HIERARCHICAL ABSTRACTION FOR COMBINATORIAL GENERALIZATION IN OBJECT REARRANGEMENT

Anonymous authors
Paper under double-blind review

ABSTRACT

Object rearrangement is a challenge for embodied agents because solving these tasks requires generalizing across a combinatorially large set of underlying entities that take the value of object states. Worse, these entities are often unknown and must be inferred from sensory percepts. We present a hierarchical abstraction approach to uncover these underlying entities and achieve combinatorial generalization from unstructured inputs. By constructing a factorized transition graph over clusters of object representations inferred from pixels, we show how to learn a correspondence between intervening on states of entities in the agent's model and acting on objects in the environment. We use this correspondence to develop a method for control that generalizes to different numbers and configurations of objects, which outperforms current offline deep RL methods when evaluated on a set of simulated rearrangement and stacking tasks. [1]

1 INTRODUCTION

A core property of intelligence is the ability to re-purpose previously acquired knowledge for solving new problems, but identifying symmetries from sensorimotor data for factorizing modeling and control into independent components has been a challenge for artificial agents, which generally fall into one of two paradigms. One paradigm runs orthogonal to the above problem by assuming human-defined abstractions to begin with (Ghallab et al., 1998; Fikes & Nilsson, 1971), including the abstraction of natural language that powers large pre-trained models (Bommasani et al., 2021). Approaches in the other paradigm do learn purely from sensorimotor data, but generally learn monolithic representations that offer no clear mechanism for reuse (Hafner et al., 2019; Schrittwieser et al., 2020).

Object rearrangement (Batra et al., 2020) offers an intuitive setting for studying this problem of learning reusable abstractions. Because the space of object configurations is combinatorially large, solving novel rearrangement problems requires the agent to recognize that the same action for moving an object from one location to another can be reused for different objects in different contexts. We can formulate this as an offline goal-conditioned reinforcement learning (RL) problem, where the agent is trained ~~on~~ to model a dataset of sensorimotor interactions and is evaluated on ~~rearranging~~ outputting a sequence of actions for rearranging objects specified in the input image to satisfy constraints depicted in a goal image.

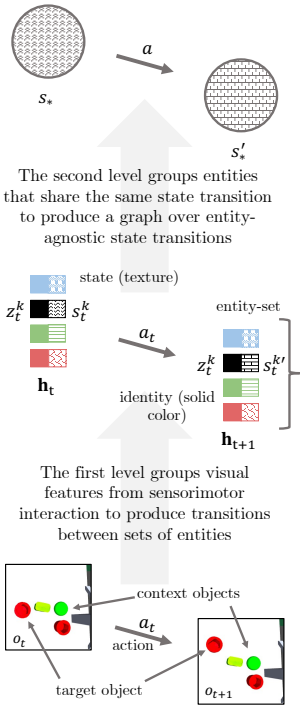


Figure 1: NCS uses a two-level hierarchy to abstract video interactions into a graph over individual state transitions. The entity representing the target object is in black.

Solving rearrangement problems is challenging because it requires long horizon planning in combinatorial task spaces. Offline RL approaches (Kostrikov et al., 2021) that do not explicitly infer factorized representations of objects – which we call entities – struggle to generalize to novel problems with more objects. On the other hand, planning with object-centric methods that do infer entities (Veerapaneni et al., 2020) is not

[1] A step-by-step explanatory video of our method can be found in the supplementary material.

easy because the difficulties of planning with learned single-step models (Janner et al., 2019) are exacerbated in combinatorial spaces.

Instead of planning with parametric models, our work takes inspiration from non-parametric planning methods that have shown success in combining neural networks with graph search to generate long-horizon plans. These methods (Yang et al., 2020; Zhang et al., 2018; Lippi et al., 2020; Emmons et al., 2020) explicitly construct a transition graph from their buffer of experience and plan by searching through the actions recorded in this transition graph with a learned distance metric. The advantage of searching over transition graphs constructed from offline data is the ability to stitch together different path segments in the offline data to solve new problems. The disadvantage is that the non-parametric nature of such methods requires transitions that will be used to solve new problems to have already been recorded in the buffer, which make conventional methods that store observations monolithically ill-suited for combinatorial generalization. For example, Fig. 2b shows that the same state transition can manifest for different objects and in different contexts, but monolithic non-parametric methods do not have learning constraints that enable them to recognizes that all scenarios exhibit the same state transition at an abstract level, which may induce an blowup in the number of nodes of the search graph. To overcome this problem, we devise a method that explicitly represents the similarity among such state transitions in different contexts.

Our method, **Neural Constraint Satisfaction** (NCS), marries the strengths of non-parametric planning with the strengths of object-centric representations. The key idea is to factorize each entity h^k extracted from an object-centric method into two parts as shown in Fig. 1: its ~~identity~~type z^k (e.g. color and shape) and its state s^k (e.g. location), ~~and enforce~~ by enforcing that the ~~identity~~type is the part that is not affected by action and the state is the part that is. The modeling component of our approach abstracts the dataset of sensorimotor interactions into a graph over state transitions of individual entities. This is done by constructing a two-level hierarchy of representations, where the first level abstracts visual features into sets of entity representations (extracting objects from pixels), and the second abstracts transitions over sets of entities into state transitions of individual entities (translations of single isolated objects). By construction, a state transition of an entity is agnostic to its ~~identity~~type ~~of the entity or~~ and the context entities present in the sensorimotor interaction from which it was learned, enabling it to be reused across multiple entities and contexts (e.g. Fig. 2b). The nodes of the graph correspond to states (e.g. locations), and the edges are labelled with actions (e.g. pick-and-move primitives) that transformed the entities between the corresponding pair of states (Fig. 3). The planning and control component of our approach then re-composes sequences of state transitions to solve new rearrangement problems by inferring what state transition can be taken given only the current and goal image observations. This is done by inferring goal entities as *constraints* on the states that the current entities should satisfy by matching their identities, *binding* a current entity and its corresponding goal constraint to nodes in the transition graph by matching their states, and using the state transition between the two nodes to retrieve what action to take.

~~Our contributions are:~~ Our main contribution is an algorithm for simulated pick-and-move rearrangement tasks that operates directly on pixel input and generalizes across a combinatorial space of different object configurations. Specifically: (1) in §2 we introduce a POMDP problem formulation that explicitly models the independence, symmetry, and factorization of latent entities. This introduces two challenges, the correspondence problem and combinatorial problem, that we tackle in this paper with a single unified method. (2) In §3, we ~~(1) We~~ present NCS as a neurosymbolic planning method that plans over entities as emergent symbols. ~~(2)~~ (3) The modeling component of NCS is the first object-centric method with a two-level hierarchy of discrete abstractions tailored for control (Fig. 1). ~~(3)~~ (4) The planning and control component of NCS presents a non-parametric planning approach for combinatorial task spaces. ~~(4) Our~~ (5) In §4 our results show that NCS outperforms both state-of-the-art offline RL methods and object-centric shooting-based planning methods across three simulated rearrangement problems.

2 Goal-Conditioned Reinforcement Learning with Entities

This section introduces a set of modifications to the standard goal-conditioned partially observed Markov decision process (POMDP) problem formulation that explicitly exposes the combinatorial structure of object rearrangement tasks of the following kind: "given an image o_1 of a set of objects on a tabletop and a goal image o_g that depicts a subset (or all) of the objects in new locations, execute a series of pick-and-move action primitives to rearrange the objects to satisfy the constraints

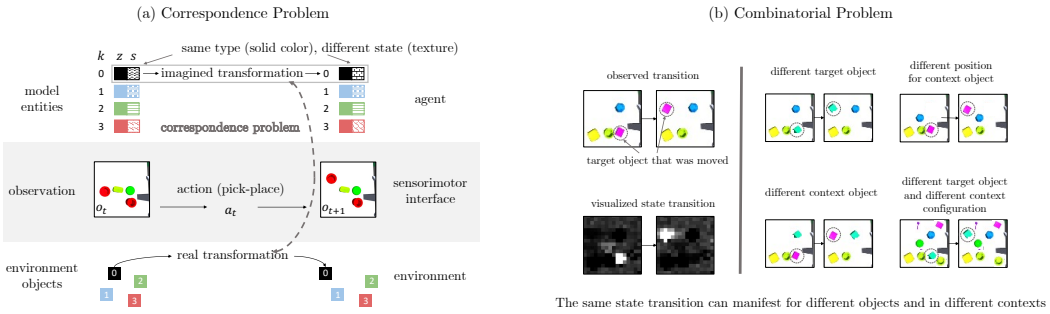


Figure 2: **Solving object rearrangement requires solving two challenging problems**. (a) The **correspondence problem** is the problem of abstracting raw sensorimotor signal into representations of entities such that there is a correspondence between how an agent intervenes on an entity and how its action affects an object in the environment. k denotes the index of the entity, z denotes its ~~identity~~type (shown with solid colors), and s denotes its state (shown with textures). The entity representing the target object that was moved is in black. (b) The **combinatorial problem** is the problem of representing the combinatorial space of tasks in a way that captures common state transitions in a way that is agnostic to the ~~identity~~type of the target object that was moved (indicated by the dotted circle) and to the configuration of context objects in the rest of the scene.

depicted in the goal image." We assume an offline RL setting, where the agent is trained on a dataset of transitions $\{(o_1, a_1, ...a_{T-1}, o_T)\}_{n=1}^{N}$ and evaluated on tasks specified as (o_1, o_g).

