Proceedings of the Ninth Annual Conference on Advances in Cognitive Systems Submitted 9/2021; published 11/2021

Hierarchical Problem Networks for Knowledge-Based Planning

Pat Langley PATRICK.W.LANGLEY@GMAIL.COM
Institute for the Study of Learning and Expertise, Palo Alto, California 94306 USA

Howard E. Shrobe HES@CSAIL.MIT.EDU
Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA 02139 USA

Abstract

In this paper, we reconsider the representation and use of expertise about sequential goal-directed activities. We discuss previous research on this topic, identify its limitations, and present a new theoretical framework – *hierarchical problem networks* – that addresses them. The core idea is that procedural knowledge consists of conditional methods that decompose problems – sets of goals – into ordered subproblems. Another innovation is that methods incorporate tests that forbid their use when certain goals are unsatisfied. We state the theory's key postulates about representation and processing, after which we describe HPD, a problem-solving architecture that makes these assumptions operational. Next we report empirical demonstrations of HPD's behavior in three planning domains, including studies of the relative importance of different types of conditions on constraining search. In closing, we review the theory's main ideas and their intellectual precursors, along with our plans for future research.

1. Introduction and Motivation

Expertise plays a critical role in both human and machine intelligence. Multi-step reasoning and heuristic search are essential to handling complex cognitive tasks but, without knowledge to constrain and guide them, they lead invariably to undirected, inefficient mental processing. This reliance on expertise cuts across all facets of cognition, including language and vision, but here we will focus on its role in sequential goal-directed activity. The prototypical example of such problem solving is the generation of action chains that achieve some objective, although similar issues arise in executing such sequences and even in understanding the actions of others.

The notion of a *plan* has a long history in artificial intelligence and cognitive science, dating back at least to Miller, Galanter, and Pribram's (1960) *Plans and the Structure of Behavior*. They used this term to refer to the mental encoding of a general physical procedure, such as hammering a nail into a board. This sense still appears on occasion in the cognitive systems literature, as in papers on the Soar architecture (Laird, 2012), but since the 1980s 'plan' has more widely been used to connote a specific sequence of actions that solve a particular problem. We will use the second sense in this paper, but we will be concerned primarily with how to represent and use generalized procedures like those that Miller et al. posited in their groundbreaking research.

There have been many proposals for how to encode such knowledge, which we review briefly in the next section along with their limitations. After this, we propose a new theory of procedural expertise – *hierarchical problem networks* – that builds on earlier work but also responds to these drawbacks. We state the theory's tenets about how goal-indexed procedures are represented and how they generate hierarchical plans efficiently. In addition, we describe HPD, a problem-solving architecture that incorporates these postulates, and we report empirical studies on planning tasks that demonstrate its abilities. We conclude by revisiting the theory's main commitments, their links to previous work, and areas in which additional research could extend the framework.

2. Representational Frameworks for Procedural Expertise

To reproduce human-like behavior, formalisms for procedural expertise must satisfy a number of criteria, only some shared with other facets of cognition (Langley, Shrobe, & Katz, in press). The most obvious is that they encode generalized knowledge about sequential activity over time. Moreover, a notation should be modular, in that procedures are composed of distinct elements, and it should support relational descriptions of situations and actions. Other desirable features include the ability to specify conditional and disjunctive behavior, decompose complex behaviors into simpler ones, and even handle recursive invocations. These are shared with formalisms for other cognitive abilities, but another element is distinctive to procedural expertise: causal links between activities and goals they achieve.

Frameworks for sequential behavior vary in the extent to which they emphasize search vs. knowledge. At one end of the continuum lie classic planning systems (Kambhampati, 1997), which specify minimal domain content (as operators that describe conditional effects of actions) but otherwise rely on heuristic search. These produce causally-connected action sequences that achieve goals, but they do not store or access explicit procedural knowledge structures. At the other end are traditional programming languages, which let one specify general procedures that include arguments, conditional and iterative statements, subroutines, and recursion. However, these formalisms seldom support relational descriptions, causal links, or explicit reasoning about goals. We desire a representation for procedural expertise that falls midway between these two extremes.

