IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received July 1, 2020, accepted August 2, 2020, date of publication August 12, 2020, date of current version October 13, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3016091

Generating 2D LEGO Compatible Puzzles Using Reinforcement Learning

CHEOLSEONG PARK[1], **HEEKYUNG YANG**[2], **AND KYUNGHA MIN**[1]
[1]Department of Computer Science, Sangmyung University, Seoul 01630, South Korea
[2]Division of SW Convergence, Sangmyung University, Seoul 01630, South Korea

Corresponding authors: Heekyung Yang (yangkh@smu.ac.kr) and Kyungha Min (minkh@smu.ac.kr)

This work was supported by the National Research Foundation of Korea (NRF) under Grant NRF-2018R1D1A1A02050292.

ABSTRACT We present a framework that generates a 2D Lego-compatible puzzle layout of greater than thousands pieces of bricks using a reinforcement learning technique. Many existing 2D legorization strategies have limitations in producing a Lego layout, which is composed of more than thousands of pieces. We attack this problem by employing a reinforcement learning technique, which accelerates the progress of various game strategies. We represent the legorization process as a game tree search problem, where each leaf node of the tree corresponds to a Lego layout. The goal of legorization is to find an optimal Lego layout that achieves maximum reward. To efficiently find a leaf node for the maximum reward layout, we reduce the search space using a dueling deep Q-Network (DQN), which is a widely used reinforcement learning model. Our framework is composed of a learning stage and a legorization stage. In the learning stage, we design a dueling DQN model and train this model using three heuristics for legorization strategies. In the legorization stage, we efficiently generate a large-scaled 2D Lego-compatible puzzle layout by reducing the search space using the trained dueling DQN. This approach enables us to produce a puzzle layout of more than a thousand of pieces, which has not been feasible for existing legorization schemes.

INDEX TERMS Lego, reinforcement learning, deep Q-network, legorization, heuristic.

I. INTRODUCTION

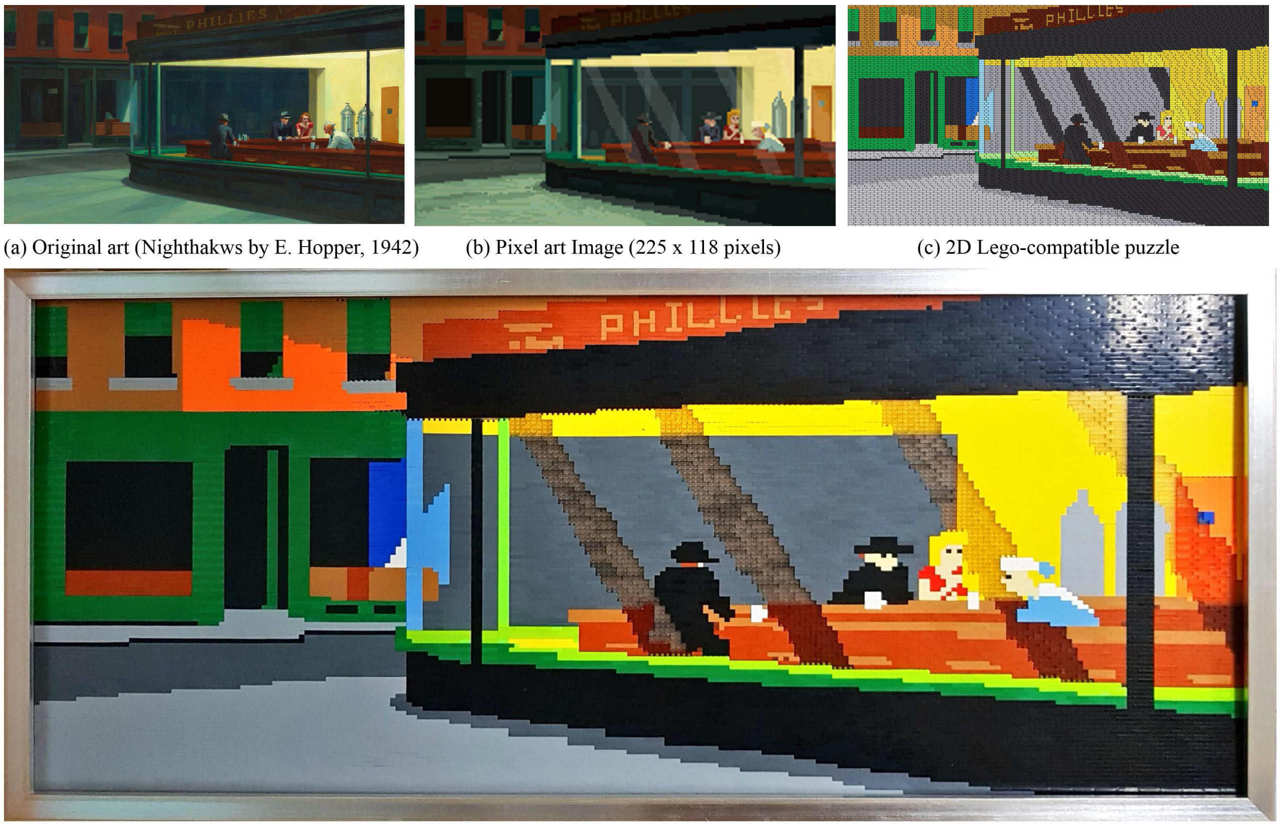
Image-based puzzles, which are played by placing small pieces in their proper positions until a target figure is completed, have been widely popular for a long time. Jigsaw puzzle is one of famous image-based puzzles. However, the pieces of a jigsaw puzzle can only be used for a given target image. We recognize a strong need for an image-based puzzle with reusable pieces. Therefore, we devise an image-based puzzle that uses Lego bricks for its pieces.

We identify another requirement for our study in the widespread favors on pixel art, which represents complex objects or scenes in a very low resolutional images. Unlike Jigsaw puzzle, which uses an image of its own resolution, Lego-based puzzles require pixel art images for their input. Fortunately, the unceasing favors on pixel art will increase the needs for the Lego-based puzzles.

Lego® bricks are one of the most widely beloved toys, enjoyed by children and adults alike. Many users complete target models by assembling bricks according to an assembly guidance map. Lego designers exploit various characters from multiple types content, such as movies, animations, games, and comics to produce their models. Recently, many Lego users have expressed their wish to build their own Lego models based on their favorite characters using common Lego bricks. However, only a few Lego specialists can design such models. Therefore, we present a framework that constructs an assembly guidance map for a Lego model based on user-selected images. By combining these two requirements, we present a framework that constructs an image-based puzzle layout using reusable Lego-compatible bricks from user-selected images, as shown in Fig. 1. Our Lego-compatible puzzle is assembled using Nano® bricks [16], whose shape is similar to a Lego® brick. The physical scale of a Nano® brick is approximately 1/8 of a Lego® brick.

Some studies [6]–[8] presented 2D legorization schemes that build layouts of Lego-compatible bricks from low resolutional images. They have limitations in producing large-scaled Lego layout, which is completed by assembling thousands of bricks. Since large-scaled Lego layout is desired by many Lego artists and enthusiasts, we aim to present a framework that generates large-scaled 2D Lego-compatible

The associate editor coordinating the review of this manuscript and approving it for publication was Jihwan P. Choi.



(a) Original art (Nighthakws by E. Hopper, 1942) (b) Pixel art Image (225 x 118 pixels) (c) 2D Lego-compatible puzzle

(d) Real Lego-compatible puzzle (88.3cm x 36.5cm with 7817 bricks)

FIGURE 1. The milestones of our algorithm: An original artwork image in (a) is pixelized to a pixel art image in (b). On (b), we build a 2D Lego-compatible puzzle in (c), which is completed to a real Lego-compatible puzzle with 7,817 pieces of Nano® bricks.

layout. Many legorization schemes suffer from the heavy computational loads for producing large-scaled Lego layout. To resolve this problem, we employ a reinforcement learning technique, which successfully improves the efficiency of many game solving strategies such as AlphaGo [17]. Our framework is distinguished from existing automatic Lego generation frameworks by three points.

The first point is that the input to our framework is a low resolution image, such as a pixel art image, whereas most existing frameworks exploit 3D voxel models for their input. By restricting the input to a low-resolution image, we can concentrate on a Lego generation strategy for a 2D Lego layout. Most Lego generation frameworks pursue various virtues of Lego layouts, including stability, aesthetics and efficiency via heuristics, such as the cover ratio, big brick, perpendicularity, vertical boundary, T-shaped joining, and covered edge by center heuristics. Because our framework aims to produce a 2D Lego layout, we can concentrate on heuristics that are more important and influential for 2D layouts. This concentration enables our framework to efficiently produce a 2D Lego layout with specific characteristics.

The second point is that we employ a game framework for the construction process of the Lego layout, where an optimal Lego layout constitutes winning the game. The game tree embeds all possible Lego layouts in its leaf nodes and the reward of the game is designed according to the heuristics we pursue in the Lego generation. We formulate the construction of a puzzle layout from an input pixel artwork image as a *game* that places proper bricks on the image until a brick layout is completed. Each brick produces a corresponding reward. At the completion of the Lego layout, we estimate the total reward of the layout by summing the individual rewards from the individual bricks. Therefore, winning the game means that we find a brick layout whose reward is maximized.

The third point is that we propose a reinforcement learning-based search strategy for the optimal Lego layout. A game is simulated in a game tree structure whose root node corresponds to the initial empty layout and whose leaf nodes correspond to complete layouts. We express the addition of a brick to a state as a child node of the node that represents that state. We search for the leaf node whose reward is the largest. An exhaustive search on the game tree that can find the optimal layout is not feasible due to the heavy computational load. Therefore, many existing schemes employ heuristic search algorithm such as beam search or A* search for game trees. These schemes, however, have a size limitation when finding an optimal layout.

Our strategy aims to reduce the search space of the tree. From our observations, the influence of a Lego brick is limited only to its neighboring bricks. Therefore, we assign a region of interest (ROI) in the tree to reduce the search space in the tree. Another observation is that we can reduce the candidates for a proper brick by properly training the selection strategy for the brick. Accordingly, we employ a dueling deep Q-network (DQN) structure, which is a widely used reinforcement learning scheme. These two strategies reduce the search space, allowing our framework to find an optimal brick layout with a reasonable computational load.

This article is organized as follows. In Section II, we briefly review the related work on legorization and reinforcement learning. We present an overview of our approach in Section III. In Sections IV and V, we explain the two stages of our approach: the learning and legorization stages. We present our results in Section VI and finally draw our conclusions in Section VII.

II. RELATED WORK

A. 3D LEGO GENERATION STUDIES

Most 3D Lego generation schemes have employed a greedy approach for building Lego layouts from 3D meshes or voxel models.

1) EARLY WORK

Early Lego generation studies depended on several heuristics that define the design scheme of a Lego model. These studies built a Lego models by minimizing a penalty function, which is designed according to several heuristics. Gower *et al.* [4] presented pioneering work devising six heuristics for Lego model design. These are the cover ratio, bigger brick, perpendicularity, vertical boundary, T-shaped joining, and covered edge by center heuristics. They designed a penalty function that considers these six heuristics and presented a heuristic approach that minimizes this function. Later, Petrovic [5] minimized the heuristics from Gower *et al.*'s study using a genetic algorithm, and van Zijl and Smal [6] employed cellular automata that merges 1×1 sized bricks to satisfy Gower *et al.*'s heuristics.

