Design Process is a Reinforcement Learning Problem

Reza Kakooee*
Department of Architecture
ETH Zurich
Switzerland
kakooeer@ethz.ch

Benjamin Dillenburger
Department of Architecture
ETH Zurich
Switzerland
dillenburger@arch.ethz.ch

Abstract

While reinforcement learning has been used widely in research during the past few years, it found fewer real-world applications than supervised learning due to some weaknesses that the RL algorithms suffer from, such as performance degradation in transitioning from the simulator to the real world. Here, we argue the design process is a reinforcement learning problem and can potentially be a proper application for RL algorithms as it is an offline process and conventionally is done in CAD software - a sort of simulator. This creates opportunities for using RL methods and, at the same time, raises challenges. While the design processes are so diverse, here we focus on the space layout planning (SLP), frame it as an RL problem under the Markov Decision Process, and use PPO to address the layout design problem. To do so, we developed an environment named RLDesigner, to simulate the SLP. The RLDesigner is an OpenAI Gym compatible environment that can be easily customized to define a diverse range of design scenarios. We publicly share the environment to encourage both RL and architecture communities to use it for testing different RL algorithms or in their design practice. The codes are available in the following GitHub repository `https://github.com/RezaKakooee/rldesigner/tree/Second_Paper`.

1 Introduction

Reinforcement learning (RL) has been found as a proper framework to support a diverse range of applications over the past few years such as playing video games [Shao et al., 2019], controlling dynamic systems [Recht, 2019], finance [Liu et al., 2021], healthcare [Yu et al., 2021], and self-driving cars [Kiran et al., 2021]. Although these applications are so diverse, what they share in common is that they are sequential decision processes (SDP). This paper argues that the design process is a reinforcement learning problem because the design process is an SDP as well. Nonetheless, applying RL methods in the design process has grasped less attention so far. This paper explains the similarities between the design process and the RL problem to open up new opportunities for RL researchers and practitioners to use RL algorithms to support different design processes.

Design is a key step in making physical things, yet it is difficult to find a single definition for the design process. This paper follows Christopher Alexander's definition of the design process as a guideline because his rich theories have been used in various domains from architectural design to software [Alexander, 1964]. Alexander believes that every design process is involved finding a fitness between two intangible components: the form and its context. The context defines the design problem and the form offers a solution for that problem, meaning that the designer's goal is to create a form that fits various requirements of its context. The design process includes an initial form that is born in the context, and altering the form in a series of steps until reaching the desired design solution.

The reinforcement learning problem is procedurally similar to the design process. RL contains an agent that interacts with its environment to learn how to create a behavior that maximizes its long-term reward. The analogy between the RL problem and the design process can be touchable if we consider the RL agent as an artificial designer, the environment as the context, and the learned behavior as the form. Environment and context are entities that determine the boundaries of the learning and design process, respectively. Both RL and the design process are concerned with optimizing a notion of the objective function.

The real challenge in the design process is that we want to find harmony between two intangible components: a form that has not been designed yet, and a context that cannot be fully described. Similarly, in RL we need an optimal policy to maximize the reward but we have not yet learned the policy and often we do not access the environment's model that generates the reward.

Despite these similarities, the application of RL methods in the design process has not yet been well-explored. Since the design processes are so diverse, this paper focuses on a specific design task called Space Layout Planning (SLP) which can be considered as a constrained combinatorial optimization problem [Michalek et al., 2002], and attempts to frame it as an RL problem under the Markov Decision Process (MDP). We introduce a new approach for layout design and describe different components of the MDP including state-space, action-space, and reward functions. We further use Proximal Policy Optimization (PPO) [Schulman et al., 2017] which is one of the state-of-the-art deep RL algorithms to address space layout design.

Spatial layout planning concerns shaping, arranging, and dimensioning spacial elements to satisfy geometrical, topological, and performance constraints while following certain objectives [Jo and Gero, 1995]. It finds patterns of 2D subdivisions on urban, building, apartment (floor plan), and even on element (room, wall, etc.) scales. Similar geometric challenges appear in all these scales for subdividing an existing region into parts to achieve a specific set of properties, such as predefined areas, neighborhoods, proportions (aspect ratio of rooms), etc. This paper aims to address this kind of problem by strategically focusing on the floor plan scale.

In the floor plan scale, the SLP is an important part of the early design phase because it highly affects the building efficiency and end-user comfortability and determines the top-level spatial structure of a building from the early stages of the design process [Granadeiro et al., 2013]. Due to innumerable alternatives in configuration, finding feasible layout design solutions falls in the NP-complete category, for which finding an optimal solution might be impossible [Gero and Kazakov, 1998].

Framing the design process as an RL problem could provide opportunities for RL methods as the design process is an offline process so we do not need to be worried about the real-time reactions of the RL agent. Also, designers conventionally use CAD software which can be considered as an environment simulator. Thus, the RL algorithm we use will not suffer from simulation to real-world performance degradation as there is no need to run the agent outside of the simulator.

The design processes have some characteristics that challenge RL methods. For instance, for a specific design problem, there could be endless design scenarios. This is challenging because RL methods usually suffer from generalization abilities that make it difficult to transfer knowledge between different design scenarios. Moreover, for any design scenario, there could be multiple solutions that might prevent the training process from efficiently transitioning from exploration to exploitation as the agent could switch between different solutions during the training process. Nonetheless, this paper aims to put RL methods into the design practice to study its performance hoping to open up new research directions for both RL and architecture communities.

2 Related works

Several approaches have been proposed in the literature to support the automatic generation of floor plan layouts, including but not limited to shape grammars [Stiny, 1980], graph synthesis [Levin, 1964], and evolutionary strategies [Dillenburger et al., 2009]. A good overview of those can be found in these two papers [Calixto and Celani, 2015] and [Veloso et al., 2019].