Research on learning for planning and problem solving has produced a variety of ways to encode such knowledge. Search-control rules[1] for selecting, rejecting, or preferring alternatives are highly modular and can influence decisions about which operators, goals, or states to choose during problem solving (e.g., Laird, 2012; Minton, 1988). Macro-operators offer a more procedure-like notation that specifies effects of a sequence of operators when applied under certain conditions (e.g., Iba, 1989). Extensions to this framework support disjunctive, iterative, and recursive constructs (e.g., Shell & Carbonell, 1989), bringing them into even closer alignment with classic notions of procedures. A third paradigm, analogical planning, stores even larger structures that are retrieved and adapted to new situations (e.g., Veloso & Carbonell, 1993). These lack explicit notations for conditional and iterative behavior, but the adaptation process can produce similar effects.

1. One can also encode search-control expertise in terms of numeric evaluation functions, but we will limit our treatment to symbolic structures that describe the organization of procedures.

The closest relatives of procedural formalisms in the planning literature are hierarchical task networks (e.g., Nau et al., 2003) or HTNs. These retain modular, relational, and sequential constructs, but they specify expertise as a set of *methods*, each of which indicates how to accomplish a named task by decomposing it, under certain conditions, into ordered subtasks. Different methods may address the same task, thus supporting disjunction and recursion. Hierarchical goal networks or HGNs offer an important variation on this idea (Shivashankar et al., 2012). These replace tasks and subtasks with goals and subgoals that methods achieve, providing direct support for goal-directed processing. This framework also simplifies the challenge of learning hierarchical procedures (Langley & Choi, 2006; Marsella & Schmidt, 1993; Reddy & Tadepalli, 1997). Both HTNs and HGNs mimic traditional procedures while offering modular, relational encodings. The tasks and subtasks of HTNs, which accept arguments, map onto routines and subroutines in standard programming languages, but they make only indirect contact with goals through primitive operators. In contrast, HGN methods are indexed by the goals they achieve while retaining HTNs' other desirable features.

However, both frameworks have limited ability to specify search-free procedures that take goal interactions into account. For instance, a standard HTN procedure for building towers in the Blocks World requires a separate method for each number of blocks in the target tower and, moreover, that method must specify the blocks as ordered arguments in its head.[2] In other words, it would include one decomposition rule with the head `(build-tower ?X ?Y)`, another for `(build-tower ?X ?Y ?Z)`, and so forth, which is neither concise nor general. One can specify more general methods, but the HTN solver would then need to carry out search for a plan that achieves all of the goals. Classic HGNs also require one method for each tower configuration, as they have no way to encode knowledge about goal ordering. This drawback suggests the need for a new formalism for procedural expertise that indicates not how to decompose *individual* tasks or goals, but how to decompose *problems* stated as *sets* of goals, which in turn lets one encode relations among them. In the next section, we present a framework that incorporates this insight.

3. Hierarchical Problem Networks

Most research on cognitive systems aims to reproduce key phenomena associated with intelligence (Langley, 2018). In this case, we desire a computational theory of procedural expertise that replicates the major characteristics of human problem solving. These include cognitive abilities for:

- Generating novel sequences of actions that achieve sets of goals;
- Taking both goals and situations into account when selecting actions;
- Decomposing complex activities into simpler ones when appropriate;
- Using domain knowledge to guide or constrain search when available; and
- Carrying out search through a problem space to solve problems when necessary.

Each of the frameworks just reviewed supports these five abilities, but we have also seen that they have limits. In this section, we present a new theory – *hierarchical problem networks* or *HPNs* – that incorporates many of their ideas but also extends them in new directions.

2. This holds for the most commonly used HTN formalisms, like SHOP2 (Nau et al., 2003). As we discuss later, various extensions have addressed this limitation. Another approach, used in Prolog (Clocksin & Mellish, 1981), encodes arguments as lists with variable length, but these lists specify the solution rather than placing constraints on it.

3.1 Representational Postulates

Theoretical accounts of cognitive systems typically begin by discussing representations, as they constrain the mechanisms that operate on them. Many of our representational statements will take the form of definitions, but they have implications for system behavior when they are combined with claims about processes. We can further distinguish between short-term structures that change rapidly over time and long-term elements that remain reasonably stable. We will start by characterizing the former class of mental entities.

We are focused on tasks that involve sequential activity, in particular the generation of plans that comprise an ordered set of actions. Like other work in this area, our framework must represent situations that arise during the course of a sequential plan. Thus, we say that:

- A **state** is a conjunctive set of relational facts that describe a situation.