2) RECENT WORK

Recently, Testuz *et al.* [9] presented a graph-based greedy approach that merges 1×1 sized bricks into bricks of various shapes. They modeled a Lego model using a graph whose nodes correspond to bricks and whose edges correspond to the connectivity of the bricks. The articulation point of the graph denotes the bricks with a low level of connection, which are resolved by rearranging the close bricks. Ono *et al.* [10] also presented a graph-based greedy merge algorithm to build a Lego model using Gower *et al.*'s heuristics. Lee *et al.* [11] merged the greedy and genetic algorithm to combine 1×1 sized bricks. Zhang *et al.* [12] presented a random merge scheme that considers stability, symmetry and color. Stephenson [13] presented a four-stage search algorithm that merges 1×1 sized bricks.

Most recently, Min *et al.* [3] presented a legorization scheme that minimizes the heuristics using an A* search algorithm. Three heuristics including stability, efficiency and aesthetics were employed to express a proper Lego model. They also presented a silhouette fitting scheme to construct a voxel model that resembles the input 3D model.

3) PHYSICALLY-STABLE MODELS

Luo *et al.* [1] presented an automatic legorization framework that included physical considerations, such as the friction between bricks and the maximum load weight on the bricks. They represented a Lego model using a graph and estimated the stability of the model by measuring the weight loaded on each brick. Kosaki *et al.* [2] presented a legorization scheme that reconstructs the 3D geometry of a Lego model by combining photographs of various views of an object. They included a momentum term for their energy function to improve the physical stability of their model.

B. 2D LEGO GENERATION WORKS

Zhang *et al.* [7] presented an interactive system for building 2D Lego models from an image. They applied pattern tiling on a feature mask generated from the input image. They drew features using the users' intuition on the feature mask and presented various layer operations and brick-editing tools to modify the pattern results. Because they presented an effective user-friendly 2D Lego model generation scheme, the lack of algorithms for automatic Lego generation means that the quality of their results depends heavily on the users' interactions.

Kuo *et al.* [8] presented a scheme for building 2D Lego models from pixel art images. They deformed the shape of the pixel art image to obtain a proper centroid. Then, they changed the colors of the image according to the available color palette. To build a model, they optimized the layout of the model by resolving dangling parts of the model and by avoiding edge coverage by the bricks.

These two studies produced 2D Lego models from pixel art images. They have a common limitation in that the size of their resulting models is restricted. We determine that their limitation arises from their greedy approach-based optimization schemes. We aim to resolve this limitation by employing an exhaustive search using reinforcement learning.

C. REINFORCEMENT LEARNING

Reinforcement learning (RL) is an area of machine learning inspired by behaviorist psychology and is concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward [14], [24]–[26]. Instead of depending on labeled training data, RL records the state, action, and reward at each iteration and trains its model according to the chosen rewards.

Mnih *et al.* [18] proposed Deep Q-Network (DQN), which can train AI agents to play an Atari game better than human players. The DQN trains a Q-network by estimating the rewards of the actions for a given state. Even though DQN improves the accuracy and efficiency of existing Q-Network in a great scale, it suffers from an overestimation problem that causes inaccurate results.

Hasselt *et al.* [19] presented double DQN, which resolves the overestimation problem by decoupling the process of selecting maximum value of the next state into different networks. The decoupled processes are the process of determining action of maximum return value and the process of estimating the values of candidate actions.

Schaul *et al.* [20] improved the uniform sampling scheme of vanilla DQN by devising a prioritized scheme from replay buffer that samples the transitions of higher training effects more frequently. This scheme estimates priority for a transition by computing the probabilities about recent TD errors, and the priority is employed for determining transitions of higher learning effects.

Bellemare *et al.* [21] presented distributional DQN scheme that is trained to build an approximation about the distribution of return values of the actions. Then, they construct an approximate probability density function for the distribution and adjust the parameters of the probability density function to reduce the difference of the function and the distribution of the return values. They employ the approximate probability density function for the training of the model.

Wang *et al.* [23] improved DQN by presenting dueling DQN, which trains the Q-network faster than the original DQN. Dueling DQN separates the optimization function into a value function $V(s)$, which depends only on the state, and an advantage function $A(s, a)$, which depends on both the action and the state. Dueling DQN trains $V(s)$ more efficiently than conventional DQN and consequently shows better robustness in training the network by using a separate $A(s, a)$.

Very recently, Hessel *et al.* [22] surveyed and compared many existing deep reinforcement techniques on Atari games. Among the compared methods including vanilla DQN [18], double DQN [19], prioritized replay buffer [20], distributional DQN [21], and dueling DQN [23]. Even though the rainbow scheme that conditionally selects a proper scheme adaptively shows best performance, dueling DQN shows very competitive performance. Therefore, we select dueling DQN in our work.

III. OVERVIEW OF THE ALGORITHM

We present a legorization framework that produces a 2D Lego layout from low resolution pixel artwork images using dueling DQN, which is a well-known RL strategy. We aim to build a brick layout, which is defined as an arrangement of proper bricks that covers all the pixels of the input image.

Our framework regards the legorization process as a game to produce a brick layout with the maximum expected reward in two stages: a *learning stage* in which the rules for choosing a proper Lego brick are learned and a *legorization stage* in which an optimal Lego layout is produced using these rules.

In the learning stage, we build a dueling DQN structure to learn the rules for choosing a proper brick at a given state. To develop the rule for choosing a proper brick, we suggest three heuristics: efficiency, stability, and aesthetics. From these heuristics, we design a metric for estimating the rewards to measure how much a brick layout satisfies these heuristics. Our dueling DQN is trained to estimate the reward for a brick and to propose a candidate list of optimal bricks. To train our dueling DQN, we build brick layouts using five sample images and construct a dataset by sampling patches from the sample images and their layouts.

In the legorization stage, we fill an input image with proper bricks using our trained dueling DQN. When choosing the proper brick, we apply an exhaustive search on the game tree that formulates the legorization process. Because an exhaustive search on the entire input image for every brick results in a computational load that is too large, we present two strategies. One strategy is to set an ROI on a pixel to choose a brick. Because bricks placed at a distance do not influence each other, an exhaustive search restricted to an ROI does not interfere with finding a layout with the maximum reward. The second strategy is to employ our trained dueling DQN to choose the candidate bricks that are expected to achieve a maximum reward. Using these two strategies, our legorization scheme produces a brick layout with a maximum reward in a reasonable computational time.

IV. LEARNING STAGE

A. PRELIMINARIES

We employ only four types of bricks 1×1, 1×2, 1×3, and 1×4 to build a Lego layout (see Fig. 4 (a)), whereas some existing works on 2D Lego layouts, e.g. [7], [8], have employed six bricks. The fundamental reason for using four bricks is to reduce the computational load when building a layout. Because our approach is designed based on game tree traversal, the number of candidate bricks corresponds to the number of child nodes in the tree, which influences the computational load. Therefore, we can achieve a significant improvement in the computational time by building a Lego layout using this strategy.

B. HEURISTICS

We employ three heuristics to estimate the reward: two of Gower *et al.*'s heuristics [4] and the color-match heuristic. Gower *et al.*'s heuristics that we employ are the *big brick* and *vertical boundary* heuristics. The big-brick heuristic, which favors bigger bricks over smaller bricks, influences the efficiency of the model, and the vertical-boundary heuristic, which requires bricks to cover the vertical boundary in the incident layer, influences the stability of the model. The color-match heuristic forces the color of our model to match that of the input image. The reward is estimated at the end of each action.

We define the heuristics as our goals when building a 2D Lego layout. The three heuristics we employ for the

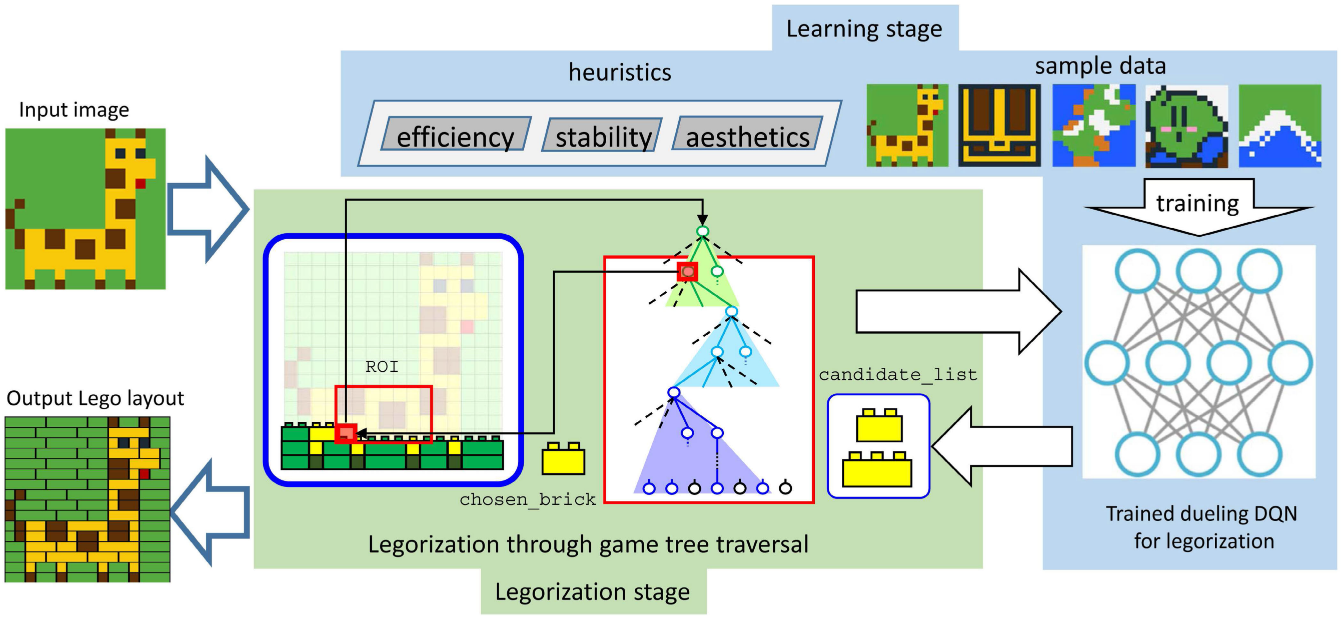


FIGURE 2. The process of our algorithm: An input image is legorized through our Legorization stage to produce a Lego layout. For the Legorization stage, we present a Learning stage where the Legorization heuristics are trained through a dueling DQN strategy [23]. The dueling DQN strategy produces a candidate list of bricks to fill the pixel of interest.

generation of the 2D layout of Lego-compatible bricks are therefore *stability*, *efficiency* and *aesthetics*. These are expressed in terms of their reward, which are measured in the legorization process.

1) STABILITY AND ITS CORRESPONDING REWARD

Our first heuristic is stability, which guarantees that the assembled layout is physically stable. We define the consecutive edges of neighboring bricks as the *vertical boundary* between two neighboring bricks (see Fig. 3 (a)). When estimating the vertical boundary, we have several rules:

Rule 1. The vertical boundary for the continuous edges is accumulated. As illustrated in Fig. 3 (a), the vertical boundary for two continuous edges has a length of 2.