The standard problem formulation is defined as $\mathcal{M} = (\mathcal{O}, \mathcal{A}, \mathcal{H}, \mathcal{G}, E, P, R)$, with observation space $\mathcal{O}$, action space $\mathcal{A}$, latent space $\mathcal{H}$, goal space $\mathcal{G}$, observation function $E : \mathcal{H} \to \mathcal{O}$, transition function $P : \mathcal{H} \times \mathcal{A} \to \mathcal{H}$, and reward function $R : \mathcal{H} \times \mathcal{G} \to \mathbb{R}$. The issue with monolithically modeling the latent space this way is that it exposes no mechanism for modeling commonalities among different scenes, such as scenes that share the same object, scenes that contain objects in the same location, or a scene with multiple instances of the same type of object. Not modeling this structure ~~means each scene in the combinatorial space of object configurations is treated as unrelated from any other scene. This~~ prevents us from designing control algorithms that exploit commonalities among scenes to reduce the intractability of searching the combinatorial space.

We thus introduce three structural assumptions to the standard formulation. The first is *independence*, that the latent space decomposes into independent subspaces as $\mathcal{H} = \mathcal{H}^1 \times ... \times \mathcal{H}^K$, each representing an independent degree of freedom, one for each object we intend to model. The second is *symmetry*, that the latent subspaces are not only independent but also symmetric, i.e. $\mathcal{H}^1 = ... = \mathcal{H}^K$, which manifests as the permutation invariance of E and R and the permutation equivariance of P (i.e. the functions for rendering, goal constraint satisfaction evaluation, and transformation under actions are shared across all objects regardless of index). We define an **entity**[2] $h \in \mathcal{H}^k$ as an independent degree of freedom that is processed symmetrically, and an **entity-set** as the set of entities $\mathbf{h} = (h^1, ..., h^K)$ that explain an observation Diuk et al. (2008); Weld. The third is *factorization*, that each entity h is factorized into ~~an identity~~a type $z \in \mathcal{Z}$ and a state $s \in \mathcal{S}$, i.e. $\mathcal{H}^k = \mathcal{Z} \times \mathcal{S}$, where z is the part that is not affected by action (e.g. its color and shape) and s is the part that is (e.g. its location), represented in Fig. 1 by the solid color and texture respectively Boutilier et al. (2000); Guestrin et al. (2003a). These assumptions – independence, symmetry, and factorization – model intuitive properties of physical objects: objects can be independently acted upon without affecting other objects, the same physical laws apply to all objects, and acting on an object does not change its identity. With such a formulation, it is natural to express goals as a set of constraints $\mathbf{h}_g = (h_g^1, ..., h_g^k)$, and to solve a task is to take actions that transforms the subset of entities in the initial observation o_1 whose identities are given by $\mathbf{z}_g$ to new states specified by $\mathbf{s}_g$.

This new formulation solves the problem of modeling the commonalities among different scenes stated above. ~~To model scenes that share the same object, we assign an entity from each scene to all share the same identity z.~~ To model scenes that contain objects in the same location, we assign the same state s to an entity from each scene. To model a scene with multiple instances of the same type of object, we assign multiple entities in the scene to share the same ~~identity~~type z.

[2]We use "object" to refer to an independent degree of freedom in the environment, and "entity" to refer to the agent's representation of the object.

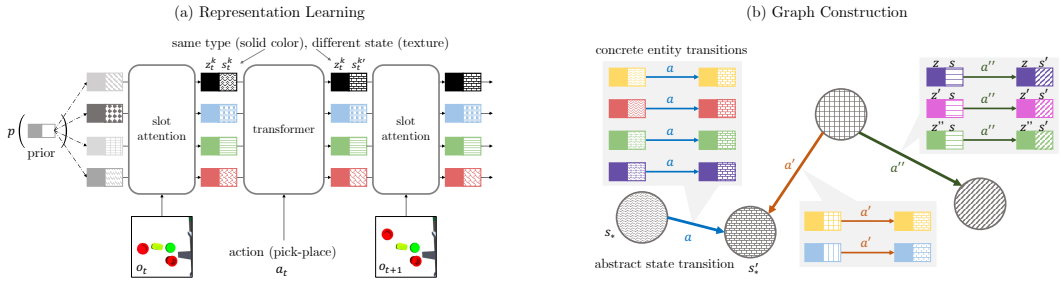


Figure 3: **Modeling**. The modeling component of NCS constructs a two-level abstraction hierarchy. (a) **Level one:** NCS learns to infer a set of entities from sensorimotor transitions with pick-and-move actions, in which one entity is moved per transition. We explicitly enforce that the identity z (shown with solid colors) of an entity remains unchanged between time-steps. The transformer dynamics model learns to sparsely predict the states s (shown with textures) of the entities at the next time-step. Empirically, NCS learns to modify only the state of the entity (shown in black) that it uses to represent the target object that was moved in the environment. *This solves the correspondence problem by forcing the network to use predict and reconstruct observations through the entity bottleneck.* (b) **Level two:** NCS abstracts transitions over entity-sets into transitions over individual states, constructing a graph where states are nodes and transitions between them are edges. This is done by clustering entity transitions that share similar initial states and similar final states. *This solves the combinatorial problem by constructing the state transitions to be agnostic to the identities or contexts of the entities from which the transitions were derived.*

Exposing this structure in our problem formulation enables us to exploit it. To do so requires solving two problems. The first is the **correspondence problem**, which is the problem of learning to represent the space $\mathcal{H}$ (by training the weights of models of E, P, and R) such that there is a correspondence between how the agent models how actions affect entities and how actions actually affect objects in the environment. The source of the correspondence problem is that the agent only receives information about the environment through the sensorimotor interface and does not know what each $\mathcal{H}^k$ should semantically mean *a priori* (e.g. "h^1 and h^2 refer to things, but I do not yet know what"), unlike standard POMDP formulations like object-oriented MDPs, relational MDPs, and factorized MDPs that do attach human-defined semantics to $\mathcal{H}^k$ (e.g. "h^1 refers to this cube, h^2 refers to that ball"). The second is the **combinatorial problem**, which is the problem of efficiently searching the combinatorial space of $|\mathcal{Z}|^K \times \frac{|\mathcal{S}|!}{(\mathcal{S}-K)!}$ entity configurations. This leads to a natural evaluation criterion, which is to test to what extent an agent can zero-shot-generalize to solve rearrangement tasks involving a new set of object configurations that is disjoint from the object configurations observed in training, assuming that the training configurations have collectively covered $\mathcal{Z}$ and $\mathcal{S}$.

To focus on the combinatorial nature of rearrangement, we are not interested in low-level manipulation, so we represent the action space as a set of primitives $(w, \Delta w)$, with Cartesian coordinates $w = (x, y, z)$, where actions sparsely affect one entity at a time and how an action affects an object's state does not depend on its identity. We are not interested in representing occluded objects, so we assume that objects are all constrained to the xy plane or xz plane and are directly visible to the camera. Following prior work (Hansen-Estruch et al., 2022; Castro et al., 2009), we make a *bisimulation* assumption that the state space can be partitioned into a finite set of equivalence classes, and that there is one action primitive that transitions between each pair of equivalence classes.

3 Neural Constraint Satisfaction

We present our method, Neural Constraint Satisfaction (NCS), for solving object rearrangement problems from pixels. Our main idea is that generalizing over such a combinatorial space requires the agent to recognize that the same action for moving an object from one location to another can be reused for different objects and in different contexts. Concretely, given a transition $o, a \rightarrow o'$ from the training set in which a single object k has been moved, the modeling problem is to represent the state transition $s^k \rightarrow s^{k\prime}$ as decoupled from the ~~identity~~type z^k of the affected entity and as decoupled from the other context entities $h^{\neq k}$ in the scene. The control problem is then to apply a composition of previously seen state transitions to satisfy goal constraints on new object configurations.

3.1 MODELING

Our modeling approach abstracts the training dataset of action-conditioned videos into a factorized graph over state transitions that can be reused across different rearrangement problems. Constructing such a graph decomposes into two representation learning problems: the first concerns how to represent objects in a way that exposes their commonalities and the second concerns how to represent actions in a way that enables them to be reused across these commonalities. Note that these two problems are both different instances of the same fundamental problem of partitioning a space into discrete groups, but at different levels of abstraction: solving the correspondence problem reduces to learning how to cluster visual features into entities and solving the ~~correspondence~~ combinatorial problem reduces to learning how to cluster transitions over entity-sets into transitions over individual states. Our model thus implements a two-level abstraction hierarchy.

Level 1: abstracting visual features into sets of entities The goal of the first level (Fig. 3a) is to map a video transition $o_1, a_1 \rightarrow o_2$ into a transition over entity-sets $(h_1^1, ..., h_1^K) \rightarrow (h_2^1, ..., h_2^K)$. Following our assumptions in §2, the criteria our entity-set transitions should satisfy are (1) entities represent different objects in the scene, i.e. they are *independent* and *symmetric* and (2) only the state s^k of entity k changes, not its ~~identity~~type z^k, i.e. the entity is *factorized*. Because each video transition depicts only a single object being moved, we also want (3) the transition to be *sparse*, i.e. only one entity h^k changes in the transition and the rest $h^{\neq k}$ remain unchanged.

We manually enforce independence, symmetry, and factorization but let sparsity emerge. We formulate the inference of entity-sets as the application of an equivariant action-conditioned sequential Bayesian filter with a mixture model as the latent state, where entity representations are the parameters of the mixture components. Given the connection between the slots produced by slot attention (SA) (Locatello et al., 2020b) and mixture components Chang et al. (2022), we concretely implement this filter by treating the slots produced by the slot attention module of the state-of-the-art SLATE architecture (Singh et al., 2022) as entities and temporally evolving them with a transformer decoder (GPT) (Vaswani et al., 2017; Radford et al., 2018). We split the entity $h \in \mathbb{R}^n$ into a ~~identity~~type half $z \in \mathbb{R}^{\frac{n}{2}}$ and a state half $s \in \mathbb{R}^{\frac{n}{2}}$. With $\mathbf{h} = (h^1, ..., h^K)$,

$$\mathbf{h}_{t+1} = SA(\hat{\mathbf{h}}_t, o_{t+1}) \qquad \hat{\mathbf{s}}_{t+1} = GPT\left(\text{queries} = \mathbf{s}_t, \text{keys/values} = [\mathbf{s}_t, a_t]\right), \tag{1}$$

where $\hat{\mathbf{h}}_{t+1} = [\mathbf{z}_t, \hat{\mathbf{s}}_{t+1}]$ and $\hat{\mathbf{h}}_0$ are independent samples from a Gaussian. This approach enforces independence because $\hat{\mathbf{h}}_0$ are independent, symmetry because the transformer is equivariant, and factorization because $\hat{\mathbf{z}}_{t+1}$ is copied from $\mathbf{z}_t$. In the object rearrangement setting, we find this is sufficient supervision to cause s to capture location and z to capture shape and color, and ~~Empirically, we find~~ that this architecture models sparse changes in the slots (Fig. 4c). We call this implementation **dynamic SLATE** (dSLATE), but our contribution of how we structurally enforce the above three criteria remains invariant to *how* the sequential Bayesian filter is implemented, which we expect will advance beyond using SA and GPT.

