A Environment Details

Here we provide detailed descriptions of each of experiment environments. See (§6) for high-level descriptions and the accompanying code for implementations.

A.1 Cover Environment Details

- **Types:**
  - The `block` type has features `height`, `width`, `x`, `y`, `grasp`.
  - The `target` type has features `width`, `x`.
  - The `gripper` type has features `x`, `y`, `grip`, `holding`.
  - The `allowed-region` type, which is used to determine whether picking or placing at certain positions is allowed, has features `lower-bound-x`, `upper-bound-x`.
- **Action space:** $\mathbb{R}^3$. An action (`dx`, `dy`, `dgrip`) is a delta on the gripper.
- **Predicates:** `Covers`, `HandEmpty`, `Holding`, `IsBlock`, `IsTarget`.
- **Contact-related predicates:** `Covers`, `HandEmpty`, `Holding`.
- **Notes:** This extends the environment of [12, 13, 21] to make the robot move in two dimensions. In the previous work, the environment was referred to as PickPlace1D.

A.2 Doors Environment Details

- **Types:**
  - The `robot` type has features `x`, `y`.
  - The `door` type has features `x`, `y`, `theta`, `mass`, `friction`, `rotation`, `target`, `is-open`.
  - The `room` type has features `x`, `y`.
  - The `obstacle` type has features `x`, `y`, `width`, `height`, `theta`.
- **Action space:** $\mathbb{R}^3$. An action (`dx`, `dy`, `drotation`) is a delta on the robot. The rotation acts on the door handle when the robot is close enough to the door.
- **Predicates:** `InRoom`, `InDoorway`, `InMainRoom`, `TouchingDoor`, `DoorIsOpen`, `DoorInRoom`, `DoorsShareRoom`.
- **Contact-related predicates:** `TouchingDoor`, `InRoom`.
- **Notes:** The rotation required to open the door is a complicated function of the door features.

A.3 Stick Button Environment Details

- **Types:**
  - The `gripper` type has features `x`, `y`.
  - The `button` type has features `x`, `y`, `pressed`.
  - The `stick` type has features `x`, `y`, `held`.
  - The `holder` type has features `x`, `y`.
- **Action space:** $\mathbb{R}^3$. An action (`dx`, `dy`, `z-force`) is a delta on the gripper and a force in the z direction (see notes below).
- **Predicates:** `Pressed`, `RobotAboveButton`, `StickAboveButton`, `AboveNoButton`, `Grasped`, `HandEmpty`.
- **Contact-related predicates:** `Grasped`, `Pressed`.
- **Notes:** Picking and pressing succeed when (1) the `z-force` action exceeds a threshold; (2) there are no collisions; and (3) when the respective objects are close enough in `x`, `y` space.

A.4 Coffee Environment Details

- **Types:**
  - The `gripper` type has features `x`, `y`, `z`, `tilt-angle`, `wrist-angle`, `fingers`.
  - The `pot` type has features `x`, `y`, `rotation`, `is-held`, `is-hot`.
  - The `plate` type has feature `is-on`.
  - The `cup` type has features `x`, `y`, `liquid-capacity`, `liquid-target`, `current-liquid`.
- **Action space:** $\mathbb{R}^6$. An action (`dx`, `dy`, `dz`, `dtilt`, `dwrist`, `dfingers`) is a delta on the gripper.

- **Predicates:** `CupFilled`, `PotOnPlate`, `Holding`, `ButtonPressed`, `OnTable`, `HandEmpty`, `PotHot`, `RobotAboveCup`, `PotAboveCup`, `NotAboveCup`, `PressingButton`, `Twisting`.
- **Contact-related predicates:** `Holding`, `HandEmpty`, `CupFilled`, `ButtonPressed`.
- **Notes:** The liquid in a cup increases when a full pot is tilted and close enough to the cup.

B Approach Details

Here we provide detailed descriptions of each approach evaluated in experiments. See (§6) for high-level descriptions and the accompanying code for implementations.

B.1 Bilevel Planning with Neuro-Symbolic Skills (BPNS)

BPNS is our main approach, as described in the main paper.