Moreover, numerous attempts have emerged over the last few years to apply machine learning algorithms to architectural applications such as supervised Bayesian networks to design residential building layouts [Merrell et al., 2010], self-organizing maps for clustering floor plans [Dillenburger, 2016], interior design [RACEC et al., 2016], measuring similarities between architectural designs [Yoshimura et al., 2019], 3D architectural form style transfer [Zhang and Blasetti, 2020], topology optimization of building facades [Bernhard et al., 2020], and house style recognition [Yi et al., 2020].

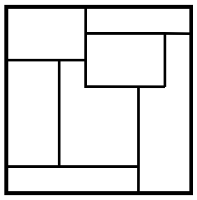

Figure 1: A sample layout consist of a few space elements

Generative Adversarial Network (GAN) [Goodfellow et al., 2014] and its different extensions have been also utilized to design floor plans, such as generating residential plans in the style of the architect Le Corbusier [Newton, 2019], generating real-time floor plans incorporated with user interaction [Chaillou, 2019] and creating floor plans for input spatial adjacency graphs [Nauata et al., 2021]. Although these approaches can synthesize space layouts according to some design constraints, particularly topological constraints, they tend to generate new samples that are similar to the training examples.

A few studies have been carried out on applying RL in architectural design [Xie et al., 2013, Ganin et al., 2018, Bapst et al., 2019, Akizuki et al., 2020], and in floor plan design. As one of the first attempts, researchers used RL to learn the best sequence of shape grammars to design floor plans [Ruiz-Montiel et al., 2013] and later extended their approach to designing small single-family dwellings that satisfy habitability and energy efficiency requirements [Mandow et al., 2020]. Their proposed method, however, highly depends on defining shape grammar rules in the early steps of the training process which is not straightforward.

The Monte-Carlo Tree Search approach was used to design floor plans under dense adjacency and non-adjacency constraints resulting in a scalable approach [Shi et al., 2020]. However, their framework is limited to only rectangular rooms and plans. Recently, multi-agent deep reinforcement learning (MARL) has been employed to synthesize spatial configuration [Veloso and Krishnamurti, 2021]. In their framework, each agent represents a specific spatial partition exploring the environment and communicating with each other to achieve certain objectives related to satisfying some geometrical and topological constraints. However, their approach relies on the assumption that the outline is not restricted to a certain shape and size.

3 The RLDesigner environment

This section introduces RLDesigner which is the environment we developed to simulate SLP.

3.1 Markov Decision Process

Reinforcement learning is a sequential decision process in which the learning happens based on the interaction between the RL agent and its environment. Such interaction is formulated under the Markov Decision Process (MDP) which contains the tuple (S, A, P, R, γ), where S represents the state of the environment, A refers to the actions available in each state, P represents the state transition probabilities, R refers to the reward function and γ is the discount factor which determines the importance of distant reward compared to immediate reward. In many RL problems, the agent does not have access to P and R. As a result, the information the agent uses to make decisions is only S and R, and likely the previous action(s).

To define the MDP components for SLP, consider Figure 1 which shows a sample layout. A space layout consists of an outline and some inlines that partition the layout to create some sub-regions. In floor plans, all lines are walls and sub-regions refer to rooms. Such a layout can be designed in two ways: occupying the layout by placing rooms and partitioning the layout by drawing walls [Veloso et al., 2019]. This papers only focus on the partitioning approach as our primary trial-and-error tests showed that it has a better exploratory ability which will be discussed later.

Traditional partitioning approaches are based on rectangular dissection shown in the first row of Figure 2. This approach has a few drawbacks: 1- unavoidable long wall starting from one side to the other side of the plan (you will have at least one long wall, see Figure 2.d), 2- only can create

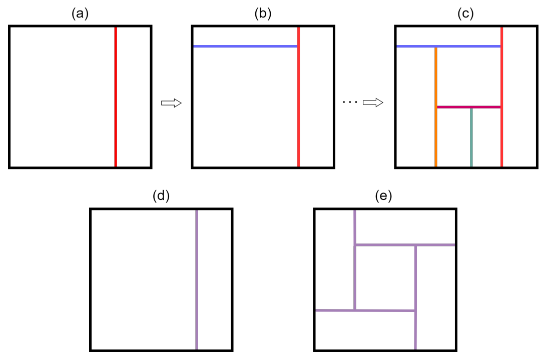

Figure 2: The first row shows the plan partitioning procedure via rectangular dissection. The second row provides two examples to visualize the shortcomings of rectangular dissection. The left plan shows an unavoidable long wall, and the right one shows a plan that is unachievable by rectangular dissection.

rectangular sub-regions and cannot create concave polygons, 3- it cannot create plans like the one shown in Figure 2.e [Shi et al., 2020].

In order to overcome these shortcomings, we propose a partitioning approach named laser-wall. Laser-wall is only a fancy name to show some of the properties of our walls. A laser-wall, which can be a straight wall or an angled-wall, consists of two parts: a hard part including two segments making the base of a wall, and a soft part including two segments that extend the base wall to create a complete wall as shown in Figure 3.b. When a base wall is placed in the plan (the thick segments in Figure 3.b), it emits the light from both segments to extend them (the thin segments in Figure 3.b) until the lights hit either a surface with a higher infiltration rate, the hard part of a wall, or the outline. A laser-wall also has a few specifications. First, the soft part of a laser-wall has a certain infiltration rate. The more the infiltration rate of the new wall is the stronger the new light is in cutting the other existing lights. Second, the light cannot infiltrate into the base walls (the hard part of the walls) and the outline. Figure 3.b shows a base wall and its lights (extensions). By this definition, each laser-wall has five important coordinates: the anchor point which is the center of the base wall, two radiation points which are the front of two segments of the base wall from which the lights radiate, and two endpoints which are the points the lights hit other walls or the outline. By this partitioning approach, we can now explain how to frame this procedure as an MDP.

