A LGB PSEUDO-CODE

Algorithm 1 and 2 present the high-level pseudo-code of any algorithm following the LGB architecture for each of the three phases.

Algorithm 1 LGB architecture
G→B phase

▷ Goal → Behavior phase
Require Env E
Initialize policy Π, goal sampler G_s, buffer B
loop
  $g \leftarrow G_s$.sample()
  $(s, a, s', g, c_p, c'_p)_{\text{traj}} \leftarrow E$.rollout($g$)
  G_s.update(c_p^T)
  B.update($(s, a, s', g, c_p, c'_p)_{\text{traj}}$)
  Π.update(B)
return Π, G_s

Algorithm 2 LGB architecture
L→G and L→G→B phases

▷ Language → Goal phase
Require Π, E, G_s, social partner SP
Initialize language goal generator LGG
dataset $\leftarrow SP$.interact(E, Π, G_s)
LGG.update(dataset)
return LGG
▷ Language → Behavior phase
Require E, Π, LGG, SP
loop
  instr. $\leftarrow SP$.listen()
  loop ▷ Strategy switching loop
    $g \leftarrow LGG$.sample(instr., c^0)
    $c_p^T \leftarrow E$.rollout(g)
    if $g == c_p^T$ **then break**

B SEMANTIC PREDICATES AND APPLICATION TO FETCH MANIPULATE

In this paper, we restrict the semantic representations to the use of the *close* and *above* binary predicates applied to $M = 3$ objects. The resulting semantic configurations are formed by:

$$c_p = [c(o_1, o_2),\ c(o_1, o_3),\ c(o_2, o_3),\ a(o_1, o_2),\ a(o_2, o_1),\ a(o_1, o_3),\ a(o_3, o_1),\ a(o_2, o_3),\ a(o_3, o_2)],$$

where *c()* and *a()* refer to the *close* and *above* predicates respectively and $(o_1,\ o_2,\ o_3)$ are the red, green and blue blocks respectively.

Symmetry and asymmetry of *close* and *above* predicates. We consider objects o_1 and o_2.

- *close* is symmetric: "o_1 is **close** to o_2" ⇔ "o_2 is **close** to o_1". The corresponding semantic mapping function is based on the Euclidean distance, which is symmetric.
- *above* is asymmetric: "o_1 is **above** o_2" ⇒ **not** "o_2 is **above** o_1". The corresponding semantic mapping function evaluates the sign of the difference of the object Z-axis coordinates.

C THE DECSTR ALGORITHM

C.1 INTRINSICALLY MOTIVATED GOAL-CONDITIONED RL

Overview. Algorithm 3 presents the pseudo-code of the sensorimotor learning phase (G→B) of DECSTR. It alternates between two steps:

- **Data acquisition.** A DECSTR agent has no prior on the set of reachable semantic configurations. Its first goal is sampled uniformly from the semantic configuration space. Using this goal, it starts interacting with its environment, generating trajectories of sensory states s, actions a and configurations c_p. The last configuration c_p^T achieved in the episode after T time steps is considered stable and is added to the set of reachable configurations. As it interacts with the environment, the agent explores the configuration space, discovers reachable configurations and selects new targets.
- **Internal models updates.** A DECSTR agent updates two models: its curriculum strategy and its policy. The curriculum strategy can be seen as an active goal sampler. It biases the selection of goals to target and goals to learn about. The policy is the module controlling the agent's behavior and is updated via RL.

Algorithm 3 DECSTR: sensorimotor phase G→B.

