Automatic Generation of HTNs From PDDL

Anders Jonsson
University Pompeu Fabra
C/Roc Boronat 138
08018 Barcelona, Spain
anders.jonsson@upf.edu

Damir Lotinac
University Pompeu Fabra
C/Roc Boronat 138
08018 Barcelona, Spain
damir.lotinac@upf.edu

Abstract

Hierarchical Task Networks, or HTNs, are a popular model in planning for representing tasks or decision processes that are organized in a hierarchy. Although HTNs are known to be at least as expressive as STRIPS planning, being expressive enough to represent highly complex decision processes is not the main reason for their popularity. On the contrary, by imposing ordering constraints on the tasks at each level of the hierarchy, an HTN can significantly simplify the search for an action sequence that achieves a desired goal.

In this paper we present a novel algorithm that automatically generates HTNs from PDDL, the standard language for describing planning domains. The HTNs that our algorithm constructs contain two types of composite tasks that interact to achieve the goal of a planning instance. One type of task achieves fluents by traversing the edges of invariant graphs in which only one fluent can be true at a time. The other type of task traverses a single edge of an invariant graph by applying the associated action, which first involves ensuring that the preconditions of the action hold. The resulting HTNs can be applied to any instance of a planning domain, and are provably sound, such that the solution to an HTN instance can always be translated back to a solution to the original planning instance. In several domains we are able to solve most or all planning instances using HTNs created from a single example instance.

Keywords: planning, classical planning, hierarchical task networks

1 Introduction

Hierarchical Task Networks, or HTNs, are a popular tool for encoding expert knowledge in the form of a task hierarchy into a planning domain. Each task in the hierarchy has several possible decompositions, each involving a fixed set of subtasks and an associated partial order on subtasks. The solution to an HTN is a sequence of decompositions and ordering choices that results in a sequence of primitive tasks that is applicable in the initial state. HTNs have been successfully used in a variety of applications, usually by severely restricting the ordering choices to simplify the search for a solution. In the extreme case, each task only has one possible decomposition in which case no search at all is necessary to find a solution. Properly designing an HTN can be a time-consuming task for a human expert, but once this work is done, there is little need to optimize search.

We ask the following question: is it possible to devise an automatic, domain-independent approach for generating HTNs automatically from a basic description of a planning domain? Although there have been earlier attempts to generate HTNs automatically [4, 7], these approaches rely on partial information about the task decomposition. In contrast, we generate HTNs directly from the PDDL encoding of a planning domain and a single representative instance. Our approach is to generate HTNs that encode invariant graphs, which are similar to lifted domain transition graphs but can be subdivided on types. In experiments, we tested our approach on planning benchmarks from the International Planning Competition (IPC). In four domains, our algorithm is able to construct HTNs that make it possible to efficiently solve any instance using blind search. The approach is partially successful in other domains, but the branching factor becomes a problem for large instances. Still, the experimental results partially answers our question in the affirmative.

2 Planning Domains

We consider the fragment of PDDL that models typed STRIPS planning domains with positive preconditions and goals. A planning domain is a tuple $\boldsymbol{d} = \langle \mathcal{T}, \prec, P, A \rangle$, where $\mathcal{T}$ is a set of types, $\prec$ an inheritance relation on types, P a set of predicates and A a set of actions. Each predicate $p \in P$ and action $a \in A$ has a parameter list ($\varphi(p)$ and $\varphi(a)$, respectively) whose elements are types. Each action $a \in A$ has a precondition $\mathrm{pre}(a)$, an add effect $\mathrm{add}(a)$, and a delete effect $\mathrm{del}(a)$. Each precondition and effect consists of a predicate p and a mapping from $\varphi(p)$ to $\varphi(a)$.

Given a domain $\boldsymbol{d}$, a STRIPS planning instance is a tuple $\boldsymbol{p} = \langle \Omega, I, G \rangle$, where Ω is a set of objects, I is an initial state, and G is a goal state. Instance $\boldsymbol{p}$ implicitly defines a set F of propositional variables or *fluents* by assigning objects in Ω of the appropriate type to the parameters of each predicate, and a set O of *operators* by assigning objects in Ω to the parameters of each action. The initial state $I \subseteq F$ and goal state $G \subseteq F$ are both subsets of fluents.