Level 2: abstracting transitions over sets of entities into individual state transitions The goal of the second level (Fig. 3b) is to construct a state transition graph from a buffer of entity-set transitions $\{(\mathbf{h}_1, ...\mathbf{h}_T)\}_{n=1}^N$ produced from the first level. Specifically, we seek each edge of this graph to represent a state transition that has been shared across multiple entities in different contexts in the training dataset, such that we can reuse such state transitions when choosing actions for solving new rearrangement problems. Given our bisimulation assumption that observed locations can be partitioned into a finite number of groups, this goal translates into a problem of clustering state transitions, and the problem of clustering state transitions reduces to clustering the states before and after the transition. We treat each cluster centroid as a node in the state transition graph, and an edge between nodes is tagged with the single action that transforms one node's state to another's. The nodes represent equivalence classes among the states and the edges represent actions that transform a state from one class to another~~e~~. The algorithm for constructing the graph is shown in Alg. 1.

To create the nodes, the first step is to identify the entity h^k that dSLATE predicted was affected by a_t in each transition $(\mathbf{h}_t, a_t, \mathbf{h}_{t+1})$ e.g. which object was moved. Our `isolate` procedure achieves this by solving $k = \arg\max_{k' \in \{1,...,K\}} d(s_t^{k'}, s_{t+1}^{k'})$ to identify the index of the entity whose state has most changed during the transition, where $d(\cdot, \cdot)$ is a distance function, detailed in ~~the~~ Table 2 of the Appendix. Having converted the buffer of transitions over entity sets $\{(\mathbf{h}_1, ...\mathbf{h}_T)\}_{n=1}^N$ to a buffer of individual entity transitions $\{(h_t^k, a_t, h_{t+1}^k)\}_{n=1}^{N \times (T-1)}$, and thus a buffer of individual state transitions $\{(s_t^k, a_t, s_{t+1}^k)\}_{n=1}^{N \times (T-1)}$, the second step is to cluster similar transitions together

by clustering the states before and after each transition. Our `cluster` procedure achieves this by clustering over all states together, which is sufficient since the end state of one transition is the starting state of another. A proper clustering should return centroids $\{s_*\}_{m=1}^{M}$ that all correspond to actual states of entities, which we hypothesize will correspond to actual locations in the training set based on how dSLATE is trained.

The purpose of the edges between the nodes is to record an action that transforms an entity from one state to another. For each entity transition (h_t^k, a_t, h_{t+1}^k) we `bind` entity h_t^k and h_{t+1}^k to their associated nodes $s_*^{[i]}$ and $s_*^{[j]}$ and create an edge between $s_*^{[i]}$ and $s_*^{[j]}$ tagged with action a, overwriting previous edges based on the assumption that with a proper clustering there should only be one action primitive per pair of nodes. Applying the `bind` procedure to an entity h^k returns the index of the cluster of the centroid s_* that is nearest to the entity's state s^k. For our experiments `cluster` is implemented with K-means and `bind` is implemented with nearest neighbors, both using the same distance metric ~~used for K-means~~ (see Table 2 in the Appendix), but other clustering algorithms and distance metrics can also be used. In our experiments (Fig. 7), we ~~also~~ show that it is also possible to have more than one action primitive per pair of nodes as long as these actions all map between states bound to the same pair of nodes.

Algorithm 1 Building the Graph

input `model`, `buffer`
for $\{(o_t, a_t, o_{t+1})\}_n$ in `buffer` **do**
  # infer entities from transition
  $\{(\mathbf{h}_t, a_t, \mathbf{h}_{t+1})\}_n \leftarrow$ `model` $(\{o_t, a_t, o_{t+1}\}_n)$.
  # identify which entity changed in transition
  $\{(h_t^k, a_t, h_{t+1}^k)\}_n \leftarrow$ `isolate` $(\{(\mathbf{h}_t, a_t, \mathbf{h}_{t+1})\}_n)$
end for
# partition transitions by clustering entities
$\{s_*\}_{m=1}^{M} \leftarrow$ `cluster` $\left(\{(s_t^k, a_t, s_{t+1}^k)\}_{n=1}^{N}\right)$
transitions between clusters are edges
initialize `graph` with nodes $s_*^{[m]}$, for $m \in [1 : M]$
for each $\{(h_t^k, a_t, h_{t+1}^k)\}_n$ **do**
  # infer cluster assignments
  $[i], [j] \leftarrow$ `bind` $\left(h_t^k\right)$, `bind` $\left(h_{t+1}^k\right)$
  # tag edge with action a_t
  `graph.edges`$[i, j] \leftarrow$ `create-edge` $\left(s_*^{[i]} \xrightarrow{a_t} s_*^{[j]}\right)$
end for
return `graph`

3.2 Planning and Control

Having constructed a graph of state transitions, our approach for control re-composes sequences of state transitions to solve new rearrangement problems. Specifically, the agent decomposes the rearrangement problem into a set of per-entity subproblems (e.g. initial and goal positions for individual objects), searches the transition graph for a transition that transforms the current entity's state to its goal state, and executes the action tagged with this transition in the environment. This problem decomposition is possible because the transitions in our graph are constructed to be agnostic to ~~identity~~type and context, enabling different rearrangement problems to share solutions to the same subproblems. The core challenge in deciding which transitions to compose is in determining which transitions are *possible* to compose. That is, the agent must determine which nodes in the graph correspond to the given goal constraints and which nodes correspond to the entities in the current observation, but the current entities $\mathbf{h}_t$ and goal constraints $\mathbf{h}_g$ must themselves be inferred from the current and goal observations o_t and o_g, requiring the agent to infer both what to do and how to do it purely from its sensorimotor interface.

Algorithm 2 Action Selection

given `model`, `graph`
input goal o_g, observation o_t
infer goal constraints and current entities
$\mathbf{h_g}, \mathbf{h_t} \leftarrow$ `model` (o_g), `model` (o_t)
align entity indices of $\mathbf{h_t}$ with those of $\mathbf{h_g}$
$\pi \leftarrow$ `align` $(\mathbf{h_t}, \mathbf{h_g})$
permute indices of $\mathbf{h_t}$ according to π
$\mathbf{h_t} \leftarrow (h_t^{\pi[1]}, ..., h_t^{\pi[K]})$
identify kth goal constraint to satisfy next
$k \leftarrow$ `select-constraint` $(\mathbf{h_t}, \mathbf{h_g})$
infer cluster assignments
$[i], [j] \leftarrow$ `bind` $\left(h_t^k\right)$, `bind` $\left(h_g^k\right)$
action that transforms node $[i]$ to node $[j]$
return `graph.edges`$[i, j]$`.action`

Our approach takes four steps, summarized in Alg. 2 and Fig. 4. The first step applies dSLATE to infer $\mathbf{h}_t$ and $\mathbf{h}_g$ from o_t and o_g (e.g. the positions and types of all objects in the initial and goal images). The second step permutes the indices of $\mathbf{h}_t$ and $\mathbf{h}_g$ to `align` the identities of h_t^k with h_g^k to match each entity with its goal (Fig. 4b). This step is necessary because the permutation symmetry of the entities does not guarantee that the entities with the same identities will have the same index

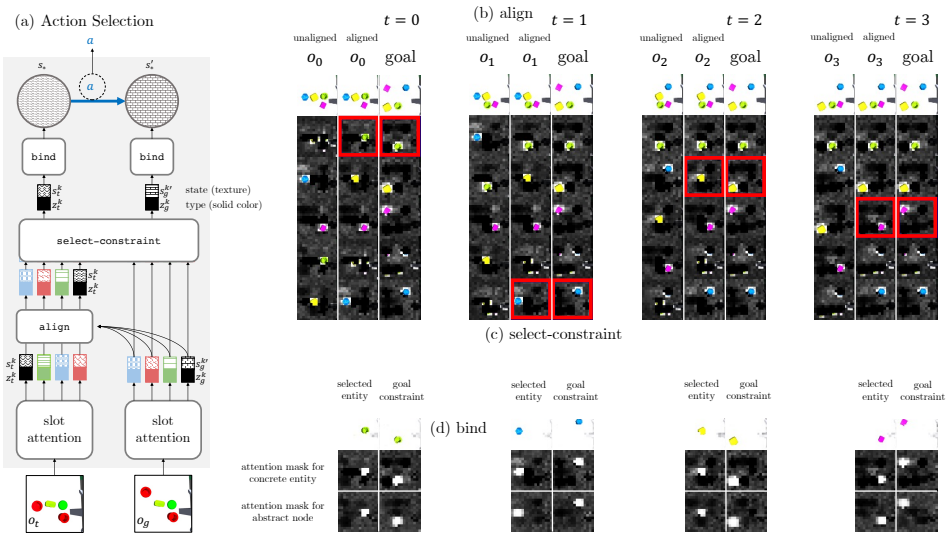


Figure 4: **Planning and control**. Given a rearrangement problem specified only by the current and goal observations (o_0, o_g), NCS decomposes the rearrangement problem into one subproblem (o_t, o_g) per entity. (a) shows the computations NCS uses to solve each subproblem and (b-d) show these steps in context. For each subproblem (o_t, o_g), NCS infers entities from both the current and goal observations. The states of the goal entities indicate constraints on the desired locations of the current entities. (b) NCS first aligns the indices of current and goal entities to have corresponding identities. (c) It selects the index k of the next goal constraint s_g^k to satisfy, as indicated by the red box. The selected goal constraint and current entity are also colored black in (a), and note that their identities are the same but states are different; we want to choose the action to transform the state of the current entity to the state of the goal constraint. (d) It binds the selected goal constraint and its corresponding current entity to nodes s_* and s'_* in the transition graph. Lastly, it identifies the edge connecting those two nodes and executes the action tagged to that edge in the environment.

