2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
October 14-18, 2024. Abu Dhabi, UAE

Do One Thing and Do It Well: Delegate Responsibilities in Classical Planning

Tin Lai[†,§,1] and Philippe Morere[†,2]

***Abstract*— We propose a novel framework and algorithm for solving classical planning problems with an implicit hierarchical solver based on the principle of delegation. This framework, the *Markov Intent Process*, features a collection of skills that 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. Our method performs robustly in noisy environments with non-deterministic action effects and 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 environmental 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.**

I. 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 for 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. Moreover, in classical planning problems where the hierarchical structure is not given, existing planners typically are not aware of how to exploit the underlying structure to compose reusable skills. Knowledge of each skill's purpose is also extremely important, as it allows seamless 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 an implicit hierarchical planning methodology for reasoning about the effects of skills and primitive actions, yielding several benefits. Planning using skill effect knowledge allows one to select the best skill for any given task directly, reducing planning time and computation by exploiting the implicit hierarchical structure. This knowledge also enables planners to reason about whether executed skills or actions were successful by comparing the expected and observed effects; this allows inferring action success conditions directly from interactions. Furthermore, the presented method plans at the highest level only and expands plans into more detailed plans *on-demand*. Thus, the 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 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. This framework advances solving classical planning by structuring the exploiting of task hierarchies as skills and plans. MIP can operate in noisy environments where states are changed by exogenous forces, which implies action effects are non-deterministic. 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 *PolicyDelegate*—based on the MIP framework. Finally, we experimentally show *PolicyDelegate* 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.

II. Related Works

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,

[†]School of Computer Science, The University of Sydney, Australia. [§]Fait Corporation, Australia. [1]tin.lai@faitcorp.com [2]philippe.morere@sydney.edu.au

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], [3]. More work has aimed to extend HTN planners to learn the skill hierarchy automatically [4], which requires experts to demonstrate. HTN can also produce plans on multiple levels of abstraction [5] and handle uncertainty in action [6]. Planning ultimately enables autonomous agents to reason on a higher abstract level [7].

The field of reinforcement learning (RL) has also produced much work on sequential decision-making [8], mostly focusing on cases with unknown transition dynamics. Classic planners include Monte-Carlo Tree Search [9] or model predictive control [10] to simulate possible futures. RL had greatly benefited from the concept of hierarchical planning [11], [12]. The option framework [11] defines temporally extended actions called *options*, which can form hierarchical plans. 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 are highly dependent on state space geometry [13], desired state space manipulations (i.e. skill effects) are always known in advance, which encourages encapsulation.

Method to learn options by identifying transition data clusters was proposed in [14], though it is limited to a few levels of hierarchy. Useful skills can also be identified through repeated interaction with the environment [15]. Symbolic planning and reinforcement learning have also been combined in [16], where the RL reward function is generated by STRIPS. 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. Tree-based planning methods [17]–[19] is another predominant class of planning methods, which utilise random sampling to construct graphs that can be reused [20]. However, most of these approaches focus on kinematic-aspect or kinodynamic-aspect [21] planning and do not perform task decomposition.

Most similar to this work is that of [22], which learns a hierarchy of skills using an expert-generated curriculum to guide learning, and [23] where HTN-like solutions are used to tackle classical planning. 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 with reasoning over action effects and conditions. Unlike previously mentioned work, our proposed method automatically delegates planning to the most specialised skill for the task, leading to faster and more efficient planning.

III. Formulation

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

Definition 1 (Markov Intent Process): A Markov Intent Process (MIP) is a tuple $\langle \mathcal{S}, \mathcal{A}, \hat{\mathcal{A}}, E, T, \mathcal{C} \rangle$ composed of a set of states $\mathcal{S}$, a set of learnable (and initially empty) skills $\mathcal{A}$, a set of primitive actions $\hat{\mathcal{A}}$ with known effects $e \in E$, a exogenous transition noise function T, and a set $\mathcal{C}$ 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 $\hat{a} \in \hat{\mathcal{A}}$.

