity of the constraints. Firstly, Evolutionary

Methods can be suitable for larger-scale design problem up to dozens of rooms for hospital and schools. Secondly, it can handle a variety of detailed user-defined constraints including number and dimensions of rooms, connectivity/adjacency between rooms, size and orientation of interior and exterior openings, a vacant area in front of exterior openings, wall thickness.

However, Evolutionary Methods are computationally-intensive and heavy-to-implement. On one hand, since its natural selection process is highly stochastic based on conducting random operators at each iteration, the computation process is extremely intensive and expensive to achieve feasible design solutions satisfying various fine-grained constraints (dimensions/size of rooms, orientation of openings, thickness of wall, etc.). On the other hand, an evolutionary algorithm is often complex and tedious to implement. It not only involves creating a series of operators (e.g. geometric translation operator, mutation operator, alignment operator, etc), but also needs to manually handcraft dozens of metrics and evaluators to assess the fitness against to all these granular constraints and requirements. The way to combine the results from all evaluators into a single score can be somehow subjective to adjust. Therefore, in practice, the evolutionary methods can be very powerful at handling various fine-grained geometric and topological constraints simultaneously for more detailed design stages, while for early conceptual design with adjacency constraints only, it may become inefficient and even unnecessary, and therefore may not be the best approach to specially solve the problem of highly-dense adjacency constraints.

2.4 Novelty of the proposed approach

The literatures have been reviewed broadly from conventional search, mathematical theory, to evolutionary methods. Limited works are found to aim at tackling highly-dense adjacency constraints. For conventional search algorithms, the time complexity will be intractable to handle highly-dense adjacency constraints, due to its limited scalability. For evolutionary methods, it may have potentials to solve large-scale and highly-dense adjacency constraints, however it’s heavy to implement and time-expensive. Therefore, the evolutionary methods are usually more suitable for detailed floor plan design with various fine-grained constraints rather than the problem discussed in this paper with highly-dense adjacency constraints only. In addition to the dense adjacency constraints, few of the above works have considered both adjacency constraints and non-adjacency constraints.

Therefore, this paper proposes a new off-policy MCTS to tackle the high-dense adjacency constraints considering both adjacency and non-adjacency types in an efficient and lightweight manner. This idea is inspired by the most recent success of MCTS in AlphaGo [29], where the authors find the process of putting rooms within the building boundary to satisfy highly-dense adjacency constraints is similar to the process of putting stones on the game board which also depends on dense adjacency conditions.

3. Modelling floor plan generation problem using off-policy Monte-Carlo tree search based reinforcement learning

In this section, the background of traditional MCTS is firstly introduced, and then give a formal definition of the floor plan problem with highly-dense constraints of both adjacency and non-adjacency types. Finally, off-policy MCTS is proposed to solve this problem.

3.1 Monte-Carlo Tree Search (MCTS)

Reinforcement learning is a learning system that keeps updating its value function v (s) (representing the expected total rewards from a state s (or action) onwards) and policy μ (representing the probability distribution of taking actions) based on the rewards r obtained in the learning process [30]. Monte-Carlo Tree Search (MCTS) is one of the key methods of reinforcement learning, which has been widely used in finding an optimal solution in large Markov decision process. As discussed in details below, floor plan design can be formulated as a Markov decision process in a way that rooms are being placed one after another within the boundary. MCTS is also very popular for playing board game, especially the games of Go [31], where AlphaGo is the most well-known example combining deep neural networks with MCTS to make promising prediction on the next move. Here, we introduce the basics of MCTS and then in Section 3.3 describe how to innovatively adjust and adapt it into a floor plan design.

At a high level, MCTS is fundamentally a Markov decision process (MDP). The aim is trying to maximize the total rewards that could be obtained during this process, which is achieved by making promising decision or actions at each time step during the process. A search tree can be used to represent the decision-making process that at each time step the agent are located at a node (i.e. state) s, and have a set of available actions to choose which take the agent towards the children nodes/states in the next time step. In this search tree, each node s has a set of statistics,

$$\{N(s), W(s), v(s)\}$$

where $N(s)$ is the visit count of state s, $W(s)$ is the accumulated total rewards of all times, and $v(s)$ is the value function which is the expected total reward.

Specifically, at each time step, the algorithm proceeds by iterating over multiple simulations from the current state, and then taking a real action. Each simulation contains four phases: selection, expansion, roll-out and backup, as shown in Figure 2.

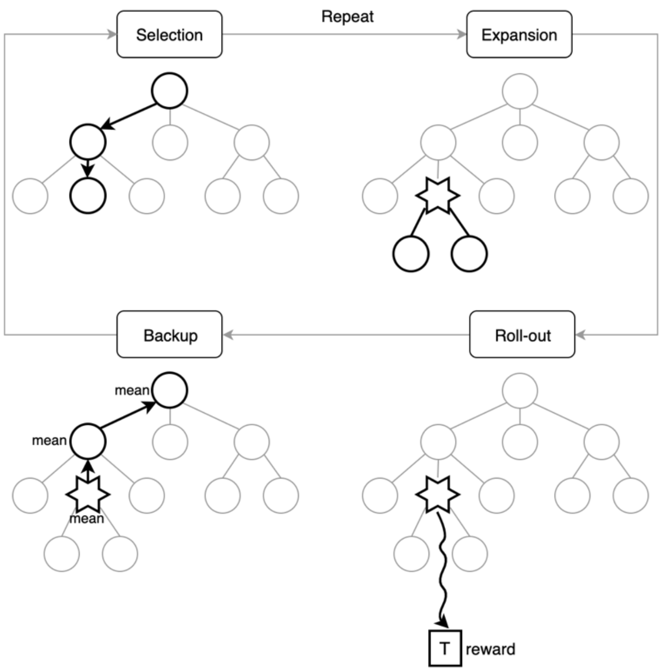

Figure 2 Four phases of simulation stage in MCTS

Basically in each simulation, the algorithm firstly selects a path from the root to a leaf node within the current tree. Then the leaf node is expanded to include its children in the tree structure, and a random roll-out is performed starting from this leaf node until reaching a terminal state. Finally, a reward obtained by evaluating against this terminal state is backed up from the expanded leaf node back to the root node.

1). Selection starts from the current state s_t (root node) to recursively choose a child based on a behaviour policy μ until a leaf node is reached. UCT [32] is one of the most popular algorithms balancing exploitation and exploration. It selects the child s_{t+1} such that:

$$s_{t+1} = \underset{s \epsilon \boldsymbol{S}_{t+1}}{\operatorname{argmax}} \left(v(s) + U(s) \right)$$

$$U(s) = \frac{\sqrt{\sum_{s' \epsilon \boldsymbol{S}_{t+1}} N(s')}}{N(s) + 1}$$

where s_t is the state of the node at time step t, $\boldsymbol{S}_{t+1}$ is the state space at time step $t+1$, i.e. all children of s_t, $v(s)$ means the value of state s, and $N(s)$ is the visit count of state s.

2). Then the leaf node is expanded and its children are added in the tree structure.

3). A roll-out is randomly conducted from the expanded node until a terminal state to obtain the reward r.

4). The reward is backup from the expanded node back to the root node s_t. The visit counts are increased, $N(s) = N(s) + 1$, and the state value is updated to the mean value: $W(s) = W(s) + r$, $v(s) = \frac{W(s)}{N(s)}$.

Each simulation consists of these four phases. After N simulations are completed from the current state s_t, a real action/decision is conducted towards its child with the highest state value $s_{t+1} = \underset{s \epsilon \boldsymbol{S}_{t+1}}{\operatorname{argmax}}\, v(s)$, and this child node becomes the new root node for the next time step. Again, in the next time step, N simulations are carried out from this new root node, and then a real action is taken, and so forth. It ends at a time step when the real action reaches the terminal state, which it's called as a real play is completed.

3.2 Formalisation of floor plan generation problem

The focus of this paper is laying out the rectangular floor plan to satisfy user-defined high-dense adjacency and non-adjacency constraints at the early design stage. The rectangular floor plan is a layout where the building boundary is rectangular and every space/room in the building boundary (including common area such as corridor) should also be rectangle-shaped [20]. Figure 1 can be an example of a rectangular floor plan.

