

Appendices to “Learning Long-Horizon Action Dependencies in Sampling-Based Bilevel Planning”

Anonymous Author(s)

A Additional Experimental Details

For the purpose of reproducibility, this appendix contains additional details of the runtime, domains, baselines, and ablations for the experiments in [Section 5](#) in the main paper.

A.1 Runtime

All experiments were run on Ubuntu 22.04.3 using 20 cores of Intel Xeon Gold 6248 and a 32GB Nvidia Volta V100. We use 30 threads to collect the data for our method and its ablations in parallel. For the 3D domains, we use PyBullet for collision checking and rendering.

A.2 Detailed Domain Descriptions

This section provides additional details of the domains.

- *Shelves Domain*: The cover must be placed after the boxes are on the shelves. The lifted abstract actions are “Move Box”, “Move Cover to Top” and “Move Cover to Bottom”. All lifted abstract actions share a single controller, whose parameters are some $\langle x, y \rangle$ absolute position of a point on the moved and on the target object, and the $\langle x, y \rangle$ offset with respect to the target destination to place the moved object on. The positions of the objects are in absolute coordinates, and the objects cannot be rotated.
- *Donut Domain*: The lifted abstract actions are “Move Robot”, “Grasp Donut” “Place Donut in Box”, “Place Donut on Shelf” and “Add Topping to Donut”. The movement, grasp position and placement position in the abstract action controllers are parameterized by the $\langle x, y \rangle$ displacement. All other controller parameters are determined using binary decisions based on a threshold on a real value (e.g. either a top or side grasp, or which topping to add). The topping machines cannot be used if they’re too far away from the robot; a similar restriction applies to the grasping and placement actions. The positions of the objects are relative to the robot, and the objects cannot be rotated. There are 10 varieties of possible toppings in the goals.
- *Statue Domain*: The lifted abstract actions are “Go through Door”, “Go through Door with Statue”, “Grab Statue” and “Place Statue”. The movement controller parameterization specifies the offset $\langle x, y \rangle$ that the robot moves by, and the grab controller specifies a thresholded value for the orientation of the statue (horizontal, vertical) in the 3D axis from the front of the robot (not perpendicular to the world). The positions of objects are in absolute coordinates, and (other than the horizontal/vertical rotations of the statue) the objects cannot be rotated. In training problems, the positions of objects are randomly offset to keep them in distribution for the larger grids of rooms in test problems.
- *Packing Domain*: The blocks initially lie down scattered across the table. The only lifted abstract action is “Place Block”. The controller is parameterized by where along the block to grasp it and where to place the block (upright) relative to the center of the box. The positions of objects are in absolute coordinates, and the rotations are in quaternions.
- *Trays Domain*: The blocks are initially upright scattered across the table. The lifted abstract actions are “Move Block” and a *dummy* “Check Trays” action that confirms (at the end of the plan) if the blocks are in the target configuration—this is necessary for the task planner to find a goal-reaching plan, but prevents the movement actions from knowing the precise target placements. The controller is parameterized by where to place the block (upright) relative to the center of the tray and a one-hot encoding of the tray. The positions of objects are in absolute coordinates, and the rotations are in quaternions.

A.3 Backjumping and Data Collection Ablations

This section describes the ablations used to validate the need for each element of our approach.

- *No backjumping (A1)*: To study the impact of backjumping on our method, this ablation disables backjumping, instead always backtracking only one step. The ablation uses exactly the same trained networks as our full approach.
- *Negative training data from the longest failed trajectory (A2)*: To assess the importance of our data collection scheme, we train our method by generating positive training data from the prefixes of controller plans from the demonstrations dataset \mathcal{D} and negative training data from the longest failed refinements of attempts at running backtracking on the problems from \mathcal{D} .
- *Negative training data from all prefixes of a failed trajectory (A3)*: This ablation is similar to A2, but instead generates negative training data from all prefixes of the longest failed refinement.

