Learning to Plan Hierarchically from Curriculum

Philippe Morere[1], Lionel Ott[1] and Fabio Ramos[1,2]

***Abstract*—We present a framework for learning to plan hierarchically in domains with unknown dynamics. We enhance planning performance by exploiting problem structure in several ways: (i) We simplify the search over plans by leveraging knowledge of skill objectives, (ii) Shorter plans are generated by enforcing aggressively hierarchical planning, (iii) We learn transition dynamics with sparse local models for better generalisation. Our framework decomposes transition dynamics into skill effects and success conditions, which allows fast planning by reasoning on effects, while learning conditions from interactions with the world. We propose a simple method for learning new abstract skills, using successful trajectories stemming from completing the goals of a curriculum. Learned skills are then refined to leverage other abstract skills and enhance subsequent planning. We show that both conditions and abstract skills can be learned simultaneously while planning, even in stochastic domains. Our method is validated in experiments of increasing complexity, with up to 2^{100} states, showing superior planning to classic non-hierarchical planners or reinforcement learning methods. Applicability to real-world problems is demonstrated in a simulation-to-real transfer experiment on a robotic manipulator.**

I. INTRODUCTION

SAMPLE efficiency is of utmost importance when robots need to learn how to act using interactions with their environment. Reinforcement learning (RL) methods are especially affected by this issue, and often require very large amounts of data before learning decent policies. This is unacceptable in many robotics scenarios where gathering data is expensive or time consuming. This problem can largely be addressed by taking advantage of structure in states, actions and environment transitions. However, learning this structure is very challenging.

Many robotic planning problems feature a hierarchical task structure; e.g. graph in Figure 1. Taking advantage of problem hierarchy by planning at abstract levels leads to multiple advantages over classic planners or RL techniques which operate at a single level only. Indeed, plans constructed from abstract actions, or *skills*, are typically much shorter as they can reuse previous skills. Hierarchical plans can also be lazy, ie. skills are only decomposed into lower-level skills when needed, allowing plans to be updated with the latest environment information. Furthermore, planning with high-level skills is often easier, as most of the environment stochasticity is absorbed into lower-level skills. These principles greatly improve sample efficiency and planning times compared to RL and classic planners, as shown in experiments.

When deploying robots in the real world, it is common to use a set of pre-defined policies to perform basic actions instead of using raw torque commands for example. These policies can either be programmed by experts or learned using RL techniques for example, to perform a specific task. Because the effect of these pre-defined policies (or *primitive skills*) is usually well-defined, it can be leveraged by planning algorithms to enhance their efficiency. However, in many cases, the conditions under which primitive skills succeed may be unknown, and must be learned from interactions with the environment. Knowledge of skill effects enables directed planning towards a given goal, and makes inefficient random exploration unnecessary. Our experiments confirm this, showing directed planning is orders of magnitude faster than Monte-Carlo Tree Search [5] and RRT [17].

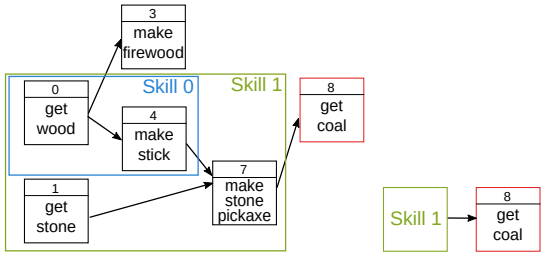


Fig. 1: Left: learning abstract skills (blue and green rectangles) from a sequence of primitive skills (boxes), after planning for a given goal (red box). Right: resulting abstracted trajectory for goal *get coal*.

We develop a framework for learning to plan hierarchically in domains with unknown dynamics. This framework aims to provide planning algorithms with more structure, and is based on several key insights. Firstly, we consider the problem of planning with primitive skills of known effects. Skill effects can be exploited by planning algorithms to determine the best available skill for a given goal. Using a *curriculum*, a sequence of goals with increasing complexity, new useful skills are incrementally made available to learn. Secondly, interesting and challenging problems often feature a hierarchical component which flat planning algorithms struggle to cope with. Our method has a strong emphasis on hierarchical planning, and encourages learned skills to reuse other skills. This results in efficient plans with very few high-level steps, as shown in Figure 1. Lastly, in structured and sparse state spaces (ie. state dimensions are mostly independent of each other), skill effects, skill conditions and transition dynamics also become sparse. In such environments, transition dynamics or skill conditions can then be learned using very simple local models, allowing scalability to problems with larger state spaces. Using local models for transitions dynamics also improves generalisation to unseen states as most state dimensions become irrelevant

This paper was published in IEEE Robotics and Automation Letters (2019).
*Correspondence to philippe.morere@sydney.edu.au
[1]The University of Sydney
[2]NVIDIA

to predict specific transitions.

Our contributions are the following:

- We present a framework suited for hierarchical planning, in which transition dynamics are decomposed into skill effects and conditions. This framework allows for reasoning on skill effects, while learning their conditions from interactions with the world.
- We propose a simple method for learning new abstract skills, using successful trajectories stemming from completing the goals of a curriculum. Skills are then refined by reasoning on the effects and conditions of previous successful trajectories.
- We extend the problem to the case of unknown transition dynamics, ie. when skill success conditions are unknown. We present a method for learning conditions from interactions with the real world, and show that conditions can be learned while planning, learning and refining skills, even in stochastic environments.
- We evaluate our approach performance on simulated problems of growing complexity against established planners and RL methods. We then demonstrate the applicability to real-world problems with a simulation-to-real transfer experiment on a robotic manipulator.

