Appendix

A. Novelty descriptions

In this section we describe the details of each novelty.

1) Rubber tree: In this novelty, as described in Table 2 the agent cannot extract *rubber* from the regular trees anymore, but a new entity *rubber-tree* appears in the environment and the agent needs to use the *rubber-tree* for *rubber* extraction. We have two versions of this novelty, in the *easy* version, the agent needs to select the *tree-tap* and standing in front of a *rubber-tree* it needs to use the action *extract-rubber* to extract rubber. In the hard version, the agent's action space is augmented with a new action called *place-tree-tap*, and in order to get rubber, the agent now needs to place the *tree-tap* in front of the *rubber-tree* and *rubber* can only be extracted when the agent is in front of the placed *tree-tap*. The agent's knowledge base does not have any *in front of* predicate. The agent's executor learns the policy to place the new entity of *tree-tap*, locate itself in front of it, and then extract *rubber*. We observe that the agent is successful in synthesizing executors with predicates missing from the knowledge base.

2) Axe to break: In this novelty, a novel entity *axe* appears in the environment and the agent cannot break *trees* without holding the *axe*. We implement two versions of the novelty, easy and hard. In the easy version, the *axe* is present in the agent's inventory, and the agent needs to learn to select the *axe* and use it in front of the *tree* to get *tree-log* into its inventory. In the hard version, the *axe* is present in the environment, and the agent needs to find it to get into its inventory and then use it to break *trees*. We observe that the agent is able to generate executors that can reason about the affordances of new entities in the environment, i.e. learn to pick up the *axe*, select it, and then *break* the *tree* by locating itself next to the *tree*.

3) Fire crafting table: In this novelty, the agent cannot access the *crafting-table* since its set on *fire*. Two novel entities *water* and *fire* appears in the environment, and novel actions *spray* and *select-water* are added to the list of available actions. In order to access the *crafting-table* the agent needs to select *water* and use the action *spray* in front of the *crafting-table* in order to remove it from *fire*, and hence successfully access it. In the easy version, the *water* is already present in the agent's inventory and in the hard version *water* is present in the environment and the agent needs to collect it to use it. The agent's sub-symbolic representation has information about the status of the *fire* (on/off). The agent's learned executor in successfully dealing with more than one novel entities, and also novel predicates that are absent in the agent's knowledge base.

4) Scrape plank: In this novelty, the *break* action has no effect, and a novel action *scrape-plank* is augmented in the agent's action space. The dynamics of the world changes in a way that the agent can no longer break *trees* to get *tree-logs*, but it can scrape planks from trees and using the action *scrape-plank* in front of a *tree* it gets 4 *planks* into its inventory. This novelty emphasizes on failure of effects of the operator rather than failure of the action itself. The executor learns a policy to generate *planks*, by utilizing the *scrape-plank* action to move ahead in the plan. Thus, the agent learns to tackle a novel scenario even when there are no new entities in the environment.

B. Additional Algorithms

We describe the algorithm which generates the set of plannable states.

The agent accumulates the set of preconditions Ψ and effects Ω of all known operators, using the pre-novelty domain knowledge. Using that, it generates a set of plannable states $\mathcal{S}_r$, which contains: 1) the states that satisfy the effects ω_{o_i} of the failed operator o_i and 2) the states that satisfy the effects of all subsequent operators $\hat{o} \in \hat{\mathcal{O}}$ in the plan $\mathcal{P}$ $((d(\tilde{s}) \supseteq \omega_{\hat{o}}) \forall \hat{o} \in \hat{\mathcal{O}})$, where the preconditions of $\hat{o}$ contain the effects of o $(\psi_{\hat{o}} \supseteq \omega_o)$(as shown in Algorithm 1). This way we generate our algorithm generates a list of all the states from where the agent can successfully plan to reach the goal state. We provide this as a disjunction of condition 1 and 2 for the agent to satisfy at each time step. In other words, if the agent satisfies either 1 or 2, it gets a positive reward ϕ_1 and the episode terminates.

