Efficient task planning using abstract skills and dynamic road map matching

Khen Elimelech, Lydia E. Kavraki, and Moshe Y. Vardi

Department of Computer Science,
Rice University,
Houston, TX 77005, USA.
{elimelech,kavraki,vardi}@rice.edu

Abstract. Task planning is the problem of finding a discrete sequence of actions to achieve a goal. Unfortunately, task planning in robotic domains is computationally challenging. To address this, in our prior work, we explained how knowledge from a successful task solution can be cached for later use, as an "abstract skill." Such a skill is represented as a trace of states ("road map") in an abstract space and can be matched with new tasks on-demand. This paper explains how one can use a library of abstract skills, derived from past planning experience, to reduce the computational cost of solving new task planning problems. As we explain, matching a skill to a task allows us to decompose it into independent sub-tasks, which can be quickly solved in parallel. This can be done automatically and dynamically during planning. We begin by formulating this problem of "planning with skills" as a constraint satisfaction problem. We then provide a hierarchical solution algorithm, which integrates with any standard task planner. Finally, we experimentally demonstrate the computational benefits of the approach for reach-avoid tasks.

1 Introduction

1.1 Background

In Artificial Intelligence (AI) and robotics research, task planning refers to the problem of finding a discrete sequence of actions to achieve a certain goal [1]. Tasks can be specified to solve various robotic problems, such as manipulation, autonomous navigation, target tracking, and others. When the planning domain is large and complex, online task planning can become a computational challenge. It is hence important to find ways to reduce this complexity, such that a task plan can be found in a timely manner. This is especially important since often in robotic domains task planning is a part of the broader Task And Motion Planning (TAMP) problem [2]. This problem also involves translating the symbolic actions to feasible motion plans for the robot — a matter that further adds to

Work on this paper was supported in part by NSF-IIS-1830549.

the computational cost. One way to reduce the task planning cost is exploiting knowledge from past planning experience, to expedite the solution of new tasks.

To this end, in our prior work [3], we introduced the concept of *abstract skills*. In robotics, the term "skill" often refers to a control policy that is learned from numerous demonstrations to achieve a certain goal (e.g., in [4,5]). Uniquely, our "skills" are not represented as policies of *actions*, but as *traces of states*, serving as plan "road maps." This approach, which we motivated in the aforementioned work, allows intuitive cross-domain skill transfer and flexible adaptation to new tasks. Further, skills can be cached after each successful task solution, which means we only require a single execution to learn a generalizable skill. This generalizability is made possible by caching the skills as road maps in an *abstract domain*, and not directly in the task domain. As such, each abstract road map can be *dynamically* reconstructed (on-demand) to match a variety of tasks, in a variety of domains, and expedite their solution. As we shall explain, this caching and reconstruction of abstract skills can be done automatically using suitable "abstraction keys." These are generic, user-specified devices that indicate how to perform the required mapping of road maps to/from an abstract domain.

For example, consider a mobile robot that managed to reach a designated goal by performing a certain maneuver. We can identify a sequence of critical way points in the robot's executed path (i.e., a road map); then, we can try to strip robot-and-domain-specific attributes from the states in the road map, before caching it as an abstract skill. Later, we can try to use this abstract skill to guide the solution of a matching task by a new robot in a new domain.

1.2 Contribution

While our previous work suggested and motivated a theoretical framework for skill transfer, the aim of this paper is to allow practical execution of those ideas. Our goal is to explain how one can use a given "library" of abstract skills, derived from past successful task solutions, to reduce the computational cost of solving new task planning problems. Specifically, we provide the following contributions: (i) formulation of the problem of "planning with abstract skills" as a constraint satisfaction problem (in Section 4); (ii) an efficient algorithm for solving this problem (in Section 5); and (iii) a practical demonstration of the algorithm and the computational benefits of using skills for task planning (in Section 6). Additional contributions, provided in the appendices, include: (iv) formulation of new abstraction keys for geometric domains (in Appendix A); and (v) complexity analysis for the algorithm (in Appendix B). For visualization and ease of explanation, we precede the contributions by providing an exemplary scenario for planning with skills in a discrete geometric domain (in Section 3).

To clarify, we do *not* seek to provide a new task planner, nor compete with existing ones. In fact, our simple-yet-effective approach integrates with any standard task planner of choice and allows to use it more efficiently, thanks to the incorporation of skills. The problem formulation and high-level algorithm are applicable to any task type, while specific practical details are provided for "reach-avoid" tasks — a common class of tasks in robotic contexts; this is yet

another contribution in relation to our prior work, in which we only examined "classical planning" tasks (with a single goal). As implied, our problem formulation and algorithm are hierarchical: at the top level, we try to match one of the cached skills to the task; and at the lower level, plan with this skill. As we shall see, matching an abstract skill to the task often allows us to *decompose* it into independent sub-tasks, which can be quickly solved in parallel. This matching and decomposition can be done automatically and dynamically during planning.

1.3 Related work

Task planning is a prominent problem in AI research, where it is often referred to as "automated planning" or "AI planning" [6]. The standard technique to solve such problems, both in AI and robotics, is using a heuristic graph search (e.g., [7,8]). Another prominent technique is using a reduction to a satisfiability problem (e.g., [9,10]), which can be solved using an automated constraint satisfier. As mentioned, our algorithm relies on and integrates with existing planners.

Hierarchical Task Network (HTN) [11] is a relevant variation of the standard task planning formalism. HTN sees planning as the problem of decomposing a task to a hierarchy of atomic sub-tasks. However, there is a significant difference between this framework and ours: here, the sub-tasks into which we decompose a problem are dynamically defined by matching a skill to the task; in contrast, in HTN, the sub-tasks are predefined, and we only search for their right construction. Hierarchical Goal Network (HGN) [12] is another relevant variation. HGN sees planning towards a goal as the problem of identifying a sequence of sub-goals. The idea of sub-goal decomposition is similar to our notion of road map matching. Yet, this formalism is more restrictive, as it (i) only considers plans that monotonously progress towards the goal; (ii) relies on domain-specific decomposition "methods"; and (iii) is not suitable for general tasks.

A similar hierarchy of planning levels, with inter-level induced constraints, is also expressed in various other robotic planners, such as SyCLoP [13], IDTMP [14], and HPN [15]; yet, these planners aim to return motion plans and not task plans, as we do. More importantly, in contrast to the mentioned works, our hierarchical approach does not suggest to plan in an abstract domain. The top level of our algorithm can be thought of as meta-planning; after that, we look for a plan directly in the task domain.

The usage of road maps for planning is most prominent in motion planning, i.e., in continuous geometric domains. Such methods, led by the Probabilistic Road Map (PRM) algorithm [16], rely on a randomly-sampled sparse graph, over which a plan can be calculated, instead of examining the entire continuous domain. Several points differentiate these works from ours: first, we consider planning in discrete domains; second, the road maps we rely on are derived from past experience and are not random; third, we do not plan over the states in the road map, as we consider them already ordered.