II. RELATED WORK

The idea of planning by reasoning about action effects and conditions has been long studied, and is a core principle behind classic planners like STRIPS [8]. These planners typically require all action effects and conditions to be specified, and often produce sequential plans, thus lacking the advantages of hierarchical planning. Hierarchical Task Networks (HTN) [24] provide a hierarchical alternative by producing plans given skills and their dependencies. HTN planners, and their extension to AND/OR graphs [6], have been successfully applied to rich and complex problems such as robot soccer [26], multi-agent assembly domains [15], and human-robot collaborative assembly [14]. However, these planners require dynamics and hierarchy to be specified.

Learning the skill hierarchy while planning with HTN is proposed in [25], but relies on expert demonstrations, and transition dynamics still need to be specified. More recent work requires a graphical task representation to automatically construct HTMs [12]. These techniques necessitate considerable expert knowledge. By opposition, our work aims to learn both dynamics and hierarchy directly from interactions with the world.

Another approach to plan in environments with unknown dynamics is reinforcement learning (RL). In model based RL [31], environment dynamics are learned from observed transitions, and the learned model can then be used in conjunction with a planning algorithm such as Monte-Carlo Tree Search [5], DESPOT [37] or RRT [17], [10] to find an optimal plan. Although powerful, these methods are not robust against small errors in learned transition dynamics, as these compound when planning over long horizons. By opposition, hierarchical planners typically plan on much shorter horizons, which mitigates this problem.

The idea of hierarchical learning and planning was also studied in RL [32]. In hierarchical RL, policies are composed sub-policies called *options*, which can be learned from interactions with the environment. Symbolic planning and RL were combined in [11] where a STRIPS planner shapes the RL agent's reward function, to achieve high-level reasoning and fast low-level reactions. The HASSLE algorithm [2] learns sub-skills by identifying clusters of raw input data, but is limited to a small and predefined number of hierarchical levels. Unlike our work, RL methods learn policies using a fixed reward signal, and thus often can't handle multiple or changing goals.

When dealing with high dimensional state and action spaces, planning and RL performance degrades quickly. Relational RL [35] is a subset of RL concerned with upgrading state and action representations with objects and their relations. These MDP variants can deal with very high dimensional and structured state spaces, by reasoning over objects. The transition dynamics can be compactly represented using dynamic Bayesian networks or decision trees [3]. Our work addresses high dimensional state and action spaces with sparsity assumption, although it could be extended to using relational RL concepts.

Planning with policies or skills instead of actions was investigated in [16], where skills are black-box controllers. The problem is extended to the case of parametrised skills by [1], which improves the range of available robot behaviours at the expense of requiring planning over both skill and parameters. Lastly, actions are replaced with predefined *algorithms* in [33], yielding advantages over planning with action directly.

This work combines the advantages of hierarchical planners such as HTN with the ability to learn transition dynamics from interactions with the environment, as achieved by RL algorithms. The presented framework leverages sparse state representations, allowing it to scale to problems with very large state spaces.

III. PROBLEM DEFINITION

We begin by defining the family of problems addressed by our method, and present a general framework for hierarchical planning in which abstract skills and primitive skill conditions can be learned from interactions with the environment.

A. Markov environment

We consider an agent interacting with a Markov environment. The environment is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{G}, T)$ composed of states $s \in \mathcal{S}$, skills $a \in \mathcal{A}$, goals $g \in \mathcal{G}$, and a transition function T. States s are vectors representing the current full state of the environment, and the environment's Markov property restricts transitions to a new state s' to only depend on the previous state s and skill a, ie. $s' = T(s, a)$. This property greatly simplifies planning, as maintaining the history of previous states is not required. Agents are given a goal g and must find a sequence of skills (called *plan*) to reach g, starting from an initial state s_0. Solving the planning problem reduces to finding plans π that reach *any* feasible goal given to the agent.

B. Framework

We extend the Markov environment definition to incorporate skill knowledge at its core. The set of initial skills given to the agent is denoted $\mathcal{A}^0$. Skills in $\mathcal{A}^0$ are called *primitive* skills, as opposition to *learned* skills which are learned from successful trajectories at a later stage.

1) Skills: A skill a (primitive or learned) is composed of the following elements:

- An effect e describes the intended state changes resulting from executing the skill. For example, an effect can be a set of state dimensions to be changed and their resulting values.
- A condition c characterises state requirements for a skill to succeed. When these conditions are fulfilled, the skill can be successfully executed, and effect e is applied to the new state. If these conditions are not met, skill execution fails and the effect e is not applied (although e could still be observed due to stochasticity).
- A plan or policy π. In the case of primitive skills, π is a predefined policy or a raw action, which can be directly executed in the environment. For learned skills, π is a sequence of skills (learned and/or primitive) which when executed achieves the desired effect e.
- A side effect e^+ describes additional non-intended state changes resulting from executing π. Primitive skills often have few or no side effects, whereas learned skills with complicated plans may trigger multiple side effects upon execution.

2) Transitions: Environment transitions are decomposed in a similar way to skill effects and conditions. The transition function T is modelled as the composition of an effect function f and a noise function g:

$$s' = T(s,a) = g(f(s,a)), \tag{1}$$

$$\text{where } f(s,a) = \begin{cases} s \oplus e \text{ if } c \text{ matches } s, \\ s \text{ otherwise,} \end{cases} \tag{2}$$

$$\text{and } g(s) = s \oplus \epsilon. \tag{3}$$

Here e and c refer to the effect and condition of skill a respectively, and $s \oplus e$ indicates that effect e is applied to state s. Function g reflects potentially stochastic dynamics by applying noise effect ϵ to the resulting state of $f(s,a)$. Skill conditions can either be given to the agent, in which case the problem reduces to strict planning, or they may be unknown (which translates to unknown transition dynamics) and need to be learned from interactions with the environment.