The RLDesigner supports both single-agent and multi-agent environments. Here, we only focus on the single-agent setting, and more information about the multi-agent setting can be found in Appendix C. In the single-agent environment, the design starts from an empty plan and a certain number of required walls. Then, in each timestep, an RL agent places a laser-wall inside the plan to partition the plan. The planning terminates when all walls are being placed. As the single-agent environment includes static walls, we name the single-agent environment as one-shot planning.

3.2 Action-space

In the single-agent environment, the action space can be defined as selecting a base wall from the wall library (consisting of six walls as shown in Figure 3), choosing an infiltration rate (an integer number normalized in $[0, 1, ..., 9]$), and picking a coordinate in order to place the base wall. After placing the base wall, the light will be emitted automatically from the two segments to make a complete wall. Each learning episode terminates when all required walls are placed in the plan. The number of walls depends on the number of rooms we want to create. Figure 3 shows the one-shot planning. Note that the last wall (the top wall in Figure 3.f) cut the first wall (the blue one) because here we assumed that the infiltration rate of the last wall is greater than the first one.

The number of coordinates depends on the plan resolution which has a high impact on the whole framework performance, as the higher resolution, the plan has, the more accurate design solution will be achieved. Nonetheless, high resolution increases the number of actions as well which can have negative effects on the learning process. For instance, in the single-agent setting one action is

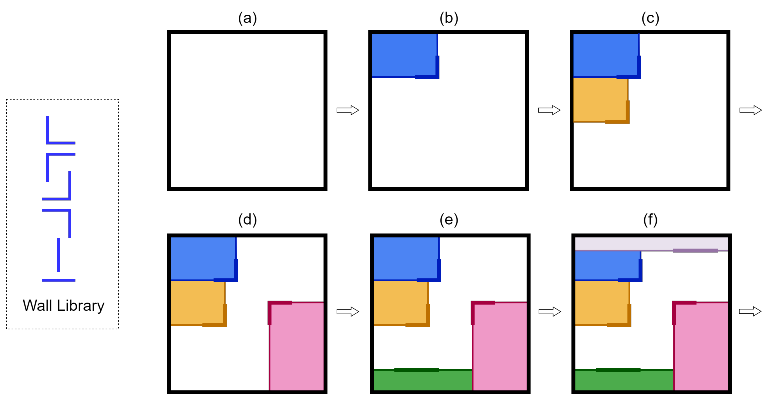

Figure 3: One-Shot Planning. Partitioning the plan with laser-wall

to select a coordinate (or cell) to place the selected base wall. Naturally, if the resolution is high, it means that there are more coordinates to choose from.

3.3 State-space

The state-space is the playground of the RL agents. The agents live in the state-space and take actions to change the state sequentially, hoping to maximize the cumulative reward. The state-space can be viewed from two perspectives: from the RL agents' perspective, who take actions according to the state to achieve the design objectives, and from the environment's perspective to evaluate the agent's performance based on the quality of the design. Thus, both agents and the environment need to perceive the state to make a decision. In the RLDesigner environment, we divide the plan into a 2D grid of discrete cells out of which the state can be defined in two ways: image representation where the state is the RGB image of the plan, and feature vector representation which includes the coordinates of each wall, the distance between five important points of each wall and the other walls, the distance between five important points of each wall and the four main corners of the outline.

3.4 Reward function

In order to define the reward function, we need to understand what a successful design (a good fit) is. The objective of a design is not just the form itself, but the ensemble of form and context [Alexander, 1964]. Alexander believes that it is more straightforward to quantify a good fit indirectly by measuring the misfit which explains how much the ultimate design deviates from the desired form. To understand it, assume that the only objective of the design is to reach out to desired areas which can be shown by A^*. The misfit can be quantified by the difference between the A^* and the area that the agent just created: $A - A^*$ (where A refers to the area the agent just created). Thus, the reward function can be defined as $-|A - A^*|$, and the goal of the agent would be to maximize the reward which automatically minimizes the misfit.

Spatial layout design can incorporate several geometrical and topological objectives such as desired area and proportion of rooms, adjacency constraints, day-light requirements, location, orientation of different space components, reachability of each room from the entrance, and so on. In the current version of the RLDesigner environment, we have only incorporated area, proportion, and adjacency. Further criteria will be added in the next versions. According to these criteria, the objective function of the design can be defined as follows:

$$\begin{aligned} \min \quad & |Adj - Adj^*| \\ \text{s.t.} \quad & A \geq A_{min} \\ & |A - A^*| \leq A_{th} \\ & 1 \leq P \leq P^* \end{aligned} \tag{1}$$

Where Adj and Adj^* refer to achieved and desired adjacencies. A, A^* and A_{min} refer to the created, desired, and the minimum acceptable areas. A_{th} is the area threshold representing the max difference between the desired and created areas that the environment non-negatively rewards. P and P^* refer to the achieved and desired aspect ratios of rooms.

The way we incorporate this objective function in the training pipeline is as follows. Consider Figure 3.b as the current state of the environment. Based on this state, the agent takes an action and creates the room shown in Figure 3.c. Now, let's assume the difference between the area the agent just created and the desired area is larger than the A_{th}. It means that the created area does not satisfy the area constraints. What we do is that we ignore the agent's action, give a negative reward to the agent, turn the state to Figure 3.b again, and let the agent to take another action. We follow a similar strategy when the aspect ratio of the created room does not meet the proportion constraints. After some time steps, the agent might create a layout that satisfies all area and proportion constraints. Then, we terminate the episode and give the agent the following reward which is based on the adjacency objective.

$$r_{global} = R - M \geq 0 \tag{2}$$

Where R is a positive, big-enough constant number, and M refers to the number of desired adjacencies that the agent missed while designing the layout. Thus, our setup consists of two types of reward, a local reward which is a binary reward (-1 when the created room does not satisfy the area and/or the proportion constraints, and 0 if it satisfies the constraints), and a global reward which evaluates the ultimate design according to the adjacency objective. We have defined more reward functions which can be found in the repository.