Planning: The number of abstract plans $N_{\text{abstract}} = 8$ for Cover and Doors, and $N_{\text{abstract}} = 1000$ for Coffee and Stick Button. We would not expect performance to substantially improve for Cover or Doors with a larger N_{abstract}, since we know that the first abstract plan is generally refinable in these environments; the smaller number was selected for the sake of experiments finishing faster. The number of samples per step $N_{\text{samples}} = 10$ for all environments.

Operator Learning: Operators whose skill datasets comprise less than 1% of the overall number of segments are filtered out. This filtering is helpful to speed up learning and planning in cases where there are rare effects or simulation noise in the demonstrations.

Policy Learning: Policies are fully-connected neural networks with two hidden layers of size 32. Models are trained with Adam for 10,000 epochs with a learning rate of $1\text{e}{-3}$ with MSE loss.

Sampler Learning: Following Chitnis et al. [13], each sampler consists of two neural networks: a generator and a discriminator. The generator outputs the mean and diagonal covariance of a Gaussian, using an exponential linear unit (ELU) to assure PSD covariance. The generator is a fully-connected neural network with two hidden layers of size 32, trained with Adam for 50,000 epochs with a learning rate of $1\text{e}{-3}$ using Gaussian negative log likelihood loss. The discriminator is a binary classifier of samples output by the generator. Negative examples for the discriminator are collected from other skill datasets. The classifier is a fully-connected neural network with two hidden layers of size 32, trained with Adam for 10,000 epochs with a learning rate of $1\text{e}{-3}$ using binary cross entropy loss. During planning, the generator is rejection sampled using the discriminator for up to 100 tries, after which the last sample is returned.

B.2 BPNS No Subgoal

BPNS No Subgoal is a variation of BPNS that does not use subgoal parameterization.

Planning: N_{abstract} is the same as BPNS and N_{samples} is not applicable.

Learning: Operator learning is the same as BPNS. For policy learning, for each input $x \circ y$ in the training data, where y is the subgoal, we use x instead, i.e., the state alone. No samplers are learned.

B.3 Graph Neural Network Metacontroller (GNN Meta)

GNN Meta is a mapping from state, abstract state, and goal to a ground skill. This baseline offers a learning-based alternative to AI planning in the outer loop of bilevel planning.

Planning: Repeat until the goal is reached: query the model on the current state, abstract state, and goal to get a ground skill. Invoke the ground skill's sampler up to 100 times to find a subgoal that leads to the abstract successor state predicted by the skill's operator. If successful, simulate the state forward; otherwise, terminate with failure.

Learning: Skill learning is identical to BPNS. This approach additionally learns a metacontroller in the form of a GNN. Following the baselines presented in prior work [13], the GNN is a standard encode-process-decode architecture with 3 message passing steps. Node and edge modules are fully-connected neural networks with two hidden layers of size 16. We follow the method of Chitnis et al. [13] for encoding object-centric states, abstract states, and goals into graph inputs. To get graph

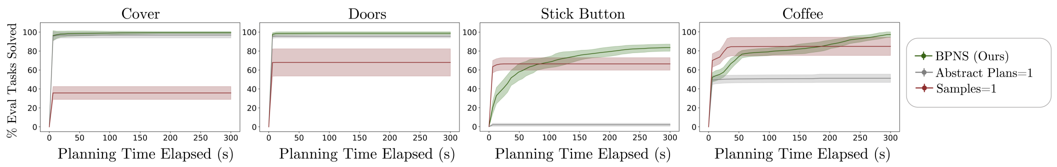


Figure 5: **Planning time analysis**. Evaluation task success rate as a function of planning time elapsed for BPNS and the two ablations. All results are over 10 random seeds and all models are trained on 1000 demonstrations. Lines are means and shaded areas are one standard deviation.

outputs, we use node features to identify the object arguments for the skill and a global node with a one-hot vector to identify the skill identity. The models are trained with Adam for 1000 epochs with a learning rate of $1\text{e}{-3}$ and batch size 128 using MSE loss.

B.4 GNN Meta No Subgoal

GNN Meta No Subgoal is the same as GNN Meta, but with skills learned via BPNS No Subgoal.

