Learning Hierarchical Policies by Iteratively Reducing the Width of Sketch Rules

Dominik Drexler[1], **Jendrik Seipp**[1], **Hector Geffner**[2,1]

[1] Linköping University, Linköping, Sweden
[2] RWTH Aachen University, Aachen, Germany
{dominik.drexler, jendrik.seipp}@liu.se, hector.geffner@ml.rwth-aachen.de

Abstract

Hierarchical policies are a key ingredient of intelligent behavior, expressing the different levels of abstraction involved in the solution of a problem. Learning hierarchical policies, however, remains a challenge, as no general learning principles have been identified for this purpose, despite the broad interest and vast literature in both model-free reinforcement learning and model-based planning. In this work, we introduce a principled method for learning hierarchical policies over classical planning domains, with no supervision from small instances. The method is based on learning to decompose problems into subproblems so that the subproblems have a lower complexity as measured by their width. Problems and subproblems are captured by means of sketch rules, and the scheme for reducing the width of sketch rules is applied iteratively until the final sketch rules have zero width and encode a general policy. We evaluate the learning method on a number of classical planning domains, analyze the resulting hierarchical policies, and prove their properties. We also show that learning hierarchical policies by learning and refining sketches iteratively is often more efficient than learning flat general policies in one shot.

Introduction

Hierarchical policies are a key ingredient of intelligent behavior where high-level actions decompose into low-level ones. For example, in the problem of going from Paris to Rome, a high-level plan would be to go from Paris to Charles de Gaulle Airport, fly to Leonardo da Vinci-Fiumicino Airport, and then travel from there to Rome. Many other actions are needed to flesh out this high-level plan, like calling and boarding a taxi, and walking to the gate. However, these actions and many others represent lower levels of abstraction.[1]

While the use of hierarchical policies and strategies has been standard in both model-based planning (Sacerdoti 1974; Tate 1977; Erol, Hendler, and Nau 1994) and model-free reinforcement learning (Parr and Russell 1997; Dietterich 2000; Barto and Mahadevan 2003) for many years, the problem of *learning hierarchical policies* with no supervision remains an open challenge. The main reason for this is that *no clear principles have been identified for describing or uncovering effective hierarchical structure* in planning problems. Researchers have looked at "bottleneck states" in the state space (McGovern and Barto 2001), precondition relaxation methods (Sacerdoti 1974), and the causal graphs of planning problems (Knoblock 1994), among other aspects (Marthi, Russell, and Wolfe 2007), but the resulting accounts have a restricted scope.

In this work, we aim to address the problem of learning hierarchical abstractions and policies from a different angle, making use of two types of planning structures introduced recently: *general policies* (Bonet and Geffner 2018) and *policy sketches* (Bonet and Geffner 2021). General policies refer to policies that are not tied to a particular planning problem but can solve families of problems drawn from the same domain (e.g., Srivastava, Immerman, and Zilberstein 2008; Bonet, Palacios, and Geffner 2009; Hu and De Giacomo 2011; Belle and Levesque 2016; Celorrio, Segovia-Aguas, and Jonsson 2019; Illanes and McIlraith 2019). Policy sketches refer to "incomplete policies" that decompose problems into subproblems expressed by means of *sketch rules*. Policy sketches come with a theory of problem decomposition based on the notion of problem *width* (Lipovetzky and Geffner 2012). When the sketch width over a class of planning problems is bounded (Bonet and Geffner 2021), then the problems can be solved greedily in polynomial time, exponential in their width. Methods for *learning sketches of bounded width* from small examples have recently been developed (Drexler, Seipp, and Geffner 2022).

It is not surprising that the notions of general policies, sketches, and problem width can be used to shed light on the problem of learning hierarchical policies. Hierarchical policies, as expressed in hierarchical task networks, for example, are not aimed at solving individual problems but families of problems over the same domain. Moreover, they decompose problems into simpler subproblems whose complexity can be characterized by the notion of width.

In this paper, we draw on these notions to introduce a crisp characterization of hierarchical policies and a method for learning them, with no supervision, from small examples (classical planning problems). The method learns to decompose the training problems into subproblems of lower width. Sketch rules express decompositions, and the scheme for reducing the width of sketch rules is applied iteratively until

[1] According to Turing Award winner Yann LeCun: "1. AI must learn to represent the world. 2. AI must learn to think and plan in ways that are compatible with gradient-based learning. 3. AI must learn hierarchical representations of action plans." (LeCun 2022)

the final sketch rules have width zero and encode a policy. The method builds on a related method for learning sketches of width bounded by a parameter developed previously by Drexler, Seipp, and Geffner (2022).

The paper is organized as follows. After a preview and a review of background notions, we introduce hierarchical policies and a method for learning them, report experimental results, and present related work and conclusions.

Preview

Figure 1 shows the type of hierarchical policy that will be learned on the *Delivery* domain (Bonet and Geffner 2021), where an agent must deliver packages from different locations in a grid by picking them up and dropping them in a target cell, one by one. The agent can move one cell at a time to reach the packages and the target cell. The domain is a variation of the "Taxi" domain, a well-known benchmark in hierarchical reinforcement learning (Dietterich 2000).

The hierarchical policy shown in the figure is a tree where every node n corresponds to a class of *subproblems* $\mathcal{Q}_n$ expressed by a sketch rule $r(n)$ of the form $C \mapsto E$ over two types of state features: Boolean features like H, true in a state if and only if the agent is holding a package, and numerical features like u, p, and t, that capture the number of undelivered packages, the distance of the agent to the nearest package and to the target cell, respectively. The rule $\{u > 0\} \mapsto \{u\downarrow\}$ near the root, for example, expresses the family of subproblems where one more package needs to be delivered (and hence, the feature u decremented if positive), while the rule $\{\neg H, u > 0\} \mapsto \{H\}$ associated with its left child, expresses the family of subproblems where some package has to be held when none is. The first class of subproblems has width 2, while the second class has width 1. Moreover, the subproblems that result from the rules at the leaf nodes all have width 0, meaning that the subproblems defined by them can be solved in one step and thus determine a single action, not necessarily unique.

Background

We review classical planning, the notion of width, general policies, sketches, and sketch width drawing from Lipovetzky and Geffner (2012), Bonet and Geffner (2021), and Drexler, Seipp, and Geffner (2022).

Classical Planning

A **planning problem** or *instance* is a pair $P = \langle D, I \rangle$ where D is a first-order *domain* with action schemas defined over predicates, and I contains the objects in the instance and two sets of ground literals, the initial and goal situations $Init$ and $Goal$. The initial situation is consistent and complete, meaning either a ground literal or its complement is in $Init$. An instance P defines a state model $S(P) = \langle S, s_0, G, Act, A, apply \rangle$ where the states in S are the truth valuations over the ground atoms represented by the set of literals that they make true, the initial state s_0 is $Init$, the set of goal states G are those that make the goal literals in $Goal$ true, and the actions Act are the ground actions obtained from the schemas and objects. The ground actions in $A(s)$ are the ones that are applicable in a state s; namely, those whose preconditions are true in s, and the state transition function $apply$ maps a state s and an action $a \in A(s)$ into the successor state $s' = apply(s, a)$. A *plan* π for P is a sequence of actions $a_0, \ldots, a_n$ that is executable in s_0 and maps the initial state s_0 into a goal state; i.e., $a_i \in A(s_i)$, $s_{i+1} = apply(s_i, a_i)$, and $s_{n+1} \in G$. A state s is *solvable* if there exists a plan starting at s, and otherwise, it is *unsolvable* (also called *dead-end*). A state s is *alive* if it is solvable and not a goal state. The *length* of a plan is the number of its actions, and a plan is *optimal* if there is no shorter plan.