3.5 Exploratory capabilities of laser-walls

One of the key challenges when using the RL method is what is called exploration. Exploration conventionally refers to the learning policy's ability to explore different parts of the environment hoping to find a better sequence of actions that results in more rewards compared to the situation where the agent tends to take only those actions that have been historically proven to be good actions. This form of exploration is related to the behavioral policy of the agent. Nonetheless, exploration could have a secondary form that is related to the environment's capacity in allowing the RL agent to explore it. Imagine a standard grid-world environment, where the start point is somewhere close to the center of the grid and the goal point is on the top-right. If the agent action set does not include action *right*, no RL algorithm will be able to find the goal point even with a very high exploration rate. This phenomenon happens when the solution space is wide but the action set is limited.

Spatial layout planning is among those environments that have a large solution space, particularly when the number of spatial components increases. This becomes even worse when the layout itself has a non-convex shape and there are multiple objectives that enlarge the solution space. Thus, finding the optimal solution not only needs more exploration but also requires smart actions that enable the agent to get to the heart of the solution space in order to find the optimal one. The laser-wall concept introduced above is a smart tool in the hand of the RL designer to find a solution in a powerful way.

As described, the proposed laser-wall has some properties that enable conveniently translating the SLP to an MDP allowing us to deal with this form of planning by RL-based approaches. Laser-wall properties empower the RL agent to partition the plan into diverse architecturally acceptable configurations. It overcomes the challenges that the rectangular dissection approach is faced with, and has the ability to explore a large solution space. The hard segments of the wall ensure the stability of the wall when it is under the exposure of the other walls (as explained the light cannot permeate the hard part of the wall), and the soft part allows the other walls to change the current state of the layout configuration which ease exploring the solution space.

4 Experiment

The developed environment is customizable and flexible in generating different design scenarios. The parameters like the number of desired walls (or rooms), desired area for each room, desired aspect ratios (proportions), and desired rooms adjacency matrix can be adapted to generate new design scenarios. The outline of the plan is not always rectangular but can be easily changed to non-rectangular shapes by simply masking one or multiple parts of the plan, as shown in Figure 4.

Table 1: Six random design scenarios for testing the RLDesigner environment.

Scenario	#Rooms	Proportion status	Desired areas (m^2)	Desired adjacencies
1	4	1	[110, 92, 57, 52]	[[1, 2], [2, 3], [2, 4], [1, 3]]
2	5	1	[83, 78, 60, 44, 33]	[[1, 2], [1, 5], [3, 4]]
3	6	0	[111, 75, 38, 34, 30, 21]	[[3, 4], [1, 6], [1, 4]]
4	7	1	[65, 60, 48, 36, 30, 28, 23]	[[2, 3], [1, 5], [6, 7], [3, 4]]
5	8	0	[85, 64, 64, 50, 41, 32, 32, 28]	[[2, 8], [4, 6], [1, 7], [2, 7]]
6	9	0	[63, 60, 58, 50, 34, 33, 27, 27, 26]	[[3, 9], [8, 9], [3, 6], [7, 8], [5, 7], [1, 8]]

Masked regions are those the agent is not allowed to place any spatial component in. The size of the plan is also a changeable parameter of the environment. In this section, we define a few design scenarios to check both environment flexibility and RL algorithms' performance in reaching the design goal (desired room adjacency matrix) with respect to the design constraints (desired areas and proportions of rooms).

In the scenarios defined in this paper, the plan is always a grid of $20 * 20$ cells, and can be masked. In half of the scenarios, we did not include proportion as the constraint (in which we set $R = 200$). In the rest, the desired proportion is between 1 and $P^* = 5$ (in which we set $R = 400$). The last column in the table shows the desired adjacencies as the objective of the design process. For example, $[1, 2]$ shows that there must be a connection between *Room 1* and *Room 2*. The adjacency matrix of the final design might also include some other direct connections which are not a part of the desired adjacencies, but we do not penalize the agent for them. We only penalize the agent when it misses desired adjacencies in the ultimate design. Besides, in all scenarios $A_{min} = 10$, and $A_{th} = 4$.

4.1 RL method

Deep RL consists of various algorithms that can be categorized as model-based RL (MBRL) and model-free RL (MFRL) algorithms according to the availability of the environment's model for the agent. The former represents the prospective goal-directed mechanism in the brain, and the latter represents retrospective habitual learning. The goal-directed mechanism either has access to the environment model or can learn a cognitive map via the interaction with the environment and utilize it for planning [Kakooee et al., 2021]. As in this paper, we encounter a planning problem, and the model of the environment can be available for the agent, both MBRL and MFRL can be used. Nonetheless, we use an MFRL algorithm called Proximal Policy Optimization (PPO) [Schulman et al., 2017].

PPO is a policy gradient method that aims to update the policy parameters gradually by the data collected through interaction with the environment, in contrast to value-based methods that learn a kind of value function to obtain the policy. PPO uses a clipped surrogate objective while ensuring exploration by embedding an entropy term into the objective function. In this paper, we use *Ray-RLlib* [Liang et al., 2018] implementation of PPO, which supports single-agent and discrete action space settings. The PPO's hyperparameters are the default values of the PPO implementation of the *RLlib*; however, we have adapted the policy network. The network architecture can be found in Appendix A.

4.2 Results

The evaluation of the agent performance is based on the *episode reward mean*, and the number of actions to find the desired solution (e.g. *episode len mean*). Figure 4 shows the performance of the PPO algorithms in the six different scenarios defined in Table 1. As mentioned above the learning procedure involves maximizing the reward by arranging the room such that the desired adjacency is archived while satisfying the constraints, including the desired room areas and proportions. The first and the third rows of Figure 4 show how learning evolves, and the second and fourth rows display the layout the agent created for each scenario. The thin lines on top of the plans represent the connections between rooms, among which the green lines represent those desired adjacencies which have been successfully achieved, the red lines represent those desired adjacencies that the agent has failed to make, and the blue ones show other connections between rooms.