B.5 GNN Behavioral Cloning (GNN BC)

GNN BC is a mapping from states, abstract states, and goals, directly to actions. This approach is model-free; at evaluation time, it is queried at each state, and the returned action is executed in the environment. The GNN architecture and training is identical to GNN Meta, except that output graphs consist only of a global node, which holds the fixed-dimensional action vector.

B.6 Samples=1

This ablation is identical to BPNS, except with $N_{\text{samples}} = 1$ during planning.

B.7 Abstract Plans=1

This ablation is identical to BPNS, except with $N_{\text{abstract}} = 1$ during planning.

C Additional Results

Here we present additional results to supplement the main results in (§6).

C.1 Planning Time Analysis

Figure 5 reports evaluation task success rate as a function of planning time for BPNS, Samples=1, and Abstract Plans=1. In Cover and Doors, performance peaks within the first few seconds of wall-clock time. This is consistent with our finding that the first abstract plan is generally refinable in these two environments. In Stick Button and Coffee, performance increases more gradually for BPNS over time. Furthermore, given a small time budget, the Samples=1 ablation sometimes solves more evaluation tasks than BPNS. In these two environments, the first abstract plan is typically not refinable; BPNS exhaustively attempts to sample that abstract plan and others before arriving at a refinable abstract plan. With Samples=1, the unrefinable abstract plans are quickly discarded after one sampling attempt. The gap that later emerges between BPNS and Samples=1 is due to tasks where one sampling attempt of a refinable abstract plan is not enough. This trend is fairly specific to the details of our bilevel planning implementation. Other search-then-sample TAMP techniques that do not exhaustively sample abstract plans before moving onto the next one may converge faster [1].

C.2 GNN Meta Additional Analysis

We were initially surprised by the poor performance of GNN Meta in Stick Button and Coffee, given that the target functions are intuitively straightforward. In Stick Button, the model should be able to use the positions of the buttons in the low-level state to determine whether they can be directly reached, or if the stick should be used instead. In Coffee, the model should use the rotation of the

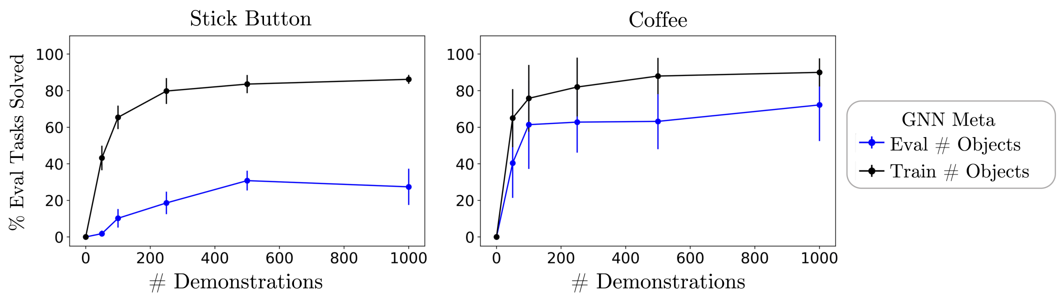


Figure 6: **GNN Meta additional results**. Task success rates for the GNN Meta baseline on tasks with more objects (Eval) or the same number of objects (Train) as seen during training. The gap suggests that the poor performance of GNN Meta in the main results (Figure 4) is largely attributable to a failure to generalize over object count. All results are over 10 random seeds. Lines are means and error bars are standard deviations. Note that in Cover and Doors, the distribution of object number is the same between train and evaluation.

pot to determine if it must be first rotated. To explain the poor performance of GNN Meta in these environments, we hypothesized that the model struggles to extrapolate to more objects than seen during training. We tested this hypothesis by evaluating the same GNN Meta models on new tasks from Stick Button and Coffee, but with object counts from the training distribution. Figure 4 shows that the performance of GNN Meta sharply improves, supporting the hypothesis.

C.3 Comparison to [12, 13]

Here we elaborate on the relationship between this work and our prior work [12, 13]. In brief, [13] extends [12] by removing the assumption that samplers are given. Our work here extends [13] by removing the assumptions that high-level controllers are given (e.g., pick, place, move), and that demonstration data is provided in terms of those high-level controllers.

