Robust Hierarchical Planning with Policy Delegation

Tin Lai  
The University of Sydney  
tin.lai@sydney.edu.au

Philippe Morere  
The University of Sydney  
philippe.morere@sydney.edu.au

ABSTRACT

We propose a novel framework and algorithm for hierarchical planning based on the principle of delegation. This framework, the Markov Intent Process, features a collection of skills which are each specialised to perform a single task well. Skills are aware of their intended effects, and are able to analyse planning goals to delegate planning to the best suited skill. This principle dynamically creates a hierarchy of plans, in which each skill plans for sub-goals for which it is specialised. The proposed planning method features on-demand execution—skill policies are only evaluated when needed. Plans are only generated at the highest level, then expanded and optimised when the latest state information is available. The high-level plan retains the initial planning intent and previously computed skills, effectively reducing the computation needed to adapt to environment changes. We show this planning approach is experimentally very competitive to classic planning and reinforcement learning techniques on a variety of domains, both in terms of solution length and planning time.

KEYWORDS

Hierarchical Planning, Reinforcement Learning, Delegation

1 INTRODUCTION

Decision making techniques enable automating many real-world tasks that would be too repetitive or even intractable to humans. Such methods require making good decisions in any situation, improved by reacting to the latest available information and revising previous intentions. This procedure is in practice extremely time sensitive, as the ability to react quickly is paramount in many real-world problems where the time available to make decisions is very limited.

Decision making typically involves generating plans by searching over the space of potential solutions. Classic planning methods search over plans by simulating various possible future scenarios. This quickly becomes prohibitively expensive in problems with larger state and/or action spaces, and greatly reduces their applicability to real-time problems. Furthermore, plans generated this way often cannot be reused in similar situations, and unforeseen state changes often require expensive re-planning. This makes such planning techniques inefficient and slow. More recent hierarchical planning and reinforcement learning methods overcome some of these issues by planning at several levels of abstraction. Hierarchical planning decomposes the problem into smaller sub-problems, for which specialised skills are learned. These skills are highly reusable, easier to learn than general policies, and achieve better performance. However, these hierarchical techniques are typically not aware of which task each skill is trained to solve. Knowledge of each skill’s purpose is extremely important, as it allows seamlessly delegating planning to more specialised skills. This lack of awareness arises from the Markov Decision Process (MDP), the base framework for most of these methods, in which the effect of actions (or skills) is unknown. Because of this, existing hierarchical planning methods can be computationally inefficient, and learning specialised skills can prove challenging.

We present a hierarchical planning methodology for reasoning about the effects of skills and primitive actions, yielding several benefits. Planning using skill effect knowledge allows to directly select the best skill for any given task, reducing planning time and computation. This knowledge also enables planners to reason on whether executed skills or actions were successful by comparing the expected and observed effects; this allows learning action success conditions directly from interactions. Furthermore, the presented method plans at the highest level only, and expands plans into more detailed plan on-demand. Thus, latest state information can be taken into account, making plans reactive to noise and adversarial actions. Also, plans do not need to be re-computed after unforeseen state changes occur, and planning computation is expended only when a higher detail level is needed. This makes planning very inexpensive and fast.

Our contributions are the following. We present a new sequential decision making framework, the Markov Intent Process (MIP), incorporating action and skill effects at its core. We formulate the notion of optimal plan in MIP, and propose to convert the sequential decision making problem into a collection of non-sequential decision making problems, which are easier to solve. We present a hierarchical intent-aware on-demand planning algorithm—called Delegate—based on the MIP framework. Finally, we experimentally show Delegate is resilient to noise and outperforms other classic planning and reinforcement learning methods, both in terms of planning time and solution length, on a variety of domains.

2 RELATED WORK

Planning by reasoning on the effects and conditions of actions has received much attention over the years. Classic planners like STRIPS [1] are based on this principle, producing sequential plans using known action effects and conditions. Hierarchical Task Networks (HTN) improve on STRIPS by providing a hierarchical alternative to generate plans [2]–[5]. However, these planners necessitate skill hierarchy expert knowledge. More work has aimed to extend HTN planners to learn the skill hierarchy automatically [6]. However, skills hierarchy learning requires experts demonstrating solutions to the problem.

The field of reinforcement learning (RL) has also produced much work on sequential decision making [7], mostly focusing on cases with unknown transition dynamics. RL defines the problem with a
We start by describing the Markov Intent Process—a novel framework with objects and their relations. Following similar ideas, the Object-Gates planning [9] is a framework that can all be executed in Tree Search [9] or model predictive control [10] to simulate possible actions. These methods often leverage classic planners such as Monte-Carlo simulation [11], [12]. The option framework [11] defines temporally extended actions called options which can form hierarchical plans. Options are characterised by initiation and termination state sets, defining a region of the state space in which options can be used. This differs from MIP skills, which can all be executed in any state, but aim at achieving very specific effects (e.g. setting the first state dimension to one). While the nature and number of options is highly dependent on state space geometry [13], desired state space manipulations (i.e. skill effects) are always known in advance, as defined by the action space; this is a stark advantage of skills over options. Method to learn options by identifying transition data clusters was proposed in [14], though it is limited to few levels of hierarchy. Symbolic planning and reinforcement learning have also been combined in [15], where the RL reward function is generated by a STRIPS. This approach combines reasoning advantages of STRIPS to the reactivity of RL. Because RL is based on the MDP framework which defines objectives with a reward function, RL based methods typically can’t handle multiple or changing goals.

