Robotics and Autonomous Systems 165 (2023) 104428

Contents lists available at ScienceDirect

Robotics and Autonomous Systems

journal homepage: www.elsevier.com/locate/robot

Automatic extension of a symbolic mobile manipulation skill set

Julian Förster [a,*], Lionel Ott [a], Juan Nieto [b], Nicholas Lawrance [a,c], Roland Siegwart [a], Jen Jen Chung [a,d]

[a] ETH Zurich, Zurich, Switzerland
[b] Microsoft Mixed Reality and AI Labs, Zurich, Switzerland
[c] CSIRO, Pullenvale, Australia
[d] The University of Queensland, St Lucia, Australia

ARTICLE INFO

Article history:
Available online xxxx

Dataset link: https://github.com/ethz-asl/high_level_planning

Keywords:
Skill representation learning
Learning to plan
Task and motion planning

ABSTRACT

Symbolic planning can provide an intuitive interface for non-expert users to operate autonomous robots by abstracting away much of the low-level programming. However, symbolic planners assume that the initially provided abstract domain and problem descriptions are closed and complete. This means that they are fundamentally unable to adapt to changes in the environment or tasks that are not captured by the initial description. We propose a method that allows an agent to automatically extend the abstract description of its skill set upon encountering such a situation. We introduce strategies for generalizing from previous experience, completing sequences of key actions and discovering preconditions to ensure computational efficiency. The resulting system is evaluated on a symbolic planning benchmark task and on object rearrangement tasks in simulation. Compared to a Monte Carlo Tree Search baseline, our strategies for efficient search have on average a 25% higher success rate at a 67% faster runtime. Code is available at https://github.com/ethz-asl/high_level_planning.

1. Introduction

Today, mobile manipulators are being developed for work in unstructured *human-centered* environments, i.e. spaces that are not specifically designed for deploying robots. To reach a level of ubiquity similar to their stationary counterparts in structured (e.g. manufacturing) domains, we need to shift to more flexible robot planning and execution that can adapt to changing environments and tasks. The goal is to deploy mobile manipulators in environments where it is impossible to foresee at design time all possible variations of tasks, disturbances, types and instances of objects, etc. In addition, a user would ideally only need to interact with the system at a mission level [1], specifying high-level goals (e.g. tidy the table) for which the robot would autonomously generate a plan that it then executes.

Symbolic planning provides a framework that is conducive to this type of user interaction [2]. Here the goal is to develop domain-independent planners that find a sequence and parameterization of skills that achieve the goal state. By operating on an abstract level, planning for long-horizon tasks becomes tractable.

While symbolic planning can be applied very naturally to robotics problems [3], it still relies on a complete definition of the problem at planning time. Additionally it typically relies on the closed world assumption, i.e. that all relevant states are observable and observed by the system. Upon encountering new tasks or new situations that cannot be correctly captured by the existing symbolic abstraction, planning will fail. However, in realistic scenarios, the open world assumption is more realistic, acknowledging that it is not possible for a single agent to have complete knowledge of all relevant states. To deal with this in practice in existing approaches, the symbolic abstraction would have to be updated manually to solve a new task.

This paper proposes an approach that overcomes the limitations imposed by a closed and complete world assumption. Upon encountering a failure during planning or plan execution due to insufficient symbolic operators, our method explores promising extensions to the currently defined set of symbolic skills to reach the goal. This is based on the assumption that the existing skills available to an agent are sufficient to solve the task at hand, but the symbolic model used for planning is not expressive enough to account for this. In this sense, our method can be seen as "learning to plan". For example, in a mobile manipulation domain, there is a very large number of outcomes that can be achieved with a "place" skill, depending on context, objects being manipulated, scene, etc. Instead of adding new robot skills for each desired outcome, we opt to extend the planning model by modifying what existing robot skills can be used for.

Since building a complete symbolic description (for arbitrary initial states) for an open world is unrealistic, we opt for a lifelong

* Corresponding author.
E-mail addresses: fjulian@ethz.ch (J. Förster), lioott@ethz.ch (L. Ott), juannieto@microsoft.com (J. Nieto), nicholas.lawrance@csiro.au (N. Lawrance), rsiegwart@ethz.ch (R. Siegwart), jenjen.chung@uq.edu.au (J.J. Chung).

https://doi.org/10.1016/j.robot.2023.104428

learning mode of operation. Instead of trying to build a complete description once at design time, we envision an incremental process of learning to solve new tasks as they occur, possibly also through interaction with a user. The exploration process needs to be as efficient as possible to make this practical. Therefore, our algorithm features:

- Generalization of existing skills to other object types and tasks;
- Sequence completion based on key actions to reduce the search space;
- Precondition discovery to keep the symbolic description modular and compact; and
- The ability to optionally take user demonstrations into account.

In summary, the contribution of this paper is a method to automatically extend the symbolic description of an agent operating under an open world assumption. We propose a novel exploration scheme which leverages the above-mentioned algorithm features to achieve flexible and accurate extensions to the symbolic description while retaining computational efficiency.

To demonstrate our approach, we evaluate it both in a PDDL benchmark domain and in a mobile manipulation domain involving continuous parameters, thus requiring task and motion planning (TAMP). The latter domain serves to showcase that, despite being general, our method can be applied to practical object rearrangement situations, including when obstacles need to be overcome or moved to achieve a goal. In both domains, we compare the exploration and planning efficiency of our approach against Monte Carlo Tree Search (MCTS). Our results show that the proposed algorithm leads to better performance in planning for unseen tasks, both in terms of success rate and task completion time.

2. Related work

Symbolic planning lends itself very well to high-level planning in robotics [3]. In various applications [4,5] impressive behaviors were achieved, using it to decide what action to take next. Typically however, the symbolic domain and problem descriptions are manually engineered, making it necessary to adapt them in case new tasks arise or if the system is to be deployed in a new environment.

TAMP approaches were developed for tasks that require planning on a geometric level [6–8]. These approaches can deal with the interplay between discrete and continuous state spaces. However, similar to purely symbolic planning, TAMP methods typically rely on hand-engineered symbolic operators [9], requiring the same manual adaptation when meeting new tasks.

Apart from using symbolic operators, other approaches exist that introduce an action hierarchy to guide planning, the *options* framework [10] being a prominent example. Another algorithm for discovering new options given a goal state classifier is proposed in [11]. While this idea is similar to our approach, we opted to use PDDL as a representation for skills due to its human-readability which simplifies retracing decisions. While the goal of hierarchical planning (also known as macro operator generation) [12,13] is to combine primitive skills into meta-skills for improved planning efficiency, it does not accommodate adaptation to new goals or unmodeled environment dynamics, which we investigate in this work.

The complexity of interacting with real-world environments has motivated decades of research on leveraging learning to enable planning or to improve its efficiency. The problem of planning in changing environments can be seen as a *MacGyver Problem* [14], alluding to the creativity of the namesake TV show character. To predict actions an agent should take to achieve a goal, existing methods are trained using human demonstrations [15], behavior cloning from expert-defined policies [16,17], or curiosity-based exploration of the state space [18]. Additionally, reinforcement learning (RL) can be applied to finding action sequences in a goal-driven manner.

In comparison however, planning-based approaches leveraging symbolic models have a number of qualitative advantages. Operating based on PDDL makes the resulting action models human readable, rendering resulting plans explainable. Further, having PDDL as a common language allows for elegant ways to provide user demonstrations and generalize from previous experience as we show in this work. The resulting descriptions are, thanks to the PDDL standard, compatible with existing TAMP systems, allowing us to build on the latest advances from that field. Finally, RL or other model-free approaches can have difficulties generalizing beyond the distribution of the training data. This can become a problem considering the significantly longer training times of the RL baselines compared to the symbolic methods reported in [19].

Apart from these qualitative differences, related work has shown that the success rate of RL methods is inferior compared to explicit planning methods [18–20]. Therefore, we do not consider RL a suitable approach to the on-demand lifelong learning style operation of our method that we describe in Section 1 and Section 6. In addition to this, leveraging abstract symbolic operators as a model to guide the search for a plan was shown to increase search efficiency for TAMP [6,19].

Learning symbolic operators to support planning is an active research topic [20,21]. However, studying this in domains that involve a mix of discrete and continuous state spaces has received little attention. In this vein, multiple methods [22,23] exist for building a symbolic description based on exploration and high-dimensional sensor data. While simultaneously learning state abstractions allows automatically building a symbolic description, it also necessitates retraining for every new task instance, limiting generalizability. Despite efforts to tackle this [24], the need for retraining cannot be completely eliminated. For this reason and until future work finds a practical solution, most TAMP approaches, including our method, rely on pre-defined state abstractions that generalize over environments, scenes and objects.

Diehl et al. [25] present a system to build symbolic planning operators from human demonstrations. While demonstrations are a very promising way to expedite learning to plan, we believe that automatic exploration as proposed in this paper is crucial to minimize the required amount of costly human interaction. Nevertheless, our method allows optionally leveraging human demonstrations if they are available.

The work of Silver et al. [19] is the closest to ours. They propose learning symbolic operators in a bottom-up fashion based on transition data that is collected using an oracle planner. While we also aim to learn symbolic operators, we adopt a goal-driven approach that allows us to narrow the search space and avoid relying on an oracle planner, thus reducing the engineering effort needed before training can occur.

In summary, we propose an efficient approach to learning symbolic action models in a goal-driven exploration scheme. The resulting models can serve as drop-in replacements for their hand-engineered counterparts that are typically used today for symbolic planning or TAMP.

3. Definitions and problem formulation

The aim of symbolic planning is to produce a sequence of transitions leading from the initial state to a goal state. The Planning

Domain Definition Language (PDDL) [26] was created to simplify the modeling of planning problems and to facilitate comparing symbolic planners. It splits the problem into a problem-invariant *domain description* and a problem-specific *problem description*, the former consisting of:

- The set of **object types** T organized in a hierarchy.
- The set of **predicates** P that are available to model a state. Each predicate can accept parameters of predefined types and evaluates to a binary value.
- The set of **actions** A (also referred to as skills) the agent can perform, each accepting parameters. An action's preconditions state which predicates need to hold before the action can be executed and its effects model what predicates change after executing the action.

The problem description contains:

- A list of all **entities** E that are relevant for the planning problem, together with a specification of their types.
- The **initial state** s_0, defining the initial values of all predicates.
- The **goal** S_g, defined as the set of states that meet the specifications of a target subset of all predicates.

With these two descriptions, which can be summarized as $d = (T, P, A, E, s_0, S_g)$, off-the-shelf symbolic planners can produce a plan that solves the task specified by the problem description. However, this assumes that the domain description is rich enough for a solution to exist. We consider the problem where this assumption does not hold and thus there is a need to automatically augment an existing domain description that is not rich enough to model the laws governing an environment, or the capabilities of the agent. This can be encountered when:

(C1) The preconditions of an action performed by the agent are not fully modeled (e.g. because of a new disturbance or the agent being deployed in a new environment). This case can be detected as a failure during plan execution.

