Leveraging Demonstrations for Learning the Structure and Parameters of Hierarchical Task Networks

Philippe Hérail, Arthur Bit-Monnot
LAAS-CNRS, Université de Toulouse, INSA, CNRS, Toulouse, France
philippe.herail@laas.fr, abitmonnot@laas.fr

Abstract
Hierarchical Task Networks (HTNs) are a common formalism for automated planning, allowing to leverage the hierarchical structure of many activities. While HTNs have been used in many practical applications, building a complete and efficient HTN model remains a difficult and mostly manual task.

In this paper, we present an algorithm for learning such hierarchical models from a set of demonstrations. Given an initial vocabulary of tasks and accompanying demonstrations of possible ways to achieve them, we present how each task can be associated with a set of methods capturing the knowledge of how to achieve it. We focus on the algorithms used to learn the structure of the model and to efficiently parameterize it, as well as an evaluation in terms of planning performance.

Introduction
Hierarchical Task Networks (HTNs) (Erol, Hendler, and Nau 1994) are an approach to automated planning that combine a declarative action-based model for describing the primitive actions achievable by a system with procedural knowledge describing how those primitives can be combined to achieve high-level tasks. Even though these hierarchical models are a formalism that allows to plan more efficiently while remaining interpretable by human engineers, it is cumbersome to design such models from scratch. To address this issue, we intend to allow the agent to learn such HTN models from previously observed execution traces, and in particular the ones resulting from a tutor’s demonstration.

The goal of such a learning system would be to be able to solve any previously demonstrated tasks through a solution of at least equivalent quality to the demonstrated one. It should also be able to generalize the demonstrations to solve new unseen tasks, or previously demonstrated tasks in a new environment. This is done by learning, for any given task in the considered domain, a set of methods that achieve the high-level objectives associated to the task. This set of methods should cover all possible ways of achieving this task with the exception of clearly suboptimal ways. Any method should be associated with a validity scope that defines whether it is applicable in a given state. When applicable, it should achieve the task.

Intuitively, if a learned hierarchical planning model has these desirable properties, an automated planner facing a task to achieve could greedily pick any applicable method and have the guarantee that it will fulfill the objectives associated to the task. While this might lead to suboptimal behavior, classical search mechanisms would allow an automated planner to derive an optimal solution.

The objective of this paper, extending our previous work (Hérail and Bit-Monnot 2022), is to present a method for building parameterized hierarchical planning models based on a set of demonstrations, each demonstration associating a high-level task to an execution trace: a sequence of primitive actions achieving it.

Related Work
Over the years, several approaches have been developed to learn hierarchical planning models.

HTN-MAKER (Hogg, Muñoz-Avila, and Kuter 2008) and HTNLearn (Zhuo, Muñoz-Avila, and Yang 2014) are both approaches aiming at learning parameterized HTNs from demonstrations, the latter of which was later extended to support partial and disordered input traces (Zhuo, Peng, and Kambhampati 2019). Both these methods require as input, in addition to the demonstrations, demonstrated tasks and subtasks annotated with preconditions and effects. The first method learns the models through goal regression while the second formulates the problem as one of maximal constraint satisfaction.

Other approaches exist that do not need subtasks as input, such as the work by Li et al. (2014) or CircuitHTN (Chen et al. 2021). These methods, however, learn HTN structures without parameters and method preconditions, limiting their ability to generalize to new environments.

Recently, the learning of Hierarchical Goal Networks (HGNs) structure instead of HTNs, has been proposed as a preliminary work by Fine-Morris and Muñoz-Avila (2019), leveraging a vector representation of the states and unsupervised learning procedures to learn such networks while limiting the burden of annotating demonstration data. The most recent work in this area focuses on numeric goals and preconditions (Fine-Morris et al. 2022).
Due to their similarities with HTNs, some work aiming at learning grammars is relevant and in particular the work on learning Combinatory Categorial Grammars (CCGs) for plan and goal recognition (Geib and Goldman 2011; Kantharaju, Ontañón, and Geib 2019). While the learned CCGs are not practically usable for planning, the authors propose several ideas for extracting interesting patterns from a set of execution traces.

Learning Problem

Hierarchical Planning Model

We define a hierarchical planning model $H$ as a lifted HTN structure which can be written as a tuple $H = (T, A, M)$ where $T$ is a set of abstract tasks, $A$ a set of primitive actions and $M$ a set of possible methods decomposing the tasks $t \in T$ into subtasks $\{t_d | t_d \in \{T \cup A\}\}$. Figure 1 shows a simple task hierarchy as an illustration.