Each operator $o \in O$ has a precondtion $\mathrm{pre}(o) \subseteq F$, and add effect $\mathrm{add}(o) \subseteq F$ and a delete effect $\mathrm{del}(o) \subseteq F$, each a subset of fluents instantiated from the preconditions and effects of the associated action a. A state $s \subseteq F$ is a subset of fluents that are true, while fluents in $F \setminus s$ are false. An operator $o \in O$ is applicable in s if and only if $\mathrm{pre}(o) \subseteq s$, and the result of applying o in s is a new state $s \ltimes o = (s \setminus \mathrm{del}(o)) \cup \mathrm{add}(o)$. A plan for $\boldsymbol{p}$ is a sequence of operators $\pi = \langle o_1, \ldots, o_n \rangle$ such that o_i, $1 \le i \le n$, is applicable in $I \ltimes o_1 \ltimes \cdots \ltimes o_{i-1}$, and π solves $\boldsymbol{p}$ if it reaches the goal state, i.e. if $G \subseteq I \ltimes o_1 \ltimes \cdots \ltimes o_n$.

3 Hierarchical Task Networks

We introduce a notation for HTN domains inspired by Geier and Bercher [2]. An HTN domain is a tuple $\boldsymbol{h} = \langle P, A, C, M \rangle$, where P is a set of predicates, A is a set of actions (i.e. primitive tasks), C is a set of compound tasks and M is a set of decomposition methods. Each task $c \in C$ and method $m \in M$ has an associated parameter list $\varphi(c)$ and $\varphi(m)$, respectively. Unlike STRIPS domains, HTN domains are untyped and we allow negative preconditions. A *task network* is a tuple $tn = \langle T, \prec \rangle$, where $T \subseteq A \cup C$ is a set of tasks and $\prec$ is a partial order on T. A method $m = \langle c, tn_m, \mathrm{pre}(m) \rangle$ consists of a compound task $c \in C$, a task network $tn_m = \langle T_m, \prec_m \rangle$ and a precondition $\mathrm{pre}(m)$. Each precondition $p \in P$ and task $t \in T_m$ has an associated mapping from $\varphi(p)$ or $\varphi(t)$ to $\varphi(m)$.

Given $\boldsymbol{h} = \langle P, A, C, M \rangle$, an HTN instance is a tuple $\boldsymbol{s} = \langle \Omega, I, tn_I \rangle$, where Ω is a set of objects, I is an initial state and tn_I is a task network. Just like for STRIPS, Ω induces sets F and O of fluents and operators, as well as sets $\mathcal{C}$ and $\mathcal{M}$ of grounded compound tasks and methods. A grounded task network has tasks in $O \cup \mathcal{C}$, and is *primitive* if all tasks are in O. The initial state $I \subseteq F$ is a subset of fluents, and the initial grounded task network $tn_I = \langle \{t_I\}, \emptyset \rangle$ has a single grounded compound task $t_I \in \mathcal{C}$.

We use $(s, tn) \rightarrow_D (s', tn')$ to denote that a pair of a state and a task network decomposes into another pair, where $tn = \langle T, \prec \rangle$ and $tn' = \langle T', \prec' \rangle$. A valid decomposition consists in choosing a task $t \in T$ such that $t' \not\prec t$ for each $t' \in T$, and applying one of the following rules:

1. If t is primitive, the decomposition is applicable if $\mathrm{pre}(t) \subseteq s$, and the resulting pair is given by $s' = s \ltimes t$, $T' = T \setminus \{t\}$ and $\prec' = \{(t_1, t_2) \in \prec \mid t_1, t_2 \in T'\}$.

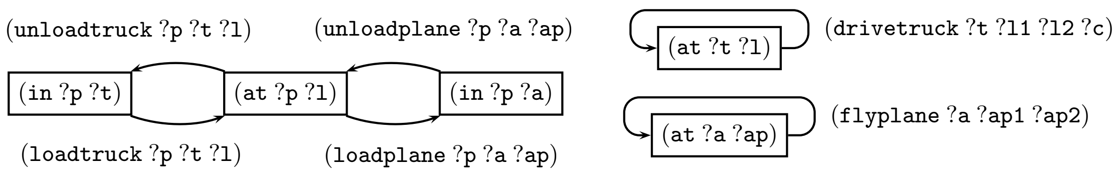

Figure 1: Invariant graphs in LOGISTICS.