Require: env E, # buckets N_b, # episodes before biased init. n_{unb}, self-evaluation probability $p_{\text{self\_eval}}$, noise function $\sigma()$
Initialize: policy Π, buffer B, goal sampler G_s, bucket sampling probabilities p_b, language module LGG.
loop
    self_eval ← random() $< p_{\text{self\_eval}}$ ▷ If $True$ then evaluate competence
    $g \leftarrow G_s$.sample(self_eval, p_b)
    biased_init ← $epoch < n_{\text{unb}}$ ▷ Bias initialization only after n_{unb} epochs
    $s^0, c_p^0 \leftarrow E$.reset($biased\_init$) ▷ c_0: Initial semantic configuration
    for $t = 1 : T$ **do**
        $a^t \leftarrow policy(s^t, c^t, g)$
        if not self_eval **then**
            $a^t \leftarrow a^t + \sigma()$
        $s^{t+1}, c_p^{t+1} \leftarrow E$.step($a^t$)
    episode ← (s, c, a, s', c')
    G_s.update(c^T)
    B.update(episode)
    $g \leftarrow G_s$.sample(p_b)
    batch ← B.sample(g)
    Π.update(batch)
    if self_eval **then**
        $p_b \leftarrow G_s$.update_LP()

Policy updates with a goal-conditioned Soft Actor-Critic. Readers familiar with Markov Decision Process and the use of SAC and HER algorithms can skip this paragraph.

We want the DECSTR agent to explore a semantic configuration space and master reachable configurations in it. We frame this problem as a goal-conditioned MDP (Schaul et al., 2015): $\mathcal{M} = (\mathcal{S}, \mathcal{G}_p, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma)$, where the state space $\mathcal{S}$ is the usual sensory space augmented with the configuration space $\mathcal{C}_p$, the goal space $\mathcal{G}_p$ is equal to the configuration space $\mathcal{G}_p = \mathcal{C}_p$, $\mathcal{A}$ is the action space, $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ is the unknown transition probability, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \{0, 1\}$ is a sparse reward function and $\gamma \in [0, 1]$ is the discount factor.

Policy updates are performed with Soft Actor-Critic (SAC) (Haarnoja et al., 2018), a state-of-the-art off-policy actor-critic algorithm. We also use Hindsight Experience Replay (HER) (Andrychowicz et al., 2017). This mechanism enables agents to learn from failures by reinterpreting past trajectories in the light of goals different from the ones originally targeted. HER was designed for continuous goal spaces, but can be directly transposed to discrete goals (Colas et al., 2019). In our setting, we simply replace the originally targeted goal configuration by the currently achieved configuration in the transitions fed to SAC. We also use our automatic curriculum strategy: the LP-C-based probabilities are used to sample goals to learn about. When a goal g is sampled, we search the experience buffer for the collection of episodes that ended in the configuration $c_p = g$. From these episodes, we sample a transition uniformly. The HER mechanism substitutes the original goal with one of the configurations achieved later in the trajectory. This substitute g has high chances of being the sampled one. At least, it is a configuration on the path towards this goal, as it is sampled from a trajectory leading to it. The HER mechanism is thus biased towards goals sampled by the agent.

Object-Centered Inductive Biases. In the proposed *Fetch Manipulate* environment, the three blocks share the same set of attributes (position, velocity, color identifier). Thus, it is natural to encode a *relational inductive bias* in our architecture. The behavior with respect to a pair of objects should be independent from the position of the objects in the inputs. The architecture used for the policy is depicted in Figure 1.

A shared network (NN_{shared}) encodes the concatenation of: 1) agent's body features; 2) object pair features; 3) current configuration (c_p) and 4) current goal g. This is done independently for all object pairs. No matter the location of the features of the object pair in the initial observations, this shared network ensures that the same behavior will be performed, thus skills are transferred between object

pairs. A sum is then used to aggregate these outputs, before a final network (NN_{policy}) maps the aggregation to actions a. The critic follows the same architecture, where a final network NN_{critic} maps the aggregation to an action-value Q. Parallel encoding of each pair-specific inputs can be seen as different modules trying to reach the goal by only seeing these pair-specific inputs. The intuition is that modules dealing with the pair that should be acted upon to reach the goal will supersede others in the sum aggregation.