A.4 Training the Baselines

Table 1 presents the hyperparameters used when evaluating the baselines and the diffusion-based samplers used in our method (classifier hyperparameters are included in Appendix C).

The hyperparameters of each method were tuned based on its prediction accuracy on the Shelves domain, using a held-out validation dataset of 20% of the data points. We picked the hyperparameters with the lowest validation loss for each method. The SeSamE-based methods (including our own) set $r_{\text{iter}} = 20$ for all domains. Our data collection method from Section 4.3 collects $r_{\text{dp}} = 4000$ data points per the iteration of the data collection loop. To ensure a fair comparison, the Gaussian and diffusion samplers were trained for a comparable amount of time, and the GNN baseline was trained for a similar amount of time as our method (including the data collection procedure).

Table 1: Hyperparameters for the baselines and samplers for our method

Method	Hyperparameter	Value
Diffusion sampler	number of training iterations	10000
	number of diffusion timesteps	100
	hidden layer sizes	2×512
	learning rate	$1e - 4$
Gaussian sampler	regressor hidden sizes	1024×2
	classifier hidden sizes	128×2
	number of training iterations	20000
	learning rate	$1e - 3$
GNN	number of epochs	1600
	number of message passings	3
	hidden sizes (encoders, models, and decoders)	1×512
	learning rate	$1e - 4$

B Data Gathering Illustrative Example

Figure B.1 illustrates an example search tree that the backtracking algorithm could produce during the data collection described in Section 4.3.

C Network Training Setup

In this appendix, for the purposes of reimplementation and reproducibility, we describe the training setup for our transformer-based classifier. We use an encoder-only transformer. We select the hyperparameters for the classifier, summarized in Table 2, based on the accuracy on the final iteration of training in the data collection algorithm from Section 4.3 on the Shelves environment—runs that caused the data collection to take prohibitively long (e.g., because accuracy in early iterations was too low) were terminated early and discarded.

We optimize a standard binary cross-entropy loss with the Adam optimizer, over randomly drawn mini-batches. In addition to the output of the featurizer network, for each token we concatenate