2. If t is compound, the decomposition method $m = \langle t, tn_m, \mathrm{pre}(m)\rangle$ is applicable if $\mathrm{pre}(m) \subseteq s$, and the resulting pair is given by $s' = s$, $T' = T \setminus \{t\} \cup T_m$ and

$$\prec' = \{(t_1, t_2) \in \prec \mid t_1, t_2 \in T'\} \cup \{(t_1, t_2) \in T' \times T_m \mid (t_1, t) \in \prec\} \cup \{(t_2, t_1) \in T_m \times T' \mid (t, t_1) \in \prec\}.$$

The first rule removes a primitive task t from tn and applies the effects of t to the current state, while the second rule uses a method m to replace a compound task t with tn_m while leaving the state unchanged. If there is a finite sequence of decompositions from (s_1, tn_1) to (s_n, tn_n) we write $(s_1, tn_1) \rightarrow_D^* (s_n, tn_n)$. An HTN instance is solvable if and only if $(s_I, tn_I) \rightarrow_D^* (s_n, \langle\emptyset, \emptyset\rangle)$ for some state s_n, i.e. the resulting task network is empty.

4 Invariants

In STRIPS planning, a mutex invariant is a subset of fluents such that at most one is true at any moment. We generalize mutex invariants and use the Fast Downward planning system [3] to generate lifted (i.e. parameterized) invariants. We then use lifted invariants to construct *invariant graphs*. One reason our approach needs a representative instance is to test whether a lifted invariant corresponds to actual mutex invariants.

We illustrate the idea using the LOGISTICS domain. Fast Downward finds a single lifted invariant {(in ?o ?v), (at ?o ?p)}, i.e. a set of predicates with associated parameters. Parameters that appear across predicates (?o) are *bound* and take on the same value for all predicates. The remaining parameters (?v and ?p) are *free* and can be assigned any object of the appropriate type. Each assignment to bound parameters corresponds to a mutex invariant, formed by the set of fluents induced by all assignments to free parameters. The meaning of the lifted invariant for LOGISTICS is that across all instances, a given object ?o is either in a vehicle or at a location.

Given an invariant, our algorithm generates one or several invariant graphs by going through each action, finding each transition of each invariant that it induces (by pairing add and delete effects and testing whether the bound objects are identical), and mapping the types of the predicates to the invariant. We then either create a new invariant graph for the bound types or add nodes to an existing graph corresponding to the mapped predicate parameters.

Figure 1 shows the invariant graphs in LOGISTICS. In the top graph (G_1), the bound object is a package ?p, in the middle graph (G_2) a truck ?t, and in the bottom graph (G_3) an airplane ?a. Note that the predicate in is not actually part of the two bottom graphs, since trucks and planes cannot be inside other vehicles. Nevertheless, the invariant still applies: a truck or plane can only be at a single place at once. Each edge corresponds to an action that deletes one predicate of the invariant and adds another. To do so, the action has to include the parameters of both predicates, including the bound objects. In the figure, the invariant notation is extended to actions on edges, which have bound and free parameters.

5 Generating HTNs

In this section we describe our algorithm for automatically generating HTNs. The idea is to construct a hierarchy of tasks that traverse the invariant graphs to achieve certain fluents. In doing so there are two types of interleaved tasks: one that achieves a fluent in a given invariant (which involves applying a series of actions to traverse the edges of the graph), and one that applies the action on a given edge (which involves achieving the preconditions of the action).

Formally, our algorithm takes as input a STRIPS planning domain $\boldsymbol{d} = \langle \mathcal{T}, \prec, P, A\rangle$ and outputs an HTN domain $\boldsymbol{h} = \langle P', A', C, M\rangle$. The algorithm first constructs the invariant graphs $G_1, \ldots, G_k$ described above. Below we describe the components of the generated HTN domain $\boldsymbol{h}$.

5.1 Predicates and Tasks

The set $P' \supseteq P$ extends P with three predicates for each $p \in P$: persist-p, visited-p, and achieving-p. Respectively we use these predicates to temporarily cause p to persist, to flag p as an already visited node during search, and to prevent infinite recursion in case p or another predicate from the same invariant is currently being achieved.

```
(:method (achieve-p)
 ()
 ((set-flags-i) (achieve-p-i)
  (clear-flags-i) (lock-p)))
```

```
(:method (achieve-p-i)
 ((p') (¬visited-p'))
 ((visit-p') (do-p'-a-i) (achieve-p-i)))
```

```
(:method (do-p-a-i)
 ()
 (((achieve-p1)  ···  (achieve-pk))  (a)
  ((unlock-p1) ··· (unlock-pk))))
```

Figure 2: Outline of the generated methods.