As the learning curve shows, the agent has learned a policy that maximizes the reward after enough interaction with the environment. Besides, the episode lengths in all scenarios approach to their minimum values during the training process. The minimum length of each scenario is equal to the number of walls or number of rooms minus one. Since the *episode reward mean* and the *episode len* *mean* curves are symmetric, we avoid plotting *episode len mean* curves, but as an example see Figure 6 in Appendix B. Since the longer the episode length is the more negative reward the agent receives, relationship between the two are symmetric. Approaching the episode lengths to their minimum values means that the agent learned to avoid taking those actions that dissatisfy the constraints.

5 Discussion and future directions

This paper showed that the design process can be considered as an RL problem because like the RL problem, the design process is a sequential decision process as well, and both aim to find a notion of fitness between two intangible components. We studied SLP as a specific design task because SLP is perceived as an important task in various design processes including architectural design practices. Inspired by the existing partitioning approaches, we proposed a novel space partitioning approach and described how an SLP can be framed as an MDP. Consequently, we developed the RLDesigner, a reinforcement learning (RL) environment for general-purpose spatial layout planning.

We also demonstrated results achieved by training PPO algorithms on some design scenarios to achieve the desired adjacency objectives while satisfying room areas and proportions constraints. The results show how useful RL is in designing the spatial layout. The major goal of the paper was to show that the design process can be formulated as an MDP and RL algorithms are able to design. Nonetheless, there are a lot of investigations that need to be done. First, while we studied the performance of the PPO agent in designing a layout, we did not compare its performance with other RL algorithms and non-RL methods like genetic algorithms, which is a widely used method in SLP, to investigate which one could be a better artificial designer for SLP. Second, for each design scenario mentioned above, we need to train the RL algorithm from scratch. Although the training is relatively fast (see Appendix D), training from scratch for every new scenario is absolutely inefficient, particularly for complex design scenarios that could consist of many spatial components and multiple constraints and objectives. Nonetheless, this paper is a part of a larger ongoing research project, in which we have the following objectives.

We already have generated a large dataset of layouts by which we aim to train an offline RL agent to learn a policy for a diverse range of design scenarios. This policy can be either directly used or be fine-tuned for new design scenarios. Since we have access to the environment, fine-tuning the agent could be possible. This strategy might help to enable transfer-learning across design scenarios and equip the learned agent with generalization capabilities. In addition, as the name suggests, SLP is a planning problem and the fact that we have access to the model of the environment allows us to compare model-based and model-free RL algorithms to check which approach can be more efficient to be used as an artificial designer in SLP.

The RLDesigner is an open-source, OpenAI Gym compatible, and customizable environment that allows human designers to adapt different components of the environment to create new design scenarios that satisfy their functional requirements. We are sharing the codes publicly to be used by both RL and architecture communities. For the RL community, it could be a reasonable choice to test both model-based and model-free RL algorithms, also the environment supports both single-agent and multi-agent settings. With simple design scenarios, it can also be useful for teaching purposes as the physics behind the SLP is simple so anyone can understand it easily. Particularly, as the environment is customizable, one can make new design scenarios for various purposes by simply changing some parameters. For the architecture community, we have provided a framework that allows coping with the family of SLP in architecture by a learning-based approach, hoping it could lead to better knowledge sharing, evaluation opportunities, and ultimately to develop more efficient layout design approaches.