Width

The **width** $w(P)$ of a planning problem P is the minimum k for which there exists a sequence $t_0, t_1, \ldots, t_m$ of atom tuples t_i from P, each consisting of at most k atoms, such that 1) t_0 is true in the initial state s_0 of P, 2) any optimal plan for t_i can be extended into an optimal plan for t_{i+1} by adding a single action, for all $i = 1, \ldots, n-1$, and 3) any optimal plan for t_m is an optimal plan for P (Lipovetzky and Geffner 2012). We call such tuple chains $t_0, t_1, \ldots, t_m$, *admissible chains*. If the width of a problem P is $w(P) = k$, then the IW(k) algorithm finds an optimal plan for P in time and space that are exponential in k. IW(k) is a breadth-first search that prunes a newly generated state if it doesn't make a k-tuple of atoms true for the first time in the search. The IW algorithm runs IW(k) in sequence, for $k = 1, 2, \ldots, n$ until the problem is solved or found to be unsolvable (where n equals the number of problem variables). The width $w(P)$ is set to 0 if P is solvable in at most one step.

General Policies and Sketches

A **general policy** π over a class of problems $\mathcal{Q}$ is a set of policy rules $C \mapsto E$ where C consists of Boolean feature conditions, and E consists of feature effects, both over a set of features Φ (Bonet and Geffner 2018). A feature f is a function of the state. The features are either Boolean, taking values in the Boolean domain, or numerical, taking values in the non-negative integers. A Boolean (feature) condition has the form p or $\neg p$ for a Boolean feature p, and $n = 0$ or $n > 0$ for a numerical feature n. A feature effect is an expression of the form p, $\neg p$, or $p?$ for a Boolean feature p in Φ, and $n\downarrow$, $n\uparrow$, or $n?$ for a numerical feature n in Φ.

A general policy π splits state transitions in a problem P from the class $\mathcal{Q}$ into "good" or "bad". A state transition (s, s') is "good" (π-*compatible*) iff there is a policy rule $C \mapsto E$ such that s makes the feature conditions in C true and the transition (s, s') makes the effects in E true. Otherwise the transition (s, s') is "bad" (not π-*compatible*). The effect p (resp. $\neg p$) is true if p (resp. $\neg p$) is true in s', and the effect $n\uparrow$ (resp. $n\downarrow$) is true if n increases (resp. decreases), i.e., $n(s') > n(s)$ (resp. $n(s') < n(s)$). The effects $p?$ or $n?$ are always satisfied, meaning that features can change in any way (or keep their values). If the effect expression E does not mention a feature, then the value of the feature must stay the same. A state trajectory $s_0, \ldots, s_n$ is π-compatible in P if s_0 is the initial state of P and the transitions (s_i, s_{i+1}) are all π-compatible. The state trajectory is maximal if s_n is the first goal state in the trajectory, or if there are no goal

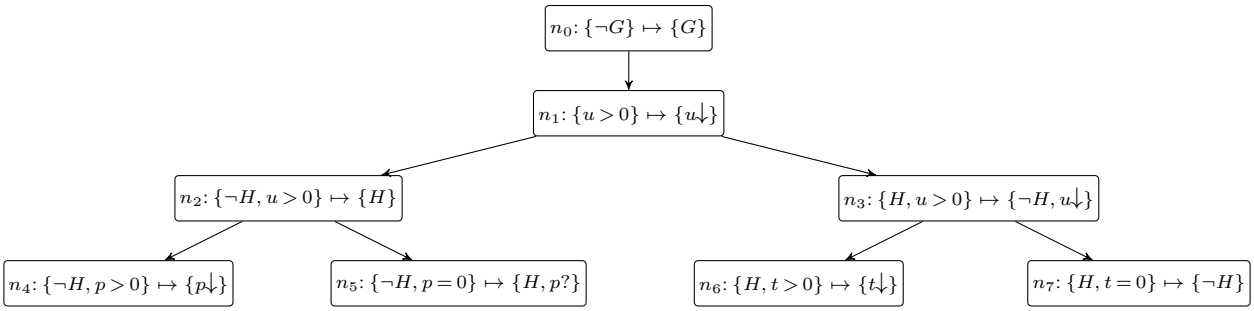

Figure 1: Hierarchical policy for the class $\mathcal{Q}$ of Delivery instances over the features $\{G, H, p, t, u\}$ where G is true iff the problem goal is true, H is true iff a package is being held, p is the distance to the nearest undelivered package, t is the distance to the target cell, and u is the number of undelivered packages. Every node n in the tree has a sketch rule $r(n) : C_n \mapsto E_n$ over the features (as shown), and represents a class $\mathcal{Q}_n$ of subproblems. For the root node n, $\mathcal{Q}_n = \mathcal{Q}$, while for the other nodes, $\mathcal{Q}_n$ is determined by $\mathcal{Q}_{n'}$ of the parent node n' and the sketch rule $r(n)$ for n. For each $P \in Q_{n'}$, $\mathcal{Q}_n$ includes the problems $P[s, G_n(s)]$ that are like P but with initial states s that makes C_n true, and goal states s' such that the pair of states $[s, s']$ satisfies E_n. The problems in $\mathcal{Q}_n$ are forced to have a smaller width than the problems in $\mathcal{Q}_{n'}$. For the policy shown, the problems in $\mathcal{Q}_n$ have width 0 for the leaf nodes n, width 1 for the nodes that are one level up, width 2 for the single node that is two levels up, and unbounded width for the root node. Sketch rules of width 0, such as those appearing in the leaf nodes are "executable" in the sense that they select indirectly an action for execution, not necessarily unique.

states in the trajectory and the trajectory is cyclic (one state repeated) or can't be extended (no π-compatible transitions (s_n, s) in P for any s). The policy π solves P if all maximal state trajectories reach a goal state, and π solves $\mathcal{Q}$ if it solves every P in $\mathcal{Q}$.

Sketches have the syntax of general policies but with slightly different semantics where state transitions (s, s') are replaced by state pairs $[s, s']$ (Bonet and Geffner 2021). In a state pair $[s, s']$, s' does not have to be reachable from s in a single step but can be an arbitrary state. A sketch R is indeed a set of policy rules $C \mapsto E$, called then sketch rules. A state pair $[s, s']$ is compatible with a sketch rule $r = C \mapsto E$ or r-compatible when the state transition (s, s') is, and it is said to be R-compatible when it is compatible with a rule in R. The set of subgoal states for r in s, denoted as $G_r(s)$ is given by the states s' such that $[s, s']$ is r-compatible or s' is a top goal in G. The sketch rules do not define a policy that can solve a problem P reactively but a *decomposition* of the problem into subproblems: when in a state s, the subgoal states s' to be reached are those in $G_R(s) = \cup_{r \in R} G_r(s)$. The sketch R thus decomposes a problem P into subproblems $P[s, G_R(s)]$ that are like P but with initial state s and set of goal states in $G_R(s)$.

The **width of a sketch** R over a collection of problems $P \in \mathcal{Q}$ is the maximum width of the subproblems $P[s, G_R(s)]$ for a class of relevant states s, $s \in S^o_R(P)$ (Bonet and Geffner 2021); details below. A sketch R is **acyclic** in $\mathcal{Q}$ if there is no sequence of states $s_1, \ldots, s_n$ over the states of a problem $P \in \mathcal{Q}$ that is R-compatible (i.e., where each pair $[s_i, s_{i+1}]$ is R-compatible) and where $s_1 = s_n$ (Drexler, Seipp, and Geffner 2022). If R is acyclic and has width k over $\mathcal{Q}$, then the problems in $\mathcal{Q}$ can be solved in time exponential in k by the SIW_R algorithm; a version of the SIW algorithm (Lipovetzky and Geffner 2012) that iteratively moves from a state s to a state s' in $G_R(s)$.

