Learning Hierarchical Problem Networks for Knowledge-Based Planning*

Pat Langley[1,2]

[1] Institute for the Study of Learning and Expertise, Palo Alto, California 94306 USA
[2] Center for Design Research, Stanford University, Stanford, CA 94305 USA
patrick.w.langley@gmail.com
http://www.isle.org/∼langley/

Abstract. In this paper, we review hierarchical problem networks, which encode knowledge about how to decompose planning tasks, and report an approach to learning this expertise from sample solutions. In this framework, procedural knowledge comprises a set of conditional methods that decompose problems – sets of goals – into subproblems. Problem solving involves search through a space of hierarchical plans that achieve top-level goals. Acquisition involves creation of new methods, including state conditions for when they are relevant and goal conditions for when to avoid them. We describe HPNL, a system that learns new methods by analyzing sample hierarchical plans, using violated constraints to identify state conditions and ordering conflicts to determine goal conditions. Experiments with on-line learning in three planning domains demonstrate that HPNL acquires expertise that reduces search on novel problems and examine the importance of learning goal conditions. In closing, we contrast the approach with earlier methods for acquiring search-control knowledge, including explanation-based learning and inductive logic programming. We also discuss limitations and plans for future research.

Keywords: Learning for problem solving · Hierarchical planning · Search-control knowledge · Learning decomposition rules

1 Introduction

Three elements underlie classic accounts of intelligence in humans and machines: reasoning, search, and knowledge. Early work on AI saw rapid progress on the first two areas, but major applications were delayed until the advent of expert systems, which focused on the third topic. These saw widespread use but they were expensive to construct and maintain. Recognition of these drawbacks fostered increased research in machine learning, which aimed to automate the acquisition of such expertise. This movement saw rapid advances in the 1980s, which in turn led to early fielded applications (Langley & Simon, 1995). These

* ILP 2022, 31st International Conference on Inductive Logic Programming, Cumberland Lodge, Windsor, UK

grew more common with the automated collection of large data sets and delivery channels made possible by the World Wide Web.

However, the vast majority of learning applications have focused on classification and regression. For settings that require sequential decision making, such as plan generation, progress has been much slower. Recent successes of reinforcement learning have been limited to domains with accurate simulators that can carry out the thousands or millions of trials required. Instead, we desire mechanisms that acquire procedural expertise as efficiently as humans through reasoning about, and learning from, solutions to individual problems.

In this paper, we present a new approach to rapid learning of expertise for solving planning tasks. We start by reviewing hierarchical problem networks, a recently proposed formalism for encoding plan knowledge, and their use in solving complex problems through decomposition. After this, we describe an approach to acquiring such structures from sample hierarchical plans. Next we report the results of experiments that demonstrate its effectiveness in three planning domains. We conclude by reviewing earlier research on learning hierarchical knowledge for problem solving and outlining directions for future work in this important but understudied area.

2 A Review of Hierarchical Problem Networks

Breakthroughs in machine learning often follow the introduction of new representations for expertise and new mechanisms for using this content. For instance, inductive logic programming was not possible until formalisms like Prolog had been developed. In this section, we review a recent notation for knowledge about planning – *hierarchical problem networks* – and how this content supports efficient problem solving in sequential domains. The aim is akin to early work on search-control rules and more recent efforts on hierarchical task and goal networks, but we will see that the older frameworks differ in important ways.

2.1 Representing Hierarchical Problem Networks

A recurring theme in both human and machine cognition is decomposition of complex problems into simpler ones. This idea is central to logic programming (Lloyd, 1984), which organizes knowledge into hierarchical structures, with nonterminal symbols serving as connective tissue. This notion is also key to hierarchical task networks (Nau et al., 2003), which encode knowledge about how to decompose sequential activities into subactivities, using task names to link across levels. In both cases, nonterminal symbols support modularity and reuse, much like subroutine names in traditional programming languages. But reliance on nonterminals is also a drawback in that learning requires creation of intermediate predicates, which is more challenging than identifying rule conditions.

Hierarchical problem networks (HPNs) offer a way to encode planning knowledge that sidesteps this complication. Like other formalisms, they divide content into rules or *methods*, each indicating how to decompose a class of problems into simpler ones. More specifically, each HPN method includes a head that describes a goal to be achieved and an operator that accomplishes it. A method also con-

Table 1. Four methods for logistics planning that include a head, state conditions, an operator, a subproblem, and optional goal conditions. These partially encode an HPN procedure that solves problems in the logistics domain efficiently. The notation assumes that distinct variables will match against different constant expressions. Bold and italic fonts for some conditions denote sources of learning discussed later.