Rule 2. The vertical boundary of continuous edges that are not covered by a brick has a length of ∞ (see Fig. 3 (b)).

Rule 3. A vertical boundary that belongs to a longer vertical boundary is not counted to avoid a duplicate estimation. In Fig. 3 (c), the vertical boundary between u' and v' is not considered, because it belongs to the vertical boundary between u and v.

The longer the vertical boundary, the less stable the layout. Even though we embed our model, which is composed of thousands of bricks within a frame that holds them stably, the assembly process for the model requires stability. Further, some of our models are composed of less than a few hundreds of bricks and do not require a frame. Physical stability is essential for these models. Therefore, stability is the first heuristic we pursue for our model.

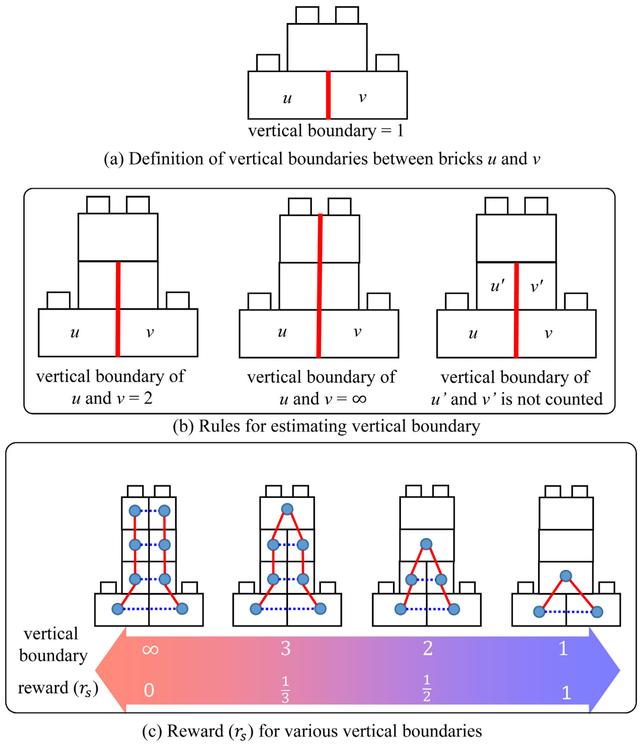

u
v
vertical boundary = 1
(a) Definition of vertical boundaries between bricks u and v
vertical boundary of u and v = 2
vertical boundary of u and v = ∞
u'
v'
vertical boundary of u' and v' is not counted
(b) Rules for estimating vertical boundary
vertical boundary
reward (r_s)
∞
3
2
1
0
1/3
1/2
1
(c) Reward (r_s) for various vertical boundaries

FIGURE 3. The stability heuristic and its rewards (r_S): As shown in (c), the bigger bricks present higher reward than the smaller bricks.

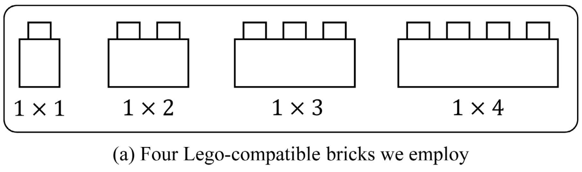
The reward for stability, denoted as r_s, considers the length of the consecutive vertical boundaries of the layout. Because a longer vertical boundary decreases the stability of the layout, we define a covering brick that combines the vertical boundary in Eq. (1):

$$r_s(u,v) = \begin{cases} \frac{1}{m}\sum \frac{1}{d(u,v)}, & u \text{ and } v \text{ are neighboring} \\ 0, & \text{edge of } u \text{ \& } v \text{ is not covered} \end{cases} \tag{1}$$

where $d(u, v)$ is the length of the vertical boundary between two neighboring bricks u and v, and m denotes the total number of edges in the layout. Fig. 3 (c) illustrates some cases of r_s.

2) EFFICIENCY AND ITS CORRESPONDING REWARD

Our second heuristic is efficiency, which requires that our puzzle layout be assembled by as a small number of bricks as possible. As illustrated in Fig. 4 (a), we employ four types of Lego-compatible bricks. An increase in the number of bricks results in a higher brick cost and a longer assembly time and, accordingly, a larger amount of effort. Therefore, the efficiency, which aims to decrease the number of bricks for a model, is the second heuristic we pursue.



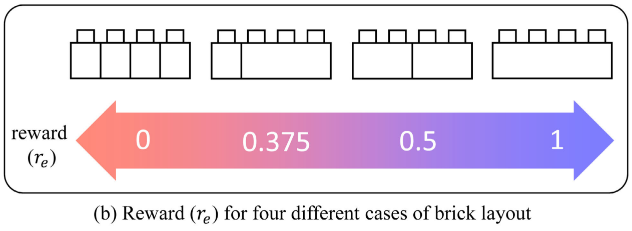


FIGURE 4. The efficiency heuristic and its rewards (r_e).

The reward for efficiency, which is denoted as r_e is designed by assigning a higher reward for bigger bricks and a lower reward for smaller bricks. Eq. (2) assigns the weight for each brick we consider.

$$brick\_weight = \begin{cases} 0, & \text{if brick is } (1 \times 1) \\ 1/2, & \text{if brick is } (1 \times 2) \\ 3/4, & \text{if brick is } (1 \times 3) \\ 1, & \text{if brick is } (1 \times 4) \end{cases} \tag{2}$$

Therefore, the sum of all the rewards can be estimated using Eq. (3).

$$r_e = \frac{1}{n}\sum brick\_weight \tag{3}$$

where n denotes the total number of bricks in the layout. Fig. 4 (b) illustrates several cases of r_e.

3) AESTHETICS AND ITS CORRESPONDING REWARD

Our third heuristic is aesthetics, which necessitates that our puzzle layout match the input pixel artwork image. Because our model is composed of bricks greater than or equal to a size of 1×1 size, some details of the input pixel artwork image may be suppressed or smeared. We aim to avoid a degradation of the input pixel artwork image. Therefore, the aesthetics, which aims to match our model to the input image, is our third heuristic. As illustrated in Fig. 5, some bricks excess the boundaries of the input image or do not match the input image: in such a case, the reward for the aesthetics, denoted as r_a becomes 0. The formula for r_a is given in Eq. (4).

$$r_a = \begin{cases} 1, & \text{if color matches} \\ 0, & \text{if color mismatch or out of brick} \end{cases} \tag{4}$$

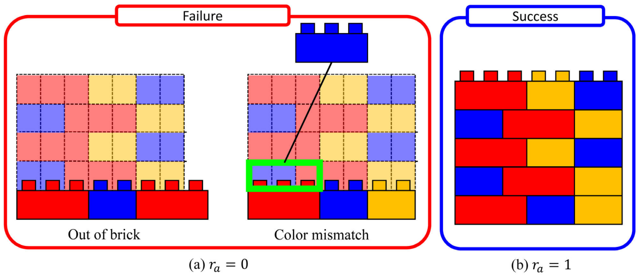


FIGURE 5. The aesthetics heuristic and its rewards (r_a).

4) SUMMING THE REWARDS

After completing a layout, the total reward is estimated by summing the r_s and r_e rewards. The r_a reward is multiplied by the sum because aesthetics heuristic is prioritized. Therefore, the formula for estimating the total reward r for a layout is given in Eq. (5):

$$r = r_a(\omega_e r_e + \omega_s r_s), \tag{5}$$

where ω_e and ω_s are the weights for the efficiency and stability, respectively, which we set both weights to 0.5.

C. PREPARATION FOR REINFORCEMENT LEARNING

1) DUELING DQN FOR REINFORCEMENT LEARNING

We choose dueling DQN [23] as our reinforcement learning technique for legorization. Dueling DQN separates the value function as V(s), the value function, and A(s, a), the advantage function, for an efficient estimation of the maximum return value. By separating and dueling V(s) and A(s, a), dueling DQN can determine whether a state is valuable or not without estimating actions from the state. Therefore, dueling DQN shows improved efficiency than vanilla DQN [18], since it avoids estimations on the states whose values are expected to be bad. The strategy of dueling DQN also shows better performance than double DQN [19], which estimates maximum computation for decoupled processes. The prioritized experience replay strategy [20] is not proper for our

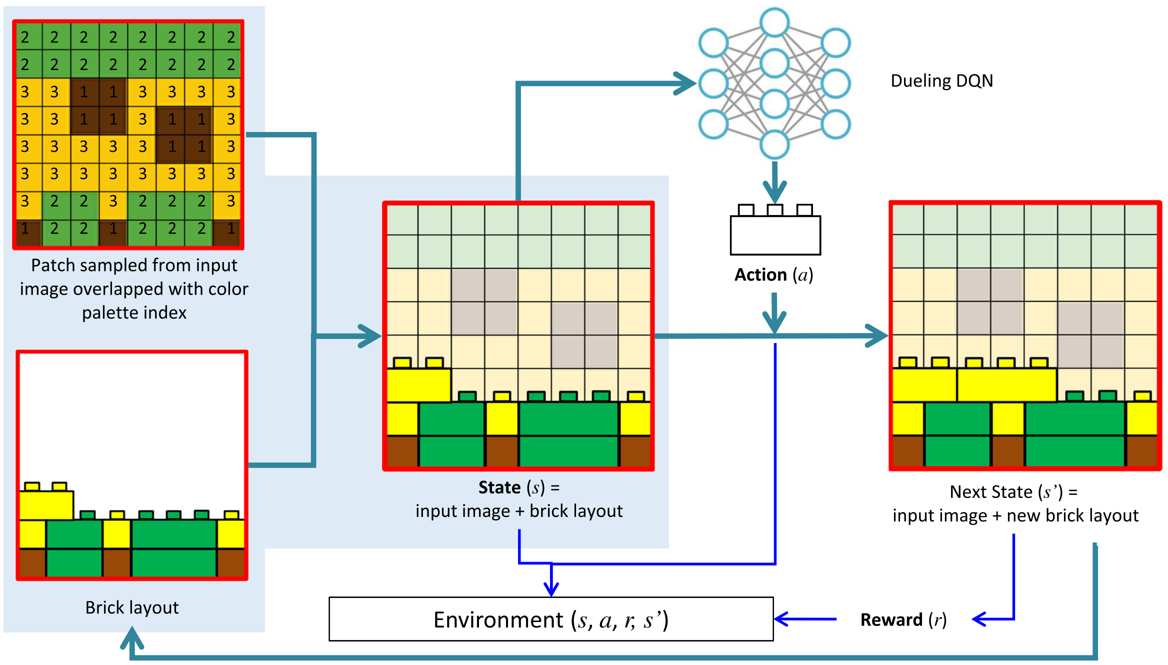

Patch sampled from input image overlapped with color palette index
Brick layout
State (s) = input image + brick layout
Dueling DQN
Action (a)
Next State (s') = input image + new brick layout
Environment (s, a, r, s')
Reward (r)

FIGURE 6. The environment of our framework: state, action, reward and next state. To determine a brick that will fill the next pixel in the current brick layout, our environment estimates and records the corresponding reward of the newly chosen brick.

framework, since our framework assigns the weights for the bricks according to our heuristics.

