

Appendix

Table of Contents

A Object Suggester	13
A.1 Dataset	13
A.2 Implementation Details	14
A.3 Training Details	14
B Learned Placement Suggester	14
B.1 TAXPose-D	14
B.2 Dataset	14
B.3 Training Details	15
C Model Deviation Estimator	15
C.1 Implementation Details	15
C.2 Dataset	15
C.3 Training Details	16
D A* Search	16
D.1 A* expansion	16
D.2 Cost Function	16
D.3 Heuristic and Goal Functions	16
D.4 Multi-goal A*	17
E Experimental Setup	18
E.1 Datasets and Tasks	18
E.2 Evaluation Details	18
E.3 Random Rollouts	18
E.4 Points2Plans Comparison	19
E.5 DP3 Baseline	19
F Experimental Results	20
F.1 Task Execution Success Rate vs. Task Complexity	20
F.2 Qualitative Analysis: Search Graphs	20

A Object Suggester

In our method, the learned object suggester predicts which object(s) in the current scene can be feasibly moved (Sec. 4.1).

A.1 Dataset

The object suggester is trained in a supervised fashion from a transition dataset whose demonstrations are *not* task-specific, so as a result, the model learns which objects in the current scene can be moved feasibly. Each piece of transition data in the dataset D consists of a point cloud observation \mathbf{o} , an action $\langle \mathbf{x}, \mathbf{T} \rangle$ taken on \mathbf{o} , and the resulting next scene point cloud \mathbf{o}' . For table bussing, the dataset consists of 154 transitions, and for block stacking, there are 78.

557 A.2 Implementation Details

558 As shown in Figure 2(a), the object suggester receives a “query mask” as input, which defines the
 559 object \mathbf{x} being input to the function $p_\phi(\mathbf{x}|\mathbf{o})$. To create this input query, we construct a binary mask
 560 over all points in the scene point cloud \mathbf{o} , where points belonging to object \mathbf{x} receive a mask value of
 561 1, while points that do not belong to object \mathbf{x} receive a mask value of 0. This input is a query mask
 562 that essentially asks the object suggester, “Should this object be moved next?” The corresponding
 563 ground-truth label for this query is then 1 since \mathbf{x} is the object being moved in the demonstration.
 564 We also construct query masks for the other objects in the scene, and those examples are labeled 0.

565 At inference time, we iterate over the set of movable objects \mathbf{X} in the scene, and for each object \mathbf{x}_i ,
 566 we construct a corresponding binary mask with which to query the object suggester. For each object
 567 \mathbf{x}_i , the object suggester returns a score $\tilde{p}_{\mathbf{x}_i}$. These queries form a collection of real number outputs,
 568 $\{\tilde{p}_{\mathbf{x}_i} \mid \mathbf{x}_i \in \mathbf{X}\}$. We normalize these outputs into the probability distribution $p_\phi(\mathbf{x})$ that predicts the
 569 likelihood that the robot should move object \mathbf{x} given the current observation.

570 A.3 Training Details

571 The object suggester takes as input a point cloud observation \mathbf{o} of the current scene and a binary
 572 query mask for the object \mathbf{x} we want to query about, i.e., *should the robot move object \mathbf{x} given*
 573 *the current point cloud observation \mathbf{o} ?* The model outputs a real number. We use a PointNet++
 574 network [38] for the architecture. The object suggester in each environment is trained for 2500
 575 steps. We use a batch size of 16 and a learning rate of 1e-3.

576 B Learned Placement Suggester

577 Given an object to move, our method samples multiple task-relevant transformations from a learned
 578 placement suggester (Sec. 4.2).

579 B.1 TAXPose-D

580 Our learned placement suggester is implemented as TAXPose-D [24]. To provide some background,
 581 TAXPose-D relies on a Conditional Variational Autoencoder (cVAE) that maps the input point cloud
 582 \mathbf{o} into a latent distribution $\tilde{p}(z|\mathbf{x}, \mathbf{o})$ over the cloud, conditioned on the object \mathbf{x} to be moved. This
 583 distribution can be normalized using softmax to a probability distribution $p(z|\mathbf{x}, \mathbf{o})$, from which we
 584 sample a latent encoding z that is decoded into a transformation that achieves the relative placement.