Removing the assumption that high-level controllers are given requires several nontrivial steps. First, in [12, 13], the demonstration data is given in terms of high-level controllers: each transition corresponds to the execution of an entire controller, and the controller identity is known. This setting makes operator learning much easier because each transition corresponds to exactly one operator. The controller identity is also used to make operator learning easier. In contrast, we are given demonstration data where the actions are low-level (e.g., end effector positions of the robot), and we seek to learn operators that correspond to the execution of *many* low-level actions in sequence. This necessitates segmentation. Second, because the controllers are fully defined in the previous work, including their continuous parameterizations, it is straightforward to set up the sampler learning problems. In contrast, we have no such continuous controller parameterization given to us in this work. One of our main insights is that subgoal states can be used as the basis for continuous parameterization. This insight follows from KD1 and has the benefit that we can automatically derive targets for learning from the demonstration data. Finally, we must learn the controllers themselves. In the previous work, these controllers were hardcoded.

With these differences in mind, to motivate our work, we designed a version of our approach that ablates policy learning, and can be seen as an application of [13]. This ablation works as follows:

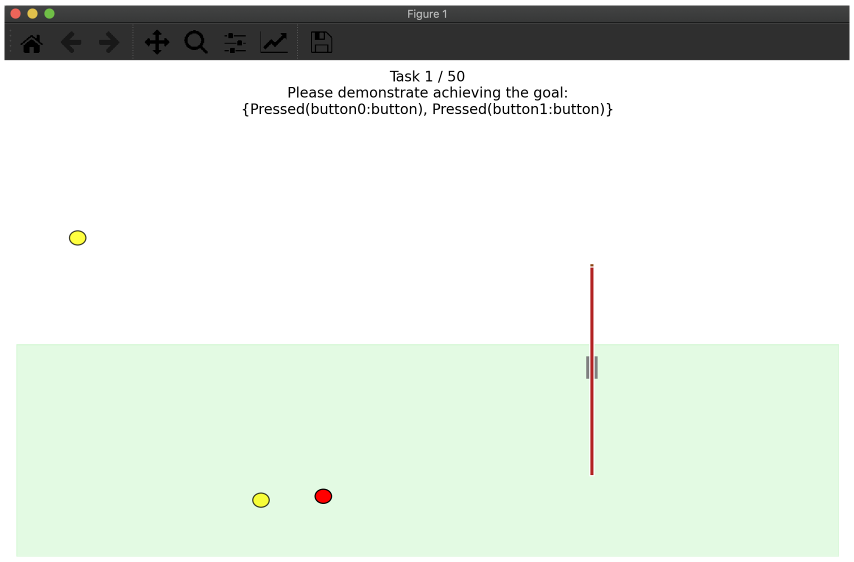
- For each transition in the demonstration data, we create one (single-step) segment.
- Partitioning and lifting are unmodified. Operator learning is also unmodified.
- Instead of policy learning, we create a “pass-through” policy architecture that consumes a continuous action (instead of a subgoal) and returns that action.
- Sampler learning is unmodified, except rather than sampling subgoals, we sample actions to give to the pass-through policy, consistent with the sampler learning of [13].

Another way to understand this ablation is that we are applying the method of [13] but supposing that the given action space consists of a single parameterized controller, with the parameter space equal to the low-level action space. We ran this ablation in the Cover environment with 1000 demonstrations and hyperparameters unchanged from our main experiments. Results are shown in the table on the right, with the numerical entries representing means (standard deviations) over 10 seeds.

Approach	Tasks Solved
Ours	99.40 (0.64)
Ablation [12]	2.80 (1.20)

Qualitatively, we see that the learned skills are the same between the two approaches, except that the ablation learns an additional skill with empty operator preconditions and effects. This additional skill is important and provides insight into the discrepancy between the two approaches. In the demonstration data, the majority of transitions do not correspond to any change in the abstract state. For example, as the robot moves in preparation for a pick or place, the abstract state is constant. These transitions lead to the empty operator effects. The preconditions are empty because there is nothing in common between the cases where no abstract effect occurs — sometimes the robot is holding an object, and sometimes it is not. This empty skill makes abstract planning very difficult because there is no signal for the planner to realize when using the skill would bring the robot closer to the goal. Note, though, that this empty skill is needed for planning, and simply removing it would make performance even worse; the remaining two skills can only handle the single step immediately preceding the respective effects (i.e., the step where an object is picked or placed).