A primitive task (or action) $a \in A$ models the basic acting capabilities of the agent, and represents directly executable primitives. They are represented using an identifying symbol and a set of parameters, such as $a = \text{action}_\text{name}(\text{arg}_1, \ldots, \text{arg}_n)$. Actions are associated with preconditions and effects that allow verifying the validity of a plan.

An abstract (or non-primitive) task $t \in T$ is associated with a set of methods $M_t$, that allow decomposing it and possible postconditions representing the predicates that must hold after executing $t$ for it to be considered a success. Similarly to actions, they are represented using an identifying symbol and a set of arguments.

A method $m \in M_t$ is a tuple $m = (\text{Pre}_m, \text{N}_m)$, where $\text{Pre}_m$ are the preconditions of the method, and $\text{N}_m$ is a sequence of subtasks in $\{T \cup A\}$, representing a possible decomposition of $t$. This totally-ordered task network represents a way to achieve the task $t$ and is only applicable in the current state if its preconditions $\text{Pre}_m$ hold.

For a given a planning domain $H$, a planning problem is an initial task network $\text{N}_p$, representing the activity that must be carried out and a initial state $s_0$ described by a set of boolean state variables. We consider that at any instant, the current state $s$ is fully observable and that it only evolves when a primitive action is executed (i.e. there are no exogenous events).

Learning of a Planning Model

Inputs to the Learning Problem

For the learning problem itself, we consider as input a fixed set $A$ of primitive actions as well as a vocabulary of non-primitive tasks $T_I$. For each primitive action, the learner knows its symbol and parameters but is not given any knowledge of its preconditions of effects.

For each task $t_f \in T_I$, the agent is given a set $D_{t_f}$ of demonstration traces from the tutor. Each trace $d \in D_{t_f}$ is an alternating sequence of states and tasks (either primitive or non-primitive), starting from a given initial state and ending in a final state in which the task $t_f$ has been successfully achieved. $d$ is considered optimal and maximally abstract with regard to the initial task vocabulary: for every demonstrated task, no other more abstract task from the initial vocabulary $T_I$ may be used to abstract a subsequence of $d$, and each demonstration is optimal according to a chosen metric. For a case where actions are uniform in cost, one may naturally consider the total number of primitive actions required to achieve $t_f$ as the optimality metric.

Learner Objectives

The primary objective of the learning process is to produce a hierarchical model that is capable of solving planning problems that were not part of the demonstrated set. This completeness property is intrinsic to the model and defines whether the model is theoretically capable of solving any possible problem of the domain at hand.

The quality of a planning model $H$ is however tightly coupled with the ability of an automated planner to exploit it to quickly derive solution plans. In particular, for any given planner we are aiming at maximizing the efficiency of the planner for solving a problem given $H$, which is typically measured as the runtime of the planner. This leads us to define the coverage of a learned model as the ratio of solved problems by a given planner under computational limits.

Approach to Model Learning

Requirements of Model Selection

Let us now give an initial intuition about the shape of models that could be learned and the implication for the learning process. Figure 2 presents several possible models (figures 2b-2e) that could be generated based on two example sequences (figure 2a).

The first one (2b) allows the choice of any of the four primitive actions $\{a, b, c, d\}$, each placed in a specific method. This model relies on a recursive call to $t$ to repropose the same choice until the task’s postconditions are achieved. While this model allows building any sequence of actions it does not help the agent towards a meaningful sequence based on demonstrations. The second model (2c) takes the opposite approach and records each known trace into a method. This model is obviously strongly tied to the demonstration set and would fail to generalize to new problems. In between these two extremes, we have the models (2d) and (2e) that present different options to abstract common subsequences. The former encodes the repeated $a \ b$ sequence in a single method and relies on the recursive call to complete the sequence. The latter delays the choice between $c$ and $d$ to after the execution of $a$ and $b$, using a synthetic task $t_s$.

These four models are just a handful of examples among the many possible models that could be generated. Denoting as $\Theta$ the set of possible models, the objective of a learning
system is to find, or at least approach, the optimal model \( \theta^* \in \Theta \)

\[
\theta^* = \arg \min_{\theta \in \Theta} \text{cost}(\theta)
\]

where \( \text{cost}(\theta) \) is a function that measures the cost of a particular model and should typically account for the size of the model as well as its capacity to solve both demonstrated and unseen problems. With this in mind we now turn our attention to the characterization of the set of possible models \( \Theta \). In a later section, we will propose a cost function to evaluate the models.

**Generation of Candidate Planning Models**

At a high level, the goal of the learning problem is to generate a model where some subtasks group together common behaviors, with a sensible parameterization of methods depending on the current task, as well as reasonable preconditions to limit the search effort of the planning engine.