Finally, works in experience-based planning (e.g., [17]) often rely on a similarity metric between tasks, in order to retrieve relevant experience. We, however, avoid that, by directly matching the given task and an abstract skill.

2 Preliminaries

To be able to formulate the problem of task planning with abstract skills, we must first formulate (i) the standard task planning problem, and (ii) the concept of abstract skills.

2.1 Task planning

Let us begin by formally defining the planning domain.

Definition 1. *A* task planning domain $D \doteq (\mathcal{S}, \mathcal{A}, \mathcal{T})$ *is comprised of a state space* $\mathcal{S}$ *(continuous or discrete), an action space* $\mathcal{A}$*, and a set* $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ *of discrete transitions. Transitions in the domain are* deterministic*, meaning,* $\forall (\boldsymbol{S}, a, \boldsymbol{S}') \in \mathcal{T}$*,* $\nexists \boldsymbol{S}'' \in \mathcal{S}$*, such that* $(\boldsymbol{S}, a, \boldsymbol{S}'') \in \mathcal{T}$*. An* execution $\boldsymbol{E}$ *in* D *is a sequence of alternating states and actions, i.e.,*

$$\boldsymbol{E} \doteq (\boldsymbol{S}_0, a_1, \boldsymbol{S}_1, a_2, \ldots, \boldsymbol{S}_n)\,. \qquad \texttt{// execution} \quad (1)$$

The execution $\boldsymbol{E}$ *is* feasible *in the domain if* $\forall i \in \{1, \ldots, n\}$*,* $(\boldsymbol{S}_{i-1}, a_i, \boldsymbol{S}_i) \in \mathcal{T}$*. In that case, we say that* $\boldsymbol{E}$ *is induced by the action sequence* $(a_1, \ldots, a_n)$*. We use* $\mathfrak{S}(\boldsymbol{E})$ *to mark the trace of states from* $\boldsymbol{E}$*.*

A task can be thought of as a collection of constraints on the states and/or actions of an execution. Such constraints can be, e.g., global, temporal, or cost-related. In this work we will focus on finite reach-avoid tasks.

Definition 2. *A* reach-avoid task $T \doteq (\mathcal{G}oals, \mathcal{S}_{avoid})$ *is comprised of a set* $\mathcal{G}oals$ *of goal regions* $\subseteq \mathcal{S}$*, and a region* $\mathcal{S}_{avoid} \subseteq \mathcal{S}$ *to avoid. For an execution* $\boldsymbol{E}$*, the task constraints require us to visit all the goal regions (in any order):*

$$\forall \mathcal{S}_{goal} \in \mathcal{G}oals,\ \exists \boldsymbol{S} \in \mathfrak{S}(\boldsymbol{E})\,,\ s.t.\ \boldsymbol{S} \in \mathcal{S}_{goal}, \qquad \texttt{// reach} \quad (2)$$

and never reach the avoid regions:

$$\mathfrak{S}(\boldsymbol{E}) \cap \mathcal{S}_{avoid} = \emptyset. \qquad \texttt{// avoid} \quad (3)$$

To ensure timely termination, we also demand that $\boldsymbol{E}$ *does not include additional transitions after all goal states are reached. Formally:*

$$\nexists i < \texttt{length}(\mathfrak{S}(\boldsymbol{E}))\,,\ s.t.\ \mathfrak{S}(\boldsymbol{E})[1\colon i]\ satisfies\ (2). \qquad \texttt{// terminate} \quad (4)$$

To solve a task, we should find a plan (i.e., a sequence of actions) to be applied from the current start state, whose execution is feasible in the domain and satisfies the task constraints.

Definition 3 (Task planning problem). *For a start state* $\boldsymbol{S}_{start}$*, a task* T*, and a domain* D*, find a finite sequence of actions* $(a_1, \ldots, a_n) \in \mathcal{A}^n$*, such that the sequence induces a feasible execution in* D*, starting from* $\boldsymbol{S}_{start}$*, and the execution satisfies all the task constraints (i.e., valid for the task).*

Note that here we only care to find a feasible plan and not an optimal one.

2.2 Abstract skills and abstraction keys

In our previous work [3], we introduced the concept of abstraction keys.

Definition 4. *A* public abstraction key $PK \doteq \left(\texttt{project}_p, \texttt{reconst}_p, \mathcal{P}\right)$ *is comprised of a parametric state projection function* $\texttt{project}_p \colon \mathcal{S} \to \Xi$*; its inverse — a parametric state reconstruction function* $\texttt{reconst}_p \colon \Xi \to \mathcal{S}$*; and a parameter space* $\mathcal{P}$*, such that* $p \in \mathcal{P}$*. The sets of parameters that are valid for projection of* $\boldsymbol{S}$ *and for reconstruction of* ξ *are marked* $\mathcal{P}_{\boldsymbol{S}}$*,* $\mathcal{P}^{\xi}$*, respectively.*

As we shall soon demonstrate, every public abstraction key allows us to perform a certain type of transformation on states. Intuitively, applying the projection function removes a property from a state — leading to an abstract state in the *PK*-induced abstract state space Ξ; applying the reconstruction function re-sets a property. The parameter p is used to specify these properties.

Using abstraction keys, we then defined the concept of *abstract skills*

Definition 5. *An* abstract skill $\boldsymbol{K} \doteq (ARM, PK)$ *is comprised of an Abstract Road Map ARM, and a public abstraction key PK. The ARM is a state trace in the PK-induced abstract state space.*

An abstract skill can generally be inferred and cached upon solution of a task — by projecting (a part of) the state trace of the plan execution $\mathfrak{S}(\boldsymbol{E})$ into an abstract domain, using a chosen public abstraction key *PK*. In that case, the parameter p chosen for this projection serves as the *private abstraction key* of that state trace. Later, this skill's *ARM* can be reconstructed (grounded) into a new domain by choosing a new parameter p' to serve as the private key for reconstruction (using the same public key *PK*). As we shall learn, *PK* serves the double purpose of expressing in which scenarios the skill is useful, and providing the required transformation for adjusting it to each such scenario.

3 Practical demonstration: grid world planning domain

To demonstrate the concept and potential of using abstract skills, we choose to examine a discrete planning domain whose state space $\mathcal{S}$ can be illustrated over a two dimensional grid with C columns and R rows, with the origin lying in the bottom-left cell. The transitions are defined from each cell to one of its eight neighbors. This domain of focus allows us to easily visualize and discuss reach-avoid tasks, as shown in Figure 1a. A library of abstract skills is visualized in Figure 1b. Each such skill is represented with its ARM – a sequence of states in a particular topological configuration. As defined before, each abstract skill also specifies a public abstraction key, which allows reconstructing the ARM into a destination domain's state space. For the grid world domain, we define here a new set of geometric abstraction keys, which convey spatial transformations: rotation, spatial translation, scaling (stretching), and a combination of all of those. By reconstructing an abstract skill's *ARM* with such a public key, through an appropriate choice of parameter (i.e., private key) p, we can rotate, stretch, and position the road map anywhere on the grid. These keys are formulated in Appendix A, and visualized in Figure 4.