C.4 Learning from Human Demonstrations



Task 1 / 50
Please demonstrate achieving the goal:
{Pressed(button0:button), Pressed(button1:button)}

Figure 7: GUI for collecting human demonstrations in Stick Button.

We ran an additional experiment with human demonstrations. To collect the demonstrations, we created a graphical user interface (GUI) and a simplified version of the Stick Button environments. The GUI is shown in Figure 7. Here the robot is a red circle, the buttons are yellow circles, the stick is a brown rectangle, the stick holder is the gray rectangle, and all buttons outside the green region are unreachable by the robot. Clicking a point on the screen initiates a translational robot movement, with the magnitude clipped so that the robot can only move so far in one action. Pressing any key initiates a grasp of the stick if the robot is close enough, or a press of the button if either the robot or stick head is close enough.

We used the GUI to collect 994 human demonstrations. We then ran BPNS with hyperparameters identical to the main results. Results are shown in Table 1. We find that the number of evaluation tasks solved by the approach trained on human demonstrations is slightly below that of automated demonstrations. This small gap can be attributed to noise, suboptimality, and inconsistencies in the human demonstrations. Overall, the strong performance of BPNS on human demonstrations suggests that the approach can be scaled.

C.5 Impact of Irrelevant Predicates

We conducted three additional experiments in the Cover environment to investigate the influence of irrelevant predicates. We used 1000 demonstrations and all hyperparameters are the same as in the main results.

Demos	% Tasks Solved	Test Time (s)	Nodes Created
Automated	83.60	51.80	421.51
Human	80.00	40.07	430.93

Table 1: Human demonstration results in Stick Button.

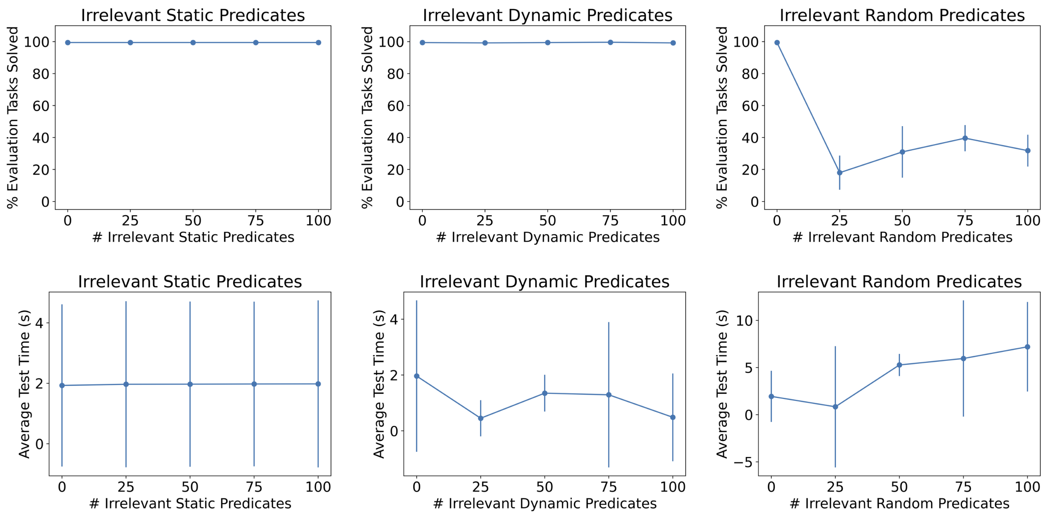


Figure 8: Impact of irrelevant predicates.

First, we added a variable number of *static* predicates, i.e., predicates whose evaluation is always True or False for an object regardless of the low-level state. Second, we added a variable number of *dynamic* (i.e., not static) predicates. Concretely, we created predicates that randomly threshold the y position of the robot. Third, in an attempt to create a setting that is adversarially bad for our framework, we added a variable number of *random* predicates, where the evaluation of each predicate is completely random on each input, with 50% probability of being True and without regard for the low-level state.