Formally, the goal is to develop an algorithm f which takes a set of user-defined adjacency constraints C as input and gives a rectangular floor plan solution RFP as output satisfying the constraints.

$$f{:}C{\rightarrow}RFP \tag{1}$$

In the problem discussed in this paper, the constraints C can usually be formulated as a dense matrix as shown in Eq.(2), where the heads of row and column stand for room ids. The value "1" stands for adjacency constraint indicating that two rooms must be adjacent, while value "-1" means non-adjacency constraint requiring the two rooms must NOT be next to each other, and value 0 simply means no specific constraint between the two rooms. Usually only the elements at the upper-right side of the diagonal line are valid for defining constraints while the rest part of the matrix is discarded and default to 0.

$$C = \begin{array}{cc} \backslash & \begin{array}{ccccc} \mathrm{room1} & \mathrm{room2} & \mathrm{room3} & \mathrm{room4} & \mathrm{room5} \end{array} \\ \begin{array}{c} \mathrm{room1} \\ \mathrm{room2} \\ \mathrm{room3} \\ \mathrm{room4} \\ \mathrm{room5} \end{array} & \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & -1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array} \tag{2}$$

For above user constraint matrix C shown in Eq.(2), one feasible solution RFP could be the rectangular floor plan shown in Figure 1, where every constraint indicated by the upper-right side of the diagonal line of the matrix is satisfied.

3.3. Off policy MCTS based reinforcement learning algorithm for floor plan generation

This paper proposes an off-policy MCTS based reinforcement learning algorithm to solve the above-defined rectangular floor plan problem with the highly-dense adjacency and non-adjacency constraint matrix. At a very high level, like the traditional MCTS described in Section 3.1, in each time step, the proposed algorithm conducts multiple (N) simulations, and then takes real action to the next best state. In each simulation as well as the real play, each room is placed one after another in sequence from the most top-left corner to the bottom-right corner within the building boundary until all rooms have been placed. Here, “top” is defined to have higher priority than “left”, which means we first look at the available points at the top-most location, and then choose the left-most one from these points. As shown in Figure 3, *room2*, *room3*, *room5*, *room1* and *room4* are placed in sequence, which can be a possible simulation result for the problem defined in Eq.(2). The simulation result is then evaluated against the user-input constraints matrix to produce a reward r measuring the fitness which is backup to the root of this time step. After multiple simulations, the best next action is conducted in real play for this time step, and then next time step starts. The process proceeds until all rooms have been placed in real play.

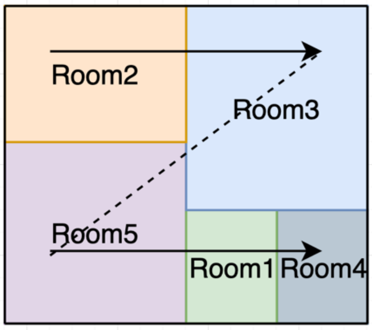

Figure 3 Rooms placed from top-right to bottom-left of our algorithms.

3.3.1 Off-policy MCTS

Although the overall architecture of the proposed Off-policy MCTS is like the traditional MCTS, there are three key differences in the proposed algorithm. The first two differences are in the simulation process as shown in Figure 4.

The first difference is that we discard the rollout phase, and instead always expand to the terminal state at the expansion phase in each simulation. Although this makes proposed algorithm more memory-intensive, however, it can improve the efficiency of repetitively traversing the tree and the accuracy of the state value $v(s)$ by recording the simulation results of all times for every visited node.

Secondly and most importantly, instead of traditional on-policy Monte-Carlo simulation to learn the value function of the behaviour policy μ, this paper proposes off-policy schema to directly learn the value function of optimal policy π. This is because the floor planning problem has a deterministic environment which is different from the uncertain environment in two-player games. In two-player games, the first player doesn't know the next state after taking an action because opponents move is unpredictable, in which case there is a need to update the value function towards the mean of total rewards in backup phase in order to handle the uncertainty of the other player which is the environment. However, in floor planning, the environment is deterministic which means the agent always knows the next state if the agent decides which action to take. Therefore, we can evaluate the optimal policy by simply updating the state value function to the max value of the total rewards in history during the backup phase,

$$v(s) = \max\{r | r \in \mathcal{R}_s\} \tag{3}$$

where $\mathcal{R}_s$ is the set of total rewards obtained in all the simulations that have visited node s. Practically in programming, the state value will only need to be updated if the backup reward r is larger than the currently stored state value: $v(s) \leftarrow r \text{ } only \text{ } if \text{ } r > v(s)$.

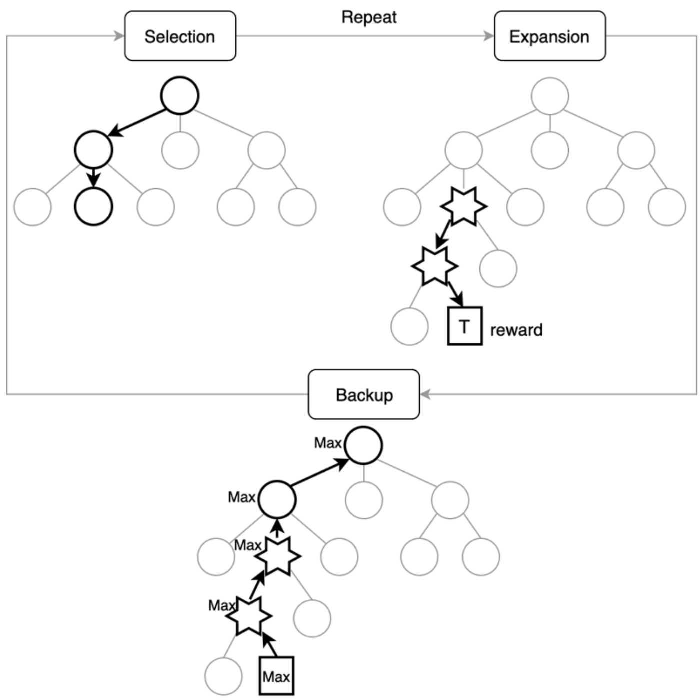

Figure 4 The simulation stage of proposed off-policy MCTS

Finally, differing from the traditional MCTS usually used in real-time two-player games which are not allowed to be restarted and replayed, proposed algorithm for floor planning design can restart if the final real solution does not fully satisfy the user's requirements. However, instead of restarting from a brand-new search tree, we reuse the previous search tree and restart the new real play from the tree's root node at the very beginning, in which way the stored statistics of the search tree will be repeatedly utilised and become richer and richer until the algorithm finally reaches an optimal solution satisfying all the user constraints. The pseudo-code of the whole algorithm is presented in Table 1.

Table 1 Proposed Off-Policy MCTS algorithm

```
Initialise root node α
Initialise number of simulations per time step: N
Count iteration of replay: M = 0
Set current node ρ←α
While True:
    While ρ is not terminal:
        For n = 1,N do:
            Run simulation from ρ
        End For
        Take real action to next time step: ρ← best_child(ρ)
    End While

    If ρ satisfies all user constraints:
        Break
    Else:
        Restart and reuse the search tree: ρ←α
        M←M + 1
End While
```

3.3.2 State and Action

For this floor planning problem, at a high level, we put each room in sequence from top-left corner to right-bottom corner. To allocate each room, we define three successive steps: the first step is to select the x coordinate of this room, the second step is to select the y coordinate of the room, and the third step is to choose which room to put into this (x,y) space. This process is illustrated in Figure 5.

Therefore, we define three types of state (node) in the search tree, namely O, X, and Y, where different types of states have different kinds of actions. The O state is at the time when a room has just been placed and the next step/action to take is to choose the x-position of the next room. Then, the X state is when the x-position of space has been determined and the action at this state is to choose the y-position of this space. The Y state is at the time when the y-position of space has been determined, and with the previously determined x-position of this space, the next action is to choose which room/id in the remaining rooms to place into this $[x,y]$ space. The flow of the states and actions can be shown in Figure 5, where we always stick to the top-left corner of the remaining empty space to place the next room.

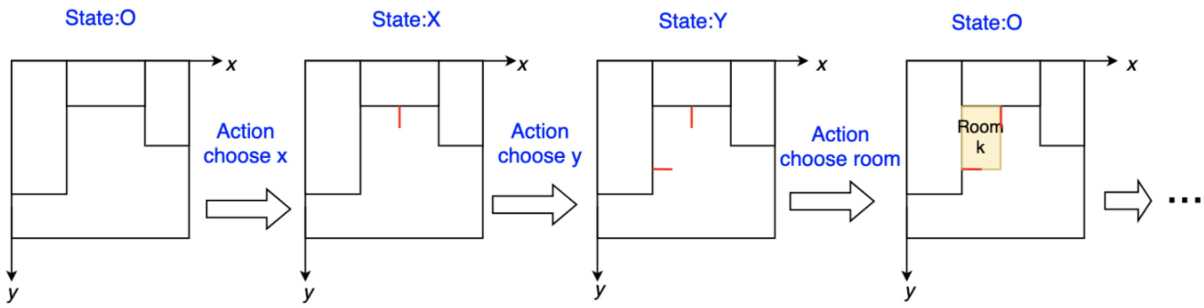

Figure 5 Illustration for the sequential actions and states of proposed algorithm

Specifically, the action space of O state depends on the number of intervals at the top horizontal line as shown in Figure 6. For each horizontal interval, there two available x-positions at the half and end of the interval. The goal is to use the least number of actions while covering all possibilities of topological conditions. For example in Figure 6, there are two horizontal intervals $[x_0,x_2]$ and $[x_2,x_4]$, with four available actions to choose for x-positions $\{x_1,x_2,x_3,x_4\}$.