Algorithm 1: Episode in MIP environment

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

Generate intent plan $\upsilon_0 = (\alpha_{e_0}, \ldots, \alpha_{e_i})$ from s_0 to s_g
for $t \leftarrow 1$ **to** N **do**
  $\hat{a}_e, \upsilon_t \leftarrow \texttt{GetAction}(\upsilon_{t-1}, s_{t-1})$
  if $s_{t-1} \in S_{\hat{a}_e}$ **then** ▷ action can execute
    $s_t \leftarrow s_{t-1} \oplus e$ ▷ e is effect of $\hat{a}$
  $s_t \leftarrow T(s_t)$ ▷ noisy environment only
  if $s_t = s_g$ **then**
    return *success*
return *failed*

Definition 2 (Primitive action): Each primitive action $\hat{a} \in \hat{\mathcal{A}}$ is associated with a set of states $\mathcal{S}_{\hat{a}} \in \mathcal{C}$, in which executing $\hat{a}$ would be successful. Successfully executing $\hat{a}$ in a state in $\mathcal{S}_{\hat{a}}$ 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 $\alpha_e \in \mathcal{A}$, identified by its effect e, is defined as a tuple $\alpha_e := (e, \pi_{\alpha_e})$ of (i) a *known* intended effect e, describing the effect of the skill if α_e is successfully executed (and no noise is observed); and (ii) a learnable policy π_{α_e} mapping a state in $\mathcal{S}$ to a plan υ. 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 $\mathcal{A}$ is *not* given to MIP agents, which need to discover and learn them.

Definition 4 (Noisy Transition): The transition function $T : \mathcal{S} \rightarrow \mathcal{S}$ applies to the current state s at every iteration, which will transit s to some (possibly same) state $s' \in \mathcal{S}$. An agent has full observability of the current state; however, it has no knowledge of T and does not know whether the state transition is due to T or its own actions. This leads to non-deterministic action effects as T can corrupt the necessary state conditions for actions to be successful.

The objective of the MIP problem is to successfully execute some target action $\hat{a}_{\text{target}}$. Since all actions have conditions that are defined by states where $\hat{a}_{\text{target}}$ would be successful (definition 2), the objective is equivalent to reaching some s in $\mathcal{S}_{\hat{a}_{\text{target}}} \in \mathcal{C}$. The distinction between primitive actions $\hat{a}$ and skills α is that primitive actions are given by the problem, often associated with a given list of actions that the agent can perform (with certain preconditions). Skills refer to a composition of several primitive actions (i.e., a learnable policy that associates certain actions with some desirable outcome). A skill α_e learns a policy $\pi_{\alpha_e}(s)$ that returns a plan by conditioning on the current state s, where a plan is an ordered list of actions and/or skills.

Definition 5 (plan): A plan υ, generated by a policy $\pi_{\alpha_e}(s)$, is a (potentially empty) sequence of skills $\alpha \in \mathcal{A}$

and/or actions $\hat{a} \in \hat{\mathcal{A}}$ 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 $\pi_e \equiv \pi_{\alpha_e}$.

Each skill α from the set of skills $\mathcal{A}$ 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. $s \in \mathcal{S}_{\hat{a}}$. Otherwise, a skill's policy should return a plan composed of other skills whose effects are necessary before the skill's effect can 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 query other skills, which in turn return another nested plan to be executed.

A primitive action $\hat{a}$ with effect e can modify the property of state s into a different state in $\mathcal{S}$. We say that an action is successful if $s \in \mathcal{S}_{\hat{a}}$ (action condition fulfilled) and failed otherwise. In addition, the transition noise function T is always applied before the agent can observe the outcome of its action, where T potentially corrupts the resulting state s with noise. We can write it as $s' = T(s \oplus e)$ being the observed state where $s' \in \mathcal{S}$ and $\oplus$ denotes applying effect e to state s. A concrete example is that an action can represents *pressing a button on a vending machine*; conditions represents *a coin being inserted beforehand*; effects represents *the vending machine outputting a snack*; and the exogenous noise T represents *the vending machine malfunctioned* (hence did not output the desire product). The set of successful state spaces for all actions $\mathcal{C}$ and transition noise function T are not necessarily known. Thus, MIP agents need to be reactive to environmental changes.

A. Action Success Conditions

MIP generalises the definition of the success conditions of an action $\hat{a}$ as the set of states $\mathcal{S}_{\hat{a}}$ 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 $\phi \colon \mathcal{S} \to \mathbb{R}$ 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 $P^j_\phi(s)$ to denote a proposition $P^j \in \mathcal{P}$ operating on state feature ϕ in state s.