Each action $a \in A$ from the input STRIPS planning domain ***d*** becomes a primitive task of ***h***. We add extra preconditions to ensure that a is not grounded on the wrong type, and that a does not delete a predicate that is supposed to persist. We also add primitive tasks for visiting, locking and unlocking each predicate $p \in P$. Visiting marks a predicate as visited, locking causes a predicate to temporarily persist, while unlocking frees a predicate so that it can be deleted again. For each invariant graph G_i, we add two primitive tasks `set-flags-`i that marks each predicate $p \in P$ in G_i as being achieved, and `clear-flags-`i that clears all flags for G_i.

We also include three types of compound tasks. For each predicate $p \in P$ that appears in any invariant graph, a task `achieve-`p that achieves p. For each invariant graph G_i and each $p \in P$ in G_i, a task `achieve-`p-i that achieves p in G_i. For each invariant graph G_i, each predicate $p \in P$ in G_i, and each outgoing edge of p (corresponding to an action $a \in A$), a task `do-`p-a-i. The first task is a wrapper task that achieves a predicate p in any invariant, while the other two are the interleaved tasks for achieving p by traversing the edges of an invariant graph G_i.

5.2 Methods

We describe the methods associated with each of the three types of compound tasks in turn. Since we use the HTN planner SHOP [6] to solve HTN instances, we outline each method in SHOP syntax in Figure 2.

The first type of task, `achieve-`p, has one associated method for each invariant graph G_i in which p appears. This method decomposes `achieve-`p into the task `achieve-`p-i, setting and clearing flags and locking p. The second type of compound task, `achieve-`p-i, has one associated method for each predicate p' in the invariant graph G_i and each outgoing edge of p' (corresponding to an action a that deletes p). Intuitively, one way to achieve p in G_i, given that we are currently at some different node p', is to traverse the edge associated with a using the compound task `do-`p'-a-i. Before doing so we mark p' as visited to prevent us from visiting p' again. After traversing the edge we recursively achieve p from the resulting node. To stop the recursion we define a "base case", a method that is applicable only when p holds and decomposes `achieve-`p-i into an empty task list.

The third type of compound task, `do-`p-a-i, has only one associated method that applies action a to traverse an outgoing edge of p in the invariant graph G_i. The tasks in the decomposition have to ensure that all preconditions $p_1, \ldots, p_k$ of a hold (excluding p, which holds by definition, as well as any static preconditions of a). Thus the method achieves all preconditions of a (locking them temporarily), then applies a, then unlocks the preconditions. To restrict the available choices when solving the HTN, we impose a total order on all tasks, except tasks (`achieve-`p_1) $\cdots$ (`achieve-`p_k) of the method `do-`p-a-i, since it may be difficult to determine in which order to achieve the preconditions of an action.

6 Optimizations

Achieving the preconditions of an action a in any order is inefficient since an algorithm solving the HTN instance may have to backtrack repeatedly. For this reason, we include an extension of our algorithm that uses a simple inference technique to compute a partial order in which to achieve the preconditions of a. We define a set of predicates whose value is supposed to persist, and check whether a path through an invariant graph is applicable given these persisting predicates. While doing so, only the values of bound variables are known, while free variables can take on any value. We match the bound variables of predicates and actions to determine whether an action allows a predicate to persist.

We use the same algorithm for precondition ordering to order a set of goal predicates P_G. We then order a set of fluents of each predicate $p \in P_G$ using a similar algorithm. To do so, the invariant graphs need to be partially grounded on each pair of fluents to be ordered. This partial grounding is done over the example instance only. The algorithm finds the indices of the parameters of p that invalidate the invariant, and generalize to any given instance of the planning domain.

7 Results

We ran our algorithm on the 9 typed STRIPS planning domains from IPC-2000 and IPC-2002, in order to directly compare the performance of our automatically generated HTNs with hand-crafted HTNs. Since hand-crafted planners were not allowed to compete in later competitions, there exist no hand-coded HTNs from those competitions to compare to.

