Supplimentary Document: Structural Concept Learning via Graph Attention for Multi-Level Rearrangement Planning

Manav Kulshrestha, Ahmed H. Qureshi
Department of Computer Science
Purdue University, United States
{mkulshre,ahqureshi}@purdue.edu

1 Data Generation

The basics of data generation are 8 possible object primitives to construct structures. All objects may be present in the top layer in any valid orientation, but middle (or supporting) layers may only contain certain objects in certain orientations conducive to stable structures. For example, an upright pyramid would only provide a top surface of a single line and a cylinder that is not upright would be likely to roll and collapse the whole structure so these are not allowed. Overall, we initially generated 8000 scenes for training and 2000 scenes for evaluation. Our generator was also used to create upwards of 10000 on-the-fly target structures for the evaluation of our pipeline. A rough pseudo code for the structure generation algorithm is specified in Algorithm 1, but the generation heuristic will be open-sourced with the final manuscript. Note that there are some implementation details regarding the algorithm not specified here, like how the placement criteria has a metric involving the area of the top surface of objects, placement pose of objects is found by fitting a plane to sampled points from the top surface of the supporting objects in the lower level, validation of orientation for placement in layers is an exhaustive search, etc.

2 Model Architecture Details and Training

All graph neural networks were implemented using PyTorch Geometric (PyG) [1], and all conventional neural networks were implemented using PyTorch [2].

2.1 Positional Encoding

We utilized the positional encoding implementation specified in [3]. Specifically, the positional encodings b_T^o for some object instance o in the target scene are given by

$$b_T^o = \langle \sin(2^0 A x_o), \cos(2^0 A x_o), \ldots, \sin(2^L A x_o), \cos(2^L A x_o) \rangle \tag{1}$$

where x_o is the 3D position vector, the position associated with o (which we get by calculating the centroid of the point cloud obtained from sampling the known default position, transformed to the target orientation), and the rows of A are the outwards facing unit-norm vertices of a twice-tessellated icosahedron. We use no offset, a scale of 1, a min degree of 0, and a max degree of 5 to calculate L, which results in an encoding of size 511. For more details, please refer to [3].

2.2 PoinNet++ based segmentation

We used PyTorch Geometric's [1] example model for segmentation, as is, without significant changes. The input point clouds from each scene were downsampled to have 1024 points using random sampling, and training was done in a supervised manner using negative log-likelihood loss calculated from the output and the ground truth point identities extracted from the simulation. We trained on a set of 8000 scenes and validated performance on 2000 scenes before use.

Algorithm 1: Generation-Algorithm($B, L^{\{K\}}, \mathcal{O}$). B are the bounds for the scene, $L^{\{K\}}$ specifies the maximum number of objects on each level, K specifies the number of levels, and $\mathcal{O}$ are all possible object instances that can be placed.

$\mathcal{O}_S \leftarrow \text{ValidSubLevelObjects}(\mathcal{O})$
$c_0 \leftarrow 0$ ▷ number of objects placed on level 0
$O \leftarrow \emptyset$ ▷ objects placed
$A^{\{K\}} \leftarrow \emptyset$ ▷ A_i contains all objects available for placement in level i
while $c_0 < L_0$ **do**
  $o \leftarrow \text{RandomlyPick}(\mathcal{O}_\text{S})$
  if $o \in B$ and $\text{NotInCollision}(o, O)$ **then**
    $\text{PlaceObject}(o)$
    $O \leftarrow O \cup \{o\}$
    $A_0 \leftarrow A_0 \cup \{o\}$
    $c_0 \leftarrow c_0 + 1$
$c^{\{N\}} \leftarrow 0$ ▷ c_i is the number of objects placed on level i
$i \leftarrow 1$
while $i \leqslant K$ **do**
  for $(a, b) \in A_{i-1}$ **do**
    $v \leftarrow \text{ValidPlacementObjects}(a, b, O, \mathcal{O})$ ▷ obj instances that can be supported by a, b
    $v \leftarrow \text{CollisionFree}(v, O)$ ▷ filters to give only collision free placements
    for $o \in v$ **do**
      $\text{PlaceOnObjects}(o, a, b)$ ▷ places o to be supported by a, b
      $A_i \leftarrow A_i \cup \{o\}$
      $O \leftarrow O \cup \{o\}$
      $c_i \leftarrow c_i + 1$
      if $c_i < L_i$ **then**
        break
    if $c_i < L_i$ **then**
      break
  for $a \in A_{i-1}$ **do**
    $v \leftarrow \text{ValidPlacementObjects}(a, O, \mathcal{O})$ ▷ object instances that can be supported by a
    $v \leftarrow \text{CollisionFree}(v, O)$ ▷ filters to give only collision free placements
    for $o \in v$ **do**
      $\text{PlaceObject}(o, a)$ ▷ places o to be supported by a
      $A_i \leftarrow A_i \cup \{o\}$
      $O \leftarrow O \cup \{o\}$
      $c_i \leftarrow c_i + 1$
      if $c_i < L_i$ **then**
        break
    if $c_i < L_i$ **then**
      break
  $i \leftarrow i + 1$
return O

2.3 PointNet++ based feature extraction

This utilized PyTorch Geometric's [1] implementation of PointNet++ with an MLP attached at the end to do classification on our set of 8 object primitives. Our MLP used had 3 layers. The first layer had an input size of 1024 and an output size of 512, the second layer had an input size of 512 and an output size of 256, and the final layer had an input size of 256 and an output size of 8. To obtain the object level features $w_T^{\{N\}}$ for each object in the target scene, we remove the final layer and take the 256-sized output to use as the object's latent features. The network was trained using a classification

task for objects in over 800 scenes (each containing an average of 9 objects) and evaluated on objects in over 200 scenes before use.