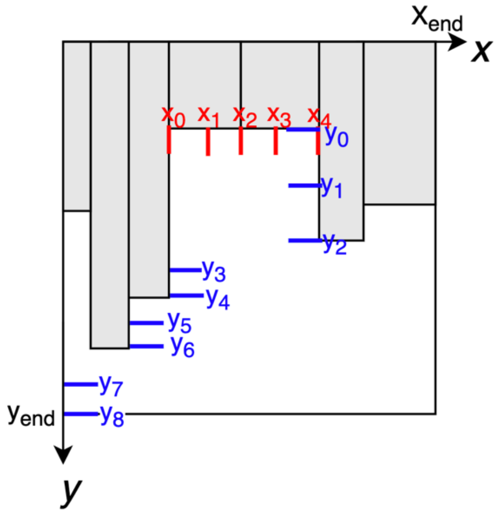

Figure 6 Example of the action space of x and y positions for a state in the proposed algorithm

Similarly, the action space of X state is to choose y-positions which depends on intervals formed by the adjacent right and left vertical lines. In Figure 6, there are four intervals: $[y_0,y_2]$, $[y_2,y_4]$, $[y_4,y_6]$ and $[y_6,y_8]$. For each interval, we choose actions located at the halfway and end positions of the interval to cover all topological possibilities. Therefore, in this case, there are eight actions to choose: $\{y_1,y_2,y_3,\ldots,y_8\}$. Only one exception here is that if in a case the x-position is selected at x_{end}, the immediate next action to select y-position should exclude y_{end}. This is to reserve available space for remaining rooms which haven't been placed yet.

The action space for the Y state is much simpler. It is to choose which room to put into the just selected $[x,y]$ space. The number of the actions in this case is the number of remaining rooms that haven't yet been placed.

Finally, after all the rooms have been placed, we first conduct horizontal expansion and then vertical expansion to fill the empty space and yield the rectangular floor plan RFP, as shown in Figure 7.

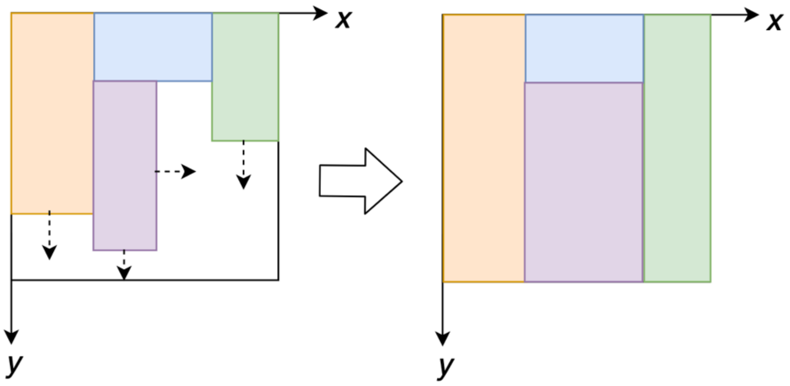

Figure 7 Expanding rooms to fulfil the building boundary after all rooms having been placed

3.3.3 Reward

Recalling the previous paragraphs, there is a need to generate a reward at the end of each simulation by evaluating the fitness of the result solution RFP against the user-defined constraints C. To do this, we will first compute the adjacency matrix M_{RFP} of the RFP solution,

$$M_{RFP} = \begin{array}{c} \backslash \\ \text{room\_1} \\ \vdots \\ \text{room\_n} \end{array} \begin{array}{c} \begin{array}{ccc} \text{room\_1} & \dots & \text{room\_n} \end{array} \\ \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{bmatrix} \end{array} \tag{4}$$

where a_{ij} is $+1$ if room_i and room_j are adjacent to each other, and -1 otherwise. There is no 0 entries in this adjacency matrix M_{RFP} of the design solution. Then the reward can be calculated and normalized through:

$$r = \frac{c_a - c_b}{c_a + c_b} \tag{5}$$

$$where \quad \begin{array}{l} (c_a - c_b) = \sum (M_{RFP} \circ C) \\ (c_a + c_b) = nonezero(C) \end{array} \tag{6}$$

where C is the user-defined constraint matrix, c_a is the number of satisfied constraints in the solution M_{RFP}, and c_b is the number of unsatisfied constraints in the solution. Thus, the reward r ranges between [-1.0, 1.0] where 1.0 means all the user-defined constraints have been satisfied by the planning solution, and -1.0 means none has been satisfied. To get numerator $(c_a - c_b)$, we first compute element-wise product between the adjacency matrix M_{RFP} of the solution and the user constraint matrix C, and then sum all the elements of the product result. For denominator $(c_a + c_b)$, we simply count the number of nonzero elements in the constraint matrix C which is the total number of user-defined constraints.

4. Evaluation

The proposed algorithm is evaluated from two perspectives: time efficiency, and capability. The first case study aims to evaluate the time efficiency of the proposed algorithm in solving adjacency constraints. The proposed algorithm is compared with the Evolution Strategy by using the floor plan problem proposed in [27]. In the second case study, the aim is to validate the capability of the proposed algorithm for solving the problem with highly-dense adjacency constraints, where the proposed algorithm is evaluated against a large dual-graph based floor plan problem which is most recently addressed in [10] through complicated graph transformations.

In both studies, the effort was made to make the problem more complex by including additional non-adjacency constraints to test the ability of the proposed algorithm in tackling both adjacency and non-adjacency constraints simultaneously. In all scenarios, we also make a comparison between proposed off-policy MCTS and the traditional on-policy MCTS.

4.1 Time efficiency

In this test, the proposed algorithm is compared with Evolution strategy and also traditional on-policy MCTS on the same floor plan problem proposed in [27]. In the original problem, there are totally 9 rooms to allocate with 11 adjacency constraints as represented in the constraint matrix C_1, where the density of constraints is ${}^{11}\!/_{9} = 1.222$ which is not very high.

$$C_1 = \begin{array}{cc} \backslash & \begin{array}{ccccccccc} \mathrm{room1} & \mathrm{room2} & \mathrm{room3} & \mathrm{room4} & \mathrm{room5} & \mathrm{room6} & \mathrm{room7} & \mathrm{room8} & \mathrm{room9} \end{array} \\ \begin{array}{c} \mathrm{room1} \\ \mathrm{room2} \\ \mathrm{room3} \\ \mathrm{room4} \\ \mathrm{room5} \\ \mathrm{room6} \\ \mathrm{room7} \\ \mathrm{room8} \\ \mathrm{room9} \end{array} & \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

The proposed algorithm runs on a single-thread, and only takes 5.2 seconds to get the optimal solution satisfying all the constraints. The result in Figure 8 shows the sequence of the nine rooms placed by the proposed algorithm one after another. The order of the rooms placed in the process is: 8, 5, 7, 4, 6, 1, 2, 9 and finally 3. The resulting score 1.0 means the final reward r which indicates that all the user-defined constraints have been satisfied.

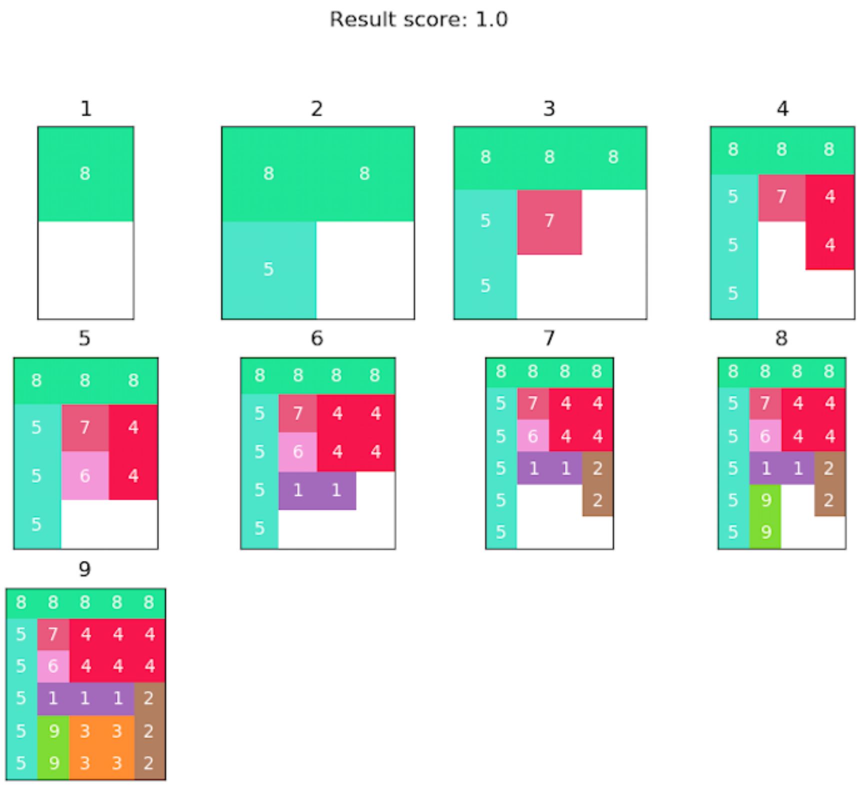

Figure 8 Result of the proposed algorithm for the planning process of the first case