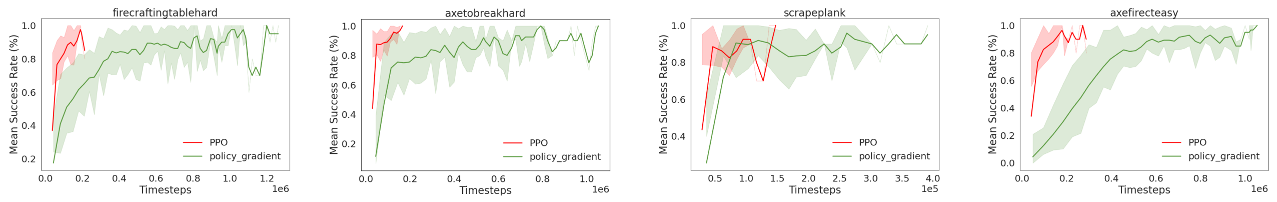
C. Baseline Approaches

In this section we describe the details of the baselines, specifically, how they were trained for pre-novelty scenarios in Section C.1, how they preformed in the pre-novelty scenario in Section C.2, followed by how transfer was performed upon novelty injections in Section D. Finally, Figure 1 compares the learning curves of these baselines on some novelties.

1) Pre-novelty reward shaping: In early experiments, the pre-novelty task with sparse rewards proved intractable for pure RL approaches. Moreover, the RAPid-Learn agent has access to the full PDDL description of the environment, which heavily biases any comparison in its favor. We therefore employ strong reward shaping to allow the RL baseline

Algorithm 1 ***PlannableStateGenerator*** $(\Sigma, \mathcal{P}, o_i) \rightarrow \mathcal{S}_r$

```
1: $\mathcal{P} = [o_1, \dots o_{|\mathcal{P}|}]$: Plan as an ordered list of operators
2: $\Psi = \{\psi_{o_1}, \psi_{o_2}, \dots, \psi_{o_{|\mathcal{P}|}}\}$: Set of preconditions of all the known operators
3: $\Omega = \{\omega_{o_1}, \omega_{o_2}, \dots, \omega_{o_{|\mathcal{P}|}}\}$: Set of effects of all the known operators
4: $\mathcal{S}_r \rightarrow \emptyset$
5: for $o_j$ in reversed$(\mathcal{P})$ do
6:     if $o_j \neq o_i$ & $\psi_{o_j} \nsupseteq \omega_{o_i}$ then
7:         $\mathcal{S}_r \cup \psi_{o_j}$
8:         for $\omega \in \omega_{o_j}$ do
9:             if $\omega \subseteq \mathcal{S}_r$ then
10:                $\mathcal{S}_r \setminus \{\omega\}$
11:            end if
12:        end for
13:    end if
14:    $\mathcal{S}_r \cup \omega_{o_i}$
15: end for
16: return $\mathcal{S}_r$
```



(a) Fire-crafting-table hard (b) Axe-to-break hard (c) Scrape plank (d) Axetobreak+FCT(easy)

Fig. 1: Baseline learning curves on the four novelties. Green curve shows the actor-critic transfer, and the red curve shows policy-reuse. Each plot demonstrates the performance during the novelty adaptation phase.

to learn the prenovelty task. A small reward (+50) is given for completing intermediate steps (e.g. crafting sticks or planks) when they are needed for the next major step in the crafting process. Larger rewards are given for the subgoals of crafting the treetap (+200), extracting rubber (+300) and finally crafting the pogostick (+1000). Note that the rewards are only given if the intermediate step is the appropriate thing to do next. I.e. no reward is given for breaking a tree, if the agent already has a treetap and enough sticks and planks in its inventory to craft the pogostick. In this case, a reward would only be given for obtaining the missing ingredient (rubber). Making the reward conditional on the stage within the crafting process was essential in getting the agent to learn the pre-novelty task. In early versions of the reward shaping, we attempted to give reward for completing intermediate steps (approaching a tree, crafting sticks, etc.). This was exploited by the agent, which learned that the reward could be optimized by periodically turning away and re-approaching a tree until the end of its life.

2) *Prenovelty performance:* The baselines as described C.1 were trained using a shaped reward. We evaluate their performance to make sure that they are at par with the RAPid-Learn approach (RAPid-Learn, being a planning agent, shows a 100% performance in the pre-novelty scenario). Table I shows how many timesteps the pre-novelty models have experienced until reaching convergence and their performance on 100 evaluation instances of the environment on 10 random seeds. Results in Table I show that both the baseline agents converge to a policy which performs with a success of the task being achieved 95% of the time.