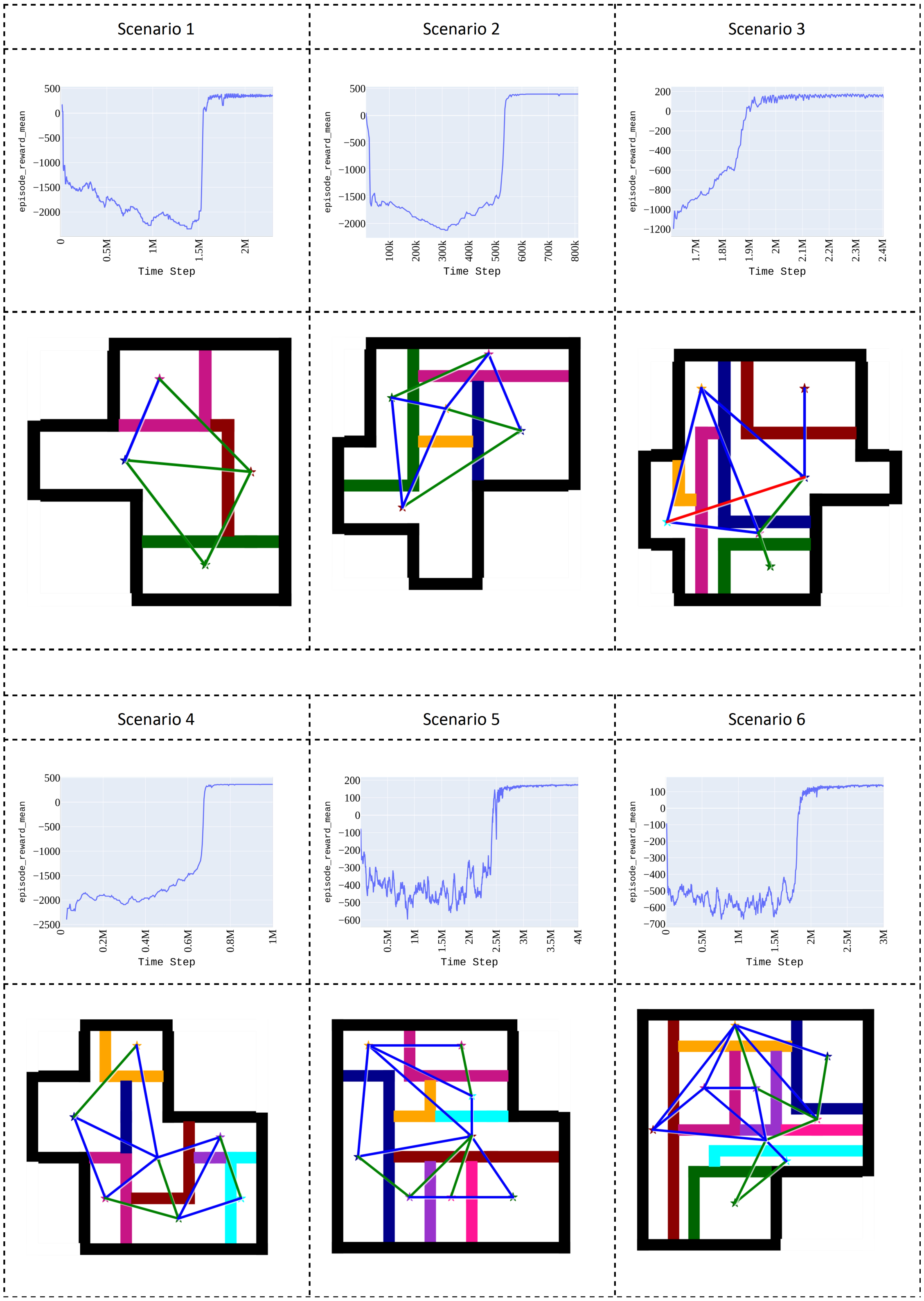

Figure 4: The learning curves of PPO and the created plans for each scenario defined in Table 1

References

Kun Shao, Zhentao Tang, Yuanheng Zhu, Nannan Li, and Dongbin Zhao. A survey of deep reinforcement learning in video games. *arXiv preprint arXiv:1912.10944*, 2019.

Benjamin Recht. A tour of reinforcement learning: The view from continuous control. *Annual Review of Control, Robotics, and Autonomous Systems*, 2:253–279, 2019.

Xiao-Yang Liu, Hongyang Yang, Jiechao Gao, and Christina Dan Wang. FinRL: Deep reinforcement learning framework to automate trading in quantitative finance. *ACM International Conference on AI in Finance (ICAIF)*, 2021.

Chao Yu, Jiming Liu, Shamim Nemati, and Guosheng Yin. Reinforcement learning in healthcare: A survey. *ACM Computing Surveys (CSUR)*, 55(1):1–36, 2021.

B Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A Al Sallab, Senthil Yogamani, and Patrick Pérez. Deep reinforcement learning for autonomous driving: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 2021.

Christopher Alexander. *Notes on the Synthesis of Form*, volume 5. Harvard University Press, 1964.

Jeremy Michalek, Ruchi Choudhary, and Panos Papalambros. Architectural layout design optimization. *Engineering optimization*, 34(5):461–484, 2002.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Jun H Jo and John S Gero. A genetic search approach to space layout planning. *Architectural Science Review*, 38(1):37–46, 1995.

Vasco Granadeiro, José P Duarte, João R Correia, and Vítor MS Leal. Building envelope shape design in early stages of the design process: Integrating architectural design systems and energy simulation. *Automation in construction*, 32:196–209, 2013.

John S Gero and Vladimir A Kazakov. Evolving design genes in space layout planning problems. *Artificial intelligence in engineering*, 12(3):163–176, 1998.

George Stiny. Introduction to shape and shape grammars. *Environment and planning B: planning and design*, 7(3):343–351, 1980.

Peter Hirsch Levin. Use of graphs to decide the optimum layout of buildings. *The Architects' Journal*, 7:809–815, 1964.

Benjamin Dillenburger, Markus Braach, and Ludger Hovestadt. Building design as individual compromise between qualities and costs: A general approach for automated building generation under permanent cost and quality control. *CUMINCAD*, 2009.

Victor Calixto and Gabriela Celani. A literature review for space planning optimization using an evolutionary algorithm approach: 1992-2014. *Proceedings of the 19th conference of the Ibero-American Society of Digital Graphics*, 2015.

Pedro Veloso, Jinmo Rhee, and Ramesh Krishnamurti. Multi-agent space planning: a literature review (2008-2017). *18th International Conference on Computer Aided Architectural Design Futures*, 2019.

Paul Merrell, Eric Schkufza, and Vladlen Koltun. Computer-generated residential building layouts. *ACM Transactions on Graphics*, pages 1–12, 2010.

Benjamin Dillenburger. *Raumindex. Ein datenbasiertes Entwurfsinstrument*. PhD thesis, ETH Zurich, 2016.

E RACEC, S BUDULAN, and A VELLIDO. Computational intelligence in architectural and interior design: a state-ofthe-art and outlook on the field. *Barcelona: Universitat Politècnica de Catalunya*, 2016.

Yuji Yoshimura, Bill Cai, Zhoutong Wang, and Carlo Ratti. Deep learning architect: Classification for architectural design through the eye of artificial intelligence. In *International Conference on Computers in Urban Planning and Urban Management*, pages 249–265. Springer, 2019.

Hang Zhang and Ezio Blasetti. 3d architectural form style transfer through machine learning. *RE: Anthropocene, Design in the Age of Humans*, 2020.

Mathias Bernhard, Reza Kakooee, Patrick Bedarf, and Benjamin Dillenburger. Topology optimization with generative adversarial networks. *AAG 2020*, 2020.

Yun Kyu Yi, Yahan Zhang, and Junyoung Myung. House style recognition using deep convolutional neural network. *Automation in Construction*, 118:103307, 2020.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

David Newton. Deep generative learning for the generation and analysis of architectural plans with small datasets. *Proceedings of ECAADE SIGRADI 2019*, 2019.

Stanislas Chaillou. Archigan: a generative stack for apartment building design. *NVIDIA Corporation*, 2019.

Nelson Nauata, Sepidehsadat Hosseini, Kai-Hung Chang, Hang Chu, Chin-Yi Cheng, and Yasutaka Furukawa. House-gan++: Generative adversarial layout refinement networks. *arXiv preprint arXiv:2103.02574*, 2021.