The overview of our process for achieving this goal from a set \( D \) of demonstrations is presented in algorithm 1.

---

**Algorithm 1 Planning Model Search**

1: \( h \leftarrow \text{GENERATE BASE MODEL} \)

2: while \( \text{QUALITY}(h) \) improves do

3: \( H_c \leftarrow \text{GEN CANDIDATE MODELS STRUCTS}(h, D) \)

4: for all \( h_c \in H_c \) do

5: \( h_c \leftarrow \text{EXTRACT MODEL PARAMETERS}(h_c, D) \)

6: \( \tilde{h} \leftarrow \text{FIND BEST MODEL}(H_c \cup \{h\}) \)

---

**Structure Generation**

Model structures are generated through the exploration of the neighborhood of the current best model, using the following operators.

In order to quickly progress towards a useful structure, one of the operators is implemented using a procedure similar to the one described in HTN-MAKER (Hogg, Muñoz-Avila, and Kuter 2008). As we consider only optimal demonstrations and totally ordered subtasks, we do not need to consider a task’s postconditions, and consider that each subsequence going up to the end of a demonstration of a task \( t \) is a way to achieve \( t \). Furthermore, as the parameters will be extracted in the next step, we only consider the demonstration as a sequence of task symbols, removing the need for complex method subsumption detection techniques, replacing it with a removal of duplicate symbol sequences.

While this procedure does provide methods that will always be useful in some case for achieving the task for which it was learned, it does not provide multiple hierarchy levels. Therefore, we designed a new operator to allow grouping some behaviors into new tasks. In order to focus the search on relevant parts of the search space, we use frequent patterns as the key to the GoKrimp algorithm (Lam et al. 2014), which is based on the Minimum Description Length (MDL) (Grünwald 1996) concept, incrementally finding the patterns that most compress the sequence dataset.

In our case, we extract patterns from a set of demonstrations \( D \) through the function described in algorithm 2, with three parameters \( l, k, n_c \in \mathbb{N} \). We consider again demonstrations as sequences of task symbols, and try to extract patterns as regular expressions (regexps). An example of mapping from a a regexp pattern to a hierarchical representation is presented in figure 3b. \( \text{len}(p) \) is defined as the number of symbols in \( p \), excluding any regexp operator.

Patterns are extracted by incrementally building a set \( P_1 \) of patterns (\( |P_1| \leq k \)), each iteration extracting the most compressing pattern \( p \) such that \( \text{len}(p) \leq l \) and then replacing all the matches of \( p \) in \( D \) (see an example figure 3a).

The pattern generation function is detailed in algorithm 3. We define the \( \text{COMPRESSED SIZE}(D, p) \) function as the function that returns the size of the demonstration set \( D \) compressed using pattern \( p \) as in the work of Lam et al. (2014), and the function \( \text{CONCAT}(p_1, p_2) \) as the one that generates a new pattern by appending \( p_2 \) to \( p_1 \).
Algorithm 2 GENERATE PATTERNS($D, k, l, n_c$)

1: $P_f \leftarrow \emptyset$
2: while $|P_f| \leq k$ do
3: $p \leftarrow$ MOST COMPRESSING PATTERNS($D, l, n_c$)
4: $P_f \leftarrow P_f \cup \{p\}$
5: SUBSTITUTE PATTERNS($D, p$)
6: return $P_f$

Algorithm 3 MOST COMPRESSING PATTERNS($D, l, n_c$)

1: $P_a \leftarrow$ TASK SYMBOLS($D$)
2: $P_c \leftarrow$ EXISTING PATTERNS($D$)
3: $P \leftarrow$ GENERATE CHOICE PATTERNS($P_c, n_c$)
4: $P \leftarrow P_a \cup P_c \cup P$
5: $P \leftarrow P; R \leftarrow \{?, \ast, +\}$
6: repeat
7: $P' \leftarrow P$
8: for all $(p, r) \in P \times R$ do
9: $P' \leftarrow P' \cup \{\text{CONCAT}(p, r)\}$
10: $P'' \leftarrow P'$
11: for all $(p, p_e) \in P' \times P$ do
12: $P'' \leftarrow P'' \cup \{\text{CONCAT}(p, p_e)\}$
13: $p^* \leftarrow \arg \min_{p \in P''} \text{COMPRESSED SIZE}($$D, p)$
14: $P \leftarrow \{p^*\}$
15: until COMPRESSED SIZE($D, p^*$) stops improving or $\text{len}(p^*) \geq l$
16: return $p^*$

Model Structure Demonstration Matching

In order to use the demonstrations to parameterize a candidate model structure $C$, we need to find a hierarchical mapping between a given demonstration $d$ and a top level task $t_d$, through a decomposition of this task as defined in $C$. This matching is done through an adaptation of the technique using HTN planning for plan verification developed by Höller et al. (2021).