Reasoning about actions and their effects is also the focus of relational RL [16], where state and action spaces are upgraded with objects and their relations. Following similar ideas, the Object-Oriented MDP framework [17] models state objects and their interactions. These ideas enable scaling RL to problems with very high dimensions, as long as they can be expressed in terms of a known decomposition into objects. Most similar to this work is that of [18], which learns a hierarchy of skills using an expert-generated curriculum to guide learning. While the planning method is similar, it is restricted to specific sets of conditions and effects, thus not applicable to more complex problems.

This work combines the advantages of hierarchical planners to that of reasoning over action effects and conditions. Unlike previously mentioned work, the proposed method automatically delegates planning to the most specialised skill for the task, leading to faster and more efficient planning.

3 FORMULATION

We start by describing the Markov Intent Process—a novel framework for hierarchical decision making—that promotes delegating tasks whenever possible.

3.1 Markov Intent Process

Definition 1 (Markov Intent Process). A Markov Intent Process (MIP) is a tuple \((S, A, \hat{A}, E, T, C)\) composed of a set of states \(S\), a set of learnable (and initially empty) skills \(A\), a set of primitive actions \(\hat{A}\) with known effects \(e \in E\), a transition noise function \(T\), and a set \(C\) of successful state spaces for each action. A MIP follows the Markov property, which states that the resulting state of a transition \(s'\) only depends on the starting state \(s\) and primitive action \(a \in \hat{A}\).

Definition 2 (Primitive Action). Each primitive action \(a \in \hat{A}\) is associated with a set of states \(S_a \subseteq C\), in which executing \(a\) would be successful. Successfully executing \(a\) in a state \(s \in S_a\) would apply the action’s effect \(e\) to the current state. This is given by the environment.

Definition 3 (High-Level Skill). A MIP skill \(a_e \in A\), identified by its effect \(e\), is defined as a tuple \(a_e := (e, \pi_e)\) of (i) a known intended effect \(e\), describing the effect of the skill if \(a_e\) is successfully executed (and no noise is observed); and (ii) a learnable policy \(\pi_e\), mapping a state in \(S\) to a plan \(\nu\). Unlike actions, skills do not require a successful state space, as it can be automatically inferred from a skill’s plan. The set of skills \(A\) is not given to MIP agents, which need to discover and learn them.

Definition 4 (Plan). A plan \(\nu\), generated by a policy \(\pi_e(s)\), is a (potentially empty) sequence of skills \(a \in A\) and/or actions \(\hat{a} \in \hat{A}\) aiming to achieve effect \(e\) from state \(s\), possibly with some unknown side effects. A plan implicitly defines a hierarchy and order of operations to achieve the skill’s intended effect. Hereafter, for notation brevity and clarity, we will refer to a skill’s policy as \(\pi_e \equiv \pi_{a_e}\).

Each skill from the set of skills \(A\) has a known immutable effect. Skill policies may generate plans composed of a single primitive action \(\hat{a}\), should it succeed in the current state \(s\), i.e. \(s \in S_{\hat{a}}\). Otherwise, a skill’s policy should return a plan composed of other skills, whose effects are necessary before the skill’s effect could be achieved by a single primitive action from \(\hat{a}\). As such, executing a skill may result in delegating parts of its plan to other skills. That is, executing a plan might recursively queries other skill which in-turn returns another nested plan to be executed.

Given a state \(s\) and a primitive action \(\hat{a}\) with effect \(e\), transitions to a new state \(s'\) occur as:

\[
s' = \begin{cases} T(s \circ e) & \text{if } s \in S_{\hat{a}} \\
T(s) & \text{otherwise.} \end{cases}
\]

(1)

Here, \(\circ\) denotes applying effect \(e\) to state \(s\), and the transition noise function \(T\) potentially corrupts the resulting state \(s'\) with noise. The set of successful state spaces for all actions \(C\) and transition noise function \(T\) are not necessarily known, thus MIP agents need to be reactive to the environmental changes.

3.2 Action Success Conditions

The MIP framework generalises the definition of the success conditions of an action \(\hat{a}\) as the set of of states \(S_{\hat{a}}\) in which \(\hat{a}\) succeeds. For simplicity, we only consider conditions that are expressed as the intersection of unit state space feature conditions.

Formally, let \(\Phi : S \to \mathbb{R}\) denote one of \(m\) functions mapping a state to a corresponding state feature. Let \(\mathcal{P}\) denote the space of propositions mapping a state feature \(\Phi(s)\) to a boolean value \(\{true, false\}\). We write \(P^i(\Phi)\) to denote a proposition \(P^i \in \mathcal{P}\) operating on state feature \(\Phi\) in state \(s\).

Definition 5 (Action Condition). The condition of action \(\hat{a} \in \hat{A}\) is defined as the minimal non-empty set of \(d \leq m\) feature propositions \(\xi_{\hat{a}} = \{P^i_{\Phi_{i_1}}(\cdot), \ldots, P^i_{\Phi_{i_d}}(\cdot)\}\), where \(i_1, \ldots, i_d\) and \(\hat{a}_1, \ldots, \hat{a}_d\) are
Therefore, $T$ complete action set status. A stronger guarantee is that of complete action set. 

$\bar{A}$ and there exists no plan $\hat{a}$ be the sets of all possible start and goal states in our problem space however, does not guarantee that the problem is solvable. To do so, $\bar{A}$ will be used to denote an action set that is at least as good as sufficient action set (i.e. $\bar{A} \subseteq \bar{A}$), and we consider solutions to MIP problems should not explicitly rely on noise. Lastly, we restrict the arbitrary transition function $T$ to be a random and non-adversarial transition function. That is because if an adversarial $T$ always negates any executed effects, there exists no plan that can make any progress.

## 3.3 Well-formed Problems

In order to guarantee MIPs are well-formed and can be solved, we first need to define constraints on skills and action conditions, e.g. no circular dependency. Let us first define the notions of success prerequisite and well-formed action sets.