`((at ?o1 ?l3))`	
`conditions:`	`((object ?o1) (truck ?t1) (location ?l3) (location ?l2)` *`(location ?l1) (in-city ?l3 ?c1) (in-city ?l2 ?c1) (in-city ?l1 ?c1)`* **`(at ?t1 ?l2) (at ?o1 ?l1))`**
`operator:`	`(unload-truck ?o1 ?t1 ?l3)`
`subproblem:`	`((at ?t1 ?l3) (in ?o1 ?t1))`
`((at ?t1 ?l1))`	
`conditions:`	`((truck ?t1) (location ?l2) (location ?l1) (city ?c1) (in-city ?l2 ?c1) (in-city ?l1 ?c1)` **`(at ?t1 ?l2))`**
`operator:`	`(drive-truck ?t1 ?l2 ?l1 ?c1)`
`subproblem:`	`((at ?t1 ?l2))`
`unless-goals`	`((in ?o ?t1))`
`((in ?o1 ?t1))`	
`conditions:`	`((object ?o1) (truck ?t1) (location ?l1)` *`(location ?l2) (in-city ?l1 ?c1) (in-city ?l2 ?c1)`* **`(at ?t1 ?l2) (at ?o1 ?l1))`**
`operator:`	`(load-truck ?o1 ?t1 ?l1)`
`subproblem:`	`((at ?t1 ?l1) (at ?o1 ?l1))`
`((in ?o1 ?t1)`	
`:conditions`	`((object ?o1) (truck ?t1) (location ?l1)` *`(airport ?l1) (location ?l2) (location ?l3) (in-city ?l1 ?c1) (in-city ?l2 ?c1) (in-city ?l3 ?c2)`* **`(at ?t1 ?l2) (at ?o1 ?l3))`**
`:operator`	`(load-truck ?o1 ?t1 ?l1)`
`:subproblem`	`((at ?t1 ?l1) (at ?o1 ?l1))`

tains a subproblem based on the operator's conditions that should be solved before one applies it. In addition, each method includes *state* conditions specifying relations that must hold for the decomposition to be appropriate. Teleoreactive logic programs (Langley & Choi, 2006) and hierarchical goal networks (Shivashankar et al., 2012; Fine-Morris et al., 2020) also index methods by goals they achieve, but neither defines subproblems in terms of operator conditions.

Table 1 shows four methods from a hierarchical problem network for logistics planning, a domain that has been widely used in the literature. The first method indicates that, to achieve *(at ?o1 ?l3)*, we apply *(unload-truck ?o1 ?t1 ?l3)*, but only if we first satisfy this operator's two conditions *(at ?t1 ?l3)* and *(in ?o1 ?t1)*. Moreover, this decomposition is only relevant when the truck *?t1* is located in the same city as the object *?o1* that we want to transport. The second method specifies how to achieve *(at ?t1 ?l1)* by applying *(drive-truck ?t1 ?l2 ?l1 ?c1)*, but again only when its current and desired location are in the same city. The last two methods detail where to load an object into a truck, with one for situations when they are in the same city and another when they differ.

Table 2. Three operators for the logistics planning domain, each specifying an action (head), conditions, and effects. The domain also includes operators (not shown) for unloading an airplane, flying an airplane between cities, and loading an airplane.

```
((unload-truck ?o ?t ?l)
 :conditions ((object ?o)(truck ?t)(location ?l)(at ?t ?l)(in ?o ?t))
 :effects    ((at ?o ?l)(not (in ?o ?t))))
((drive-truck ?t ?l1 ?l2 ?c)
 :conditions ((truck ?t)(location ?l1)(location ?l2)(city ?c)
              (in-city ?l1 ?c)(in-city ?l2 ?c)(at ?t ?l1))
 :effects    ((at ?t ?l2)(not (at ?t ?l1))))
((load-truck ?o ?t ?l)
 :conditions ((object ?o)(truck ?t)(location ?l)(at ?t ?l)(at ?o ?l))
 :effects    ((in ?o ?t)(not (at ?o ?l))))
```

However, HPN methods can also include an optional field that specifies when we should **not** invoke them. This refers not to the current state but rather to other goals in the current problem that are unsatisfied in the state. The second method in Table 1 includes such an *unless-goals* condition, which indicates that we should not attempt to achieve *(at ?t1 ?l1)* by driving truck *?t1* if *(in ?o ?t1)* is an open goal. The reasoning is straightforward: any package that we want in the truck should already be in the vehicle before we move it. Such constraints are similar in spirit to rejection rules in problem solvers like PRODIGY (Minton, 1988), but they are embedded in methods rather than stored separately.

Naturally, an HPN knowledge base also requires definitions for the domain operators that appear in methods. As in other AI planning frameworks, these give the name and arguments for an action, its effects in terms of added and deleted relations, and the conditions under which these effects occur. Table 2 presents operator descriptions for three actions from the logistics domain: unloading a truck, driving a truck, and loading a truck. The domain knowledge also includes three analogous actions for unloading an airplane at an airport, flying it from one airport to another, and loading it at an airport. Instances of these operators serve as terminal nodes in hierarchical plans, whereas instances of HPN methods correspond to nonterminal nodes.

2.2 Hierarchical Problem Decomposition

The HPN framework assumes the presence of a problem solver that can use available methods to generate plans. We can specify this performance task more explicitly in terms of inputs and outputs:

- *Given*: A set of HPN *methods* and associated domain *operators*;
- *Given*: An *initial state* specified as a conjunctive set of *relations*;
- *Given*: A *problem* specified as a conjunctive set of *goals*;
- *Find*: A *hierarchical plan* that transforms the initial state into one that satisfies all problem goals.