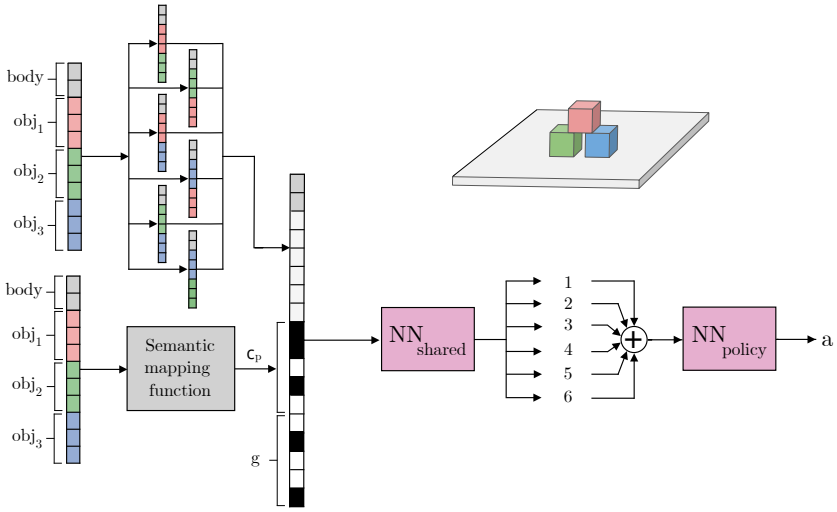

Figure 1: Object-centered modular architecture for the policy.

Although in principle our architecture could work with combinations of objects (3 modules), we found permutations to work better in practice (6 modules). With combinations, the shared network would need to learn to put block A on block B to achieve a predicate *above*$(o_i,\ o_j)$, and would need to learn the reverse behavior (put B on A) to achieve the symmetric predicate *above*$(o_j,\ o_i)$. With permutations, the shared network can simply learn one of these behaviors (e.g. A on B). Considering the predicate *above*$(o_A,\ o_B)$, at least one of the modules has objects organized so that this behavior is the good one: if the permutation $(o_B,\ o_A)$ is not the right one, permutation $(o_A,\ o_B)$ is. The symmetry bias is explained in Section **??**. It leverages the symmetry of the behaviors required to achieve the predicates *above*$(o_i,\ o_j)$ and *above*$(o_j,\ o_i)$. As a result, the two goal configurations are:

$$g_1 = [c(o_1, o_2),\ c(o_1, o_3),\ c(o_2, o_3),\ a(o_1, o_2),\ a(o_1, o_3),\ a(o_2, o_3)],$$
$$g_2 = [c(o_1, o_2),\ c(o_1, o_3),\ c(o_2, o_3),\ a(o_2, o_1),\ a(o_3, o_1),\ a(o_3, o_2)],$$

where g_1 is used in association with object permutations $(o_i,\ o_j)$ with $i < j$ and g_2 is used in association with object permutations $(o_j,\ o_i)$ with $i < j$. As a result, the shared network automatically ensures transfer between predicates based on symmetric behaviors.

Implementation Details. This part includes details necessary to reproduce results. The code is available at https://sites.google.com/view/decstr/.

Parallel implementation of SAC-HER. We use a parallel implementation of SAC (Haarnoja et al., 2018). Each of the 24 parallel worker maintains its own replay buffer of size 10^6 and performs its own updates. Updates are summed over the 24 actors and the updated network are broadcast to all workers. Each worker alternates between 2 episodes of data collection and 30 updates with batch size 256. To form an epoch, this cycle is repeated 50 times and followed by the offline evaluation of the agent on each reachable goal. An epoch is thus made of $50 \times 2 \times 24 = 2400$ episodes.

Goal sampler updates. The agent performs self-evaluations with probability $self\_eval = 0.1$. During these runs, the agent targets uniformly sampled discovered configurations without exploration noise. This enables the agent to self-evaluate on each goal. Goals are organized into buckets. Main Section **??** presents our automatic bucket generation mechanism. Once buckets are formed, we compute C, LP and P, based on windows of the past $W = 1800$ self-evaluation interactions for each bucket.