Ning Xie, Hirotaka Hachiya, and Masashi Sugiyama. Artist agent: A reinforcement learning approach to automatic stroke generation in oriental ink painting. *IEICE TRANSACTIONS on Information and Systems*, 96(5):1134–1144, 2013.

Yaroslav Ganin, Tejas Kulkarni, Igor Babuschkin, SM Ali Eslami, and Oriol Vinyals. Synthesizing programs for images using reinforced adversarial learning. In *International Conference on Machine Learning*, pages 1666–1675. PMLR, 2018.

Victor Bapst, Alvaro Sanchez-Gonzalez, Carl Doersch, Kimberly Stachenfeld, Pushmeet Kohli, Peter Battaglia, and Jessica Hamrick. Structured agents for physical construction. In *International Conference on Machine Learning*, pages 464–474. PMLR, 2019.

Yuta Akizuki, Mathias Bernhard, Reza Kakooee, Marirena Kladeftira, and Benjamin Dillenburger. Generative modelling with design constraints-reinforcement learning for object generation. *Generative Modelling with Design Constraints-Reinforcement Learning for Object Generation*, 2020.

Manuela Ruiz-Montiel, Javier Boned, Juan Gavilanes, Eduardo Jiménez, Lawrence Mandow, and José-Luis PéRez-De-La-Cruz. Design with shape grammars and reinforcement learning. *Advanced Engineering Informatics*, 27(2):230–245, 2013.

Lawrence Mandow, José-Luis Pérez-de-la Cruz, Ana Belén Rodríguez-Gavilán, and Manuela Ruiz-Montiel. Architectural planning with shape grammars and reinforcement learning: Habitability and energy efficiency. *Engineering Applications of Artificial Intelligence*, 96:103909, 2020.

Feng Shi, Ranjith K Soman, Ji Han, and Jennifer K Whyte. Addressing adjacency constraints in rectangular floor plans using monte-carlo tree search. *Automation in Construction*, 115:103187, 2020.

Pedro Veloso and Ramesh Krishnamurti. Self-learning agents for spatial synthesis. In *Formal Methods in Architecture*, pages 265–276. Springer, 2021.

Reza Kakooee, Mohammad Taghi Hamidi Beheshti, and Mehdi Keramati. Developing a reinforcement learning algorithm to model pavlovian approach bias on bidirectional planning. *The Neuroscience Journal of Shefaye Khatam*, 2021.

Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, and Ion Stoica. RLlib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2018.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] In Section 3 we explained how design process can be framed as an RL problem, and in Seciton 4 we showed the performance of PPO as an artificial designer.
   (b) Did you describe the limitations of your work? [Yes] See Section 5
   (c) Did you discuss any potential negative societal impacts of your work? [No] We believe our work does not have any negative societal impacts.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [No] We do not have any theoretical results
   (b) Did you include complete proofs of all theoretical results? [No] We do not have any theoretical results
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] We attached the codes as supplemental material but do not share the URL at the moment due to double-blind procedure.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] We use the default hyperparameters of the RLlib library. See Section 4 and Appendix A
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [No] We have not performed the experiments for multiple times as the core motivation of the paper is not the algorithmic efficiency but rather is to show how the design process can be a great application for RL methods. Thus, the experimental results only introduce proof-of-concept.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] See Appendix D
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] We use RLlib library and cited it. See Section 4
   (b) Did you mention the license of the assets? [Yes] The LICENCE can be found in Ray-RLlib GitHub repository.
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes] We are publicly share the codes of the environemnt we have developed and the RL training pipline which we implemented by using RLlib.
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [No] We do not use any existing datasets.
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [No] We do not use any dataset.
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [No] Not applicable.
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [No] Not applicable.
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [No] Not applicable.

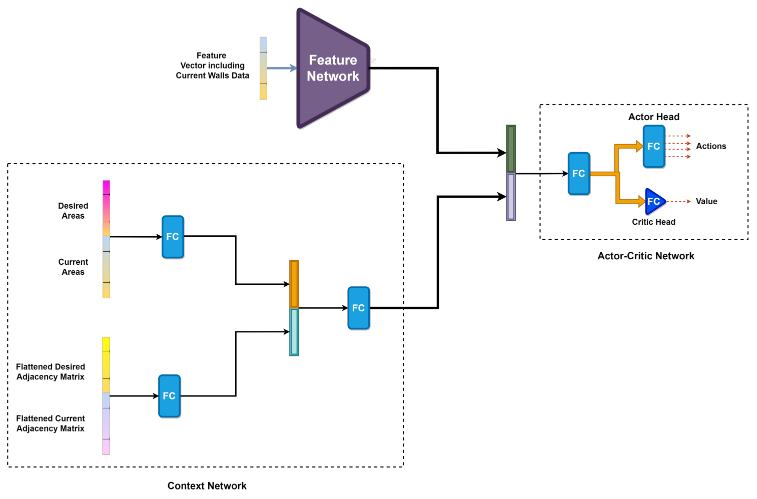

Figure 5: The policy network consisting of three sub-networks: feature, context, and actor-critic networks

Appendix

A Policy network architecture

Our policy network consists of three sub-networks shown in Figure 5. The first one is a feature network that encodes the current layout into the latent space. It consists of convolutional and linear layers if the state-space is the RGB image of the plan and only linear layers otherwise.

The second one is a context network whose goal is to encode the design constraints and objectives. As mentioned earlier, there could be endless design scenarios. Thus, we need to inform the agent for which design scenarios we want to find a solution. To do so, we feed the network with both desired areas and adjacencies, and the areas and adjacencies of the current state of the environment. We path them through multiple linear layers and concatenate their outputs with the output of the feature network. We do not incorporate the proportion constraints into the network because proportion constraint is always fixed and does not change across scenarios. Thus, we only check the proportion constraints when evaluating the agent's action. Finally, the third network is the actor-critic model that tries to predict the action based on the embeddings encoded by the first two sub-networks.