(C2) The agent is tasked with achieving a newly introduced predicate (e.g. through specification of a goal state by a user), for which it is not modeled how to achieve it using the available actions. Characteristic of this case is that the planner fails to output a valid plan given d.

In the following, we split the set of actions into two subsets, A_b and A_m with $A = A_b \cup A_m$. Here, A_b is a set of basic actions $a_b \in A_b$ available to the agent initially, where each action,

$$a_b = \left(\theta_{a_b}, \text{pre}(\theta_{a_b}), \text{eff}(\theta_{a_b}), \phi_{a_b}\right), \tag{1}$$

takes parameters θ_{a_b}, and has a low-level implementation ϕ_{a_b} that grounds it. The term pre$(\cdot)$ denotes action a_b's preconditions, a list of binary predicate states (in Eq. (1) parameterized by a_b's parameters θ_{a_b}) that need to hold before a_b can be executed. In the same fashion, eff$(\cdot)$ represents a_b's effects, a list of predicate states that take effect after executing a_b. As an example, consider a grasp action. Its list of preconditions could be defined as [(`in-reach target-object robot`), (`empty-hand robot`)], where `in-reach` and `empty-hand` are the predicate labels, and the other tokens in the round brackets represent parameter assignments.

Meta actions $a_m \in A_m$ are defined similarly as,

$$a_m = \left(\theta_{a_m}, \text{pre}(\theta_{a_m}), \text{eff}(\theta_{a_m}), \rho_{a_m}\right). \tag{2}$$

Instead of using a grounding ϕ, these actions are defined using a sequence of basic actions $\rho_{a_m} = (a_{b,1}, a_{b,2}, \ldots)$ where $a_{b,i} \in A_b$. When choosing a meta action a_m, its sequence is executed under the hood. Introducing meta actions allows the inclusion of additional preconditions and effects that are not captured by the underlying basic actions, which we aim to keep fixed.

To solve cases (C1) and (C2), we aim to adapt applicable types, preconditions and effects of existing meta actions in A_m, leading to A'_m, as well as add new meta action(s) $a^+_m \in A^+_m$ as necessary. The planner should be able to use $d' = (T, P, A_b \cup A'_m \cup A^+_m, E, s_0, S_g)$ to devise a plan that leads to successfully achieving S_g.

As mentioned in Section 2, assuming access to predicate groundings is common for robotic applications of symbolic planning and TAMP. Thus, in this work, we make a similar assumption, i.e. that grounding classifiers provide measurements of instantiations of the predicates in P, which are then used to adapt the symbolic description.

4. Exploration for skill set extension

4.1. Overview

To get a better intuition of the problem and an overview of our proposed method, consider the following example. A mobile manipulation system (MMS) is equipped with a basic skill set $A_b = \{$navigate, grasp, place, move$\}$ and a generic symbolic description modeling each skill's preconditions and effects as actions. Confronted with a new user-provided goal to place a cup `inside` a drawer (where `inside` is a predicate that no symbolic operator has in its effect set), a symbolic planner using the initial symbolic description fails to output a valid plan, since it is unknown how to reach the effect `inside` (see case (C2) in Section 3). From here, our proposed exploration algorithm (visualized in Fig. 2) takes over. The *exploration* module (Section 4.2) forms the core, sampling and executing sequences of basic actions and testing if the goal is achieved. To keep the sampling effective despite the large search space that is caused by long sequences and continuous parameters, we rely on *sequence completion* (Section 4.3). To leverage guidance from a user, there is the option to include simple *demonstrations* (Section 4.5). Finally, when a successful sequence that achieves the goal is found, we use a *precondition discovery* procedure (Section 4.6) to determine which actions are required under different conditions, thus only executing the necessary actions. The newly identified meta actions constituting the successful sequence are added to the symbolic description so that the symbolic planner can devise successful plans for this situation in the future.

Continuing our example, assume that after solving the initial problem of placing a cup `inside` a drawer, the system is given a similar goal, e.g. to place a plate `inside` a different drawer. Although we cannot assume that the same plan we found before will work (e.g. due to different dimensions of the involved objects), we would like to benefit from previous experience. This knowledge transfer is achieved by the *generalization* module (Section 4.7).

In the following sections, we first introduce the core exploration algorithm (Section 4.2), before detailing the extensions mentioned above, and how they fit into the core algorithm.

4.2. Exploration

Exploration (Algorithm 1) begins once (C1) or (C2) (see Section 3) occurs. At its core, the algorithm samples sequences and their parameterizations which are then executed and tested for success. The inputs are: a set of admissible goal states S_g, a set of objects O to consider during the exploration (either all objects in the scene or a subset to reduce the search space), a time budget T_{max}, and a maximum sequence length l_{max} that limits the search depth.

Algorithm 1: Exploration

Input: Set of goal states S_g, relevant objects O, initial state s_0, time budget $T_{\max}$, max. seq. len. $l_{\max}$

1 **while** `ElapsedTime()` $< T_{max}$ **do**
2 $\quad \tilde{S} \leftarrow$ `SampleSequence`(l_{max});
3 $\quad \tilde{P} \leftarrow$ `SampleParameters`$(\tilde{S}, O)$;
4 $\quad \hat{S}, \hat{P}, I_{\text{key}} \leftarrow$ `SequenceCompletion`$(\tilde{S}, \tilde{P}, s_0)$; `// see Sec. 4.3`
5 $\quad$ success $\leftarrow$ `Execute`$(\hat{S}, \hat{P}, S_g)$;
6 $\quad$ **if** success **then** break;
7 **if** success **then**
8 $\quad S, P \leftarrow$ `SequenceRefinement`$(\hat{S}, \hat{P}, S_g, I_{key})$;
9 $\quad C \leftarrow$ `PreconditionDiscovery`(S, P, O, I_{key}); `// see Sec. 4.6`
10 $\quad$ `ExtendSymbolicDescription`(S_g,S,P,C);

We assume that the set of relevant objects O is provided by a perception module ([27] for example, or any other system with equivalent capabilities). As long as the perception module can detect new and previously unseen objects, our system can readily include them in its exploration procedure without further modifications. For this work, we assume such a perception module to be given in order to focus on planning aspects. This is a common assumption, also among related work [19,25].

To obtain a sequence candidate $\tilde{S} = \{\tilde{a}_1, \ldots, \tilde{a}_{\tilde{n}}\}$, $\tilde{n}$ basic actions are independently and uniformly sampled from A_b. This procedure is labeled as `SampleSequence` in Algorithm 1. Although the final successful sequence length depends on the problem at hand, we found that the best compromise between general applicability and runtime is to only sample sequences of length $l_{\max}$. In case only part of a sampled sequence is needed to achieve a goal, we have a `SequenceRefinement` strategy to shorten the sequence to the relevant part. This will be discussed below. For a discussion on alternative sampling strategies, refer to Appendix.

Parameters (i.e. placement positions, objects, etc.) for the actions in the sampled sequences are determined with `SampleParameters` according to the following strategy. Discrete parameters are obtained by sampling uniformly from the subset of entities $O_{i,j} \subseteq O$ that satisfy the type restriction of action $\tilde{a}_i$'s jth parameter. For continuous parameters, a sampling strategy needs to be defined. For placement positions considered in this work for example, we sample uniformly in a volume around the objects in O. This strategy is purposefully independent of the predicate that needs to be fulfilled to ensure a broad range of predicates can be used with a single simple strategy, avoiding the need for predicate-specific samplers which are common in TAMP systems.

Once sequence and parameterization are determined, executing the candidate plan in a physics simulator using `Execute` checks whether the system successfully terminates in S_g, thus satisfying the goal predicates.

When a successful sequence is found, the exploration loop stops and the symbolic domain description is extended. In preparation for this, we run `SequenceRefinement` to reduce the sequence to the minimum length that is still successful in achieving the goal by removing actions iteratively and testing whether the resulting sequence still achieves the goal successfully. This will be discussed in detail in Section 4.4.

For the extension of the symbolic description, implemented in `ExtendSymbolicDescription`, our algorithm determines parameters, preconditions and effects of the refined sequence, which is then combined into a single new meta action a^*_m. Fig. 3 shows an example for this procedure. Effects are determined symbolically during a forward pass through the sequence by collecting the effects of each individual action and removing effects from the list that are undone by actions later in the sequence:

$$\text{eff}(a^*_m) = \bigcup_{i=1}^{n} \text{eff}(a_i) \tag{3}$$

where n is the length of the successful sequence and a_i is the ith action in the sequence. In addition, the user-defined goal is added to the list. Preconditions are extracted similarly, by collecting the preconditions of all actions during a backward pass and removing preconditions that are satisfied by actions earlier in the sequence:

$$\text{pre}(a^*_m) = \bigcup_{i=n}^{1} \text{pre}(a_i). \tag{4}$$

The parameters of all actions in the sequence are collected, accounting for cases where several actions act upon the same entity.

Furthermore, we want to avoid that our symbolic description allows applying meta actions to entities that are incompatible in reality. To this end, new symbolic types are introduced for all parameter variables of the newly created meta actions, branching off the original types of the entities assigned to the parameter variables. These entities are then assigned the new types in addition to their existing types. This prevents incorrect inference results that could be caused by having a meta action being applicable to new entities it was not tested with.

Returning to our running example, this means that after exploration, cup and drawer are assigned new sub-types, see Fig. 4. Assume that the scene also contains a second cup, as well as a bottle that does not fit into the drawer. When tasked with putting either object into the drawer, symbolic planning would fail because the new action is only applicable to entities with the new sub-types. Without the sub-type limitation, the system would assume the bottle could be placed into the drawer in the same way. Instead, due to the symbolic planning failure, our algorithm would be called to determine whether the goal can be achieved with the new object. If yes, the correct sub-type can be added to the new object. We deem this procedure necessary, since in general it is not possible to automatically define conclusive rules for determining whether an action in the symbolic description is applicable to a given object. To ensure we can manipulate all objects in the environment, we rely on our algorithm being called for every unseen object. If done naively, this would perform exploration from scratch every time, which would be computationally prohibitive. However, we are able to overcome this challenge by using a generalization module which will be discussed in Section 4.7.

4.3. Sequence completion

Output sequences for mobile manipulation tasks can be fairly long. However, we note that often just a small subset of *key actions* are crucial for the success of the task, while the other actions merely fulfill preconditions of the key actions. Based on this insight, we can drastically reduce our exploration effort by sampling shorter sequences on line 3 of Algorithm 1 and leveraging the symbolic planner to solve for the corresponding complete and feasible sequence in our `SequenceCompletion` routine.