Modular architecture. The shared network of our modular architecture NN_{shared} is a 1-hidden layer network of hidden size 256. After all pair-specific inputs have been encoded through this module, their output (of size 84) are summed. The sum is then passed through a final network with a hidden layer of size 256 to compute the final actions (policy) or action-values (critic). All networks use $ReLU$ activations and the Xavier initialization. We use Adam optimizers, with learning rates 10^{-3}. The list of hyperparameters is provided in Table 1.

Table 1: Sensorimotor learning hyperparameters used in DECSTR.

Hyperparam.	Description	Values.
nb_mpis	Number of workers	24
nb_cycles	Number of repeated cycles per epoch	50
$nb\_rollouts\_per\_mpi$	Number of rollouts per worker	2
$nb\_updates$	Number of updates per cycle	30
$start\_bias\_init$	Epoch from which initializations are biased	100
W	Curriculum window size	1800
$self\_eval$	Self evaluation probability	0.1
N_b	Number of buckets	5
$replay\_strategy$	HER replay strategy	$future$
k_replay	Ratio of HER data to data from normal experience	4
$batch\_size$	Size of the batch during updates	256
γ	Discount factor to model uncertainty about future decisions	0.98
τ	Polyak coefficient for target critics smoothing	0.95
lr_actor	Actor learning rate	10^{-3}
lr_critic	Critic learning rate	10^{-3}
α	Entropy coefficient used in SAC	0.2
$automatic\_entropy$	Automatically tune the entropy coefficient	$False$

Computing resources. The sensorimotor learning experiments contain 8 conditions: 2 of 10 seeds and 6 of 5 seeds. Each run leverages 24 cpus (24 actors) for about 72h for a total of 9.8 cpu years. Experiments presented in this paper requires machines with at least 24 cpu cores. The language grounding phase runs on a single cpu and trains in a few minutes.

C.2 LANGUAGE-CONDITIONED GOAL GENERATOR

Language-Conditioned Goal Generator Training. We use a conditional Variational Auto-Encoder (C-VAE) (Sohn et al., 2015). Conditioned on the initial configuration and a sentence describing the expected transformation of one object relation, it generates compatible goal configurations. After the first phase of goal-directed sensorimotor training, the agent interacts with a hard-coded social partner as described in Main Section **??**. From these interactions, we obtain a dataset of 5000 triplets: initial configuration, final configuration and sentence describing one change of predicate from the initial to the final configuration. The list of sentences used by the synthetic social partner is provided in Table 2. Note that *red*, *green* and *blue* refer to objects $o_1,\ o_2,\ o_3$ respectively.

Content of test sets. We describe the 5 test sets:

1. Test set 1 is made of input pairs $(c_i,\ s)$ from the training set, but tests the coverage of all compatible final configurations $\mathcal{C}_f$, 80% of which are not found in the training set. In that sense, it is partly a test set.
2. Test set 2 contains two input pairs: $\{[0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0]$, *put blue close_to green*$\}$ and $\{[0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0]$, *put green below red*$\}$ corresponding to 7 and 24 compatible final configurations respectively.
3. Test set 3 corresponds to all pairs including the initial configuration $c_i = [1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0]$ (29 pairs), with an average of 13 compatible final configurations.
4. Test set 4 corresponds to all pairs including one of the sentences *put green on_top_of red* and *put blue far_from red*, i.e. 20 pairs with an average of 9.5 compatible final configurations.

5. Test set 5 is all pairs that include both the initial configuration of test set 3 and one of the sentences of test set 4, i.e. 2 pairs with 6 and 13 compatible goals respectively. Note that pairs of set 5 are removed from sets 3 and 4.

Table 2: List of instructions. Each of them specifies a shift of one predicate, either from false to true ($0 \rightarrow 1$) or true to false ($1 \rightarrow 0$). **block A** and **block B** represent two different blocks from {red, blue, green}.