The context network can be ignored if the goal is to train the agent from scratch for each design scenario. In this situation, the agent explores the environment and finds the goal state by chance. Then, the agent tries to exploit the learned trajectory while keeping the exploration alive. A context network will be necessary when the task is to train the agent for multiple design scenarios at once or transfer knowledge across different design scenarios.

B Supplementary figures

Figure 6 shows how the episode length in *Scenario 1* approaches to its minimum value which is 3 as in this scenario we only have four rooms, so we can design the plan by three laser-walls.

C Multi-agent

In the multi-agent environment, each wall represents an agent, and each agent is responsible for creating a room. For example, the kitchen agent is responsible for creating a kitchen with some properties like a $20m^2$ area. The design starts with an initial plan including some randomly placed laser-walls. Then, in each timestep, an agent takes an action (discussed below) in order to change the plan hoping to make it better with respect to the design constraints and objectives. Thus, the multi-agent environment consists of dynamic laser-walls which can be transformed within the layout.

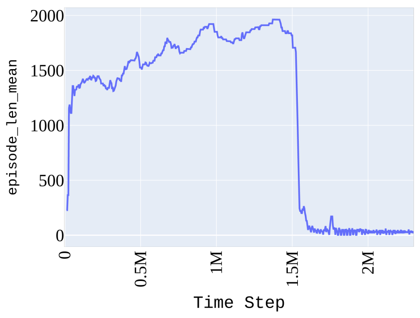

Figure 6: The *episode len mean* curve for *Scenario 1*.

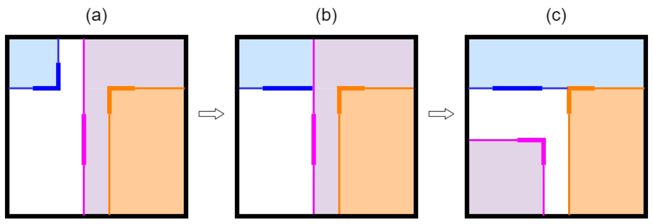

Figure 7: Multi-Agent Designing. (a): The layout design begins with three walls placed in random coordinates with this order: brown, magenta, and blue. Each agent owns the smaller region that it creates. (b): The blue agent rotated its right segment through 90° *CW*. (c): The magenta agent rotated its left segment 90° *CCW*.

The multi-agent planning terminates when all design constraints are met and the objectives are satisfied.

C.1 Action-space

In the multi-agent setting, the planning begins with some laser-walls randomly placed within the plan in a predefined order. The learning process includes transforming the base walls in order to reach the design objectives. Each agent can take one of the following actions in a timestep: rotating the base wall through $\mp 90°$, rotating only one segment of the base wall through $\mp 90°$, moving the base wall one cell to left, right, up, and down, flipping the base wall horizontally and vertically, and choosing an infiltration rate. Figure 7 shows how multi-agent planning is executed when the infiltration rate of all walls is the same.

C.2 State-space

State-space for both single-agent and multi-agent environments are the same; however, in the multi-agent setting the state-space is highly volatile because of two reasons. First, the order the agents take action matters so that the state-space representation could be different even for the same wall configuration as shown in the first row of Figure 8. The wall configuration is the same in both plans; however, in the left one, the wall's order is brown, magenta, and blue, while in the right one the order is brown, blue, and magenta.

Second, each wall partitions the remaining space into two sub-regions and we always assign the smaller area as the corresponding area of that wall. This leads to the situation that we call the switching effect. To understand the switching effect consider the second row of Figure 8 and imagine the left plan is the current state of the environment and it is the blue agent's turn to take an action. Let's assume the blue agent decides to rotate its left segment through 90° *CCW*. Since we assign the small area to the agent, the blue agent owns the blue region shown in the right plan. Thus, the area of the blue agent suddenly switches from the top (Figure 8.c) to the middle of the plan (Figure 8.d).

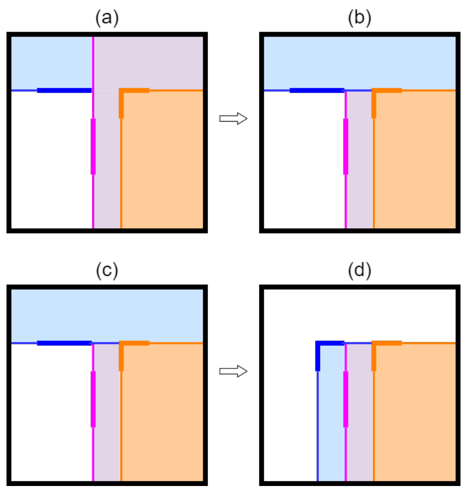

Figure 8: Challenges of the multi-agent designing. The first row shows how the order of walls can change the ultimate layout. The second row displays the switching effect causing a highly fragile state-space.

This high volatility makes the learning process more difficult but creates more opportunities because, firstly, the order of the wall can be embedded in the learning process by defining a master agent (can be also called laser-man) which learns what is the best order for the walls in each state of the environment. Secondly, the switching effect allows the agent to rapidly switch to other parts of the solution space that otherwise might need a lot of actions to be reached. Moreover, the area assigning is a task that can be embedded in the learning process as well so that the agent itself decides to own the small area or the big one. This is a relevant learning task as in many design scenarios we might need to create large rooms.

C.3 Reward

The reward in the multi-agent setting is similar to the single-agent. The only difference is that at end of the episode the environment gives the reward to all agents, but within an episode, the reward belongs to only the agent who just took an action, not the rest of the agents.

D Computational resources and source codes

Training PPO agent for each of the defined scenarios takes about an hour on average on a single GPU machine with NVIDIA GeForce RTX 2080 Ti, AMD® Ryzen threadripper 2920x 12-core processor × 24, and 64 GiB RAM.

The RLDesigner environment and the PPO agent codes are publicly available on the first author's GitHub.