585 TAXPose-D represents the latent space as a discrete multinomial distribution over the set of points,
 586 which is beneficial for handling discrete multi-modal placements. For example, to place a cup inside
 587 a bowl as opposed to on top of a plate, this sampling strategy avoids transition regions between dis-
 588 tinct subspaces in the latent space, making it possible to map the sampled points to valid placements.

589 **Iterative rescore.** Applied to our approach, neighboring values of the latent space tend to be
 590 decoded into similar placements. Therefore, to sample diverse and likely placements, we iter-
 591 atively re-score the latent distribution $\tilde{p}(z|\mathbf{o})$ around the previous sample. In iteration i , we compute
 592 the latent distribution $\tilde{p}_i(z|\mathbf{x}, \mathbf{o})$, which is normalized using softmax into probability distribution
 593 $p(z|\mathbf{x}, \mathbf{o})$. Latent encoding z_i is sampled from $p(z|\mathbf{x}, \mathbf{o})$. Then, the latent distribution is re-scored
 594 according to:

$$\tilde{p}_{i+1}(z|\mathbf{x}, \mathbf{o}) = p_i(z|\mathbf{x}, \mathbf{o}) \cdot \frac{\|z - z_i\|^2}{\max_z \|z - z_i\|^2},$$

595 which reduces the probability of sampling neighboring embeddings. This process is repeated in the
 596 next iteration.

597 B.2 Dataset

598 In our framework, all objects are considered rigid and move solely under rigid transformations.
 599 This allows us to interpret any multi-object rearrangement task as a sequence of relative placements
 600 between the object being moved and the surrounding scene. Moreover, given a point cloud scene,

we can map any action on the scene with the pair $\langle \mathbf{x}_j, \mathbf{T}_j \rangle$, where transformation \mathbf{T}_j is applied to object \mathbf{x}_j .

We train our placement suggerter over demonstrations of each task. The demonstrations are given in the form of a sequence of RGB-D images recorded from two cameras together with the semantic names of the objects in the scene. The initial observation is converted into a segmented point cloud using the extrinsic camera values and Grounded Segment Anything [32].

With two cameras, we can only get partial observations of the scene, which can lead to challenges, for example losing track of the positions of objects when they move to occluded parts of the scene. To avoid this, we collect all the subsequent data for training by transforming each object’s initial point cloud by the transformation applied at that time step in the demonstration.

B.3 Training Details

We use a batch size of 4 and a learning rate of $1e-4$. The size of the input point cloud is 1024. In general, we use the same set of standard hyperparameters from TAXPoseD [24] to train each of our placement suggesters. The placement suggesters were trained for 330k steps, 310k steps, and 170k steps for the block stacking, constrained packing, and table bussing tasks, respectively.

C Model Deviation Estimator

The model deviation estimator guides search away from unlikely transitions by predicting the deviation between a planned action in the search tree and its actual execution (Sec. 4.3).

C.1 Implementation Details

To supervise the MDE training, we collect an offline transition dataset from robot-executed plans and label it, similarly to previous work [4]. Each transition in the dataset consists of a point cloud observation \mathbf{o} and a suggested action $\mathbf{a} = \langle \mathbf{x}, \mathbf{T} \rangle$. Applying the transformation \mathbf{T} to object \mathbf{x} produces the expected next state \mathbf{o}' .

To label this transition with the ground-truth deviation, we execute the action and observe the actual next point cloud $\tilde{\mathbf{o}}$. Then, we calculate the ground-truth label for this transition $(\mathbf{o}, \langle \mathbf{x}, \mathbf{T} \rangle)$ by computing the sum of object-wise Chamfer distances between the expected next point cloud and the observed next point cloud, divided by the sum of object-wise Chamfer distances between the expected next cloud and the initial point cloud:

$$\sum_{i=1}^M \frac{CD(\mathbf{o}'_{\mathbf{x}_i}, \tilde{\mathbf{o}}_{\mathbf{x}_i}) + \varepsilon}{CD(\mathbf{o}'_{\mathbf{x}_i}, \mathbf{o}_{\mathbf{x}_i}) + \varepsilon}, \quad (1)$$

where i indexes each object in the scene, $\mathbf{o}_{\mathbf{x}_i}$ is the point cloud of object \mathbf{x}_i , and $CD(\cdot, \cdot)$ is the Chamfer distance. The numerator measures how different the observed next state is from the expected next state. The denominator normalizes the deviation by measuring the expected amount of change from the initial state. To avoid division by zero for static objects, we add a small ε .