Example 1. *A* general policy π *for Delivery can be defined over the features* $\Phi = \{H, p, t, u\}$ *from Figure 1 as*

$r_1 : \{\neg H, p > 0\} \mapsto \{p\downarrow, t?\}$ *; go to nearest pkg*
$r_2 : \{\neg H, p = 0\} \mapsto \{H, p?\}$ *; pick it up*
$r_3 : \{H, t > 0\} \mapsto \{t\downarrow, p?\}$ *; go to target*
$r_4 : \{H, u > 0, t = 0\} \mapsto \{\neg H, u\downarrow, p?\}$ *; deliver pkg*

The first rule (r_1) selects an action that decreases the distance to the nearest package ($p\downarrow$), when the agent is not holding a package ($\neg H$), regardless of whether this means moving towards or away from the target cell ($t?$). The other rules say to pick a package when possible (r_2); to move to the target cell when holding a package (r_3); and to drop the package at the target cell (r_4). A sketch of width 1 *for Delivery can be defined instead by means of two sketch rules with the same syntax but with different semantics:*

$r_1 : \{\neg H\} \mapsto \{H\}$ *; get hold of pkg*
$r_2 : \{H, u > 0\} \mapsto \{\neg H, u\downarrow\}$ *; deliver pkg*

The first sketch rule (r_1) says to get hold of a package, while the second sketch rule (r_2) says to deliver the package being held. Both tasks have width 1 *and can be solved in linear time (Lipovetzky and Geffner 2012).*

Learning General Policies and Sketches

A general policy π and the necessary features Φ can be learned at the same time by finding the "best" truth assignment that satisfies a propositional theory $T(\mathcal{P}, \mathcal{F})$, where $\mathcal{P}$ represents small training instances P from the target class of problems $\mathcal{Q}$, and $\mathcal{F}$ represents a pool of features derived automatically from $\mathcal{P}$ and the known domain predicates (Francès, Bonet, and Geffner 2021). A similar propositional encoding $T(\mathcal{P}, \mathcal{F}, k, m)$ has been developed for learning sketches of width bounded by k where k is a parameter, normally 0, 1, or 2 (Drexler, Seipp, and Geffner 2022). The main difference is that for learning sketches, the pairs (s, s')

in the encoding are no longer state transitions at distance 1, but arbitrary state pairs $[s, s']$. The width of the subproblems $P[s, G_R(s)]$ is bounded by k by associating the set of states s' in $G_R(s)$, i.e., the "good pairs $[s, s']$, with the closest states from s where a subgoal t of width bounded by k from s holds. The sketch rules and the selected features are then extracted from the best satisfying assignment in the same way as policy rules are.

Hierarchical Policies

We focus now on the characterization of the hierarchical policies that we want to learn, and how to learn them.

Modularity: Semantics of Sketches Revisited

We use the language of sketches but introduce a small change in their semantics. For an instance P and sketch R, the set $S^o_R(P)$ of *relevant states* of P (above) is usually defined as the minimal set of states such that: 1) the initial state of P is in $S^o_R(P)$, 2) if a non-goal state $s \in S^o_R(P)$, then the states $s' \in G_R(s)$ that are closest to s are also in $S^o_R(P)$. Under what we call the **modular semantics**, the set of relevant states $S_R(P)$ is defined instead as follows:

Definition 2. *For a sketch R over an instance P, the set $S_R(P)$ of **relevant states** of P* under the modular semantics *is the minimal set such that: 1) the initial state of P is in $S_R(P)$, 2) if a non-goal state $s \in S_R(P)$, then the states $s' \in G_r(s)$ are also in $S_R(P)$, for each rule $r : C \mapsto E$ in R such that C is true in s.*

In this definition, $G_r(s)$ stands for the set of states s' such that the pair $[s, s']$ satisfies the sketch rule r, and does not necessarily include the goal states of P as above. There are two other key differences between the new sets of relevant states $S_R(P)$, and the previous sets $S^o_R(P)$. First, the sets of subgoal states $G_r(s)$ for each of the rules r in R are not aggregated in a set $G_R(s)$: they are considered separately. Any sketch rule $r : C \mapsto E$ whose condition C holds in s, can be used to define the subgoal states s' in the modular semantics. Second, these states s' are no longer expected to be "closest" to s, as subproblems in hierarchical decompositions cannot be expected to be solved optimally. Clearly, $S^o_R(P) \subseteq S_R(P)$, meaning that the sketch R under the modular semantics gives rise to more subproblems. The revised definition of sketch width can then be phrased as follows:

Definition 3. *The **width of a sketch rule** r in R for a problem P* under the modular semantics *$w(P)$ is the maximum width of the problems $P[s, G_r(s)]$ over all the relevant non-goal states s in $S_R(P)$ where the condition C of r holds.*

Definition 4. *The **width of a sketch** R for P* under the modular semantics *$w_R(P)$ is the highest width of the sketch rules $r \in R$ for P, provided that all states s in $S_R(P)$ satisfy the condition of some rule in R (i.e., all relevant non-goal states are covered by R). Otherwise, $w_R(P)$ is defined as the number of atoms in P. The width of a sketch R over $\mathcal{Q}$ is the maximum width of R over the problems P in $\mathcal{Q}$.*

Since the number of (relevant) subproblems under the modular semantics is larger than under the standard semantics, it is easy to show that the width of a sketch R over $\mathcal{Q}$ under the standard semantics is upper bounded by the width of R over $\mathcal{Q}$ *under the modular semantics*, and both coincide when the rules in the sketch R are disjoint (no two conditions are true in the same state). From now on, we assume the modular semantics when discussing sketch width.

Hierarchical Policies

We consider hierarchical policies Π given by trees where every node n is associated with a sketch rule $r(n)$ over a set of features that are well defined for the target class of problems $\mathcal{Q}$ from a common planning domain:

Definition 5. *A hierarchical policy Π for a class of problems $\mathcal{Q}$ is a single rooted tree where every node n has a sketch rule $r(n)$ over well-defined features in $\mathcal{Q}$.*

We are interested in hierarchical policies for $\mathcal{Q}$ that solve all instances P in $\mathcal{Q}$. The hierarchical policies for $\mathcal{Q}$ that are *valid* split the class $\mathcal{Q}_n$ of problems P associated with a node n in the tree, into classes of problems $\mathcal{Q}_{n'}$ for the children n' of n that are simpler, as measured by their width. For the root node $\mathcal{Q}_n = \mathcal{Q}$, and for the leaf nodes, $\mathcal{Q}_n$ contains subproblems of width 0 solvable in one step. The class of problems $\mathcal{Q}_{n'}$ for a non-root node n' is determined by the parent class $\mathcal{Q}_n$ and the sketch rule $r(n')$ associated with n':

Definition 6. *A hierarchical policy Π for $\mathcal{Q}$ is* valid *if the rules $r(n)$ determine classes $\mathcal{Q}_n$ of subproblems from $\mathcal{Q}$ that together obey the following constraints:*

1. ***Root node** n: The rule $r(n)$ is $\neg G \mapsto G$ where G is a dummy Boolean feature that holds only in the goal states of a problem $P \in \mathcal{Q}$, and $\mathcal{Q}_n = \mathcal{Q}$.*
2. ***Inner node** n: The rules $r(n')$ of the children n' of n encode an acyclic **sketch** $R(n)$ for $\mathcal{Q}_n$ with a **width** that is smaller than the width of $\mathcal{Q}_n$. The class of problems $\mathcal{Q}_{n'}$ at each child n' with rule $r = r(n') = C \mapsto E$ is defined as*

$$\mathcal{Q}_{n'} = \{P[s, G_r(s)] \mid s \in S_R(P), s \models C, P \in \mathcal{Q}_n\}$$

3. ***Leaf node** n: The class of problems $\mathcal{Q}_n$ has **width** 0 (solvable by executing a single action).*

In other words, the rules $r(n')$ of the children n' of n in a valid hierarchical policy Π for $\mathcal{Q}$ decompose the class of problems $\mathcal{Q}_n$ at n into classes of subproblems $Q_{n'}$ of lower width, starting with $\mathcal{Q}_n = \mathcal{Q}$ at the root node and ending with classes of problems $\mathcal{Q}_n$ of width 0 at the leaves. The sketch rules $C \mapsto E$ at the leaf nodes n are indeed policy rules, as in a state s that makes C true, states s' that along with s can make the effect E true, can be reached by performing one action in s. Before we go into more details about the execution of hierarchical policies, we illustrate the above definition with an example.