Definition 6 (Action condition): The condition of action $\hat{a} \in \hat{\mathcal{A}}$ is defined as the minimal non-empty set of $d \leq m$ feature propositions $\zeta_{\hat{a}} = \{ P^{j_1}_{\phi_{i_1}}(\cdot), \ldots, P^{j_d}_{\phi_{i_d}}(\cdot) \}$, where $i_1, \ldots, i_d$ and $j_1, \ldots, j_d$ are all distinct, such that

$$\bigwedge_{P^j_{\phi_i}(\cdot) \in \zeta_{\hat{a}}} P^j_{\phi_i}(s) = \begin{cases} true & \text{if } s \in \mathcal{S}_{\hat{a}}, \\ false & \text{if } s \in \mathcal{S} \setminus \mathcal{S}_{\hat{a}}. \end{cases} \tag{1}$$

Intuitively, action condition $\zeta_{\hat{a}}$ from definition 6 defines the necessary state features for action $\hat{a}$ to be successful in s.

B. 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 7 (Success prerequisite): We say that $\hat{a}_i$ is a *success prerequisite* of action $\hat{a}_j$ in $s \notin \mathcal{S}_{\hat{a}_j}$ if $\hat{a}_i$ is a necessary action in all possible action sequences that can transit s to $s' \in \mathcal{S}_{\hat{a}_j}$.

Definition 8 (Well-formed action set): An action set $\hat{\mathcal{A}}$ is said to be well-formed if it contains no ill-formed action; that is, no circular dependencies occur when unrolling conditions of $\hat{a} \in \hat{\mathcal{A}}$. Formally, $\hat{a}$ is ill-formed if there exists a state $s \in \mathcal{S} \setminus \mathcal{S}_{\hat{a}}$ for which $\hat{a}$ itself is one of the success prerequisites.

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.

IV. Planning Through Delegation

Our proposing planner, named *PolicyDelegate*, takes a classical planning problem as an input and outputs an HTN-like problem such that any solution to that HTN problem is also a solution to the classical problem. *PolicyDelegate* operates by imposing an implicit hierarchical structure and forces the planner to delegate responsibilities to specialised skills for achieving their respective sub-goals.

A. 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"* [24]. 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 the latest state changes (due to noise, for example) and greatly reduces planning time.

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

Contrary to the MDP framework, no reward function is defined; thus, solving an MIP cannot be defined as maximising rewards. Instead, planning in an MIP requires a goal state $s_g \in \mathcal{S}_g$ to be specified. Solving a MIP optimally is equivalent to finding an optimal plan for all goal states in $\mathcal{S}_g$. 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 υ_e^{*H} is

$$\upsilon_e^{*H} = \arg\min_{\upsilon_e^H} \| \upsilon_e^H \|. \tag{2}$$

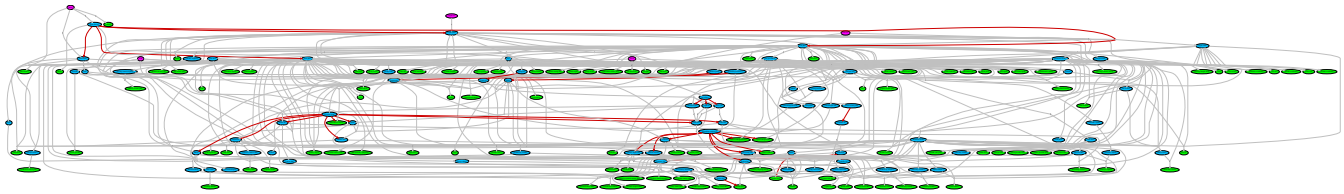

Fig. 1: Full environment dynamics of the *Factorio* domain, representing dependency between actions and states. Magenta nodes denote actions that have no conditions (i.e. the only entry point in the environment if the initial state is all zeroes), blue nodes denote intermediate nodes, and green nodes denote leaves in the graph (i.e. possible goals in the environment). Grey edges denote actions with consuming effects (e.g. in Figure 2, executing the action `makeStoneFurance` will cancel state feature *hasStone* because it is used as material), and red edges denote actions without such an effect (e.g. executing action `makeStoneBrick` will not cancel *hasStoneFurance* because it is a tool).

The planning horizon H restricts the length of all plans within υ^H (and υ^H itself) to be no greater than H, i.e $\mid \upsilon_e^H \mid <= H$ and all plans within υ_e^H have horizon H. For some short horizons and goals, a valid υ_e^H may not exist.

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

C. Delegate

We now describe how *PolicyDelegate* leverages skill success conditions to find effective plans efficiently. 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 about the effects and conditions of the available primitive actions. From here onward, we consider the case where conditions are known.