2) ENVIRONMENT

The RL *environment*, which is defined as the set of all components required to execute learning, includes the *state* (s), *action* (a), *reward* (r) and *next state* (s'). According to the environment, we build a state and choose an action whose reward is then calculated. (s, a, r, s') are stored in replay memory for training.

a: STATE

We express the state for legorization as an input pixel art image and the brick layout filling the input image. The blue rectangle in Fig. 6 illustrates the state of our framework. Note that the bricks are filled from the lower left corner of the image. The brick layout, which is empty in the initial state, covers more and more pixels as legorization proceeds. In the final state, the brick layout covers all the pixels of the input image. In practice, we consider a patch of $k \times k$ pixels sampled from the input image for the training of the dueling DQN.

b: ACTION

The action of our model is to choose a brick that will maximize the reward of the layout. Since we consider the bricks whose color is identical to the pixel to fill, we have four candidates as illustrated in Fig. 4 (a).

c: REWARD

The reward is formulated from the heuristics suggested in Section IV-B. We estimate the reward for a layout using Eq. (5).

For a state s, dueling DQN produces an action whose expected reward is the maximum. In our framework, an action a is the choice of a brick that fills the pixels of the input image. By adding a to s, we formulate the next state s' and a reward r is calculated from that state. The environment, composed of the state, action, reward, and next state, is recorded for the training of the dueling DQN. The legorization environment is illustrated in Fig. 6. We repeat this process until the brick layout fills the entire input image.

We train the dueling DQN by continuously adding bricks until we reach a final state. At the final state, we estimate the loss function and train our dueling DQN by backpropagating the value of the loss function. We repeat this training process several times until the value of the loss function falls below a given threshold.

D. LEARNING DUELING DQN

1) THE STRUCTURE OF THE DUELING DQN

DQN estimates the reward from actions on an input state. We design our DQN structure based on the network of Wang *et al.* [23]. For an input whose magnitude is $k \times k \times 2$, we execute convolution operations three times. The first convolution uses a 5×5 mask whose padding is assigned to be 2 to preserve the magnitude of the input on the resulting feature map. The second and third convolutions use a 3×3

mask whose padding size is 1. After the convolution layer, we assign two fully connected layers. The structure of our dueling DQN is illustrated in Fig. 7. The estimation of the final reward is computed by adding $V(s)$ and $A(s, a)$ from the network.

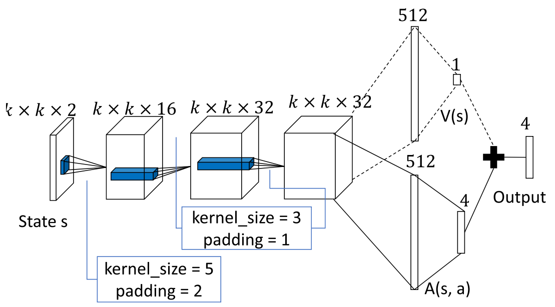


FIGURE 7. The structure of our dueling DQN ($k = 8$).

2) Q FUNCTION

The Q function at a state s with an action a, which is denoted as $Q(s, a)$, returns the estimation of the maximum final reward, when an action a is executed on a state s. Action a is one of four candidates: $\{1 \times 1, 1 \times 2, 1 \times 3, 1 \times 4\}$. By Eq. (4), the color of a brick is restricted to match the color of the corresponding pixel in the input image. Therefore, the action a considers only the type of the bricks, not the color of the bricks. Therefore, the reward we have to estimate is $Q(s_0, a)$, where s_0 is the initial state and a is an action at the state. We denote r as the reward of an action a at a state s. We also denote s' as a state that is transferred from s by an action a. The reward from s' by an action a' is denoted as $Q(s', a')$. From this, $Q(s, a)$ is estimated by adding r, the reward from a at s, and the maximum value of $Q(s', a')$. We apply a weight term γ for the discount factor of the future reward. $Q(s, a)$ is defined in Eq. (6). The relation between $Q(s, a)$, r and $Q(s', a')$ is illustrated in Fig. 8.

$$Q(s, a) = r + \gamma max_{a'} Q(s', a'), \tag{6}$$

where $max_{a'} Q(s', a')$ for final state s' is q_{max}.

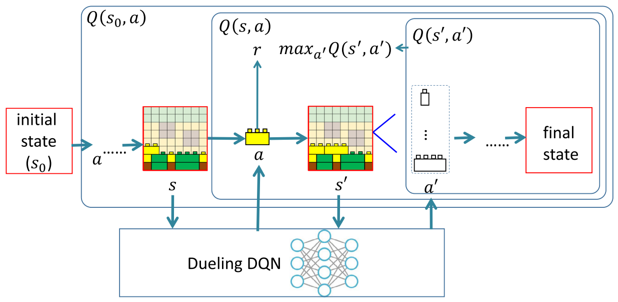


FIGURE 8. The definition of the Q function.

3) TRAINING PROCESS

The training of the DQN employs backpropagation. For the backpropagation, we estimate the error between the estimated value and the target value. The target value, which is recorded during the legorization process, is noted as s, a, and r, and the estimated value comes from the Q function. $Q(s, a)$, the result of the Q function, is the estimation of the final reward. Because $Q(s, a)$ is the estimation of the final state, we estimate the reward from the current reward r by adding the maximum reward, which is estimated as $\gamma max_{a'} Q(s', a')$. Therefore, the loss is estimated using Eq. (7).

$$loss = \frac{1}{2}[r + \gamma max_{a'} Q(s', a') - Q(s, a)]^2 \tag{7}$$

For the training stage, we select five 16×16-sized pixel art images and train our DQN model 50,000 times for each image (see Fig. 9 (a)). The outcomes of the training process are shown in Fig. 9 (b).

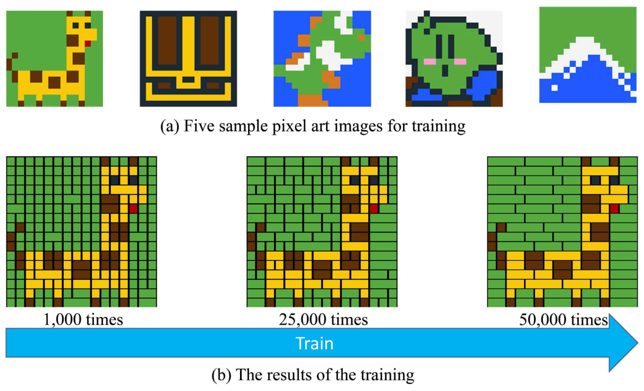

(a) Five sample pixel art images for training
1,000 times
25,000 times
50,000 times
Train
(b) The results of the training

FIGURE 9. The training samples and training results.

In the training, we set batch size as 32 and learning rate as 0.001. The γ in Eq. (6) and (7) is set as 0.99. The weight decay is set as 0.5. We illustrate a reward curve for the training process in Fig. 10.

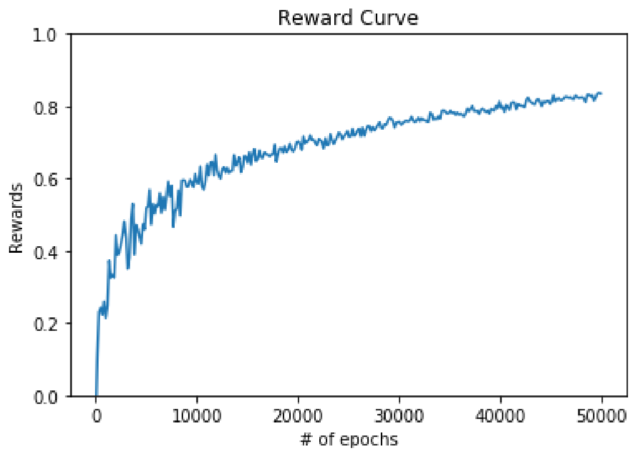


FIGURE 10. The reward curve for our training.

V. LEGORIZATION STAGE

A. PREPROCESSING

The input of our legorization stage is a low resolution pixel art image and a color palette, which is constructed from the colors of the Lego-compatible bricks. We build an image map by substituting the colors of the input image into an index, which corresponds to a color in the palette. Therefore, if the color from the input image is not included in the color palette, we find the closest color to it in the color palette by minimizing the distance in Lu*v* coordinates.

B. LEGORIZATION STRATEGY

We add a Lego-compatible brick starting in the lower left corner of an input image until the entire image is filled with bricks. As shown in Algorithm 1, we execute this process for each line of the input image starting on the bottom line (line 3 of Algorithm 1). During this process, the pixel where we are going to place a brick is denoted as the *pixel of interest* (*POI*), and the brick that will be placed on the POI is the *brick of interest* (*BOI*) (line 6 of Algorithm 1). The function *Choose_optimal_brick* is defined in Algorithm 2. To determine the BOI, we need to consider the expected reward of a brick layout constructed from the POI. To estimate this reward, we simulate the legorization process from the POI to the remaining region of the input image and pick the BOI that optimizes the total reward of the Lego layout. The simulation process is executed through Algorithm 3. We include two key ideas in this simulation. The first key idea is that bricks located sufficiently far from each other do not influence each other. Therefore, we build an *ROI* where we simulate the legorization process. The ROI is computed adaptively according to the location and environment of the POI. Our second key idea is to employ an RL-based game tree search strategy for the simulation. Each node of the tree has four child nodes, which correspond to the four candidate bricks. The child node to explore is decided via the dueling DQN, our RL model, which was trained in the previous stage. Our legorization strategy is expressed in Algorithm 1.

C. ADAPTIVE ROI

We build an ROI as a box of $N \times M$ pixels whose lower left corner is the POI. For the pixels in the box, we exclude any pixels whose colors are different from the color of the POI because our aesthetics heuristic necessitates that pixels of different colors be filled with different bricks. We assign set N to 12, allowing three 1×4 bricks can be placed, and set M to 4, so that four lines are covered. Note that bricks located at farther distances than this do not influence each other. Fig. 11 illustrates various cases of POIs and ROIs.

D. LEGORIZATION VIA GAME TREE SIMULATION

We build a game tree whose nodes correspond to the Lego layout. The root node of the tree corresponds to an empty layout, whereas the leaf nodes correspond to final layouts that fill the entire image. Each node has four child nodes, each

Algorithm 1 Legorization

```
Input: W (width of input image), H (height of input
    image), PixelArtImage (input pixel art image)
Output: brick_layout (brick layout for the input image)
 1: brick_layout ← $\phi$
 2: POI ← (1, 1) // leftmost-bottom pixel
 3: while (POI.y <= H) do
 4:   while (POI.x <= W) do
 5:     ROI ← Set_ROI (POI, PixelArtImage)
 6:     BOI ← Choose_optimal_brick (POI, ROI,
        brick_layout)
 7:     brick_layout ← brick_layout ∪ BOI
 8:     POI.x ← POI.x + BOI.width
 9:     if (POI.x > W) then
10:       POI ← (1, POI.y + 1)
11:       break
12:     end if
13:   end while
14: end while
15: return brick_layout
```

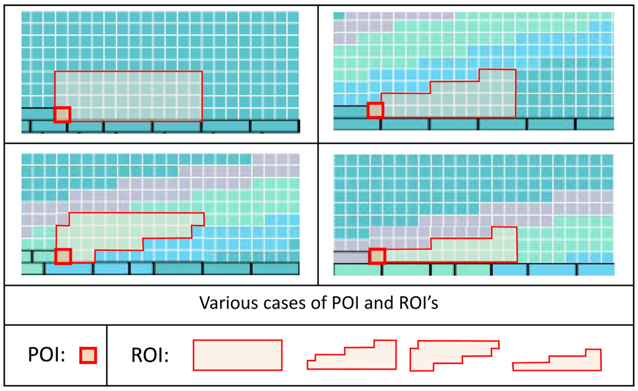


FIGURE 11. Various cases of ROIs on various POIs and their environments.

of which matches one of the four bricks we employ in the legorization process. Therefore, a child node corresponds to a layout with one brick added to the layout of the parent node. We add the bricks starting in the lower left corner of the input image for each line of the input image. We illustrate the game tree simulation in Fig. 12 (a).