Additionally, to make the problem more complex with non-adjacency constraints, we add additional non-adjacency constraints in the above original constraint matrix. For example, we want room1 to be only adjacent with room 2, 3, 4, 5 and not adjacent with any other rooms, so we can specify “-1” for the elements between room1 and room6, 7, 8, 9. We determine the non-adjacency constraints in a way that none of the solutions in original work [27] satisfies. This is to verify if the proposed algorithm can discover any solution with adjacency relations different from the original work. Thus, 21 additional non-adjacency constraints are insert into the original matrix C_1, which results in a highly-dense constraints matrix C_1^{non} with 9 rooms and 32 constraints (including 11 adjacency and 21 non-adjacency constraints) leading to a very high constraint density of ${}^{32}/_{9} = 3.556$.

$$C_1^{non} = \begin{array}{c|ccccccccc} \backslash & \text{room1} & \text{room2} & \text{room3} & \text{room4} & \text{room5} & \text{room6} & \text{room7} & \text{room8} & \text{room9} \\ \hline \text{room1} & 0 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ \text{room2} & 0 & 0 & -1 & 0 & -1 & -1 & -1 & -1 & -1 \\ \text{room3} & 0 & 0 & 0 & -1 & 0 & -1 & -1 & -1 & 1 \\ \text{room4} & 0 & 0 & 0 & 0 & -1 & 1 & -1 & -1 & -1 \\ \text{room5} & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ \text{room6} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 \\ \text{room7} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \\ \text{room8} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \text{room9} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

In this case, the proposed algorithm takes 13.8 seconds on a single-thread to get the optimal solution for C_1^{non}. The solution is shown in Figure 9. The result score/reward is 1.0 indicating both all the adjacency and nonadjacency constraints have been satisfied by the proposed solution. It validates that the proposed algorithm can address both types of adjacency and nonadjacency constraints.

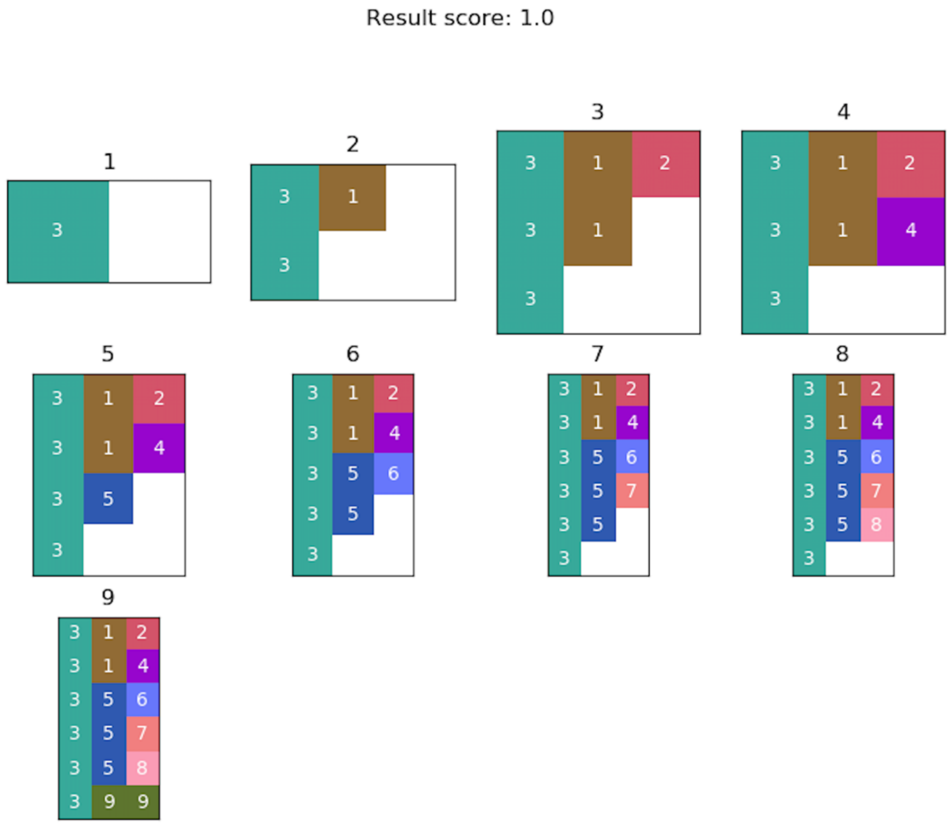

Figure 9 Solution to constraint matrix with nonadjacency constraints in the first case

Table 2 compares the time efficiency of the proposed algorithm, and traditional on-policy MCTS. We can see that the time cost of the proposed off-policy MCTS is only around

5.2 seconds for C_1 (original adjacency constraints) and 13.8 seconds for C^{non}_{1} (original adjacency and additional non-adjacency constraints) with only a single thread, while the original evolution strategy (ES) work [27] spends 2100 seconds (around 35 mins) for C_1 with two threads on dual-core. This is because the original ES work has additionally addressed more detailed geometric constraints (room size, orientation, etc). This exactly justifies as we previously mentioned that ES is more powerful and suitable for more detailed and later design stage considering diverse fine-grained constraints rather than the highly-dense adjacency constraints only discussed in this paper. In contrast, the proposed algorithm is more efficient and light-weighted for adjacency constraints only in the early conceptual design stage.

Therefore, the proposed algorithm and evolutionary methods have distinct differences regarding advantages, disadvantages and suitability for different use cases. For the proposed algorithm, the advantages are that it is more light-weight for implementation and it is very efficient to address highly-dense topological adjacency constraints. The disadvantage is that it can’t handle detailed geometric constraints. This makes it more suitable to be applied in initial floor plan at early design stage. For evolutionary methods, the advantage is that it is very powerful for addressing various constraints all together. The disadvantage is that it’s heavy to implement, and becomes unnecessary and less efficient when coming to solve adjacency constraints only. This makes it more suitable for detailed later design stage.

Table 2 Comparison the performance between the proposed algorithm, GA, and On-policy MCTS

Test ID	Algorithm	Time cost (s)	Environment	Constraints
1	the proposed off-policy MCTS	**5.2**	Single-threaded 2.3 GHz Intel one Core	C_1

2	the proposed off-policy MCTS	**13.8**	Single-threaded 2.3 GHz Intel one Core	C_1^{non}
3	Traditional On-policy MCTS	4.4	Single-threaded 2.3 GHz Intel one Core	C_1
4	Traditional On-policy MCTS	>300	Single-threaded 2.3 GHz Intel one Core	C_1^{non}

Figure 10 compares with the performance between proposed off-policy MCTS and traditional on-policy MCTS, where each point shows the reward obtained after each real play and immediately a new real play will restart by reusing the search tree until the full reward 1.0 (optimal solution) is achieved, as illustrated in proposed algorithm Table 1. For original adjacency constraints C_1 (without non-adjacency constraints), we set the hyperparameter N (number of simulations per time step) to be 250, and the results show that there is no significant difference between the proposed algorithm and the on-policy MCTS. Both can quickly achieve an optimal solution (full reward 1.0) with zero or one restart of real play. However, the proposed algorithm significantly outperforms the traditional on-policy MCTS when considering the additional nonadjacency constraints as in C_1^{non}. With N set to 1000, the proposed algorithm can still rapidly reach full reward with 13 seconds and no need to restart real play, while the traditional on-policy approach is not able to find the optimal solution for more than 300s with multiple restarts. Therefore, the proposed algorithm is more robust than the traditional on-policy MCTS in terms of the highly-dense constraints including both adjacency and non-adjacency constraints.