Two supporting structures, the *problem stack* and the *state sequence*, are altered during the problem-solving process. The first is simply a stack of problems and their ancestors, whereas the latter is a list of states in reverse order of generation.

Plan generation recursively decomposes the top-level problem into subproblems. The decomposition mechanism relies on three main subprocesses: method matching, method selection, and method expansion. Problem solving iteratively examines the topmost element on the problem stack and, if necessary, uses HPN methods to place new subproblems above it. These are popped when satisfied, which in turn allows application of the associated operator. This produces a new state that may lead to new subproblems. Before adding a subproblem or applying an operator, the procedure checks the stack for problem loops (with identical goals) and state loops (with identical states), rejecting a candidate if either occurs. The overall process is similar to that used with hierarchical task networks and goal networks. The key difference is that it relies on a stack of *problems* rather than tasks or individual goals, which supports checking of *unless* conditions that avoid goal interactions. Langley and Shrobe (2021) describe HPD, a hierarchical planner that implements these processing postulates.

In summary, planning in this framework involves search through a space of decompositions defined by methods, operators, problem goals, and an initial state. The need for backtracking can arise when the HPN's methods have overly general state conditions or when they lack necessary *unless-goals* conditions. However, given appropriate knowledge about a domain, the HPD problem solver mimics a deterministic procedure in that it selects a useful decomposition at each step. Langley and Shrobe reported this behavior experimentally in three planning domains, as well as the search that results when state and goal conditions are omitted from the knowledge base.

3 Learning HPNs from Sample Solutions

Now that we have described hierarchical problem networks as a representational framework and their use in efficient generation of hierarchical plans, we can turn to the acquisition of this knowledge. As with the performance task, we specify the learning problem in terms of inputs and outputs:

- *Given*: A set of domain operators that specify conditional effects of actions;
- *Given*: A set of training tasks comprising initial states and conjunctive goals;
- *Given*: A hierarchical plan for each task that achieves its associated goals;
- *Find*: A hierarchical problem network that solves the training tasks efficiently and that generalizes well to new cases.

This statement does not specify precisely the two measures of success, 'efficiently' or 'generalizes well'. Naturally, we desire learned knowledge that solves planning tasks with little or no search, and we want improvement on this front to occur rapidly, but these are empirical issues, not definitional ones.

In this section, we present an approach to addressing the problem of acquiring hierarchical problem networks from sample solutions. First we describe the inputs to the learning process more clearly and introduce an example that we

draw upon later. After this, we explain how the structure of training problems maps onto new HPN methods and how the approach extracts state and goal conditions for each method. We also report an implementation of these ideas in the HPNL system, which builds on the existing HPD problem solver.

3.1 Inputs to HPN Learning

As we have seen, one input to the learning process is a set of domain operators, each of which describes the conditional effects of some action. Table 2 presented three such operators from the logistics domain for unloading an object from a truck, driving a truck from one location to another, and loading a package into a truck. In addition, each operator-effect pair maps onto a naive HPN method that indicates how one can achieve the effect by first achieving the operator's conditions and then applying it. Thus, we can generate a naive hierarchical problem network from the operators alone, but it can still require substantial search to solve problems. To move beyond this incomplete knowledge, we need experience with specific training problems (i.e., initial states with goal descriptions). In addition, we need hierarchical plans that decompose the top-level problems into subproblems to make contact with their starting situations. Each component of these plans links some goal to a conjunction of subgoals through an operator instance, with terminal literals corresponding to elements of the initial state.

Figure 1 depicts a hierarchical plan for a simple logistics problem with the single goal *(at o1 l3)* and an initial state containing *(at o1 l1)*, *(at t1 l2)*, and static relations like *(location l1)* and *(in-city l1 c1)*. The plan indicates that, to achieve *(at o1 l2)*, we can apply the operator instance *(unload-truck o1 t1 l2)*, which in turn requires solving subgoals *(in o1 t1)* and *(at t1 l3)*. Furthermore, to achieve *(in o1 t1)*, we can apply *(load-truck o1 t1 l1)*, which means addressing the subgoals *(at t1 l1)* and *(at o1 l1)*. The latter holds in the initial state and *(at t1 l1)* can be satisfied by invoking *(drive-truck t1 l2 l1 c1)*, provided that *(at t1 l2)*, which is true at the outset. In addition, we can achieve *(at t1 l3)* by applying *(drive-truck t1 l1 l3 c1)*. The plan does not mention other locations, trucks, packages, or relations that are not relevant to the solution. We might instead require the learner to find its own hierarchical plan for each training problem, as in early work on learning for search control (Sleeman et al., 1982). However, the learner would still need to transform the resulting hierarchical plans into a set of HPN methods, so we have chosen to focus on this central issue rather than on such preprocessing steps.