3) Sparse states: Keeping scalability to larger problem in mind, we enforce sparse state representations; eg. as binary vectors of features. Sparse state representations make skill effects and conditions compact, as each skill only operates on a few dimensions of the state space. This sparse requirement results in easier condition learning, and allows reasoning and planning even in high dimensional state spaces. Although this is not the scope of this paper, learning a disentangled state representation could be achieved using auto-encoders [34].

The presented decomposition of skills and transition dynamics into conditions and effects enables much richer reasoning over action intentions and goals. The method presented in the next section leverages this principle to learn skills, conditions, and plan hierarchically.

IV. METHOD

We present a method for learning to plan hierarchically[1], following the framework defined in Section III-B. The proposed technique plans backwards from desired goal to starting state, using a collection of skills $\mathcal{A}$. Skills are composed with one another by matching the effects of a skill with the conditions of the next. Plans successfully achieving desired goals are abstracted into new skills, then added to $\mathcal{A}$. New skills are refined to reuse other skills of $\mathcal{A}$, so as to ensure planning results in short and highly hierarchical plans. This refinement process relies on reasoning about skill effects and conditions; in problems with unknown dynamics, skill conditions can be learned from interactions with the environment, using a probabilistic model.

A. Planning with hierarchical skills

The presented planner takes advantage of the rich collection of skills $\mathcal{A}$ and their hierarchical nature. Planning backwards, it recursively finds skills that satisfy the conditions of its current goal. Starting from goal g, the planner finds the skill $a \in \mathcal{A}$ with effect closest to g, sets a's success conditions as its new goal, and plans again. If no skill exactly matches g, the returned plan – composed of the skill whose effect is closest to g – approaches the vicinity of g. This helps subsequent planning (or random skills) to reach g. The hierarchical goal-regression planner is detailed in Algorithm 1.

This framework promotes aggressive hierarchical planning and leads to very fast planning, as the maximum recursion rec_{max} can be low ($rec_{max} = 3$ was sufficient in experiments). Also, this hierarchical planner returns lazy plans; plans are never reduced to sequences of primitive skills. Thus executing the first primitive skill of a plan only requires expanding its first element. This property allows plans to be adapted automatically by expanding higher-level skills only when the newest state is available, hence making plans more robust to stochastic transitions and unforeseen side effects.

B. Skill learning

Planning performance greatly depends on the quality and diversity of skills in $\mathcal{A}$; augmenting $\mathcal{A}$ using successful plans is essential. Skill learning only requires a successful trajectory and an intended goal, and thus equally applies to previous successful plans and expert demonstrations. Skills can also be learned from trajectories executed by another agent or robot, although directly transferring the collection of learned skills $\mathcal{A} \backslash \mathcal{A}^0$ is easier and faster.

Learning a new skill amounts to finding a plan π achieving a given effect e. Once a successful skill sequence $\{a_0, a_1, .., a_n\}$ was obtained for a given goal g, the execution success of each skill a_i can trivially be checked using their effect information.

[1]Python code available at https://github.com/PhilippeMorere/learning-to-plan-hierarchically

Function *plan(s, g, $rec = 0$)*
  if $rec > rec_{max}$ *or g statisfied in s* **then**
    return *Random skill a from* $\mathcal{A}$.
  end
  $a \leftarrow$ skill from $\mathcal{A}$ valid in s, with effect closest to g.
  if *a succeeds in state s* **then**
    return a.
  else
    return $\{plan(s, condition(a), rec + 1), a\}$.
  end

Algorithm 1: Goal-regression planner

Data: Trajectory τ, desired effect e.
Result: Refined plan π.
$G \leftarrow$ build directed acyclic graph from τ.
for *node* $n \in G$ **do**
  $a \leftarrow$ shortest skill from $\mathcal{A} \backslash \mathcal{A}^0$ with
    effect $\supset n$.effect and condition $\subset n$.condition.
  if $a \neq \emptyset$ **then**
    $l \leftarrow$ nodes of G with effect $\subset a$.effect.
    Replace nodes l in G with new node from a.
  end
end
$\pi \leftarrow$ skill sequence in G ending with leaf effect e.

Algorithm 2: Skill refinement

The list of successful skills forms a *trajectory* τ. Using the initial state s_0, g and τ, a new skill can be created with initial plan τ and intended effect e, computed as going from s_0 to g. Skill conditions c and side effects e^+ are computed by composing the conditions and side effects of skills from τ. Learned skills are added to $\mathcal{A}$ and made available to the planner. Note that this procedure only describes how to initialise a new skill using a successful trajectory. Skills should then be refined to find better plans.

C. Skill refinement

After a skill is initialised using a successful trajectory, refining it results in finding a better plan π which reuses other high-level skills; see Figure 1. A given trajectory τ can be converted to a directed acyclic graph (DAG) G in which nodes are skills and τ_0 is the root. Nodes are linked such that node n's skill condition require all skills in $parents(n)$ to be executed first (ie. the effect of $parents(n)$ are included in node n's skill conditions). Once graph G is constructed, groups of nodes are replaced by their corresponding higher-level skill from $\mathcal{A} \backslash \mathcal{A}^0$, when their effects and conditions match. After all possible replacements were made, graph G is often composed of few – and mostly high-level – skills. Given the trajectory's desired effect e, a refined plan π can be constructed from G by including all parents of leaf node with effect e. The refinement process is described in Algorithm 2.