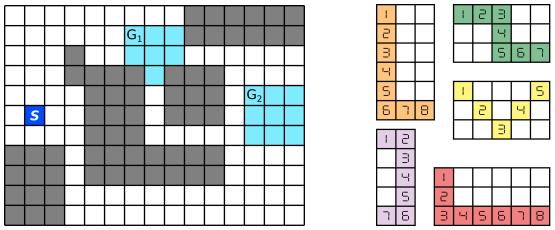


Fig. 1: (a) Exemplary scenario: a grid world planning domain, an initial state (in blue), and a reach-avoid task to solve (in light blue — goal regions, in grey — avoid regions). (b) Abstract road maps (ARMs) representing a library of abstract skills.

To clarify, generally, the public key-induced abstract state space Ξ, which contains the states in the ARM, may be structurally different from $\mathcal{S}$ (e.g., of a lower dimension). We also emphasize that these spaces do not need to be globally isomorphic to allow reconstruction, but only express local invertibility. In this example, the abstract space can be simply thought of as a more compact two-dimensional grid, based on the number and topology of states in the ARM. Further, we only choose this scenario for its ability to intuitively and visually illustrate the concepts presented in the paper. The formulation to follow is relevant to any discrete transition system over a state space – whether this space is discrete, continuous, geometric, or symbolic.

4 Planning with skills as a constraint satisfaction problem

Consider a start state $\boldsymbol{S}_{\text{start}}$, and a task T in domain D, as previously defined. Now, assume that alongside the planning domain and task specifications, one has access to a $\mathcal{L}ibrary$ (database) of abstract skills, each comprised of an abstract road map, and a public abstraction key. Such skills are learned from previous solutions of other tasks, either in this domain or another. We wish to understand if one of the skills in the $\mathcal{L}ibrary$ can be properly transferred to our domain D, and, if so, use it to efficiently find a solution to T. To provide an automatable solution to this problem, we must first formulate it. This problem is inherently hierarchical, and requires overcoming two challenges: first, *skill matching*, and, then, *action recovery*. Let us formulate each of these challenges.

4.1 Skill matching

First, we discuss the problem of skill matching, in which we wish to understand which skills from the $\mathcal{L}ibrary$ are potentially useful for solving our task. Practically, for each skill $\boldsymbol{K}$ that we consider, we want to answer the question: "can we reconstruct its Abstract Road Map ARM into a road map in our state space that is compliant with our current state, task T, and domain D?" Intuitively, for the reach-avoid task in our exemplary scenario, where the abstraction

key allows to perform spatial transformation on states, we wish to ask: "can the abstract skill's ARM be somehow rotated, stretched, and placed on the grid, such that it overlaps with the start state and goal regions, and does not overlap with the regions to avoid?" A visualization of this idea is given in Figures 2a-2b.

To formally answer this question, for a skill $\boldsymbol{K}$, comprised of an abstract road map ARM and a public key $\left(\texttt{project}_p, \texttt{reconst}_p, \mathcal{P}\right)$, we shall determine the existence of a parameter $p \in \mathcal{P}$ that complies to the following constraints. First, we demand that p is a valid choice as a private abstraction key for reconstruction:

$$p \in \mathcal{P}^{ARM}, \text{ i.e., } \forall \xi \in ARM,\ p \in \mathcal{P}^{\xi}. \qquad \texttt{// } p \texttt{ is a private key} \quad (5)$$

Second, we demand that the initial state of the Reconstructed Road Map $RRM \doteq \texttt{reconst}_p\left(ARM\right)$ aligns with our current state:

$$RRM[1] = \boldsymbol{S}_{\text{start}}. \qquad \texttt{// RRM matches current state} \quad (6)$$

Third, we demand that all the states in RRM would indeed be mapped into the domain's state space (as PK may be defined on a state space larger than ours):

$$RRM \subseteq \mathcal{S}. \qquad \texttt{// RRM matches domain} \quad (7)$$

Finally, we demand that RRM satisfies all the state-related task constraints:

$$RRM \text{ respects state-related constraints of } T \qquad \texttt{// RRM matches task} \quad (8)$$

The detailed formulation of this demand surely depends on the type of task. For reach-avoid tasks, all three task constraints are defined on the states of the execution ($\mathfrak{S}\left(\boldsymbol{E}\right)$); hence, we should practically demand that assigning RRM instead of $\mathfrak{S}\left(\boldsymbol{E}\right)$ would satisfy these constrains.

Meaning, we demand that the states in the RRM reach all goal regions:

$$\forall \mathcal{S}_{\text{goal}} \in \mathcal{G}oals,\ \exists \boldsymbol{S} \in RRM, \text{ s.t. } \boldsymbol{S} \in \mathcal{S}_{\text{goal}}, \qquad \texttt{// RRM reaches} \quad (9)$$

that these states avoid the "avoid" regions:

$$RRM \cap \mathcal{S}_{\text{avoid}} = \emptyset, \qquad \texttt{// RRM avoids} \quad (10)$$

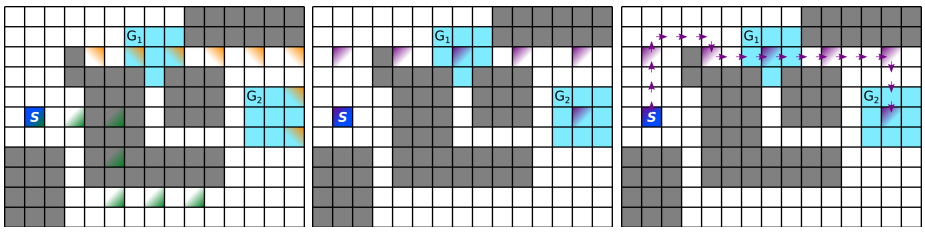
and that the RRM ends in one of the goal regions:

$$\nexists i < \texttt{length}\left(RRM\right), \text{ s.t. } RRM[1\colon i] \text{ satisfies (9).} \texttt{// RRM terminates} \quad (11)$$

Overall, we can formulate the the following constraint satisfaction problem:

$$\begin{aligned}
\texttt{given:}\quad & \text{An abstract skill } \boldsymbol{K} = (ARM, PK) \in \mathcal{L}ibrary, \\
& \text{a task } T, \text{ a domain } D, \text{ and a start state } \boldsymbol{S}_{\text{start}} \\
\texttt{find:}\quad & \text{A parameter } p \in PK.\mathcal{P}, \\
\texttt{s.t.:}\quad & p \textbf{ is a valid private abstraction key} \text{ as in (5) }, \\
& \textbf{RRM matches current state} \text{ as in (6) }, \\
& \textbf{RRM matches domain} \text{ as in (7) }, \\
& \textbf{RRM matches task} \text{ as, e.g., in (9) } \wedge \text{ (10) } \wedge \text{ (11).}
\end{aligned} \quad (12)$$

If we manage to find an abstract skill and a private key p that satisfy these constraints, we say that this is a "successful match".