Planning in a MIP for a given goal state $s_g \in \mathcal{S}_g$ from a starting state $s_0 \in \mathcal{S}_0$ 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 a sequence of effects must exist if the action space is complete. An initial plan υ_0 for task (s_0, s_g) is simply generated by retrieving the skills corresponding to these effects, $\upsilon_0 = (\alpha_{e_0}, \ldots, \alpha_{e_i})$. This procedure happens whenever planning starts with a new goal state, as described in Alg. 1. This initial plan, called *intent plan*, only contains high-level instructions; executing it will require further planning, which can be computed on demand.

D. Skill delegation policy

Once an intent plan υ_0 is formed, generating actions to execute in specific states is achieved by querying the policies of skills within υ_0. If the first element of υ_0 is a primitive action, there's no need to call any skill policy, and the action can be executed directly. Otherwise, if the first element of υ_0 is a skill α, then its delegation policy π_α is queried at the current state s_t. The resulting plan $\upsilon = \pi_\alpha(s_t)$ is appended to the beginning of the intent plan υ_0 (after α 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 2: Delegate algorithm

```
function GetAction(υ, s_t)
    x, υ_rest ← separate first element of υ
    if x ∈ Â then
        return x, υ_rest
    else                                              ▷ x ∈ A
        υ_nested ← DelegatePolicy(x, s_t)
        υ ← υ_nested : υ_rest                         ▷ concatenation
        return GetAction(υ, s_t)
function DelegatePolicy(α_e, s_t)
    ζ_t ← { P^j_φi(·) ∈ ζ_â_e | ¬P^j_φi(s_t) }        ▷ unmet conditions
    if ζ_t = ∅ then
        return (â_e)                                  ▷ terminal plan
    else
        υ ← ∅
        for P^j_φi ∈ ζ_t do                           ▷ for all unmet conditions
            α_ej ← skill with effect that satisfies P^j_φi
            υ ← υ : α_ej                              ▷ concatenation
        return υ
```

Generating a plan by querying a skill policy is achieved by analysing the success conditions $\zeta_{\hat{a}_e}$ of the skill's primitive action $\hat{a}_e$. The procedure is described in `DelegatePolicy`, Algorithm 2. From Definition 6, the set of conditions unmet in the current state can be computed as

$$\zeta_t = \left\{ P^j_{\phi_i}(\cdot) \in \zeta_{\hat{a}_e} \;\middle|\; \neg P^j_{\phi_i}(s_t) \right\}.$$

If ζ_t is not empty, then we can construct a plan composed of skills whose effects would satisfy each of the individual elements ζ_t. This plan is returned as the evaluation of the skill's policy in the current state. In the event of ζ_t being an empty set, a terminal plan composed only of the skill's primitive action $\hat{a}_e$ is returned.

Definition 9 (Terminal plan): A plan υ is called a terminal plan if it contains no skills, i.e. $\forall x \in \upsilon, x \notin \mathcal{A}$. Empty plans

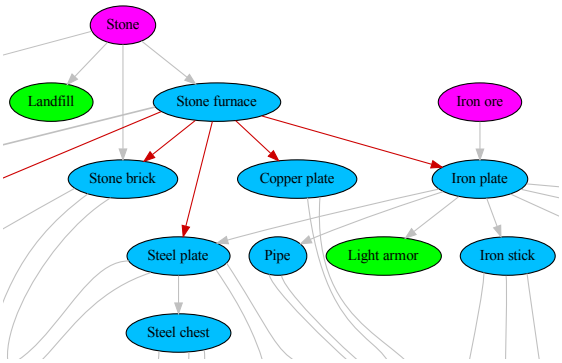

Fig. 2: Cropped sub-graph of the *Factorio* domain. Each node denotes a state and an action that has a positive effect on that state; incoming edges of a node denote the conditions of that action. For example, the action `makeSteelPlate` is conditioned on *hasSteelPlate* being zero, and *hasStoneFurnace*, *hasIronPlate* being one; the action effect is cancelling state dimension *hasIronPlate* and satisfying *hasSteelPlate*. State dimension *hasSteelPlate* is, in turn, conditioned by other actions such as `makeSteelChest`.

where $|\upsilon| = 0$ 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.

E. Planning termination

Let us now analyse conditions under which planning terminates. As per Definition 8, a well-formed problem should contain no circular dependency. However, a policy for skill α could wrongly return a plan that contains skills that would generate plans containing α itself. We call this plan *ill-formed*, as it would result in circular dependencies.