Transition type	Sentences
Close $0 \rightarrow 1$ ($\times 3$)	*Put* ***block A*** *close_to* ***block B***, *Bring* ***block A*** *and* ***block B*** *together*, *Get* ***block A*** *and* ***block B*** *close_from each_other*, *Get* ***block A*** *close_to* ***block B***.
Close $1 \rightarrow 0$ ($\times 3$)	*Put* ***block A*** *far_from* ***block B***, *Get* ***block A*** *far_from* ***block B***, *Get* ***block A*** *and* ***block B*** *far_from each_other*, *Bring* ***block A*** *and* ***block B*** *apart*,
Above $0 \rightarrow 1$ ($\times 6$)	*Put* ***block A*** *above* ***block B***, *Put* ***block A*** *on_top_of* ***block B***, *Put* ***block B*** *under* ***block A***, *Put* ***block B*** *below* ***block A***.
Above $1 \rightarrow 0$ ($\times 6$)	*Remove* ***block A*** *from_above* ***block B***, *Remove* ***block A*** *from* ***block B***, *Remove* ***block B*** *from_below* ***block A***, *Put* ***block B*** *and* ***block A*** *on_the_same_plane*, *Put* ***block A*** *and* ***block B*** *on_the_same_plane*.

Testing on logical expressions of instructions. To evaluate DECSTR on logical functions of instructions, we generate three types of expressions:

1. 100 instructions of the form "A and B" where A and B are basic instructions corresponding to shifts of the form *above* $0 \rightarrow 1$ (see Table 2). These intersections correspond to stacks of 3 or pyramids.
2. 200 instructions of the form "A and B" where A and B are *above* and *close* instructions respectively. B can be replaced by "not B" with probability 0.5.
3. 200 instructions of the form "(A and B) or (C and D))", where A, B, C, D are basic instructions: A and C are *above* instructions while B and D are *close* instructions. Here also, any instruction can be replaced by its negation with probability 0.5.

Implementation details. The encoder is a fully-connected neural network with two layers of size 128 and *ReLU* activations. It takes as input the concatenation of the final binary configuration and its two conditions: the initial binary configuration and an embedding of the NL sentence. The NL sentence is embedded with an recurrent network with embedding size 100, *tanh* non-linearities and biases. The encoder outputs the mean and log-variance of the latent distribution of size 27. The decoder is also a fully-connected network with two hidden layers of size 128 and *ReLU* activations. It takes as input the latent code z and the same conditions as the encoder. As it generates binary vectors, the last layer uses *sigmoid* activations. We train the architecture with a mixture of Kullback-Leibler divergence loss (KD_{loss}) w.r.t a standard Gaussian prior and a binary Cross-Entropy loss (BCE_{loss}). The combined loss is $BCE_{\text{loss}} + \beta \times KD_{\text{loss}}$ with $\beta = 0.6$. We use an Adam optimizer, a learning rate of 5×10^{-4}, a batch size of 128 and optimize for 150 epochs. As training is fast ($\approx$ 2 min on a single cpu), we conducted a quick hyperparameter search over β, layer sizes, learning rates and latent sizes (see Table 3). We found robust results for various layer sizes, various β below 1. and latent sizes above 9.

Table 3: LGG hyperparameter search. In bold are the selected hyperparameters.

Hyperparam.	Values.
β	[0.5, **0.6**, 0.7, 0.8, 0.9, 1.]
layers size	[**128**, 256]
learning rate	[0.01, **0.005**, 0.001]
latent sizes	[9, 18, **27**]

D BASELINES AND ORACLE

The language-conditioned LB baseline is fully described in the main document.

D.1 EXPERT BUCKETS ORACLE

In the EXPERT BUCKETS oracle, the automatic bucket generation of DECSTR is replaced with an expert-predefined set of buckets using *a priori* measures of similarity and difficulty. To define these buckets, one needs prior knowledge of the set of unreachable configurations, which are ruled out. The 5 predefined buckets contain all configurations characterized by:

- Bucket 1: a single *close* relation between a pair of objects and no *above* relations (4 configurations).
- Bucket 2: 2 or 3 *close* relations and no *above* relations (4 configurations).
- Bucket 3: 1 stack of 2 blocks and a third block that is either away or close to the base, but is not close to the top of the stack (12 configurations).
- Bucket 4: 1 stack of 2 blocks and the third block close to the stack, as well as pyramid configurations (9 configurations).
- Bucket 5: stacks of 3 blocks (6 configurations).