In a game tree simulation, choosing a brick for a layout corresponds to choosing a child node to explore. We estimate the rewards of the complete layout produced by the candidate bricks and choose the brick that produces the complete layout for the maximum reward. For such an estimation, several strategies have been proposed. A brute-force strategy is an exhaustive search that explores all the child nodes and estimates the rewards of all the possible layouts. This strategy guarantees a layout that optimizes our heuristics; however, it is not feasible for heavy computational loads. Some researchers have employed heuristic search algorithms,

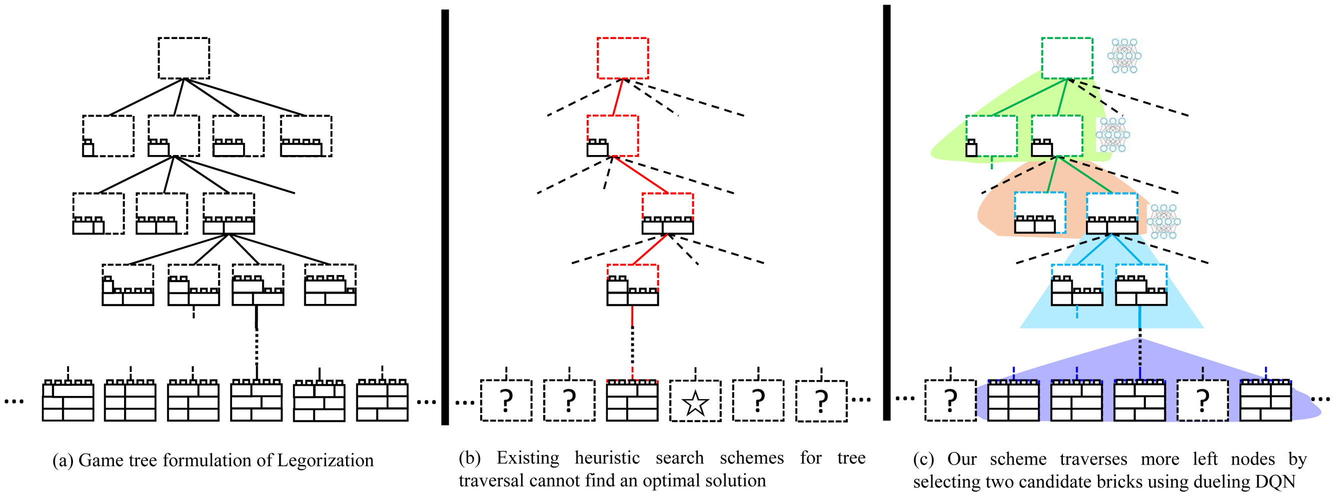

(a) Game tree formulation of Legorization
(b) Existing heuristic search schemes for tree traversal cannot find an optimal solution
(c) Our scheme traverses more left nodes by selecting two candidate bricks using dueling DQN

FIGURE 12. Illustration of the game tree and a comparison of existing schemes to our scheme. Since our scheme reduces the number of child nodes to two most promising bricks, our scheme can find a leaf node whose reward is maximum in an efficient way.

such as beam search [15] or A* search [3] to efficiently traverse the tree. Their schemes, however, do not guarantee an optimal solution or an efficient computation.

Our idea is to employ dueling DQN, an RL scheme, to determine candidate brick lists for the child node to explore. Choosing one candidate brick is similar to heuristic search schemes, and choosing four candidate bricks corresponds to an exhaustive search. Therefore, we select two bricks for our list. We execute a simulation on the ROI for the bricks on the candidate list. Because the bricks outside the ROI do not influence the BOI, tree traversal on the ROI is sufficient to estimate the layout with the maximum reward. This process is illustrated in Algorithm 2. We compare existing heuristic schemes and our approach in Fig. 12 (b) and (c).

Algorithm 2 Choose_Optimal_Brick

```
Input: POI (pixel of interest), ROI (region of interest),
    brick_layout (current brick layout)
Output: chosen_brick (a brick that optimizes the total
    reward)
 1: candidate_list ← duelingDQN (POI, ROI,
    brick_layout)
 2: optimal_r ← 0
 3: chosen_brick ← NULL
 4: for ( candidate in candidate_list ) do
 5:     r ← Simulate (POI, ROI, candidate, brick_layout)
 6:     if ( r > optimal_r ) then
 7:         optimal_r ← r
 8:         chosen_brick ← candidate
 9:     end if
10: end for
11: return chosen_brick
```

During the simulation, we repeat the process of choosing bricks using dueling DQN and filling the ROI with bricks with optimal rewards. After the simulation, we estimate the expected reward for the candidate bricks at the POI and choose a brick that produces the optimal reward. This process is described in Algorithm 3.

Algorithm 3 Simulate

```
Input: POI (pixel of interest), ROI (region of interest),
    candidate_brick (candidate brick to simulate),
    brick_layout (current brick layout)
Output: optimal_r (optimal reward)
 1: brick_layout ← brick_layout ∪ candidate_brick
 2: POI ← Update_position (POI, candidate_brick)
 3: candidate_list ← duelingDQN (POI, ROI,
    brick_layout)
 4: optimal_r ← 0
 5: for ( candidate in candidate_list ) do
 6:     r ← Simulate (POI, ROI, candidate, brick_layout)
 7:     if ( r > optimal_r ) then
 8:         optimal_r ← r
 9:     end if
10: end for
11: return optimal_r
```

VI. IMPLEMENTATION AND RESULTS

We implemented our legorization framework using Python 2.7 and PyTorch 0.3.1, which is a Python-compatible library for deep learning. The framework was implemented and executed on Linux Ubuntu on a personal computer with an Intel® Core i7 CPU, 16 GByte of memory and nVidia® Titan X GPU.

We applied the trained dueling DQN to the input pixel art images in Fig. 13 to produce the 2D Lego puzzles in Figs. 1, 14, and 15.

TABLE 1. The results of our scheme: The resulting Lego puzzle is evaluated in terms of its efficiency and stability for images in Fig. 13. As shown in the table, some of the models are completed using more than 7,000 bricks, which was not possible for the existing schemes.

	Resolution	No. of bricks	Efficiency (size of brick)					Stability (length of vertical boundary)					
			1×1	1×2	1×3	1×4	sum	1	2	3	4	≥ 5	sum
(a)	225×118	7,414	81 (1.1%)	572 (7.7%)	1,719 (23.2%)	5,042 (68.0%)	7,414 (100%)	5,482 (98.7%)	72 (1.3%)	1 (0.02%)	0 (0.0%)	0 (0.0%)	5,455 (100%)
(b)	150×75	3,322	90 (2.7%)	425 (12.8%)	918 (27.6%)	1,889 (56.9%)	3,322 (100%)	1,340 (97.0%)	38 (2.8%)	1 (0.22%)	0 (0.0%)	0 (0.0%)	1,381 (100%)
(c)	250×114	7,970	66 (0.8%)	631 (7.9%)	1,920 (24.1%)	5,353 (67.1%)	7,970 (100%)	6,389 (98.6%)	89 (1.4%)	2 (0.03%)	0 (0.0%)	0 (0.0%)	6,480 (100%)
(d)	250×114	7,875	62 (0.8%)	400 (5.1%)	2,014 (25.6%)	5,399 (68.6%)	7,875 (100%)	6,543 (98.3%)	108 (1.6%)	3 (0.05%)	0 (0.0%)	0 (0.0%)	6,654 (100%)
(e)	100×100	2,875	82 (2.9%)	185 (6.5%)	856 (29.8%)	1,745 (60.8%)	2,868 (100%)	1,398 (98.2%)	26 (1.8%)	0 (0.00%)	0 (0.0%)	0 (0.0%)	1,424 (100%)
(f)	45×46	604	7 (1.2%)	84 (13.9%)	157 (26.0%)	356 (58.9%)	604 (100%)	236 (97.6%)	5 (2.1%)	0 (0.00%)	0 (0.0%)	0 (0.0%)	241 (100%)

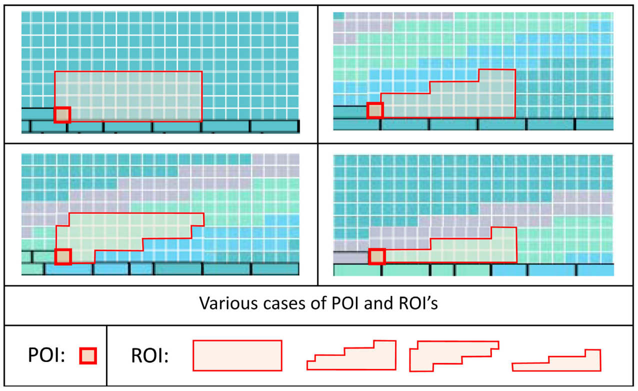


FIGURE 13. Various cases of ROIs on various POIs and their environments.

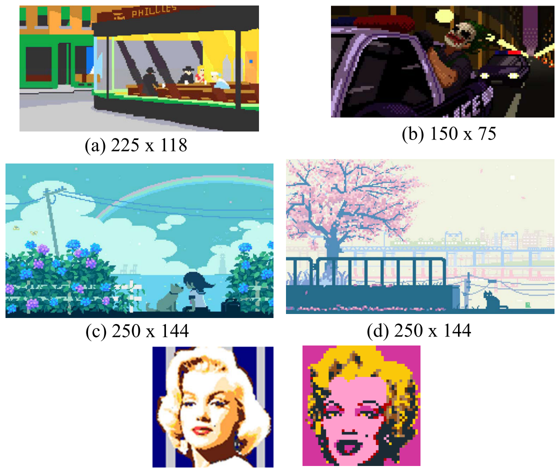


FIGURE 14. The input pixel art images.

We show layouts of the Lego-compatible puzzle in Fig. 14. In Fig. 15, we compare two layouts for a portrait at different resolutions. We physically completed two Lego models using Nano® bricks, which is a cheaper substitute for Lego bricks, as shown in Figs. 1 and 16.