Definition 10 (Ill-formed plan): A plan υ is said to be ill-formed if it contains skills that are ancestors of υ. Formally, we define the direct parent of all skills within υ as skill α_j, such that plan υ was generated in some state s_i as $\upsilon = \pi_{\alpha_j}(s_i)$. Then υ is *ill-formed* iff $\forall \alpha \in \text{ancestors}(\upsilon), \exists \alpha \in \upsilon$, where ancestors are all parents of parents until the intent plan υ_0, and the parent of a plan is equivalent to parent the skill that generated it.
In practice, preventing ill-formed plans 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 eventually be generated, hence ensuring planning termination.

F. Analysis of a concrete example

We now provide a concrete example of planning through delegation, illustrating the properties of *PolicyDelegate*.

In the following, we will use *hasState* to denote the presence of a state feature of interest, $\alpha_{e:\textit{hasState}}$ to denote a skill with intended effect satisfying *hasState*, and $\hat{a}_{\texttt{doAction}}$ to denote a primitive action with specified outcome. We consider part of the Factorio environment, represented as a graph in Figure 2. Skill $\alpha_{e:\textit{hasIronPlate}}$ must execute action $\hat{a}_{\texttt{makeIronPlate}}$ to achieve its intended effect, however the action requires both state features *hasIronOre* and *hasStoneFurnace*. Therefore, skill $\alpha_{e:\textit{hasIronPlate}}$ should form a plan that involves skills $\alpha_{e:\textit{hasIronOre}}$ and $\alpha_{e:\textit{hasStoneFurnace}}$, planning for effects activating missing state features. Since $\hat{a}_{\texttt{getIronOre}}$ has no conditions, the policy of skill $\alpha_{e:\textit{hasIronOre}}$ will return a terminal plan $\upsilon_{\alpha_{e:\textit{hasIronOre}}} = (\hat{a}_{\texttt{getIronOre}})$, which can be executed directly. However, $\hat{a}_{\texttt{makeStoneFurnace}}$ is conditioned on *hasStone* and hence, skill $\alpha_{e:\textit{hasStoneFurnace}}$ should in turn generate a nested plan for $\alpha_{e:\textit{hasStone}}$.

A possible plan execution scenario for $\upsilon_{e:\textit{hasSteelPlate}}$ 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.

$$\underline{\pi_{e:\textit{hasSteelPl}}(s_0)} \tag{3}$$
$$\hookrightarrow(\underline{\alpha_{e:\textit{hasStFur}}}, \alpha_{e:\textit{hasIrPl}}, \alpha_{e:\textit{hasSteelPl}})$$
$$\hookrightarrow(\underline{\pi_{e:\textit{hasStFur}}(s_0)}, \alpha_{e:\textit{hasIrPl}}, \alpha_{e:\textit{hasSteelPl}})$$
$$\hookrightarrow(\underline{(\alpha_{e:\textit{hasSt}}, \alpha_{e:\textit{hasStFur}})}, \alpha_{e:\textit{hasIrPl}}, \alpha_{e:\textit{hasSteelPl}})$$
$$\hookrightarrow(\underline{(\pi_{e:\textit{hasSt}}(s_0)}, \alpha_{e:\textit{hasStFur}}), \alpha_{e:\textit{hasIrPl}}, \alpha_{e:\textit{hasSteelPl}})$$
$$\hookrightarrow(((\hat{\boldsymbol{a}}_{\texttt{getSt}}), \underline{\pi_{e:\textit{hasStFur}}(s_1)}), \alpha_{e:\textit{hasIrPl}}, \alpha_{e:\textit{hasSteelPl}})$$
$$\hookrightarrow(((\hat{\boldsymbol{a}}_{\texttt{getSt}}), (\hat{\boldsymbol{a}}_{\texttt{mkStFur}})), \underline{\pi_{e:\textit{hasIrPl}}(s_2)}, \alpha_{e:\textit{hasSteelPl}}) \tag{4}$$
$$\hookrightarrow(((\hat{\boldsymbol{a}}_{\texttt{getSt}}), (\hat{\boldsymbol{a}}_{\texttt{makeStFur}})), \underline{(\alpha_{e:\textit{hasIrOre}}, \alpha_{e:\textit{hasIrPl}})}, \alpha_{e:\textit{hasSteelPl}})$$
$$\hookrightarrow(((\hat{\boldsymbol{a}}_{\texttt{getSt}}), (\hat{\boldsymbol{a}}_{\texttt{mkStFur}})), (\underline{\pi_{e:\textit{hasIrOre}}(s_2)}, \alpha_{e:\textit{hasIrPl}}), \alpha_{e:\textit{hasSteelPl}})$$
$$\hookrightarrow(((\hat{\boldsymbol{a}}_{\texttt{getSt}}), (\hat{\boldsymbol{a}}_{\texttt{mkStFur}})), ((\hat{\boldsymbol{a}}_{\texttt{getIrOre}}), \underline{\pi_{e:\textit{hasIrPl}}(s_3)}), \alpha_{e:\textit{hasSteelPl}})$$
$$\hookrightarrow(((\hat{\boldsymbol{a}}_{\texttt{getSt}}), (\hat{\boldsymbol{a}}_{\texttt{mkStFur}})), ((\hat{\boldsymbol{a}}_{\texttt{getIrOre}}), (\hat{\boldsymbol{a}}_{\texttt{mkIrPl}})), \underline{\pi_{e:\textit{hasSteelPl}}(s_4)})$$
$$\hookrightarrow(((\hat{\boldsymbol{a}}_{\texttt{getSt}}), (\hat{\boldsymbol{a}}_{\texttt{mkStFur}})), ((\hat{\boldsymbol{a}}_{\texttt{getIrOre}}), (\hat{\boldsymbol{a}}_{\texttt{mkIrPl}})), (\hat{\boldsymbol{a}}_{\texttt{mkSteelPl}}))$$