If primitive skill conditions are known, skill refining can be executed directly after initialisation. Conversely, if conditions need to be learned from interaction data, skills may only be refined once the conditions of their successful trajectory sub-skills are confident enough.

D. Condition learning

Both planning and skill refinement require skill success condition knowledge to reason about trajectories. In problems with *unknown* dynamics, the conditions of all primitive skills in $\mathcal{A}^0$ need to be learned. Note that learned skills are composed of lower-level skills and therefore it is not necessary to learn their conditions, as they can be computed by inspecting the skill's plan.

Every interaction with the environment generates a transition tuple (s, a, s'). Skill success is easily assessed by comparing the effect e of a to the observed effect between states s and s'. After a given primitive skill a is executed m times, starting states and skill successes form a dataset $\mathcal{D}_a = \{s_i, success_i\}_{i=1}^m$ from which conditions can be learned. Learning a discriminative model $p(success|s)$ of skill success given a starting state is not required to use learned conditions for planning. Rather, learning a generative model $p(s|success = True)$ is more interesting, as it allows the goal-regression planner to use states sampled from the generative model as goals.

Conditions are learned using a two stage process. First, we identify the smallest subset of sufficient dimensions to predict skill success given a state. This is consistent with the framework's requirements of learning sparse skill success conditions. This step is implemented using orthogonal matching pursuit (OMP) [18], [28]. When applied to $\mathcal{D}_a$, OMP returns coefficients for each dimension of the state space. High coefficients are associated with state dimensions greatly impacting skill success, whereas coefficients close to zero reflect state dimensions with little or no impact. Using the non-zero coefficients returned by OMP, a sparse version of $\mathcal{D}_a$ can be constructed, denoted $\bar{\mathcal{D}}_a$. The second stage of the learning process involves learning a generative model using $\bar{\mathcal{D}}_a$. Any generative model can be used, and we find a Gaussian mixture model (GMM) [27] is sufficient to learn simple conditions. Increasing the number of components used by GMM allows learning more complicated skill conditions.

E. Learning from curriculum

Learning good skills is important to reduce planning complexity. Indeed, learning too many skills may result in an unnecessary large search procedure over available skills, whereas learning too few skills leads to flatter planning which requires higher planning horizons. Deciding when and whether to learn skills is a very challenging problem, and several solutions to address it are proposed in [19], [4]. While some of these methods could be applied to our work, we chose to focus on learning skills from a curriculum for simplicity. A curriculum is a sequence of goals of increasing complexity, designed by an expert to help learning. The curriculum is composed of a sequence of useful goals, and that mastering earlier goals of the curriculum helps achieving the latter ones. Following this principle, every element of the curriculum is considered a useful skill to learn. In opposition, learning through random exploration, or even intrinsic exploration [21], does not necessarily help discover *useful* goals and skills.

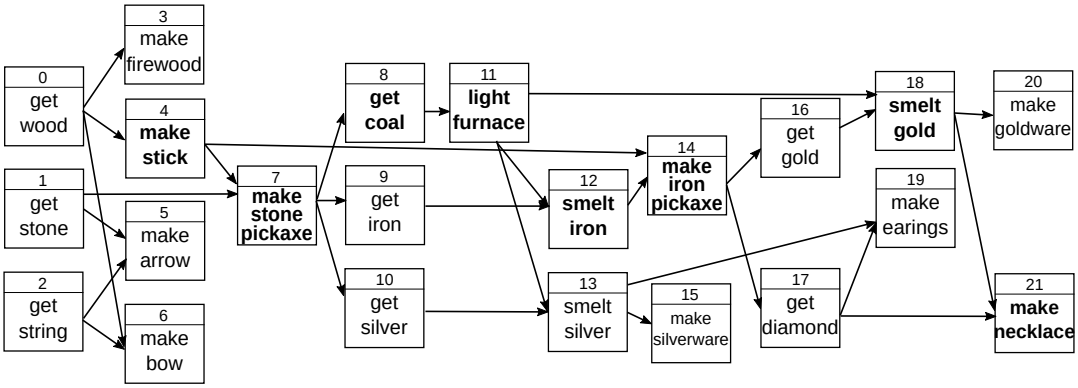


Fig. 2: Dynamics for Mining domain, and curriculum in bold. Nodes are skills and incoming edges are success conditions.

TABLE I: Example of primitive skill conditions learned by the hierarchical agent on the Crafting domain after 41 training episodes.