Pre-novelty		
Agent	**Time to learn (timesteps)**	**Pre-novelty performance (success rate %)**
		Mean ± SD
Actor-Critic Transfer	2.47×10^5	0.95 ± 0.22
Policy Reuse	1.48×10^6	0.95 ± 0.22

TABLE I: Performance and timesteps trained for the best pre-novelty model that was later used in transfer learning the novelty environments

D. Baselines transfer mechanism

The injected novelties alter the shape of the observation and action spaces of the environment. E.g. the Fire crafting table novelty adds new observations for the *water* item in the inventory and the LiDAR-sensor, as well as the *select water* and *spray* actions. In order to allow transferring the model from pre-novelty to post-novelty environments, we pad the observation and action spaces of all environments (pre- and post-novelty) with placeholder elements such that all of them have the same shape. Observation placeholders simply always return zero. When the model chooses to perform a placeholder action, the action actually passed on to the environment is always the *approach treelog*.

E. Ablation study

To examine the impact of hierarchical actions combined with the knowledge-guided-exploration function, we perform experimental evaluation in absence of hierarchical actions. For our experiments, hierarchical actions are defined by the (`approach`) operator, followed by the entity to approach. This hierarchical action is composed of the primitive actions (`turn left, turn right` and `move forward`). An example of an hierarchical action is `approach tree_log`, where the agent determines the location of one of the `tree_log` in the environment and formulates a plan to reach the entity. This plan is computed using the A^* algorithm [**?**].

Incorporation of hierarchical actions simplifies novelty adaptation, as the agent has the opportunity to accommodate the novelty in fewer interactions with the environment. Fig 2. We can observe that for the complex *fire-crafting-table hard* novelty, the *RAPid-EG* approach fails to converge to a successful policy. The difficulty for the agent to successfully complete the task can be attributed to two reasons:

- The agent has to follow a series of movement actions to first collect the *water* in its inventory, and then to reach the *crafting table* where it needs to spray the *water* by holding it, in order to diffuse the fire.
- While it performs the above mentioned steps, it should avoid reaching the state from which it cannot plan again. This will happen if the agent crafts crafts extra sticks in its inventory, making it impossible to craft a pogo stick with the resources available in the environment and in its inventory.

In absence of any informed exploration, the baseline approach 'RAPid-EG' takes longer to converge to a successful

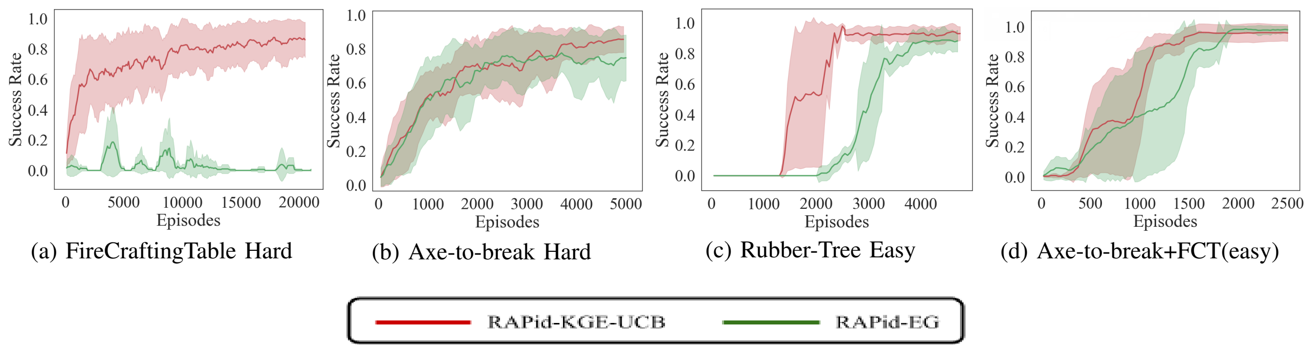


Fig. 2: The plots show the learning curves for four novelties on ablation study (no hierarchical-action operators were used). Red curve shows RAPid-Learn with knowledge-guided-exploration function, and green curve shows without the knowledge-guided-exploration but a vanilla epsilon-greedy exploration.

policy in absence of hierarchical actions. Thus, we can see that hierarchical actions play a crucial role in increasing the sample efficiency of the outlined approach.

F. Training details

1) Convergence criteria: To ensure that the learner has learned to reach the goal, we verify if the agent achieved the goal $\delta_g \geq \delta_G$ in the last η episodes with an average reward $\delta_r \geq \delta_R$. Furthermore, to ensure the agent has converged, and is not learning any further, we verify if the success rate for the agent in the past $\eta + \upsilon$ episodes is equal to the success rate for the agent in the past η episodes.

Parameter	**Value**
learning_rate	$3e-4$
batch_size	64
n_epochs	10
n_steps	2048
gamma	0.98
gae_lambda	0.95
clip_range	0.2
max_grad_norm	0.5
ent_coef	0.0
vf_coef	0.5
optimiser	Adam

TABLE II: Hyperparameters used for the PPO models (both pre- and post-novelty).