Results for each of the three experiments are shown in Figure 8. Static predicates have no apparent impact on evaluation performance. Qualitatively, we see that the preconditions of the learned operators have more preconditions, corresponding to the static predicates that are always True. The form of the operators, and the rest of the learned skills, are otherwise unchanged. The dynamic predicates also have little to no impact on evaluation performance. The learned operators again have additional preconditions, but also have additional dynamic predicates in the effects. However, these dynamic predicates are relatively "well-behaved", whereas in more complicated environments, predicate evaluations could be much less regular. This motivates the random predicates experiments, where indeed we see a substantial drop in evaluation performance. This drop is precipitated by a much larger and more complex set of learned operators, which makes abstract planning and learning more difficult. Altogether, these results confirm that the choice of predicates is important.

C.6 Impact of Irrelevant Objects

We conducted two additional experiments in the Cover environment to test the influence of irrelevant objects. We used 1000 demonstrations and all hyperparameters the same as in the main results.

In the first experiment, we added a variable number of irrelevant blocks during training, and in the second experiment, we added them instead during evaluation. The blocks are irrelevant because they are not involved in goals; they are also placed off the table so as to not cause collisions with the original blocks. The blocks *do* have the same type as the original blocks, so they will not be simply filtered out during type matching.

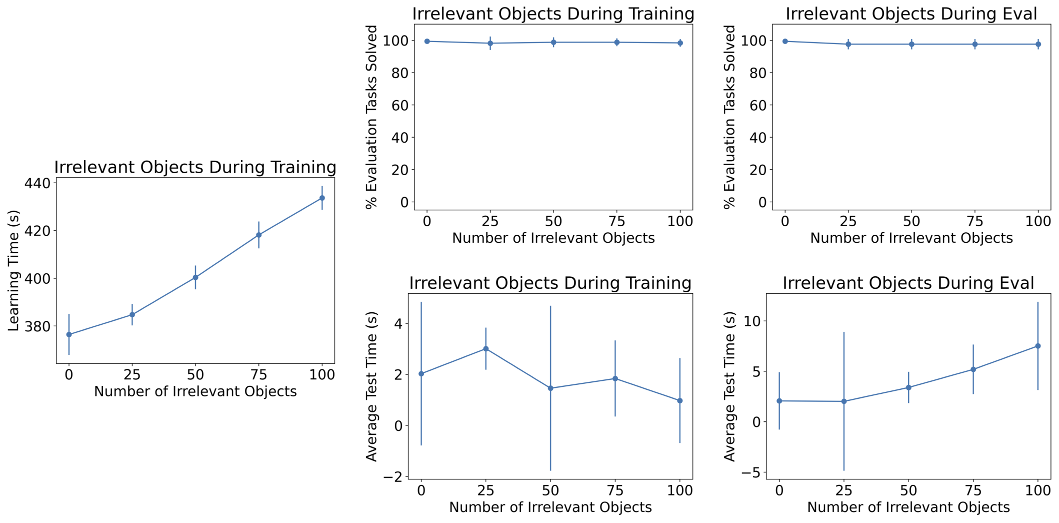


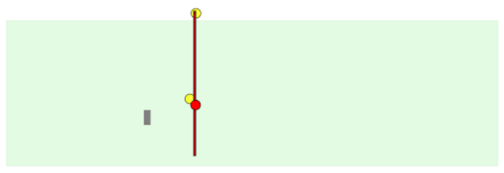
Figure 9: Impact of irrelevant objects.

Results for each of the experiments are shown in Figure 9. Adding the irrelevant objects during training has no impact on evaluation performance. This is expected, since our preprocessing pipeline naturally filters out the irrelevant objects. The irrelevant objects exhibit a small impact on learning time due to the cost of evaluating predicates. Adding the irrelevant objects during evaluation has a small impact on evaluation performance, although success rate is robust with up to 100 irrelevant objects. The increase in evaluation time is due to predicate evaluation and an increased branching factor during abstract planning.