**Definition 6 (Success prerequisite).** We say that $\hat{a}_i$ is a success prerequisite of action $\hat{a}_j$ in $s \notin S_{\hat{a}_j}$ if $\hat{a}_j$ is a necessary action in all possible action sequences that can transits $s$ to $s' \in S_{\hat{a}_j}$.

**Definition 7 (Well-formed action set).** An action set $\bar{A}$ is said to be well-formed if it contains no ill-formed action; that is, no circular dependencies occur when unrolling conditions of $\hat{a} \in \bar{A}$. Formally, $\hat{a}$ is ill-formed if there exists a state $s \in S \setminus S_{\hat{a}}$ for which $\hat{a}$ itself is one of the success prerequisites.

Definition 7 prevents any ill-formed actions from generating an illogical circular dependency, which would otherwise cause impossible situations where in order to reach states where $\hat{a}$ succeeds, action $\hat{a}$ must first be executed (i.e. $\hat{a}$ a prerequisite for itself). A well-formed action set guarantees the conditions are logically sound; however, does not guarantee that the problem is solvable. To do so, we sart by defining the notion of sufficient action set.

**Definition 8 (Sufficient action set).** Let $S_0 \subseteq S$ and $S_g \subseteq S$ be the sets of all possible start and goal states in our problem space respectively, $\bar{A}_{suff}$ be an action set that satisfies definition 7, and $s_0 \in S_0$ and $s_g \in S_g$. If there exists at least one action sequence $v = (\hat{a}_e, \ldots, \hat{a}_r)$ that can reach $s_g$ from $s_0$, where $\hat{a}_e, \ldots, \hat{a}_r \in \bar{A}_{suff}$ and $e_0, \ldots, e_j \in E$, then $\bar{A}_{suff}$ is said to be a sufficient action set for the problem space. Formally, denoting $\bar{Y}_{\bar{A}_{suff}}$ as the space of all possible plans formed by using only elements in $\bar{A}_{suff}$,

$$\forall s_0 \in S_0 \; \forall s_g \in S_g \; \forall v = (\hat{a}_e, \ldots, \hat{a}_r) \in \bar{Y}_{\bar{A}_{suff}},$$

$$s_g = \left( (s_0 \oplus e_0) \oplus \cdots \oplus e_j \right). \quad (3)$$

Notice that if we were to apply the transition noise function $T$ to the previous formulation, $\bar{A}_{suff}$ would no longer be sufficient, as a noisy transitions may result in states in which definition 8 does not hold; i.e. there might exists $s' = T(s)$ where $s \in S_0$ but $s' \notin S_0$, and there exists no plan $v \in \bar{Y}_{\bar{A}_{suff}}$ such that $v$ can reach $s_g$ from $s$. Therefore, $T$ could corrupt the current state into an unrecoverable status. A stronger guarantee is that of complete action set.

**Definition 9 (Complete action set).** An action set $\bar{A}_{comp}$ that satisfies definition 7 is said to be a complete action set if it can complete all possible problems, even in the presence of noise $T$; i.e. if there exists at least one action sequence $v \in \bar{Y}_{\bar{A}_{comp}}$ that can reach any $s_g \in S$ from any $s_0 \in S$.

This is a special case of definition 8 where it reduces $S_0 = S_g = S$, and it is immediate that $\bar{A}_{suff} \subseteq \bar{A}_{comp}$ as $\bar{A}_{comp}$ will always be sufficient to complete any problem. In practice, a sufficient action set is often adequate if the noise function $T$ is relatively well behaved, e.g. in cases where $T$ can equally bring the state $s$ into unrecoverable state and bring it back to a favourable state space region. In addition, we consider achieving goal state purely through noise as a trivial solution, which should not considered a possible solution. Hereafter, $\bar{A}$ will be used to denote an action set that is at least as good as sufficient action set (i.e. $\bar{A}_{suff} \subseteq \bar{A}$), and we consider solutions to MIP problems should not explicitly rely on noise. Lastly, we restrict the arbitrary transition function $T$ to be a random and non-adversarial transition function. That is because if an adversarial $T$ always negates any executed effects, there exists no plan that can make any progress.

## 3.4 Optimal Plans in a Markov Intent Process

In the following, we will use $| \cdot |$ to denote the number of elements within a plan, and $\| \cdot \|$ to denote the length of the full plan (fully flattened plan where we recursively unroll all high-level skills), i.e.

$$\| v_e \| = \sum_{i=1}^{n_e} \| v_{e,i} \|,$$

(4)

where $v_{e,i}$ is the $i^{th}$ element of $v_e$ corresponding to either an inner-plan or primitive action. If $v_{e,i}$ is a primitive action, $\| v_{e,i} \| = 1$.

Contrary to the MDP framework, no reward function is defined, thus solving a MIP cannot be defined as maximising rewards. Instead, planning in a MIP requires a goal state $s_g \in S_g$ to be specified. Solving a MIP optimally is equivalent to finding an optimal plan for all goal states in $S_g$. The optimal plan for $s_g$ from a starting state $s_0$ is the shortest plan to $s_g$, given a limited planning horizon $H$. Formally, denoting $e$ the effect from $s_0$ to $s_g$, the optimal plan $v_e^H$ is

$$v_e^H = \arg \min_{v_{e,i}} \| v_e^H \|,$$

(5)

The planning horizon $H$ restricts the length of all plans within $v_{e}^H$ (and $v_{e}^H$ itself) to be no greater than $H$, i.e. $| v_{e}^H | \leq H$ and all plans within $v_{e}^H$ have horizon $H$. For some short horizons and goals, a valid $v_{e}^H$ may not exist.