For example, a common modular encoding for the Blocks World relies on the relations `on`, `ontable`, `clear`, `holding`, and `hand-empty`. Each predicate takes one or more arguments, which may be shared across different facts to specify a physical configuration. We must also specify the purpose or objective of the sequential activity. To this end, we say that:

- A **goal** is a desired relation, possibly with variables, and a **problem** is a set of goals that describe a class of desired states.

This description is also modular and relational, and it uses the same formalism as states, but it can omit some elements, which means that it can specify a *set* of desirable situations. Note that our formulation differs from some earlier ones in that a problem does not refer to an initial state.

The aim of planning is to find solutions to a problem that achieve its goals. Each action changes some elements of the current state to produce a successor state. For a plan to be successful, its actions must produce a state that satisfies all the problem's goals. In some cases, these actions may be only partially ordered, although we will focus on total orderings here. However, we are especially interested in a particular class of plans that provides additional structure to the solution:

- A **hierarchical plan** is a recursive decomposition of a problem into subproblems in which operator instances serve as terminal nodes.

This idea is well established, dating back to some of the earliest AI work on problem solving (e.g., Newell, Shaw, & Simon, 1960). Of course, it has also played a central role in more recent research on HTNs, HGNs, and kindred approaches to plan generation.

Our theoretical framework also assumes that a problem solver stores long-term structures about how to generate plans. We refer to this as *procedural knowledge* because it encodes strategies or 'programs' for classes of problems. We can make statements about the form of this content:

- Procedural knowledge is encoded as a set of **methods**, each of which decomposes one of a problem's goals into a set of ordered subproblems.

This postulate is shared with the HTN and HGN frameworks, and even with logic programs, but these classic approaches decompose goals into subgoals or tasks into subtasks. We maintain instead that each method decomposes a problem (a set of goals) into ordered subproblems (also sets of goals), which imposes a very different organization on candidate solutions.

Each HPN method indicates how to break down a goal in the current problem, but a given decomposition may only be appropriate in some situations. Thus, we also posit that:

- A method specifies the **state-related conditions** under which a decomposition is acceptable.

Such conditions on the state are standard in existing frameworks like HTNs and HGNs, so our theory is not distinctive on this front. However, we further postulate that decomposition rules can include a second type of condition which serves a different purpose:

- A method specifies **goal-related conditions** under which a decomposition is **not** acceptable.

For instance, in the Blocks World, if one has goal for A to be on B, then one should *not* apply a decomposition rule to achieve this aim if there exists an *unsatisfied* goal for B to be on C. The reason is straightforward; if we place A on B before putting B on C, then we must undo the first in order to achieve the second. The HTN and HGN frameworks do not incorporate such constraints in their methods, although they can appear in search-control rules.

However, hierarchical methods with state and goal conditions are not by themselves sufficient to specify solutions for planning tasks. For this reason, the HPN framework also includes long-term cognitive structures that make contact with executable or observable actions:

- An **operator** is a special type of method that specifies the conditional effects of an action.

As we have seen, operator instances serve as terminal nodes in hierarchical plans, but we must specify their effects on states to make them operational. In the Blocks World, there is an operator for stacking A on B when the hand is holding A and when B is clear. The effects under these conditions are that A is on B and that the hand is no longer holding B.

Finally, the theory of hierarchical problem networks specifies an important connection between such domain operators and the form taken by the elements of procedural expertise. More precisely, it claims that this knowledge takes a particular form:

- Each method has an effect of some operator *O* as its head, a subset of *O*'s conditions as its first subproblem,[3] and the application of *O* as its second subproblem.

This structural constraint gives hierarchical problem networks a very different feel from classic HTNs or HGNs. Instead, they come closer to generalized traces of partial-order plans that can encode solutions to tasks with arbitrary numbers of objects. We will return to this point when we discuss extensions to the framework, including ways that a problem solver might learn new methods.

3.2 Processing Postulates

Now that we have presented claims about representation, we can discuss processes that operate over these structures. The first postulate is suggested by the framework's focus on how procedural knowledge is used to generate hierarchical plans:

- Problem solving **recursively decomposes** a problem into subproblems in order to find an operator sequence that achieves the problem's goals.

This states that problem solving occurs in a top-down, goal-driven manner, in which each step breaks down a problem into component subproblems. A typical refinement replaces a goal in a

3. Not all of *O*'s conditions contribute to the first subproblem; some may instead appear in the method's state conditions.

problem *P* with one subproblem to satisfy an operator *O*'s conditions, another one to apply *O*, and a final subproblem to achieve *P*'s other goals. The theory also includes details about this activity. In particular, it posits that decomposition is a serial process that introduces one refinement at a time:

- Problem decomposition **iteratively** examines the topmost element of a **problem stack** and then places new subproblems above it.