Using our running running example, a successful sequence that achieves the goal state "cup inside drawer" would consist of the following actions:

Algorithm 2: Sequence completion

Input : Sequence $\tilde{S}$, parameters $\tilde{P}$, initial state s_0
Output: Completed sequence $\hat{S}$, parameters $\hat{P}$, indices of key actions I_{key}

Function `SequenceCompletion`$(\tilde{S}, \tilde{P}, s_0)$
  $\hat{S}, \hat{P}, I_{\text{key}} \leftarrow [\,], [\,], [\,]$;
  $X \leftarrow s_0$; `/* track current state */`
  foreach $\tilde{a}_i, \tilde{\theta}_i \in \tilde{S}, \tilde{P}$ **do**
    $p \leftarrow$ `GetPreconditions`$(\tilde{a}_i)$;
    $\check{S}_i, \check{P}_i \leftarrow$ `SolvePDDL`$(initial = X, goal = p)$;
    $X \leftarrow$ `ApplyEffects`$(X, (\check{S}_i, \tilde{a}_i), (\check{P}_i, \tilde{\theta}_i))$;
    $\hat{S} \leftarrow \hat{S} + \check{S}_i + \tilde{a}_i$; $\hat{P} \leftarrow \hat{P} + \check{P}_i + \theta$;
    I_{key}.append $\left(\text{length}(\hat{S}) - 1\right)$;
  return $\hat{S}, \hat{P}, I_{\text{key}}$;

```
["navigate to drawer","grasp drawer",
 "move drawer","place drawer",
 "navigate to cup","grasp cup",
 "navigate to drawer","place cup"].
```

With sequence completion, we can infer this sequence from the considerably shorter sequence:

```
["move drawer","place cup"].
```

Consequently, the task of finding a sequence that achieves a certain goal is reduced to finding the key actions from which such a sequence can be constructed using the symbolic planner, thus greatly reducing the search space. Note that our algorithm does not know a priori whether sampled actions are key actions. Only completing and testing a sequence for success reveals whether the sampled actions are key actions.

Our procedure of sequence completion is laid out in Algorithm 2. For each key action $\tilde{a}_i$ in the sampled sequence, we solve a symbolic planning problem (using `SolvePDDL`) that has the preconditions of the current key action as desired goal, resulting in a fill sequence $\check{S}_i = ({}_i\check{a}_1, \ldots, {}_i\check{a}_{\check{n}_i})$ and corresponding parameterizations $\check{P}_i$. Both are appended to the completed sequence $\hat{S}$ and parameters $\hat{P}$. Furthermore, before the next iteration, effects of both the fill sequence and the currently considered key action are symbolically applied to state X (using `ApplyEffects`), which was initialized with s_0. After iterating over all key actions, the completed sequence returned by `SequenceCompletion` has the form,

$$\hat{S} = \Big(\underbrace{{}_1\check{a}_1, \ldots, {}_1\check{a}_{\check{n}_1}}_{\check{S}_1}, \tilde{a}_1, \underbrace{{}_2\check{a}_1, \ldots, {}_2\check{a}_{\check{n}_2}}_{\check{S}_2}, \tilde{a}_2, \ldots, {}_{\tilde{n}}\check{a}_{\check{n}_{\tilde{n}}}, \tilde{a}_{\tilde{n}}\Big). \tag{5}$$

4.4. Sequence refinement

As mentioned in Section 4.2, our `SequenceRefinement` procedure shortens a successful sequence found through exploration to the minimum length required to still be successful. This is important to avoid the need for assumptions on the successful sequence lengths. Instead, we sample longer sequences of key actions that can contain solution sequences, and rely on sequence refinement to shorten after a successful sequence was determined.

As a first step of the procedure, we truncate the sequence after the action that achieves the desired goal, since by consequence any additional actions are superfluous for our problem. Next, we iterate through key actions from second-to-last to first. On its turn, a key action is replaced by other actions between its neighboring key actions, as well as removed. The resulting key action sequences are completed using sequence completion and then tested for success. Among successful sequences, the shortest one with the least remaining key actions is selected to continue from. In Fig. 5, the `SequenceRefinement` procedure is visualized using an example.

4.5. User demonstrations

The concept of sequence completion allows for another elegant way to improve efficiency. Since humans are very good at planning for manipulation tasks, it seems natural to leverage a user's knowledge. To achieve this, a user can optionally supply our system with one or several key actions that are critical to achieve a goal. In addition, crucial parts of the parameterization can be given. The key action sequence shown in Section 4.3 is an example for a potential user demonstration, `move` and `place` being the demonstrated key actions while `drawer` and `cup` are the corresponding parameters. Note that this demonstration does not provide values for all parameters of the key actions. It is up to the user to decide which parameter(s) they want to provide a demonstration for. All other parameters will be filled by our method's exploration procedure. For more examples of demonstrations, see Section 5.3.

The exploration procedure with sequence completion can then be used to fill in any missing parameters of the key actions as well as any actions that are missing before or in between the key actions. All in all, this feature provides an interesting middle ground, making it easy for the user to provide a demonstration without the need to specify all details of a sequence and at the same time reducing the search space during exploration.

4.6. Precondition discovery

In practice it can happen that not all steps of a discovered sequence are needed every time a similar goal needs to be reached. For example, if during exploration, an obstacle was present (drawer closed in our running example) and the agent correctly learned that the obstacle needs to be removed (drawer needs to be opened) before the goal can be achieved (object placed inside), the actions to remove the obstacle (opening the drawer) will only be needed in the future if the obstacle is present (drawer is closed) in the individual situation.

The symbolic description should correctly capture what parts of a sequence actually achieve the goal and what parts solely fulfill preconditions for the goal-achieving actions. We tackle this `PreconditionDiscovery` (Algorithm 3) by simulating a discovered successful sequence and observing any predicate changes that are not modeled as action effects (*side effects*).

The function `FindRelevantPredicates` determines the predicates ρ to be considered, by finding all possible instantiations of the predicates P known to the system with all possible combinations of objects O. For example, assume that a predicate "on" is defined in the given symbolic description, and there are multiple potential supporting and supported objects in the domain. For every combination of supporting and supported object, an instantiation of the "on" predicate would be added to ρ. The grounding functions of the instantiated predicates ρ can be evaluated using `MeasurePredicates`. In turn, `DetectChanges` returns the difference between two lists of measured predicates, ignoring all changes that are expected according to the symbolic description of the current action.

As shown in Algorithm 3, the successful sequence discovered by the exploration algorithm (Section 4.2) is executed action

Algorithm 3: Precondition discovery

Input : Sequence S, param. P, relevant objects O
Output: Precondition candidates C

Function `PreconditionDiscovery`(S, P, O)
  $C \leftarrow [\,]$; // precondition candidates
  $\rho \leftarrow$ `FindRelevantPredicates`(O);
  $p_{\text{post}} \leftarrow$ `MeasurePredicates`(ρ);
  for $a, \theta \in S, P$ **do**
    $p_{\text{pre}} \leftarrow p_{\text{post}}$;
    `Execute`(a, θ);
    $p_{\text{post}} \leftarrow$ `MeasurePredicates`(ρ);
    $p_{\text{new}} \leftarrow$ `DetectChanges`(p_{pre}, p_{post}, a, θ);
    $C \leftarrow C + p_{\text{new}}$;
  $C \leftarrow$ `FilterToggling`(C);
  return C;

by action, interleaved with discovering side effects using the functions introduced above. Each detected change is considered a candidate for a precondition of the final key action that achieves the goal.

However, to avoid adding superfluous preconditions and thus fragmenting the discovered sequence more than necessary, we filter the candidates (with `FilterToggling`), removing candidates that get toggled throughout the sequence execution, i.e. set and later unset or vice versa, because any toggled predicates look unchanged to the last action which achieves the user-defined goal. Thus, toggled predicates cannot be preconditions for that action.

Finally, if precondition discovery is used, the operating mode of the function `ExtendSymbolicDescription` (used in Algorithm 1 and described in Section 4.2) changes. Instead of creating a single new meta action for the whole sequence, precondition candidates are consolidated per key action, and new meta actions are created for each key action, adding the discovered precondition candidates as effects. The final meta action, corresponding to the last key action that achieves the goal gets the precondition candidates as preconditions. This has the effect that the symbolic planner will only schedule actions from this sequence that are necessary because the corresponding precondition of the final goal-fulfilling actions is not yet set. An example is given in Section 5.5.

4.7. Generalization and reuse of previous experience

It is common in mobile manipulation applications that over time, similar goals need to be achieved, but for different objects and circumstances. In such a case, we want our system to leverage previous experience in order to find a solution without exploration from scratch. Assume that in our running example, the agent already knows how to place the cup inside the drawer and now wants to place the plate inside the drawer. Ideally, the existing experience should be used when figuring out how to achieve the new goal.

In this work, we achieve this by first temporarily relaxing the type specifications of entities that are part of the goal specification, allowing them to generalize to all available types and thus fit any parameter of any action, see example in Fig. 6. If the symbolic planner succeeds in finding a plan under these conditions, the actions forming that plan may help to achieve the goal. After extracting the action that actually achieves the goal (called the *generalization candidate*), exploration continues as in Section 4.2. However, every time `SampleSequence` is called, the generalization candidate is taken as a given part of the sampled sequence. This has the purpose of finding auxiliary actions and parameterizations that, together with the extracted action, form a successful sequence.

Once this sequence is found, the symbolic description is adapted. If the generalization candidate turns out to be part of the goal-reaching sequence, the corresponding action description and the types of the goal entities are adjusted to be compatible with one another. Apart from making the exploration more efficient by generalizing previous experience, this procedure has the advantage that it contributes to making the action space in the symbolic description as large as necessary, but simultaneously keeps it as small as possible.

5. Experiments and results analysis

5.1. MCTS baseline

We compare our approach against MCTS. MCTS successively builds a tree, where nodes represent world states and edges represent actions. To ensure fair comparison, the baseline shares the action implementations and the simulation for determining action outcomes with our method. Our implementation of MCTS is inspired by [7]. At every iteration, a decision is made on whether to expand the current node (i.e. try a new action from the node's state) or to select a child of the current node and continue from there. The decision is based on a progressive widening law [28], which is a feature of the Upper Confidence Tree (UCT) algorithm, and operates as follows: The node is expanded if $\lfloor N^{\alpha} \rfloor > \lfloor (N-1)^{\alpha} \rfloor$, where $\lfloor . \rfloor$ is the floor operation, N is the number of times the current node was visited, and $\alpha \in [0, 1]$ is a parameter controlling the balance between expanding and selecting a child (the higher α, the more we expand). In this work, we use $\alpha = 0.6$ for a good balance between expanding and exploring existing child nodes. When expanding, we only select from actions that are feasible from the current state according to the actions' preconditions.

When selecting an existing child of the current node to continue from (instead of expanding the current node), we determine the child according to the following rule:

$$i_c = \underset{i \in 1,\dots,K}{\arg\max}\, \bar{S}_{i,m_i} + \gamma \sqrt{\frac{\log N}{m_i}}, \tag{6}$$

where i_c is the index of the selected child, K is the number of children of the current node, $\bar{S}_{i,m_i}$ is the average reward after selecting this child until now, γ is an exploration constant (we use $\gamma = \sqrt{2}$ in this work), and m_i is the number of times child i was selected.