Because there is no fixed reward function (unlike in MDPs), MIP’s agents must be able to generate plans for any given goal state. This is much harder than maximising a reward engineered for a single task, and makes MIPs naturally applicable to multi-task, transfer, and life-long learning tasks.

## 4 Planning Through Delegation

Planning optimally in a MIP requires solving Equation 5, which involves an intractable search over all possible plans. We present instead a tractable method for approximating it, generating a successful on-demand plan for a given goal.

### 4.1 Overview

The proposed planning methodology is based on the principle of delegation; skills are defined such that they "Do one thing and do it well" [19]. Each skill has a known effect $e$, and can generate a
plan to achieve this effect from any state $s$. In the easiest situation, achieving $e$ in $s$ only involves returning a primitive action $\hat{a}$ whose effect is $e$. However, in most cases, $\hat{a}$ would fail if directly executed in $s$, and intermediate effects must first be applied to $s$ (through more planning) before $\hat{a}$ can be successfully executed. Once intermediate effects are identified, plans for each of them can be generated on demand from other skills learned specifically for these effects. This lazy plan generation process enables automatically reacting to latest state changes (due to noise for example) and greatly reduces planning time.

4.2 Formulation

Formally, planning through delegation can be defined as follows. Let each skill $a$ (with effect $e$) from the set of skills $\mathcal{A}$ be formulated as a MDP with no transition function nor discount factor $(S, Y_e, R_e)$, denoted $M_e$, following the contextual bands framework [20]. The space of potential plans $\mathcal{P}_e$ to achieve $e$ contains at least all actions in $\mathcal{A}$ whose effect is $e$, and grows as more plans are evaluated by skill $a$. All plans in $\mathcal{P}_e$ must have at most length $H$. The reward function $R_e$ is defined as follows:

$$R_e(s, e) = \begin{cases} 1 & \text{if executing } e \text{ in } s \text{ results in } e, \\ 0 & \text{otherwise.} \end{cases}$$ (6)

Because skill effects are known, checking whether executing a plan from $s$ to $s'$ resulted in $e$ can be checked by inspecting whether $s'$ is consistent with $s \oplus e$. This can be done by the agent, without any additional knowledge (and without skill success condition knowledge).

Equation 6 promotes choosing plans with lower full plan length. Hence if the optimal plan $u^*$ for a given skill of effect $e$ is in $\mathcal{P}_e$, an agent maximising Equation 6 will generate an optimal plan for $e$, as defined in Equation 5. If $u^* \notin \mathcal{P}_e$, then more exploration is needed until $u^*$ is found. This problem can be solved as-is using contextual bandits solvers [21]–[23], in which a trade-off between exploration and exploitation is built-in. However, knowledge of skill success conditions, whether known in advance or learned through interactions, enables better reasoning over potential plans to consider and facilitates finding solutions.

4.3 Delegate

We now describe how Delegate leverages skill success conditions to efficiently find plans maximising Equation 6. While MIPs provide agents with knowledge of primitive action effects, there are two possible variants concerning success condition information: i) action success conditions are given to the agent, or ii) action success conditions are not given but agents can learn them through trial and error. In either approach, agents can construct plans by reasoning on the available primitive actions’ effects and conditions. From here onward, we consider the case where conditions are known.

Planning in a MIP for a given goal state $s_g \in S_g$ from a starting state $s_0 \in S_0$ begins by identifying a sequence of effects $e_0, \ldots, e_i$ which results in $s_g$ if applied to $s_0$, i.e. $s_g = s_0 \oplus e_0 \oplus \ldots \oplus e_i$. Such sequence of effects must exist if the action space is complete. An initial plan $\pi_0$ for task $(s_0, s_g)$ is simply generated by retrieving the skills corresponding to these effects, $\pi_0 = (\alpha_{e_0}, \ldots, \alpha_{e_i})$. This procedure happens whenever a planning starts with new goal state, as described in Algorithm 1. This initial plan, called intent plan, only contains high-level instructions; executing it will require further planning, which can be computed on demand.

4.3.1 Skill delegation policy. Once an intent plan $\pi_0$ is formed, generating actions to execute in specific states is achieved by querying the policies of skills within $\pi_0$. If the first element of $\pi_0$ is a primitive action, there’s no need to call any skill policy, and the action can be executed directly. Else, if the first element of $\pi_0$ is a skill $a$, then its delegation policy $\pi_a$ is queried at the current state $s$. The resulting plan $\pi = \pi_a(s)$ is appended to the beginning of the intent plan $\pi_0$ (after $a$ was removed). The same process is applied to the new intent plan until its first element is a primitive action. This procedure is detailed in GetAction, Algorithm 2.

---

Algorithm 1: Episode in MIP environment

Input: $T, s_0, s_g$, maximum episode length $N$

1. Generate intent plan $\pi_0 = (\alpha_{e_0}, \ldots, \alpha_{e_i})$ from $s_0$ to $s_g$

2. for $t \leftarrow 1$ to $N$

3. $a_t, v_t \leftarrow$ GetAction($u_{t-1}, s_t$)

4. if $s_{t-1} \in S_{a_t}$ then

5. $s_t \leftarrow s_{t-1} \oplus e_t$  // action can execute

6. $s_t \leftarrow T(s_t)$  // noisy environment only

7. if $s_t = s_g$ then

8. return success

9. return failed

---

Algorithm 2: Delegate algorithm

1. function GetAction($s$, $e$)

2. $x, v_{\text{rest}} \leftarrow$ separate first element of $v$

3. if $x \in \mathcal{A}$ then

4. return $x, v_{\text{rest}}$

5. else

6. $v_{\text{nested}} \leftarrow$ DelegatePolicy($\alpha_s$, $s$)

