Research Progress Report

Shaoxuan YUAN

July 2024

1 Introduction

LEGO® wall is a great decoration and can display the shape of many objects. However, to build such a model, LEGO® designers have to spend a significant amount of time and energy, which is highly inefficient. We aim to find a computational strategy to solve this problem.

Challenges. Solving this problem has several challenges. First, to match the shape of an input object, we need to numerate different bricks along the contour to form a search space. Second, we should construct an optimization model and find a solution which matches the shape best and has no conflicts. Moreover, when filling the LEGO® wall, we must make sure each brick is connected, otherwise it cannot be constructed in real world. Finally, physical factors should be taken into consideration to make sure it is stable in real world.

Ideas. In the beginning, we reset the size of input image to 640×540 to make sure the generated model has the same ratio of length to width as the real world. Then, we use OpenCV to obtain the contour of the input image and put it on a grid. Next, we numerate the bricks along the contour and construct a graph structure. To find the set of bricks which matches the input shape best, we use an IP solver to optimize it. When filling the LEGO® wall, we use a method similar to greedy algorithm to make sure each brick is connected.

2 Related Work

In recent years, many work related to LEGO® have been explored. Zhou et al. [3] show a computational method to create LEGO® sketch models. Further, Luo et al. [2] present an optimization process to build stable 3D LEGO® sculptures, and Liu et al. [1] analyse the 3D block-stacking structures and improve their physical stability. Based on these researches, we will develop a strategy to generate stable LEGO® wall models.

3 Task Definition and Methods

Problem definition. The inputs include some simple images which represent the shapes of the objects. We aim to construct LEGO® wall models using a set of LEGO® bricks; see the set in Figure 1. The output models should match the shapes of the objects without any conflicts between bricks. We also need to make sure that the generated models are buildable and stable in real world.

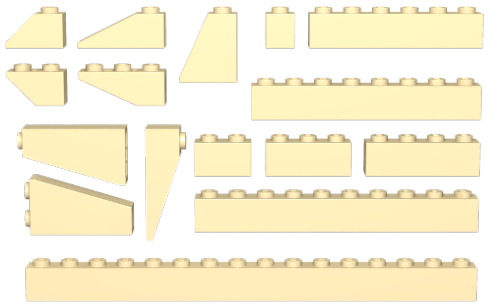

Figure 1: The set of bricks we use

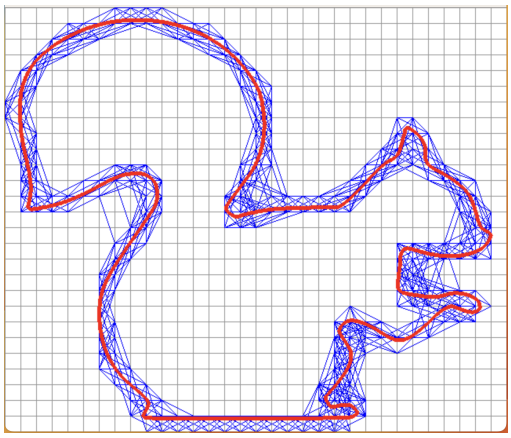

Figure 2: Enumerating candidate edges

Methods. This part shows main methods of generating a LEGO® wall model, which includes three main steps.

Step(1) Graph Creating. After obtaining the contour of an input image, we should find a polygon that matches the shape best. At first, we enumerate sloping and one-

unit edges along the contour to form a set of candidate edges, denoted by $\mathcal{V} = \{s_i\}$; see Figure 2. We also make sure each candidate edge has the same direction as its nearest contour edge. Next, we define $\mathcal{E}$ as the conflict set. We can associate each edge s_i with a brick. In this step, some edges can be associated with two bricks if we only match these edges with the slope edges of the bricks; see Figure 3. Therefore, we use area as our another criterion. For edges which can be associated with two bricks, we calculate the overlap area between these two bricks and the inside of the closed contour respectively and select larger one to fit the contour better; see Figure 4. The edge tuple $(s_i, s_j) \in \mathcal{E}$ if the bricks associated with the edges have intersected part. Then, we construct the graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$.