(a) (b) (c)

Fig. 2: (a) Unsuccessful skill matching: the "orange skill" reconstruction does not match the initial state, and the "green" one does not match the task. (b) Successful skill matching. (c) Action recovery: finding actions to follow the reconstructed road map.

4.2 Action recovery

Let us consider the abstract skill $\boldsymbol{K}$ was matched to our task by finding a suitable private key p for reconstruction. With this key, we are able to reconstruct the abstract road map as $RRM \doteq \texttt{reconst}_p\,(ARM)$. Since the reconstructed road map RRM contains only states, the next challenge is, naturally, to recover a sequence of actions in our domain, which follows the RRM and completes the task. A visualization of this concept is given in Figure 2c.

Formally, we look for a sequence of $actions = (a_1, \ldots, a_n) \in \mathcal{A}^n$ that complies to three constraints. First, we demand that following the actions induces a feasible execution in the domain D, when applied from our current state:

$$\forall a_j \in actions,\ \exists \boldsymbol{S} \in \mathcal{S},\ \text{s.t. } (\boldsymbol{S}_j, a_j, \boldsymbol{S}) \in \mathcal{T},\ \text{where } \boldsymbol{S}_j \doteq a_{j-1} \circ \cdots \circ a_1(\boldsymbol{S}_{\text{start}}). \quad \texttt{// execution is feasible in domain} \tag{13}$$

Second, we demand that following the sequence of actions must "tag" all the states in the road map RRM, in order:

$$\forall i \in \{1, \ldots, \texttt{length}\,(RRM)\},\ \exists t_i \in \{1, \ldots, \texttt{length}\,(actions)\},\ \text{s.t.}$$
$$RRM[i] = a_{t_i} \circ \cdots \circ \ldots a_1(\boldsymbol{S}_{\text{start}}) \quad \wedge \quad i > j \iff t_i > t_j. \quad \texttt{// execution follows RRM} \tag{14}$$

Third, although we verified that the states in RRM satisfy the state-related task constraints, we should still verify that executing the sequence of actions does not violate any of the standing constraints, and completes the task:

$$\boldsymbol{E} = (\boldsymbol{S}_{\text{start}}, a_1\, \boldsymbol{S}_1, a_2, \ldots, \boldsymbol{S}_n) \text{ satisfies standing constraints in } T. \quad \texttt{// execution is valid for task} \tag{15}$$

Again, the detailed formulation of this demand depends on the type of task. For reach-avoid tasks, since following the road map inherently satisfies the "reach" and "terminate" constraints, we should only verify that the execution respects the "avoid" constraints (as in (3)).

Overall, this problem can be written as the constraint satisfaction problem:

$$\begin{aligned}
\texttt{given:}\quad & \text{A reconstructed road map } RRM, \\
& \text{a task } T, \text{ a domain } D, \text{ and a start state } \boldsymbol{S}_{\text{start}}, \\
\texttt{find:}\quad & \text{A finite sequences of } actions = (a_1, \ldots, a_n) \in \mathcal{A}^n, \\
\texttt{s.t.:}\quad & \textbf{execution is feasible in domain} \text{ as in (13)}, \\
& \textbf{execution follows the RRM} \text{ as in (14)}, \\
& \textbf{execution is valid for task} \text{ as, e.g., in (3) .}
\end{aligned} \tag{16}$$

If we manage to find such a satisfactory action sequence, then the task is solved.

4.3 The joint problem: skills as dynamic constraints

By matching a skill before looking for a task-satisfying action sequence, we essentially impose a set of additional constraints on the task, intended to restrict the solution space and, by such, reduce the computational cost of solving the task. For a simple reach-avoid task, this means replacing the basic task constraint "visit $\boldsymbol{S}_{\text{start}}$ and finally $\boldsymbol{S}_{\text{goal}}$", with a more restrictive temporal constraint "visit $RRM[1] \equiv \boldsymbol{S}_{\text{start}}$, then $RRM[2]$, ..., and finally $RRM[end] \equiv \boldsymbol{S}_{\text{goal}}$". These skill-induced constraints are not known until we find a matching abstract skill and reconstruct its ARM; further, these constraints can be revoked and replaced with new constraints, induced by another matching skill. We hence say that abstract skills represent *dynamic constraints* for task planning.

Despite being hierarchically defined, the skill matching and action recovery problems are not independent of each other. Naturally, the constraints in the action recovery problem are based on the matched skill. While in the reverse, a matched skill should be reconsidered if we cannot recover a feasible action sequence that complies to it, and it prevents us from finding (an otherwise possible) solution to the problem. Thus, the problem of planning with skills is formally defined as a constraint satisfaction problem over the joint space of abstract skills $\times$ private abstraction keys $\times$ action sequences, under the union of constraints from both problems. Nevertheless, a naive solution, e.g, by searching this joint state space, or feeding the problem as a whole to a constraint solver, is expected to be inefficient, as it ignores the hierarchical structure of the constraints. To address this issue, we suggest an efficient and hierarchical solution approach to this problem in the following section.

5 Solving the problem: a hierarchical planning algorithm

In the previous section, we formulated the problem of planning with abstract skills as a constraint satisfaction problem. We now wish to explain one way in which it can be solved. To take advantage of the hierarchical nature of the constraints, we propose a bi-level planning algorithm.

At the "first level", we try to match a skill to our problem by looking for a skill for which exists a suitable private key. Using this key, we can reconstruct the

Algorithm 1: Task planning with abstract skills (for general tasks).

Algorithm `planWithAbstractSkills`(*task* T, *domain* D, *start state* $\boldsymbol{S}_{start}$, *library of abstract skills* $\mathcal{L}ibrary$, *generic task planner Planner*)
  forall *skill* $\boldsymbol{K} \in \mathcal{L}ibrary$ **do**
    $RRM \leftarrow$ `matchSkill`($\boldsymbol{K}$, T, D, $\boldsymbol{S}_{start}$)
    if $RRM \neq$ **Null** `// matching skill found`
     then
      actions $\leftarrow$ `recoverActions`(RRM, T, D, *Planner*) `// plan with skill's road map`
      if *actions* $\neq$ **Null then**
        return *actions* `// success!`
      else
        continue `// action recovery failure, skill unfeasible`
    else
      continue `// skill does not match, try the next one`
  end
  return *Planner*.`plan`($\boldsymbol{S}_{\text{start}}$, T, D) `// plan as usual (w/o skills)`

skill's ARM into a RRM and use it as input to the "second level". There, we look for a feasible and valid action sequence, which follows the RRM and completes the task. If no suitable action sequence is found, we declare the matched skill unfeasible. Then, we should report this failure back to the first level and update its solution, e.g., by looking for another skill, or an alternative private key for the matched skill. We would then attempt to re-solve the second layer, and repeat the process as needed, until a complete solution for both layers is found. If at some point no solution can be found at the first layer (e.g., no more skills to examine), we should fall back to standard task planning, without skills; by such, this algorithm is inherently complete. We remind again that this approach does not come to replace standard task planners, and that it actually relies on a given planner to solve the action recovery problem. This high-level approach, which does not depend on the type of task, is summarized in Algorithm 1.