7. $v \leftarrow v_{\text{nested}} \oplus v_{\text{rest}}$  // concatenation

8. return GetAction($u$, $s$)

9. function DelegatePolicy($\alpha_s$, $s$)

10. $\xi_t \leftarrow \left\{ \text{for } s \in S \right\}$  // unmet conditions

11. if $\xi_t = \emptyset$ then

12. return $(\hat{a}_t)$  // terminal plan

13. else

14. $v \leftarrow \emptyset$

15. for $a_t \in \xi_t$ do

16. $\alpha_{e_j} \leftarrow$ skill with effect that satisfies $p_{e_j}^{\alpha}$

17. $v \leftarrow v \oplus \alpha_{e_j}$  // concatenation

18. return $v$
Generating a plan by querying a skill policy is achieved by analysing the success conditions $\xi_\alpha$ of the skill’s primitive action $\hat{a}_\alpha$. The procedure is described in DelegatePolicy, Algorithm 2. From Definition 5, the set of conditions unmet in the current state can be computed $\xi = \left\{ \frac{p^i}{\phi_i} \cdot (\cdot) \in \xi_\alpha \right\}$. If $\xi$ is not empty, then we can construct a plan composed of skills whose effects would satisfy each of the individual elements $\xi$. This plan is returned as the evaluation of the skill’s policy in the current state. The event of $\xi$ being an empty set, a terminal plan composed only of the skill’s primitive action $\hat{a}_\alpha$ is returned.

Definition 10 (Terminal plan). A plan $\nu$ is called a terminal plan if it contains no skills, i.e. $\forall s \in \nu, \forall x \in A$. Empty plans where $|\nu| = 0$ are also considered terminal plans. Terminal plans can be executed directly.

Skill policies are queried recursively, until a termination criterion is met—when a terminal plan is generated.

4.3.2 Planning termination. Let us now analyse conditions under which planning terminates. As per Definition 7, a well-formed problem should contain no circular dependency. However, a policy for skill $\alpha$ could wrongly return a plan that contains skills that would generate plans containing $\alpha$ itself. We call this plan ill-formed, as it would result in circular dependencies.

Definition 11 (Iill-formed plan). A plan $\nu$ is said to be ill-formed if it contains skills that are ancestors of $\nu$. Formally, we define the direct parent of all skills within $\nu$ as skill $\alpha_j$, such that plan $\nu$ was generated in some state $s_i$ as $\nu = \Pi_\alpha(s_i)$. Then $\nu$ is ill-formed if $\forall \alpha \in \eta(\nu), \exists \alpha \in \nu$, where ancestors are all parents of parents until the intent plan $\nu_0$, and the parent of a plan is equivalent to parent the skill that generated it.

In practice, preventing ill-formed plan is achieved by imposing constraints on skill policies, such that no new plan includes any of its ancestors. Because the number of skills available to the planner is finite, the maximum plan depth is bounded. This guarantees a terminal plan will be generated eventually, hence ensuring planning termination.

4.4 Analysis of a concrete example

We now provide a concrete example of planning through delegation, illustrating the properties of DelegatePolicy. In the following, we will use hasState to denote the presence of a state feature of interest, $\sigma_e$ hasState to denote a skill with intended effect satisfying hasState, and $\hat{a}_\alpha$ hasAction to denote a primitive action with specified outcome. We consider part of the Factorio environment, represented as a graph in Figure 2. Skill $\sigma_e$ hasIronPlate must execute action $\hat{a}_\alpha$ makeIronPlate to achieve its intended effect, however the action requires both state features hasIronOre and hasStoneFurnace. Therefore, skill $\sigma_e$ hasIronPlate should form a plan that involves skills $\sigma_e$ hasIronOre and $\sigma_e$ hasStoneFurnace. Planning for effects activating missing state features. Since $\hat{a}_\alpha$ getIronOre has no conditions, the policy of skill $\sigma_e$ hasIronOre will return a terminal plan $\nu_e = \Pi_\alpha(s_i)$, which can be executed directly. However, $\hat{a}_\alpha$ makeStoneFurnace is conditioned on hasStone and hence, skill $\sigma_e$ hasStoneFurnace should in turn generate a nested plan for $\sigma_e$ hasStone.

A possible plan execution scenario for $\nu_e$ hasSteelPlate is detailed below. Note that we abbreviate “Stone” as “St”, “Furnace” as “Fur”, “Plate” as “Pl”, “make” as “mk” and “Iron” as “Ir” in the followings; and we assume none of the conditions are met in the initial states $s_0$.

The element that is being unrolled at each line are are underlined, the ⇒ symbol denotes one possible policy, and the executed primitive action are in bold. Notice that the scenario shown begins in state $s_0$, and every time an action is executed the environment transits to a new state $s_i$. The new state is then used by the next skill’s policy, until the environment finishes at $s_5$ (the state after executing the last action). If the entire scenario is successful, the effect $e$ hasIronPlate is included in $s_5$. Hence, each skill attempts to plan for the conditions that it requires, and delegates “how to achieve those conditions” by querying the respective skill policies.