The element that is being unrolled at each line is underlined, the $\hookrightarrow$ symbol denotes one possible policy, and the executed primitive action is 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+1}. 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.

Thus, the role of a skill is 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 fulfils all necessary conditions before its primitive action can be executed successfully.

G. Reactivity to noise

Although the given scenario did not include any noise, skills would automatically adapt to the latest state changes if the 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 3, skill $\alpha_{e:\textbf{\textit{hasSteelPl}}}$ generates

a plan that includes itself as a skill in the last element (i.e. $\pi_{e:\textit{hasSteelPl}}(s_0) = (\alpha_{e:\textit{hasStFur}}, \alpha_{e:\textit{hasIrPl}}, \boldsymbol{\alpha_{e:\textit{hasSteelPl}}})$). Typically, after all elements but the last one was executed (i.e. $(\ldots, \ldots, \alpha_{e:\textit{hasSteelPl}})$), the last skill generates a terminal plan composed of its primitive action if all of its conditions are fulfilled at this stage (as is the case for all of the skills in the concrete example shown).

In the event of noise corrupting states, plans are dynamically adapted on the fly to correct or even take advantage of the noise. Suppose state dimension *hasStFur* was fulfilled but subsequently cancelled by noise before the plan finished executing. Then $s'_4 = T(s_4)$, and the executing $\pi_{e:\textit{hasSteelPl}}$ in s'_4 results in plan $(\alpha_{e:\textit{hasStFur}}, \alpha_{e:\textit{hasSteelPl}})$ because $\pi_{e:\textit{hasSteelPl}}$ automatically corrects the plan to fulfil the cancelled state feature *hasStFur*. This property is also very valuable when noise enables one of the state features required to execute the plan. In this case, parts of the plan would be skipped altogether, executing only actions that would succeed.

H. 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 reduce redundant actions by mitigating possible action overlaps when planning for multiple effects. An example is shown in Equation 4, where the policy $\pi_{e:\textit{hasIrPl}}$ constructs a plan that does not include skill $\alpha_{e:\textit{hasStFur}}$ even though *hasStFur* is one of the required conditions for $\alpha_{e:\textit{hasIrPl}}$ (see Figure 2). This is explained by $\pi_{e:\textit{hasSteelPl}}(s_0)$ generating a plan that includes $\alpha_{e:\textit{hasStFur}}$ in Equation 3. Hence when policy $\pi_{e:\textit{hasIrPl}}(s_2)$ is queried in Equation 4, *hasStFur* is already satisfied in s_2, and there is no need to execute $\alpha_{e:\textit{hasStFur}}$ again.

V. 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. 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. Environments *Mining*, *Crafting*, and *Random* are as presented in [22], 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 the crafting and construction dependency of a video game, which features *consuming effects* and more complex effects are toggling several state features simultaneously.

We now compare the proposed method to baselines on all environments, with and without noise. Results are provided in Table I. Note that RRT is not applicable in the Factorio environment because it requires a one-to-one mapping from the action effect to the 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 the environment with consuming effects.