A. ANALYSIS

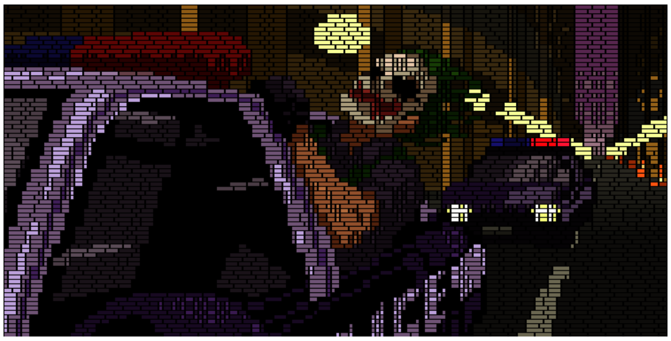
The performance of the resulting Lego puzzles is illustrated in Table. 1 and Fig. 17. The performance is measured via two aspects: efficiency and stability. Aesthetics, which is tightly preserved, is not considered in the performance analysis.

1) EFFICIENCY

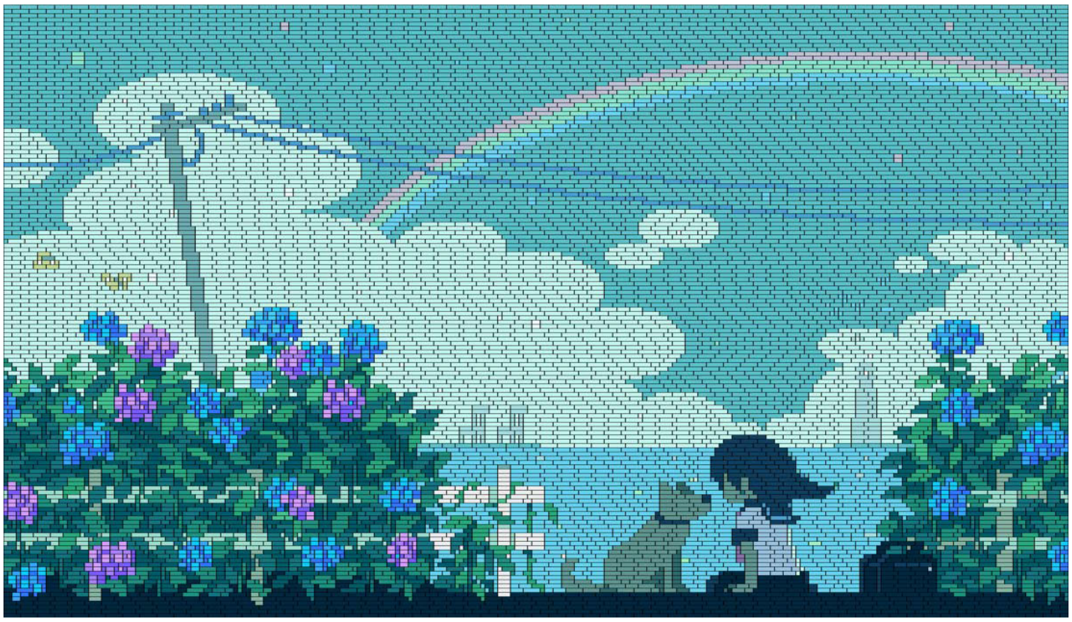
For the efficiency, we measure the distribution of the brick size, which is composed of 1×1, 1×2, 1×3, 1×4 bricks. The distribution of the brick size is illustrated in the green blocks in Table. 1 and Fig. 17 (a). As shown, the biggest brick, whose size is 1×4, is exploited at a rate of $42.1 \sim 67.1\%$ and the smallest brick, whose size is 1×1 is exploited at a rate of $0.9 \sim 2.4\%$.

2) STABILITY

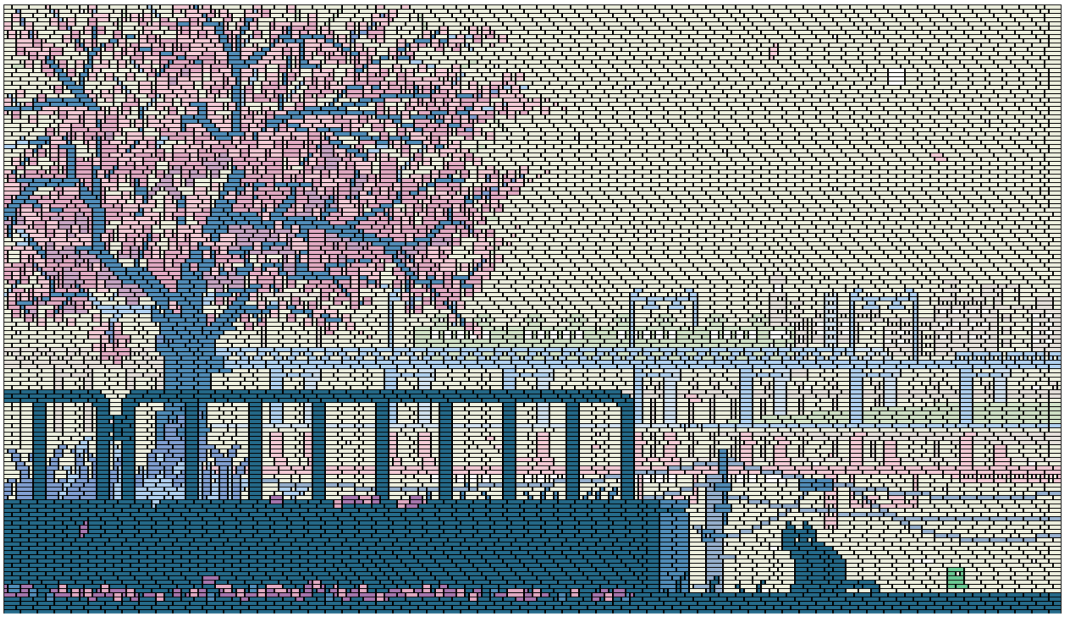
For the stability, we measure the length of the vertical boundary, which is defined as the number of consecutive brick boundaries in the vertical direction. For the most stable model, the vertical boundary is expected to be one. A long vertical boundary indicates that the Lego layout may be separated into several parts, which results in a very unstable layout. We count the vertical boundaries of bricks of identical colors because bricks of different colors are not considered in the stability heuristics. As illustrated by the blue blocks in Table. 1 and Fig. 17 (b), our resulting Lego layouts from six inputs consist of $97.0 \sim 98.7\%$ vertical boundaries of length of 1. They show no vertical boundaries that are longer



(a) The Lego model from Fig. 14 (b) whose resolution is 150 x 75



(b) The Lego model from Fig. 14 (c) whose resolution is 250 x 144



(c) The Lego model from Fig. 14 (d) whose resolution is 250 x 144

FIGURE 15. The results of our algorithm for high resolution layouts. As shown, (a) has 11,250 pixels, and (b) and (c) have 36,000 pixels. The legorization process of these high resolutional images are not feasible for the existing schemes.

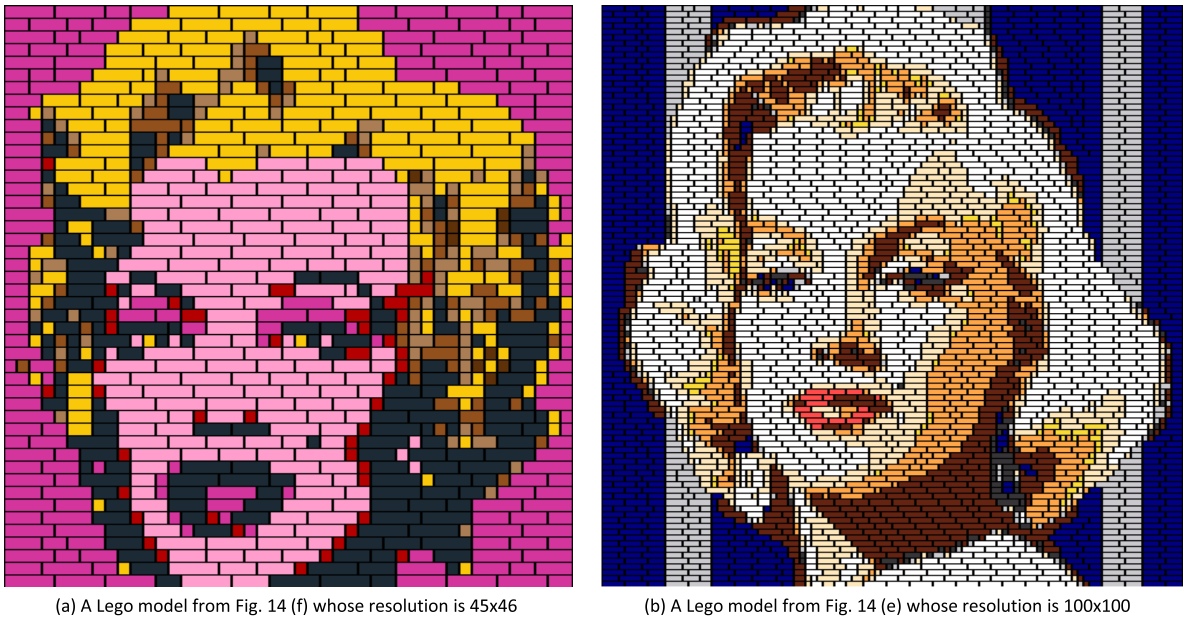

(a) A Lego model from Fig. 14 (f) whose resolution is 45x46 (b) A Lego model from Fig. 14 (e) whose resolution is 100x100

FIGURE 16. The results of our algorithm for portraits. The left model is completed using original Lego® brick, while the right model is completed using Nano®. Note that the resolution of the left panel is lower than that of the right panel.

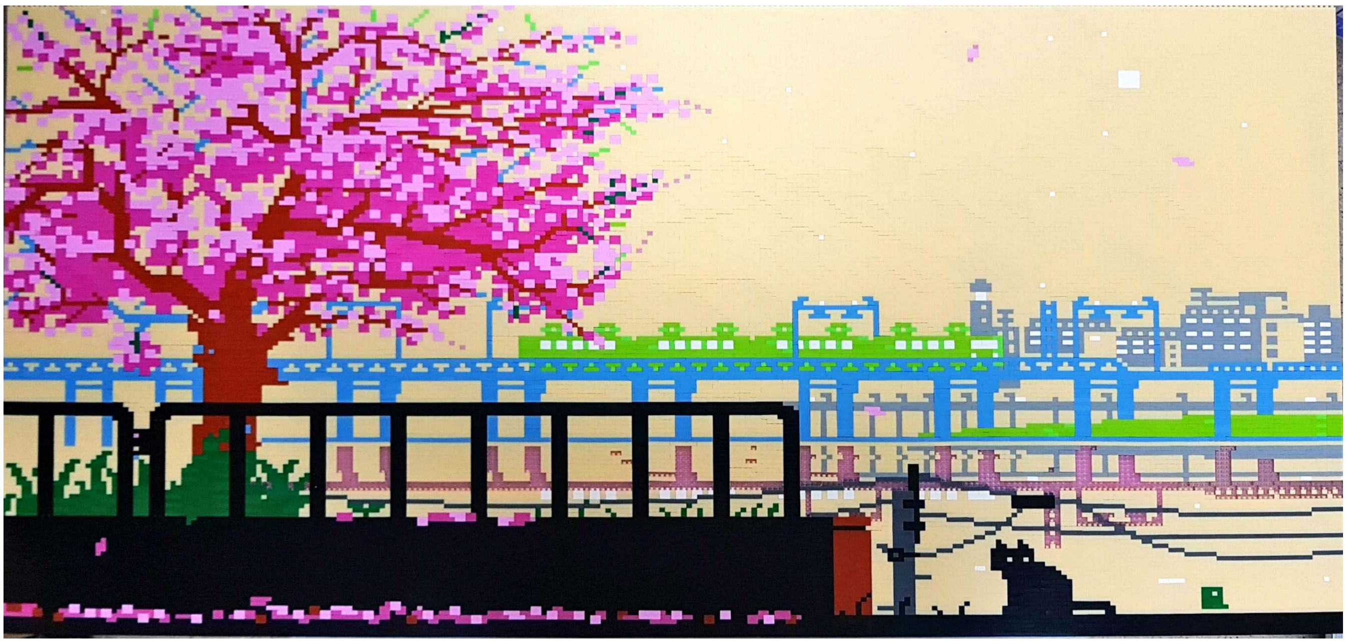


FIGURE 17. A real model from an image in Fig. 13 (d) whose resolution is 250 × 144. The corresponding Lego-compatible puzzle is illustrated in Fig. 14 (c). The number of Nano® bricks required to complete this model is 7,875.