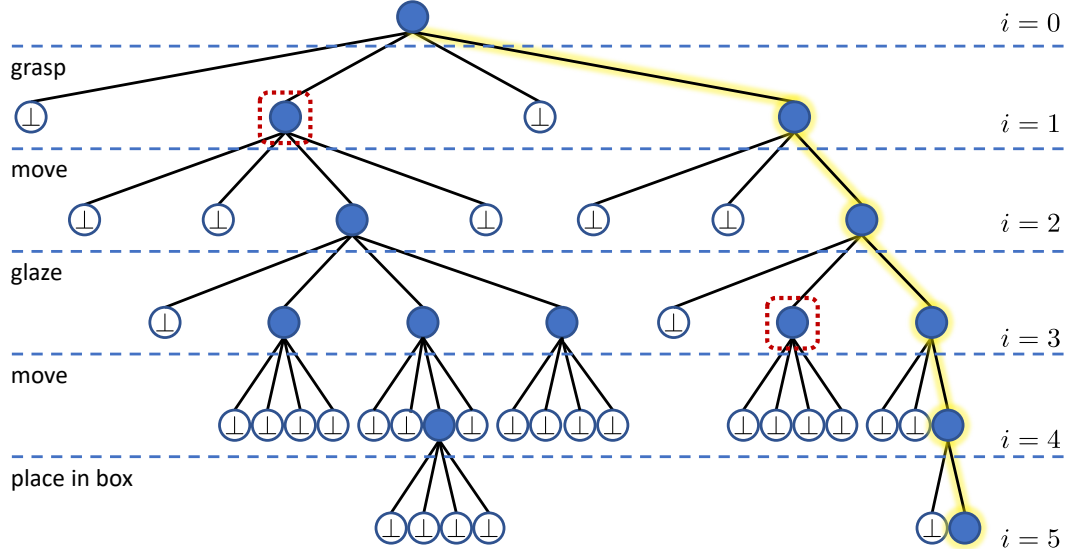


Figure B.1: Example data collection search graph. This is the subgraph of the refinability graph described in Section 3.3 found via backtracking search on the Donut domain, using $r_{dp} = 4$. Rows separated by dashed lines represent layers of the graph (as labeled by i), filled nodes represent continuous states s_i found during backtracking search, and empty nodes labeled by \perp indicate that the sampled parameters resulted in controller failure. The path highlighted in yellow corresponds to a successful execution, so all nodes and corresponding prefixes of the plan are positive samples for h . The two nodes boxed by red dotted lines exemplify the negative samples that lead to refinement failure, as described in Section 4.3.

Table 2: Feasibility classifier training hyperparameters

Hyperparameter	Value
Model learning rate (without the transformer)	$1e - 4$
Transformer learning rate	$1e - 5$
Number of training iterations	5000
Batch size	4000
Featurizer network hidden layer sizes	2×256
Featurizer network output size	256
Sinusoidal embedding dimensionality	128
Sinusoidal embedding base	130
Transformer token width	128
Transformer feedforward block hidden size	512
Number of transformer heads	8
Number of transformer residual blocks	4

489 a one-hot encoding of which featurizer network produced it (identifying the lifted abstract action
 490 and whether the state was produced by its controller, as explained in Section 4.2), and a binary
 491 flag to indicate whether the corresponding ground abstract action was the latest one to be refined
 492 (i.e., the action ω_j such that $j = i$). These additional inputs aid the network with locating the key
 493 information in the sequence of tokens. As is standard practice in the literature, we also concatenate
 494 the sinusoidal positional encoding to each token. As described in [23], we offset the positions of the
 495 tokens by a random value to improve out-of-distribution generalization with respect to the length of
 496 the task plan. Before passing the token to the transformer, we pass it through a linear map to ensure
 497 matching dimensionality. Our transformer then uses multi-head attention: the token is split into a
 498 number of transformer heads, each processing a chunk of the mapped token.

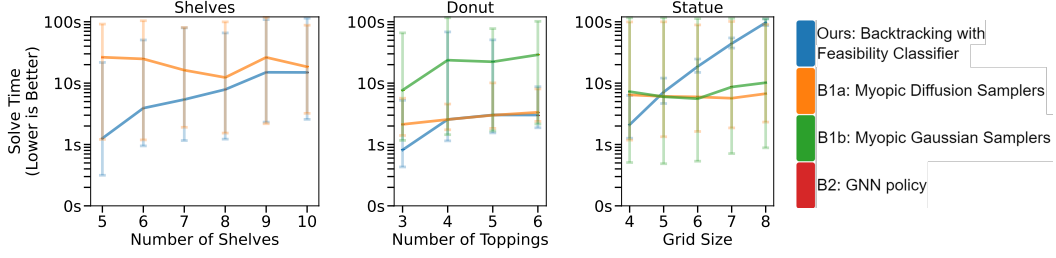


Figure D.2: Solve times for the generalization experiments on 2D domains across varying domain sizes. Other than the Statue domain, our approach is faster than the sampler-based baselines (B1a and B1b) across all environment sizes. We omit the GNN baseline (B2) because it does not do planning and therefore is trivially faster than planning approaches. Averaged across 8 seeds, ranges represent minimum and maximum values. Note that 0% plots are dropped for clarity.