	HTNPrecon			HTNGoal			FDBlind			Hand-crafted		
Freecell[60]	0	-	-	0	-	-	5	228	17834	60	-	-
Blocks[35]	0	-	-	12	50.84	6877	18	32.5	7856	35	0.3	0
Rovers[20]	20	1.7	16	20	2.0	16	6	219	32787	20	1	1
Logistics[80]	80	8.3	75	80	29.2	75	10	1.58	432	80	-	-
Driverlog[20]	7	74.5	5080	8	60.1	4913	7	40.2	3421	20	-	-
Zenotravel[20]	4	1527.8	194477	6	1453.8	161365	8	162	5994	20	0.2	0
Miconic[150]	150	0.66	0	150	0.75	0	55	509	75372	150	0.0	0
Satellite[20]	18	0.59	1.2	20	1	0.7	6	702	22982	20	-	-
Depots[22]	4	59.73	4655	15	1178.8	50404	4	53.5	6034	22	-	-

Table 1: Results in the IPC-2000 and IPC-2002 domains, with the number of instances of each domain shown in brackets.

We performed experiments with two versions of our algorithm. The base algorithm that achieves the preconditions and goals in any order was slow in testing, so we activated precondition ordering in both versions. The first version, HTNPrecon, achieves the goals in the order they appear in the PDDL definition. The second version, HTNGoal, implements our goal ordering strategy in addition to precondition ordering.

Table 1 shows the results for the 9 domains. For each planner we report the number of instances solved and the maximum time taken to solve an instance. We also report the maximum number of backtracks (in thousands). For hand-crafted domains which could not be solved by the JSHOP planner we provide only coverage. As expected, our goal ordering strategy mainly improves the performance of the algorithm in Blocks and Depots, the two domains that are most sensitive to goal ordering. In addition, goal ordering enables us to solve two additional instances in Satellite. The hand-crafted HTNs successfully solve all instances of all domains with little backtracking; however, our algorithm generates HTNs in a fraction of a second, while the hand-crafted HTNs were carefully designed by human experts.

8 Conclusion

In this paper we have presented what we believe to be the first domain-independent algorithm for generating HTNs. All the algorithm needs is a PDDL description of the planning domain and a single representative instance. In four domains, the algorithm successfully generates HTNs that can be used to efficiently solve any instance, thus being competitive with HTNs designed by human experts and heuristic search algorithms. Although the success of the algorithm is limited in the remaining domains, we believe that there are still many potential benefits. In many cases, even though the resulting HTN is not constrained enough, subtasks identified by the algorithm may still be useful. This is the case, for example, in Blocks, where the resulting HTN contains tasks and methods for putting a block on top of another block. In such cases, the algorithm can help suggest initial decompositions that can later be refined by a human expert.

The avenue for future research that we find most promising is to test different restrictions on the invariant graphs. If the representative instance can still be solved under some restriction, the resulting HTN may still be able to solve other instances, and the restriction has the effect of reducing the branching factor. In essence, this mechanism would reduce the number of ways to traverse the invariant graphs.

Another option is to translate the resulting HTNs back to classical planning instead of using an HTN solver [1, 5]. In this way we can take advantage of the reduced branching factor offered by the HTNs, and use heuristic search planners to solve the resulting classical planning instances.

References

[1] R. Alford, U. Kuter, and D. Nau. Translating HTNs to PDDL: A Small Amount of Domain Knowledge Can Go a Long Way. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 1629–1634, 2009.

[2] T. Geier and P. Bercher. On the Decidability of HTN Planning with Task Insertion. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI'11)*, pages 1955–1961, 2011.

[3] M. Helmert. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence*, 173:503–535, 2009.

[4] C. Hogg, H. Munoz-Avila, and U. Kuter. HTN-MAKER: Learning HTNs with Minimal Additional Knowledge Engineering Required. In *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI'08)*, pages 950–956, 2008.

[5] M. Lekavý and P. Návrat. Expressivity of STRIPS-Like and HTN-Like Planning. In *Proceedings of the 1st KES Symposium on Agent and Multi-Agent Systems - Technologies and Applications (KES-AMSTA'07)*, pages 121–130, 2007.

[6] D. Nau, T. Au, O. Ilghami, U. Kuter, W. Murdock, D. Wu, and F. Yaman. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research*, 20:379–404, 2003.

[7] H. Zhuo, D. Hu, C. Hogg, Q. Yang, and H. Munoz-Avila. Learning HTN Method Preconditions and Action Models from Partial Observations. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 1804–1809, 2009.