Model Parameterization

For each method in the model, we need to identify the parameters that should be passed to its subtasks. Furthermore, for synthetic tasks, we need to determine the tasks’ parameters themselves.

The parameterization process is then as follows:

1. Identify a superset of the possible parameters for synthetic tasks and methods from the primitive tasks’.
2. Express the parameterization problem into one of MAX-SMT (Nieuwenhuis and Oliveras 2006) and solve it, with the objective of minimizing the number of parameters in the final model.
3. Add a new step to remove parameters not used to constrain methods instantiations, to reduce the search space during the planning phase.

Parameter Generation

To extract the set of possible parameters for a model, we start by splitting it into sub-models, each comprised of a top-level non-primitive task, its methods and their direct subtasks (primitive and non-primitive). Such a sub-model is presented in Figure 4a, where $t$ is a synthetic task for which two methods $m_1$ and $m_2$ were learned.

Algorithm 4 PROPAGATE ARGS UPWARDS($h_{sub}$)

1: for all $m \in M_h$ do
2: for all $\hat{p} \in \text{args}($SUBTASKS$(m))$ do
3: $\text{args}(m) \leftarrow \text{args}(m) \cup \{\hat{p}\}$
4: if $m \notin M_p$ then
5: $\hat{p}^* \leftarrow (p, M_p \cup \{m\})$
6: $\text{args}(t_h) \leftarrow \text{args}(t_h) \cup \{\hat{p}^*\}$

To extract the arguments for the set of subhierarchies $H_{sub}$ corresponding to an HTN $H$, arguments are propagated upwards from the subtasks to the top level task for each subhierarchy, as described in algorithm 4. Non-primitive subtasks in the subhierarchies are then updated to keep a consistent signature. The process is repeated until a fixed point is reached.

To enforce termination of the algorithm in the case of recursive task definitions, we augment each parameter $p$ with the set of methods $M_p$, it has been propagated through, defining $\hat{p} = (p, M_p)$. We also define a condition to allow methods to act as filters (alg. 4 line 4), preventing methods from crossing twice a given method boundary. This behavior is illustrated in the example figure 4, where $\ast$ and $\dagger$ superscripts (associated with $m_1$ and $m_2$ respectively) are used to visualize the set $M_p$ for each parameter $p$. In figure 4d, $A_{1,t}$ and

Pattern generation starts by initializing a set $P$ with the task symbols in the demonstrations, the previously generated patterns and a set of choice patterns, as well as a set $R$ containing the standard regex operators $\{?, \ast, +\}$. The choice patterns are generated by taking a random set of $n_c$ existing patterns and combining them together using the regex operator. $P$ is used to initialize the set of candidate patterns $P_c$. The set of candidate patterns is then extended by adding possible regex patterns (line 8), and this new set is then extended again by adding potential following task or patterns (line 11). The best compressing pattern of the set is then kept and the process is repeated until either the compression stops improving anymore or the pattern length reaches the limit $l$.

The set of candidate patterns and a set of choice patterns, as well as a set

...patterns and a set of choice patterns, as well as a set $R$ containing the standard regex operators $\{?, \ast, +\}$. The choice patterns are generated by taking a random set of $n_c$ existing patterns and combining them together using the regex operator. $P$ is used to initialize the set of candidate patterns $P_c$. The set of candidate patterns is then extended by adding possible regex patterns (line 8), and this new set is then extended again by adding potential following task or patterns (line 11). The best compressing pattern of the set is then kept and the process is repeated until either the compression stops improving anymore or the pattern length reaches the limit $l$. 

Figure 3: Pattern substitution and hierarchical representation example for $p = a+b$.
$A_{2,t}$ are not propagated upwards from $m_1$ as they are originating from $A_1$ and $A_2$ via method $m_1$.