These buckets are the only difference between the EXPERT BUCKETS baseline and DECSTR.

D.2 LGB-C BASELINE

The LGB-C baseline represent goals not as semantic configurations but as particular 3D targets positions for each block, as defined for example in Lanier et al. (2019) and Li et al. (2019). The goal vector size is also 9 and contains the 3D target coordinates of the three blocks. This baselines also implements decoupling and, thus, can be compared to DECSTR in the three phases. We keep as many modules as possible common with DECSTR to minimize the amount of confounding factors and reduce the *under-fitting* bias. The goal selection is taken from DECSTR, but converts semantic configuration into specific randomly-sampled target coordinates for the blocks, see Figure 2. The agent is not conditioned on its current semantic configuration nor its semantic goal configuration. For this reason, we do not apply the symmetry bias. The binary reward is positive when the maximal distance between a block and its target position is below 5 cm, i.e. the size of a block (similar to (Andrychowicz et al., 2017)). To make this baseline competitive, we integrate methods from a state of the art block manipulation algorithm (Lanier et al., 2019). The agent receives positive rewards of 1, 2, 3 when the corresponding number of blocks are well placed. We also introduce the multi-criteria HER from Lanier et al. (2019). Finally, we add an additional object-centered inductive bias by only considering, for each Deep Sets module, the 3D target positions of the corresponding pair.That is, for each object pair, we ignore the 3D positions of the remaining object, yielding to a vector of size 6. Language grounding is based on a C-VAE similar to the one used by DECSTR. We only replace the cross-entropy loss by a mean-squared loss due to the continuous nature of the target goal coordinates. We use the exact same training and testing sets as with semantic goals.

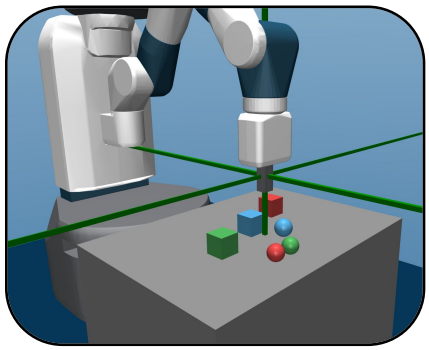

Figure 2: The LGB-C baseline samples target positions for each block (example for a pyramid here).

E ADDITIONAL RESULTS

E.1 COMPARISON DECSTR - LGB-C IN SKILL LEARNING PHASE

Figure 3 presents the average success rate over the 35 valid configurations during the skill learning phase for DECSTR and the LGB-C baseline. Because LGB-C cannot pursue semantic goals as such, we randomly sample a specific instance of this semantic goal: target block coordinates that satisfy the constraints expressed by it. Because LGB-C is not aware of the original semantic goal, we cannot measure success as the ability to achieve it. Instead, *success* is defined as the achievement of the corresponding specific goal: bringing blocks to their respective targets within an error margin of 5 cm each. In short, DECSTR targets semantic goals and is evaluated on its ability to reach them. LGB-C targets specific goals and is evaluated on its ability to reach them. These two measures do not match exactly. Indeed, LGB-C sometimes achieves its specific goal but, because of the error margins, does not achieve the original semantic goal.

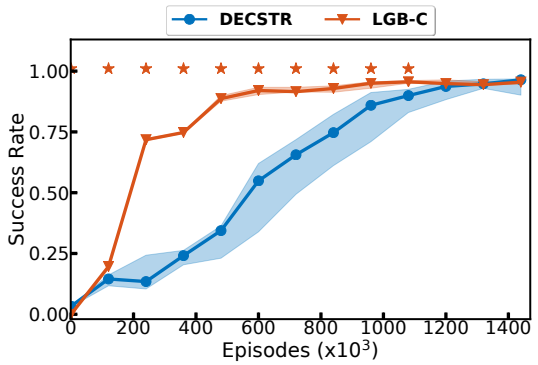

Figure 3: Comparison DECSTR and LGB-C in the skill learning phase.