A node is terminal if the corresponding action execution failed, a maximum search depth is reached, or the resulting state achieves the goal specification. Once a terminal node is reached, the algorithm resumes from the root node until a solution is found or the time budget is used up.

5.2. Domains and setup

To evaluate the proposed method and to compare it to the baseline, we ran experiments in two domains. For the **PDDL benchmark domain** we make use of the well-known *Rovers* domain description, which has been used in editions of the International Planning Competition [29] for decades. To simulate a situation in which the symbolic description does not capture everything the agent can do in the world, we initialize the agent with shortened descriptions from which we progressively remove preconditions and effects, simulating case (C2) as described in Section 3. The original descriptions (which the agent has no access to) are used to determine the true executability (depending

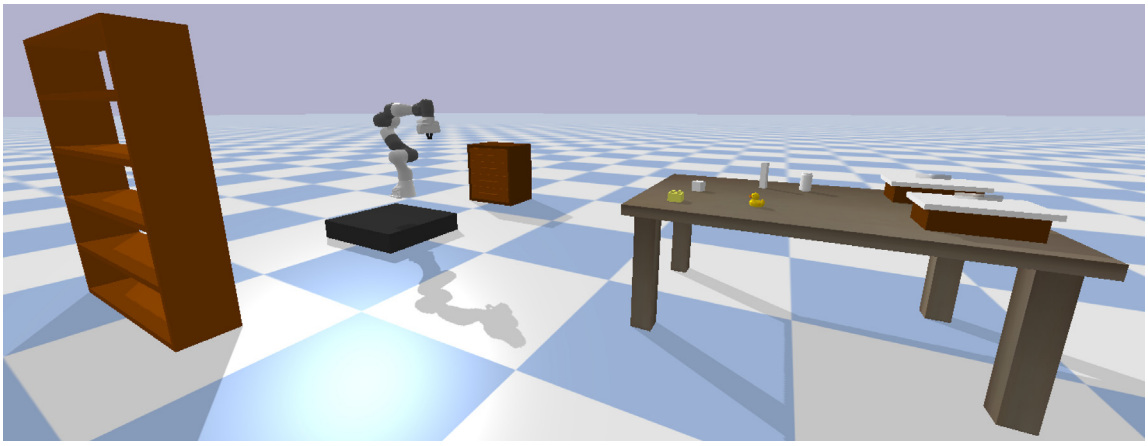


Fig. 1. Simulation environment setup that is used to demonstrate the proposed automatic skill set extension method. The unstructured environment is designed for object rearrangement tasks.

on preconditions) and effects of actions taken by the agent. In the *Rovers* domain, which is conceptually visualized in Fig. 7, the objective is to collect planetary data using a rover. Specifically we make use of the actions needed for collecting soil data, which include sampling soil, emptying the soil storage, navigating between adjacent nodes of the undirected graph which represents the terrain, and finally communicating the soil data back to the lander. The scenarios we considered in the rover domain are listed in Table 1. Experiments (r-a)–(r-d1) and (r-e) require the discovery of sequences of increasing length with no experience from previous scenarios. In contrast, experiments (r-d2) and (r-d3) are designed to demonstrate the generalization capabilities of our algorithm and so here the planner is initialized with the knowledge it gained by solving (r-d1).

The **rearrangement task domain** is used to evaluate our method in a realistic robotics use case. A key difference to the PDDL benchmark domain is the existence of continuous action parameters, making it a TAMP problem. Action outcomes are determined using a PyBullet-based physics simulation [30]. The simulation environment is shown in Fig. 1. Since this work focuses on high-level planning, we used simplified implementations of the robot skills. The navigation skill teleports the robot in simulation to the collision-free location closest to the desired goal location. For the grasping skill, grasp poses are pre-defined for all objects. More details on robot skills and predicates used for the experiments can be found in our open source implementation at https://github.com/ethz-asl/high_level_planning. The tasks evaluated in the rearrangement task domain are shown in Table 2. Scenarios (c2)–(c4) as well as (d2) test the benefit of our generalization step from prior knowledge.

We found that for most tasks in the rearrangement task domain, target objects and objects spatially close to them may be relevant to solving a task. Therefore, for all experiments both with our method and the baseline, the set of relevant objects O includes target objects and all objects within Euclidean distance d_{rel} from any target object. For this work, we set $d_{\text{rel}} = 10$ cm. In the PDDL benchmark domain, we do not limit the number of objects, so all objects from the domain are included in the exploration.

All experiments were conducted using an *Intel Core i7-9750H* laptop CPU. As symbolic planner, we use *Metric-FF* [31]. Our algorithm writes symbolic description files (domain and problem as introduced in Section 3), calls the planner on them and parses the planner's output for further processing. All code is implemented in Python.

5.3. Procedure

We conducted experiments with our algorithm (with and without demonstrations) and the MCTS baseline on the scenarios shown in Tables 1 and 2. Each algorithm was run 10 times for each of the PDDL benchmark domain scenarios and 50 times for each of the rearrangement task scenarios, each run with a time budget of 900 s. Before every run, the simulated scene was reset to its initial state, depicted in Fig. 1 for the rearrangement task domain. For our method, the symbolic description available to the agent in the beginning of a run was also reset. For the rearrangement task domain, it was reset to the four basic skills plus any initial knowledge resulting from a previous successful exploration if given. For the PDDL benchmark domain, each run was reset to the symbolic description as listed in Table 1. Upon hitting the time budget, a run was aborted and marked as failed. For both our method and the MCTS baseline, a maximum sequence length l_{max} needs to be specified to constrain search depth. With the intention to overestimate the truly needed length, we chose a length of 4 key actions for scenarios (a)–(c4) and (r-a)–(r-e) and 6 for scenarios (d1)–(d2) for our method. We observed that on average, there are 4 actions per key action in the expanded sequence, leading to selecting a maximum search tree depth of 16 and 24, respectively, for MCTS. For runs of our method with demonstrations, the demonstration was given in the form of key actions and parameters for each scenario as shown in Table 3. Note that only selected parameters are supplied as demonstration. Parameters not mentioned in Table 3 needed to be discovered by our method.

5.4. Results: PDDL benchmark domain

The results of the experiment scenarios in the PDDL benchmark domain are shown in Fig. 8. We report the distribution of

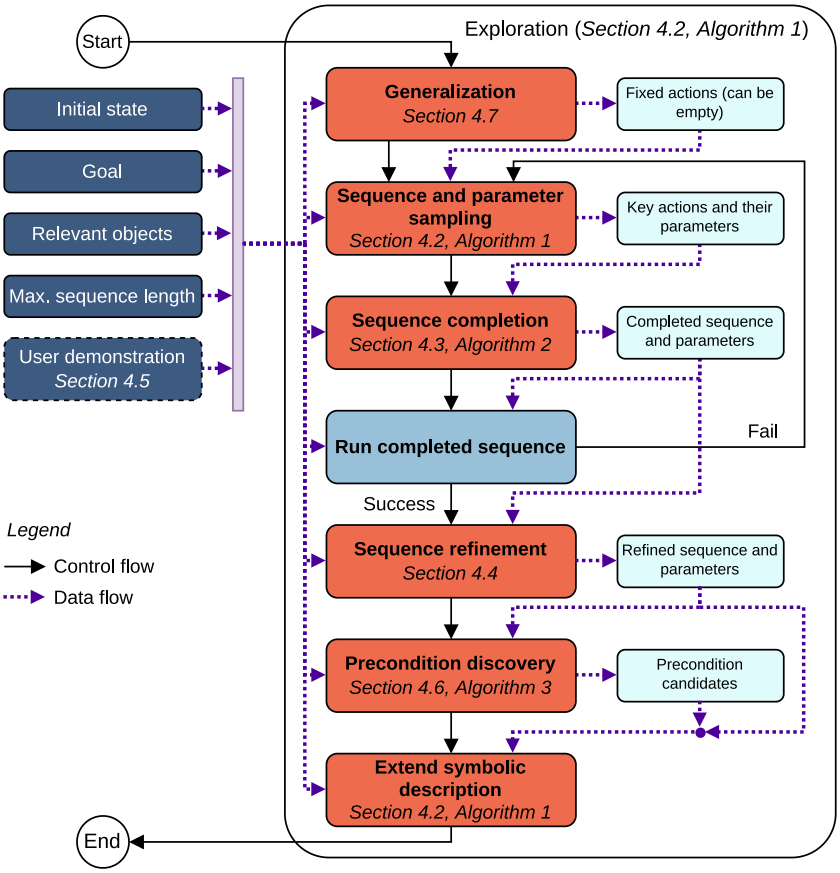


Fig. 2. Overview of the exploration for skill set extension with inputs, proposed algorithm components, a physics simulator and intermediate results. Note that the user demonstration is an optional input. For better readability, data flows from the inputs are consolidated. In reality, not all algorithm components require all inputs. For more details, refer to the descriptions in Section 4.

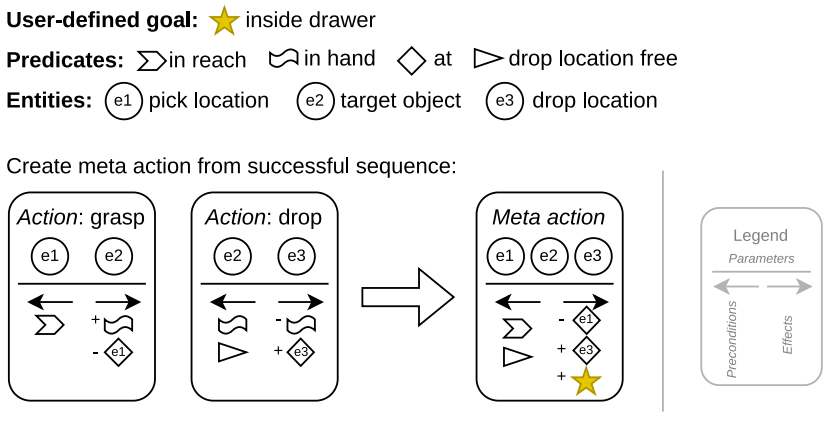


Fig. 3. Process of combining multiple actions of a sequence into a single meta action, visualized using our running example of grasping and placing an object. The resulting meta action contains preconditions of actions that are not fulfilled by actions earlier in the sequence, as well as effects of actions that do not get undone by actions later in the sequence.

time spent until a successful solution was found for successful runs as well as the number of unsuccessful runs.

For scenarios (r-a)–(r-c), which feature increasingly incomplete symbolic descriptions (i.e. longer key action sequences to be discovered), our method's performance is comparable to MCTS. The slightly higher runtimes of our method can be attributed to the overhead of calling the symbolic planner for sequence completion. From (r-d1) onwards, the larger search space that needs to be covered causes higher failure rates for MCTS (30% and higher). For our method, sequence completion allows sampling shorter sequences and completing them, enabling us to scale to longer sequences, and leading to no failures in this scenario. The significantly more difficult problem (r-e), which requires finding a sequence of length 15, can also be solved using our method with a success rate of 70%. In contrast, already in scenario (r-d2), which requires finding a sequence of length 9, MCTS never succeeds and this carries through to scenario (r-e).