3.2 Identifying HPN Structure

Classic research on learning for hierarchical decomposition has addressed three questions. The first concerns how to break a sample solution into segments in order to identify the structure of new methods. The second issue deals with when to associate the same head or task name with distinct methods. The final matter involves what conditions to place on each method. Different approaches to learning hierarchical procedures address these design choices in different ways, but the HPN framework offers straightforward answers to the first two questions.

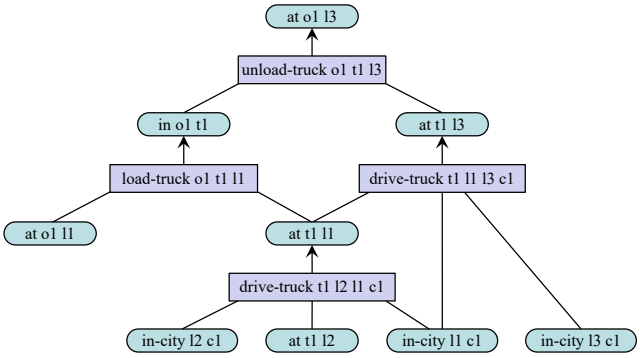

Fig. 1. A hierarchical plan for a simple logistics task that involves four problem decompositions and associated operator instances.

Recall that sample problem solutions, whether provided by a human or generated through search, specify a hierarchical plan. Each literal in this structure (except for terminal nodes) is associated with an operator instance that achieves it and the instantiated set of conditions for that operator. Each triple of this sort maps onto a candidate HPN method. For instance, the topmost goal *(at o1 l3)* in Figure 1, the operator *(unload-truck o1 t1 l3)* that achieves it, and the latter's conditions together map onto the first method in Table 1. The operator's static conditions, such as *(object o1)*, contribute to the method's state conditions, although we will see soon that others may augment them. Dynamic conditions, such as *(at t1 l3)* and *(in o1 t1)*, become elements of the *:subproblem* field. Constant arguments of predicates are replaced with variables in a consistent way throughout the method. The four operator instances in Figure 1 correspond to four distinct methods that such a process might add to the knowledge base.

The second issue, when to assign a common head to different methods, also has a simple response in the HPN framework. Because each method refers to a goal that it achieves, every structure that addresses the same goal will have the same head. For instance, three methods constructed from the hierarchical plan in Figure 1 would both have heads with the predicate *at*, even though one involves the *unload* operator and the others invoke the *drive* operator. This approach is very different from techniques for learning classical hierarchical task networks, whose heads include task names that are distinct from state predicates. However, it is similar to early mechanisms for learning teleoreactive logic programs (Langley & Choi, 2006; Nejati et al., 2006).

3.3 Inferring State Conditions

The organization of each sample solution determines the structure of the learned HPN, but we must still identify the conditions under which to apply each rule. Let us start with state-related conditions. One option would use *explanation-based learning* to analyze logical dependencies and identify conditions from each

sample solution, which would seem to support rapid learning. For example, for the top-level decomposition in Figure 1, it would collect all terminal nodes in the hierarchical plan as conditions for a new method. However, empirical studies revealed that such analytic techniques can create complex, specific rules that are expensive to match and generalize poorly. This leads to a *utility problem*, where learning reduces search but increases planning time (Minton, 1988). Another response would use *inductive logic programming* or similar schemes for relational learning. These often carry out general-to-specific search for conditions that find simpler, more general rules, but typically require multiple examples to discriminate positive from negative cases, which reduces the learning rate. We desire a way to identify state conditions on HPN methods that combines learning from individual training solutions with an inductive bias toward generality.

Remember that a key purpose of state conditions in the HPN framework is to ensure that all arguments of a method's operator are bound. Some of these variables are already specified in the head, but others may remain unconstrained. For instance, the third method in Table 1 says that we can achieve *(in ?o1 ?t1)* by applying *(load-truck ?o1 ?t1 ?l1)*, so the object and truck appear in the head, but not the location. Information about argument types (e.g., that *?l1* is a location) do not suffice to eliminate ambiguity. Fortunately, we can combine knowledge implicit in operators with details about the state in which a decomposition occurs to infer additional conditions. Figure 1 reveals that *(in o1 t1)* is achieved by *(load-truck o1 t1 l1)*, which has *(at t1 l1)* as one of its conditions. The latter relation is achieved in turn by *(drive-truck t1 l2 l1 c1)*, which deletes *(at t1 l2)* and adds *(at t1 l1)*, so the two literals are mutually exclusive, at least in this context. This suggests *(at t1 l2)*, appropriately generalized, as a state condition for the third method, as denoted by bold font in the table.

We can use operator specifications to identify constraints for this purpose: any relation that an operator deletes and replaces with another implies a mutual exclusion. Examples include *(load-truck ?o ?t ?l)*, which removes the literal *(at ?o ?l)* with *(in ?o ?t)*, and *(unload-truck ?o ?t ?l)*, which does the opposite. Three such constraints arise in the logistics domain: an entity cannot be *at* two places, it cannot be *in* two vehicles, and it cannot be *at* a place and *in* a vehicle. To find state conditions for a method M with operator O, for each argument of O not bound in M's head, we can note which relations held when the sample decomposition occurred but are inconsistent with O's conditions.