E.2 AUTOMATIC BUCKET GENERATION.

Figure 4 depicts the evolution of the content of buckets along training (epochs 1, 50 and 100). Each pie chart corresponds to a reachable configuration and represents the distribution of configurations into buckets across 10 different seeds. Blue, orange, green, yellow, purple represent buckets 1 to 5 respectively and grey are undiscovered configurations. At each moment, the discovered configurations are equally spread over the 5 buckets. A given configuration may thus change bucket as new configurations are discovered, so that the ones discovered earlier are assigned buckets with lower indexes. Goals are organized by their bucket assignments in the *Expert Buckets* condition (from top to bottom).

After the first epoch (left), DECSTR has discovered all configurations from the expert buckets 1 and 2, and some runs have discovered a few other configurations. After 50 epochs, more configurations have been discovered but they are not always the same across runs. Finally, after 100 epochs, all configurations are found. Buckets are then steady and can be compared to expert-defined buckets. It seems that easier goals (top-most group) are discovered first and assigned in the first-easy buckets (blue and orange). Hardest configurations (stacks of 3, bottom-most group) seem to be discovered last and assigned the last-hardest bucket (purple). In between, different runs show different compositions, which are not always aligned with expert-defined buckets. Goals from expert-defined buckets 3 and 4 (third and fourth group from the top) seem to be attributed different automatic buckets in different runs. This means that they are discovered in different orders depending on the runs. In summary, easier and harder goals from expert buckets 1 - 2 and 5 respectively seem to be well detected by our automatic bucket generations. Goals in medium-level expected difficulty as defined by expert buckets seem not to show any significant difference in difficulty for our agents.

E.3 DECSTR LEARNING TRAJECTORIES

Figure 5 shows the evolution of internal estimations of the competence C, the learning progress LP and the associated sampling probabilities P. Note that these metrics are computed online by

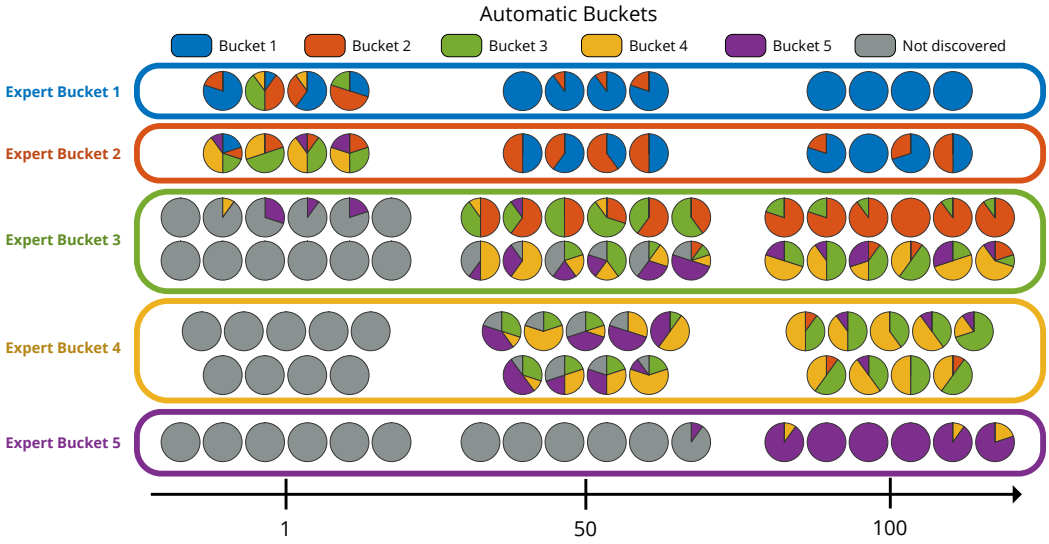

Figure 4: Evolution of the content of buckets from automatic bucket generation: epoch 1 (2400 episodes, left), 50 (middle) and 100 (right). Each pie chart corresponds to one of the 35 valid configurations. It represents the distribution of the bucket attributions of that configuration across 10 runs. Blue, orange, green, yellow, purple represent automatically generated buckets 1 to 5 respectively (increasing order of difficulty) and grey represents undiscovered configurations. Goals are organized according to their expert bucket attributions in the *Expert Buckets* condition (top-bottom organization).