In the scenarios discussed until now, our method rediscovered the preconditions and effects we removed. For each key action, a new meta action was created that is functionally equivalent to the respective unabridged basic action, with the missing precondition and/or effect information added.

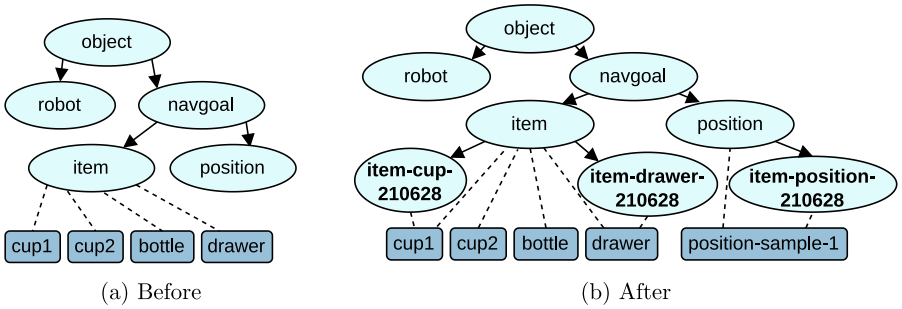


Fig. 4. Types and entities assigned to them, before and after extending the symbolic description adding new sub-types (**bold**) for "cup1", "drawer" and a newly introduced position sample. Entities not involved in the exploration ("cup2" and "bottle" in this example) can be assigned to the new types later if the new meta action is tested with them.

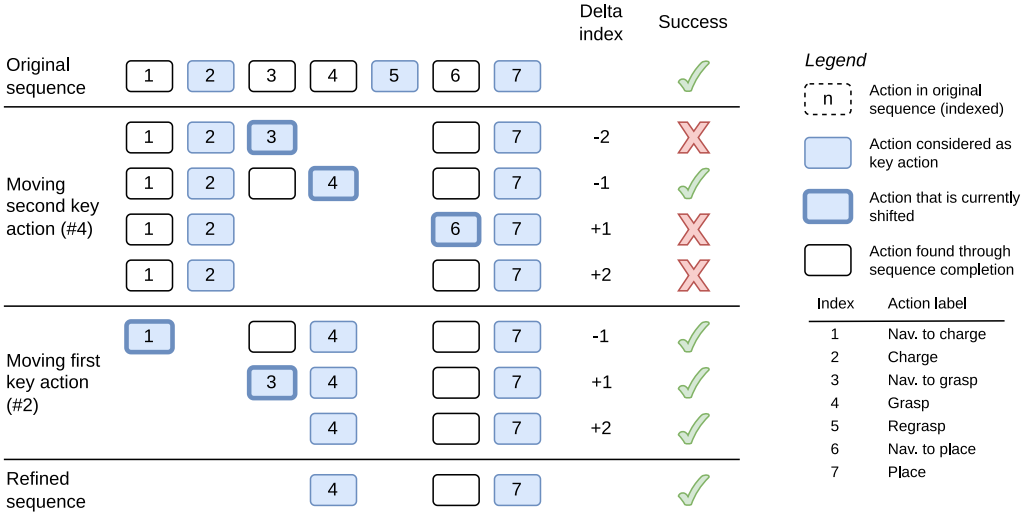


Fig. 5. Schematic example of the `SequenceRefinement` procedure that is carried out after a successful sequence is found. The procedure tests sequences with modified key actions. Finally, the shortest sequence with the least remaining key actions is returned. In this example, the goal is to place an object at a target location. The sequence found during exploration contains superfluous key actions (charging and re-grasping) which are removed using sequence refinement, bringing the number of key actions down to 2 and the sequence length down to 3.

Table 1
Experiment scenarios in the **PDDL benchmark domain** (Rovers). The goal for every scenario is to communicate soil data (c.s.d.) of a specified waypoint. "ID" stands for experiment ID, l is the minimum length of a successful sequence, l_k refers to the minimum number of key actions in a successful sequence, and "Prior" refers to the information available to the algorithm at planning time, consisting of the agent's experience after the specified scenario ID.

ID	Goal	Domain modification	l	l_k	Prior
(r-a)	c.s.d. `waypoint0`	Removed effect `communicated_soil_data` from the `communicate_soil_data` action	4	1	None
(r-b)	c.s.d. `waypoint0`	In addition to (r-a): removed `have_soil_analysis` precondition and effect	4	2	None
(r-c)	c.s.d. `waypoint0`	In addition to (r-b): removed `empty` precondition and effect	5	3	None
(r-d1)	c.s.d. `waypoint3`	`waypoint3` is one hop further away than `waypoint0`	7	3	None
(r-d2)	c.s.d. `waypoint5`	`waypoint5` is one hop further away than `waypoint3`	9	3	(r-d1)
(r-e)	c.s.d. `waypoint8`	`waypoint8` is three hops further away than `waypoint5`	15	3	None
(r-d3)	c.s.d. `waypoint8`	Same as (r-e)	15	3	(r-d1)

Table 2
Experiment scenarios in the **rearrangement task domain**. Explanations of abbreviations and symbols can be found in the caption of Table 1.

ID	Goal	Initial state	l	l_k	Prior
(a)	Cube on cupboard	Cube lying on table	4	1	None
(b)	Tall box inside shelf	Tall box standing on table	4	1	None
(c1)	Cube inside container1	Cube on table, container1 covered with lid	8	2	None
(c2)	Duck inside container1	Duck on table, container1 covered with lid	8	2	After (c1)
(c3)	Cube inside container2	Cube on table, container2 covered with lid	8	2	After (c1)
(c4)	Duck inside container2	Duck on table, container2 covered with lid	8	2	After (c1)
(d1)	Cube inside container2	Cube inside container1, both containers covered with lids	12	3	None
(d2)	Duck inside container2	Duck inside container1, both containers covered with lids	12	3	After (d1)

Table 3
Demonstrations supplied to our method for the experiment IDs defined in Tables 1 and 2.

	ID	Sequence Demo	Parameter Demo
PDDL benchmark domain	(r-a)	["communicate_soil_data1"]	[{"p1": "waypoint0"}]
	(r-b)	["sample_soil", "communicate_soil_data1"]	[{"p": "waypoint0"}, {"p1": "waypoint0"}]
	(r-c)	["drop", "sample_soil", "communicate_soil_data1"]	[{}, {"p": "waypoint0"}, {"p1": "waypoint0"}]
	(r-d1)	["drop", "sample_soil", "communicate_soil_data1"]	[{}, {}, {}]
	(r-d2)	["drop", "sample_soil", "communicate_soil_data1"]	[{}, {}, {}]
	(r-e)	["drop", "sample_soil", "communicate_soil_data1"]	[{}, {}, {}]
	(r-d3)	["drop", "sample_soil", "communicate_soil_data1"]	[{}, {}, {}]
Rearrangement task domain	(a)	["place"]	[{"obj": "cube1"}]
	(b)	["place"]	[{"obj": "tall_box"}]
	(c1)	["place", "place"]	[{"obj": "lid1"}, {"obj": "cube1"}]
	(c2)	["place", "place"]	[{"obj": "lid1"}, {"obj": "duck"}]
	(c3)	["place", "place"]	[{"obj": "lid2"}, {"obj": "cube1"}]
	(c4)	["place", "place"]	[{"obj": "lid2"}, {"obj": "duck"}]
	(d1)	["place", "place", "place"]	[{"obj": "lid1"}, {"obj": "lid2"}, {"obj": "cube2"}]
	(d2)	["place", "place", "place"]	[{"obj": "lid1"}, {"obj": "lid2"}, {"obj": "duck"}]

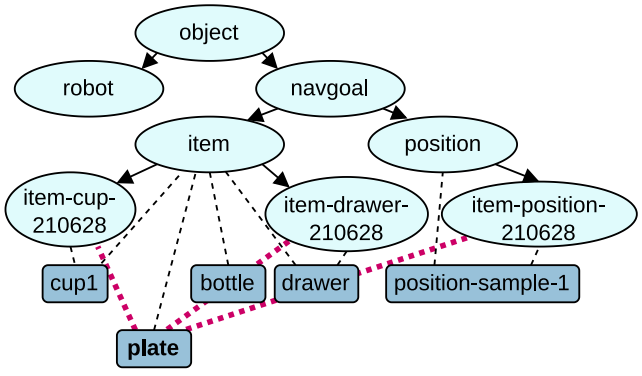


Fig. 6. Example for type constraint relaxation at the beginning of the generalization procedure. The existing meta action to place a cup into a drawer was discovered with "cup1". To extend it to "plate", the new entity is temporarily allowed to generalize to available types (violet dotted lines), thus fitting the existing meta action. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Experiments (r-d2) and (r-d3) show that our method can greatly benefit from such experience collected in a previous exploration run. Both experiments leverage knowledge gained from solving (r-d1) and achieve 100% planning success with exploration times that are reduced by several orders of magnitude independent of the sequence length. The exploration efficiency gain from our generalization step is directly observable by comparing the runtimes of (r-e) and (r-d3), which are both attempting to solve the same planning problem. (r-d3) is given access to the updated domain description after solving (r-d1) whereas (r-e) explores from scratch. The resulting reduction in exploration time is on the order of 10^4.

The results also indicate that supplying a simple user demonstration speeds up the exploration procedure significantly, and yields a success rate of 100% for all the experiments we conducted. In Table 4, we report averaged success rates and runtimes, which highlight our observations described above. Note that the runtime numbers ignore instances where the methods timed out due to failing to find a solution within the time budget.

5.5. Results: Rearrangement task domain

In Fig. 9, we report the results of experiments in the rearrangement task domain. Here, the performance of the MCTS baseline is consistently worse in all experiments, both in success rate and execution time, which is also confirmed by average success rates and runtimes reported in Table 4. Compared to experiments in the PDDL benchmark domain, simulating candidate sequences takes a significant time proportion, making our method's overhead of calling the symbolic planner negligible.

The results demonstrate the benefits of our sequence completion and generalization routines. Our approach exploits the underlying PDDL planner for sequence completion, thereby only needing to explore over the shorter key action sequences. By design, this is more efficient compared to MCTS, which must search over the full sequence of actions. Runtime results for scenarios (a)–(c1) and (d1) directly show this increase in exploration cost between our method and MCTS.

In addition, our method has a chance to find a successful sequence at every iteration during exploration. This is possible since a successful sequence can be discovered as a part of the full sequence we sample to have a fixed and overestimated length. Sequence refinement allows our system to extract the shortest sub-sequence required to achieve the goal. In contrast, MCTS is limited to sequentially building up its search tree from short sequences to longer ones.

MCTS also offers no mechanism to leverage previous experiences over tasks and thus always has to explore from scratch. Our approach explicitly encodes new meta actions into the domain description and provides a framework for generalization to new tasks. This allows for shorter runtimes in scenarios (c2)–(c4) and additionally a greatly improved success rate in (d2), where MCTS fails in 37/50 runs whereas our approach only fails in 1/50 runs. The remaining computation needed to generalize is mainly due to sampling continuous parameters, since these cannot be assumed to generalize between entities (e.g. a new placement pose is needed to place the cube from (c1) into a different receptacle).