k, i.e. z_t^k might not correspond to z_g^k. We ~~implement~~ compute this permutation using the Hungarian algorithm that minimizes the matching cost between $\mathbf{z}_t$ and $\mathbf{z}_g$, where the cost between z_t^k and z_g^j is their Euclidean distance. The third step selects which goal constraint h_g^k to satisfy next (Fig. 4c). In the case where objects can move independently, we implement this `select-constraint` procedure by determining which constraint h_g^k has the highest difference in state with its counterpart h_t^k, which reduces to solving the same arg~~x~~max problem as in `isolate`. We discuss how we handle more complex dependencies in Appendix A.3. Lastly, we `bind` h_t^k and ~~h_t^g~~ h_g^k to the graph based on their state components and return the action tagged to the edge between their respective nodes (Fig. 4d). If an edge does not exist between the nodes, then we simply take a random action.

4 Experiments

This section evaluates to what extent NCS can generalize over combinatorially-structured task spaces from perceptual input. The crucial test for combinatorial generalization is to solve a set of tasks from a part of the combinatorial space of transitions over object configurations that is disjoint from the part of the space given for training. The most straightforward way to partition this space is to evaluate on different numbers of objects from training. Concretely, we compare NCS to several offline reinforcement learning baselines on three challenging rearrangement environments and show a significant gap in performance between our method and the next best method. We also study the efficacy of individual components of NCS by comparing against ablations which replace graph search with receding horizon planning (MPC) or do not factorize the entities in the graph (MGS).

Environments. *block-rearrange* is our simplest environment; *robogym-rearrange* and *block-stack* each add complexity along different axes. In *block-rearrange* (Fig. 5a), all objects are the same size, shape, and orientation. $\mathcal{S}$ covers 16 locations in a grid. $\mathcal{Z}$ is the continuous space of red-green-blue values from 0 to 1. *robogym-rearrange* (Fig. 5b) adapts the OpenAI (2020) rearrange environment and removes the assumption from *block-rearrange* that all objects have the same size, shape, ori-

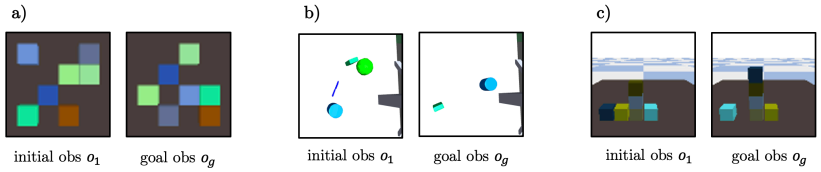


Figure 5: **Environments** Our environments are (a) *block-rearrange*, (b) *robogym-rearrange*, and (c) *block-stacking*. (a) shows a complete specification of goal constraints; (b) shows a partial specification that only specifies the desired locations for two objects; (c) shows the case where some goal constraints are already satisfied in the initial observation, which in this example are the three blocks in the middle column of o_1.

entation, and predefined locations. *block-stack* (Fig. 5c) adds preconditions and postconditions on whether objects can be moved: blocks can only be picked if from the top of a stack, and blocks can only be placed at a given height if there is an object beneath to support it (otherwise it falls).

Experimental setup. We evaluate two settings: *complete* and *partial*. In the *complete* setting, all objects have an associated constraint, whereas in the *partial* setting, only a subset of objects have associated constraints, and this is reflected in the goal image (Fig. 5b). In *block-rearrange*, all constraints are unsatisfied in the start state. In *robogym-rearrange*, four constraints are unsatisfied in the start state. In *block-stacking*, three constraints are unsatisfied in the start state. We measure as the "Fractional Success Rate" the mean change in the number of satisfied constraints divided by the number of initially unsatisfied constraints for each episode.

Baselines. We implement a version of object-centric MPC (Veerapaneni et al., 2020) by replacing our factorized graph search with CEM. We investigate a monolithic ablation of NCS that constructs a graph over transitions between sets of entity states rather than individual entity states (MGS). We compare with state-of-the-art pixel-based behavior cloning (BC) and implicit Q-learning (IQL) implementations based off of Kostrikov (2021). Our last baseline just takes random actions (Rand).

4.1 Quantitative Results

Figure 1 shows that NCS performs significantly better than the baselines in combinatorial generalization (about a 5-10x improvement), suggesting that modeling the properties of independence, symmetry, and factorization of entities enables NCS to collapse the combinatorial structure of the task space. We trained all methods on an offline dataset of 5000 trajectories showing only 4 objects. We evaluate on 4-7 objects for 100 episodes across 10 seeds. These task spaces are enormous: if we discretize the number of colors to 256, the number of possible configurations of k objects in the simplest environment of *block-rearrange* is $256^k \times \frac{16!}{(16-k)!}$ which for $k = 7$ is 4.2×10^{24}. Most of the baselines perform no better or only slightly better than random. Although in theory an equivariant relational dynamics model as in (Veerapaneni et al., 2020) also collapses the combinatorial structure of the task space, in practice it is difficult to perform shooting-based planning with a model trained to predict a single step forward (Janner et al., 2019). As a result, our MPC baseline performs poorly because its rollouts are poor, and it is significantly more computationally expensive to run (11 hours instead of 20 minutes). Additional results are in Appendix.

4.2 Qualitative Analysis

Fig. 4b provides a qualitative visualization of key steps in NCS for *robogym-rearrange*, visualizing the `align`, `select-constraint`, and `bind` procedures. We observe that NCS discovers that states and identities correspond to object locations and appearance respectively. Fig. 6 shows a t-SNE (Van der Maaten & Hinton, 2008) plot that clusters entities inferred from the *robogym* environment. One feature we notice about the t-SNE plot is that there are multiple cluster indices that map onto similar groups of points. This reveals that multiple different regions of $\mathcal{S}$ appear to be modeling similar states. We also tried merging redundant clusters, but found that this did not improve quantitative performance.

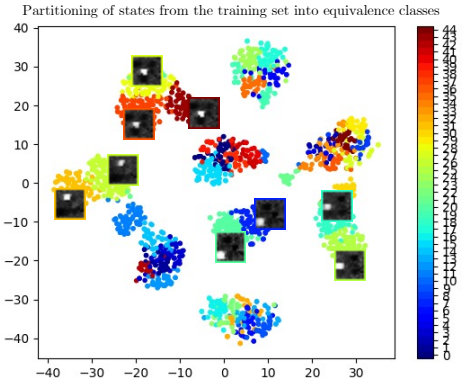


Figure 6: **Nodes as equivalent classes over states** This figure shows the clustering of states inferred for the *robogym* environment, where each cluster centroid is treated as a node in our transition graph. Each cluster is labeled with an attention mask computed by averaging the attention masks that slot attention produces for the entities associated with the cluster.

Table 1: Combinatorial generalization from training data collected from environments with 4 objects reporting fractional success rate. MGS stands for monolithic graph search. Run with 10 seeds, standard error computed.

(a) *block-rearrange*, complete specification.

Method	4	5	6	7
Ours	**0.94** ± 0.01	**0.93** ± 0.00	**0.93** ± 0.00	**0.89** ± 0.00
Rand	0.06 ± 0.02	0.07 ± 0.03	0.07 ± 0.03	0.08 ± 0.03
MPC	0.16 ± 0.06	0.12 ± 0.04	0.11 ± 0.04	0.10 ± 0.03
MGS	0.07 ± 0.03	0.06 ± 0.02	0.07 ± 0.02	0.08 ± 0.03
IQL	0.07 ± 0.01	0.03 ± 0.00	0.02 ± 0.00	0.02 ± 0.00
BC	0.03 ± 0.00	0.02 ± 0.00	0.01 ± 0.00	0.01 ± 0.00

(b) *block-stacking*, complete specification.

Method	4	5	6	7
Ours	**0.58** ± 0.02	**0.48** ± 0.02	**0.35** ± 0.01	**0.24** ± 0.01
Rand	0.02 ± 0.01	0.01 ± 0.01	0.02 ± 0.01	0.02 ± 0.01
MPC	0.03 ± 0.01	0.01 ± 0.01	0.02 ± 0.01	0.01 ± 0.01
MGS	0.13 ± 0.00	0.01 ± 0.01	0.02 ± 0.01	0.01 ± 0.01
IQL	0.19 ±0.02	0.12 ±0.01	0.13 ±0.02	0.07 ±0.01
BC	0.21 ±0.01	0.13 ±0.01	0.13 ±0.01	0.08 ±0.01

(c) *robogym-rearrange*, complete specification.

Method	4	5	6	7
Ours	**0.64** ± 0.01	**0.47** ± 0.01	**0.49** ± 0.01	**0.41** ± 0.01
Rand	0.01 ± 0.0	0.01 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
MPC	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
MGS	0.01 ± 0.0	0.01 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
IQL	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
BC	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0

(d) *block-stacking*, partial specification.