Example 7. *Consider the hierarchical policy for Delivery shown in Figure 1. The root node n_0 trivially satisfies Clause 1 of Definition 6 of a valid hierarchical policy. Also, the sketch $R(n_0)$ that results from the rules of the children of n_0, i.e., $r(n_1)$, forms an acyclic sketch (u is always decremented) for the class of problems $\mathcal{Q}_{n_0} = \mathcal{Q}$ whose width is not bounded. For each P in $\mathcal{Q}_{n_0}$, $\mathcal{Q}_{n_1}$ contains in turn the*

(sub)problems $P' = P[s, G_r(s)]$, $r = r(n_1)$ where one undelivered package must be delivered (thus decrementing u), which have width bounded by 2. Similarly, for each such P', $\mathcal{Q}_{n_2}$ contains the problems $P' = P[s, G_r(s)]$, $r = r(n_2)$, where a package must be picked up and the feature H made true. These problems have width 1, and the same is true for the problems in $\mathcal{Q}_{n_3}$. Indeed, the sketch $R(n_1)$ given by the rules $r(n_2)$ and $r(n_3)$ is acyclic (proof omitted) and has width 1 for the class of problems $\mathcal{Q}_{n_1}$. Finally, $\mathcal{Q}_{n_4}$ for the leaf node n_4 expresses the subproblems where the agent is not holding a package and must get closer to the closest one (any one). These problems have width 0 and can be solved in a single step by any action that generates a state transition satisfying rule $r(n_4)$. Indeed, the sketch $R(n_1)$ given by the rules $r(n_4)$ and $r(n_5)$ is acyclic and has width 0 for the class of problems $\mathcal{Q}_{n_3}$. Similar properties hold for the other nodes in the tree, making it represent a valid hierarchical policy for the class of problems $\mathcal{Q}$.

Execution of Hierarchical Policies

A hierarchical policy Π is executed on a problem $P \in \mathcal{Q}$ by using a stack with entries $\langle s, n\rangle$, where s is a state from P and n is a node in Π. The execution also tracks a current state s'. At any point during the execution of the policy, the entries $\langle s, n\rangle$ in the stack describe the subproblems $P[s, G_r(s)]$ that are being solved, where $r = r(n)$. Initially, the stack contains the pair $\langle s_0, n_0\rangle$ where s_0 is the initial state of P and n_0 is the root node of Π, and the current state is $s' := s_0$. Then the execution considers three cases iteratively until the stack becomes empty:

1. If the top entry of the stack is $\langle s, n\rangle$, the current state s' is not in $G_r(s)$ for $r = r(n)$, and n is not a leaf node, a (any) child n' of n is chosen with rule $r(n')$ whose condition is true in s'. Then the entry $\langle s', n'\rangle$ is pushed onto the stack without changing the current state s'.
2. If the top entry of the stack is $\langle s, n\rangle$, s' is not in $G_r(s)$ for $r = r(n)$, and n is a leaf node, then an applicable action a in s' is selected and applied for obtaining a state s'' such that their pair $[s', s'']$ complies with rule r. (This must be possible in a valid policy.) The current state is set to s''.
3. If the top entry of the stack is $\langle s, n\rangle$ and s' is in $G_r(s)$ for $r = r(n)$, then the entry is popped from the stack without changing the current state s'.

For valid hierarchical policies, this execution is always possible and ends in a goal state:

Theorem 8. *If a hierarchical policy Π is* valid *for the class of problems $\mathcal{Q}$, then executing Π on a problem $P \in \mathcal{Q}$ ends in a goal state of P.*

Proof. Consider a hierarchical policy Π that is valid for a class of problems $\mathcal{Q}$ and a problem $P \in \mathcal{Q}$ with initial state s_0. A configuration c is a pair $c = ((x_1, \ldots, x_m), s')$ where s' is the current state and $(x_1, \ldots, x_m)$ are the entries in the stack with x_m being the top entry. The initial configuration is $c_0 = ((\langle s_0, n_0\rangle), s_0)$. A configuration is a goal configuration if the stack is empty and s' is a top goal of P. We have to show that 1) for all $i = 0, \ldots, t$ there exists a step a_i in the execution such that applying step a_i in the non-goal configuration c_i results in a successor configuration c_{i+1}, and 2) t is finite and the last configuration c_{t+1} is a goal configuration.

We prove 1) by induction over t. Induction basis ($t = 0$): The statement is trivially true. Induction hypothesis: Assume that the statement is true for all t. Induction step ($t \to t+1$): We have to show that if c_t is not a goal configuration then there exists a step a_t such that applying step a_t in c_t results in a successor configuration c_{t+1}. Informally speaking, we have to show that the execution does not fail in a non-goal configuration. Consider a non-goal configuration $c_t = ((x_1, \ldots, x_m), s')$, top stack entry $x_m = \langle s, n\rangle$, and rule $r = r(n)$. If $s' \in G_r(s)$ then applying Step 3 results in the successor configuration $c_{t+1} = ((x_1, \ldots, x_{m-1}), s')$. And otherwise, if $s' \notin G_r(s)$, then we need to distinguish whether n is an inner node or a leaf node. First, assume n is a leaf node. The class of problems $\mathcal{Q}_n$ at each leaf node has width 0 (Def. 6.3). Hence, there must exist an applicable action a with $s'' = apply(s', a)$ such that s'' is in $G_r(s')$. Hence, applying Step 2 results in the successor configuration $c_{t+1} = ((x_1, \ldots, x_m), s'')$. Second, assume n is an inner node and let $R = R(n)$ be the sketch at n. The width of R is bounded by $k > 0$, and the width of the classes of subproblems $\mathcal{Q}'_n$ at each child node n' of n are smaller (Def. 6.2). Hence, for all relevant non-goal states $s'' \in S_R(P_n)$ and $P_n \in \mathcal{Q}_n$ there exists a child node n' of n with rule $r' = r(n') \in R$ such that s'' satisfies the conditions of r' and the width of the rule is bounded by k (Def. 3 and 4). We have to show that s' is a relevant state. If $s = s'$ then $s' \in S_R(P)$ because the initial state s_0 of $P_n \in \mathcal{Q}_n$ is a relevant state (Def. 2). Furthermore, if $s \neq s'$ then in configuration c_{t-1} Step 3 was applied and hence $s' \in G_r(s)$ and therefore $s' \in S_R(P_n)$ because states $G_r(s)$ are also in $S_R(P_n)$ (Def. 2). Thus, applying Step 1 results in the successor configuration $c_{t+1} = ((x_1, \ldots, x_m, x_{m+1}), s'')$ where $x_{m+1} = \langle s'', n'\rangle$.

Last, we prove 2). All sketches $R(n)$ for all inner nodes n are acyclic (Def. 6.2). Hence, the same relevant state in $S_{R(n)}(P_n)$ will never be the current state s' more than once for all $P_n \in \mathcal{Q}_n$ and for all inner nodes n. Hence, eventually the stack becomes empty and s' is a top goal, and therefore, the last configuration c_{t+1} is a goal configuration. □

Example 9. *Consider the hierarchical policy for Delivery shown in Figure 1 and an instance with initial state s_0 where there are two undelivered packages p, p' and the current location is different from the location of p and p'. The execution works as follows. Initially, the stack contains the single entry $\langle s_0, n_0\rangle$, and the current state s' is s_0. In Step 1 (Case 1), $\langle s_0, n_1\rangle$ is pushed onto the stack because the hand is empty ($\neg H$) in s'. In Step 2 (Case 1), $\langle s_0, n_4\rangle$ is pushed onto the stack because the distance to the nearest package is greater than zero ($p > 0$) in s'. In Step 3 (Case 2), an action is chosen that moves the agent closer to a package, landing in state s_1, and s' is set to s_1. In Step 4 (Case 3), $\langle s_0, n_4\rangle$ is popped from the stack because $s' \in G_{r(n_4)}(s_0)$. The remainder of the execution works similarly.*

Hierarchical Policies vs. Flat General Policies

A valid hierarchical policy is *simple* if 1) the sketch rules of the children of a given node have mutually exclusive conditions, and 2) the sketch rules are *persistent* in the sense that, during the execution, when the top of the stack is $\langle n, s\rangle$ for a leaf node n, and the current state is s', the conditions of the rules in all nodes n' above n in the policy tree are true in s. Any valid hierarchical policy Π that is also simple can be transformed into a flat general policy.