The results shown in Fig. 9 indicate that being provided with a user demonstration, our algorithm finds solutions faster, and with a success rate of 99%. The only scenario in which failures occurred is (b), as a very accurate place position, which is not provided as part of the demonstration (see Table 3 for which parameters were supplied as demonstrations), needs to be discovered. This leads to the exploration timing out before a feasible place parameterization was found. Overall, the results show that leveraging user

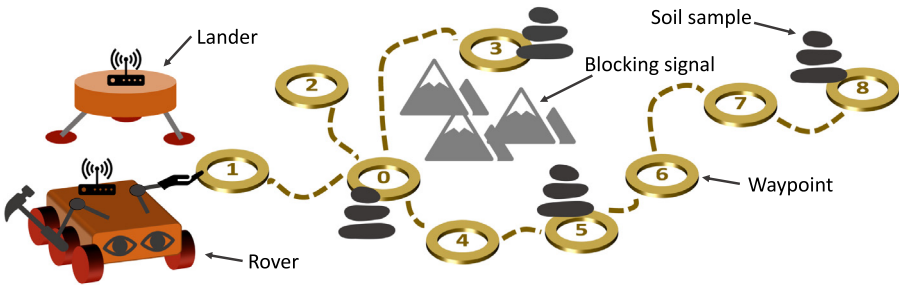


Fig. 7. In the rover domain, one (or several) rovers are tasked with collecting samples. For this, the rover can navigate between waypoints, sample soil, and communicate results back to the lander, for which it needs to be at a waypoint the lander is visible from.

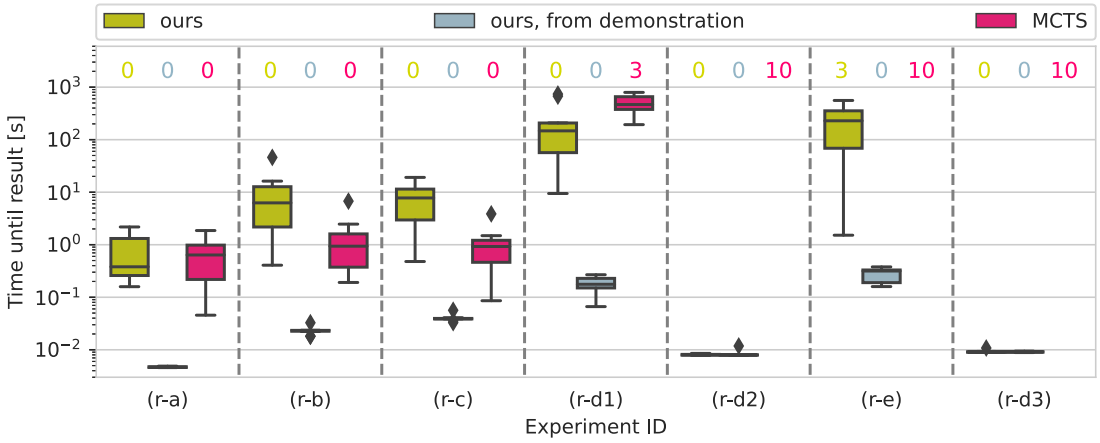


Fig. 8. Runtimes and success rates of our method (without and with demonstrations) and the MCTS baseline for experiments in the **PDDL benchmark domain**. Numbers in the top row indicate the number of failed runs out of 10. Only successful runs are included in the boxplots.

Table 4
Success rate and timing results, averaged over successful runs for each method. n is the total number of runs with the specified method in the specified domain.

Domain	Method	n	Avg. success rate [–]	Avg. runtime [s]
PDDL benchmark	ours	70	0.96	60.92
	ours, from demo.	70	1.0	0.08
	MCTS	70	0.53	96.15
Rearrangement task	ours	400	0.88	111.60
	ours, from demo.	400	0.99	110.31
	MCTS	400	0.74	337.70
Combined	ours	470	0.89	103.46
	ours, from demo.	470	0.99	93.75
	MCTS	470	0.71	310.86

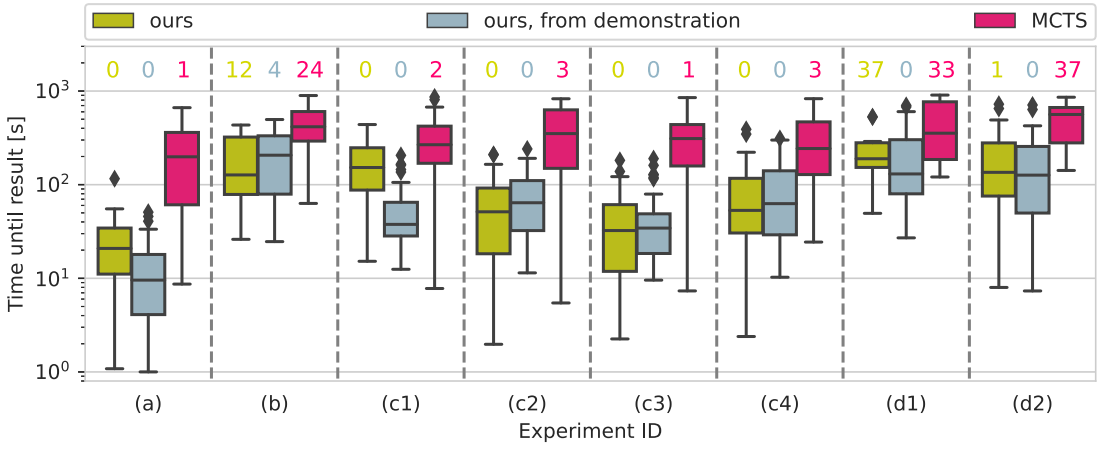


Fig. 9. Runtimes and success rates of our method (with and without demonstrations) and the MCTS baseline for experiments in the **rearrangement task domain**. Numbers in the top row indicate the number of failed runs out of 50. Only successful runs are included in the boxplots.

```
(:action place-inside-211005224421
  :parameters (...)
  :precondition (and
    (in-reach ?position_sample_4)
    (not (empty-hand))
    (grasped-with ?cube1 ?grasp0)
    (not (on ?container1 ?lid1)))
  :effect (and
    (inside ?container1 ?cube1)
    (empty-hand)
    (not (grasped-with ?cube1
      ?grasp0))))
```

(a) Place cube into container

```
(:action place-on-211005224421
  :parameters (...)
  :precondition (and
    (in-reach ?position_sample_7)
    (not (empty-hand))
    (grasped-with ?lid1 ?grasp0))
  :effect (and
    (not (on ?container1 ?lid1))
    (empty-hand)
    (not (grasped-with ?lid1
      ?grasp0)))
)
```

(b) Remove lid from container

Fig. 10. PDDL descriptions of actions that were automatically added to the symbolic description after exploration for task (c1) (see Table 2). Dependency-inducing predicate is highlighted in **bold**.

demonstrations are a viable avenue for effectively and efficiently improving a mobile manipulation system (MMS) skill set with only a small added need for manual input.

Qualitatively, the meta actions discovered by our algorithm were as expected. As an example, Fig. 10 shows two actions our algorithm added to the symbolic description after the exploration procedure for task (c1) completed. The action shown on the left achieves the user-defined goal of placing the cube inside the container. It has the precondition that the lid is not on the container, which can be achieved by the action shown on the right. Splitting the actions in this way through precondition discovery allows the agent to skip the action on the right if no lid is initially present on the container. For shorter tasks ((a) and (b)), a single meta action is added. For tasks requiring multiple steps ((c1)–(d2)), one meta action is added to achieve the goal, and additional ones are added for fulfilling preconditions. Thanks to the precondition discovery, the set of meta actions is thus kept modular.

Furthermore, in Fig. 11 we show an exemplary qualitative result for the generalization occurring in scenario (c4). As described in Section 5.2 and in Table 2, the goal of scenario (c4) is to place the duck into a container, given the experience from scenario (c1). This previous experience consists of having learnt to place a cube into a different container. It is encoded by the action description shown in Fig. 11 and the new types added to the cube and the first container in the list of entities before running (c4). During exploration for scenario (c4), our method then discovered that the same sequence of actions can be used to successfully place a different object into a different container, thanks to the generalization component introduced in Section 4.7. Therefore, instead of creating a new symbolic action for this second exploration run, our generalization procedure causes the new objects' type lists to be extended, such that the existing action is applicable to the new objects. Apart from speeding up the exploration significantly (as shown in Fig. 9), the generalization procedure helps to keep the symbolic description small and thus efficient.

The experiments also showed that the complexity increases dramatically for longer sequence lengths. For example, in scenario (d1), the success rates are low, indicating that no solution could be found within the available time budget. With the motivation to improve efficiency, we evaluated variations of our method. During exploration, our method samples key action sequences of length $l_{\max}$. Intuitively, to find a successful sequence faster, we could constrain sampling to the correct sequence length for the current task. However, this length is generally unknown a priori. Nevertheless, we can increase attention to shorter sequence lengths through custom sampling strategies. Experiments varying the sequence length schedule (e.g. exploring a fixed number of 1 key action plans, followed by plans with 2 key actions, etc., or looping through sampling plans with 1, then 2 key actions, and so on) showed that such strategies can perform better in certain cases compared to the general version of our method (which only ever samples plans with the specified maximum number of key actions). However, we conclude that the performance benefit is not worth sacrificing general applicability nor is it worth the additional overhead introduced by manually specifying the sampling schedule. A full description of the tested sampling strategies and the associated results are available in Appendix.

6. Conclusion and future work

In this work, we propose an important component needed to deploy an autonomous agent in open-ended domains. Due to high complexity, many entities in the environment, and changing demands of system operators (users), encountering unseen tasks is inevitable. In consequence, an agent's model of its interaction with the world, which is required by symbolic planning or TAMP, needs to grow and evolve to meet these diverse challenges.

Instead of relying on manual adaptation, we presented a symbolic planning system that automatically extends its abstract skill set to achieve goals for which either the symbolic planning or the plan execution failed. Our results show that the proposed algorithm extends the symbolic description only as much as necessary to achieve the goals, to make sure that planning stays sound and tractable, mainly thanks to generalization and precondition discovery. Furthermore, our measures to run the exploration more efficiently, i.e. sequence completion and generalization, greatly reduce the computational complexity, outperforming the MCTS baseline we compared against, thus increasing our method's value in practice. The exploration for new tasks can optionally be further accelerated through partial demonstrations, which strike a balance between being easy for a user to supply, and effectively improving success rate and runtime.

In the future, to avoid the need for manually grounding predicates, we plan to learn models of predicates based on demonstrations and interactions with the environment. Furthermore, we aim to leverage data gathered in multiple encounters of a task to increase the accuracy of preconditions and effects of action abstractions.