Method	4	5	6	7
Ours	**0.34** ± 0.01	**0.27** ± 0.01	**0.21** ± 0.01	**0.15** ± 0.01
Rand	0.03 ± 0.01	0.03 ± 0.01	0.03 ± 0.01	0.03 ± 0.01
MPC	0.04 ± 0.02	0.03 ± 0.01	0.02 ± 0.01	0.04 ± 0.01
MGS	0.03 ± 0.01	0.03 ± 0.01	0.02 ± 0.01	0.02 ± 0.01
IQL	0.14 ±0.01	0.06 ±0.01	0.04 ±0.00	0.03 ±0.01
BC	0.17 ±0.01	0.07 ±0.01	0.04 ±0.00	0.03 ±0.00

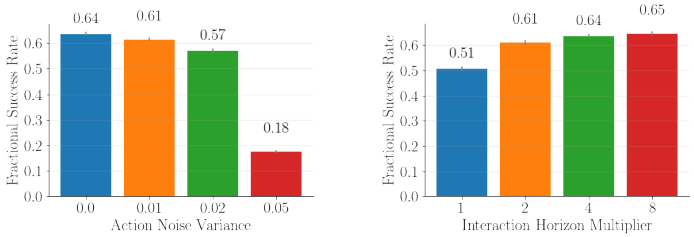


Figure 7: **Stress testing NCS** This figure shows the performance of NCS on *robogym-rearrange* as we vary the amount of noise added to the actions (left) and vary the interaction horizon, defined as a multiple of the minimum steps needed to complete the task (right).

4.3 More challenging settings

We analyzed NCS in more challenging settings that reflect the noisy nature of real-world robotics. As Fig. 7 (left) shows, NCS is robust to the addition of Gaussian noise to the action at every time step, up until the noise variance is comparable to the maximum distance for successful picking and goal placements. The performance remains high given significantly fewer interaction steps (Fig. 7, right).

5 Related Work

The problem of discovering re-composable representations is generally motivated by combinatorial task spaces. The traditional approach to enforcing this compositional inductive bias is to compactly represent the task space with MDPs that have a graph-structured latent space, such as factored MDPs Boutilier et al. (1995; 2000); Guestrin et al. (2003a), relational MDPs Wang et al. (2008); Guestrin et al. (2003b); Gardiol & Kaelbling (2003), and object-oriented MDPs Diuk et al. (2008); Abel et al. (2015). The drawback of these neurosymbolic approaches is that they generally assume entities have manually-defined semantics (Chang et al., 2016; Zadaianchuk et al., 2022; Bapst et al., 2019; Zhang et al., 2018), whereas our work discovers the entity semantics through data.

Other works that discover such entity semantics are so-called object-centric methods (Greff et al., 2017; Van Steenkiste et al., 2018; Greff et al., 2019; 2020; Locatello et al., 2020a; Kipf et al., 2021; Zoran et al., 2021; Singh et al., 2021). However, these approaches evaluate the quality of inferred entities via segmentation metrics, which does not assesses what we want the object representations to *do*. Instead, we directly evaluate our inferred entities by *how well they can reused for solving tasks*. (Goyal et al., 2021; 2020) also seek to represent transformations over entities, but in contrast to their work, which considers the problem of predicting future observations, our work goes a step further by composing transformations for planning and control. Kulkarni et al. (2019) considers how object representations improve exploration, but we consider the offline setting which requires zero-shot generalization. Veerapaneni et al. (2020) also evaluate on control tasks, but their shooting-based

planning method in suffers from compounding errors as other learned single-step models do (Janner et al., 2019), while our non-parametric approach of composing previously seen transitions enables us to plan for longer horizons.

Non-parametric approaches have become popular for long horizon planning (Yang et al., 2020; Zhang et al., 2018; Lippi et al., 2020; Emmons et al., 2020; Zhang et al., 2021), but the drawback of these other approaches is they represent the entire observation monolithically, which causes a blowup of nodes in combinatorial task spaces, making it infeasible for these methods to be applied in rearrangement tasks that require generalizing to novel object configurations with different numbers of objects. Similar to our work, Huang et al. (2019) also tackles rearrangement problems by searching over a constructed latent task graph, but they require a demonstration during deployment time, whereas NCS does not because it reuses context-agnostic state transitions that were constructed during training. Zhang et al. (2021) conducts non-parametric planning directly on abstract subgoals rather than object-centric states — while similar, the downside of using subgoals rather than abstract states is that those subgoals are not used to represent equivalent states and therefore cannot generalize to new states at test time. Our method, NCS, captures both reachability between known states and new, unseen states that can be mapped to the same abstract state.

6 Discussion

We have deconstructed the object rearrangement into the correspondence and combinatorial problems and proposed NCS as a solution to both. By drawing upon the principles of symmetry, independence, and factorization, NCS implements the insight that same action for moving an object from one location to another can be reused for different objects and in different contexts. Thus NCS outperforms various offline RL methods in combinatorial generalization across three simulated object rearrangement problems. NCS can be seen as physically solving a embodied constraint satisfaction probem, where states are variables, identities are variable values, and actions carry out variable assignments. Crucially the variables, their domains, the assignment operator, and the constraints are all learned from the sensorimotor interface, hence the name Neural Constraint Satisfaction.

Limitations and future work. NCS relies on a nonparametric, non-learning-based approach for control to highlight the generalization capability of our representation of the combinatorial task space, but this limits NCS to only composing previously seen transitions for previously seen entities. Collapsing the combinatorial space along state transitions already provides significant gains but does not adapt to the introduction of novel objects at test time. NCS is currently implemented with tools such as SLATE and K-means that have much potential for improvement. We expect future variations of NCS will improve upon our results by replacing SLATE and K-means with their future successors.

References

David Abel, D. Ellis Hershkowitz, Gabriel Barth-Maron, Stephen Brawner, Kevin O'Farrell, James MacGlashan, and Stefanie Tellex. Goal-based action priors. In *ICAPS*, pp. 306–314. AAAI Press, 2015.

Victor Bapst, Alvaro Sanchez-Gonzalez, Carl Doersch, Kimberly Stachenfeld, Pushmeet Kohli, Peter Battaglia, and Jessica Hamrick. Structured agents for physical construction. In *International conference on machine learning*, pp. 464–474. PMLR, 2019.

Dhruv Batra, Angel X Chang, Sonia Chernova, Andrew J Davison, Jia Deng, Vladlen Koltun, Sergey Levine, Jitendra Malik, Igor Mordatch, Roozbeh Mottaghi, et al. Rearrangement: A challenge for embodied ai. *arXiv preprint arXiv:2011.01975*, 2020.

Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

Craig Boutilier, Richard Dearden, and Moises Goldszmidt. Exploiting structure in policy construction. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, pp. 1104–1111, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

Craig Boutilier, Richard Dearden, and Moisés Goldszmidt. Stochastic dynamic programming with factored representations. *Artif. Intell.*, 121(1-2):49–107, 2000.

Berk Calli, Arjun Singh, Aaron Walsman, Siddhartha Srinivasa, Pieter Abbeel, and Aaron M Dollar. The ycb object and model set: Towards common benchmarks for manipulation research. In *2015 international conference on advanced robotics (ICAR)*, pp. 510–517. IEEE, 2015.

Pablo Samuel Castro, Prakash Panangaden, and Doina Precup. Equivalence relations in fully and partially observable markov decision processes. In *Twenty-First International Joint Conference on Artificial Intelligence*. Citeseer, 2009.

Michael Chang, Thomas L Griffiths, and Sergey Levine. Object representations as fixed points: Training iterative refinement algorithms with implicit differentiation. *arXiv preprint arXiv:2207.00787*, 2022.

Michael B Chang, Tomer Ullman, Antonio Torralba, and Joshua B Tenenbaum. A compositional object-based approach to learning physical dynamics. *arXiv preprint arXiv:1612.00341*, 2016.

Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. 2016.

Carlos Diuk, Andre Cohen, and Michael L. Littman. An object-oriented representation for efficient reinforcement learning. In *ICML*, volume 307 of *ACM International Conference Proceeding Series*, pp. 240–247. ACM, 2008.

Scott Emmons, Ajay Jain, Misha Laskin, Thanard Kurutach, Pieter Abbeel, and Deepak Pathak. Sparse graphical memory for robust planning. *Advances in Neural Information Processing Systems*, 33:5251–5262, 2020.

Ben Eysenbach, Russ R Salakhutdinov, and Sergey Levine. Search on the Replay Buffer: Bridging Planning and Reinforcement Learning. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d' Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper/2019/file/5c48ff18e0a47baaf81d8b8ea51eec92-Paper.pdf.

Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3):189–208, 1971. ISSN 0004-3702. doi: https://doi.org/10.1016/0004-3702(71)90010-5. URL https://www.sciencedirect.com/science/article/pii/0004370271900105.

Natalia Gardiol and Leslie Kaelbling. Envelope-based Planning in Relational MDPs. In S. Thrun, L. Saul, and B. Schölkopf (eds.), *Advances in Neural Information Processing Systems*, volume 16. MIT Press, 2003. URL https://proceedings.neurips.cc/paper/2003/file/4a06d868d044c50af0cf9bc82d2fc19f-Paper.pdf.

M. Ghallab, A. Howe, C. Knoblock, D. Mcdermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL—The Planning Domain Definition Language, 1998. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.212.

Anirudh Goyal, Alex Lamb, Phanideep Gampa, Philippe Beaudoin, Sergey Levine, Charles Blundell, Yoshua Bengio, and Michael Mozer. Object files and schemata: Factorizing declarative and procedural knowledge in dynamical systems. *arXiv preprint arXiv:2006.16225*, 2020.

Anirudh Goyal, Aniket Didolkar, Nan Rosemary Ke, Charles Blundell, Philippe Beaudoin, Nicolas Heess, Michael C Mozer, and Yoshua Bengio. Neural production systems. *Advances in Neural Information Processing Systems*, 34:25673–25687, 2021.

Klaus Greff, Sjoerd Van Steenkiste, and Jürgen Schmidhuber. Neural expectation maximization. *arXiv preprint arXiv:1708.03498*, 2017.

Klaus Greff, Raphaël Lopez Kaufman, Rishabh Kabra, Nick Watters, Christopher Burgess, Daniel Zoran, Loic Matthey, Matthew Botvinick, and Alexander Lerchner. Multi-object representation learning with iterative variational inference. In *International Conference on Machine Learning*, pp. 2424–2433. PMLR, 2019.