The transformation procedure works as follows. Each sketch rule $r(n) : C \mapsto E$ of a leaf node n in Π is converted into a policy rule $r_n : C \dot\cup C' \mapsto E \dot\cup E'$ where C' collects the feature conditions of all the rules above n in the hierarchy, and E' contains the effects $p?$ and $n?$ for every feature f used in Π except for those appearing in the sketch rule in one of the siblings of n. The procedure can easily be generalized to obtain a general policy π_n that solves the class of problems $\mathcal{Q}_n$ at node n.

Theorem 10. *If a valid hierarchical policy Π for $\mathcal{Q}$ is simple then the transformation above yields a (flat) general policy π that solves $\mathcal{Q}$.*

Proof sketch: The mutual exclusive and persistent conditions ensure that the resulting policy rules apply in the states where the corresponding sketch rules apply during the execution of the hierarchical policies, and that they will not interfere with each other.

Example 11. *Consider the valid hierarchical policy $\Pi_{Miconic}$ shown in Figure 3. Notice that $\Pi_{Miconic}$ is simple because the sketch rules are persistent, i.e., $b=0$ implicitly holds during the execution in $r(n_4)$ and $r(n_5)$, and $b>0$ implicitly holds during the execution in $r(n_6)$ and $r(n_7)$. The (flat) general policy π over features $\{b, d, p, w\}$ is*

$r_1 : \{b=0, w=0\} \mapsto \{b?, d?, w\uparrow\}$
$r_2 : \{b=0, w>0\} \mapsto \{b\uparrow, d?, w?\}$
$r_3 : \{b>0, d=0\} \mapsto \{b?, d\uparrow, w?\}$
$r_4 : \{b>0, d>0\} \mapsto \{b\downarrow, d\downarrow, w?\}$

Learning Hierarchical Policies

We learn hierarchical policies Π for a potentially infinite class of problems by using a small subset $\mathcal{P}$ of $\mathcal{Q}$ comprised of a few small instances. We show empirically that these training sets are sufficient to learn hierarchical policies that generalize to the the whole class $\mathcal{Q}$, and for some domains, we provide formal proofs that guarantee this generalization.

The key operation for learning a valid hierarchical policy Π for $\mathcal{Q}_n = Q$ is to learn an acyclic sketch $R(n)$ over $\mathcal{Q}_n$ of width lower than that of $\mathcal{Q}_n$. Once this is done, there will be a child n' of n for each of the rules $r(n')$ in the sketch $R(n)$. Moreover, if the instances in $\mathcal{Q}_n$ are small enough, the resulting subclasses of problems $\mathcal{Q}_{n'}$ for each child can be easily derived from $\mathcal{Q}_n$ and each of the rules $r(n')$. Then the process can be repeated until generating nodes n'' with classes of problems of $\mathcal{Q}_{n''}$ of width 0 that have no children.

The method sketched above for learning hierarchical policies leaves open the choice of the width for the children problems $\mathcal{Q}_{n'}$, which have to be lower than the width of the parent problems $\mathcal{Q}_n$. We say that a *hierarchical policy has width* k, and write it as Π_k, if the learned sketch $R(n_0)$ for the root node has width k. For the internal nodes n, we only demand that the width of the children problems is $k-1$ if the problems at n have width k. Clearly, a policy Π_k can have at most depth k, and the hierarchical policy Π_k for $k = 0$ is just a flat policy.

A method learning a sketch R for a given class of problems $\mathcal{Q}$ with width bounded by a parameter k has been expressed as the task of finding a best truth assignment that satisfies a propositional theory (Drexler, Seipp, and Geffner 2022). Here, we adjust the theory to consider the modular semantics of sketches implicitly used by hierarchical policies.

Encoding for Learning Modular Sketches

For a given set of training instances $\mathcal{P} \subseteq \mathcal{Q}$ from a planning domain D, a set of features $\mathcal{F}$, width k, and the maximum number of sketch rules m, we construct the propositional theory $T(\mathcal{P}, \mathcal{F}, k, m)$. To describe its variables, we use the following symbols: s, s' range over all states in the training set, f ranges over all features in $\mathcal{F}$, v ranges over all feature conditions or '?', v' ranges over all feature effects or '?', and i ranges over all rule indices $1, \ldots, m$. The propositional variables in $T(\mathcal{P}, \mathcal{F}, k, m)$ are:

- $select(f)$: feature f is included in Φ
- $cond(i, f, v)$: rule i has condition v for f
- $\mathit{eff}(i, f, v')$: rule i has effect v' for f
- $subgoal(s, t, i)$: rule i has subgoal t of width $\leq k$ in s
- $sat_rule(s, s', i)$: pair $[s, s']$ is compatible with rule i
- $sat_cond(s, i)$: state s satisfies conditions of rule i
- $r\text{-}reach(s)$: state s is in $S_R(P)$

To describe the constraints in $T(\mathcal{P}, \mathcal{F}, k, m)$, we use the same symbols as above, with the difference that s now only ranges over all *alive* states. In addition, t ranges over subgoal tuples with width at most k in s, $\mathit{dist}(s, s')$ is the shortest distance from s to s', $\mathit{dist}(s, t)$ is the length of an admissible chain that ends in subgoal tuple t for s, $S^*(s, t)$ are all states that result from applying optimal plans from $P[s, t]$ in s. The constraints are

C1 $cond(i, f, v) \vee \mathit{eff}(i, f, v') \rightarrow select(f)$, unique v, v',
C2 $r\text{-}reach(s) \rightarrow \vee_i sat_cond(s, i)$,
C3 $r\text{-}reach(s) \wedge sat_cond(s, i) \rightarrow \vee_t subgoal(s, t, i)$,
C4 $r\text{-}reach(s_0)$ for initial state s_0,
C5 $r\text{-}reach(s) \wedge sat_rule(s, s', i) \rightarrow r\text{-}reach(s')$,
C6 $subgoal(s, t, i) \rightarrow \wedge_{s' \in S^*(s,t)} sat_rule(s, s', i)$,
C7 $sat_rule(s, s', i) \rightarrow \vee_{\mathit{dist}(s,t) \leq \mathit{dist}(s,s')} subgoal(s, t, i)$,
C8 $sat_rule(s, s', i) \leftrightarrow [s, s']$ compatible with rule i,
C9 $sat_cond(s, i) \leftrightarrow s$ satisfies conditions of rule i, and
C10 the collection of m rules is acyclic.

Theorem 12. *The propositional theory $T(\mathcal{P}, \mathcal{F}, k, m)$ is satisfiable iff there exists an acyclic sketch R over $\mathcal{P}$ with width $\leq k$ under the modular semantics with at most m rules.*

Proof. "$\Rightarrow$": Assume that the theory $T(\mathcal{P}, \mathcal{F}, k, m)$ is satisfiable. The sketch R over features Φ can be read off directly from the variables $cond(i, f, v)$, $eff(i, f, v')$ and $select(f)$. It remains to show that the width is bounded by k. We need to check all relevant states $S_R(P)$ for $P \in \mathcal{P}$. The initial state s_0 is in $S_R(P)$ because C4 is satisfied. Consider a non-goal state $s \in S_R(P)$. There must be a rule r such that s satisfies the conditions of r because C2 is satisfied. For this rule r, there is a subgoal t that bounds the width because C3, C6, C7 are satisfied. Relevant states are properly extended by the subgoal states because C5 is satisfied. States that satisfy the conditions of a rule are identified because C8 is satisfied. Similarly, state pairs that are r-compatible for some $r \in R$ are identified because C9 is satisfied. Hence, the sketch width is bounded by k. Finally, R is acyclic because C10 is satisfied.