Unfortunately, the resulting conditions may not suffice and static literals may be necessary to ensure proper argument bindings. For instance, in Logistics the conditions *(in-city ?l2 ?c1)* and *(in-city ?l1 ?c1)* in Table 1's third method indicate that *?l2*, the truck's location, and *?l1*, the package's location, must be in the same city. The fourth method specifies a different configuration in which we should load the truck at an airport when the package is in a different city. We can identify such static conditions by chaining outward from unbound arguments in a sample solution. For the state in which a decomposition occurs, we find literals that contain such an argument, note any new arguments that are introduced, and find relations that contain them. This process continues until no new arguments

arise, at which point we replace constants with variables in a consistent manner. Table 1 uses italics to highlight conditions found through this mechanism. The second method, for achieving *(at ?t ?l1)* with *drive-truck*, does not include any because it inherits all necessary static conditions from its operator.

3.4 Identifying Goal Conditions

Now let us turn to the construction of *unless-goals* conditions. These serve to constrain the order in which methods are selected during hierarchical decomposition. The only example of an unless condition in Table 1 occurs in the second method, which indicates that one should not attempt to achieve *(at ?t1 ?l1)* with *(drive-truck ?t1 ?l2 ?l1 ?c1)* if there is an unsatisfied goal of the form *(in ?o ?t1)*. To identify such priorities, we examine the sample solution to find which state literals held when each goal decomposition took place. We then note which operator instance achieved each goal and calculate its composed conditions and effects, much as in techniques for forming macro-operators (e.g., Iba, 1989). When two goals appear as siblings in the sample hierarchical plan, we see whether the composed effects of the subplan carried out later would 'clobber' the composed conditions of the one carried out earlier. If so, then we add an *unless-goals* condition that requires the method learned from the later subplan **not** be selected when the goal achieved by the earlier subplan still remains unsatisfied.

For example, the goals *(in o1 t1)* and *(at t1 l3)* in Figure 1 each have associated subplans that achieve them. The composed conditions of the first subplan include the state literal *(at t1 l1)*, which is needed for loading *o1* into *t1*. However, the composed effects of the second subplan includes deletion of *(at t1 l1)* because this is a necessary result of driving *t1* from *l1* to *l3*. Thus, carrying out the two subplans in reverse order will not succeed and, to avoid this negative interaction, we add the *unless-goals* condition *(in ?o ?t1)* to the second method in Table 1. A different sample solution that involves flying an airplane from one city to another would lead to a similar goal condition, although a method for this goal-operator combination does not appear in the table.

3.5 Implementation Details

We have implemented this approach to learning hierarchical problem networks in Steel Bank Common Lisp. The resulting system – HPNL – inputs domain operators, training problems with associated hierarchical plans, and constraints about which relations are mutually exclusive. The program processes sample solutions one at a time, adding new methods as suggested by these traces. Before the system creates a method, it compares the candidate with existing structures to see if they are structurally isomorphic by checking whether their heads, state conditions, and operators map onto each other. In such cases, it increments a counter rather than adding a duplicate and the HPD problem solver uses these scores to select among multiple matches, favoring methods with higher counts.

HPNL acquires hierarchical methods in an incremental, monotonic manner that is unaffected by the order of training problems. In practice, it constructs only a few methods from each solution trace, as subplans often have similar struc-

tures, and in some cases none are created at all. For classic planning domains, handcrafted programs can require only one or two methods per operator-effect pair. Nevertheless, HPNL treats each hierarchical plan as a potential learning experience, as new configurations of initial states and goals are always possible. When the implementation detects a goal interaction, it adds an analogous *unless-goals* condition to each method with the same goal-operator pair.

4 Empirical Evaluation

Our approach to learning hierarchical problem networks holds promise for rapid acquisition of planning expertise, but we must still demonstrate its effectiveness. To this end, we designed and ran experiments that studied the learning behavior of the HPNL implementation. For a given domain, we manually created a set of methods that solved problems with little or no search. We also generated a set of state-goal configurations and used the expert knowledge to generate a hierarchical plan for each such problem. We primed HPNL with content about domain operators and constraints,[3] then presented sample problems and plans sequentially, using each one first as a test case and then as a training case. We were concerned with two measures of performance – number of decompositions and CPU time – which we recorded on each run. We were also interested in rate of improvement, so we collected learning curves that plotted performance against number of training samples, averaged across different training orders.