Effects	Real conditions	Learned conditions
$s_0 \leftarrow 1$	None	None
$s_1 \leftarrow 1$	None	None
$s_2 \leftarrow 1$	None	None
$s_3 \leftarrow 1$	$s_0 = 1$	$s_0 = 1$
$s_4 \leftarrow 1$	$s_0 = 1$	$s_0 = 1$
$s_5 \leftarrow 1$	$s_1 = s_2 = 1$	$s_1 = s_2 = 1$
$s_6 \leftarrow 1$	$s_0 = s_2 = 1$	$s_2 = 1$
$s_7 \leftarrow 1$	$s_1 = s_4 = 1$	$s_1 = s_4 = 1$
$s_8 \leftarrow 1$	$s_7 = 1$	$s_7 = 1$
$s_9 \leftarrow 1$	$s_7 = 1$	$s_7 = 1$
$s_{10} \leftarrow 1$	$s_7 = 1$	$s_7 = 1$
$s_{11} \leftarrow 1$	$s_8 = 1$	$s_8 = 1$
$s_{12} \leftarrow 1$	$s_9 = s_{11} = 1$	$s_9 = s_{11} = 1$
$s_{13} \leftarrow 1$	$s_{10} = s_{11} = 1$	$s_{10} = 1$
$s_{14} \leftarrow 1$	$s_4 = s_{12} = 1$	$s_{12} = 1$
$s_{15} \leftarrow 1$	$s_{13} = 1$	$s_{13} = 1$
$s_{16} \leftarrow 1$	$s_{14} = 1$	$s_{14} = 1$
$s_{17} \leftarrow 1$	$s_{14} = 1$	$s_{14} = 1$
$s_{18} \leftarrow 1$	$s_{11} = s_{16} = 1$	$s_{16} = 1$
$s_{19} \leftarrow 1$	$s_{13} = s_{17} = 1$	$s_{17} = 1$
$s_{20} \leftarrow 1$	$s_{18} = 1$	$s_{18} = 1$
$s_{21} \leftarrow 1$	$s_{17} = s_{18} = 1$	$s_{17} = s_{18} = 1$

V. EXPERIMENTS

We present planning results for the presented method in several environments. All environments follow the framework presented in Section III. As such, dynamics are represented as a DAG, where each node is a primitive skill and connections represent conditions. See Figure 2 for an example. For simplicity, effects are defined as a state dimension changing from a value of 0 to 1. In all experiments, goals are defined as regions of the state space where some state dimensions are fixed to a desired value. For example, both states $(1, 0, 1)$ and $(1, 0, 0)$ match the goal defined as $s_0 = 1$.

We compare the following methods in experiments. *MCTS* [5] and *RRT* [17] are non-hierarchical planners that require transition dynamics to be given. They both build a tree of possible futures, by simulating executing skills and their effects on the state. Once a search budget is exhausted, they return the action corresponding to the shortest skill trajectory achieving the goal. We compare MCTS with several search budgets of 100, 300 and 1000, and exploration constant set to $\frac{1}{\sqrt{2}}$. RRT's search mechanism is biased towards the goal with probability 0.1, and is run for 1000 and 10000 steps.

Q-learning [36] is a classic RL method, and does not require the transition dynamics to be specified. A learning procedure is necessary before plans can be generated. Q-learning is trained using a reward function designed for a specific goal (reward of 0 for reaching the goal and -1 otherwise), and thus would need to be trained anew for each goal. In experiments, we allow a maximum of 5000 learning episodes, which are sequences of up to 100 actions. If Q-learning successfully finds the goal in 19 out of the last 20 episodes, we stop the learning procedure. We use a tabular version of Q-learning, with known convergence guarantees. The discount factor is set to 0.99, the learning rate to 0.1, and an epsilon-greedy policy with 0.2 probability of random action is used.

DQN [20] is a recent extension to Q-learning, using a neural network to model the Q function. The training procedure and parameters are akin to that of Q-learning. One hidden layer with 16 units is used, and the epsilon-greedy policy parameter linearly decays from 1.0 to 0.1.

The *Goal-regression* planner, defined in Algorithm 1, is a simple planner that plans backwards starting from the goal. It recursively plans for the conditions of the last goal until the starting state is reached. In experiments, the maximum recursion of this planner is set to 100. This planner does not use hierarchical planning and requires known transition dynamics.

Our method, denoted *Hierarchical*, is compared in three configurations. In its simplest form, it requires a list of non-primitive skills to be specified by an expert, as well as transitions to be known; it does not require training. The second variant learns non-primitive skills automatically from data, while still requiring known transition dynamics. The last form learns both skills and transitions from data. The last two variants both require training with a problem specific curriculum (either a list of goals or demonstrations). As learning skills with known dynamics require a single successful episode, the algorithm advances to the next curriculum stage after its first success on the task. When also learning dynamics, it advances after 5 successes on a task. The three *Hierarchical* planners are given a maximum planning recursion of $rec_{max} = 3$. The GMM used to learn conditions uses 3 components and the OMP tolerance is set to 3.

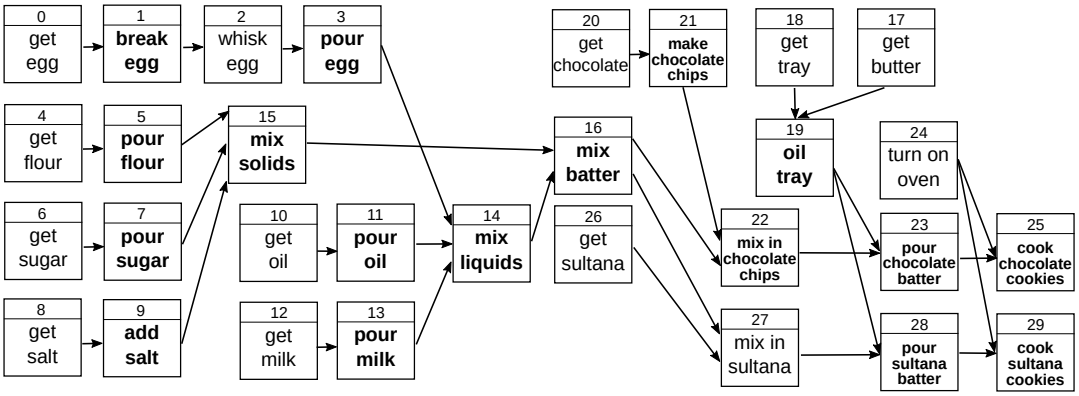


Fig. 3: Dynamics for Baking domain, and curriculum in bold. Nodes are skills, incoming edges are success conditions.