"$\Leftarrow$": Assume that there exists an acyclic sketch R with width $w_R(\mathcal{P}) \leq k$ consisting of at most m rules. C1 is true because R consists of at most m rules. C10 is true because the sketch is acyclic over $S_R(P)$. The sketch R induces a set of relevant $S_R(P)$ in each problem $P \in \mathcal{P}$ by definition. C4 is true because the initial state $s_0 \in S_r(P)$, and C5 is true because it extends $S_R(P)$ by the subgoal states $G_r(s)$. C2 is true because the width of each non-goal state $s \in S_r(P)$ is bounded by k for some rule r whose precondition is satisfied in s. C3 is true because the rule r has a subgoal t because the conditions of r are true in s. C6 is true because optimal plans for t are plans for subproblem P at s for r, and C7 is true because the optimal plans for t are also plans for $P[s, G_r(s)]$. C8 and C9 are true because each rule defines state pairs that are r-compatible and states where the feature conditions C of r are true. $\square$

The hierarchical policy Π_k for a collection $\mathcal{Q}$ of instances is constructed then, as explained above, by learning sketches $R(n)$ for the class of problems $\mathcal{Q}_n$ of width k, starting with the root node n and $\mathcal{Q}_n = \mathcal{Q}$, assigning the rules in $R(n)$ to the children n' of n, and then learning sketches for the classes of subproblems $\mathcal{Q}_{n'}$ that have width $k - 1$, and so on, until the nodes have subproblems with width zero and no further children.

Experiments

We implemented the learning and testing pipelines in Python and C++, respectively. The benchmark set consists of seven tractable classical planning domains from the International Planning Competition (IPC) and four tractable domains from the paper by Francès, Bonet, and Geffner (2021). The benchmarks, code, and data are available online (Drexler, Seipp, and Geffner 2023). For learning, we use the DLPlan library (Drexler, Francès, and Seipp 2022) to automatically generate the feature pool $\mathcal{F}$ based on a description logics grammar (Baader et al. 2003) over the domain predicates. The complexity of a feature is the number of applied grammar rules. Our features have a maximum complexity of 9, except for Delivery, where we need a maximum complexity of 15 to learn a hierarchical policy. We use a limit of $m = 4$ for the number of rules in the sketches, which also limits the number of children per node in the hierarchy. We run learning experiments on a single machine with 32 cores, 96 GiB of memory, and a time limit of 24 hours. For testing, we use the 30 tasks per domain from the Autoscale benchmark set (Torralba, Seipp, and Sievers 2021). For domains without Autoscale tasks, we generate 30 large instances with PDDL generators (Seipp, Torralba, and Hoffmann 2022). We run testing experiments on single CPU cores with 8 GiB of memory and a time limit of 30 minutes.

Data Generation

For each domain D, we generate a set of training instances $\mathcal{P}$ with at most 2000 states using PDDL generators (Seipp, Torralba, and Hoffmann 2022). In domains with unsolvable states, it is helpful to be able to distinguish solvable from unsolvable states. For this reason, in the Spanner domain, the features learned by Ståhlberg, Francès, and Seipp (2021) are included.

Incremental Learning

While it would be possible to combine all training instances $\mathcal{P}$ to form a single large propositional theory $T(\mathcal{P}, \mathcal{F}, k, m)$, we follow the method in previous work (2022) that considers the data incrementally. The method orders the training instances by size (number of states) and, in each step, adds the smallest instance to the propositional theory where the current sketch fails to find a new sketch R until R solves all training instances. If it adds an instance that is larger than all others in the training set, it removes all smaller instances. We encode the propositional theory as an answer set program (ASP) and solve it using the Clingo solver (Gebser et al. 2019). We chose ASP because it is a declarative programming language for solving combinatorial optimization problems and, therefore, is a higher-level language compared to SAT, which makes development more flexible. We exploit indistinguishable constraints to reduce the size of the encodings (Francès, Bonet, and Geffner 2021). To obtain simple solutions, we minimize the total complexity of the selected features, i.e., $\min \sum_{f \in \Phi} complexity(f)$.

Results

Learning. Table 1 shows learning data. In a pairwise comparison between the three width values, none dominates the others regarding runtime. However, $k > 0$ often uses less memory and time compared to $k = 0$. The main reason for the efficiency gains arising from learning hierarchical policies Π_k in place of flat policies (hierarchical policies Π_0) is that the former requires fewer states because of the decomposition.

Testing. Table 2 shows empirically that all learned hierarchical policies Π_k generalize to larger instances. In our comparison, we also include SIW_R, where we use the sketches from Π_k of the first decomposition, and the two state-of-the-art domain-independent satisficing planners LAMA (Richter and Westphal 2010) and BFWS (Lipovetzky and Geffner 2017). In Spanner, neither LAMA nor BFWS can solve a single instance. In comparison, the flat and hierarchical policies solve the hardest instance in at most 41 seconds except

for Visitall where the computation of the distance to an unvisited location is the bottleneck. Flat and hierarchical policies are usually faster than LAMA, BFWS, and SIW_R because their execution does not involve search, yet in Reward and Visitall, the use of expensive distance features significantly slows down executions.

Since the learned hierarchical policies use interpretable features, it is possible to formally check whether they are valid for the whole problem class $\mathcal{Q}$. For illustration purposes, we state the validity of the learned hierarchical policies for two domains, with full proofs in the an extended version of the paper (Drexler, Seipp, and Geffner 2023).

In the class of problems $\mathcal{Q}_{\text{Spanner}}$, there is a uni-directional path connecting a shed with a gate. There is a man in the shed, a set of spanners distributed on the path to the gate, and as many loose nuts at the gate as there are spanners. Each spanner breaks after tightening a nut. The objective is to tighten all nuts. There are actions for moving the man forward, picking spanners when the man is at the location of an unpicked spanner, and tightening a nut, rendering the used spanner unusable.

Proposition 13. *The learned hierarchical policy $\Pi_{\textit{Spanner}}$ shown in Figure 2 is valid for $\mathcal{Q}_{\textit{Spanner}}$.*

In the class of problems $\mathcal{Q}_{\text{Miconic}}$, there is an elevator, a set of floors, and a set of people. The objective is to move each person from their origin to their destination floor. There are actions for boarding and unboarding a person. The elevator has unlimited capacity and can move from any floor to any other floor with a single action.

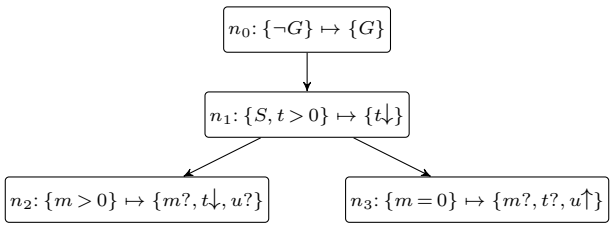

Figure 2: Hierarchical policy Π_{Spanner} for $\mathcal{Q}_{\text{Spanner}}$ over the set of features $\{G, m, S, t, u\}$ where G is true iff all nuts are tightened, m is the number of spanners that are at the current location of the man, S is true iff state is solvable, t is the number of unpicked spanners and loose nuts, and u is the number of locations that are unreachable by the man. The meanings are "tighten all nuts" (n_0), "pickup spanners or tighten nuts" (n_1, n_2), and "move forward if the man's position contains no spanners nor nuts" (n_3).

Proposition 14. *The learned hierarchical policy $\Pi_{\textit{Miconic}}$ shown in Figure 3 is valid for $\mathcal{Q}_{\textit{Miconic}}$.*

Failed Experiments

Apart from the domains in the tables, we also experimented with three more domains: Childsnack, Grid, and Blocksworld with atomic actions. In Childsnack, we hit the time limit of 24 hours wall-clock time. In Grid, we failed to generate sufficiently small instances because the PDDL generator ran into an infinite loop. In Blocks, we were unable to find a hierarchical policy using $m = 4$ and the derived pool of features $\mathcal{F}$.