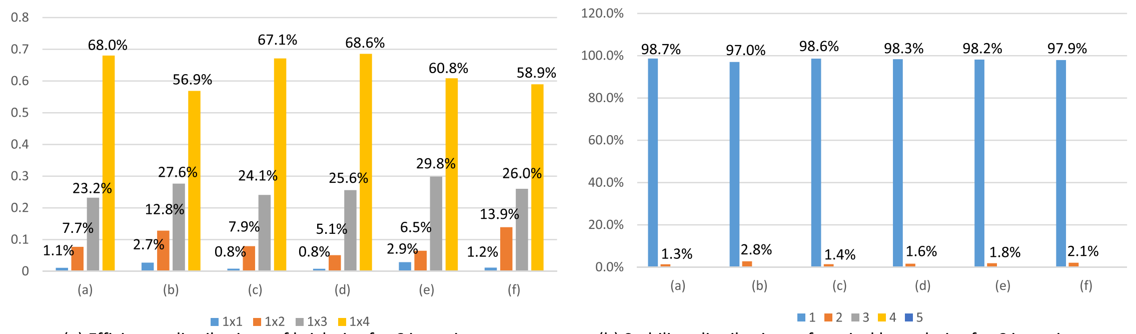
than four. Only $0.02 \sim 0.22\%$ of the vertical boundaries have a length of 3.

3) COMPUTATION TIME

We compare two existing methods including exhaustive search method and heuristic search method [3] with our method. For further comparison, we compare the computation time for four-brick set and six-brick set. The six-brick set is composed of the four bricks in the four-brick set and 1×6 and 1×8 bricks. The computation time is presented in Table. 2 and illustrated in Fig. 18.

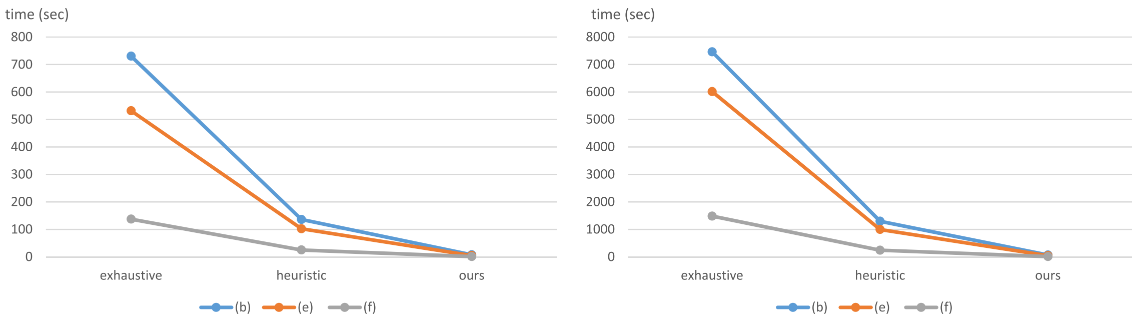
B. COMPARISON

We compared our scheme with several existing schemes, including beam search [15], cellular automata [6], graph-based search [9], and pixel2brick [8]. Because some of these schemes only process monochrome images, we generated



(a) Efficiency: distributions of brick size for 6 input images (b) Stability: distributions of vertical boundaries for 6 input images

FIGURE 18. Graphs of the performance: For the term of efficiency, the six figures in Fig. 13 is completed using 56.0% ∼ 68.6% of biggest bricks. For stability, the vertical boundary is less than 2.8% for all the input images.



(a) Computation time using four bricks (b) Computation time using six bricks

FIGURE 19. The comparison of computation time for three methods: exhaustive search, heuristic search and our method: The computation time of an exhaustive search for a model of 3,322 bricks takes much more than a model of 604 bricks, while ours show similar computational time.

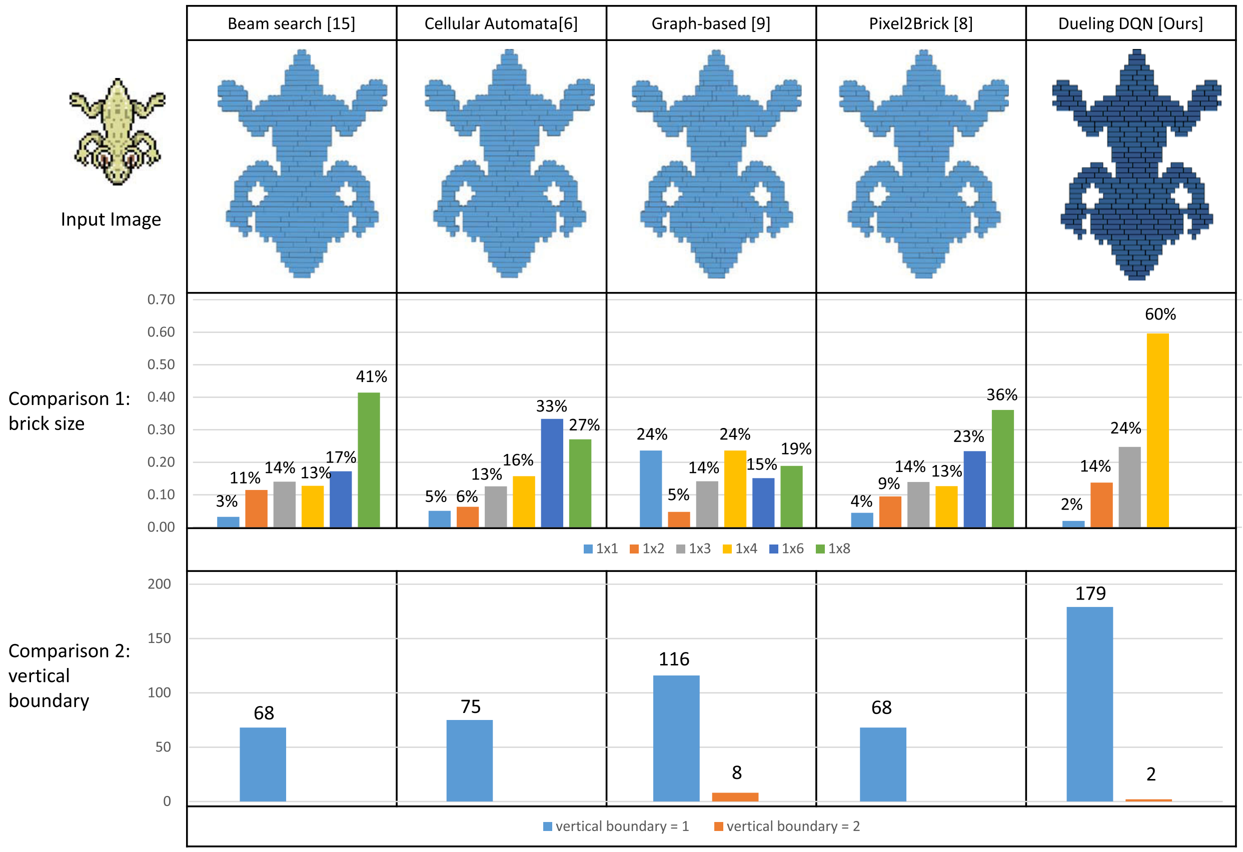


FIGURE 20. The first comparison of our result with existing studies. We compare five existing studies including beam search [15], cellular automata [6], graph-based method [9], pixel2brick [8] and ours. Note that they use different sets of bricks for their models.

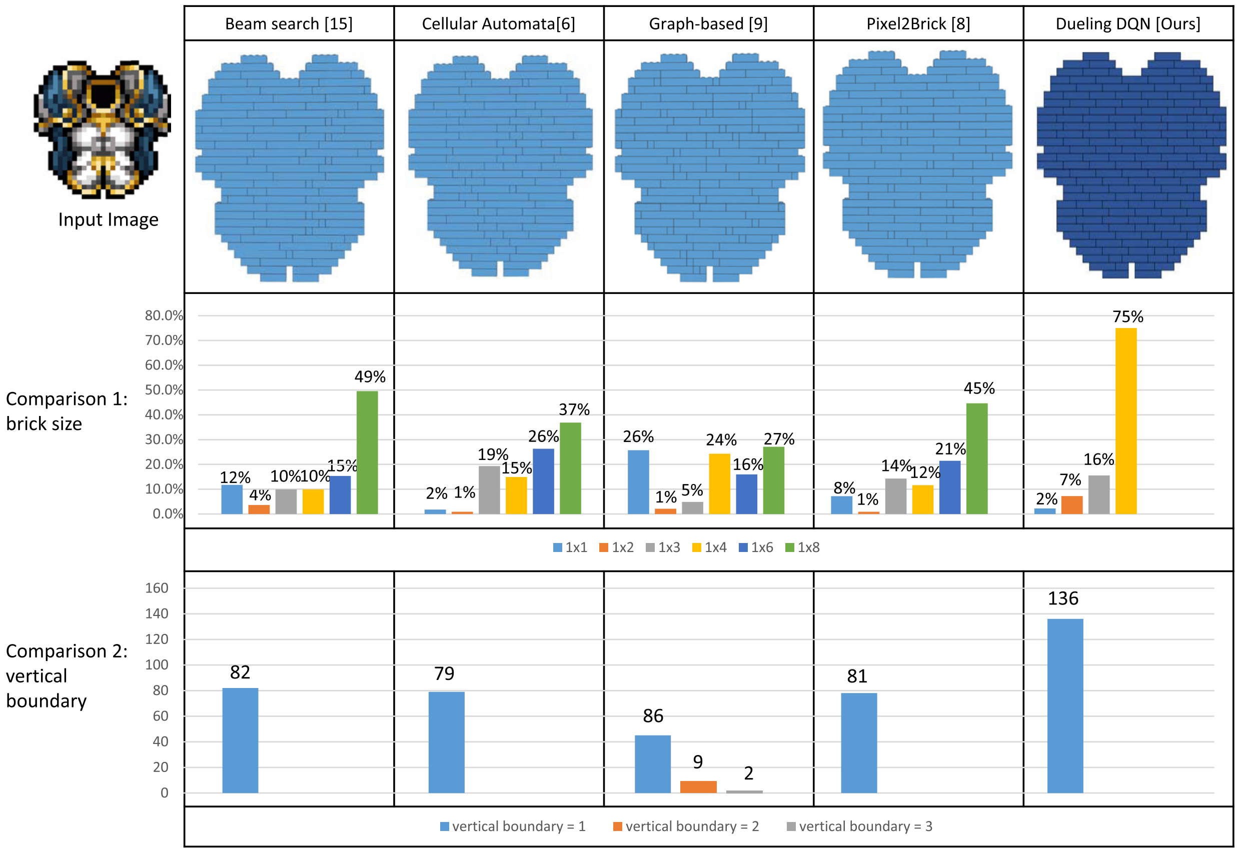
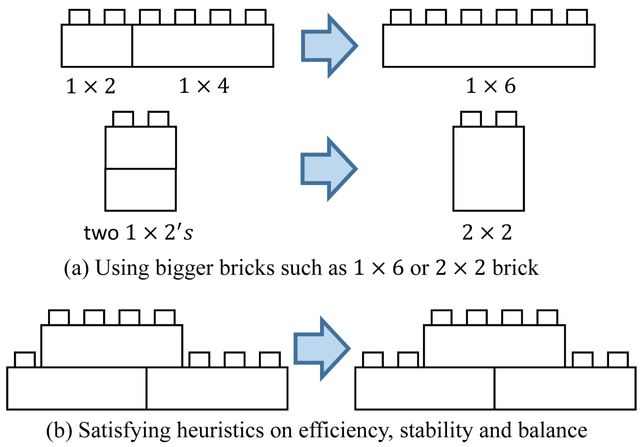