$$\pi_e \text{ hasSteelPlate}(s_0)$$

⇒ $\Pi_e$ hasFlatPlate, $\alpha_e$ hasStdPl, $\alpha_e$ hasSteelPlate

⇒ $\Pi_e$ hasFlatPlate, $\alpha_e$ hasStdPl, $\alpha_e$ hasSteelPlate

⇒ $\alpha_e$ hasFlatPlate, $\alpha_e$ hasStdPl, $\alpha_e$ hasSteelPlate

⇒ $\alpha_e$ hasFlatPlate, $\alpha_e$ hasStdPl, $\alpha_e$ hasSteelPlate

⇒ $\alpha_e$ hasFlatPlate, $\alpha_e$ hasStdPl, $\alpha_e$ hasSteelPlate

⇒ $\alpha_e$ hasFlatPlate, $\alpha_e$ hasStdPl, $\alpha_e$ hasSteelPlate

⇒ $\alpha_e$ hasFlatPlate, $\alpha_e$ hasStdPl, $\alpha_e$ hasSteelPlate

⇒ $\alpha_e$ hasFlatPlate, $\alpha_e$ hasStdPl, $\alpha_e$ hasSteelPlate

⇒ $\alpha_e$ hasFlatPlate, $\alpha_e$ hasStdPl, $\alpha_e$ hasSteelPlate

⇒ $\alpha_e$ hasFlatPlate, $\alpha_e$ hasStdPl, $\alpha_e$ hasSteelPlate

⇒ $\alpha_e$ hasFlatPlate, $\alpha_e$ hasStdPl, $\alpha_e$ hasSteelPlate

⇒ $\alpha_e$ hasFlatPlate, $\alpha_e$ hasStdPl, $\alpha_e$ hasSteelPlate

⇒ $\alpha_e$ hasFlatPlate, $\alpha_e$ hasStdPl, $\alpha_e$ hasSteelPlate

⇒ $\alpha_e$ hasFlatPlate, $\alpha_e$ hasStdPl, $\alpha_e$ hasSteelPlate

⇒ $\alpha_e$ hasFlatPlate, $\alpha_e$ hasStdPl, $\alpha_e$ hasSteelPlate

The role of a skill is thus to construct a plan that enables the associated primitive action to execute successfully. That is, regardless of the current environment state, each skill is in charge of constructing a plan that can fulfill all necessary conditions before its primitive action can execute successfully.

4.4.1 Reactivity to noise. Although the given scenario did not include any noise, skills would automatically adapt to latest state changes if noise were present. Note how non-terminal plans generated by a skill always include the skill itself at the plan’s last element. This rule plays a different role in the presence and absence of disrupting noise. When no noise disrupts the plan, in Equation 7, skill $\sigma_e$ hasSteelPlate generates a plan that includes itself as a skill in the last element (i.e. $\pi_e$ hasSteelPlate($s_0)$ = $\Pi_e$ hasFlatPlate, $\alpha_e$ hasStdPl, $\alpha_e$ hasSteelPlate). Typically, after all elements but last were executed (i.e. (⋅, ⋯, ⋯,
4.4.2 Benefit of the decoupled approach. Planning through delegation helps decoupling knowledge of different tasks, by formalising each skill as a separate MDP specialised for a single task. Aside from making it easier to plan for a single objective, it helps reducing redundant actions by mitigating possible action overlaps when planning for multiple effects. An example is shown in Equation 8, where the policy $\pi_{c,\text{hasStFur}}$ constructs a plan that does not include skill $\alpha_{c,\text{hasStFur}}$ even though hasStFur is one of the required conditions for $\alpha_{c,\text{hasSteelPlate}}$ (see Figure 2). This is explained by $\pi_{e,\text{hasSteelPlate}}(s_0)$ generating a plan that includes $\alpha_{c,\text{hasStFur}}$ in Equation 7. Hence when policy $\pi_{c,\text{hasStFur}}(s_2)$ is queried in Equation 8, hasStFur is already satisfied in $s_2$, and there is no need to execute $\alpha_{c,\text{hasStFur}}$ again.

5 EXPERIMENTS

The proposed planning method is now experimentally evaluated on a variety of domains of increasing complexity. We provide code for the algorithm, domains and baselines. We first describe the environments and experimental setup, then analyse the proposed method’s sensitivity to noise, and finally compare it to several baselines.

5.1 Environments

Environments are modelled as directed acyclic dependency graphs, with nodes being state features (i.e. the value 0 or 1 of each node is a feature) and edges representing actions and their success conditions. We run comparisons on five different environments with different properties, as shown in Table 1. Environments Mining, Crafting and Random are as presented in [18], where all effects turn a single state feature from 0 to 1. MiningV2 is a variant of Mining with more complex effects – consuming effects – which turn some state features from 1 back to 0. For example, the effect of crafting an iron pickaxe consumes a stick and iron, i.e. hasIronPickaxe changes from 0 to 1 and hasIron and hasStick change from 1 to 0. Lastly, we present the more complex environment Factorio, directly generated from

| Env. | Consuming effect | Ep. length | Num. actions | Num. nodes | Avg. edges per node | State space size |
|------|------------------|------------|--------------|------------|---------------------|-----------------|
| Mining | No | 40 | 22 | 22 | 2.27 ± 1.14 | 4.19 × 10^6 |
| MiningV2 | Yes | 80 | 22 | 22 | 3.18 ± 1.97 | 4.19 × 10^6 |
| Baking | No | 60 | 30 | 30 | 2.00 ± 3.73 | 1.07 × 10^9 |
| Random | No | 100 | 100 | 100 | 1.32 ± 0.71 | 1.27 × 10^10 |
| Factorio | Yes | 300 | 193 | 194 | 6.10 ± 15.45 | 2.51 × 10^58 |

Table 1: Properties of environments tested in experiments.
the crafting and construction dependency of a video game. This environment features consuming effects and even more complex effects toggling several state features simultaneously.