If the deviation is high, then it is likely that the next observed state will not match that proposed by the placement suggerter, so we want to direct A* search away from such child nodes.

C.2 Dataset

We collect an offline transition dataset and label it with ground-truth deviation values for supervised learning of the model deviation estimator (MDE).

We collected 510 transitions in the simulation block stacking environment. This was done by repeatedly generating a random initial scene, sampling $k = 5$ suggestions for each object from the learned placement suggerter, and rolling each of those suggestions out from the initial scene for 1 time step. We did not train an MDE for the constrained packing tasks.

For the table bussing tasks, we collected 126 transitions in the real world. This was done by resetting the scene to some initial state, generating a plan that satisfies the goal using our main method, and then rolling out the plan for as many steps as possible (until either task execution success or failure).

MDE Hyperparameters		
	Block Stacking	Table Bussing
clip_max	3.2	5000
ε	1	0.01

Table 4: Hyperparameters used in training the model deviation estimators.

645 C.3 Training Details

646 We use a PointNet++ network [38] for the MDE architecture. We use a batch size of 16 and a
647 learning rate of 1e-3. We train each MDE for 3000 steps. The point cloud inputs to the model are
648 mean-centered on the points belonging to movable objects in the scene.

649 The calculated ground-truth deviations may cover a wide range of values and contain outliers, so
650 we standardize the data by clipping large outliers to a hyperparameter clip_max and then use Min-
651 MaxScaler from scikit-learn to transform the values to the range [0, 1]. See Table 4 for specific
652 values of hyperparameters used, including the values used for ϵ from Equation 1.

653 D A* Search

654 Our method leverages A* search to search over the space of possible object rearrangements
655 (Sec. 4.4).

656 D.1 A* expansion

657 Starting from the root node, our method proposes $b = k \cdot M$ child nodes, where M is the number
658 of objects in the scene. For each object in the scene, a placement suggerer suggests k child nodes.
659 We use $k = 10$ for block stacking, $k = 5$ for constrained packing, and $k = 3$ for table bussing
660 experiments.

661 After expanding the root node, moving the same object consecutively is equivalent to composing
662 both transformations. Based on this fact and to avoid infinite loops, the last object moved is excluded
663 from generating new suggestions in the next expansion.

664 D.2 Cost Function

665 **Action cost.** In both environments, we set the action cost to be 0.01.

666 **Collision cost.** If the action leading to the node is $\langle \mathbf{x}_n, \mathbf{T}_n \rangle$, we reason about collisions of object
667 \mathbf{x}_n with other movable objects in the scene during “picking” motions and “placing” motions. Con-
668 cretely, we voxelize the scene and compute the percentage of voxels belonging to object \mathbf{x}_n that
669 overlap with the other objects in the scene: $C_c(n) = \text{voxel_overlap}(\mathbf{o}_n, \mathbf{x}_n)$

670 **Deviation cost.** This deviation is predicted by a learned model deviation estimator (Sec. 4.3) as:
671 $C_d(n) = \delta(\mathbf{o}_n, \mathbf{a}_n)$.

Probability cost. This is derived from the outputs of the learned placement suggerer $p_\theta(\mathbf{T}|\mathbf{x}, \mathbf{o})$
and the object suggerer $p_\phi(\mathbf{x}|\mathbf{o})$. The likelihood of the action $\mathbf{a}_n = \langle \mathbf{x}_n, \mathbf{T}_n \rangle$ is given by $p_\theta(\mathbf{T} = \mathbf{T}_n | \mathbf{x}_n, \mathbf{o}_{n_p}) p_\phi(\mathbf{x} = \mathbf{x}_n | \mathbf{o}_{n_p})$, where \mathbf{o}_{n_p} is the point cloud of the parent node n_p . Therefore, we discourage lower likelihood actions by defining:

$$C_p(n) = 1 - p_\theta(\mathbf{T} = \mathbf{T}_n | \mathbf{x}_n, \mathbf{o}_{n_p}) p_\phi(\mathbf{x} = \mathbf{x}_n | \mathbf{o}_{n_p}),$$

672 where w_c, w_d and w_p are the weights assigned to the collision, deviation, and probability costs,
673 respectively.

674 D.3 Heuristic and Goal Functions

675 D.3.1 Heuristic Functions

676 In each of the environments, we calculate the heuristic $h(n)$ directly from the point cloud observation
677 corresponding to node n , without access to ground-truth information such as object poses. Instead,
678 we use the point cloud and object segmentation to estimate the object poses.