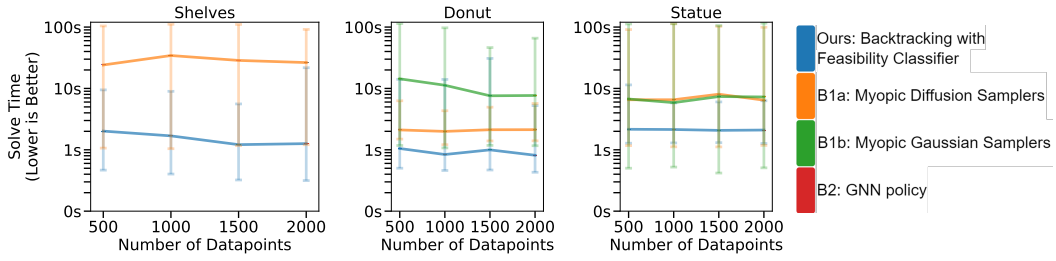


Figure D.3: Solve times for the data efficiency experiments on 2D domains across initial dataset sizes. Our approach is on average faster than the sampler-based baselines (B1a and B1b) across all dataset sizes. We omit the GNN baseline (B2) because it does not do planning and therefore is trivially faster than planning approaches. Averaged across 8 seeds, ranges represent minimum and maximum values. Note that 0% plots are dropped for clarity.

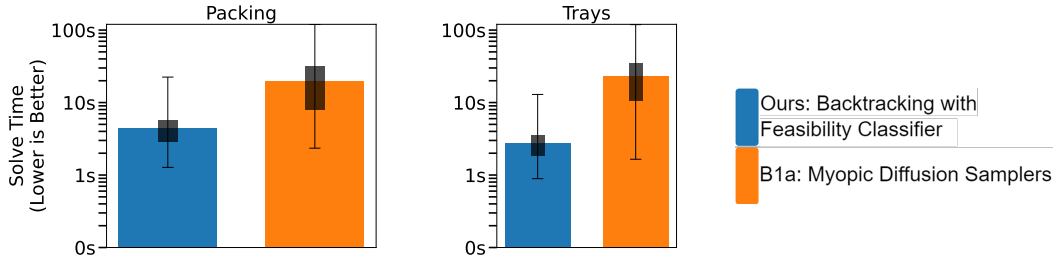


Figure D.4: Solve times for the experiments on the PyBullet domains compared to the best-performing baseline on the 2D environments. Our approach is over $3\times$ faster than the baseline. Averaged across 8 seeds, error bars represent standard deviation, and ranges represent minimum and maximum values.

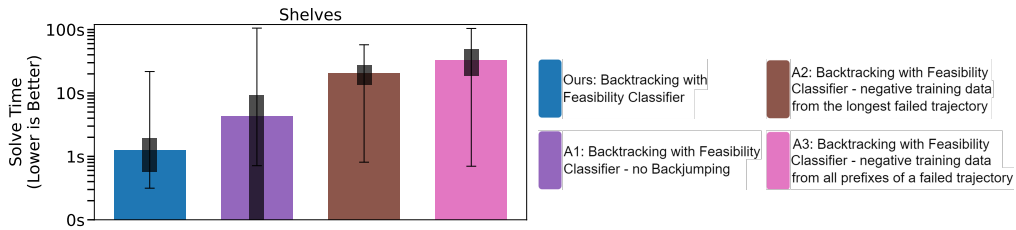


Figure D.5: Solve times for the experiments on the 2D Shelves domain compared to the ablations. Our method is over $3\times$ faster than the ablation of backjumping, and over $10\times$ faster than the ablations of data collection. Averaged across 8 seeds, error bars represent standard deviation, and ranges represent minimum and maximum values.

D Environment Timing Results

In this appendix, we present the timing results of the experiments described in [Section 5.4](#). The timings only consider the tasks for which each baseline succeeded, which is why our method sometimes exhibits a higher runtime than the baselines—recall from [Section 5](#) that the baselines failed to solve many problems due to timeouts. Our method on average performs better than the sampler-based baselines (B1a and B1b; [Figures D.2, D.3, and D.4](#)) and ablations ([Figure D.5](#)) on all experiments except the generalization experiments for the Statue environment. The timings for the GNN baseline (B2) were not included because the policy is designed to compute a single solution; if the solution works, it succeeds, and otherwise it immediately fails. In consequence, when it does succeed, it is much faster ($\sim 1000\times$) than our approach.