5.2 Experimental setup

The proposed method is experimentally compared to the following baselines. Monte-Carlo Tree Search (MCTS) [9] is a planning method that builds a (non-exhaustive) tree of possible futures, executing the action with the best long-term score (defined as a reward function of the goal state). MCTS was chosen as a baseline for its successes in robotics and RL. The amount of search is defined by a budget; experiments analyse values of 100, 1000, and 5000. The search is performed at every time-step after an action is executed. MCTS exploration constant is set to $\frac{1}{\sqrt{T}}$. Rapidly-exploring Random Trees (RRT) [24] is another sample-based planner which grows an expanding tree from starting state to free space. RRT can enhance its search by biasing it towards the goal state with low probability (0.05). In experiments, we use maximum allowed tree sizes of 1000 and 10000. The performance of RRT is bounded by the size of its search tree, where our experiments tested 1000 and 10000 as the maximum node counts. Note that a search tree is built at every time-step to re-plan for any state changes and noises. Q-Learning [25] is a RL method that requires training to enhance its performance. It is guaranteed to converge to the optimal policy, given enough time [26]. It can only handle planning for a single goal, defined in a reward function. Unlike other baselines, it does not have access to transition dynamics, although these are implicitly learned during training. We used a discount factor $\gamma = 0.99$, a learning rate of 0.1 and an $\epsilon$-greedy policy with random action probability 0.1. A reward of 0 is given for reaching the goal, and −1 rewards are given otherwise. Q-values are tabular and initialised to 0, ensuring exploration by making the agent optimistic. The method is trained for 20,000 training episodes (typically less than 1 hour). Hierarchical [18] is a hierarchical planner that constructs plans of user-defined skills, similarly to the proposed method. Planning is achieved by planning backwards from goal to state. The method, however, can only handle non-consuming effects. All experiments are averaged over 10 runs on single core 2.2GHz. We report planning success rates, i.e. number of times the agent manages to reach the given goal. Plan length and planning times are reported for successful runs only.

All planners are tested against environments that are noise-free and with noisy-transition $T(\cdot)$. A noise probability of 0.05 is chosen, which denotes the probability of one dimension inverting its value every time the agent executes an action (regardless of whether the action is valid or not).

5.3 Sensitivity to noise

We begin by analysing the effect of transition noise on the proposed planner. All environments are evaluated with transition noise changing a single state feature from 0 to 1 or 1 to 0 with different probabilities ranging from 0 to 1. Results displayed in Figure 3 show the method is almost unaffected by noise. In more complex environments, the noise is less likely to tamper state features affected by the current section of the plan than it is to disturb any other

![Figure 3: Delegate performance (median, 25% and 75% quartiles) on all domains for different levels of noise.](image)

state feature. As such, noise tends to be more beneficial in complex environments, as shown by decreased plan length. The opposite happens in easier environments. The number of successful actions also decreases, as noise effects are sometimes aligned with the plan. Planning times only mildly decrease with noise.

5.4 Results

We now compare the proposed method to baselines on all environments, with and without noise. Results are provided in Table 2. Note that RRT is not applicable in the Factorio environment because it requires a one-to-one mapping from action effect to state dimension to expand its tree. Because Q-Learning builds a mapping between state-action pairs and their Q values, it ran out of memory (64 GB) after experiencing too many different pairs (e.g. in the Factorio environment). The Hierarchical planner is not applicable to MiningV2 and Factorio because it cannot reason on environment with consuming effects.

Overall the proposed method Delegate consistently outperforms other baselines on all environments, whereas other methods are very sensitive to increasing environment complexity and noise. In particular, methods that do not reason on action conditions don’t succeed in complex environments (e.g. Random and Factorio). Q-learning achieves near-optimal plan length on Mining, as
We proposed a framework and method for planning through delegation, by dividing the sequential decision making problem into multiple non-sequential problems with one specific goal each. Each skill only plans for what it needs and when it needs it, delegating how to achieve necessary sub-goals to the most specialised skill. The presented planner generates plans on-demand, making it resilient to noisy transitions and able to adapt to latest state changes. We experimentally show it outperforms other classic planner and RL methods on a variety of environments.

As future work, the method could be extended to unknown environments. By subdividing the planning task into several action conditions, which would be learned from observing action RL methods on a variety of environments.

| T(·) prob. | Q-Learning | MCTS (100) | MCTS (1000) | MCTS (5000) | RRT (1000) | RRT (1000) | Hierarchical | Policy Delegate |
|------------|------------|------------|-------------|-------------|------------|------------|--------------|----------------|
| Mining     | Success    | 80.0%      | 30%         | 70%         | 80.0%      | 0.0%       | 40.0%        | 100%          | 100%          |
|            | Time       | 13.25 ± 0.5| 38.67 ± 1.15| 35.14 ± 4.94| 34.25 ± 4.13| –          | 32.25 ± 4.42 | 13.0 ± 0.0    | 13.0 ± 0.0    |
| MiningV2   | Success    | 50.0%      | 10%         | 10%         | 20.0%      | 60.0%      | 40.0%        | 100%          | 100%          |
|            | Time       | 13.0 ± 0.0 | 33.0 ± 0.0  | 38.0 ± 0.0  | 37.0 ± 1.41| 31.17 ± 9.30| 31.75 ± 1.89| 13.1 ± 0.3162| 12.65 ± 1.565 |
| Baking     | Success    | 0.0%       | 0%          | 0%          | 80.0%      | 100%       | 0.0%         | N/A           | 100%          |
|            | Time       | 0.024 ± 0.039| 21.94 ± 2.659| 119.9 ± 19.52| 184.4 ± 46.39| 11.01 ± 0.207| 65.71 ± 6.534| 0.007 ± 0.001| 0.028 ± 0.008 |
| Random     | Success    | 40.0%      | 30%         | 70%         | 70.0%      | 20.0%      | 50.0%        | N/A           | 100%          |
|            | Time       | 107.2 ± 68.88| 43.33 ± 4.509| 56.71 ± 22.34| 65.43 ± 12.92| 73.5 ± 9.192| 25.6 ± 23.37| –             | 14.4 ± 2.683  |
| Factorio   | Success    | 20.0%      | 40%         | 20%         | 30.0%      | 100%       | 100%         | 100%          | 100%          |
|            | Time       | 43.0 ± 46.67| 27.5 ± 21.38| 22.0 ± 15.56| 50.0 ± 13.08| 33.7 ± 13.05| 29.0 ± 1.886| 22.6 ± 8.003| 22.6 ± 5.016 |
|            | Success    | 0.0%       | 0%          | 0%          | 0%         | 0%         | 0%           | 0%            | 100%          |
|            | Time       | 1.028 ± 0.030| 206.5 ± 36.05| 1745 ± 478.9| 6564 ± 23.47| 41.82 ± 1.724| 257.5 ± 3.028| 0.084 ± 0.037| 0.104 ± 0.044 |
|            | Success    | 0.0%       | 10%         | 0%          | 0.0%       | 10.0%      | 60.0%        | N/A           | 100%          |
|            | Time       | 0.902 ± 0.175| 206.5 ± 36.05| 1745 ± 478.9| 3383 ± 269.2| 45.56 ± 2.103| 2137 ± 67.2| 0.091 ± 0.031| 0.089 ± 0.026 |
| Factorio   | Success    | 0.0%       | 10%         | 0%          | 0.0%       | N/A        | N/A          | N/A           | 100%          |
|            | Time       | 8400 ± 3913| 48570 ± 86.7| –            | –          | –          | –            | –             | –             |
|            | Success    | 0.0%       | 10%         | 0%          | 0.0%       | N/A        | N/A          | N/A           | 100%          |
|            | Time       | 68.0 ± 0.0  | –            | –            | –          | –          | –            | –             | –             |