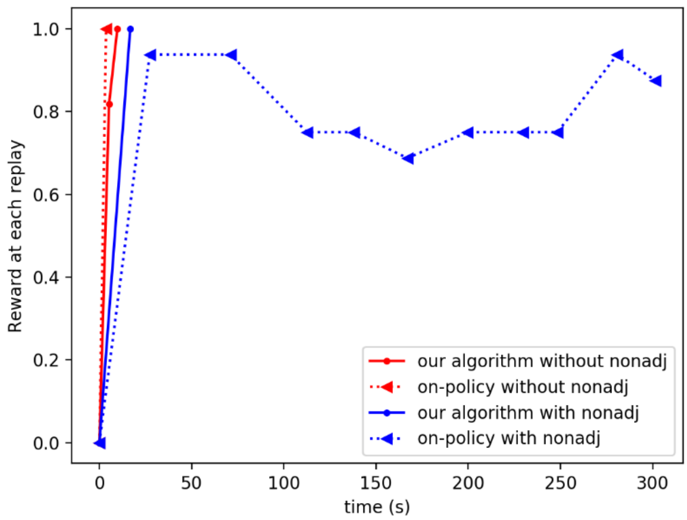

Figure 10 Comparison between the proposed algorithm and traditional on-policy MCTS

4.2 Scalability

In the second case study, in order to test the capability of the proposed algorithm for larger-scale and much higher-dense constraints, a larger-scale dual graph problem recently proposed by Wang et al. [10] was used. This problem is defined in Figure 11, which illustrates the user-defined connectivity constraints. Two nodes linked by an edge indicate that the corresponding two rooms must be adjacent in the floor plan, while two nodes that are not linked by an edge indicate the corresponding two rooms must be non-adjacent in the floor plan. The goal is to find a rectangular floor plan that satisfies both the adjacency and non-adjacency constraints defined in this dual graph. The way the original authors proposed is to first find an existing template floor plan whose dual graph is very similar to the dual graph of the original problem. In this case, the dual graph of the existing template as shown in Figure 12(a) does not contain room10. Then they apply complex graph transformation rules on this existing floor plan template to insert room10 in order to transform it to satisfy the original user-defined dual graph as shown in Figure 12(b). This method works very well and can help reuse and utilize existing floor plan for additional customized constraints. However, it requires the practitioners to obtain access to abundant existing floor plan legacy and resources.

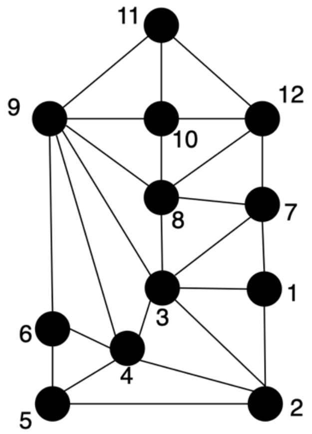

Figure 11. Dual graph of user requirement

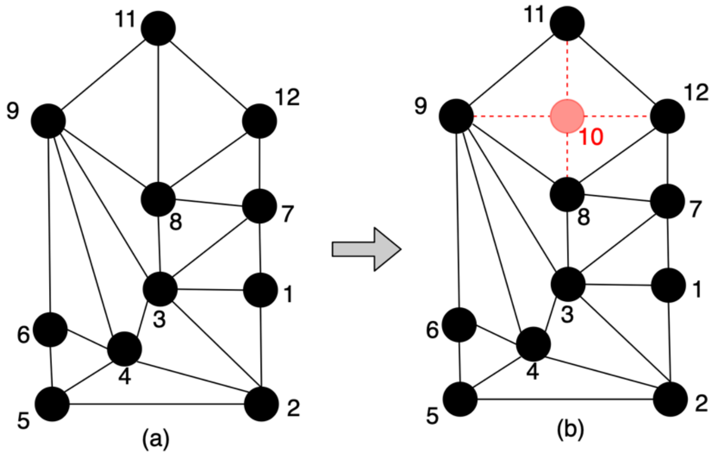

Figure 12. Graph transformation from the existing floor plan template

In this section, we apply the proposed algorithm to generate the floor plan solution simply from scratch. Firstly, we convert the original large dual graph (Figure 11) to a constraint matrix C^{non}_{2}. It contains 12 rooms, and totally 66 constraints with 25 adjacency constraints and 41 non-adjacency constraints. The density of constraints in this problem is extremely high with a density value to be ${}^{66}/_{12} = 5.5$

$$C_2^{non} = \begin{array}{c} \begin{array}{ccccccccccccc} \backslash & rm1 & rm2 & rm3 & rm4 & rm5 & rm6 & rm7 & rm8 & rm9 & rm10 & rm11 & rm12 \end{array} \\ \left[\begin{array}{ccccccccccccc} rm1 & 0 & 1 & 1 & -1 & -1 & -1 & 1 & -1 & -1 & -1 & -1 & -1 \\ rm2 & 0 & 0 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ rm3 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & 1 & 1 & -1 & -1 & -1 \\ rm4 & 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 & 1 & -1 & -1 & -1 \\ rm5 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & -1 & -1 & -1 & -1 \\ rm6 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 1 & -1 & -1 & -1 \\ rm7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & -1 & 1 \\ rm8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & -1 & 1 \\ rm9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & -1 \\ rm10 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ rm11 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ em12 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \end{array}$$

We use the same computational hardware configuration as Section 4.1: Single-threaded 2.3 GHz Intel one Core. With hyperparameter N set to 3000 and taking C_2^{non} as input, the result shows that the proposed algorithm yielded the optimal solution within 900 seconds satisfying all the 66 constraints of 12 rooms. The solution is shown in Figure 13. This validates the capability of the proposed algorithm to solve large-scale dual graph floor plan problem with extremely high-dense adjacency and non-adjacency constraints, where the density value of constraints is over 5.

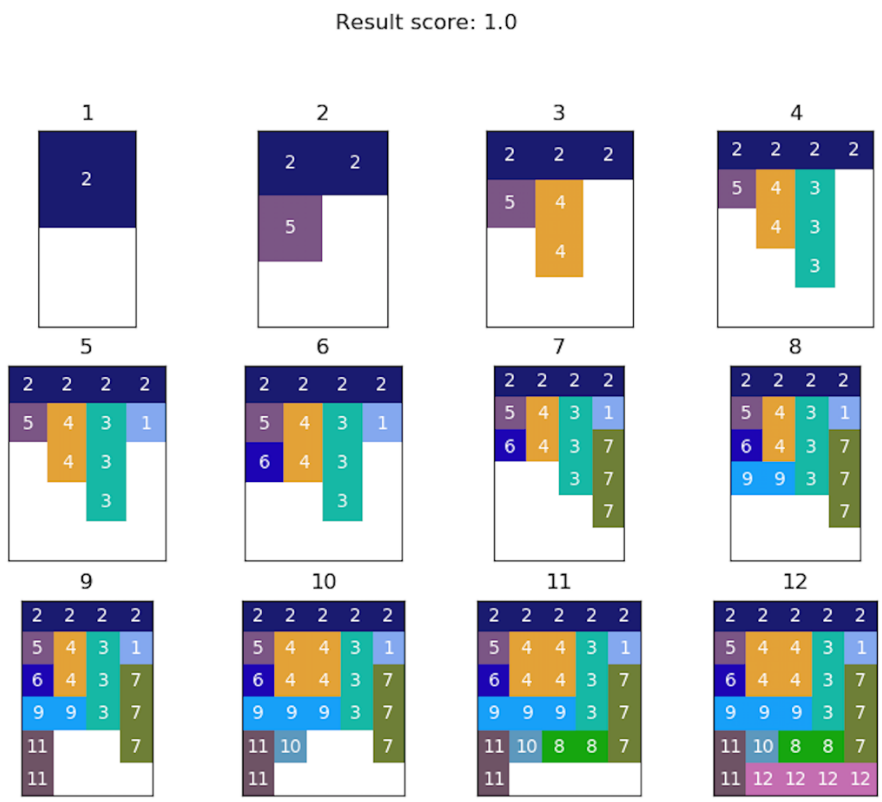

Figure 13. The solution to the original dual graph

In the above dual graph Figure 11, two nodes without an edge mean non-adjacency constraint between the two rooms. However, in some case, unlinked nodes are interpreted as “no constraints” meaning the corresponding rooms can either be adjacent or non-adjacent. For such purpose, we can simply relax the non-adjacency constraints in the original constraint matrix C^{ngn}_{2} by changing all the “-1” (non-adjacency constraints) to “0” (no constraints) which therefore generates a new constraint matrix C_2 of 12 rooms with 25 adjacency constraints as shown below. It means that we only want to guarantee that the linked nodes in the dual graph Figure 11 are still adjacent to each other in the floor plan solution while the unlinked nodes are free to either be adjacent or non-adjacent rooms in the floor plan solution. We can see that this new matrix C_2 (without non-adjacency constraints) also keeps with a high constraint density of ${}^{25}/_{12} = 2.083$.

$$C_2 = \begin{array}{cc} \backslash & \begin{array}{cccccccccccc} rm1 & rm2 & rm3 & rm4 & rm5 & rm6 & rm7 & rm8 & rm9 & rm10 & rm11 & rm12 \end{array} \\ \begin{array}{c} rm1 \\ rm2 \\ rm3 \\ rm4 \\ rm5 \\ rm6 \\ rm7 \\ rm8 \\ rm9 \\ rm10 \\ rm11 \\ em12 \end{array} & \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

With the same computational resources and hyperparameter settings, the proposed algorithm spends around 1000 seconds to get the optimal solution for C_2 as shown in Figure 14, and the corresponding dual graph of this solution is shown in Figure 15. We can see that the original dual graph (Figure 11) now becomes a subgraph of this dual graph (Figure 15) which has additional three edges highlighted in red colour.