Klaus Greff, Sjoerd van Steenkiste, and Jürgen Schmidhuber. On the binding problem in artificial neural networks. *arXiv preprint arXiv:2012.05208*, 2020.

C. Guestrin, D. Koller, R. Parr, and S. Venkataraman. Efficient Solution Algorithms for Factored MDPs. *Journal of Artificial Intelligence Research*, 19:399–468, October 2003a. ISSN 1076-9757. doi: 10.1613/jair.1000. URL https://jair.org/index.php/jair/article/view/10341. tex.ids: guestrin2003EfficientSolutionAlgorithms.

Carlos Guestrin, Daphne Koller, Chris Gearhart, and Neal Kanodia. Generalizing plans to new environments in relational mdps. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, IJCAI'03, pp. 1003–1010, San Francisco, CA, USA, 2003b. Morgan Kaufmann Publishers Inc.

Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination. *arXiv preprint arXiv:1912.01603*, 2019.

Philippe Hansen-Estruch, Amy Zhang, Ashvin Nair, Patrick Yin, and Sergey Levine. Bisimulation makes analogies in goal-conditioned reinforcement learning. *arXiv preprint arXiv:2204.13060*, 2022.

De-An Huang, Suraj Nair, Danfei Xu, Yuke Zhu, Animesh Garg, Li Fei-Fei, Silvio Savarese, and Juan Carlos Niebles. Neural task graphs: Generalizing to unseen tasks from a single video demonstration. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 8565–8574, 2019.

Michael Janner, Justin Fu, Marvin Zhang, and Sergey Levine. When to trust your model: Model-based policy optimization. *Advances in Neural Information Processing Systems*, 32, 2019.

Thomas Kipf, Gamaleldin F Elsayed, Aravindh Mahendran, Austin Stone, Sara Sabour, Georg Heigold, Rico Jonschkowski, Alexey Dosovitskiy, and Klaus Greff. Conditional object-centric learning from video. *arXiv preprint arXiv:2111.12594*, 2021.

Ilya Kostrikov. JAXRL: Implementations of Reinforcement Learning algorithms in JAX, 10 2021. URL https://github.com/ikostrikov/jaxrl.

Ilya Kostrikov, Ashvin Nair, and Sergey Levine. Offline reinforcement learning with implicit q-learning. *arXiv preprint arXiv:2110.06169*, 2021.

Tejas D Kulkarni, Ankush Gupta, Catalin Ionescu, Sebastian Borgeaud, Malcolm Reynolds, Andrew Zisserman, and Volodymyr Mnih. Unsupervised learning of object keypoints for perception and control. *NeurIPS*, 2019.

Martina Lippi, Petra Poklukar, Michael C Welle, Anastasiia Varava, Hang Yin, Alessandro Marino, and Danica Kragic. Latent space roadmap for visual action planning of deformable and rigid object manipulation. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 5619–5626. IEEE, 2020.

Francesco Locatello, Dirk Weissenborn, Thomas Unterthiner, Aravindh Mahendran, Georg Heigold, Jakob Uszkoreit, Alexey Dosovitskiy, and Thomas Kipf. Object-centric learning with slot attention. *arXiv preprint arXiv:2006.15055*, 2020a.

Francesco Locatello, Dirk Weissenborn, Thomas Unterthiner, Aravindh Mahendran, Georg Heigold, Jakob Uszkoreit, Alexey Dosovitskiy, and Thomas Kipf. Object-centric learning with slot attention. In *NeurIPS*, 2020b. URL https://proceedings.neurips.cc/paper/2020/hash/8511df98c02ab60aea1b2356c013bc0f-Abstract.html.

OpenAI. Robogym. https://github.com/openai/robogym, 2020.

Luis Pineda, Brandon Amos, Amy Zhang, Nathan O. Lambert, and Roberto Calandra. Mbrl-lib: A modular library for model-based reinforcement learning. *Arxiv*, 2021. URL https://arxiv.org/abs/2104.10159.

Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. *arXiv preprint arXiv:2102.12092*, 2021.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

Gautam Singh, Fei Deng, and Sungjin Ahn. Illiterate DALL·E learns to compose. *arXiv preprint arXiv:2110.11405*, 2021.

Gautam Singh, Fei Deng, and Sungjin Ahn. Illiterate DALL-e learns to compose. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=h0OYV0We3oh.

Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*, pp. 5026–5033. IEEE, 2012.

Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

Sjoerd Van Steenkiste, Michael Chang, Klaus Greff, and Jürgen Schmidhuber. Relational neural expectation maximization: Unsupervised discovery of objects and their interactions. *arXiv preprint arXiv:1802.10353*, 2018.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Rishi Veerapaneni, John D Co-Reyes, Michael Chang, Michael Janner, Chelsea Finn, Jiajun Wu, Joshua Tenenbaum, and Sergey Levine. Entity abstraction in visual model-based reinforcement learning. In *Conference on Robot Learning*, pp. 1439–1456. PMLR, 2020.

C. Wang, S. Joshi, and R. Khardon. First order decision diagrams for relational mdps. *Journal of Artificial Intelligence Research*, 31:431–472, Mar 2008. ISSN 1076-9757. doi: 10.1613/jair.2489. URL http://dx.doi.org/10.1613/jair.2489.

Daniel S Weld. Solving Relational MDPs with First-Order Machine Learning. pp. 8.

Ge Yang, Amy Zhang, Ari S. Morcos, Joelle Pineau, Pieter Abbeel, and Roberto Calandra. Plan2vec: Unsupervised Representation Learning by Latent Plans. In *Proceedings of The 2nd Annual Conference on Learning for Dynamics and Control*, volume 120 of *Proceedings of Machine Learning Research*, pp. 1–12, 2020.

Andrii Zadaianchuk, Georg Martius, and Fanny Yang. Self-supervised reinforcement learning with independently controllable subgoals. In *Conference on Robot Learning*, pp. 384–394. PMLR, 2022.

Amy Zhang, Adam Lerer, Sainbayar Sukhbaatar, Rob Fergus, and Arthur Szlam. Composable Planning with Attributes. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80, pp. 5837–5846. JMLR.org, 2018.

Lunjun Zhang, Ge Yang, and Bradly C Stadie. World model as a graph: Learning latent landmarks for planning. In Marina Meila and Tong Zhang (eds.), *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 12611–12620. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/zhang21x.html.

Daniel Zoran, Rishabh Kabra, Alexander Lerchner, and Danilo J Rezende. Parts: Unsupervised segmentation with slots, attention and independence maximization. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 10439–10447, 2021.

A Implementation Details

This section details the implementation design decisions for each component of NCS.

A.1 Level 1: abstracting visual features into sets of entities

Dynamic SLATE (dSLATE) maps a video demonstration to a sequence of transitions over sets of entities. It consists of two main components: SLATE (Singh et al., 2022) and a transformer (Vaswani et al., 2017; Radford et al., 2018) dynamics model. Because SLATE itself also uses a transformer for generating image reconstructions, dSLATE uses two transformers: one to recover the observation model E and one to recover the dynamics model P of the structured MDP. We use SLATE $\prod_k \mathcal{H}^k \times \mathcal{O} \rightarrow \prod_k \mathcal{H}^k$ to infer entities $\mathbf{h}_t$ from observation o_t and initial guess $\hat{\mathbf{h}}_t$. The entity h^k is also referred to as a *slot* in (Singh et al., 2022; Locatello et al., 2020b) and is split in half as $h^k = (z^k, s^k)$. The first guess for each entity $\hat{h}_1^k$ is sampled independently and identically distributed from a unit Gaussian, whose parameters are also trained. The hyperparameters are given in Tab. 2.

SLATE preprocesses the image with a discrete variational autoencoder (Ramesh et al., 2021) into a grid of image features, encodes these features into a grid of tokens, infers slots from this token grid with Slot Attention (Locatello et al., 2020b), which also produces an attention mask `attn` over the features each slot attends to. These slots are trained using a transformer decoder (Vaswani et al., 2017; Radford et al., 2018) to autoregressively reconstruct the tokens using the slots as keys/values. We use the dynamics model $\prod_k \mathcal{S}^k \times \mathcal{A} \rightarrow \prod_k \mathcal{S}^k$ to predict a guess for $\hat{\mathbf{s}}_{t+1}$ given the action a_t and the inferred entities $\mathbf{s}_t$ from the previous time-step. This dynamics model is implemented also as a transformer decoder, taking the entity states as queries, and the entity states and action as keys/values. This enables us the dynamics model to model the future state of the slots as an equivariant function of how the action affects it and of how it interacts with other entity states.

We trained dSLATE on an offline dataset of 5000 video demonstrations of length five, with each frame transition showing one of four objects being moved to a different location. We used five slots, one more than the number of objects, following the convention used in Van Steenkiste et al. (2018); Veerapaneni et al. (2020).

A.2 Level 2: abstracting transitions over sets of entities into individual state transitions

NCS constructs a graph of state transitions from a buffer of transitions over entity sets produced by dSLATE, as described in Alg. 1. Each transition records the ~~identity~~type z, state s, and `attn` of each entity h of the entity sets $\mathbf{h}_t$ and $\mathbf{h}_{t+1}$, where `attn` is the attention mask that slot attention produces on the visual features for the entity h. We also record the *pre-condition* of the transition, which we explain further in Appdx. A.3. Given that state corresponds to location in the rearrangement context, both s or `attn` can serve as the representation of the state. We found that we obtained better clusterings when we used `attn` as the state for the *block-** tasks and s as the state for the *robogym-rearrange* task, so we used `attn` as the state for the *block-** tasks and s as the state for the *robogym-rearrange* task. We also empirically found that certain choices of distance metric used for K-means `cluster`ing and `bind`ing (implemented as nearest-neighbors) depended on which choice of state representation we used, and this is summarized in Table 3. The K-means implmentation is adapted from `https://github.com/overshiki/kmeans_pytorch`. We found that increasing the number of slot attention iterations improved the entities representations especially when generalizing to more numbers of objects, so even though we dSLATE trained with slot attention three iterations, for inferring the slots from the buffer we used seven iterations. Lastly, we found that the number of clusters used to for K-Means is the most important hyperparameter for creating a graph that reflected the state transitions. We swept over 16 to 50 clusters and report the optimal number of clusters we found in Table 4.