679 In the block stacking environment, the heuristic function is a discrete value indicating how many of
680 the blocks are “out of place” relative to the goal configuration specified by the goal function. For
681 example, say the goal is to stack the blocks in red-green-blue order from top-to-bottom, and all three
682 blocks are unstacked on the table. Then the heuristic value for this state is 2, since both the green and
683 blue blocks are not in place. If the robot then stacks the green block on top of the blue block, then
684 the heuristic value decreases to 1, since now only the red block is out of place. To determine whether
685 the blocks are in the desired relative positions to satisfy the goal function, we estimate the pose of
686 each block from its point cloud and then check their heights and whether their (x, y) coordinates are
687 aligned, within a certain error threshold.

In the table bussing environment, the heuristic function is a continuous value indicating how far the
objects are from a target point. The target point is selected as the center of either one of the plates’
point clouds. Each object in the scene also has a specified center, calculated as:

$$c = \frac{\max(x, y) + \min(x, y)}{2}$$

where x, y are the coordinates of the object’s point cloud in the world frame. The heuristic for node
 n is therefore defined as the sum of the Euclidean distances of each object’s center to the target point
on the XY plane, as:

$$h(n) = \sum_{i=1}^M c_i(x, y)$$

688 where M is the number of objects in the scene.

689 We ensure that the heuristic function underestimates the distance from the current node to the goal,
690 so any goal state will have heuristic value 0. Since heuristics are not the main focus of this paper,
691 we leave the task of learning these heuristic functions from demonstration data to future work.

692 D.3.2 Goal Functions

693 In the block stacking environment, the goal function is closely related to the A* heuristic function;
694 namely, a node n whose corresponding point cloud observation \mathbf{o} satisfies the goal function $\mathcal{G}(\mathbf{o}) =$
695 1 if and only if it has $h(n) = 0$.

696 In the table bussing environment, the goal function checks if all of the objects are stacked on top
697 of a plate. This is done by checking if the Euclidean distance in the XY plane from each object’s
698 center to the reference plate’s center is below a certain threshold.

699 D.4 Multi-goal A*

700 Rather than terminating search immediately upon finding a state that satisfies the goal condition, we
701 let search continue until one of the following occurs: 1) the open list is empty, 2) the node expansion
702 limit is reached, or 3) m goals have been found, where m is a hyperparameter. During search, we
703 record all goal nodes: nodes that satisfy the goal condition $\mathcal{G}(\mathbf{o}_n) = 1$. After search terminates, we
704 select one of the goal nodes using a score function and output the corresponding path.

705 Once selected, we backtrack from the goal node n_g to the initial node n_1 , which gives us a sequence
706 of actions \mathbf{a} that constitute the output plan $P = \{\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_{T-1}\}$. We then attempt to execute this
707 plan with a robot manipulator. If A* search exceeds a maximum node expansion budget, we say that
708 no plan is found and terminate the search.

709 We allow A* to search until 1) the open list is empty, 2) the node expansion limit is reached, or
710 3) m goals have been found, where m is a hyperparameter. This outputs multiple possible goal
711 nodes, from which we select one using a score function and output its corresponding path. In real-
712 world table bussing, we use $m = 10$, and the score function selects the path with the lowest sum
713 of collision costs. In simulation, we use $m = 1$, and the path whose goal node has the best block
714 stacking alignment is chosen.

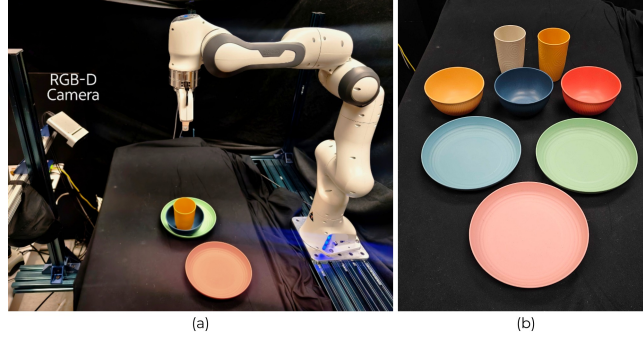


Figure 7: **Real-world setup.** Left: Our real-world setup of the table bussing environment with the RGB-D camera, Franka arm, and a set of objects. Right: All of the objects (plates, bowls, and cups) seen in the table bussing environment.