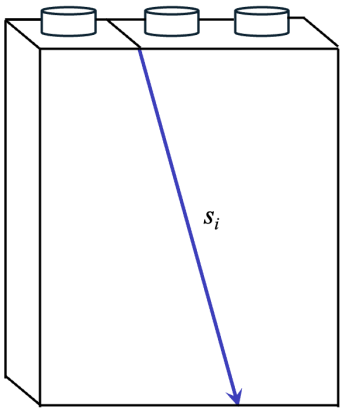

Figure 3: The associated bricks of an edge

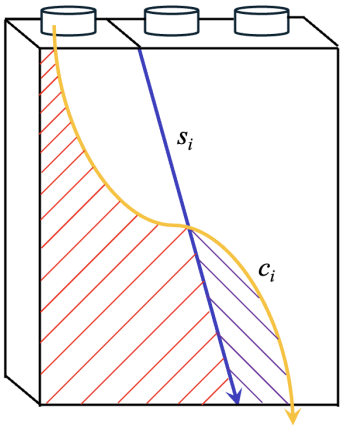

Figure 4: The overlap area of two bricks

Step(2) Graph Optimization. At first, we link each edge $s_i \in \mathcal{V}$ to a binary variable x_i and define two factors to show how well an edge fits the corresponding contour.

- *Distance deviation* L_d. We project s_i onto the contour and find the nearest contour edge, denoted by c_i. Then we obtain sample points on s_i and c_i equidistantly and compute the Chamfer distance between these points.
- *Distance variation* L_v. After sampling points along s_i and c_i, we compute the standard deviation of point-wise distances of these points. It reflects how well edge s_i fits the shape of the corresponding c_i.

Moreover, we construct our objective function:

$$\mathcal{W} = \omega_d \mathcal{L}_d + \omega_v \mathcal{L}_v \tag{1}$$

$\mathcal{L}_d$ equals to the sum of all distance deviation of selected edges and normalize the sum by the contour length. $\mathcal{L}_v$ equals to the sum of all distance variation of selected edges and normalize the sum by the contour length.

Next, we add constraint conditions to our optimization model.

- *Non-empty.* When optimizing the graph, we select at least one edge from $\mathcal{V}$ to make sure the solution is non-empty:

$$\sum_{i=1}^{n} x_i > 0 \tag{2}$$

- *Closure.* For each vertex v in the grid, we should let its in-degree equals to its out-degree to make sure the solution is a closed polygon:

$$\sum_{s_i \in \mathcal{V}_{in}(v)} x_i = \sum_{s_j \in \mathcal{V}_{out}(v)} x_j \tag{3}$$

  where $s_i \in \mathcal{V}_{in}(v)$ if v is the end point of $\vec{s_i}$ and $s_j \in \mathcal{V}_{out}(v)$ if v is the start point of $\vec{s_j}$.

- *Non-conflicts.* For each edge tuple $(s_i, s_j) \in \mathcal{E}$, we cannot select both of them to make sure there is no conflicting bricks:

$$\sum_{(s_i, s_j) \in \mathcal{E}} x_i x_j = 0 \tag{4}$$

- *Non-opposite-direction.* For two edges s_i, s_j with same end points but in opposite direction, we cannot select both of them to make sure all edges have the same direction (clockwise or counterclockwise):

$$\sum_{\vec{s_i} = -\vec{s_j}} x_i x_j = 0 \tag{5}$$

Finally, we apply an IP solver to minimize the objective function and obtain the best polygon; see Figure 5.

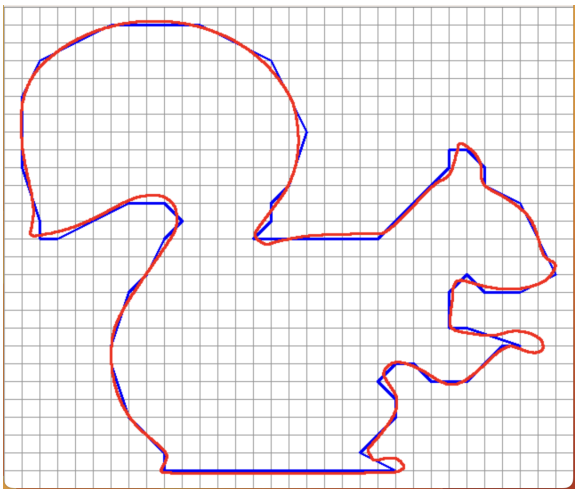

Figure 5: Optimized polygon

Step(3) Bricks Filling. In this step, we select the appropriate bricks to fill the obtained polygon. To achieve this goal, we first place all slope and bottom bricks. Next, we remove all non-vertical edges and sort remaining edges to match them in pairs. Then, we use a strategy similar to greedy algorithm to fill the polygon. We form a set $\mathcal{D}$ containing all cells that have supported points. We construct the model from the bottom to the top line by line. When constructing a line, we give preference to longer bricks. If we find at least one of the cells occupied by a brick is in $\mathcal{D}$, we will put the brick there and add new supported cells to $\mathcal{D}$. If we find it cannot be supported, we will stop and construct next line. Repeat this procedure until all cells are filled; see Algorithm 1.