Generally, the "skill matching" and the "action recovery" problems can be solved independently, in any desired method; though, as expressed before, the exact constraint formulation, and, by such, the solution methods, depend on the type of task (and the public abstraction key). Next, we develop practical solution procedures for each of these sub-problems, for reach-avoid tasks.

5.1 Abstract-skill matching for reach-avoid tasks

To match an abstract skill we need to address two sub-problems: (1) choosing a skill for examination, and (2) looking for a private key for its reconstruction. In this work, we can assume that the skills are chosen at random from the library, until a match is found. More generally, we can consider "smarter" skill ordering, e.g., based on the road map length, or an estimated probability of skill feasibility.

For a chosen skill, we should find an appropriate private key for reconstruction of its ARM, by solving the skill matching problem formulated in (12). For reach-avoid tasks, this can be done with a directed search, as we propose here. This search method is summarized as the `matchSkill` procedure in Algorithm 2.

First, we should select one of the goal regions $\mathcal{S}_{\text{goal}} \in \mathcal{G}oals$ to be the termination region. Then, we should start by looking for parameter values $p \in \mathcal{P}^{ARM}$ with which we can reconstruct a road map that starts at $\boldsymbol{S}_{\text{start}}$ and terminates at $\mathcal{S}_{\text{goal}}$, as this, conveniently, does not require reconstructing the entire ARM. In our case, the state and parameter spaces are numerical, and the reconstruction function is geometric; hence, considering the goal region $\mathcal{S}_{\text{goal}}$ is defined by a radius r around a specific state $\boldsymbol{S}_{\text{goal}}$, the compliant parameters can be expressed as the solution space to the following system:

$$\begin{cases} \texttt{reconst}_p\left(ARM[1]\right) = \boldsymbol{S}_{\text{start}}, \\ ||\texttt{reconst}_p\left(ARM[end]\right) - \boldsymbol{S}_{\text{goal}}||_2 \leq r. \end{cases} \tag{17}$$

Solutions can be numerically or symbolically derived. Of course, if $r = 0$, this become simply a system of a equations. If the goal region is comprised of a discrete set of states, we can solve such an equation system for different states from the region and find the superset of solutions. In Appendix A, we provide a concrete derivation of this system and explain how to solve it, considering the geometric abstraction keys we used in our example. This system has an easy-to-find, closed solution, achievable in constant time, which does not depend on the length of the road map. In the more general case, when the constraints are not numerical, the relevant parameters can be derived by intersecting $\mathcal{P}^{ARM}$ with $\mathcal{P}_{\boldsymbol{S}_{\text{start}}}$ and $\mathcal{P}_{\boldsymbol{S}_{\text{goal}}}$. A formulation of this concept appears in our prior work [3].

If we managed to find a non-empty set of parameters that satisfy the start and goal constraints, we should select a parameter p among them and use it to derive a reconstructed road map $RRM = \texttt{reconst}_p\left(ARM\right)$. If RRM satisfies the remaining task and domain constraints, we can return it as a private key, alongside the reconstructed road map RRM, and move on to the second layer. If the RRM does not satisfy the remaining task and domain constraints, we may (i) sequentially examine additional parameters (solutions of the system), until we find a valid private key; or (ii) consider another final region from $\mathcal{G}oals$, and repeat the process; or (iii) simply declare the skill "non-matching" and examine another one from the $\mathcal{L}ibrary$.

5.2 Action recovery using dynamic task decomposition

At the second level of the planning algorithm, we need to recover the action sequence that follows the reconstructed road map RRM and solves the task, as formulated in (16). As previously mentioned, since a matched skill must not violate the task constraints, and since for a matched skill's $RRM \subseteq \mathfrak{S}\left(\boldsymbol{E}\right)$, following the RRM (constraint (14)) often inherently resolves some of the task constraints. This is specifically true for a reach-avoid task, where following the reconstructed

Algorithm 2: Search-based skill matching (for reach-avoid tasks).

```
Procedure matchSkill(skill K, task T, domain D, start state S_start)
    forall S_goal ∈ T.Goals do                    // assume a certain final goal
        p ← K.PK.getGoalCompliantParam(K.ARM, S_start, S_goal)
        if p ≠ Null then
            RRM ← K.PK.reconst_p(ARM)             // reconstruct road map
            if RRM.isInStateSpace(D.S) then
                if RRM satisfies T's "reach" (9) and "avoid" (10)
                 constraints then
                    return RRM                    // private key found, i.e., match!
            continue                              // task/domain constraint failure, consider
             another final goal (or another compliant parameter)
    end
    return Null                                   // skill inapplicable

Procedure PK.getGoalCompliantParam(ARM, start state S_start,
 goal region S_goal)
    params ← solve the symbolic system (17) (or similar)   // params that
     satisfy the start & final goal constraints
    if params ≠ ∅ then
        p ← chooseElementRandomly{params}
        return p
    return Null                                   // start/final goal constraint failure
```

road map of a matched skill inherently satisfies the task's "reach" and "termination" constraints and leaves us to worry only for global "avoid" constraints during action recovery. We identify here that in such cases, where the only standing constraints for action recovery are global constraints, the action recovery problem becomes *decomposable*. This decomposition is dynamically determined according to the skill-induced constraint, which forces the returned action sequence to pass through a series of way points on RRM. Thus, instead of looking for one long action sequence between the first and end states of RRM, we can look for $\texttt{length}(RRM) - 1$ shorter action sequences $actions^i = (a^i_1, \ldots, a^i_{n_i})$, corresponding to the $\texttt{length}(RRM) - 1$ segments of a road map (between each pair of consecutive states).

Essentially, this means that imposing the road map constraint allows us to decompose a reach-avoid task T into smaller reach-avoid sub-tasks T^i, $\forall i \in \{1, \ldots, \texttt{length}(RRM) - 1\}$. The start state for each sub-task T^i is the corresponding road map state $RRM[i]$, the (only) goal state is the next state in the road map $RRM[i+1]$, and the global avoid regions $\mathcal{S}_{\text{avoid}}$ are same as in T. The solution of all these tasks should be sequenced, to provide the overall solution.