To apply our method in a real-world scenario, two main additions would be required. First, a perception module as mentioned in Section 4.2 is needed. Second, we need implementations for the basic skills that our method is relying on. The skills we chose in this work can be implemented with state-of-the-art tools such as, e.g. grasp synthesis and motion planning both for

Goal

```
(:goal (inside container2 duck)
```

Action

```
(:action place-inside-2110
  :parameters (
    ?cube - item-cube-2110
    ?container - item-container-2110
    [...])
  :precondition (and
    (in-reach ?position_sample_4)
    (grasped-with ?cube ?grasp0))
  :effect (and
    (inside ?container ?cube)
    (not (grasped-with ?cube
      ?grasp0))))
```

Entities before generalizing

```
(:objects
  white_cube - item item-cube-2110
  container1 - item item-container-2110
  duck       - item
  container2 - item)
```

Entities after generalizing

```
(:objects
  white_cube - item item-cube-2110
  container1 - item item-container-2110
  duck       - item item-cube-2110
  container2 - item item-container-2110)
```

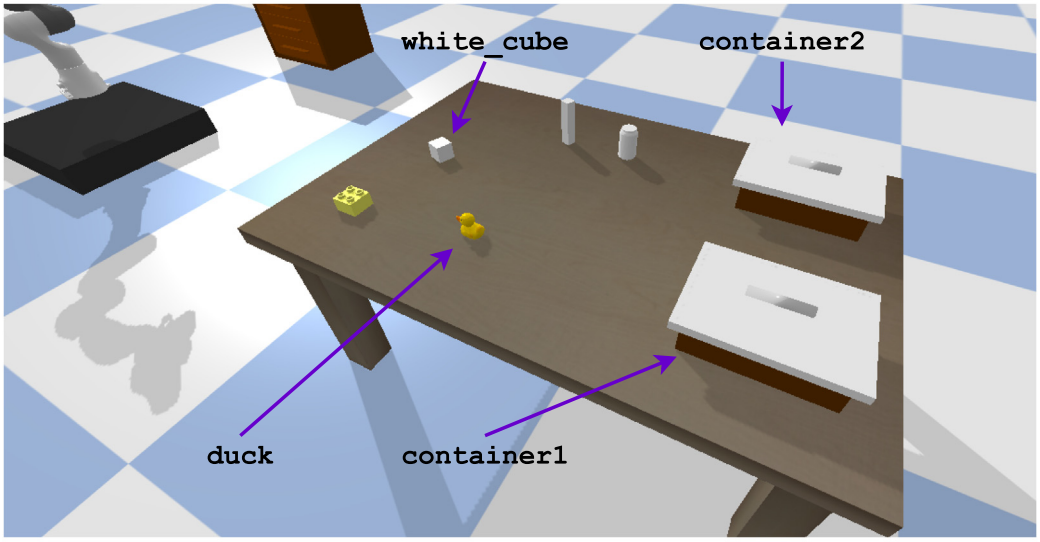


Fig. 11. Qualitative result for the generalization occurring in scenario (c4) (see Table 2). The action description (middle left) and the entities before generalizing (top right) stem from previously solving scenario (c1). From that initial state, planning for the goal (top left) initially fails, since the new action's type restrictions do not allow it to be applied to `duck` and `container2`. However, our generalization procedure is able to discover that the action is applicable to the new objects, this is reflected in the entities after generalizing by adding the respective types to the new objects (middle right).

robot navigation and manipulation. Given these components, we envision the whole system to operate as follows: upon receiving a new goal defined by a user, the symbolic planner attempts to find a solution with the current symbolic description. If successful, the solution sequence is executed and its success is monitored using the perception system and the learned predicate definitions mentioned above. Upon failure, either during planning or during execution of the sequence, the exploration algorithm is executed in simulation to identify a successful sequence, which can then be executed on the real robot. In future work, we plan demonstrate this integrated system.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Julian Foerster reports financial support was provided by ABB Corporate Research. Julian Foerster reports financial support was provided by Huawei Technologies Co Ltd.

Data availability

The code for this work is published at https://github.com/ethz-asl/high_level_planning.

Acknowledgments

This work was supported in part by ABB Corporate Research, the European Union's Horizon 2020 research and innovation programme under grant agreement No 955356, and the ETH Foundation with an unrestricted gift from Huawei Technologies.

Appendix. Sequence sampling variations

The general configuration of our method that is discussed in the main paper will from now on be referred to as "ours, full length, all actions" (OFLA). As stated in Section 5.5 of the main paper, we investigate methods for reducing the search space that needs to be covered during exploration. We use the **rearrangement task domain** for this set of experimental evaluations.

The first simplification we introduce in this vein is limiting the set of actions we sample from in line 3 of Algorithm 1. Specifically, the *navigation* and *grasp* skills are excluded from key action sampling. The rationale for this is that sequences with these two skills as key actions only rarely lead to achieving the goal, while they can be inferred during sequence completion from other key actions in most cases.

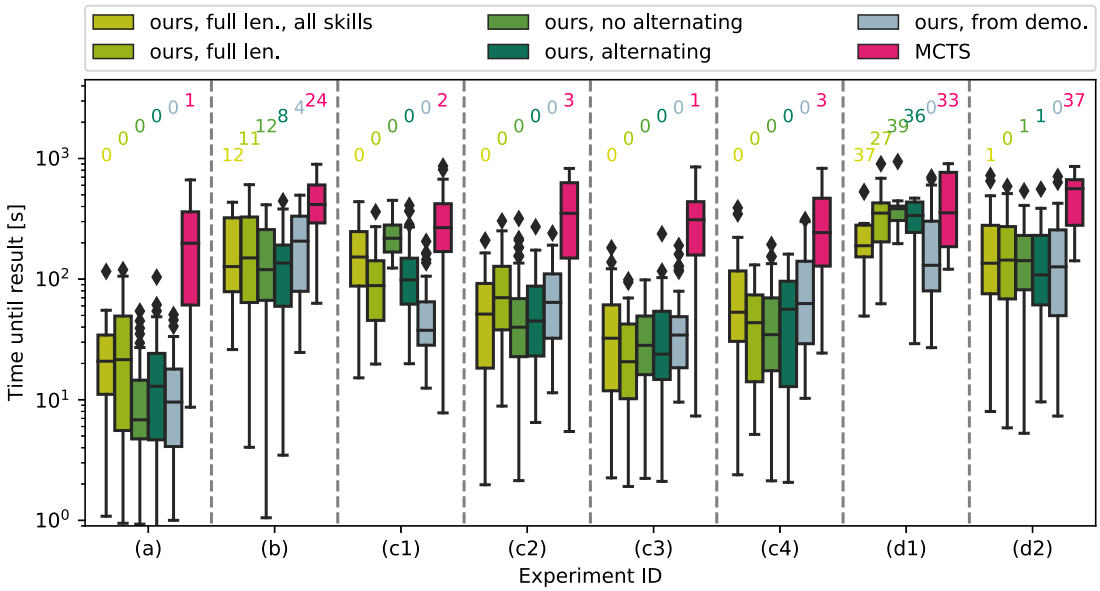


Fig. A.12. Runtime comparison of all sequence exploration methods for the scenarios shown in Table 2. Each method was run 50 times for each scenario, with a time budget of 900 s. The numbers at the top of the diagram indicate how many of the 50 trials timed out before finding a feasible solution (lower is better). Only the times taken by successful runs are included in the plot.

In addition to this, we investigate focusing search on shorter key action sequences. We introduce the following three alternatives to OFLA.

1. *Full length*, labeled as "ours, full length" (OFL). Similar to OFLA, a sequence of maximum length is sampled in every iteration, i.e. $l = l_{\max}, \ldots, l_{\max}$, the only difference being that this variant (same as the following two) samples from the reduced action set described above.
2. *Alternating sequence length*, labeled as "ours, alternating" (OA). In every iteration, a different length is selected according to the order $l = 1, 2, 3, \ldots, l_{\max}, 1, 2, \ldots$. This leads to sampling each sequence length equally often within the time budget.
3. *Increasing sequence length*, labeled as "ours, no alternating" (ONA). Each sequence length is tested for a fixed duration $t_{\text{seq}} = T_{\max}/l_{\max}$, i.e. $l = 1, 1, \ldots, 1, 2, 2, \ldots, l_{\max}, l_{\max}$. This leads to sampling each sequence for an equal duration.

The sequence length is selected in every iteration of the exploration algorithm by the respective method and passed to the sequence sampler in line 3 of Algorithm 1. All three of these methods sample actions from the reduced action set described above. Results are reported in Fig. A.12.

For all methods, the computational complexity tends to increase and the success rate tends to drop with increasing sequence length. An exception is experiment (b), where a precise placing location has to be found by all methods, which increases the average duration until success. We expected ONA to fare best in this case since it starts by exclusively sampling sequences of length 1, which is exactly the minimum number of key actions needed in this case. However, the results show that longer sequence lengths during sampling are not disadvantageous. We believe the reason for this is that a successful action can also be found as part of a longer sequence. Sequence refinement ensures that unnecessary parts of a sequence are discarded before using it for extending the symbolic description. For problems where sampling longer sequences is required, ONA performs the worst as expected, because its sampling budget for shorter-length sequences must be exhausted before longer sequences are considered. For OA, OFL and OFLA, this is not the case, resulting in better performance in scenarios (c1) and (d1).

Comparing OFLA to the other variants shows that the advantage of sampling from fewer skills is not as pronounced as initially expected. This is positive, indicating that our method can generalize to other skill sets without selecting skills to exclude. Nevertheless, the option to exclude skills could still be beneficial for very long scenarios or an exceptionally large basic skill set. Excluding skills could either be handled by a domain expert, or more elegantly using an automated system to judge how likely certain skills are key actions based on past experience, and adjusting the sampling scheme accordingly. Investigating this is left to future work.

Overall, it is beneficial to know the exact sequence length required to solve the task as this reduces the exploration effort. Of course, this information is not available at planning time in most cases. Since it is possible to extract a shorter sequence from a longer one via our precondition discovery and sequence refinement, OFL and OFLA may be preferential to the other sequence exploration variants. OFL and OFLA are the most successful for long sequence lengths while only incurring a small planning time overhead at short sequence lengths.

In scenarios where prior knowledge is available ((c2)–(c4), (d2)), performance is improved for all variants of our method, showing the benefit of generalization. Note that OA, ONA, OFL and OFLA perform similarly in these scenarios, since the considered sequence length is the same for all when generalizing.

References

[1] G. Ajaykumar, M. Steele, C.-M. Huang, A survey on end-user robot programming, ACM Comput. Surv. 54 (8) (2021) 1–36, http://dx.doi.org/10.1145/3466819.

[2] D. Long, M. Fox, Progress in AI planning research and applications, UPGRADE: Eur. J. Inform. Prof. 3 (5) (2002) 10–25.

[3] E. Karpas, D. Magazzeni, Automated planning for robotics, Ann. Rev. Control, Robot. Autonom. Syst. 3 (2019) 1–23.