Table 2: Experimental results from various methods and environments (μ ± σ over 10 runs)

6 CONCLUSION

We proposed a framework and method for planning through delegation, by dividing the sequential decision making problem into multiple non-sequential problems with one specific goal each. Each skill only plans for what it needs and when it needs it, delegating how to achieve necessary sub-goals to the most specialised skill.

The problem dimension is small enough. However, Q-learning is quickly overwhelmed by increasing problem sizes, and it does not converge in the allocated training time or even runs out of memory. Similar results are observed in MCTS and RRT, however, the solutions obtained are far from optimal. The two methods also requires explicit re-planning at each time-step to account for state changes and noise. As a result, they require lengthy runtime to complete a single episode. Hierarchical is able to achieve near-optimal result in most applicable environments, and its execution speed is slightly lower than Delegate. However, in contrast to Delegate it is susceptible to noise and complex environments as the methods plans out a horizon of action sequence, which could be invalidated by environment noise.

REFERENCES

[1] R. E. Fikes, and N. J. Nilsson, Strips: A new approach to the application of theorem proving to problem solving, Artificial Intelligence, 1971.

[2] E. D. Sacerdoti, “The nonlinear nature of plans,” Tech. Rep., 1975.

[3] K. Erol, “Hierarchical task network planning: Formalization, analysis, and implementation,” Ph.D. dissertation, 1996.
[4] K. Erol, J. A. Hendler, and D. S. Nau, Umcp: A sound and complete procedure for hierarchical task-network planning, Proceedings of the International Conference on AI Planning Systems, 1994.

[5] D. Nau, Y. Cao, A. Lotem, and H. Munoz-Avila, Shop: Simple hierarchical ordered planner, in International Joint Conference on Artificial Intelligence, 1999.

[6] N. Nejati, P. Langley, and T. Konik, Learning hierarchical task networks by observation, in International Conference on Machine Learning, 2006.

[7] R. S. Sutton, and A. G. Barto, Reinforcement learning: An introduction. MIT press, 2018.

[8] R. S. Sutton, Dyna, an integrated architecture for learning, planning, and reacting, ACM SIGART Bulletin, 1991.

[9] R. Coulom, Efficient selectivity and backup operators in monte-carlo tree search, in International Conference on Computers and Games, 2006.

[10] G. Williams, N. Wagener, B. Goldberg, P. Drews, J. M. Rehg, B. Boots, and E. A. Theodorou, Information theoretic mpc for model-based reinforcement learning, in 2017 IEEE International Conference on Robotics and Automation (ICRA), IEEE, 2017, 1714–1721.

[11] R. S. Sutton, D. Precup, and S. Singh, Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning, Artificial Intelligence, 1999.

[12] T. G. Dietterich, Hierarchical reinforcement learning with the maxq value function decomposition, Journal of Artificial Intelligence Research, 2000.

[13] Ö. Şimşek, A. P. Wolfe, and A. G. Barto, Identifying useful subgoals in reinforcement learning by local graph partitioning, in Proceedings of the 22nd International Conference on Machine Learning, ACM, 2005, 816–823.

[14] B. Bakker, and J. Schmidhuber, Hierarchical reinforcement learning based on subgoal discovery and subpolicy specialization, in Conference on Intelligent Autonomous Systems, 2004.

[15] M. Grounds, and D. Kudenko, Combining reinforcement learning with symbolic planning, in Adaptive Agents and Multi-Agent Systems, 2008.

[16] M. van Otterlo, A survey of reinforcement learning in relational domains, CTIT Technical Report Series, no. 05-31, 2005.

[17] C. Diuk, A. Cohen, and M. L. Littman, An object-oriented representation for efficient reinforcement learning, in International Conference on Machine Learning, 2008.

[18] P. Morere, L. Ott, and F. Ramos, Learning to plan hierarchically from curriculum, IEEE Robotics and Automation Letters, 2019.

[19] E. S. Raymond, The Art of Unix Programming. Addison-Wesley Professional, 2003.

[20] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire, The nonstochastic multiarmed bandit problem, SIAM journal on computing, vol. 32, no. 1, 48–77, 2002.