E Experimental Setup

E.1 Datasets and Tasks

Block Stacking (Simulation). Our demonstration data for this task consists of 96 transitions $(\mathbf{o}_j, \mathbf{a}_j)$ collected using two depth cameras in PyBullet [39]. We train and test on a version of the environment with three blocks.

Constrained Packing (Simulation). Our demonstration data consists of 8000 transitions from a dataset containing 4-step task demonstrations. This dataset was released by the authors of Points2Plans [8].

Table Bussing (Real World). Our demonstration data for this task consists of 156 transitions collected using an RGB-D video camera in the real world (Figure 7). We then use Grounded Segment Anything [32] and CoTracker3 [33] to extract the transitions $(\mathbf{o}_j, \mathbf{a}_j)$ from the video demonstrations. We evaluate our method on two versions of this environment: 1) two plates, a bowl, and a cup, 2) one plate, two bowls, and a cup.

E.2 Evaluation Details

Simulation block stack experiment results are based on a suite of 23 different initial scene configurations, with runs over five different seeds. The evaluation results are averaged and show a 95% confidence interval. Real-world table bussing experiment results are based on a suite of 14 different initial scene configurations. These configurations range in complexity from 2 to 5 steps, where a step refers to a single object reconfiguration. Figure 4 illustrates examples of 4-step, 3-step, and 2-step configurations in both simulation and real world settings.

For real-world experiments, we use Contact-GraspNet [34] to find appropriate grasps and then execute the transformations with pick-and-place motion primitives. These execution methods can find a robot path based on the current point observation and the predicted action, without any extra ground-truth information.

E.3 Random Rollouts

One might wonder whether the observed improvement with the search method merely results from the number of evaluated nodes. To investigate this, we replace A* search with a series of random rollouts. This method chooses a child node uniformly at random from $b = k \cdot M$ possible child nodes, where M is the number of objects. Compared to Beam Search, this method should see more diverse expansions.

In this ablation study, we allow the Random Rollouts method to continue rolling out action trajectories until the total number of expanded nodes across all of the rollouts exceeds the node expansion budget of our main method. Each rollout is allowed to continue to a maximum path length (6 steps). For block stacking tasks, we allow the search to expand a maximum of 200 nodes. We choose to

normalize random rollouts by the total number of node expansions since we did not optimize our code in terms of wall-clock runtime – we leave this to future work. As a reference, our method takes between 1 and 113 seconds to plan for a single task, with an average planning time of 27 seconds per task.

E.4 Points2Plans Comparison

Constrained Packing. We define success in the constrained packing task to be when all of the objects are arranged on top of the shelf surface without being stacked on top of each other. This is a modified but similar version of the success metric used in Points2Plans [8], since their metric depends on goal predicates and feasibility-related predicates – for example, a predicate that determines when an object’s placement location is blocked by another object that has already been placed. Our goal function cannot mimic theirs exactly since we do not use relational abstractions; as a result, ours differs in that it permits placing objects behind other objects, as long as the already-placed objects are not disturbed.

Evaluation. The authors of Points2Plans [8] have only released 5 example scenes from their test dataset, so we run each of the scenes with 100 different random seeds and report the success rates for these 500 resulting runs in Figure 5.

We also modify the action primitives from Points2Plans [8] to better align with our method. Specifically, rather than using MoveIt’s inverse kinematics, we compute the Jacobian of the delta pose between the current pose and a pre-defined grasp pose of a block. We then use this Jacobian to apply joint velocities to the manipulator. This process is repeated inside a control loop until the robot reaches the grasp pose. We leverage this control loop to try each available grasp pose until the block is grasped successfully. Since grasping is orthogonal to our contribution, we employ a model similar to a suction gripper: when the manipulator reaches within a delta error of the block’s grasp pose, we attach the block to the gripper and immediately close the fingers to ensure a stable grasp.

E.5 DP3 Baseline

In Table 2, we compare the execution performance of our method with 3D Diffusion Policy (DP3) [35] as a baseline.

Dataset. We train DP3 for a block stacking task, where the goal is to stack the blocks in red, green, blue order from top to bottom. There is only one goal configuration in the demonstrations since DP3 is not goal-conditioned. The training dataset has 23 initial states where the red, green, and blue blocks are arranged in various positions, either stacked or unstacked in different configurations. We collected 23 expert demonstrations in the PyBullet simulator [39], utilizing Pybullet’s inverse kinematics tool and collision checker. For each movement of a block, 240 observations and state-action pairs are collected. The observations are the dense, segmented point clouds of the blocks. The states and actions consist of the end effector’s x, y, z coordinates, quaternion, and the gripper widths.