[4] D. Leidner, A. Dietrich, F. Schmidt, C. Borst, A. Albu-Schäffer, Object-centered hybrid reasoning for whole-body mobile manipulation, in: IEEE International Conference on Robotics and Automation, 2014.

[5] S.S. Srinivasa, D. Ferguson, C.J. Helfrich, D. Berenson, A. Collet, R. Diankov, G. Gallagher, G. Hollinger, J. Kuffner, M.V. Weghe, HERB: A home exploring robotic butler, Auton. Robots 28 (1) (2010) 5–20, http://dx.doi.org/10.1007/s10514-009-9160-9.

[6] C.R. Garrett, R. Chitnis, R. Holladay, B. Kim, T. Silver, L.P. Kaelbling, T. Lozano-Pérez, Integrated task and motion planning, Ann. Rev. Control, Robot. Auton. Syst. 4 (1) (2021) 265–293, http://dx.doi.org/10.1146/annurev-control-091420-084139.

[7] T. Ren, G. Chalvatzaki, J. Peters, Extended task and motion planning of long-horizon robot manipulation, 2021, arXiv:2103.05456.

[8] Z. Wang, C.R. Garrett, L.P. Kaelbling, T. Lozano-Pérez, Learning compositional models of robot skills for task and motion planning, Int. J. Robot. Res. 40 (6–7) (2021) 866–894, http://dx.doi.org/10.1177/02783649211004615.

[9] C.R. Garrett, T. Lozano-Pérez, L.P. Kaelbling, PDDLStream: Integrating symbolic planners and blackbox samplers via optimistic adaptive planning, in: Proceedings of the International Conference on Automated Planning and Scheduling, vol. 30, (no. 1) 2020, pp. 440–448.

[10] R.S. Sutton, D. Precup, S. Singh, Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning, Artificial Intelligence 112 (1–2) (1999) 181–211, http://dx.doi.org/10.1016/S0004-3702(99)00052-1.

[11] A. Bagaria, G. Konidaris, Option discovery using deep skill chaining, in: International Conference on Learning Representations, 2020.

[12] P. Morere, L. Ott, F. Ramos, Learning to plan hierarchically from curriculum, IEEE Robot. Autom. Lett. 4 (3) (2019) 2815–2822, http://dx.doi.org/10.1109/LRA.2019.2920285.

[13] L. Chrpa, M. Vallati, T.L. McCluskey, On the online generation of effective macro-operators, in: 24th International Joint Conference on Artificial Intelligence, 2015, pp. 1544–1550.

[14] V. Sarathy, M. Scheutz, MacGyver problems: AI challenges for testing resourcefulness and creativity, Adv. Cogn. Syst. 6 (2018) 31–44.

[15] D. Angelov, Y. Hristov, M. Burke, S. Ramamoorthy, Composing diverse policies for temporally extended tasks, IEEE Robot. Autom. Lett. 5 (2) (2020) 2658–2665.

[16] B. Kim, L. Shimanuki, Learning value functions with relational state representations for guiding task-and-motion planning, in: Conference on Robot Learning, PMLR, 2020, pp. 955–968.

[17] R. Strudel, A. Pashevich, I. Kalevatykh, I. Laptev, J. Sivic, C. Schmid, Learning to combine primitive skills: A step towards versatile robotic manipulation, in: IEEE International Conference on Robotics and Automation, 2020, arXiv:1908.00722.

[18] A. Curtis, M. Xin, D. Arumugam, K. Feigelis, D. Yamins, Flexible and efficient long-range planning through curious exploration, in: International Conference on Machine Learning, 2020, pp. 2238–2249.

[19] T. Silver, R. Chitnis, J. Tenenbaum, L.P. Kaelbling, T. Lozano-Perez, Learning symbolic operators for task and motion planning, in: IEEE/RSJ International Conference on Intelligent Robots and Systems, 2021, pp. 3182–3189, URL https://ieeexplore.ieee.org/abstract/document/9635941.

[20] V. Sarathy, D. Kasenberg, S. Goel, J. Sinapov, M. Scheutz, SPOTTER: Extending symbolic planning operators through targeted reinforcement learning, in: International Conference on Autonomous Agents and Multiagent Systems, 2021, pp. 1118–1126, Online.

[21] A. Arora, H. Fiorino, D. Pellier, M. Métivier, S. Pesty, A review of learning planning action models, Knowl. Eng. Rev. 33 (20) (2018) 1–25, http://dx.doi.org/10.1017/S0269888918000188.

[22] G. Konidaris, L.P. Kaelbling, T. Lozano-Perez, From skills to symbols: Learning symbolic representations for abstract high-level planning, J. Artificial Intelligence Res. 61 (2018) 215–289.

[23] E. Ugur, J. Piater, Bottom-up learning of object categories, action effects and logical rules: From continuous manipulative exploration to symbolic planning, in: IEEE International Conference on Robotics and Automation, IEEE, 2015, pp. 2627–2633.

[24] S. James, B. Rosman, G. Konidaris, Learning portable representations for high-level planning, in: International Conference on Machine Learning, 2020, pp. 4682–4691.

[25] M. Diehl, C. Paxton, K. Ramirez-Amaro, Automated generation of robotic planning domains from observations, in: IEEE/RSJ International Conference on Intelligent Robots and Systems, 2021, pp. 6732–6738, http://dx.doi.org/10.1109/IROS51168.2021.9636781.

[26] M. Fox, D. Long, PDDL2.1: An extension to PDDL for expressing temporal planning domains, J. Artificial Intelligence Res. 20 (2003) 61–124, http://dx.doi.org/10.1613/jair.1129.

[27] M. Grinvald, F. Furrer, T. Novkovic, J.J. Chung, C. Cadena, R. Siegwart, J. Nieto, Volumetric instance-aware semantic mapping and 3D object discovery, IEEE Robot. Autom. Lett. 4 (3) (2019) 3037–3044, http://dx.doi.org/10.1109/LRA.2019.2923960.

[28] D. Auger, A. Couëtoux, O. Teytaud, Continuous upper confidence trees with polynomial exploration – consistency, in: Joint European Conference on Machine Learning and Knowledge Discovery in Databases, Vol. 7908, Springer, 2013, pp. 194–209, http://dx.doi.org/10.1007/978-3-642-40988-2_13.

[29] D. Long, M. Fox, The 3rd international planning competition: Results and analysis, J. Artificial Intelligence Res. 20 (2003) 1–59, http://dx.doi.org/10.1613/jair.1240, URL https://www.jair.org/index.php/jair/article/view/10351.

[30] E. Coumans, Y. Bai, Pybullet, a Python module for physics simulation for games, robotics and machine learning, 2016, http://pybullet.org.

[31] J. Hoffmann, The metric-FF planning system: Translating "ignoring delete lists" to numeric state variables, J. Artificial Intelligence Res. 20 (2003) 291–341, http://dx.doi.org/10.1613/jair.1144.

Julian Förster received his bachelor's degree in mechanical engineering at ETH Zurich (2015) as well as a master's degree with distinction in mechanical engineering with a specialization in robotics and control systems at ETH Zurich (2018). Currently, he is a doctoral student at the Autonomous Systems Lab at ETH Zurich, with research interests in planning for robotics, scene understanding and machine learning.

Lionel Ott is a senior researcher in the Autonomous Systems Lab at ETH Zürich. Prior to this, he was a research fellow in the School of Computer Science at the University of Sydney. He obtained his Ph.D. at the University of Sydney in 2014.

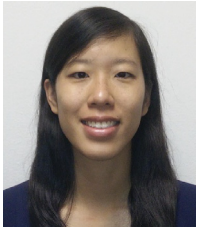
His research lies at the intersection of machine learning and robotics, intending to develop methods, techniques, and systems that allow an autonomous agent to build and maintain a representation of the ever-changing world in which the agent operates to achieve the given task. Thus a recurring theme in his research is the focus on model learning and decision-making in frameworks that can account for and reason about uncertainties.

Juan Nieto is a Principal Research at the Mixed Reality and AI Zurich Lab working with a team of scientists and engineers to develop advanced perception capabilities for HoloLens at Microsoft. Before joining Microsoft he was Deputy Director at the Autonomous Systems Lab (ASL) at ETH Zurich. In ASL, together with Roland Siegwart, he led a team of Ph.D. students, engineers and postdocs to bring new concepts for mobile robots, and to develop new algorithms for robot autonomy. He obtained his Ph.D. at the University of Sydney in the Australian Centre for Field Robotics. He is best known for his work in perception and simultaneous localization and mapping for mobile robots. More recently, his research was focused on developing new ideas for building tighter connections between perception and control to enable robots to perform physical interaction in a safe and autonomous manner. In Microsoft, his research is focused in building new perception capabilities for Mixed Reality applications involving both people and robots.

Nicholas Lawrance is a senior research scientist in the Robotics and Autonomous Systems Group at CSIRO Data61 in Queensland, Australia. His research focuses on adaptive planning, particularly in the presence of environmental uncertainty. Research interests include stochastic reasoning, adaptive sampling, and modeling of uncertain, continuous phenomena. Applications include aerial and underwater domains, particularly for long-duration robotic missions including wind estimation and autonomous soaring for fixed-wing aerial vehicles. He completed his Ph.D. at the Australian Centre for Field Robotics at the University of Sydney (2011), and worked as a postdoctoral scholar in the Robotic Decision Making Laboratory (RDML) at Oregon State University (2013–2017), and as a senior researcher at the Autonomous Systems Lab (ASL) at ETH Zürich (2018–2022).

Roland Siegwart is a Professor for autonomous mobile robots with ETH Zurich, Founding Co-Director of the Technology Transfer Center, Wyss Zurich and Board Member of multiple high-tech companies. From 1996 to 2006, he was a Professor with EPFL Lausanne, held visiting positions with Stanford University and NASA Ames and was Vice President of ETH Zurich from 2010 to 2014. His research interests include the design, control, and navigation of flying, and wheeled and walking robots operating in complex and highly dynamic environments.

Prof. Siegwart received the IEEE RAS Pioneer Award and IEEE RAS Inaba Technical Award. He is among the most cited scientists in robots world-wide, Co-Founder of more than half a dozen spin-off companies, and a strong promoter of innovation and entrepreneurship in Switzerland.



Jen Jen Chung is an Associate Professor in Mechatronics within the School of Information Technology and Electrical Engineering at The University of Queensland. Her current research interests include perception, planning and learning for robotic mobile manipulation, algorithms for robot navigation through human crowds, informative path planning and adaptive sampling. Prior to working at UQ, Jen Jen was a Senior Researcher in the Autonomous Systems Lab (ASL) at ETH Zürich from 2018–2022 and was a postdoctoral scholar at Oregon State University researching multiagent learning methods from 2014–2017. She completed her Ph.D. on information-based exploration–exploitation strategies for autonomous soaring platforms at the Australian Centre for Field Robotics in the University of Sydney. She received her Ph.D. (2014) and B.E. (2010) from the University of Sydney.