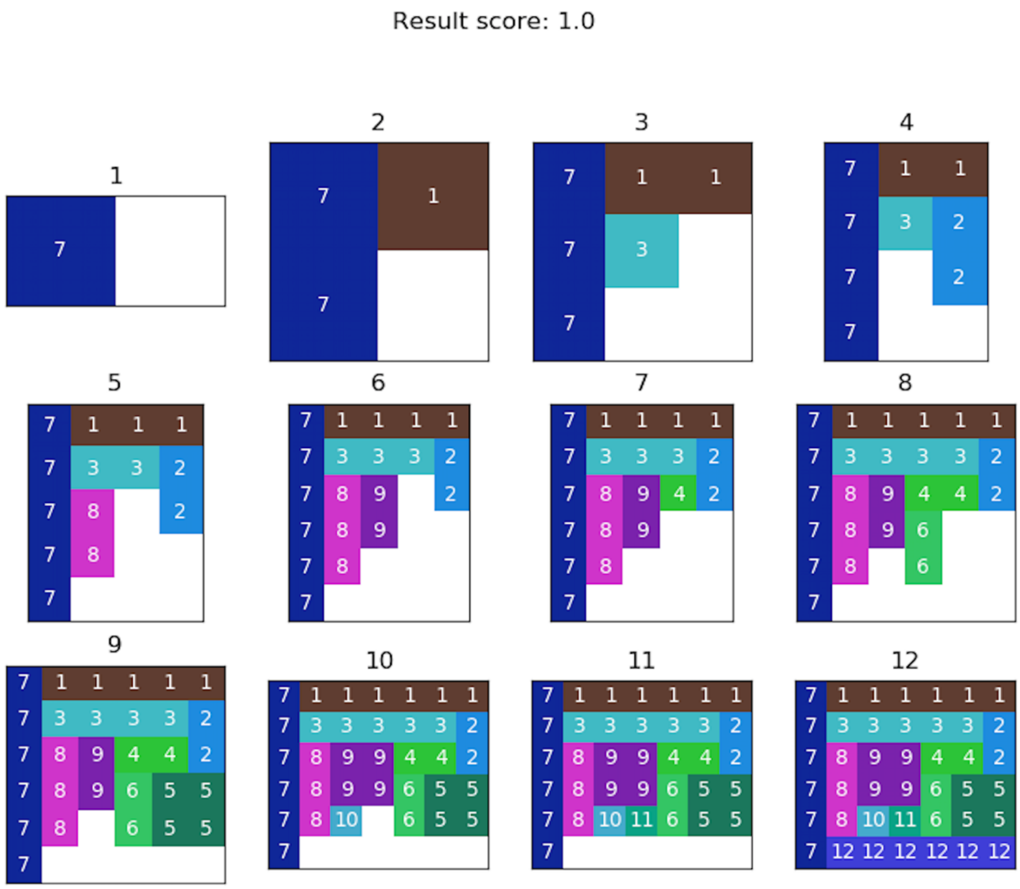

Figure 14 Solution to dual graph without nonadjacency constraints

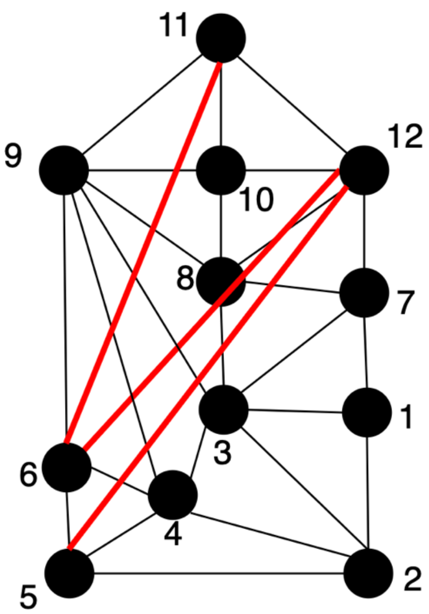

Figure 15 Corresponding dual graph for the solution to C_2

Figure 16 compares the performance between the proposed algorithm and the traditional on-policy MCTS for both the original constraint matrix C^{non}_{2} and the later relaxed constraint matrix C_2. We can see that proposed off-policy MCTS has more capacity for this kind of high-dense adjacency constraints problem. It shows the proposed proposed off-policy MCTS only conducts 3-4 replays to reach the full reward 1.0 (optimal solution) within 1000 s for both original constraints (with non-adjacency constraints) and relaxed constraints (without non-adjacency constraints). In contrast, the traditional on-policy MCTS is shown to be not able to find the optimal solution by using more than 10 replays in the first hour, where the rewards oscillated between 0.5 and 0.9 with difficulty to converge to 1.0.

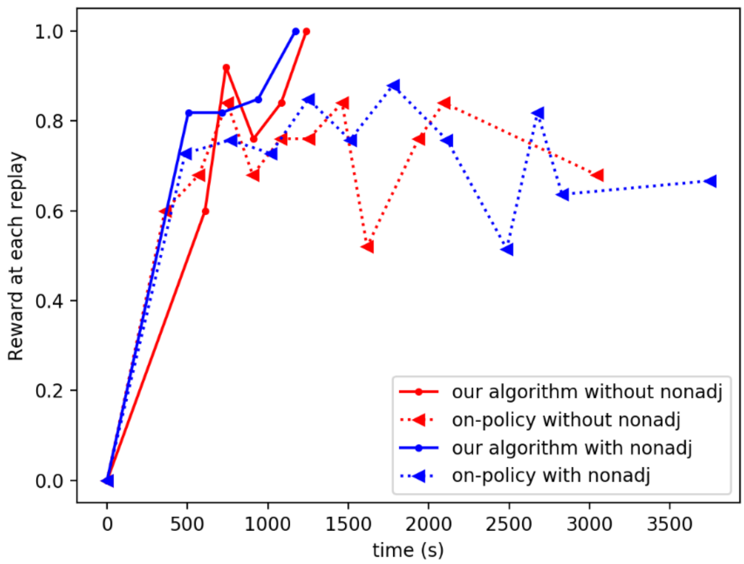

Figure 16. Performance comparison between the proposed algorithm and traditional on-policy MCTS for dense constraint matrix C^{non}_{2} and matrix C_2

5 Limitations and discussion

5.1 Orthogonal polygon boundary and Multi-story buildings

As presented above, this paper only shows how this algorithm can be applied to solve rectangular floor plan where both rooms and building boundary are in rectangular shape. However, we argue here that the proposed algorithm can also be similarly used for orthogonal polygons boundary. By following the rules in Section 3.3, the algorithm starts from most top-left point to place the next room, where “top” has higher priority than “left”, which means that when placing next room, we first look at the top-most available locations, and then choose the left-most point from these top-most locations as the spot to place next room. Therefore, the sequence of placing rooms in orthogonal polygons boundary looks like Figure 17 below. Similarly, the actions, states and rewards presented in Section 3.3.2 and Section 3.3.3 can be applied in the same way here as well. This could be a potential work of interest in the future.

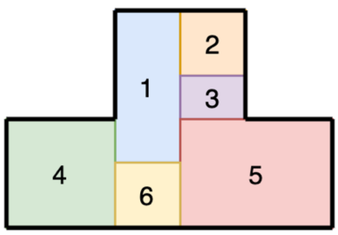

Figure 17. Potential floor plan for orthogonal polygons boundary by the proposed algorithm

It’s technically similar to apply for multi-story buildings. In floor plan for multi-story building, it has one key additional constraint that we need to care about, which is that there are common spaces such as lift/stair/bathroom that should be located at the same position across all floors. To deal with this, we can firstly start a single common MCTS to locate these common spaces since they are located at the same location across all floors, which is then followed by separate sub MCTS threads in parallel to locate the rest of rooms in each floor respectively for satisfying the corresponding adjacency constraints, as shown in the Figure 18 below. This can also be a valuable work for future efforts.