The problem stack is a dynamic structure that stores intermediate results. The idea of processing the top element of a stack appears in other accounts of multi-step cognition that apply one rule at a time. Indeed, this assumption is even more widespread than commitments to hierarchical decomposition.

Moreover, the theory of hierarchical problem solving enumerates the mechanisms that take place on each cognitive cycle, specifically stating that:

- Problem decomposition relies on three main subprocesses: method **matching**, method **selection**, and method **expansion**.

The first stage finds all methods whose head unifies with unsatisfied goals in the current problem, *P*, whose state conditions match the state and whose goal conditions do *not* match. This produces a set of applicable method *instances*, each of which binds variables to constants. The second stage chooses one method instance, *M*, for use when decomposing *P*. Finally, the subproblems of *M* are added to the stack, where they influence processing on later cycles.

The problem solver must also know when to halt decomposition, which occurs when the current problem's goals are satisfied or when it involves applying an operator. In such cases, it pops the problem from the stack and focuses attention on the one below it. However, sometimes the selected decomposition does not lead to a solution, which requires a final postulate:

- Problem solving involves **search through a space of decompositions** defined by the methods, problem goals, and initial state.

The problem stack lends itself to depth-first search with backtracking, although that is not the only way to organize exploration, and heuristics for method selection can also aid processing. Moreover, appropriate conditions on methods eliminate search so that problem solving operates in a deterministic manner, with any remaining choices giving equivalent serializations of a partial-order plan.

3.3 Comparison to Other Frameworks

As noted earlier, hierarchical problem networks share key features with other planning frameworks, but they also differ from these predecessors in important ways. Table 1 compares classic operator-driven planning, HTNs, HGNs, and HPNs in terms of seven assumptions about representation and processing. The first row shows that all four paradigms support the generation of sequential plans that achieve goals, but that only the latter three use hierarchical methods to decompose complex activities into simpler ones and apply them only when certain conditions hold in the current state. Thus, HPNs draw substantially on earlier approaches to hierarchical planning, although they come closest to HGNs, which also index methods by the goals they achieve, and to the ICARUS architecture (Langley & Choi, 2006), which introduced this idea in the context of reactive control.

The final three rows review some more distinctive characteristics. HPNs differ from other hierarchical frameworks, at least in their most commonly used forms, by decomposing problems –

Table 1. Comparison of hierarchical problem networks with three other approaches to planning in terms of seven characteristics. The symbol • indicates that a feature is present, whereas ○ denotes that a feature is absent. Subsection 3.1 discusses the distinguishing characteristics in greater detail.

REPRESENTATIONAL AND PROCESSING ASSUMPTIONS	*Classic Planners*	*HTN Planners*	*HGN Planners*	*HPN Planners*
Generate sequential plans that achieve goals	•	•	•	•
Decompose complex activities hierarchically	○	•	•	•
Methods require that relations hold in state	○	•	•	•
Methods indexed by goals they achieve	○	○	•	•
Decompose problems into subproblems	○	○	○	•
Methods require that goals are not unsatisfied	○	○	○	•
Methods are linked to primitive operators	○	○	○	•

sets of goals – rather than decomposing individual tasks or goals. This feature enables a second one: the ability to specify methods that apply only when certain goals are not unsatisfied in the current state, which in turn constrains the order in which these goals are addressed. Finally, HPNs assume that each method is linked directly to a primitive operator that determines its head and subproblems. Together, these assumptions provide greater representational power than traditional hierarchical techniques, offering an effective and compact way to encode procedural expertise while still supporting heuristic search when such knowledge is limited.

4. The HPD Problem-Solving Architecture

We have developed a problem-solving architecture – the Hierarchical Problem Decomposer (HPD) – that incorporates these postulates. As Langley (2018) notes, there are usually many ways to make a theory operational and thereby testable, and here we present only one alternative. We start by examining HPD's representational formalism, then describe the mechanisms that operate over it.

4.1 Representation in the HPD Architecture

The HPD architecture embodies the representational commitments listed in the previous section, reflecting them in its syntax for encoding long-term and short-term structures, as in many cognitive architectures (Langley, Laird, & Rogers, 2009). This provides a programming language for denoting states, problems, and procedures that underlie problem solving. The notation has much in common with those used in logic programming (Lloyd, 1984) and hierarchical task networks (Nau et al., 2003), but there are also some important differences with implications for processing.