2.4 GAT Graph Encoder

Our graph encoder g_Φ contains 2 graph attention convolution layers, each of which convolves around every node i in the graph using its neighbor set $\mathcal{N}(i)$. The exact node feature update done by the convolutional layer is given by

$$n_i' = \alpha_{i,i}\Phi n_i + \sum_{j \in \mathcal{N}(i)} \alpha_{i,j}\Phi n_i \tag{2}$$

where Φ is the learnable parameter for the update mechanism and the attention coefficients α, which quantifies the importance of neighboring nodes, is given by

$$\alpha_{i,j} = \frac{\exp(a^T \sigma(\Phi[n_i \,||\, n_j]))}{\sum_{k \in \mathcal{N}(i) \cup \mathcal{N}(j)} \exp(a^T \sigma(\Phi[n_i \,||\, n_k]))} \tag{3}$$

where σ is the non-linear activation LeakyReLU with a slope parameter of 0.2. Each of the graph convolution layers had 16 attention heads with averaging used as the aggregation function. The first graph attention layer has an input size of 511 and an output size of 256 whereas the other graph attention layer has an input size of 256 and an output size of 128. Training was done in a supervised manner with

2.5 MLP Edge Decoder

The edge decoder h_Ψ can be queried with a pair of high-level features z_i, z_j, resulting from the graph network encoder, and will decode them into the structural relationship between them. This relationship of dependence is asymmetric, and the decoder is used to query every ordered pair of nodes in the graph to obtain the respective dependence probabilities $\rho^{\{N \times N\}}$ where $\rho_{i,j} = h([z_i || z_j]; \Psi)$. h_Ψ has 2 fully connected layers and uses LeakyReLU as its non-linear activation function after the first layer only. The first layer has an input size of $2 \cdot 128 = 256$ and an output size of 128, whereas the second layer has an input size of 128 and an output size of 1. We apply SoftMax to get the associated probabilities for training with binary cross-entropy loss. The reason for the input layer for h_Ψ being twice the output size for g_Φ is because finding the existence probability for the edge (i, j) involves concatenating the high-level node features z_i, z_j before inputting them into the edge decoder. The graph encoder g_Φ and edge decoder h_Ψ were trained together in a supervised manner to minimize the binary cross entropy loss between the adjacency matrices of the predicted and ground truth dependency graphs, which, in turn, was obtained from simulation information (specifically, the y-components of the contact force vectors on each pair of objects).

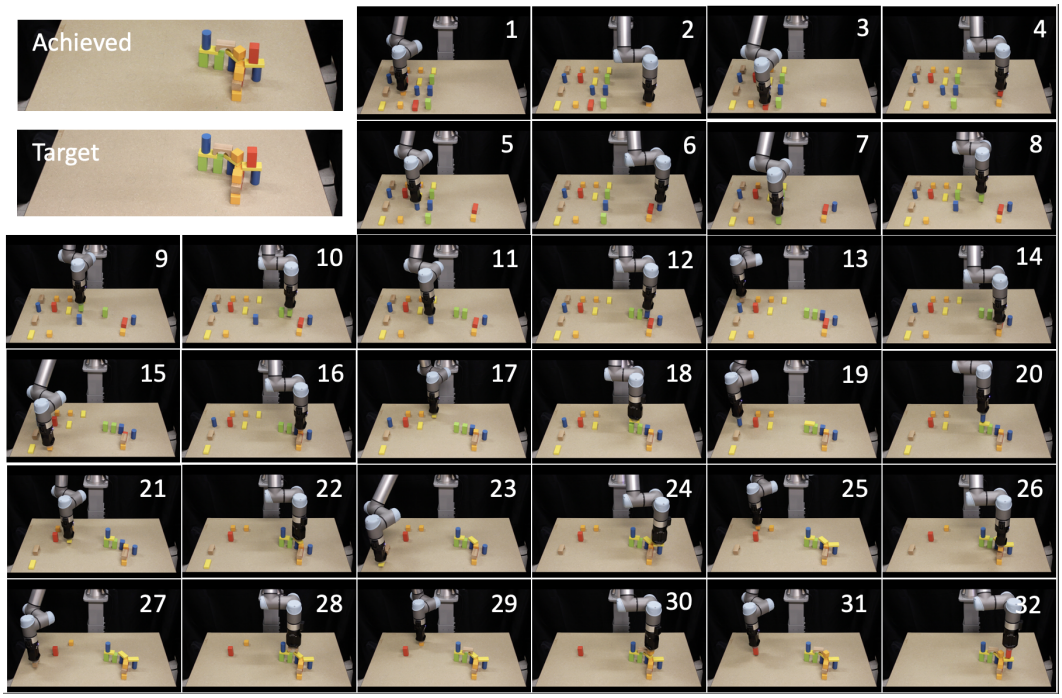

Figure 1: Our approach performing progressive pick-and-place (with all steps shown) actions based on its multi-level rearrangement plan to achieve (top left) a target arrangement (middle left).

References

[1] M. Fey and J. E. Lenssen. Fast graph representation learning with pytorch geometric. *International Conference on Learning Representations (ICLR)*, 2019.

[2] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. *Conference on Neural Information Processing Systems (NIPS)*, 2017.

[3] J. Ortiz, A. Clegg, J. Dong, E. Sucar, D. Novotny, M. Zollhoefer, and M. Mukadam. isdf: Real-time neural signed distance fields for robot perception. *Robotics: Science and Systems (RSS)*, 2022.