Each of these sub-problems can be solved independently with any solver that we would use to solve the original task planning problem, and even different solvers. In particular, we can use the suggested skill-based solver. This means

Algorithm 3: Action recovery with a road map (for reach-avoid tasks).

```
Procedure recoverActions(RRM, task T, domain D,
  generic task planner Planner)
    subplans ← []
    l ← length(RRM)
    for i from 1 to l − 1 do                                  /* in parallel */
        sub_T ← new reach-avoid task      // decompose task to sub-tasks
        sub_T.avoid_region ← T.avoid_region
        sub_T.goal_state ← RRM[i + 1]
        subplan[i] ← Planner.plan(RRM[i], sub_T, D) // solve sub-tasks
        if subplan[i] = Null then
            return Null                        // sub-task unfeasible, abort
    end
    return subplan[1], . . . , subplan[l − 1]
```

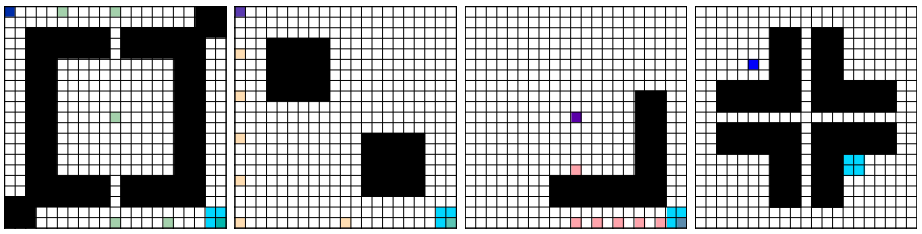
that we can inherently use skills in recursion. Furthermore, while sequential solution of the sub-tasks is already expected to be more efficient than solving the original problem (due to its super-linear complexity), the independent sub-tasks can even solve them in parallel! This action recovery paradigm is summarized as the `recoverActions` procedure in Algorithm 3. Note that when the task is not decomposable (e.g., if it specifies also temporal constraints), we should solve the more general action recovery problem, with the skill-induced constraint. In that case, RRM can be thought of intuitively as a "lead" for the action search.

Handling feasibility failures The final part of the algorithm is concerned with the policy for handling skill "feasibility failures", i.e., "what to do" when an attempt to follow the RRM of a matched skill turns out to be unfeasible. For decomposable reach-avoid tasks, a skill is declared unfeasible if the solution of at least one of the sub-tasks induced by the RRM fails. Due to the limited scope of this paper, in Algorithm 1, we simply consider new skills on such failures. In future work, we plan to thoroughly develop more advanced policies, e.g., allowing to skip unreachable states in the road map. Such policies will allow us to avoid "wasting" the computational effort already invested in partial action recovery.

6 Experimental Results

To put our approach to the test, we examined four reach-avoid tasks in the "grid world" domain $D_{200\times200}$ we introduced in Section 3. Each of these tasks is specified using a single goal region, and "avoid" regions. These tasks represent different search space topologies and obstruction levels. A visualization of these tasks, alongside the chosen start state for each task, is provided in Figures 3a-3d. Note that the drawn grid is of coarser resolution than the actual one considered.

From solution of previous tasks, we derived and cached a library of abstract skills, using the geometric keys we mentioned in Section 3 and formulated in



(a) Task 1. **(b)** Task 2. **(c)** Task 3. **(d)** Task 4.

Fig. 3: The reach-avoid tasks we considered in the grid world domain; initial state is in blue, goal regions are in light blue. If a skill from a library (Figure 1b) matched the problem, its reconstructed road map RRM is also drawn (in respective color).

Appendix A. Note that formulation of this process is provided separately in a follow-up to this work. The ARMs of the skills in the $\mathcal{L}ibrary$, each of which is derived from a solution of a single past task, are visualized in Figure 1b. Such tasks can generally be solved in a different domain than the one they will be used in, including grid worlds of different sizes and with different action spaces.

To show the versatility of the approach, we considered two standard search-based planners: BFS and A*. Each of these planners hold particular benefits: while A* typically achieves better computation times, by taking advantage of a distance-to-goal heuristic, BFS is also useful in domains in which we do not have a distance measure. Then, considering each planner, each of our tasks was solved in two ways: (i) by naively using the task solver; and (ii) by using the given solver with exploitation of abstract skills, as suggested in Algorithm 1. For comparative reasons, for the second option, we solved the action recovery twice – with basic sequential solution of the sub-tasks, and by allowing their parallel solution (as explained in Section 5.2). The computation times for the different task solutions are presented in Table 1. The algorithms were implemented in Python with no special optimization. The experiments were conducted on a desktop computer, equipped with an Intel Core i7-6700K CPU and 32GB of RAM.

For tasks 1-3, we managed to successfully match a skill to the problem. The reconstructed road map RRM for each of the matched skills, leading from the start to the goal, is shown in Figures 3a-3c. Evidently, matching a skill allowed us to significantly reduce the computation time in comparison to standard usage of the planners. This reduction was a result of the task decomposition, even without considering sub-task parallelization, which can help reduce the cost of the problem even more. While the improvement ratio depends on many factors, including the planner, task, and topology and length of the road map, these results motivate our claim for the benefit of planning with abstract skills. We recall that theoretical justification to this improvement is given in Appendix B. For task 4, no matching skill was found, leading us to deploy the task planner naively. Nonetheless, the cost of the unsuccessful matching attempt was negligible in comparison to the planning cost. Tasks 3 also demonstrates the flexibility of the state-based representation against predefined macro-actions, as the road map was not compromised by the "obstacle" — we dynamically planned around it.

Table 1: Planner efficiency w/o and w/ skills (w/o and w/ sub-task parallelization).

		BFS			A*		
		Baseline	w/ skills	w/ skills (par)	Baseline	w/ skills	w/ skills (par)
Task 1	Skill matching (sec)	–	0.001	0.001	–	0.001	0.001
	Action recovery (sec)	57.514	36.899	24.343	2.814	0.2610	0.120
	# searched nodes	6337	9219	9219	890	511	511
Task 2	Skill matching (sec)	–	0.0005	0.0005	–	0.0005	0.0005
	Action recovery (sec)	114.225	32.234	20.155	0.958	0.119	0.042
	# searched nodes	8848	9811	9811	467	415	415
Task 3	Skill matching (sec)	–	0.0008	0.0008	–	0.0008	0.0008
	Action recovery (sec)	120.315	18.492	10.275	3.599	0.110	0.070
	# searched nodes	9199	6171	6171	1344	337	337
Task 4	Skill matching (sec)	–	0.0007	0.0007	–	0.0007	0.0007
	Action recovery (sec)	75.341	75.341	75.341	3.569	3.569	3.569
	# searched nodes	7286	7286	7286	1270	1270	1270

Table 1 also presents the number of graph node expanded by the search algorithms. In most cases, following the road map caused us to expand less graph nodes (grid cells) during the search and, by such, led to a reduction in its cost. Yet, curiously, following the road map sometimes caused us to expand *more* graph nodes and still led to a reduction in the search cost. This is a clear expression of the benefit of the problem decomposition. The cost of graph node expansion is not constant and also grows as the search progresses, as we must repeatedly check if new nodes were previously visited, against a growing list of visited nodes. Thus, although the total number of expanded nodes grew, since it was divided among the solutions of multiple sub-tasks, the total cost of operation was smaller. In our experiments, this increase in the number of expanded nodes only happened with the BFS planner, which performs an exploratory search. In this case, the increase was caused by re-exploring regions of the domain in the solution of multiple sub-tasks. This can potentially be mitigated via information sharing among the sub-task solutions, as we plan to examine in our future work.