For example, Table 2 (a) specifies a state from the Blocks World that involves 13 distinct literals. These use the predicates `on`, `ontable`, `clear`, `holding`, and `hand-empty`, which can share arguments to describe relational configurations. Similarly, Table 2 (b) presents a problem from the Blocks World. This also consists of relational literals, but it refers to *desired* state elements and it omits ones that hold no interest. Here the problem's goals specify a tower with `A` on `B`, `B` on `C`, and

Table 2. A state description from the Blocks Worlds encoded as a set of relational facts and a problem description denoted as a set of goals.

```
(a) Initial state:
    ((block A)(block B)(block C)(block D)
     (ontable A)(ontable B)(ontable C)(ontable D)
     (clear A)(clear B)(clear C)(clear D)(hand-empty))
(b) Goal description:
    ((on A B)(on B C)(ontable C))
```

C on the table, but they do not mention ontable, clear, holding, or hand-empty, nor do they refer to block D. Problem statements in HPD may also include unbound variables as arguments of goal predicates, although they do not appear in this example.

Table 3 presents a hierarchical problem network for the Blocks World. Each of the four methods includes a head, a set of state conditions, an optional set of goal conditions, and an ordered set of subproblems. For instance, the first method's head is (on ?X ?Y), which refers to a goal in the current problem. The :conditions field specifies this rule should only apply when the current state includes (block ?X) and (block ?Y). In addition, the :unless-goals field indicates that the method should *not* apply if there exists an *unsatisfied* goal of the form (on ?Y ?ANY), where ?Y is bound in an earlier field but ?ANY is not, or the form (ontable ?Y). Finally, the :subproblems field lists two subproblems to replace the goal in the head. The first specifies the goals ((holding ?X)(clear ?Y)) and the second to apply the operator (stack ?X ?Y).

Operators are a special type of method that describe the conditional effects of actions. Table 4 shows four operators from the Blocks World, in HPD syntax, that many readers will find familiar. Each entry specifies a name and arguments in its head, conditions that must match the current state for application, and changes to the state when such application occurs. For example, the first operator has the head (stack ?X ?Y) and the conditions (holding ?X) and (clear ?Y). The effects are that these relations cease to hold, but that (on ?X ?Y) and (hand-empty) become true. Operators do not include :unless-goals conditions, because they deal with isolated actions, or a :subproblems field, because they serve as terminal nodes in plans.

As noted earlier, each method in a hierarchical problem network has a clear relation to some operator. For example, the head of the first method in Table 3, (on ?X ?Y), is an effect of the operator stack in Table 4. The first subproblem contains conditions of the operator achievable by other actions, whereas the second subproblem contains the operator's head, indicating that it should be applied after its conditions are met. A similar relation holds between the second method in Table 3 and the operator pickup, although one of the latter's conditions, (ontable ?X), is in the method's :conditions field rather than its first subproblem. The reason is that one cannot achieve the goal (ontable ?X) without first achieving (holding ?X), which is circular.

HPD's formalism for expertise maps directly onto classic features of procedures or programs. Each method corresponds to a conditional statement that applies only when relevant. Moreover, a method specifies how to decompose a problem into subproblems, which correspond to subroutine calls, some of which lead to recursion. Programs terminate upon reaching an applicable operator

Table 3. Six methods for the Blocks World that include a head, state conditions, optional goal conditions, and one or two subproblems. In each case, the final subproblem refers to application of an operator. This HPN enodes a procedure that solves a broad class of problems in the domain without the need for search.

```
((on ?X ?Y)
 :conditions   ((block ?X)(block ?Y))
 :subproblems  (((clear ?Y)(holding ?X))((stack ?X ?Y)))
 :unless-goals ((on ?Y ?ANY)(ontable ?Y)))
((holding ?X)
 :conditions   ((block ?X)(ontable ?X))
 :subproblems  (((clear ?X)(hand-empty))((pickup ?X)))
 :unless-goals ((clear ?ANY)))
((holding ?X)
 :conditions   ((block ?Y)(block ?X)(on ?X ?Y))
 :subproblems  (((clear ?X)(hand-empty))((unstack ?X ?Y)))
 :unless-goals ((clear ?ANY)))
((clear ?Y)
 :conditions   ((block ?Y)(block ?X)(on ?X ?Y))
 :subproblems  (((clear ?X)(hand-empty))((unstack ?X ?Y))))
((hand-empty)
 :conditions   ((block ?X)(holding ?X))
 :subproblems  (((putdown ?X)))
 :unless-goals ((clear ?ANY)))
((ontable ?X)
 :conditions   ((block ?X))
 :subproblems  (((holding ?X))((putdown ?X))))
```