A.3 Action selection

Using dSLATE and the transition graph from NCS, NCS returns which action to execute in the environment given a goal and current observation, as described in Alg. 2. It first infers goal constraints $\mathbf{h}_g$ and current entities $\mathbf{h}_t$ from the goal observation o_g and current observation o_t. It then uses the `align` procedure to align the indices of the entities in $\mathbf{h}_g$ and $\mathbf{h}_t$ and uses the `select-constraint` to choose the index k of the entity to affect. It binds h_t^k and h_g^k to the

Number of epochs		200
Episodes per epoch		5K
Episode length		5
Batch size		32
Peak LR		(see caption)
LR warmup steps		30000
Dropout		0.1
Discrete VAE	Vocabulary Size	4096
	Temp. Cooldown	1.0 to 0.1
	Temp. Cooldown Steps	30000
	LR (no warmup)	0.0003
	Image Size	(see caption)
	Image Tokens	Image Size / 4
transformer decoder	Layers	4
	Heads	4
	Hidden Dim.	192
Slot attention	Slots	5
	Iterations	3
	Slot Heads	1
	Slot Dim. (h^k)	192
	~~identity~~type Dim. (z^k)	96
	State Dim. (s^k)	96
transformer dynamics	Layers	4
	Heads	4
	Hidden Dim.	96

Table 2: **Hyperparameters for training dSLATE** These hyperparameters are almost identical to those found in Singh et al. (2022, Fig. 7), but because dSLATE operates on video demonstrations rather than static images, we changed some hyperparameters to save memory cost. We changed the batch size from 50 to 32, the number of transformer layers and heads from 8 to 4, the number of slot attention iterations from 7 to 3 without observing a significant change in performance. We used a peak learning rate of 0.0002 and an image size of 64 for **-rearrange*. We used a peak learning rate of 0.0003 and an image size of 96 for *block-stacking*.

State representation	`attn`	s
`isolate` distance metric $d(\cdot,\cdot)$	cosine	cosine
`cluster` distance metric	IoU	squared Euclidean
`bind` distance metric	cosine	squared Euclidean

Table 3: **Hyperparameters for constructing the transition graph with NCS**. Given that state corresponds to location in the rearrangement context, both s or `attn` can serve as the representation of the state. We found that we obtained better clusterings when we used `attn` as the state for the *block-** tasks and s as the state for the *robogym-rearrange* task, so we used `attn` as the state for the *block-** tasks and s as the state for the *robogym-rearrange* task.

	block-rearrange	*robogym-rearrange*	*block-stacking*
number of clusters	30	45	47

Table 4: **Number of clusters used for constructing the nodes of the transition graph.**

graph and returns the action associated with the edge between their respective nodes. Because NCS is a non-parameteric method, it could be the case the graph does not contain such an edge. In this case, we sample a random action in the environment, but future work will replace this step with a more sophisticated method.

To implement `align` we use the `scipy.optimize.linear_sum_assignment` implementation of the Hungarian algorithm, with Euclidean distances between the z^k's as the matching cost.

In `select-constraint`, we are given the set of current entities $\mathbf{h}_t$ whose indices are aligned with the goal constraints $\mathbf{h}_g$ and returns the index k of the goal constraint to satisfy next. By NCS' construction, the edge between the nodes that h_t^k and h_g^k are bound to is the state transition that would be executed if the action associated to the edge were taken in the environment. The `select-constraint` procedure consists of three steps: (1) filtering possible transitions from impossible transitions (2) ranking transitions (3) sampling a transition.

Filtering The filtering step implements NCS' model of possibility and impossibility. In the filtering step, we consider, for each k, the transition between the nodes that h_t^k and h_g^k are bound to and mark the transition as possible or impossible. It then returns the indices k over $\mathbf{h}_t$ and $\mathbf{h}_g$ whose associated transition from h_t^k and h_g^k is possible.

According to NCS, a state transition between node $[i]$ and node $[j]$ is *possible* if its preconditions are met and there exists an edge for that state transition in the graph, and *impossible* otherwise. An intuitive example of an impossible transition is to place a block midair at some intended height, but this transition becomes possible if prior to the transition there already exists a stack of blocks that would support the block if the block were to be placed at that intended height. The existence of this supporting stack is thus the *precondition* for the transition to occur, rendering the stacked block *dependent* on the blocks supporting it.

When there are no dependencies among the entities, as in the **-rearrange* tasks where any object can be moved to any open location without considering where other objects are, any transition present in the graph is possible. When there are dependencies among the entities, as in *block-stacking*, we take the precondition of the transition into account. Although a precondition of a transition from node $[i]$ to node $[j]$ could be a function of both the source node $[i]$ and destination node $[j]$, for simplicity in this paper we consider preconditions as only a function of the destination node $[j]$, which rules out the possibility of placing a block in midair like the above example.

Because a precondition in general is a set of constraints that need to be satisfied for the transition to be possible, we represent the precondition of a transition into node $[j]$ (denoting the state $s_*^{[j]}$) as the set of context states $s_*^{[j']}$ that are always present whenever the state $s_*^{[j]}$ is present. Concretely, a block at height 3 at location x is always accompanied by the presence of some block at height 2 and some block at height 1, both at location x; the states denoting (location x, height 2) and (location x, height 1) are the context states of the state (location x, height 3) and therefore serves as its precondition. We thus implement the precondition of a transition into node $[j]$ by recording the indices j' of the nodes of states that are always present when node $[j]$ is a destination node. To test whether a precondition is satisfied for a given scene, we check if all nodes in the precondition set have a corresponding concrete entity that can be bound to it.

Ranking The filtering step removes the indices from the entities $\mathbf{h}_t$ and goal constraints $\mathbf{h}_g$ whose transitions are impossible, yielding a possibly smaller set of entities $\tilde{\mathbf{h}}_t$ and constraints $\tilde{\mathbf{h}}_g$. That is, if $|\mathbf{h}_t| = |\mathbf{h}_g| = K$, then $|\tilde{\mathbf{h}}_t| = |\tilde{\mathbf{h}}_g| = \tilde{K} \leq K$.

The goal of the ranking step is to compute a ranking among the indices of $\tilde{\mathbf{h}}_t$ and $\tilde{\mathbf{h}}_g$ for choosing which index k to actually select to affect with an action. Intuitively, we should rank indices k according to how different s_t^k and s_g^k are because a difference indicates that the constraint h_g^k is not satisfied. We reuse the distance metric $d(\cdot, \cdot)$ used for `isolate` to implement this ranking.

Sampling The goal of the sampling step is to select a $k \in \{1, ..., \tilde{K}\}$ whose associated entity we will affect with an action, given our ranking. One way to do this is to simply choose k as $k = \arg\max_{k' \in \{1,...,\tilde{K}\}} d(s_t^{k'}, s_{t+1}^{k'})$ as in `isolate`, but we empirically found that sampling k from a Categorical distribution whose pre-normalized probabilities are given by $d(s_t^{k'}, s_{t+1}^{k'})$ resulted in better task performance so we used this stochastic sampling approach. One explanation for why using the argmax may be worse is that it relies on the distance metric $d(\cdot, \cdot)$, and the state representation s, to be such that the distance metric flawlessly assigns high value to entities k that need to be moved and low value to entities k that do not need to be moved. But because the state space $\mathcal{S}$ is learned through the dSLATE training process without explicit supervision on the geometry of the space, a pair of points that should be farther apart than another set of points may not be accurately reflected by using a fixed distance metric $d(\cdot, \cdot)$. Future work will investigate imposing explicit supervision on the geometry of $\mathcal{S}$.

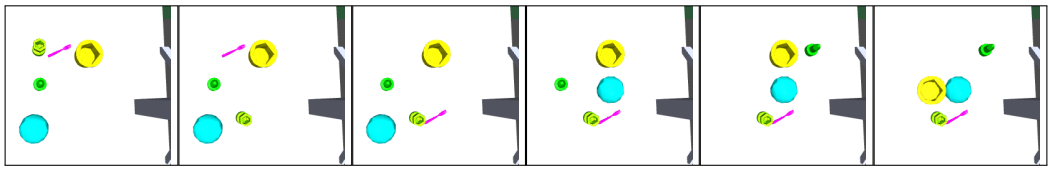


Figure 8: An example of solving a task in the robogym rearrange environment used in this paper.

B Baseline Implementation Details

Random (Rand) The random policy takes actions using `env.action_space.sample()`.

Behavior cloning (BC) This approach trains a policy to output the actions directly taken in the provided dataset. We use an MSE loss to train the policy to imitate the actions.

Implicit Q-learning (IQL) IQL is a simple, offline RL approach that uses temporal difference (TD) learning with the dataset actions and trains a behavior policy value function. To produce an optimal value function, IQL estimates the maximum of the Q-function using expectile regression with an asymmetric MSE using the following objectives:

$$L_V(\psi) = \mathbb{E}_{(s,a)\sim\mathcal{D}}[L_2^\tau(Q_{\hat{\theta}}(s,a) - V_\psi(s))] \text{ where } L_2^\tau(u) = |\tau - \mathbb{1}(u<0)|u^2 \tag{2}$$

$$L_Q(\theta) = \mathbb{E}_{(s,a,s')\sim\mathcal{D}}[(r(s,a) + \gamma V_\psi(s') - Q_\theta(s,a))^2] \tag{3}$$

$$L_\pi(\phi) = \mathbb{E}_{(s,a)\sim\mathcal{D}}[\exp(\beta(Q_{\hat{\theta}}(s,a) - V_\psi(s)))\log \pi_\phi(a|s)]. \tag{4}$$

The $V(s)$ estimates are used for TD-backups and the optimal policy is extracted with advantage-weighted behavioral cloning.