Training. The DP3 agent was trained with an action execution horizon of 4 and an observation history of 2. All training parameters are the same as those in the original DP3 model [35].

Evaluation. Since DP3 is not goal-conditioned, the goal configuration at evaluation time is consistent with the one seen in the training dataset. For a fair comparison with our approach, we thus evaluate both DP3 and our method on five different tasks consisting of five different initial blocks configurations with the same goal.

The evaluation metrics are the success rate and the task completion rate. For each initial configuration in the test dataset, success rate is the total number of times the DP3 agent successfully stacks all three blocks in the red-green-blue order divided by the total number of trials. For each initial configuration, 2 to 4 steps are required to complete the task. Each step involves stacking or unstacking a block. The task completion rate is the average number of steps completed divided by the total number of steps required for an initial configuration.

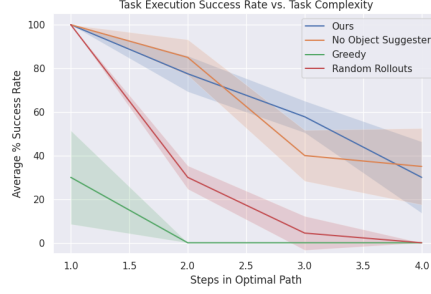


Figure 8: Task execution success rate as a function of task complexity in the simulation block stacking environment. Results are averaged over 5 seeds and show a 95% confidence interval.

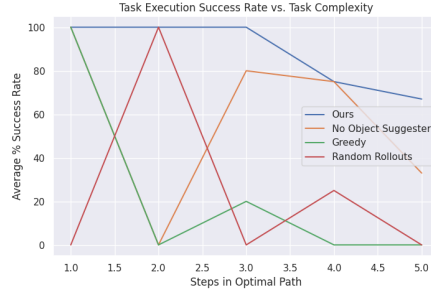


Figure 9: Task execution success rate as a function of task complexity in the real-world table bussing environment.

F Experimental Results

F.1 Task Execution Success Rate vs. Task Complexity

Figures 8 and 9 show task execution success rates of our method and ablations as a function of task complexity in simulation and in the real world, respectively. These reinforce the conclusions that:

- The Greedy ablation is too myopic to achieve much success on tasks that require more than one step to a goal configuration.
- The Random Rollouts ablation achieves some success, but its performance rapidly deteriorates as task complexity increases beyond two steps.
- Our main method outperforms the ablation with no object suggester.

F.2 Qualitative Analysis: Search Graphs

We visualize 2 types of graphs for our method:

- 1) An expanded graph that marks all the nodes expanded during search. Plans found inside this graph are marked via green edges. Figures 10 through 12 show expanded graphs for three different table bussing initial configurations.
- 2) A plan graph that visualizes different paths to goal configurations found during multi-goal A* search. Figures 13 through 15 show plan graphs for three different table bussing initial configurations.

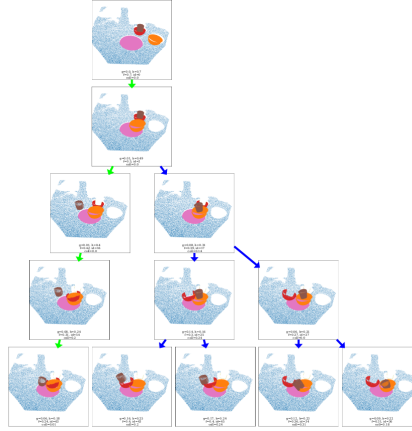


Figure 10: **Expanded graph example 1.** The graph represents the expanded nodes for a 2-step table bussing configuration where a bowl is on a plate and a cup is inside a bowl on the table. A plan is found that first moves the cup to the table, then stacks the bowls, and places the cup on the plate. The plan found is marked with green colored edges



Figure 11: **Expanded graph example 2.** The graph represents the expanded nodes for a 4-step table bussing configuration where a cup is initially placed on a plate. A plan is found that first moves the cup to the table, then stacks both the bowls, and then moves the cup back. The plan found is marked with green colored edges