Overall, the proposed method *PolicyDelegate* 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 a near-optimal plan length for mining, as the problem dimension is small enough. However, increasing problem sizes quickly overwhelm Q-learning, 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 require explicit re-planning at each time step to account for state changes and noise. As a result, they require a lengthy runtime to complete a single episode. *Hierarchical* is able to achieve near-optimal results in most applicable environments, and its execution speed is slightly lower than *PolicyDelegate*. However, in contrast to *PolicyDelegate*, it is susceptible to noise and complex environments as the methods plan out a horizon of action sequence, which could be invalidated by environment noise.

VI. Conclusion

We proposed a MIP framework and method for planning and reasoning through delegation. We divided 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 the latest state changes. We experimentally show it outperforms other classic planner and RL methods in a variety of environments.

The proposed MIP framework is helpful in addressing the high dimensionality and sparse reward issue in RL methods. Rather than trying to learn a perfect policy that directly brings the agent from the starting state to its target state, the idea of *PolicyDelegate* is to decompose the policy and learn smaller but computationally tractable policies that are good at a dedicated skill. Each skill is responsible for one desired outcome, and when those outcomes are needed, the higher-level policy will hierarchically influence other well-trained policies.

For future work, the method would be extended to unknown action conditions, which would be learned from observing action successes and failures. By subdividing the planning task into several simpler problems, learning spare conditions would be relatively easier. Lastly, a model-free approach based on classic contextual bandits could also be considered,

	$T(\cdot)$ prob.		Q-Learning	MCTS (100)	MCTS (1000)	MCTS (5000)	RRT (1000)	RRT (10000)	Hierarchical	PolicyDelegate
Mining	0.00	Success	80.0%	30%	70%	80.0%	0.0%	40.0%	**100%**	**100%**
		$\| v \|$	13.25 ± 0.5	38.67 ± 1.155	35.14 ± 4.947	34.25 ± 4.132	–	32.25 ± 4.425	$\mathbf{13.0 \pm 0.0}$	$\mathbf{13.0 \pm 0.0}$
		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	$\mathbf{0.007 \pm 0.001}$	0.028 ± 0.008
	0.05	Success	50.0%	10%	10%	20.0%	60.0%	40.0%	**100%**	**100%**
		$\| v \|$	13.0 ± 0.0	33.0 ± 0.0	38.0 ± 0.0	37.0 ± 1.414	31.17 ± 9.304	31.75 ± 1.893	13.1 ± 0.3162	$\mathbf{12.65 \pm 1.565}$
		Time	0.023 ± 0.024	21.94 ± 2.659	119.9 ± 19.52	80.73 ± 10.28	9.862 ± 2.325	72.44 ± 10.77	$\mathbf{0.009 \pm 0.002}$	0.031 ± 0.016
MiningV2	0.00	Success	0.0%	90%	**100%**	**100%%**	0.0%	0.0%	N/A	**100%**
		$\| v \|$	–	47.33 ± 11.03	34.2 ± 5.051	37.1 ± 8.212	–	–	–	$\mathbf{15.0 \pm 0.0}$
		Time	0.131 ± 0.051	55.79 ± 15.36	241.1 ± 86.49	362.4 ± 97.02	21.25 ± 0.885	136.8 ± 0.486	–	$\mathbf{0.028 \pm 0.007}$
	0.05	Success	40.0%	30%	70%	70.0%	20.0%	50.0%	N/A	**100%**
		$\| v \|$	107.2 ± 68.88	43.33 ± 4.509	56.71 ± 22.34	65.43 ± 12.92	73.5 ± 9.192	25.6 ± 23.37	–	$\mathbf{14.4 \pm 2.683}$
		Time	0.080 ± 0.031	55.79 ± 15.36	241.1 ± 86.49	215.7 ± 60.7	21.63 ± 1.193	91.95 ± 57.03	–	$\mathbf{0.041 \pm 0.021}$
Baking	0.00	Success	0.0%	0%	70%	80.0%	**100%**	**100%**	**100%**	**100%**
		$\| v \|$	–	–	49.86 ± 3.132	47.62 ± 5.012	37.2 ± 3.706	28.2 ± 0.919	$\mathbf{24.0 \pm 0.0}$	$\mathbf{24.0 \pm 0.0}$
		Time	0.101 ± 0.012	32.27 ± 14.08	178.3 ± 56.21	943.3 ± 219.3	10.93 ± 1.042	53.93 ± 2.449	$\mathbf{0.012 \pm 0.002}$	0.047 ± 0.009
	0.05	Success	20.0%	40%	20%	30.0%	**100%**	**100%**	**100%**	**100%**
		$\| v \|$	43.0 ± 46.67	27.5 ± 21.38	$\mathbf{22.0 \pm 15.56}$	50.0 ± 13.08	33.7 ± 13.05	29.0 ± 1.886	22.6 ± 8.003	22.6 ± 5.016
		Time	0.075 ± 0.036	32.27 ± 14.08	178.3 ± 56.21	138.6 ± 20.78	9.986 ± 3.943	57.45 ± 4.622	$\mathbf{0.017 \pm 0.006}$	0.052 ± 0.019
Random	0.00	Success	0.0%	0%	0%	0.0%	0.0%	0.0%	70.0%	**100%**
		$\| v \|$	–	–	–	–	–	–	66.57 ± 23.02	$\mathbf{39.0 \pm 0.0}$
		Time	1.028 ± 0.003	206.5 ± 36.05	1745 ± 478.9	6564 ± 23.47	41.82 ± 1.724	275.7 ± 3.028	$\mathbf{0.084 \pm 0.037}$	0.104 ± 0.044
	0.05	Success	0.0%	10%	10%	0.0%	0.0%	10.0%	60.0%	**100%**
		$\| v \|$	–	47.0 ± 0.0	$\mathbf{25.0 \pm 0.0}$	–	–	9.0 ± 0.0	63.33 ± 15.6	35.9 ± 9.457
		Time	0.902 ± 0.175	206.5 ± 36.05	1745 ± 478.9	3383 ± 269.2	45.56 ± 2.103	213.7 ± 67.2	0.091 ± 0.031	$\mathbf{0.089 \pm 0.026}$
Factorio	0.00	Success	Out of mem.	0%	0%	0.0%	N/A	N/A	N/A	**100%**
		$\| v \|$	–	–	–	–	–	–	–	$\mathbf{234.0 \pm 0.0}$
		Time	–	1345 ± 361.5	12470 ± 817.4	48570 ± 86.7	–	–	–	$\mathbf{0.348 \pm 0.156}$
	0.05	Success	Out of mem.	10%	0%	0.0%	N/A	N/A	N/A	**100%**
		$\| v \|$	–	68.0 ± 0.0	–	–	–	–	–	$\mathbf{219.4 \pm 44.82}$
		Time	–	1345 ± 361.5	12470 ± 817.4	34900 ± 427.6	–	–	–	$\mathbf{0.315 \pm 0.155}$