Related Work

General Policies. A number of works have addressed the problem of learning policies that apply to many planning instances involving different state spaces and, in many cases, different (ground) action spaces. Hierarchical policies, as expressed, for example, in hierarchical task networks (HTNs), are normally general in this sense. Our account builds on a particular **language** for expressing general policies that consists of policy rules over a given set of Boolean and numerical features (Bonet and Geffner 2018) that express qualitative changes over their values (Srivastava et al. 2011; Bonet and Geffner 2020). The approach for learning such policies over pools of features derived from the domain predicates is also borrowed from this line of work (Bonet, Francès, and Geffner 2019; Francès, Bonet, and Geffner 2021).

General Problem Decomposition. HTNs decompose tasks into simpler ones, and a number of methods for learning them have been studied (Zhuo, Muñoz-Avila, and Yang 2014; Hogg, Muñoz-Avila, and Kuter 2016). These methods, however, learn decompositions using different inputs like annotated traces and decompositions. In this work, we make use of a different language for expressing problem decompositions that is similar to the language of general policies but with slightly different semantics where **sketch rules** express subproblems (Bonet and Geffner 2021; Drexler, Seipp, and Geffner 2021). Sketches of **width** bounded by a constant k can be learned without supervision (Drexler, Seipp, and Geffner 2022). Reward machines are related to sketches involving Boolean features only and have been used to convey subgoal structure in the setting of reinforcement learning (Icarte et al. 2022).

Hierarchical Reinforcement Learning. A number of approaches have been developed to express and exploit hierarchical structure in the model-free setting of reinforcement learning: including options (Sutton, Precup, and Singh 1999), hierarchies of machines (Parr and Russell 1997) and MaxQ hierarchies (Dietterich 2000). While this "control knowledge" is often provided by hand, a vast literature has explored different methods for learning these hierarchies without supervision in both RL and planning. Researchers have looked at "bottleneck states" in the state space (McGovern and Barto 2001), precondition relaxations (Sacerdoti 1974), causal graphs of planning problems (Knoblock 1994), "eigenpurposes" of the matrix dynamics (Machado, Bellemare, and Bowling 2017), and informal width-based considerations (Junyent, Gómez, and Jonsson 2021), among other ideas, but their scope has not been found to be general or powerful enough.

Conclusions

Hierarchical policies are a key ingredient of intelligent behavior, but no general, crisp principles have been identified for characterizing and learning them. In this work, we have

	Π_0							Π_1							Π_2						
Domain	M	T	$\|S\|$	$\|\mathcal{F}\|$	C	$\|\Phi\|$	$\|R\|$	M	T	$\|S\|$	$\|\mathcal{F}\|$	C	$\|\Phi\|$	$\|R\|$	M	T	$\|S\|$	$\|\mathcal{F}\|$	C	$\|\Phi\|$	$\|R\|$
Blocks-clear	2	0.26	22	503	2	2	2	1	0.04	5	129	2	3	2	2	0.10	5	129	2	2	2
Blocks-on	–	–	–	–	–	–	–	53	20.60	22	1804	7	3	4	38	15.16	20	1661	5	4	2
Delivery	–	–	–	–	–	–	–	12	39.18	48	936	15	4	2	8	10.42	28	936	15	4	2
Gripper	3	0.43	28	458	4	2	4	4	3.00	28	458	4	3	2	6	1.44	34	463	4	3	2
Miconic	14	12.48	36	855	6	3	4	4	0.40	18	429	6	3	2	6	0.31	18	429	6	4	2
Reward	3	0.47	14	519	6	2	2	1	0.13	8	458	6	2	2	3	0.84	12	739	6	2	2
Spanner	4	1.43	19	488	9	3	3	10	8.75	96	563	10	4	2	17	16.74	76	530	10	4	2
Visitall	11	10.22	22	1005	9	2	2	3	1.69	9	760	9	2	2	5	4.15	11	859	9	2	2

Table 1: Learning hierarchical policies Π_k for a class of problems $\mathcal{Q}$. This involves learning sketches $R(n)$ for the class of problems $\mathcal{Q}_n$ of width k, starting with the root node n and $\mathcal{Q}_n = \mathcal{Q}$, assigning the rules in $R(n)$ to the children n' of n, and then learning sketches for the classes of subproblems $\mathcal{Q}_{n'}$ that have width $k-1$, until the nodes have subproblems of width 0. We report the peak memory in GiB during learning (M), and the total CPU time in hours spent on learning (T). (We use the parallel mode of Clingo, so the actual wall-clock times are much lower.) For the encodings in the last iteration of the learning procedure, we report the largest number of states ($|S|$), and the largest number of features ($|\mathcal{F}|$). For the resulting collection of sketches, we report the largest complexity of a feature $f \in \Phi$ (C), the number of features ($|\Phi|$), and the maximum branching factor ($|R|$). We use "–" to indicate that the learning procedure failed because of insufficient time.

	LAMA		BFWS		R_1		R_2		Π_0		Π_1		Π_2	
Domain	S	T	S	T	S	T	S	T	S	T	S	T	S	T
Blocks-clear	30	32	30	1015	30	61	30	62	30	30	30	30	30	29
Blocks-on	30	23	23	586	30	341	4	180	–	–	30	25	30	23
Delivery	4	999	28	143	30	68	0	–	–	–	30	22	30	22
Gripper	30	2	30	6	30	5	30	317	30	2	30	2	30	2
Miconic	30	7	30	22	30	5	30	80	30	7	30	7	30	7
Reward	30	381	30	36	30	12	30	12	30	37	30	41	30	39
Spanner	0	–	0	–	30	1440	30	1395	30	11	30	11	30	11
Visitall	29	189	25	735	30	23	30	20	30	685	30	675	30	783

Table 2: Testing the learned hierarchical policies Π_k. We compare them against SIW_R with sketches R_1, R_2 of width $1, 2$, and two domain-independent planners LAMA and BFWS. We show the number of solved instances out of 30 (S), and the maximum time in seconds for solving an instance for which all algorithms find a solution, excluding algorithms that solve no instance at all (T). We use "–" to indicate that a planner failed to find any solution or no policy (sketch) was learned.

built on the notions of general policies, sketches, and width developed for classical planning, to define valid hierarchical policies for a class of problems $\mathcal{Q}$ as trees, where every node n is associated with a sketch rule $r(n)$ and a class of subproblems $\mathcal{Q}_n$. The sketch $R(n)$ at node n, given by the rules $r(n')$ of the children nodes decomposes the problems in $\mathcal{Q}_n$ into classes of subproblems $\mathcal{Q}'_n$ of smaller width, starting with $\mathcal{Q}_{n_0} = \mathcal{Q}$ for the root node n_0, and ending with classes of zero width at the leaves. We have also developed a method for learning hierarchical policies, reported experimental results, and established some of their formal properties. The current limitations result from the reliance on particular feature pools, derived from the domain predicates, and the scalability of combinatorial solvers. Similar issues have been addressed recently in the setting of generalized planning by considering a different class of solvers based on deep learning (Ståhlberg, Bonet, and Geffner 2022).

Acknowledgments

The research of H. Geffner has been supported by the Alexander von Humboldt Foundation with funds from the Federal Ministry for Education and Research. The research has also received funding from the European Research Council (ERC), Grant agreement No. No 885107, and Project TAILOR, Grant agreement No. 952215, under EU Horizon 2020 research and innovation programme, the Excellence Strategy of the Federal Government and the NRW Länder, and the Knut and Alice Wallenberg (KAW) Foundation under the WASP program. The computations were enabled by resources provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS) and the Swedish National Infrastructure for Computing (SNIC) at National the Supercomputer Centre at Linköping University partially funded by the Swedish Research Council through grant agreements no. 2022-06725 and no. 2018-05973.