C.7 Disabling Filtering of Low-Data Skills

Environment	Filter?	% Tasks Solved	Test Time (s)	Nodes Created
Cover	Yes	99.40 (0.64)	2.02 (2.84)	7.15 (0.42)
Cover	No	99.40 (0.64)	1.99 (2.79)	7.30 (0.59)
Doors	Yes	98.80 (0.80)	1.33 (0.59)	41.94 (4.56)
Doors	No	98.80 (0.80)	1.70 (1.05)	50.84 (24.27)
Stick Button	Yes	83.60 (1.78)	51.80 (11.47)	421.51 (81.28)
Stick Button	No	23.80 (9.68)	121.43 (57.14)	2725.17 (1898.47)
Coffee	Yes	98.00 (1.18)	49.39 (10.22)	53.68 (7.07)
Coffee	No	98.20 (1.04)	47.72 (9.52)	53.95 (7.01)

Table 2: Disabling filtering of low-data skills.



We ran an additional experiment where we disabled the filtering out of skills with low data. We used 1000 demonstrations and identical hyperparameters to our main experiments. Results are shown in Table 2, with the numerical entries representing means (standard deviations) over 10 seeds. The results show that evaluation performance in Cover, Doors, and Coffee is largely unaffected by filtering, while the performance in Stick Button is substantially affected. The performance in Stick Button can be traced back to the rare situation illustrated in the

Env	Approach	% Tasks Solved	Test Time (s)	Nodes Created
Cover	Ours	99.40 (0.64)	2.02 (2.84)	7.15 (0.42)
Cover	Oracle	98.80 (0.30)	0.03 (0.01)	7.01 (0.17)
Doors	Ours	98.80 (0.80)	1.33 (0.59)	41.94 (4.56)
Doors	Oracle	100.00 (0.00)	1.04 (0.09)	84.12 (20.17)
Stick Button	Ours	83.60 (1.78)	51.80 (11.47)	421.51 (81.28)
Stick Button	Oracle	90.40 (1.99)	0.18 (0.03)	320.32 (46.92)
Coffee	Ours	98.00 (1.18)	49.39 (10.22)	53.68 (7.07)
Coffee	Oracle	100.00 (0.00)	0.07 (0.01)	67.25 (8.12)

Table 3: Comparison to oracle skills.

image on the right. Typically, when the robot (red) presses a button (yellow) with the stick (brown), the robot is not above any other button. However, in this case, the robot is coincidentally above a second button while executing the stick press. This leads to an operator with an effect set that includes both the button being pressed and the robot being above a second button. That operator is ultimately detrimental to planning because in the vast majority of cases, it is not possible for the robot to press the button while being above a second button, so refining this operator usually fails. This operator also has a very small amount of training data, which makes the associated policy and sampler unreliable. For similar reasons, we prefer to filter out skills with too little training data by default.

C.8 Comparison to Oracle

We collected statistics for an oracle approach that uses manually-designed skills. We use identical hyperparameters to the main results. The statistics are reported in Table 3, with the numerical entries representing means (standard deviations) over 10 seeds, and where "Ours" is the main approach, BPNS, trained with 1000 demonstrations. In Doors and Coffee, the learned skills require fewer node creations during abstract search to find a plan. This difference can be attributed to (1) overly general preconditions in the operators of our manually designed skills; and (2) more targeted sampling when using the learned samplers versus manually designed samplers. In all environments, the wall-clock time taken to plan with our learned skills is far greater than that of the oracle. From profiling, we can see that this difference is largely due to neural network inference time in both the learned samplers and learned skill policies. In contrast, the manually designed skills are written in pure Python, and can therefore be evaluated very efficiently.

C.9 Learned Operator Examples

See Figure 10 for examples of learned operators in each environment. Below, we describe the operators that are typically learned at convergence for each of the environments. These descriptions are based on inspection of the operator syntax, speaking to their interpretability.

- **Cover:**
  1. Pick up a block.
  2. Place a block on a target.
- **Doors:**
  1. Move to a door from the main part of a room (not in a doorway).
  2. Move to a door from another doorway.
  3. Move through an open door.
  4. Open a door.
- **Stick Button:**
  1. Move from free space to press a button with the gripper.