TABLE II: Example of skills learned by the hierarchical agent while learning skill conditions (Table I) on the Crafting domain after 41 training episodes. $skill(s_i)$ refers to a learned skill defined in a previous row with effect $s_i \leftarrow 1$.

Effect	Plan	Flattened plan
$s_4 \leftarrow 1$	$0, 4$	$0, 4$
$s_7 \leftarrow 1$	$1, skill(s_4), 7$	$1, 0, 4, 7$
$s_8 \leftarrow 1$	$skill(s_7), 8$	$1, 0, 4, 7, 8$
$s_{11} \leftarrow 1$	$skill(s_8), 11$	$1, 0, 4, 7, 8, 11$
$s_{12} \leftarrow 1$	$skill(s_{11}), 9, 12$	$1, 0, 4, 7, 8, 11, 9, 12$
$s_{14} \leftarrow 1$	$skill(s_{12}), 14$	$1, 0, 4, 7, 8, 11, 9, 12, 14$
$s_{18} \leftarrow 1$	$skill(s_{14}), 16, 18$	$1, 0, 4, 7, 8, 11, 9, 12, 14, 16, 18$
$s_{21} \leftarrow 1$	$skill(s_{18}), 17, 21$	$1, 0, 4, 7, 8, 11, 9, 12, 14, 16, 18, 17, 21$

All methods plan and execute one skill at a time, and are terminated if unable to reach the goal after executing 100 planned skills. Planning results are compared in terms of number of training steps n_{train} (when applicable), plan length $|\pi|$, and time required to generate the plan t_{plan}. All algorithms are run on a single CPU core (2.2GHz), are averaged over 10 runs, and results are reported in Table III.

A. Crafting problem

The *Crafting* environment was introduced by [29], and its deterministic transition dynamics graph is composed of $n = 22$ nodes (see Figure 2) with an average number of 1.3 conditions per node (see Table I). This problem models a robot needing to gather raw materials to craft tools, which are in turn required to gather more advanced materials and craft other objects. Each of the 22 skills corresponding to graph nodes results in an equivalent state change, eg. executing skill *craft wood stick* changes state predicate *has wood stick* to $True$. This problem features $2^{22} \approx 4.10^6$ different states.

The curriculum used for this problem focuses on graph nodes that are used more often (ie. have more than one child), shown in bold in Figure 2. Examples of skills learned using this curriculum with their hierarchical and flattened plans are given in Table II. The hierarchical plans demonstrate the skill refinement algorithm's capabilities for identifying skills within successful demonstrations and composing learned skills. Table I shows conditions learned by the hierarchical with unknown transition dynamics. While not all conditions are correctly learned, most of them match the real conditions and these are sufficient to generate good plans.

B. Baking problem

The *Baking* environment models a robot generating plans to bake cookies. Different steps of baking process are included, such as *mix batter* or *oil tray*. The robot is assumed to have previously learned every one of the individual $n = 30$ primitive skills of the transition graph shown in Figure 3. From the abstracted skill planning level, the problem is deterministic, and the average number of skill conditions is 1. The environment has $2^{30} \approx 10^9$ different states. We generated a curriculum with the same principle as for the previous environment, including skills irrelevant to the testing goal, and omitting some of the steps. The curriculum is shown in bold in Figure 3.

C. Randomly generated problems

Larger environments are generated by randomly generating directed acyclic task graphs with a fixed number of nodes $n = 100$. The number of conditions for each skill is drawn from a Poisson distribution ($\lambda = 2$). The generated graphs are denser than that of the previous experiments with an average condition number of 2, and so less adapted to purely sequential plans. Furthermore, we also introduce stochastic transitions, following Equation 3, where states are corrupted by switching a random dimension with probability $p = 0.2$. This problem has $2^{100} \approx 10^{30}$ states. The curriculum is automatically generated by ordering non-root transition graph nodes by increasing number of ancestors, only selecting every second node as an intermediary goal. Because expert skills cannot easily be created for randomly generated transitions, the hierarchical planner with given skills is not run on this problem.

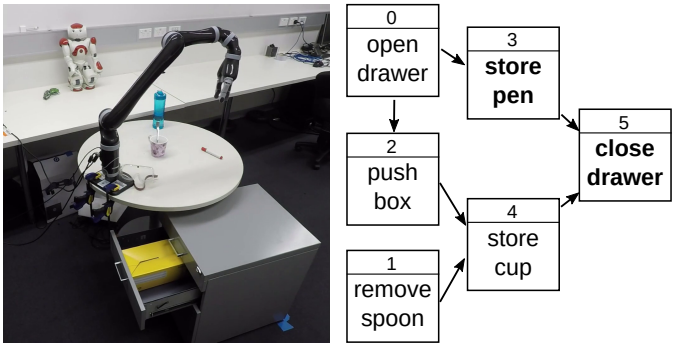
D. Robotics problem: transfer from simulation to real

This last experiment aims to show policies generated in simulation can be applied to a real robot. The problem features a 6 DOF robotics manipulator, shown in Figure 4a, aiming to tidy a table by storing items in a drawer. Similarly to previous problems, execution dependencies need to be resolved, eg. a

TABLE III: Results in terms of plan length $|\pi|$, number of training episodes n_{train}, and planning time t_{plan} in seconds.