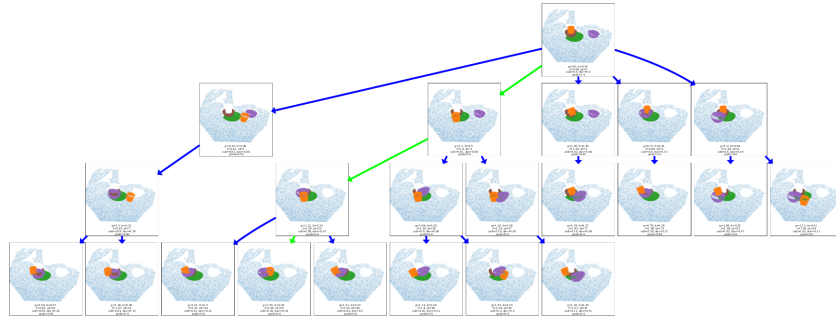


Figure 12: **Expanded graph example 3.** The graph represents the expanded nodes for a 3-step table bussing configuration where a cup is initially placed inside a bowl on top of the plate, and a separate bowl is placed on the table. A plan is found that first moves the cup to the table, stacks the remaining bowl from the table on top of the bowl already on the plate, then moves the cup back inside the bowls. The plan found is marked with green colored edges

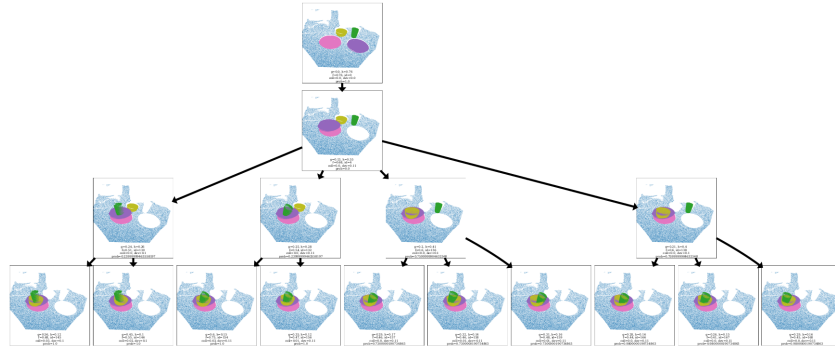


Figure 13: **Plan graph example 1.** The graph represents all of the plans and goals found for a 3-step table bussing configuration with multi-goal A* where 2 plates, a bowl and a cup are placed separately on the table. We see in this graph that A* search finds **multimodal** paths to achieve the goal. It may either produce a plan that places the cup on the plate first and then stacks the bowls, or it may stack the bowls first and then place the cup inside the bowls.

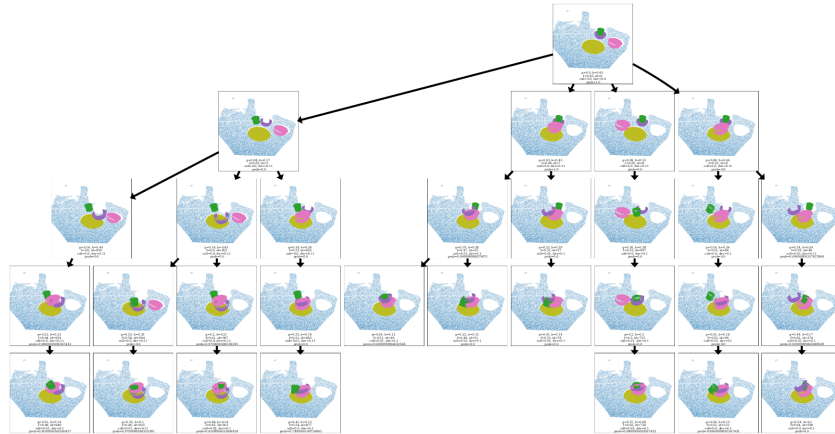


Figure 14: **Plan graph example 2.** The graph represents all of the plans and goals found for a 3-step table bussing configuration where a plate is placed on the table, alongside a cup inside a bowl, with another bowl placed separately on the table. We see in this graph that A* search finds **multimodal** paths to achieve the goal. It may either stack move the cup inside the bowl to the table first, and then stack the bowls before placing the cup back inside. Or, it may first move a bowl on top of the plate, move the cup to the table, and stack the bowls before placing the cup back inside. Illegal plans that move the bowl when a cup is on top of it are not chosen due to their high collision score.