Model predictive control (MPC) This approach uses model predictive control with the cross entropy method (CEM) to select actions, using the transformer dynamics model of dSLATE to perform rollouts in latent space. This is similar to the approached used in OP3 (Veerapaneni et al., 2020), except that we use more recently proposed architectural components (slot attention (Locatello et al., 2020b) instead of IODINE (Greff et al., 2019), a transformer instead of a graph network (Battaglia et al., 2018; Van Steenkiste et al., 2018; Chang et al., 2016)) so our MPC results are not directly comparable to that of OP3. We use the same dSLATE checkpoint that was used for NCS.

We implement this MPC baseline using the `mbrl-lib` library (Pineda et al., 2021) with 10 CEM iterations, an elite ratio of 0.05, and a population size of 250 which was the best configuration we found that fit within a wall clock budget of two days for 8 objects and 100 test episodes. We swept over CEM iterations of $[5, 10, 20]$, elite ratio of $[0.05, 0.1, 0.2]$, and population sizes of $[250, 500, 1000]$, and found that the elite ratio was the most important hyperparameter.

The cost function is computed by first aligning the predicted slots $\mathbf{h}_T$ and goal constraints $\mathbf{h}_g$ using the same `align` procedure in A.3, and then adding up the squared Euclidean distance between slots as $cost = \sum_k (h_T^k - h_g^k)^2$.

Monolithic graph search (MGS) This approach is an ablation to NCS that does not construct a graph over state transitions of individual entities but instead constructs a graph over state transition over entity sets, i.e. each transition is $(\mathbf{s}, a, \mathbf{s}')$ rather than $(s^k, a, s^{k\prime})$. As with MPC, we use the same dSLATE checkpoint that was used for NCS.

The purpose of this ablation is to elucidate the benefit of factorizing the transition graph over entities rather than entity sets. Because nodes in this transition graph represent a set of entity states rather than individual entity states, we use Dijkstra's algorithm, as in (Eysenbach et al., 2019; Yang et al., 2020; Zhang et al., 2018) to plan a unbroken path from the node the initial observation is bound to to the node a goal observation is bound to. For each time-step, we plan a path along the nodes using Dijkstra's algorithm, then return the action associated with the first edge along that path. Like NCS, MGS is a non-parametric model, which means that for a set of entities to be bound to a node in the graph, that node must contain the exact set of entity states corresponding to the states of the entities. If we do not successfully bind to the graph, or if we do not find a path between the current node and the goal node, we sample a random action as NCS does.

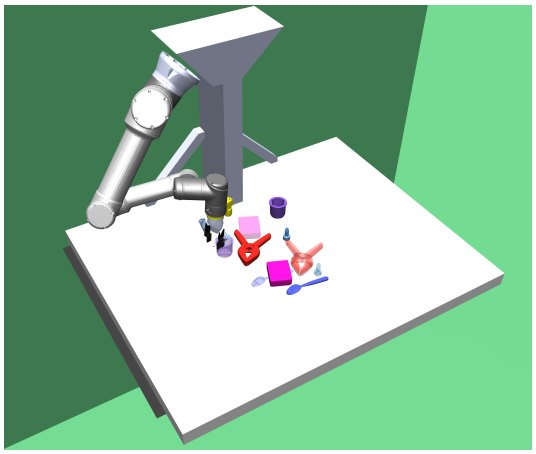


Figure 9: The original Robogym rearrange setup

C Environment Details

Block-rearrange and *Block-stack* are implemented in PyBullet (Coumans & Bai, 2016) while *Robogym-rearrange* is implemented in Mujoco (Todorov et al., 2012).

Environments The *robogym-rearrange* environment (see figures 8 and 9) is adapted from the rearrange environment in OpenAI's Robogym simulation framework (OpenAI, 2020) and removes the assumption from *block-rearrange* that all objects are the same size, shape, and orientation and the assumption of predefined locations. Furthermore, due to 3D perspective, the objects can look slightly different in different locations. The objects are uniformly sampled from a set of 94 meshes consisting of the YCB object set Calli et al. (2015) and a set of basic geometric shapes, with colors sampled from a set of 13. The camera angle is a bird's eye view over the table, and the size of each object is normalized by its longest dimension, so tall thin objects appear smaller. The objects' target positions are randomly sampled such that they don't overlap with each other or any of the initial positions, and the target orientation is set to be unchanged. Due to the continuous nature of this environment, we define a match threshold of at most 0.05 for both the initial pick position and the goal placement (the table dimensions are 0.6 by 0.8).

The *block-stack* environment adds preconditions and postconditions on whether objects can be moved: blocks can only be picked if they are at the top of a stack, and blocks can only be placed at a given height if there is already an object beneath to support it (otherwise it falls).

Sensorimotor interface Each observation is a tuple of an initial image displaying the current observation and a goal image displaying constraints to be satisfied – the goal locations of the objects. Each action is a tuple $(w, \Delta w)$, where w is a three-dimensional Cartesian coordinate (x, y, z) in the environment arena.

For the $*$-*rearrange* tasks, objects are initialized at random non-overlapping locations that also do not overlap with their goal locations. For these tasks the z (height) coordinate is always fixed. For the *block-stack* task, the goal locations are generated by randomly picking objects from the tops of stacks and placing them on other stacks. For this task the y (depth) coordinate is always fixed.

An object is picked if w is within a certain threshold of its location. For *block-*$*$ where object locations are fixed points in a grid the object is snapped to the nearest grid location to $w + \Delta w$. Constraints are considered satisfied if objects are placed within a certain threshold of their target location.

D Additional Results

This section presents additional results and analyses of NCS. We first analyze the sensitivity of task performance to several hyperparameters used in NCS from creating the graph: the number of clusters, number of buffer size, and the number of slots used in slot attention. We next tested whether

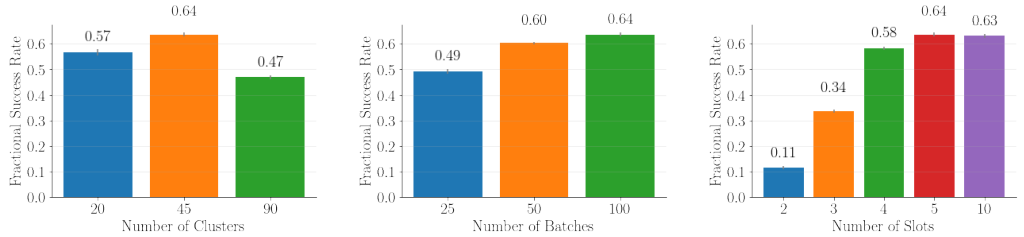


Figure 10: The performance of our method as the number of initialized clusters and batches from the training set used to construct the graph, and the number of slots are varied.

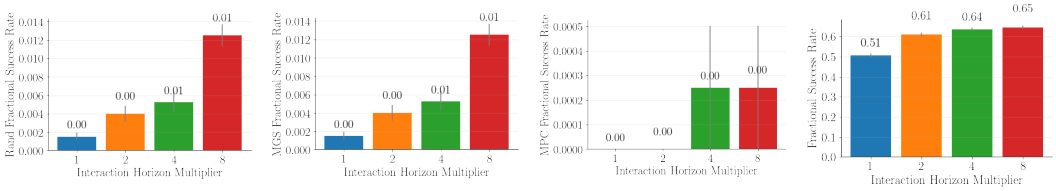
giving our model-based baselines more computation time compared to NCS would improve their performance to be comparable to NCS's.

D.1 Analysis of key hyperparameters

In this section, we analyze the sensitivity of our method to various hyperparameters, evaluated in the robogym environment with four objects in the complete goal specification. As Fig. 10 shows, performance depends on the number of initialized clusters and the number of batches from the training set used to construct the graph. With too few clusters, the clusters are too coarse-grained to differentiate objects in significantly different positions, while with too many the performance deteriorates as the data is needlessly split into duplicate clusters. Performance improves with more data, as the graph has better coverage. Although our model performs worse when there are insufficient slots to represent all objects present in the environment plus one empty slot, performance is barely impacted by having double the number of necessary slots. Our method can thus still work in environments with an unknown but upper-bounded number of objects.

D.2 More computation time for model-based baselines

We tested whether doubling the computation time for the model-based baselines would improve their performance to be comparable to NCS's. For the results in the main paper, we capped the length of the episode as 4x the minimum number of actions required to solve the task. In Fig. 11, we vary this interaction horizon multiplier from 1x to 8x. NCS degrades less with shorter interaction horizons compared to the baselines. We find that MGS performs similar to the random baseline. Since MGS takes a random action if it cannot bind the given entity set to its graph, this result suggests that the space of subsets of entities is so combinatorially large that MGS does not successfully bind to the graph most of the time. We verified that this is the case by inspecting when MGS takes random actions. MPC performs the worst out of all the methods, performing worse than random. We tested that the cost function described in B ranks latents that match the goal constraint with lower cost than randomly sampled latents, which suggests that the main source of error is due to the inaccuracy in the prediction rollouts. This can be expected, as learned models suffer from compounding errors when rolled out (Janner et al., 2019) and prior methods that use MPC for object-centric methods only roll out for very short horizons (Veerapaneni et al., 2020).



(a) Rand (b) MGS (c) MPC (d) NCS

Figure 11: **Varying interaction horizon.** The performance of the MGS (b) and MPC (c) baselines compared to NCS (d, reproduced from Fig. 7) and the random baseline (a) on *robogym-rearrange* as we vary the interaction horizon (as a multiple of the minimum steps needed to complete the task). *Note that the scale of the y-axis is not the same.* While a longer horizon improves performance, NCS still achieves at least 50x better accuracy with an interaction horizon multiplier of 1 than the performance obtained by increasing the interaction horizon multiplier for the model-based baselines to 8.