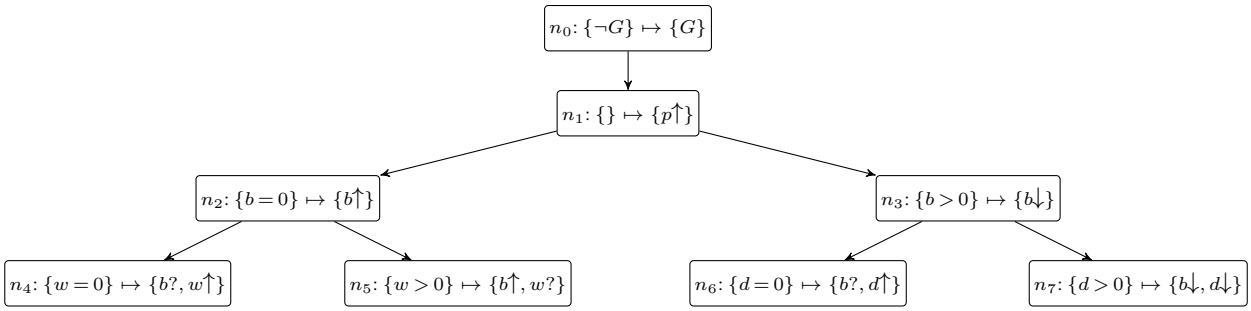

Figure 3: Hierarchical policy Π_{Miconic} for $\mathcal{Q}_{\text{Miconic}}$ over the set of features $\{G, b, d, p, w\}$ where G is true iff all people are served, w is the number of people that can be boarded in the lift from their origin floor, b is the number of boarded people, d is the number of boarded people that can unboarded at their destination floor, and p is the number of served people. The meanings are "serve all people" (n_0), "serve people" (n_1), "move people from origin floor into lift" (n_2), "move people from lift to destination floor" (n_3), "move lift to origin floor with waiting people" (n_4), "board people at origin floor" (n_5), "move lift to destination floor" (n_6), and "unboard people at destination floor" (n_7).

References

Baader, F.; Calvanese, D.; McGuinness, D. L.; Nardi, D.; and Patel-Schneider, P. F., eds. 2003. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press.

Barto, A. G.; and Mahadevan, S. 2003. Recent Advances in Hierarchical Reinforcement Learning. *Discrete Event Dynamic Systems*, 13(1): 41–77.

Belle, V.; and Levesque, H. J. 2016. Foundations for Generalized Planning in Unbounded Stochastic Domains. In *Proc. KR*, 380–389.

Bonet, B.; Francès, G.; and Geffner, H. 2019. Learning Features and Abstract Actions for Computing Generalized Plans. In *Proc. AAAI 2019*, 2703–2710.

Bonet, B.; and Geffner, H. 2018. Features, Projections, and Representation Change for Generalized Planning. In *Proc. IJCAI 2018*, 4667–4673.

Bonet, B.; and Geffner, H. 2020. Qualitative Numeric Planning: Reductions and Complexity. *Journal of Artificial Intelligence Research*, 69: 923–961.

Bonet, B.; and Geffner, H. 2021. General Policies, Representations, and Planning Width. In *Proc. AAAI 2021*, 11764–11773.

Bonet, B.; Palacios, H.; and Geffner, H. 2009. Automatic Derivation of Memoryless Policies and Finite-state Controllers Using Classical Planners. In *Proc. ICAPS 2009*, 34–41.

Celorrio, S. J.; Segovia-Aguas, J.; and Jonsson, A. 2019. A review of generalized planning. *Knowl. Eng. Rev.*, 34: e5.

Dietterich, T. G. 2000. Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition. *Journal of Artificial Intelligence Research*, 13: 227–303.

Drexler, D.; Francès, G.; and Seipp, J. 2022. Description Logics State Features for Planning (DLPlan). https://doi.org/10.5281/zenodo.5826139.

Drexler, D.; Seipp, J.; and Geffner, H. 2021. Expressing and Exploiting the Common Subgoal Structure of Classical Planning Domains Using Sketches. In *Proc. KR 2021*, 258–268.

Drexler, D.; Seipp, J.; and Geffner, H. 2022. Learning Sketches for Decomposing Planning Problems into Subproblems of Bounded Width. In *Proc. ICAPS 2022*, 62–70.

Drexler, D.; Seipp, J.; and Geffner, H. 2023. Code and Data for Learning Hierarchical Policies. https://doi.org/10.5281/zenodo.7725701.

Erol, K.; Hendler, J. A.; and Nau, D. S. 1994. HTN Planning: Complexity and Expressivity. In *Proc. AAAI 1994*, 1123–1128.

Francès, G.; Bonet, B.; and Geffner, H. 2021. Learning General Planning Policies from Small Examples Without Supervision. In *Proc. AAAI 2021*, 11801–11808.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2019. Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming*, 19: 27–82.

Hogg, C.; Muñoz-Avila, H.; and Kuter, U. 2016. Learning Hierarchical Task Models from Input Traces. *Computational Intelligence*, 32(1): 3–48.

Hu, Y.; and De Giacomo, G. 2011. Generalized Planning: Synthesizing Plans that Work for Multiple Environments. In *Proc. IJCAI*, 918–923.

Icarte, R. T.; Klassen, T. Q.; Valenzano, R.; and McIlraith, S. A. 2022. Reward Machines: Exploiting Reward Function Structure in Reinforcement Learning. *Journal of Artificial Intelligence Research*, 73: 173–208.

Illanes, L.; and McIlraith, S. A. 2019. Generalized Planning via Abstraction: Arbitrary Numbers of Objects. In *Proc. AAAI*, 7610–7618.

Junyent, M.; Gómez, V.; and Jonsson, A. 2021. Hierarchical width-based planning and learning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, 519–527.

Knoblock, C. A. 1994. Automatically Generating Abstractions for Planning. *Artificial Intelligence*, 68(2): 243–302.

LeCun, Y. 2022. A Path Towards Autonomous Machine Intelligence. https://openreview.net/forum?id=BZ5a1r-kVsf.

Lipovetzky, N.; and Geffner, H. 2012. Width and Serialization of Classical Planning Problems. In *Proc. ECAI 2012*, 540–545.

Lipovetzky, N.; and Geffner, H. 2017. Best-First Width Search: Exploration and Exploitation in Classical Planning. In *Proc. AAAI 2017*, 3590–3596.

Machado, M. C.; Bellemare, M. G.; and Bowling, M. 2017. A Laplacian Framework for Option Discovery in Reinforcement Learning. In *Proc. ICML 2017*, 2295–2304.

Marthi, B.; Russell, S.; and Wolfe, J. A. 2007. Angelic Semantics for High-Level Actions. In *Proc. ICAPS 2007*, 232–239.

McGovern, A.; and Barto, A. G. 2001. Automatic Discovery of Subgoals in Reinforcement Learning using Diverse Density. In *Proc. ICML 2001*, 361–368.

Parr, R.; and Russell, S. 1997. Reinforcement Learning with Hierarchies of Machines. In *Proc. NeurIPS*, 1043–1049.

Richter, S.; and Westphal, M. 2010. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *Journal of Artificial Intelligence Research*, 39: 127–177.

Sacerdoti, E. D. 1974. Planning in a Hierarchy of Abstraction Spaces. *Artificial Intelligence*, 5: 115–135.

Seipp, J.; Torralba, Á.; and Hoffmann, J. 2022. PDDL Generators. https://doi.org/10.5281/zenodo.6382173.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2008. Learning Generalized Plans Using Abstract Counting. In *Proc. AAAI 2008*, 991–997.

Srivastava, S.; Zilberstein, S.; Immerman, N.; and Geffner, H. 2011. Qualitative Numeric Planning. In *Proc. AAAI 2011*, 1010–1016.

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2022. Learning Generalized Policies without Supervision Using GNNs. In *Proc. KR 2022*, 474–483.

Ståhlberg, S.; Francès, G.; and Seipp, J. 2021. Learning Generalized Unsolvability Heuristics for Classical Planning. In *Proc. IJCAI 2021*, 4175–4181.

Sutton, R. S.; Precup, D.; and Singh, S. 1999. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112: 181–211.

Tate, A. 1977. Generating Project Networks. In *Proc. IJCAI*, 888–893.

Torralba, Á.; Seipp, J.; and Sievers, S. 2021. Automatic Instance Generation for Classical Planning. In *Proc. ICAPS 2021*, 376–384.

Zhuo, H. H.; Muñoz-Avila, H.; and Yang, Q. 2014. Learning hierarchical task network domains from partially observed plan traces. *Artificial Intelligence*, 212: 134–157.