TABLE I: Experimental results from various methods and environments ($\mu \pm \sigma$ over 10 runs)

which would reformulate the problem under the learning context.

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," PhD thesis, 1996.

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

[5] B. Marthi, S. J. Russell, and J. A. Wolfe, "Angelic hierarchical planning: Optimal and online algorithms.," in *ICAPS*, 2008.

[6] U. Kuter, D. Nau, M. Pistore, and P. Traverso, "Task decomposition on abstract states, for planning under nondeterminism," *Artificial Intelligence*, vol. 173, no. 5-6, pp. 669–695, 2009.

[7] T. Y. Lai, "Robot learning and planning with a probabilistic perspective," PhD thesis, The University of Sydney, 2023.

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

[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. Goldfain, 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.

[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.

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

[15] T. Lai, "Discover Life Skills for Planning as Bandits via Observing and Learning How the World Works," in *IEEE/RSJ Proceedings of The International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2022.

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

[17] T. Lai, F. Ramos, and G. Francis, "Balancing Global Exploration and Local-Connectivity Exploitation with Rapidly-Exploring Random disjointed-Trees," in *Proceedings of The International Conference on Robotics and Automation*, 2019.

[18] T. Lai, W. Zhi, T. Hermans, and F. Ramos, "Parallelised diffeomorphic sampling-based motion planning," in *5th Annual Conference on Robot Learning*, 2021.

[19] T. Lai and F. Ramos, "Adaptively Exploits Local Structure With Generalised Multi-Trees Motion Planning," *IEEE Robotics and Automation Letters*, vol. 7, no. 2, pp. 1111–1117, 2021.

[20] T. Lai and F. Ramos, "LTR*: Rapid Replanning in Executing Consecutive Tasks with Lazy Experience Graph," in *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2022.

[21] T. Lai, W. Zhi, T. Hermans, and F. Ramos, "Neural Kinodynamic Planning: Learning for Kinodynamic Tree Expansion," in *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2024.

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

[23] L. De Silva, L. Padgham, and S. Sardina, "Htn-like solutions for classical planning problems: An application to bdi agent systems," *Theoretical Computer Science*, vol. 763, pp. 12–37, 2019.

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