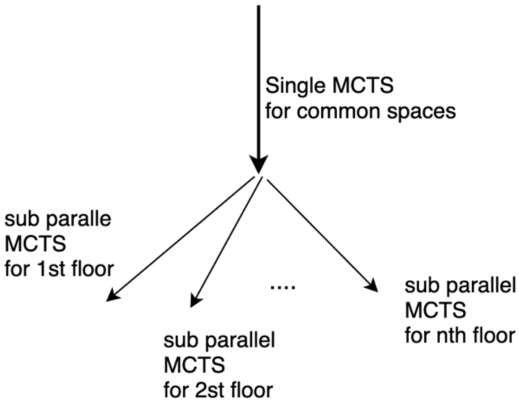

Figure 18. MCTS process for multi-story building

5.2 Integrating with linear/mathematic programming for further additional constraints

As mentioned above, the proposed algorithm generates floor plan at early design stage in an efficient and scalability manner. It provides initial floor layout which satisfy highly dense adjacency and non-adjacency constraints, however it doesn't consider other fine-grained constraints such as geometric and dimensional constraints. There is a need to integrate the proposed algorithm with other algorithms (e.g. mathematic programming) as a workflow. In this workflow, the proposed algorithm generates an initial floor layout to satisfy the adjacency relations, which is then fed into mathematic programming system to address additional fine-grained constraints.

At a high level, after the proposed algorithm generates an initial floor layout satisfying all the topological adjacency constraints, mathematic programming can be subsequently conducted on this initial layout to make further adjustments to satisfy additional geometric constraints while keeping the adjacency relationship intact. Figure 19 shows the workflow to achieve this and specific steps to integrate the proposed algorithm and mathematic programming.

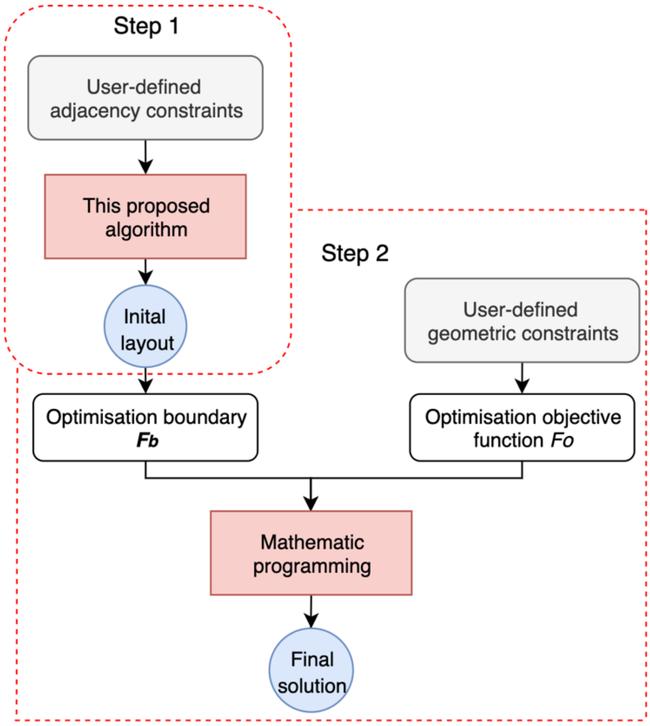

Figure 19. Workflow integrating the proposed algorithm and mathematic programming for additional geometric-dimensional constraints

In step 1, the proposed algorithm is conducted to satisfy the user-defined highly-dense adjacency and non-adjacency constraints. An initial layout is generated satisfying these user-defined adjacency constraints. This initial layout defines a set of topological relationships between rooms, which are used as the optimisation boundary in following mathematic programming process.

In step 2, mathematic programming is conducted for satisfying the user-defined geometric constraints, similar to previous work [5, 18]. In this step, we need to define both optimisation boundary and optimisation objective function, where optimisation boundary is defined according to the topological relationships, while optimisation objective function is defined according to the additional geometric-dimensional constraints we want to address in mathematic programming. The goal is to minimize the objective function within the optimisation boundary.

The optimisation boundary is defined according to the topological relationships of the initial layout generated in step 1, because we want to keep the topological relationships intact. The boundary is in form of a system of simultaneous equations or inequalities $\boldsymbol{F_b}(x_1, x_2,\ldots,x_n,y_1,y_2,\ldots,y_n)$, where x_i and y_i are the width and height of room *i* respectively. For a simple example shown in Figure 20, the optimisation boundary $\boldsymbol{F_b}$ can be represented as:

$$\begin{cases} x_1 + x_2 = W \\ x_3 + x_4 = W \\ y_1 + y_3 = H \\ y_2 + y_4 = H \\ y_1 = y_2 \\ y_3 = y_4 \\ x_1 < x_3 \\ x_4 < x_2 \\ 0 < x_{i \in [1,2,3,4]} \\ 0 < y_{i \in [1,2,3,4]} \end{cases}$$

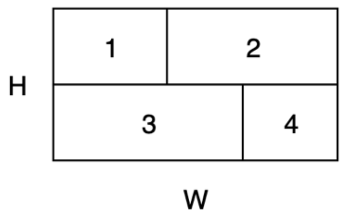

Figure 20 A simple example of initial layout yield in step 1

On the other side, the optimisation objective function is defined according to the additional geometric-dimensional constraints that we want to address in this mathematic programming, where we try to minimize the discrepancy between the initial layout and geometric constraints. For example, if the geometric constraints include:

(1) width of room1 is larger than 3.5 m,

(2) the area of room2 is bigger than 10 m^2,

(3) the height of room 3 is 4.0 m, and

(4) the width to height ratio of room4 should be smaller than 1.2,

then the optimisation objective function (subject to be minimized) can be represented as:

$$F_o = w_1(\max(0, 3.5 - x_1)) + w_2(\max(0, 10 - x_2 y_2)) + w_3|4 - y_3| + w_4\left(\max\left(0, \frac{x_4}{y_4} - 1.2\right)\right)$$

where w_i is the weight assigned to room i in order to balance the geometric compliance for each room. Please note, in case if the objective function F_o is linear, mathematic programming essentially becomes linear programming.

Once the optimisation boundary $\boldsymbol{F_b}$ and the optimisation objective function F_o are defined, mathematic programming can be conducted to find the solution minimizing the objective function within the boundary. This solution is the optimal layout that satisfies geometric constraints as much as possible while keeps the original adjacency relationships intact. Therefore, in this way, the proposed algorithm and mathematic/linear programming can be feasibly integrated into a workflow, where the proposed algorithm firstly tackles adjacency constraints, followed by mathematical programming subsequently addressing additional geometric constraints.

5.3 Further proof on existence checking

Although this paper proposed an efficiency algorithm to search for an optimal RFP solution corresponding to adjacency constraints, however, the paper hasn't proposed an efficiency way to check the existence of a RFP for a given adjacency matrix. As mentioned, [8] and [9] proposed a linear time algorithm to check if there are rectangular duals and, if so, to generate rectangular duals for any n-vertex planar triangulated graphs. But it only applies when the adjacency constraints represent a planar triangulated planar (PTP) graph. Most recently, [20] aimed at checking the existence of a RPF and constructing the RPF for any graphs that is not restricted to PTP graph. They came up with a rule-based approach which needs to enumerate all possible MRFP graphs (maximal rectangular floor plan graphs) and subsequently check if the targeted graph is a subgraph of one of the MRFP graphs. This is a remarkable contribution, while still a non-trivial approach. Therefore, there is still a need for future works to propose more efficient methods for checking the existence of RFP for any given graphs.

6. Conclusions

Inspired by the recent advanced searching and planning algorithms applied in AlphaGo, we propose a novel off-policy Monte-Carlo Tree Search to tackle the complex highly-dense adjacency and non-adjacency constrained floor plan problem in a time efficient and scalable manner. The proposed algorithm updates the state-value function to the max value of the historical total rewards it has ever seen instead of the average of the historical rewards in traditional on-policy MCTS. Two case studies are conducted to evaluate the time efficiency and scalability of the proposed algorithm respectively. The first case study shows that in terms of time efficiency, the proposed algorithm significantly outperforms Evolution strategy and traditional on-policy MCTS using two constraint matrixes with density values to be 1.222 and 3.556 respectively. The second case study further validates the capacity of the proposed algorithm by solving a large-scale dual graph problem with extremely high constraint density being more than 5.5.

The proposed algorithm extends the research in the domain on automated floor layout generation to include high-density adjacency constraints using reinforcement learning based on Off-policy MCTS. The proposed algorithm demonstrated the potential of application of Off policy MCTS algorithms to address the floor layout generation problem, in addition to the traditional methods using search-based methods, evolutionary algorithms and proofs. In particular, the proposed algorithm tackles the limitation of search and evolutionary algorithms to manage highly-dense adjacency and non-adjacency constraints during the early stage design. Although the implementation that was used in this paper is a simplification of the actual problem (with complex floor layout), the promising results from the evaluation give a grounding for further research in this area to explore more complex floor layouts by remodelling the state representation of the problem.