**Parameter Deduplication** The parameter generation process aims at being exhaustive and covering all possible cases, regardless of the demonstrations traces at hand. In many cases, this results in many more parameters than actually needed to regenerate the traces. We cast the problem of deduplicating arguments as MAX-SMT with equality logic and interpreted functions, under the objective of maximizing the number of unifications permitted by the demonstration traces. At a high level, we have three kinds of constraints on the sub-hierarchies’ arguments, extracted from the sub-hierarchies and examples:

- **Structural hard constraints**, to enforce consistency between a task reference definition (when considered as a top level one) and its occurrences as methods’ subtasks. These constraints are defined as in the equation below:

$$\forall (i,j) \in |\text{args}(t)|, \text{arg}_i(t) = \text{arg}_j(t) \Rightarrow \text{arg}_i(t_{sub}) = \text{arg}_j(t_{sub})$$

- **Inequality hard constraints**, added when the demonstrations show that two parameters must be distinct in a given instantiation.

- **Equality soft constraints**, extracted from all the examples that show two parameters that are identical in a given example.

Figure 5 shows some constraints that can be extracted through a combination of the structure of the hierarchy presented figure 4e and the information contained in a given set of demonstrations. Figure 5c shows the extracted model after simplification using the demonstrations set $D$ presented figure 5a and solving the associated constraint system. Some constraints are presented in figure 5b as an example.

**Model Quality Evaluation**

To evaluate the quality of the learned model, we use the metric described in our previous work (Herail and Bit-Monnot 2022), based on the MDL principle (Grünwald 1996). This metric exploits data compression as a way to drive the model search towards abstracting redundant parts in the demonstrations. The quality of a given planning model $H$ is defined as the weighted sum of the model size ($L_{\text{model}}(H)$) and the demonstration dataset size, reconstructed using $H$ ($L_{\text{dem}}(D|H)$), as presented below:

$$L(H, D) = \alpha L_{\text{model}}(H) + L_{\text{dem}}(D|H) \quad (1)$$

At this point we have generated several models, each of which has been parameterized. The metric above allows choosing a single model among the candidates. This selected model will be used as the baseline for the next iteration, where it will typically be extended with new tasks and methods, to cover more demonstrations traces. The process stops once this metric shows no improvement, meaning that the model modifications stop improving the abstractions of the demonstrations.

**Learned Models Evaluation**

In order to assess the validity of our approach we tested our learner on a variety of planning domains:

**CHILDSNACK** A simple domain whose reference model only contains one task with two alternative methods with only primitive subtasks. The learning set consists of 50 demonstrations of the serving task.

**TRANSPORT** A standard deliver-with-trucks scenario. The learning set for this domain consists of 20 demonstrations of the delivery task.
LOGISTICS A variant of the TRANSPORT domain, extended with several cities connected via airplanes, only able to move to specific airport locations. This is the version that was used in the development of HTN-MAKER (Hogg, Muñoz-Avila, and Kuter 2008), modified to use typing instead of predicates. The learning set consists of 60 demonstrations of the delivery task.

For each domain, the models were learned from traces extracted from a random subset of the 2020 International Planning Competition (IPC) (Behnke, Höller, and Bercher 2021) instances or HTN-MAKER’s training set when applicable. Trace selection was biased towards simple problems in order to reduce learning times, which were all well under one hour (wall-clock time) using 6 threads on a portable workstation, equipped with 32 GB of RAM and an Intel Core i7-10850H CPU (6/12 @ 2.70GHz). Demonstration traces used for learning only decompose one single task, while the test instances may require several instantiations of the learned tasks to be solved to be considered a success.

Figure 6 shows planning times for unseen problems for several learned models with the Lilotane planner. Each model is a result of a different parameterization of the learner, included in the graphs’ legend. The proposed approach shows that it is possible to learn parameterized HTN domains that are close to handcrafted ones in terms of planning performance on a subset of the domains from the IPC competition.

Compared to other approaches such as HTN-MAKER (Hogg, Muñoz-Avila, and Kuter 2008) or HTNLearm (Zhao, Muñoz-Avila, and Yang 2014), the work needed from the tutor is limited, requiring only a limited number of demonstrations and no annotated intermediate tasks. These intermediate tasks are learned by abstracting common behaviors detected through frequent pattern mining, which our approach is then able to parameterize sensibly using the proposed MAX-SMT approach.

Furthermore, the nature of the search process allows to improve a learned model if new knowledge becomes available, by reusing the current model as the new starting point.

Despite the very limited information exploited by our algorithm, our experiments show that the learned models are already competitive with handwritten ones. Future work shall focus on improving the addition of the frequent patterns to the candidate structures during the search process in order to extract efficient ones in more complex domains. The parameter extraction process is also a planned improvement avenue, as considering the surrounding states during the refinement of a task could provide useful insights for detecting the most relevant parameters.

Conclusion & Future Work

The proposed approach shows that it is possible to learn parameterized HTN domains that are close to handcrafted ones in terms of planning performance on a subset of the domains from the IPC competition.
References