and they 'return' a result for each subproblem that is encoded as a hierarchical plan. The major disconnect lies in HPD's reliance on goals, rather than routine names, to index and invoke methods, but these serve the same function as traditional procedural calls.

We should also consider the form of solutions that the HPD architecture generates. Table 5 shows a hierarchical plan for the initial state and problem in Table 2, with indentations indicating levels in the solution. The top-level problem, `((on A B)(on B C)(ontable C))`, appears in the first line. The plan breaks this down into three subproblems: `((clear C)(holding B))`, `((stack B C))`, and `((ontable C)(on A B))`. The first subproblem in turn has three subproblems: `((clear B) (hand-empty))`, `((pickup B))`, and `((clear C))`. Some terminal nodes involve applying an operator, whereas others correspond to subproblems that the current state already satisfies, and do not require any further effort. This solution has a right-branching structure, but other plans, say for clearing a block embedded in a tower, will be left branching.

4.2 Processing in the HPD Architecture

In accordance with the theory, HPD operates in cycles. On each iteration, processing focuses on the topmost problem *P* in the problem stack in the context of the current state *S*. The primary steps include pushing new subproblems of *P* onto the stack, popping *P* from the stack, applying

Table 4. Four operators for the Blocks World, each specifying an action (head), conditions, and effects.

```
((stack ?X ?Y)
 :conditions   ((block ?X)(block ?Y)(holding ?X)(clear ?Y))
 :effects      ((on ?X ?Y)(hand-empty)(not(clear ?Y))(not(holding ?X))))
((pickup ?X)
 :conditions   ((block ?X)(ontable ?X)(clear ?X)(hand-empty))
 :effects      ((holding ?X)(not(hand-empty))(not(ontable ?X))))
((unstack ?X ?Y)
 :conditions   ((block ?X)(block ?Y)(on ?X ?Y)(clear ?X)(hand-empty))
 :effects      ((clear ?Y)(holding ?X)(not(on ?X ?Y))(not(hand-empty))))
((putdown ?X)
 :conditions   ((block ?X)(holding ?X))
 :effects      ((ontable ?X)(hand-empty)(not(holding ?X))))
```

an operator to update *S*, and adding or removing elements from the hierarchical plan. The action that HPD takes depends on the results of comparing the problem *P*, the state *S*, and the heads and conditions of methods. One of four situations will hold, each with an associated response:

- If the topmost problem *P* matches the current state *S* (i.e., all of *P*'s goals are satisfied), then HPD simply pops *P* from the problem stack.
- If the topmost problem *P* is to apply an operator *O* and if *O*'s conditions match the current state *S*, then HPD instantiates *O*'s effects, uses them to update *S*, and removes *P* from the stack.
- If a method instance *M* is applicable to problem *P* in state *S*, then HPN pushes *M*'s subproblems onto the problem stack and inserts them as children of *P* in the hierarchical plan.
- If the plan length exceeds a limit, or if no method applies to problem *P* in state *S*, then HPD pops *P* from the stack and removes *P* and its siblings from their parent in the hierarchical plan.

The final situation will not occur during a problem-solving run if appropriate methods, with the necessary conditions, are known. In such cases, HPD will select a reasonable decomposition on each cycle and find a hierarchical plan without needing to backtrack. In contrast, if the system lacks these conditions, then it can be led astray and must carry out search before it finds a solution.

Let us consider in more detail these stages of processing. To match a method *M* against a problem *P* and state *S*, HPD first compares the goals in *M*'s head *H* with *P*'s unsatisfied goals *G* (i.e., those not satisfied by *S*). If *H* unifies with *G*, then the system compares *M*'s conditions *C* with the state *S*, subject to bindings from the head. If *C* matches against *S* in a consistent way, then HPD compares the goal descriptions *U* in *M*'s `:unless-goals` field with *P*'s unsatisfied goals, again taking bindings into account. If *U* matches successfully against *G*, then the method instance fails because the unless condition blocks its applicability.