Method	**Learning Skills**	**Learning Transitions**	**Crafting** (n=22)			**Baking** (n=30)			**Random graph** (n=100)		
			n_{train}	$\|\pi\|$	t_{plan}	n_{train}	$\|\pi\|$	t_{plan}	n_{train}	$\|\pi\|$	t_{plan}
MCTS (100)	No	No	–	38.9	21.3	–	73.0	68.1	–	44.0	65.3
MCTS (300)	No	No	–	24.5	34.2	–	55.9	135.08	–	85.3	318.1
MCTS (1000)	No	No	–	27.2	115.6	–	48.4	362.1	–	75.3	8584
RRT (1000)	No	No	–	65.4	23.1	–	42.4	16.9	–	64.8	12.7
RRT (10000)	No	No	–	45.4	123.6	–	29.0	86.6	–	45.7	87.1
Q-Learning	No	**Yes**	580.0	70.9	0.039	5000	88.7	0.057	5000	51.7	0.129
DQN	No	**Yes**	5000	94.9	0.480	5000	–	–	5000	–	–
Goal-regression	No	No	–	24.1	0.0082	–	39.5	0.0183	–	73.67	0.063
Hierarchical	No	No	–	13	0.0384	–	24.0	0.104	–	–	–
Hierarchical	**Yes**	No	10.0	13.1	0.0328	17.0	25.0	0.0978	17.0	25.3	0.109
Hierarchical	**Yes**	**Yes**	62.8	13.7	0.1376	96.0	29.3	0.3714	96.1	28.7	0.340

box within the drawer must be pushed to the side before a cup or a pen can be stored next to it. These dependencies are shown in Figure 4b. The manipulator is pre-trained by an expert to execute each individual skill of the graph. This graphical representation is used to learn in simulation, with curriculum $[3, 5]$, and generate a *tidy-up* plan. The resulting plan $[0, 1, 2, 4, 3, 5]$ was generated in 10ms and is optimal; it is executed on the real robot using pre-trained skills and completes the task, as shown in a supplementary video.



(a) (b)

Fig. 4: (a) Robotics manipulator used in simulation-to-real Drawer experiment. (b) Dynamics graph for Drawer problem.

VI. DISCUSSION

Multiple key results can be observed from Table III. The presented hierarchical planning algorithms (whether learning skills and transitions or not) find shorter plans than the non-hierarchical equivalent *Goal-regression*. This is especially noticeable on problems where the underlying transition graph has increased branching factor like in the *Random* domain. This is because the problems featured in experiments have a hierarchical nature, and not abstracting plans makes planning brittle to small errors when skill effects and conditions are not perfectly matched.

Planning with goal regression methods is order of magnitude faster than MCTS and RRT. This is to be expected because RRT has no novelty seeking mechanism, and MCTS is not given explicit knowledge of the goal, thus it needs to stumble upon it by chance. This gets worse in problems with higher state dimensions and skill number, whereas goal regression planners don't seem to be affected (as shown by a similar planning time) as they don't need to deal with exploration. For the same reasons, Q-learning and DQN quickly require extensive training time and fail to reach the convergence criteria within the allocated number of training episodes. DQN, which does not have the same convergence guarantees as tabular Q-learning, also failed to find a valid plan on the two more complicated domains.

Learning transitions requires a substantial number of extra learning steps, but ultimately reaches performance similar to the same planner given transitions. Learning skills from curriculum compared to planning with predefined expert skills does not seem to impact planning length much – provided the given curriculum is good. Learning skills using a curriculum also requires very few learning steps. Lastly, planning methods seem robust to stochastic transitions, as shown in the *Random* domain.

Although showing impressive results, the hierarchical planner still has limitations. The environments used in experiments feature relatively simple conditions, defined as the union of a few state dimensions. It would be interesting to extend the condition learning algorithm to handle more complicated conditions. The current method also requires a curriculum to be designed, conveying what goals are reusable in future tasks. This limitation is relatively mild, given that curricula designed for humans are available for many real-life tasks, which our method could be adapted to leverage.

VII. CONCLUSION AND FUTURE WORK

We presented a framework and algorithm for learning to plan hierarchically in domains with unknown dynamics, in which transition dynamics are decomposed into skill effects and conditions. We proposed a method for learning both abstract skills and their success conditions from interaction with the environment. We validated our method in experiments of increasing complexity (with up to 2^{100} states), demonstrating superior planning to classic non-hierarchical planners or reinforcement learning methods. Lastly, we showed the algorithm is applicable to robotics problems in a simulation-to-real transfer problem.

The presented algorithm is also able to interact with humans. Because skill conditions are explicitly modelled, the planner is aware of what it doesn't know and can ask queries such as *What do I need to craft a pickaxe?*. Human responses as conditions (*get a wood stick and stone*) or a sequence of

goals (*craft a wood stick, get stone, craft pickaxe*) can both be used to refine skill conditions.

The avenues for future work are numerous: Learning conditions using causal learning [13] or inductive logic programming [22] could be greatly beneficial. Having a causal model would allow agents to actively explore the space of possible transition dynamics, by choosing actions that run causal tests. This kind of active exploration could result in significant improvement over choosing random actions. Generating and updating a curriculum automatically would reduce the amount of expert knowledge required to apply our method. The work of [9] uses generative adversarial networks to automatically generate challenging goals, and could be combined with our work. It is yet unclear how the presented algorithm would scale to much larger problems such as life-long learning. Skill forgetting and/or better refinement would probably be necessary to tackle these challenging problems. Although our framework features sparse state spaces, more domain structure could be exploited using concepts from the oriented-object MDP framework [7]. This would enable reasoning about object function, properties and skill affordances [30]. Actions available to the agent could also be restricted depending on the objects observed in the state space [23].