DECSTR, as it self-evaluates on random discovered configurations. Learning trajectories seem to be uniform across different runs, and buckets are learned in increasing order. This confirms that the time of discovery is a good proxy for goal difficulty. In that case, configurations discovered first end up in the lower index buckets and are indeed learned first. Note that a failing automatic bucket generation would assign goals to random buckets. This would result in uniform measures of learning progress across different buckets, which would be equivalent to uniform goal sampling. As Main Figure **??** shows, DECSTR performs much better than the *random goals* conditions. This proves that our automatic bucket algorithm generates useful goal clustering.

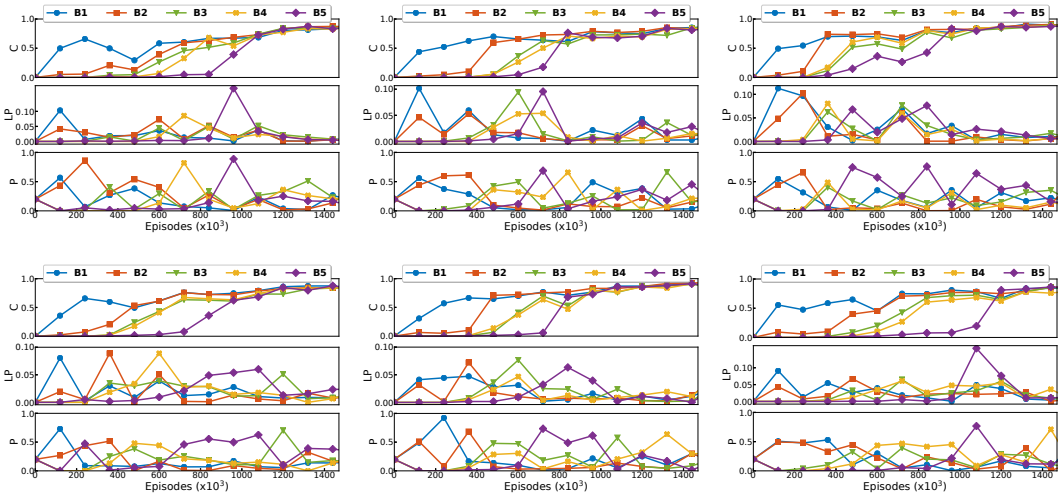

Figure 5: Learning trajectories of 6 DECSTR agents.

REFERENCES

Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight Experience Replay. *arXiv preprint arXiv:1707.01495*, 2017.

Cédric Colas, Pierre-Yves Oudeyer, Olivier Sigaud, Pierre Fournier, and Mohamed Chetouani. CURIOUS: Intrinsically motivated multi-task, multi-goal reinforcement learning. In *International Conference on Machine Learning (ICML)*, pp. 1331–1340, 2019.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.

John B. Lanier, Stephen McAleer, and Pierre Baldi. Curiosity-driven multi-criteria hindsight experience replay. *CoRR*, abs/1906.03710, 2019. URL `http://arxiv.org/abs/1906.03710`.

Richard Li, Allan Jabri, Trevor Darrell, and Pulkit Agrawal. Towards practical multi-object manipulation using relational reinforcement learning. *arXiv preprint arXiv:1912.11032*, 2019.

Tom Schaul, Daniel Horgan, Karol Gregor, and David Silver. Universal value function approximators. In *International Conference on Machine Learning*, pp. 1312–1320, 2015.

Kihyuk Sohn, Honglak Lee, and Xinchen Yan. Learning structured output representation using deep conditional generative models. In *Advances in neural information processing systems*, pp. 3483–3491, 2015.