Acknowledgement

The authors thank the Lloyds Register Foundation for funding this research through Data-Centric Engineering Programme at the Alan Turing Institute.

Reference

[1] RODRIGUES, E., GASPAR, A. R. & GOMES, Á. 2013. An evolutionary strategy enhanced with a local search technique for the space allocation problem in architecture, Part 1:

Methodology. *Computer-Aided Design*, 45, 887-897, https://doi.org/10.1016/j.cad.2013.01.001.

[2] LIGGETT, R. S. 2000. Automated facilities layout: past, present and future. *Automation in Construction*, 9, 197-215, https://doi.org/10.1016/S0926-5805(99)00005-9.

[3] LI, H. & LOVE, P. E. D. 2000. Genetic search for solving construction site-level unequal-area facility layout problems. *Automation in Construction*, 9, 217-226, https://doi.org/10.1016/S0926-5805(99)00006-0.

[4] CAMOZZATO, D., DIHL, L., SILVEIRA, I., MARSON, F. & MUSSE, S. R. 2015. Procedural floor plan generation from building sketches. *The Visual Computer*, 31, 753-763, https://doi.org/10.1007/s00371-015-1102-2.

[5] FLEMMING, U. Representation and Generation of Rectangular Dissections. 15th Design Automation Conference, 19-21 June 1978 1978. 138-144, https://doi.org/10.1109/DAC.1978.1585160.

[6] RODRIGUES, E., SOUSA-RODRIGUES, D., TEIXEIRA DE SAMPAYO, M., GASPAR, A. R., GOMES, Á. & HENGGELER ANTUNES, C. 2017. Clustering of architectural floor plans: A comparison of shape representations. *Automation in Construction*, 80, 48-65, https://doi.org/10.1016/j.autcon.2017.03.017.

[7] RODRIGUES, E., GASPAR, A. R. & GOMES, Á. 2013. An approach to the multi-level space allocation problem in architecture using a hybrid evolutionary technique. *Automation in Construction*, 35, 482-498, https://doi.org/10.1016/j.autcon.2013.06.005.

[8] BHASKER, J. and SAHNI, S., 1987. A linear time algorithm to check for the existence of a rectangular dual of a planar triangulated graph. Networks, 17(3), pp.307-317, https://doi.org/10.1002/net.3230170306.

[9] BHASKER, J. and SAHNI, S., 1988. A linear algorithm to find a rectangular dual of a planar triangulated graph. Algorithmica, 3(1), pp.247-278, https://doi.org/10.1007/BF01762117.

[10] WANG, X.-Y., YANG, Y. & ZHANG, K. 2018. Customization and generation of floor plans based on graph transformations. *Automation in Construction*, 94, 405-416, https://doi.org/10.1016/j.autcon.2018.07.017.

[11] VELOSO, P., CELANI, G. & SCHEEREN, R. 2018. From the generation of layouts to the production of construction documents: An application in the customization of apartment plans. *Automation in Construction*, 96, 224-235, https://doi.org/10.1016/j.autcon.2018.09.013

[12] STINY, G. & GIPS, J. 1972. Shape grammars and the generative specification of painting and sculpture in: C.V. Freiman (Ed.), Information Processing 71. North Holland, Amsterdam. ISBN: 0-7204-2063-6

[13] WOODBURY, R. F. & BURROW, A. L. 2006. Whither design space? *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 20, 63-82, https://doi.org/10.1017/S0890060406060057

[14] LIGGETT, R. S. & MITCHELL, W. J. 1981. Optimal space planning in practice. *Computer-Aided Design*, 13, 277-288, https://doi.org/10.1016/0010-4485(81)90317-1

[15] MELLER, R. D. & BOZER, Y. A. 1997. Alternative approaches to solve the multi-floor facility layout problem. *Journal of Manufacturing Systems*, 16, 192-203, https://doi.org/10.1016/S0278-6125(97)88887-5.

[16] AFRAZEH, A., KEIVANI, A. & FARAHANI, L. N. 2010. A new model for dynamic multi floor facility layout problem. *Advanced Modeling and Optimization*, 12, 249-256. https://camo.ici.ro/journal/v12n2.htm

[17] GOETSCHALCKX, M. & IROHARA, T. 2007. Efficient formulations for the multi-floor facility layout problem with elevators. *Optimization Online*, 1-23. http://www.optimization-online.org/DB_HTML/2007/02/1598.html

[18] UPASANI, N., SHEKHAWAT, K. and SACHDEVA, G., 2019. Automated Generation of Dimensioned Rectangular Floorplans. arXiv preprint arXiv:1910.00081.

[19] KOŹMIŃSKI, K. & KINNEN, E. 1985. Rectangular duals of planar graphs. *Networks*, 15, 145-157, https://doi.org/10.1002/net.3230150202.

[20] SHEKHAWAT, K. 2018a. Enumerating generic rectangular floor plans. *Automation in Construction*, 92, 151-165, https://doi.org/10.1016/j.autcon.2018.03.037.

[21] SHEKHAWAT, K. and DUARTE, J.P., 2018b. Introduction to generic rectangular floor plans. AI EDAM, 32(3), pp.331-350, https://doi.org/10.1017/S0890060417000671.

[22] SHEKHAWAT, K. and DUARTE, J.P., 2019, June. A Graph Theoretical Approach for Creating Building Floor Plans. In International Conference on Computer-Aided Architectural Design Futures (pp. 3-14). Springer, Singapore. https://doi.org/10.1007/978-981-13-8410-3_1

[23] KALAY, Y. E. 2004. *Architecture's new media: Principles, theories, and methods of computer-aided design*, Cambridge, Massachusetts, MIT Press. ISBN: 978-0262112840

[24] WONG, S. S. Y. & CHAN, K. C. C. 2009. EvoArch: An evolutionary algorithm for architectural layout design. *Computer-Aided Design*, 41, 649-667, https://doi.org/10.1016/j.cad.2009.04.005.

[25] GERO, J. S. & KAZAKOV, V. A. 1998. Evolving design genes in space layout planning problems. *Artificial Intelligence in Engineering*, 12, 163-176, https://doi.org/10.1016/S0954-1810(97)00022-8.

[26] QUIROZ, J. C., LOUIS, S. J., BANERJEE, A. & DASCALU, S. M. Towards creative design using collaborative interactive genetic algorithms. 2009 IEEE Congress on Evolutionary Computation, 18-21 May 2009 2009. 1849-1856. https://doi.org/10.1109/CEC.2009.4983166

[27] RODRIGUES, E., GASPAR, A. R. & GOMES, A. 2013. An evolutionary strategy enhanced with a local search technique for the space allocation problem in architecture, Part 2: Validation and performance tests. *Computer-Aided Design*, 45, 898-910, https://doi.org/10.1016/j.cad.2013.01.003.

[28] DINO, I. G. 2016. An evolutionary approach for 3D architectural space layout design exploration. *Automation in Construction*, 69, 131-150, https://doi.org/10.1016/j.autcon.2016.05.020.

[29] SILVER, D., SCHRITTWIESER, J., SIMONYAN, K., ANTONOGLOU, I., HUANG, A., GUEZ, A., HUBERT, T., BAKER, L., LAI, M., BOLTON, A., CHEN, Y., LILLICRAP, T., HUI, F., SIFRE, L., VAN DEN DRIESSCHE, G., GRAEPEL, T. & HASSABIS, D. 2017. Mastering the game of Go without human knowledge. *Nature*, 550, 354, https://doi.org/10.1038/nature24270.

[30] SUTTON, R. S. & BARTO, A. G. 2018. *Reinforcement learning: An introduction*, MIT press. ISBN: 978-0262039246

[31] SILVER, D., HUANG, A., MADDISON, C. J., GUEZ, A., SIFRE, L., VAN DEN DRIESSCHE, G., SCHRITTWIESER, J., ANTONOGLOU, I., PANNEERSHELVAM, V., LANCTOT, M., DIELEMAN, S., GREWE, D., NHAM, J., KALCHBRENNER, N., SUTSKEVER, I., LILLICRAP, T., LEACH, M., KAVUKCUOGLU, K., GRAEPEL, T. & HASSABIS, D. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529, 484, https://doi.org/10.1038/nature16961.

[32] GELLY, S. & SILVER, D. 2007. Combining online and offline knowledge in UCT. *Proceedings of the 24th international conference on Machine learning*. Corvalis, Oregon, USA: ACM, https://doi.org/10.1145/1273496.1273531.