REFERENCES

[1] Barrett Ames, Allison Thackston, and George Konidaris. Learning symbolic representations for planning with parameterized skills. In *International Conference on Intelligent Robots and Systems*, 2018.

[2] Bram Bakker and Jürgen Schmidhuber. Hierarchical reinforcement learning based on subgoal discovery and subpolicy specialization. In *Conference on Intelligent Autonomous Systems*, 2004.

[3] Craig Boutilier, Richard Dearden, and Moisés Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 2000.

[4] Emma Brunskill and Lihong Li. Pac-inspired option discovery in lifelong reinforcement learning. In *International Conference on Machine Learning*, 2014.

[5] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International Conference on Computers and Games*, 2006.

[6] LS Homem De Mello and Arthur C Sanderson. And/or graph representation of assembly plans. *Transactions on robotics and automation*, 1990.

[7] Carlos Diuk, Andre Cohen, and Michael L Littman. An object-oriented representation for efficient reinforcement learning. In *International Conference on Machine Learning*, 2008.

[8] Richard E Fikes and Nils J Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 1971.

[9] Carlos Florensa, David Held, Xinyang Geng, and Pieter Abbeel. Automatic goal generation for reinforcement learning agents. In *International Conference on Machine Learning*, 2018.

[10] Caelan Reed Garrett, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Sample-based methods for factored task and motion planning. In *Robotics: Science and Systems*, 2017.

[11] Matthew Grounds and Daniel Kudenko. Combining reinforcement learning with symbolic planning. In *Adaptive Agents and Multi-Agent Systems*. 2008.

[12] Bradley Hayes and Brian Scassellati. Autonomously constructing hierarchical task networks for planning and human-robot collaboration. In *International Conference on Robotics and Automation*, 2016.

[13] Paul W Holland. Statistics and causal inference. *Journal of the American Statistical Association*, 1986.

[14] Lars Johannsmeier and Sami Haddadin. A hierarchical human-robot interaction-planning framework for task allocation in collaborative industrial assembly processes. *Robotics and Automation Letters*, 2017.

[15] Ross A Knepper, Dishaan Ahuja, Geoffrey Lalonde, and Daniela Rus. Distributed assembly with and/or graphs. In *Workshop on AI Robotics at the Int. Conf. on Intelligent Robots and Systems (IROS)*, 2014.

[16] George Konidaris, Leslie Pack Kaelbling, and Tomas Lozano-Perez. From skills to symbols: Learning symbolic representations for abstract high-level planning. *Journal of Artificial Intelligence Research*, 2018.

[17] Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning. 1998.

[18] Stéphane G Mallat and Zhifeng Zhang. Matching pursuits with time-frequency dictionaries. *Transactions on Signal Processing*, 1993.

[19] Amy McGovern and Andrew G Barto. Automatic discovery of subgoals in reinforcement learning using diverse density. In *International Conference on Machine Learning*, 2001.

[20] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv, arXiv:1312.5602*, 2013.

[21] Philippe Morere and Fabio Ramos. Bayesian rl for goal-only rewards. In *Conference on Robot Learning*, 2018.

[22] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 1994.

[23] Lakshmi Nair and Sonia Chernova. Action categorization for computationally improved task learning and planning. In *International Conference on Autonomous Agents and MultiAgent Systems*, 2018.

[24] Dana Nau, Yue Cao, Amnon Lotem, and Hector Munoz-Avila. Shop: Simple hierarchical ordered planner. In *International Joint Conference on Artificial Intelligence*, 1999.

[25] Negin Nejati, Pat Langley, and Tolga Konik. Learning hierarchical task networks by observation. In *International Conference on Machine Learning*, 2006.

[26] Oliver Obst. Using a planner for coordination of multiagent team behavior. In *International Workshop on Programming Multi-Agent Systems*, 2005.

[27] Douglas Reynolds. Gaussian mixture models. *Encyclopedia of Biometrics*, 2015.

[28] Ron Rubinstein, Michael Zibulevsky, and Michael Elad. Efficient implementation of the k-svd algorithm using batch orthogonal matching pursuit. Technical Report CS-2008-08, 2008.

[29] Sungryull Sohn, Junhyuk Oh, and Honglak Lee. Hierarchical reinforcement learning for zero-shot generalization with subtask dependencies. In *Advances in Neural Information Processing Systems*, 2018.

[30] Mohan Sridharan, Ben Meadows, and Rocio Gomez. What can i not do? towards an architecture for reasoning about and learning affordances. In *International Conference on Automated Planning and Scheduling*, 2017.

[31] Richard S Sutton. Dyna, an integrated architecture for learning, planning, and reacting. *ACM SIGART Bulletin*, 1991.

[32] Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 1999.

[33] Anderson Rocha Tavares, Sivasubramanian Anbalagan, Leandro Soriano Marcolino, and Luiz Chaimowicz. Algorithms or actions? a study in large-scale reinforcement learning. In *International Joint Conference on Artificial Intelligence*, 2018.

[34] Valentin Thomas, Jules Pondard, Emmanuel Bengio, Marc Sarfati, Philippe Beaudoin, Marie-Jean Meurs, Joelle Pineau, Doina Precup, and Yoshua Bengio. Independently controllable features. *arXiv, arXiv:1708.01289*, 2017.

[35] M van Otterlo. A survey of reinforcement learning in relational domains. *CTIT Technical Report Series*, (05-31), 2005.

[36] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine Learning*, 1992.

[37] Nan Ye, Adhiraj Somani, David Hsu, and Wee Sun Lee. Despot: Online pomdp planning with regularization. *Journal of Artificial Intelligence Research*, 2017.