For example, suppose the top problem on the stack is the goal set `((on A B)(on B C))` and the current state is

```
((block A)(block B)(block C)(ontable A)(ontable B)
 (ontable C)(clear A)(clear B)(clear C)(hand-empty)).
```

Table 5. A hierarchical plan that solves the problem in Table 2 (b) given the initial state in Table 2 (a).

```
((on A B)(on B C)(ontable C))
    ((clear C)(holding B))
        ((clear B)(hand-empty))
        ((pickup B))
        ((clear C))
    ((stack B C))
    ((ontable C)(on A B))
        ((clear B)(holding A))
            ((clear A)(hand-EMPTY))
            ((pickup A))
            ((clear B))
        ((stack A B))
        ((ontable C))
```

In this situation, the head of the first method in Table 3, `(on ?X ?Y)`, matches the problem description in two distinct ways. One goal, `(on A B)`, matches with bindings `?X` → `A` and `?Y` → `B`, while the other goal, `(on B C)`, matches with bindings `?X` → `B` and `?Y` → `C`. The `:conditions` field matches successfully for both sets of bindings, but the first match is rejected because the `:unless-goals` condition, `(on ?Y ?ANY)`, matches a goal, `(on B C)`, that is *not* satisfied. No analogous unsatisfied goal exists for the second match, so HPD treats it as applicable. Such goal conditions constrain the order in which it addresses a problem's goals.

Once HPD has found a set of acceptable methods instances, it selects one of them for application. The current implementation picks a candidate at random, but it could instead incorporate a mechanism for conflict resolution like that used in production systems (Neches, Langley, & Klahr, 1987). For instance, the system might favor methods whose heads unify with more unsatisfied problem goals or whose subproblems introduce fewer unsatisfied goals. Such preferences would serve as heuristics to guide search through the space of decompositions. They would not be guaranteed to reduce effort or to generate better plans, but they could aid performance substantially on average.

After the architecture has selected a method instance *M* to decompose the topmost problem *P*, it instantiates *M*'s subproblems based on the accumulated variable bindings and adds them on top of the problem stack. If *M* includes two subproblems, *S1* and *S2*, then HPD introduces not only *S1* and *S2*, in that order, but also a third subproblem that comprises any goals in *P* not mentioned in *M*'s head, including ones that are currently satisfied. In some situations, this set difference will be empty, but in other cases goals will remain that the system must address later. In addition to adding these subproblems to the stack, HPD stores them as children of *P* in the hierarchical plan.

Returning to our example, the first method in Table 3 specifies two subproblems. After substituting bound variables, these become `((holding B)(clear C))` and `((stack B C))`, which HPD pushes onto the problem stack. However, because the current problem contains two goals, `((on A B)(on B C))`, and the decomposition only addresses one of them, the system also adds a third subproblem – `((on A B))` – that includes the remaining goal. After the application of this method instance, the stack retains the original top-level problem, but three new ones

now sit above it. In this manner, HPD continues to decompose problems and push their subproblems onto the stack, where they become the foci of attention on future cognitive cycles.

This activity continues until the topmost problem *P* is to apply an operator *O* and *O*'s conditions match the current state *S*. In such cases, HPD instantiates *O*'s effects, uses them to update *S*, and removes *P* from the stack. If the new top-level problem is satisfied by the new state, then the system removes it as well; otherwise, it finds a relevant method to solve it. If things go well, the architecture eventually addresses each problem on the stack, including the original one, and returns a hierarchical plan. When HPD cannot find a decomposition that it has not already tried and rejected, it abandons the current problem and backtracks. However, given appropriate methods, the system avoids search entirely and mimics an algorithmic procedure. State and goal conditions join forces to produce this behavior, the former influencing which bindings are selected and the latter ensuring that goals are addressed in the right order. As Table 3 illustrates, the encoding of procedural knowledge for a given domain can be remarkably simple. Besides the operators themselves, often this requires only one or two methods per operator that specify how to decompose problems into subproblems.

4.3 Implementation and Use Details

We have implemented the HPD architecture in Steel Bank Common Lisp. The software supports the syntax outlined above for specifying states, problems, and hierarchical problem networks. It also incorporates mechanisms for interpreting these structures. These include modules for matching, selecting, and applying HPN methods in ways that update the problem stack and the state, as well as for backtracking when the need arises. In other words, HPD both operationalizes the theory of hierarchical problem solving presented earlier and offers a programming language for stating knowledge about goal-directed sequential activities.