7 Conclusion

This paper came to explain how to incorporate abstract skills into task planners. These abstract skills, which were introduced in our previous work, are represented as abstract state road maps, derived from previous task solutions; these road maps can be automatically adjusted and deployed to guide the solution of new tasks (even in new domains). We began by formulating the problem as a constraint satisfaction problem. We then presented a high-level hierarchical solution algorithm, which allows us to use standard task planners more efficiently, using a precursory "skill matching" process. Finally, we proved the potential benefits of this approach in the solution of a series of reach-avoid tasks. There, considering two different planners, we managed to significantly reduce the plan computation time. Future work will explore several extensions of this promising approach. These include better handling of skill feasibility failures, compositional planning from skills, application in more complex task planning domains (e.g., PDDL-based), and application in continuous planning domains.

References

1. Karpas, E., Magazzeni, D.: Automated planning for robotics. Annual Review of Control, Robotics, and Autonomous Systems 3(1), 417–439 (2020), https://doi.org/10.1146/annurev-control-082619-100135
2. Garrett, C.R., Chitnis, R., Holladay, R., Kim, B., Silver, T., Kaelbling, L.P., Lozano-Pérez, T.: Integrated task and motion planning. Annual Review of Control, Robotics, and Autonomous Systems 4(1), 265–293 (2021), https://doi.org/10.1146/annurev-control-091420-084139
3. Elimelech, K., Kavraki, L.E., Vardi, M.Y.: Automatic cross-domain task plan transfer by caching abstract skills. In: Workshop on the Algorithmic Foundations of Robotics (WAFR) (Jun 2022)
4. Shridhar, M., Manuelli, L., Fox, D.: CLIPort: What and where pathways for robotic manipulation. In: Conference on Robot Learning (CoRL) (Nov 2021)
5. Wang, Z., Garrett, C.R., Kaelbling, L.P., Lozano-Pérez, T.: Learning compositional models of robot skills for task and motion planning. The International Journal of Robotics Research (IJRR) 40(6-7), 866–894 (2021)
6. Ghallab, M., Nau, D., Traverso, P.: Automated planning and acting. Cambridge University Press (2016)
7. Hoffmann, J., Nebel, B.: The ff planning system: Fast plan generation through heuristic search. Journal of Artificial Intelligence Research (JAIR) 14, 253–302 (2001)
8. Helmert, M.: The fast downward planning system. Journal of Artificial Intelligence Research 26, 191–246 (2006)
9. Rintanen, J.: Madagascar: Scalable planning with SAT. Proceedings of the 8th International Planning Competition (IPC-2014) 21, 1–5 (2014)
10. Cashmore, M., Fox, M., Long, D., Magazzeni, D.: A compilation of the full PDDL+ language into SMT. In: International Conference on Automated Planning and Scheduling (ICAPS). pp. 79–87 (Jun 2016)
11. Georgievski, I., Aiello, M.: HTN planning: Overview, comparison, and beyond. Artificial Intelligence 222, 124–156 (2015)
12. Shivashankar, V., Kuter, U., Nau, D., Alford, R.: A hierarchical goal-based formalism and algorithm for single-agent planning. In: International Conference on Autonomous Agents and Multiagent Systems (AAMAS). pp. 981–988 (Jun 2012)
13. Plaku, E., Kavraki, L.E., Vardi, M.Y.: Motion planning with dynamics by a synergistic combination of layers of planning. IEEE Transactions on Robotics (T-RO) 26(3), 469–482 (2010)
14. Dantam, N.T., Kingston, Z.K., Chaudhuri, S., Kavraki, L.E.: Incremental task and motion planning: A constraint-based approach. In: Robotics: Science and systems (R:SS) (Jul 2016)
15. Kaelbling, L.P., Lozano-Pérez, T.: Hierarchical task and motion planning in the now. In: IEEE International Conference on Robotics and Automation (ICRA). pp. 1470–1477 (May 2011)
16. Kavraki, L., Svestka, P., Latombe, J.C., Overmars, M.: Probabilistic roadmaps for path planning in high-dimensional configuration spaces. IEEE Trans. Robot. Automat. 12(4), 566–580 (1996)
17. Chamzas, C., Cullen, A., Unhelkar, V., E. Kavraki, L.: Learning to retrieve relevant experiences for motion planning. In: IEEE International Conference on Robotics and Automation (ICRA) (May 2022)

A Geometric abstraction keys for spatial transformation

Let us define several new abstraction keys, in which the projection and reconstruction functions represent spatial transformations on two-dimensional geometric states. Note that, in contrast to keys presented in our prior work [3], these convey no dimensionality reduction nor a change in the state structure. Each of the public abstraction keys to follow allows us to perform a specific type of spatial transformation on states: rotation, scaling (stretching), and spatial translation. By applying the reconstruction function on the abstract states in an ARM, with a choice of parameter p, we can set the value of the respective property (angle/scale/position) of the RRM. This is demonstrated in Figure 4.

First, we defined a *rotation key*:

$$\texttt{RK}.\mathcal{P} \doteq \{\theta \in \mathbb{R} \mid 0 \leq \theta < 2\pi\} \tag{18}$$

$$\texttt{RK.project}_{\theta}\colon \begin{pmatrix} col \\ row \end{pmatrix} \mapsto \begin{pmatrix} \cos -\theta & -\sin -\theta \\ \sin -\theta & \cos -\theta \end{pmatrix} \cdot \begin{pmatrix} col \\ row \end{pmatrix}, \tag{19}$$

$$\texttt{RK.reconst}_{\theta}\colon \begin{pmatrix} col \\ row \end{pmatrix} \mapsto \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \cdot \begin{pmatrix} col \\ row \end{pmatrix} \tag{20}$$

Second, we define a spatial *translation key*:

$$\texttt{TK}.\mathcal{P} \doteq \left\{(x, y) \in \mathbb{R}^2\right\}, \tag{21}$$

$$\texttt{TK.project}_{x,y}\colon \begin{pmatrix} col \\ row \end{pmatrix} \mapsto \begin{pmatrix} col - x \\ row - y \end{pmatrix}, \tag{22}$$

$$\texttt{TK.reconst}_{x,y}\colon \begin{pmatrix} col \\ row \end{pmatrix} \mapsto \begin{pmatrix} col + x \\ row + y \end{pmatrix} \tag{23}$$

Third, we define a *scaling key*:

$$\texttt{SK}.\mathcal{P} \doteq \left\{(\alpha, \beta) \in \mathbb{R}^2 \mid \alpha, \beta \neq 0\right\}, \tag{24}$$

$$\texttt{SK.project}_{\alpha,\beta}\colon \begin{pmatrix} col \\ row \end{pmatrix} \mapsto \begin{pmatrix} \frac{1}{\alpha} \cdot col \\ \frac{1}{\beta} \cdot row \end{pmatrix} \tag{25}$$

$$\texttt{SK.reconst}_{\alpha,\beta}\colon \begin{pmatrix} col \\ row \end{pmatrix} \mapsto \begin{pmatrix} \alpha \cdot col \\ \beta \cdot row \end{pmatrix} \tag{26}$$

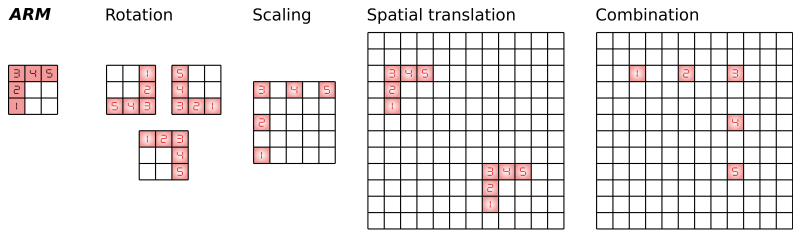


Fig. 4: Abstraction keys for geometric transformation of an abstract road map.