Algorithm 1: Bricks Filling Algorithm

Input: A list $\mathcal{P}$ of edges of polygon

$\mathcal{D} \leftarrow \{\}$;
$\mathcal{B} \leftarrow$ A list of bricks without slope edges from the set of bricks;
$\mathcal{P} \leftarrow \text{Sort}(\mathcal{P})$;
Place slope and bottom bricks;
Add connected cells generated by slope and bottom bricks to $\mathcal{D}$;
Remove all edges from $\mathcal{P}$ except non-bottom vertical ones;
while $\mathcal{P} \neq \emptyset$ **do**
    $(s_i, s_j) \leftarrow$ pop a pair from $\mathcal{P}$;
    $\mathcal{T} \leftarrow$ A list of the number of different bricks needed to fill the area between s_i and s_j;
    $f \leftarrow 0$;
    for $i = 0$ *to* $|\mathcal{B}|$ **do**
        $brick \leftarrow \mathcal{B}[i]$;
        for $j = 0$ *to* $\mathcal{T}[i]$ **do**
            if *at least one of the occupied cells of brick is in* $\mathcal{D}$ **then**
                Put the brick in this place;
                Add connected cells generated by brick to $\mathcal{D}$;
                $s_i \leftarrow$ the end one-unit vertical edge of brick;
                if s_i *and* s_j *coincide* **then**
                    Remove (s_i, s_j) from $\mathcal{P}$;
                end
            else
                $f \leftarrow 1$;
                break;
            end
        end
        if f *equals to 1* **then**
            break;
        end
    end
end

4 Results

We have used our computational methods to generate some LEGO® wall models; See Figure 6.

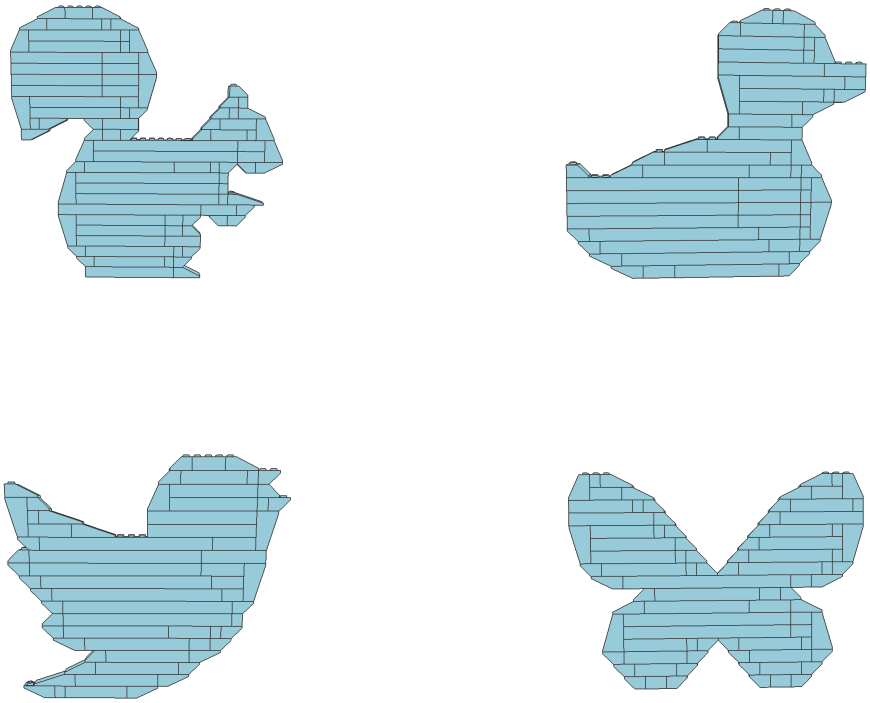

Figure 6: The result models

5 Following Research

In the following research, we will consider the connectivity and stability of the generated LEGO® wall models. We will use force analysis and quantify these physical factors. Then, we will make adjustments to the generated models and make sure its stability in real world. We will also consider using more plate bricks, dealing with multi-colour and multi-contour inputs and adding different directions.

References

[1] Liu, R., Deng, K., Wang, Z., and Liu, C. Stablelego: Stability analysis of block stacking assembly. *arXiv preprint arXiv:2402.10711* (2024).

[2] Luo, S.-J., Yue, Y., Huang, C.-K., Chung, Y.-H., Imai, S., Nishita, T., and Chen, B.-Y. Legolization: Optimizing lego designs. *ACM Transactions on Graphics (TOG) 34*, 6 (2015), 1–12.

[3] Zhou, M., Ge, J., Xu, H., and Fu, C.-W. Computational design of lego® sketch art. *ACM Transactions on Graphics (TOG) 42*, 6 (2023), 1–15.