We evaluated HPNL's learning behavior on three different problem-solving domains to provide evidence of generality. These settings will be very familiar to readers of the AI planning literature:

- **Blocks World** involves changing an initial configuration of blocks into a target configuration. States were encoded with six dynamic predicates affected by four operators – *stack*, *unstack*, *putdown*, and *pickup*. The expert HPN program for this domain included six methods, two for *holding* and one each for other dynamic predicates. Problems involved four to six blocks and ranged from four to six goals, while solution lengths were six to 16 steps.
- **Logistics** requires transporting packages from initial to target locations. States involved seven static relations but only two dynamic predicates – *at* and *in* – that were altered by six operators. The expert knowledge base included seven methods, four for achieving the *at* relation and three for the *in* predicate. Tasks included two packages and two goals in one or two cities, each with three locations, with solutions varying from four to 19 steps.
- **Depots** involves moving crates from some pallets to others and stacking them in specified towers. There were six static predicates and six dynamic relations – *available*, *lifting*, *at*, *on*, *in*, and *clear* – influenced by five operators. Expert knowledge comprised eight methods, two for *on*, two for *lifting*, and one each for other dynamic predicates. Problems contained two or three depots, three to four crates, and three to four goals, while solutions were seven to 21 steps.

[3] We constructed these constraints manually, although in principle they could have been extracted automatically from the operators' definitions.

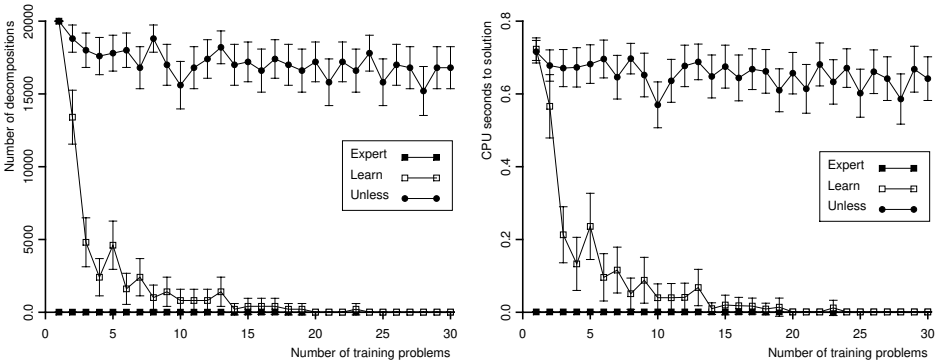

Fig. 2. Number of decompositions (left) and CPU time (right) needed to solve problems in the Blocks World by an expert HPN knowledge base, HPN methods learned from sample plans, and learned methods that lack *unless-goals* conditions. Each curve reports mean values with 95 percent confidence intervals over 100 random problem orders.

For each domain, we trained and tested the system on 30 distinct problems, limiting effort on each task to 20,000 method decompositions (i.e., nodes visited in the search tree). We also limited plan length to 30 steps in the Blocks World and 50 steps in the other domains. For each run, we recorded both CPU seconds and the decompositions carried out, assigning the maximum number if the system failed to find a solution. To factor out order effects, we called the system 100 times with random problem sequences and averaged performance across runs.

For this initial study, we wanted to understand HPNL's rate of learning, how acquired expertise compares to handcrafted methods, and how important goal conditions are to success. For these reasons, we examined three experimental conditions – handcrafted structures, learned methods, and the same methods without *unless-goals* conditions. We did not compare the HPNL's behavior to other systems for acquiring plan knowledge because, in most cases, their inputs would not be comparable. Moreover, our aim was not to show which approach is superior but to demonstrate the viability of our new framework.

Figure 2 presents results for the Blocks World. The left graph plots the number of decompositions required to solve each task, whereas the right displays the CPU times. As expected, the manually created HPN for this domain solved each problem with little or no search and took negligible processing time. More important, the HPNL system, which started with overly general methods, initially needed substantial search to find solutions, but it acquired expertise rapidly and its performance on both measures reached near expert level by the 20th task.

The figure also shows HPNL's behavior when we removed the ability to test *unless-goals* conditions on learned methods, and thus the ability to master goal orderings. The lesioned system's performance improved slightly with training but continued to require massive search, which indicates that goal conditions are crucial for expert behavior in this domain. In addition, note that the learning

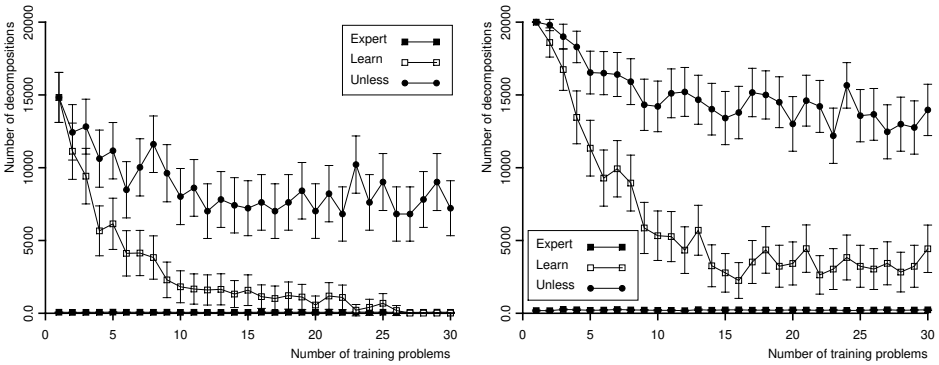

Fig. 3. Number of decompositions needed to solve problems in Logistics (left) and Depots (right) by an expert HPN knowledge base, HPN methods learned from sample plans, and learned methods that lack *unless-goals* conditions. Each curve reports means and 95 percent confidence intervals over 100 random problem orders.

curves for the number of decompositions and time have very similar shapes. This suggests that HPNL does not suffer from the classic utility problem, in which learning reduces search but actually slows processing. This phenomenon arose in recursive domains like the Blocks World, so its absence here is encouraging.