2. Move from above another button to press a button with the gripper.
3. Move from free space to pick up a stick.
4. Move from above a button to pick up a stick.
5. Move from free space to press a button with the stick.
6. Move from above another button to press a button with the stick.

- **Coffee:**
  1. Pick up the coffee pot.
  2. Put the coffee pot on the hot plate.
  3. Turn on the hot plate.
  4. Move from above no cup to pour into a cup.
  5. Move from above another cup to pour into a cup.
  6. Twist the coffee pot.
  7. Pick up the coffee pot after twisting.

C.10 Predicate Invention Preliminary Results

We ultimately envision a continually learning robot that uses symbols to learn skills and skills to learn symbols in a virtuous cycle of self-improvement. One plausible path toward realizing this vision would start with a set of manually designed symbols, as we did in this work, or skills, as done by Konidaris et al. [4]. Alternatively, we could start with demonstrations alone. In this case, we need to answer the chicken-or-the-egg question: which should be learned first, skills or symbols? Here we present very preliminary results suggesting the viability of learning symbols (predicates) first, and then skills from those learned symbols.

Metric	**Manual**	**Learned**
% Eval Tasks Solved	99.40 (0.64)	99.20 (0.92)
# Predicates	5.00 (0.00)	5.40 (0.80)
Eval Time (s)	2.00 (2.78)	2.82 (3.10)
Learning Time (s)	372.01 (4.55)	4898.16 (692.09)

We follow the approach of Silver et al. [21], which starts with a minimal set of *goal predicates* that are sufficient for describing the task goals, and then uses demonstrations to invent new predicates. Specifically, we focus on the Cover environment, where there is only one goal predicate: `Covers`. Given 1000 demonstrations, after learning predicates (see [21] for a description of the approach), we run BPNS skill learning and planning, with the configuration identical to the main experiments. Results are shown in the table on the right. Each entry is a mean (standard deviation) over 10 random seeds. The Manual column uses the manually designed predicates from our main experiments and the Learned column uses learned predicates. Further investigation is needed, but the results do suggest that learning skills on top of learned predicates is a viable direction. We also report the number of predicates learned, evaluation time, and learning time. Consistent with the prior work, we see that additional predicates can be learned that sometimes lead to faster planning during evaluation. We also see that the time bottleneck for the overall system is predicate invention, not skill learning.

```
Learned-Op0:
  Parameters: [?x0:block, ?x1:gripper]
  Preconditions: [HandEmpty(), IsBlock(?x0:block)]
  Add Effects: [Holding(?x0:block, ?x1:gripper)]
  Delete Effects: [HandEmpty()]
```

Example learned operator for picking a block in *Cover*.

```
Learned-Op0:
  Parameters: [?x0:door, ?x1:robot]
  Preconditions: [InDoorway(?x1:robot, ?x0:door),
                  TouchingDoor(?x1:robot, ?x0:door)]
  Add Effects: [DoorIsOpen(?x0:door)]
  Delete Effects: [TouchingDoor(?x1:robot, ?x0:door)]
```

Example learned operator for opening a door in *Doors*.

```
Learned-Op0:
  Parameters: [?x0:gripper, ?x1:stick]
  Preconditions: [AboveNoButton(),
                 HandEmpty(?x0:gripper)]
  Add Effects: [Grasped(?x0:gripper, ?x1:stick)]
  Delete Effects: [HandEmpty(?x0:gripper)]
```

Example learned operator for grasping the stick in *Stick Button*.

```
Learned-Op0:
  Parameters: [?x0:cup, ?x1:pot, ?x2:gripper]
  Preconditions: [Holding(?x2:gripper, ?x1:pot),
                  PotHot(?x1:pot),
                  NotAboveCup(?x2:gripper, ?x1:pot)]
  Add Effects: [CupFilled(?x0:cup),
                PotAboveCup(?x1:pot, ?x0:cup),
                RobotAboveCup(?x2:gripper, ?x0:cup)]
  Delete Effects: [NotAboveCup(?x2:gripper, ?x1:pot)]
```

Example learned operator for pouring in *Coffee*.

Figure 10: **Learned operator examples**.