FIGURE 21. The second comparison of our result with existing studies. We compare five existing studies including beam search [15], cellular automata [6], graph-based method [9], pixel2brick [8] and ours. Note that they use different sets of bricks for their models.

TABLE 2. Comparison of computation time: Two existing methods including exhaustive search and heuristic search [3] and two different versions of our method are compared. Our final result is produced by four-brick version.

	Resolution	Type of bricks	Existing methods exhaustive search (sec.)	Existing methods heuristic search [3] (sec.)	Our method (sec.)
(b)	150 × 75	Four bricks	730.2	136.2	7.6
		Six bricks	7459.3	1296.3	68.2
(e)	100 × 100	Four bricks	531.6	102.4	4.9
		Six bricks	6013.2	997.4	45.3
(f)	45 × 46	Four bricks	137.3	25.2	1.2
		Six bricks	1483.8	243.1	10.7



(a) Using bigger bricks such as 1 × 6 or 2 × 2 brick

(b) Satisfying heuristics on efficiency, stability and balance

FIGURE 22. Limitations of our framework.

monochrome Lego layouts for comparison. We used the comparison data in [6], [8], [9], [15] from [8]. The resulting Lego-layouts are illustrated in the top row of Fig. 19 and 20. We compared the results using two heuristics: efficiency and stability.

1) COMPARISON 1: EFFICIENCY

The efficiency for a Lego puzzle is achieved by using larger bricks more than smaller bricks. Because we use only four bricks, such as 1 × 1, 1 × 2, 1 × 3, and 1 × 4, an exact comparison with other studies that also use 1 × 6 and 1 × 8 bricks is not straightforward. However, the distribution of the bricks used in the resulting Lego puzzle indicates that 60% of the bricks used by our scheme are 1 × 4 sized bricks. The distributions of the bricks according to their sizes are illustrated in the middle row of Fig. 19 and 20.

2) COMPARISON 2: STABILITY

The stability of Lego puzzle is estimated based on the vertical boundary. The best Lego puzzle is a puzzle whose vertical boundaries are of length 1. Many of the existing studies, including [6], [8], [15], generate Lego puzzles whose largest vertical boundary is one. Our scheme produced a Lego layout that has 181 vertical boundaries. Of these, only two vertical boundaries are of length 2. Even though our result possesses vertical boundaries of length 2, the stability is not compromised. Therefore, we argue that our scheme produces stable layouts for Lego puzzles. The distribution of vertical boundaries are illustrated in the bottom row of Fig. 19 and 20.

C. LIMITATION

After applying our framework to produce Lego layouts of various resolutions, we found several limitations to our method. First, the simulation on an ROI to determine a proper brick at a POI can be reused for the pixel to the POI. Because the brick on the next pixel is a child node of the brick on the POI, some descendants of the node are traversed during the simulation on its parent node. Therefore, reusing the simulation could greatly improve the legorization performance.

Second, we employed four bricks for the Lego layout, whereas many existing works employed six bricks. Using four bricks, our framework reduces the computational load for building a layout at the sacrifice of increasing the number of bricks in the layout, which damages the efficiency heuristic. Further, bricks lying on multiple layers can be employed to further improve the efficiency (see Fig. 22 (a)).

Third, we employed the efficiency, stability and aesthetics heuristics for our Lego layout. We did not consider the balance of the bricks, which constrains the edge of the bricks to lie on the center of the bricks in the upper layer or lower layer. Balance can improve the visual satisfaction of a Lego layout (see Fig. 22 (b)).

VII. CONCLUSION AND FUTURE WORK

We presented a reinforcement learning-based framework for generating a 2D Lego-compatible puzzle from a pixel art images. Our strategy is to design a game that builds a successful 2D Lego-compatible puzzle. We build a brick layout of the puzzle by searching the space of the layouts using RL.

In future work, we plan to build a framework that learns a high-quality puzzle layout created by an artist. Further, we aim to build a pixelation scheme that produces the input for our framework.

ACKNOWLEDGMENT

(Heekyung Yang and Kyungha Min contributed equally to this work.)

REFERENCES

[1] S.-J. Luo, Y. Yue, C.-K. Huang, Y.-H. Chung, S. Imai, T. Nishita, and B.-Y. Chen, "Legolization: Optimizing LEGO designs," *ACM Trans. Graph.*, vol. 34, no. 6, pp. 1–12, Nov. 2015.

[2] T. Kozaki, H. Tedenuma, and T. Maekawa, "Automatic generation of LEGO building instructions from multiple photographic images of real objects," *Comput.-Aided Des.*, vol. 70, pp. 13–22, Jan. 2016.

[3] K. Min, C. Park, H. Yang, and G. Yun, "Legorization with multi-height bricks from silhouette-fitted voxelization," *KSII Trans. Internet Inf. Syst.*, vol. 12, no. 6, pp. 2782–2805, 2018.

[4] R. Gower, A. Heydtmann, and H. Petersen, "LEGO: Automated model construction," in *Proc. 32nd Eur. Study Group Ind.*, 1998, pp. 81–94.

[5] P. Petrovic, "Solving LEGO brick layout problem using evolutionary algorithms," in *Proc. Norwegian Conf. Comput. Sci.*, 2001, pp. 87–97.

[6] L. van Zijl and E. Smal, "Cellular automata with cell clustering," in *Proc. Automata*, 2008, pp. 425–440.

[7] M. Zhang, J. Mitani, Y. Kanamori, and Y. Fukui, "Blocklizer: Interactive design of stable mini block artwork," in *Proc. ACM SIGGRAPH Posters*, 2014, p. 18.

[8] M.-H. Kuo, Y.-E. Lin, H.-K. Chu, R.-R. Lee, and Y.-L. Yang, "Pixel2Brick: Constructing brick sculptures from pixel art," *Comput. Graph. Forum*, vol. 34, no. 7, pp. 339–348, Oct. 2015.

[9] R. Testuz, Y. Schwartzburg, and M. Pauly, "Automatic generation of constructable brick sculptures," in *Proc. Eurographics*, 2013, pp. 81–84.

[10] S. Ono, A. Andre, and Y. Chang, "Automatic generation of LEGO from the polygonal data," in *Proc. Int. Workshop Adv. Image Technol.*, 2013, pp. 262–267.

[11] S. Lee, J. Kim, J. W. Kim, and B.-R. Moon, "Finding an optimal LEGO brick layout of voxelized 3D object using a genetic algorithm," in *Proc. Genetic Evol. Comput. Conf. (GECCO)*, 2015, pp. 1215–1222.

[12] M. Zhang, Y. Igarashi, Y. Kanamori, and J. Mitani, "Designing mini block artwork from colored mesh," in *Proc. Smart Graph.*, 2015, p. 2.

[13] B. Stephenson, "A multi-phase search approach to the LEGO construction problem," in *Proc. 9th Annu. Symp. Combinat. Search*, 2016, pp. 89–97.

[14] *Reinforcement Learning*. [Online]. Available: https://en.wikipedia.org/wiki/Reinforcement_learning

[15] D. Winkler. *Automated Brick Layout*. Accessed: 2005. [Online]. Available: http://www.brickshelf.com/gallery/happyfrosh/BrickFest2005/automatedbricklayout.pdf

[16] *Nano Block*. [Online]. Available: https://en.wikipedia.org/wiki/Nanoblock

[17] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016.

[18] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015.

[19] H. Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double Q-learning," in *Proc. AAAI Conf. Artif. Intell.*, 2016, pp. 2094–2100.

[20] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2016, pp. 1–21.

[21] M. Bellemare, W. Dabney, and R. Munos, "A distributional perspective on reinforcement learning," in *Proc. Int. Conf. Mach. Learn.*, 2017, pp. 449–458.

[22] M. Hessel, J. Modayil, H. Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," in *Proc. AAAI Conf. Artif. Intell.*, 2018, pp. 3215–3222.

[23] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, "Dueling network architectures for deep reinforcement learning," in *Proc. Int. Conf. Mach. Learn.*, 2016, pp. 1995–2003.

[24] M. Assran, J. Romoff, N. Ballas, J. Pineau, and M. Rabbat, "Gossip-based actor-learner architectures for deep reinforcement learning," 2019, *arXiv:1906.04585*. [Online]. Available: http://arxiv.org/abs/1906.04585

[25] Y. Li, "Deep reinforcement learning: An overview," 2017, *arXiv:1701.07274*. [Online]. Available: http://arxiv.org/abs/1701.07274

[26] V. Mnih, A. Puigdomènech Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," 2016, *arXiv:1602.01783*. [Online]. Available: http://arxiv.org/abs/1602.01783

CHEOLSEONG PARK received the B.S. degree from Sangmyung University, Seoul, South Korea, in 2016, where he is currently pursuing the degree. His research interests include computer graphics, image processing, 3D-mesh processing, and deep learning.

HEEKYUNG YANG received the B.S. and M.S. degrees and the Ph.D. degree in computer science from Sangmyung University, Seoul, South Korea, in 2010, 2012, and 2019, respectively. She is currently an Assistant Professor with the Division of SW Convergence, Sangmyung University. Her main research interests include computer graphics, especially NPR (non-photorealistic rendering), computer vision, and deep learning.

KYUNGHA MIN received the B.S. degree in computer science and engineering from POSTECH, in 1992, the M.S. degree in computer science from KAIST, in 1994, and the Ph.D. degree in computer science and engineering from POSTECH, in 2000. He is currently a Professor with the Department of Computer Science, Sangmyung University, Seoul, South Korea. His main research interests include computer graphics and deep learning.