The leftmost graph in Figure 3 shows results for the Logistics domain. As before, the handcrafted knowledge base solves all problems very rapidly with almost no search. In this case, learning reduces the number of decompositions more slowly, only reaching expert level by the 26th training problem. Again, the asymptote is substantially higher when the system does not use *unless-goals* conditions, although it fares better than in the Blocks World. Although not shown, the learning curve for CPU time again parallels the one for number of decompositions. Results for the Depots domain, in the rightmost graph of Figure 3, are less impressive in that HPNL never quite reaches the expert level and some search through the space of hierarchical plans remains. This suggests that the constraint-based learning mechanism is finding overly specific conditions that do not generalize to new problems as much as desired.

Examination of structures acquired in the three domains is consistent with this interpretation. For the Blocks World, HPNL created only six methods and these were nearly identical to the six handcrafted ones. In contrast, the system learned 29 methods for Logistics and 30 for Depots, substantially more than the 11 and 10 rules in their expert knowledge bases. Further inspection revealed that many methods were idiosyncratic variants of the handcrafted ones. For instance, the system learned 11 different Logistics methods for achieving the *at* relation with *unload-truck*: when the target and truck locations are the same, when the truck is at the airport, and others. We posit two reasons for this behavior: direct replacement of constants with variables introduces identities that are not logically required and the technique for adding static relations finds conditions

not needed to constrain variables. Neither issue arose in the Blocks World, which is why the system found fewer, more general methods in that domain.

In summary, our experimental findings were mixed. In two domains, HPNL achieved expert-level performance from under 30 sample plans, but learning did not eliminate search in the third domain and specialized conditions reduced its generalization. Another issue was that the learned methods sometimes found much longer solutions than did handcrafted ones. Thus, our approach to learning expertise for hierarchical planning achieved some but not all of our original aims.

5 Related Research

Our approach to learning expertise for problem solving incorporates ideas from the previous literature, but it also makes novel contributions. Langley and Shrobe (2021) have reviewed the HPN framework's relation to other hierarchical formalisms, so we will focus here on the acquisition of expertise. Some early work on learning search-control knowledge (Sleeman et al., 1982) emphasized induction of state-related conditions on when to apply operators, whereas later efforts on learning search-control rules (e.g., Minton, 1988) and macro-operators (Iba, 1989) invoked explanation-based techniques. Most relied on forward search from states, but some systems chained backward from goals, as in our framework.

A few early researchers addressed the acquisition of hierarchical structures for planning and problem solving. Ruby and Kibler's (1991) SteppingStone and Marsella and Schmidt's (1993) PRL acquired decomposition rules, but they did not associate operators with them. Shavlik (1990) reported an approach that learned hierarchical rules, some recursive, using analytic techniques, but it worked within the situation calculus. The closest relative to our work from this era was Reddy and Tadepalli's (1997) X-Learn, which also learned rules for decomposing goals but used inductive logic programming to identify conditions.

There has also been research on learning hierarchical task networks and related structures. For instance, Ilghami et al. (2002) used successful HTN plans as training cases, invoking a version-space technique to induce conditions on methods. Hogg et al. (2008) instead used a form of explanation-based learning to derive conditions on methods from definitions of operators used in sample traces. In this arena, the nearest kin is Nejati et al.'s (2006) work on learning goal-driven teleoreactive logic programs from sample plans, which also acquired methods from individual problems but used simpler analysis than other efforts.

Schmid and Kitzelmann (2011) present a very different approach that learns recursive programs by detecting and generalizing patterns in a small set of sample traces. Their formulation differs from ours but shares the aim of rapidly acquiring search-free procedures. Finally, Cropper and Muggleton (2015) report a system that uses meta-interpretive learning, which combines abductive reasoning with abstract knowledge, to create hierarchical rules for sequential behavior with invented nonterminal symbols. In contrast, our approach to learning hierarchical problem networks avoids the need for such predicates because it indexes acquired methods by goals they achieve. In summary, our framework shares features with earlier research on procedural learning but also has important differences.

6 Concluding Remarks

In this paper, we reviewed hierarchical problem networks and presented a novel approach to learning them from sample plans. A primary innovation was the use of domain constraints to identify simple conditions on hierarchical methods from single subplans, which avoids both the overly complex rules generated by explanation-based techniques and the reliance of relational induction on multiple training cases. Another important contribution was a means for learning *unless-goals* conditions on methods that ensure goals are tackled in the proper order. Experiments in three domains provided evidence that this approach leads to effective planning knowledge which substantially reduces both search and problem-solving time on novel problems from the same domain.