Lastly, since the former keys transform states over the same state space (two dimensional positions), these keys can inherently be composed. Thus, we can define the *combination key*, which can represent any combination of rotation, translation, and scaling:

$$\texttt{CK}.\mathcal{P} \doteq \{p \doteq (\theta, x, y, \alpha, \beta) \mid \theta \in \texttt{RK}.\mathcal{P},\ (x, y) \in \texttt{TK}.\mathcal{P},\ (\alpha, \beta) \in \texttt{SK}.\mathcal{P}\} \tag{27}$$

$$\texttt{CK.project}_p \doteq \texttt{SK.project}_{\alpha,\beta} \circ \texttt{TK.project}_{x,y} \circ \texttt{RK.project}_{\theta}, \tag{28}$$

$$\texttt{CK.reconst}_p \doteq \texttt{RK.reconst}_{\theta} \circ \texttt{TK.reconst}_{x,y} \circ \texttt{SK.reconst}_{\alpha,\beta} \tag{29}$$

Note that these keys are not tied to the grid world domain. Specifically, they can also be used for transferring skills to/from continuous state spaces; we therefore also allow non-integer values in their parameter spaces. The keys can also trivially be adapted to convey the respective transformations in higher-dimensional state spaces.

A.1 Skill matching

As mentioned in Section 5.1, when considering these keys, we can often solve the skill matching problem easily and symbolically. Let us consider a start state $\boldsymbol{S}_{\text{start}}$, a task with a desired end goal state $\boldsymbol{S}_{\text{goal}}$, and an abstract skill $\boldsymbol{K}$, represented by the abstract road map ARM, and the "combination" public abstraction key. As previously described in (17), the parameter values p for this public key that satisfy the "start state" and "final goal" constraints can be found by solving the following system of equations:

$$\begin{cases} \texttt{CK.reconst}_p(ARM[1]) = \boldsymbol{S}_{\text{start}}, \\ \texttt{CK.reconst}_p(ARM[end]) = \boldsymbol{S}_{\text{goal}}, \end{cases} \tag{30}$$

which is trivially equivalent to

$$\begin{cases} \texttt{TK.reconst}_{x,y} \circ \texttt{SK.reconst}_{\alpha,\beta}(ARM[1]) = \texttt{RK.reconst}_{-\theta}(\boldsymbol{S}_{\text{start}}), \\ \texttt{TK.reconst}_{x,y} \circ \texttt{SK.reconst}_{\alpha,\beta}(ARM[end]) = \texttt{RK.reconst}_{-\theta}(\boldsymbol{S}_{\text{goal}}). \end{cases} \tag{31}$$

This system can be easily solved by standard symbolic or numerical solvers. In fact, in our grid world case, where the start and goal states are integer vectors, it can also be solved using a standard linear solver by pre-resolving the non-linear rotation. Specifically, to make sure the reconstructed states remain on the grid, we can limit the rotation angle to one of four possible options: $\theta \in \left\{0, \frac{\pi}{2}, \pi, \frac{3\pi}{2}\right\}$. For each possible rotation, we are left to consider an easily-solvable linear system (with four variables and four equations), only containing the translation and scaling. The final solution would be the union of solutions from the four systems.

B Complexity analysis

With the skill-based planning approach, we replace one constraint satisfaction problem with a more complicated one. On one hand, we add variables to the problem (the abstract skill and private key) and, by such, potentially increase the computational cost of the solution ("increase the size of the search space"). On the other hand, a skill imposes new constraints on the actions, which potentially reduces the cost of the solution ("reduces the size of the search space"). Since the intended purpose of the skill-based approach is to reduce the planning cost, it is important to justify and explain when this equilibrium becomes favorable.

Let $\texttt{planEffort}(n)$ represent the maximal/expected computational effort of finding a path between states of distance n (actions). By first matching a skill with an ARM of $k+1$ (equally-spaced) states and decomposing this problem, we can potentially reduce this computational effort to $k \cdot \texttt{planEffort}\left(\frac{n}{k}\right)$. Of course, following a skill can lead to a longer plan, which would imply a higher planning cost, i.e., $k \cdot \texttt{planEffort}\left(\phi \cdot \frac{n}{k}\right)$, where ϕ is the factor by which the path length increases by following the road map.

Let L mark the number of skills examined until finding the applicable skill, and $\texttt{matchEffort}(Skill)$ — the cost of skill matching. Since we can also potentially pay a cost for partially following the road map of an unfeasible skill, we use $\texttt{subTasksSolved}(Skill)$ to mark the number of sub-tasks solved until the skill is declared un/feasible; with our algorithm, the value of this function is $\in \{1, \ldots, k-1\}$ if the skill is unfeasible, and exactly k for the successful skill.

Thus, the theoretical computational cost of planning with skills is:

$$\sum_{i=1}^{L} \left(\texttt{matchEffort}(Skill_i) + \texttt{subTasksSolved}(Skill_i) \cdot \texttt{planEffort}\left(\phi_i \cdot \frac{n}{k_i}\right) \right). \tag{32}$$

If we assume that every applicable skill is also feasible, and that we are indeed able to solve sub-tasks in parallel, then this expression reduces to:

$$\sum_{i=1}^{L} \left(\texttt{matchEffort}(Skill_i) \right) + \texttt{planEffort}\left(\phi_i \cdot \frac{n}{k_i}\right). \tag{33}$$

Since we assume (i) the matching effort to be minor in comparison to the planning effort, as seen in our experiments; (ii) the planning effort for task planners to grow *super-linearly* with n; and (iii) that using the skill does not increase the returned plan length in a factor of k_i; this means a reduction in computation effort, as intended. In fact, the higher the growth rate of $\texttt{planEffort}(n)$, in respect to n, the more preferable planning with skills becomes.

For example, when action recovery is done with a BFS graph search, and considering action set of size m, this would mean reducing the action recovery effort from m^n to $m^{\frac{\phi \cdot n}{k}}$ (measured by the number of expanded graph nodes).

Of course, the previous discussion implicitly assumes that we can indeed find an applicable skill to solve the problem. Otherwise, we can inherently fall back to a standard planning, and only pay for unsuccessful matching.