2) RAPid-Learn hyperparameters: See Table III for the hyperparameters used in the RAPid-Learn experiments.

3) Baseline Hyperparameters: See Table II for an overview of the hyperparameters used to train the actor-critic transfer baselines. Note that with the exception of gamma, these are the default parameters used in [**?**]. The Policy Reuse baseline models utilize the exact same implementation

Parameter	**Value**
max-epsilon (ϵ_{max})	0.3
min-epsilon (ϵ_{min})	0.05
guided curriculum parameter ($\rho_{\max}$)	0.3
guided curriculum parameter ($\rho_{\min}$)	0.05
parameter decay speed	$\ln(0.01)/2000$
UCB- parameter (c)	0.0005
UAB-parameter(μ)	2
update_rate	10
max episodes (e_{max})	100000
max timesteps (T)	300
Hidden Layers	24 (single layer network)
Discount factor (γ)	0.98
learning rate (α)	$1e-3$
reward threshold (δ_R)	900
episodes threshold (δ_G)	100
no of success (η)	96
positive reinforcement (ϕ_1)	1000
negative reinforcement (ϕ_2)	-350

TABLE III: Hyperparameters for RAPid-Learn

that underlies the Learner in RAPid-Learn, with identical hyperparameter settings, as shown in Table III

G. Domain PDDL

Below is the original PDDL domain given to RAPid-Learn for the pogostick-crafting task inspired by Minecraft in novelgridworlds.

```
(:requirements :typing :strips :fluents)
(:types
    wall - physobj
    entity - physobj
    plank - physobj
    crafting_table - physobj
    tree_tap - physobj
    var - object
    rubber - physobj
    physobj - physical
    actor - physical
    pogo_stick - physobj
    stick - physobj
    tree_log - physobj
    air - physobj
)

(:predicates
    (holding ?v0 - physobj)
    (floating ?v0 - physobj)
    (facing ?v0 - physobj)
)

(:functions
    (world ?v0 - object)
    (inventory ?v0 - object)
)

(:action approach
    :parameters    (?physobj01 - physobj ?physobj02 - physobj )
    :precondition  (and
        (>= ( world ?physobj02) 1)
        (facing ?physobj01)
    )
    :effect  (and
        (facing ?physobj02)
        (not (facing ?physobj01))
    )
)

(:action crafttree_tap ;
    :parameters    ()
    :precondition  (and
        (>= ( inventory plank) 4)
        (>= ( inventory stick) 1)
        (facing crafting_table)
    )
    :effect  (and
        (increase ( inventory tree_tap) 1)
        (decrease ( inventory plank) 5)
        (decrease ( inventory stick) 1)
    )
)

(:action craftplank
    :parameters    ()
    :precondition  (>= ( inventory tree_log)) 1)
    :effect  (and
        (increase ( inventory plank) 4)
        (decrease ( inventory tree_log) 1)
    )
)

(:action break
    :parameters    ()
    :precondition  (and
        (facing tree_log)
        (not (floating tree_log))
    )
    :effect  (and
        (facing air)
        (not (facing tree_log))
        (increase ( inventory tree_log) 1)
        (increase ( world air) 1)
        (decrease ( world tree_log) 1)
    )
)

(:action craftstick
    :parameters    ()
    :precondition  (>= ( inventory plank) 2)
    :effect  (and
        (increase ( inventory stick) 4)
        (decrease ( inventory plank) 2)
    )
)

(:action extractrubber
    :parameters    ()
    :precondition  (and
        (>= ( inventory tree_tap) 1)
        (facing tree_log)
        (holding tree_tap)
    )
    :effect  (and
        (increase ( inventory rubber) 1)
    )
)

(:action craftpogo_stick
    :parameters    ()
    :precondition  (and
        (>= ( inventory plank) 2)
        (>= ( inventory stick) 4)
        (>= ( inventory rubber) 1)
        (facing crafting_table)
    )
    :effect  (and
        (increase ( inventory pogo_stick) 1)
        (decrease ( inventory plank) 2)
        (decrease ( inventory stick) 4)
        (decrease ( inventory rubber) 1)
    )
)

(:action select
    :parameters    (?physobj01 - physobj )
    :precondition  (and
        (>= ( inventory ?physobj01) 1)
        ; (holding air)
    )
    :effect  (and
        (holding ?physobj01)
        (not (holding air))
    )
)

)
```