Despite the progress that we have reported here, our mechanisms for acquiring plan knowledge require further development to improve their rate of learning. One promising approach involves abandoning simple substitution of constants with variables, which encodes accidental identities, and instead propagates dependencies through sample plans. This should reduce the number of idiosyncratic methods and improve generalization to new problems. Another response is to replace the chaining technique that finds static relations with a specialized form of inductive logic programming. Recall that state conditions serve to distinguish desirable bindings for operator arguments that will achieve the method's goal from ones that will not. Each sample decomposition offers one positive instance of such bindings and typically provides multiple negative cases that can support induction of conditions about both dynamic and static relations.

We intend to implement both techniques in future research and run experiments that identify the conditions under which they are effective. We should also compare our approach to classic methods for analytic learning, including macro-operator formation, and to inductive logic programming. Finally, we should examine learning mechanisms' behaviors on additional planning domains to further demonstrate their generality and to uncover any unexpected drawbacks. Nevertheless, our initial results have been encouraging and suggest that our framework for acquiring expertise in hierarchical planning merits additional study.

Acknowledgements This research was supported by Grant N00014-20-1-2643 from the US Office of Naval Research, which is not responsible for its contents. We thank Howie Shrobe, Boris Katz, Gary Borchardt, Sue Felshin, Mohan Sridharan, and Ed Katz for discussions that influenced the ideas reported here.

References

Cropper, A., Muggleton, S. H.: Learning efficient logical robot strategies involving composable objects. In: *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence*, pp. 3423–3429. AAAI Press: Buenos Aires, Argentina (2015)

Fine-Morris, M., et al.: Learning hierarchical task networks with landmarks and numeric fluents by combining symbolic and numeric regression. In: *Proceedings of the Eighth Annual Conference on Advances in Cognitive Systems* (2020)

Hogg, C., Muñoz-Avila, H., Aha, D. W.: HTN-Maker: Learning HTNs with minimal additional knowledge engineering required. In: *Proceedings of the Twenty-Third National Conference on Artificial Intelligence*. AAAI Press (2008)

Iba, G. A.: A heuristic approach to the discovery of macro-operators. *Machine Learning* **3**, 285–317 (1989)

Ilghami, O., Nau, D. S., Muñoz-Avila, H., Aha, D. W.: CaMeL: Learning method preconditions for HTN planning. In: *Proceedings of the Sixth International Conference on AI Planning and Scheduling*, pp. 131–141. AAAI Press: Toulouse, France (2002)

Langley, P., Choi, D.: Learning recursive control programs from problem solving. *Journal of Machine Learning Research* **7**, 493–518 (2006)

Langley, P., Shrobe, H. E.: Hierarchical problem networks for knowledge-based planning. In: *Proceedings of the Ninth Annual Conference on Advances in Cognitive Systems*. Cognitive Systems Foundation (2021)

Langley, P., Simon, H. A.: Applications of machine learning and rule induction. *Communications of the ACM*, **38**, November, 55–64 (1995)

Lloyd, J. W.: *Foundations of logic programming*. Springer-Verlag: Berlin, Germany (1984)

Marsella, S. C., Schmidt, C. F.: A method for biasing the learning of nonterminal reduction rules. In: Minton, S. (ed.), *Machine learning methods for planning*. Morgan Kaufmann: San Francisco, CA (1993)

Minton, S.: Quantitative results concerning the utility of explanation-based learning. *Proceedings of the Seventh National Conference on Artificial Intelligence*, pp. 564–569. Morgan Kaufmann: St. Paul, MN (1988)

Nau, D., Au, T-C., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D., Yaman, F.: SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research* **20**, 379–404 (2003)

Nejati, N., Langley, P., Könik, T.: Learning hierarchical task networks by observation. In: *Proceedings of the Twenty-Third International Conference on Machine Learning*, pp. 665–672. Pittsburgh, PA (2006)

Reddy, C., Tadepalli, P.: Learning goal-decomposition rules using exercises. In: *Proceedings of the Fourteenth International Conference on Machine Learning*, pp. 278–286. Morgan Kaufmann: Nashville, TN (1997)

Ruby, D., Kibler, D.: SteppingStone: An empirical and analytical evaluation. In: *Proceedings of the Tenth National Conference on Artificial Intelligence*, pp. 527–532. AAAI Press: Menlo Park, CA (1991)

Schmid, U., Kitzelmann, E.: Inductive rule learning on the knowledge level. *Cognitive Systems Research*, **12**, 237–248 (2011)

Shavlik, J.: Acquiring recursive and iterative concepts with explanation-based learning. *Machine Learning*, **5**, 39–70 (1990)

Shivashankar, V., Kuter, U., Nau, D., Alford, R.: A hierarchical goal-based formalism and algorithm for single-agent planning. *Proceedings of the Eleventh International Conference on Autonomous Agents and Multiagent Systems*, pp. 981–988. Valencia, Spain (2012)

Sleeman, D., Langley, P., Mitchell, T.: Learning from solution paths: An approach to the credit assignment problem. *AI Magazine*, **3**, 48–52 (1982)