To apply HPD in a specific scenario, the user loads a file that contains a set of hierarchical methods, an initial state described as a set of relational literals, and a set of goals that encode a problem. The architecture is called with these arguments, along with the maximum length for acceptable plans. The interpreter repeatedly focuses on the topmost problem, popping it from the stack if satisfied, selecting an untried method that adds new subproblems to stack if not, or abandoning the current problem and backtracking if necessary. Upon completion, HPD returns a hierarchical plan that transforms the initial state into one that satisfies the problem goals.

5. Empirical Studies of Hierarchical Problem Networks

Hierarchical problem networks offer a promising framework for encoding and using procedural expertise, but it is important to demonstrate that they support the abilities listed in Section 2. Thus, we must show that HPD can generate action sequences which achieve a set of goals, take both goals and situations into account when selecting actions, decompose complex activities into simpler ones, carry out search when choices arise, and use domain knowledge to constrain this search. To this end, we developed HPD knowledge bases for three planning domains and a suite of test problems for each one. In this section, we describe the domains, the hierarchical problem networks, and basic tests of their behavior. Next we report lesion studies that remove some of the methods' contents to determine their effects on search. We also present controlled experiments that vary problem characteristics to show how HPD programs scale to increasing complexity.

5.1 Demonstrations of HPD's Abilities

We selected three well-studied domains from the AI literature. These have typically been used to evaluate planning systems that rely on heuristic search, but people who are familiar with them can solve novel problems with little or no backtracking, which makes them ideal for showing HPD's ability to encode procedural expertise. For each domain, we describe the predicates used to specify goals and states, the operators that transform states, and the hierarchical problem networks that we developed to solve tasks. We also report on the number of problems used to test the HPD programs and their complexity in terms of goals and solution lengths.

The Blocks World involves changing an initial configuration of blocks into another configuration that satisfies a goal description. There are six predicates for states – `block`, `on`, `ontable`, `clear`, `holding`, and `hand-empty` – and four operators – `stack`, `unstack`, `putdown`, and `pickup`. The HPD program for this domain, shown in Table 3, includes six methods, including two for the `holding` relation and one each for the other dynamic predicates. We tested this knowledge base on 20 tasks from the Blocks World. Solution lengths varied from four to 12 steps and every problem involved five goals. For each task, HPD found the expected hierarchical plan without backtracking. In other words, the architecture used the methods like a deterministic procedure, as intended.

The Logistics domain requires transporting packages from initial to target locations. Static relations include `object`, `truck`, `airplane`, `location`, `city`, `airport`, and `in-city`, while the only dynamic predicates are `at` and `in`. The six operators are `load-truck`, `drive-truck`, `unload-truck`, `load-airplane`, `fly-airplane`, and `unload-airplane`. The knowledge base for logistics included seven methods, four for the relation `at` and three for `in`. We evaluated the HPD program on ten problems from this domain, with the number of goals varying from one to three and with solution lengths ranging from four to 18 steps. As before, the architecture found a hierarchical plan for each problem with no search. In a few cases, it made it extra round trips for packages with the same initial and target locations, but it never required backtracking.

The Depots domain combines elements of the Blocks World and Logistics, in that it involves moving crates from some pallets to others and stacking them in specified arrangements. Here the static predicates include `place`, `truck`, `pallet`, `surface`, `crate`, and `hoist`, while the dynamic relations are `available`, `lifting`, `at`, `on`, `in`, and `clear`. There are five operators are `drive`, `lift`, `drop`, `load`, and `unload`, and we provided HPD with eight methods, two for the relation `on` and `lifting` and one for every other dynamic predicate. We ran the architecture on ten Depots problems that involved from two to three goals and whose solutions ranged from eight to 18 steps. As expected, this knowledge base found solutions to every task without resorting to search, as its methods made correct choices at each stage of the problem-solving process.

5.2 Lesion Studies of HPD's Problem Solving

Our initial results provide evidence that hierarchical problem networks are an effective way to encode procedures, but they do not reveal their sources of power. To gain further insights, we devised lesion studies to compare the behavior of the basic HPD programs with that for three variants with facets of their methods removed. In one version, we eliminated both state and goal conditions, so that each method included only static state conditions. For instance, for the third method in Table 4, we excised the `:unless-goals` field and moved the state condition `(on ?X ?Y)` into

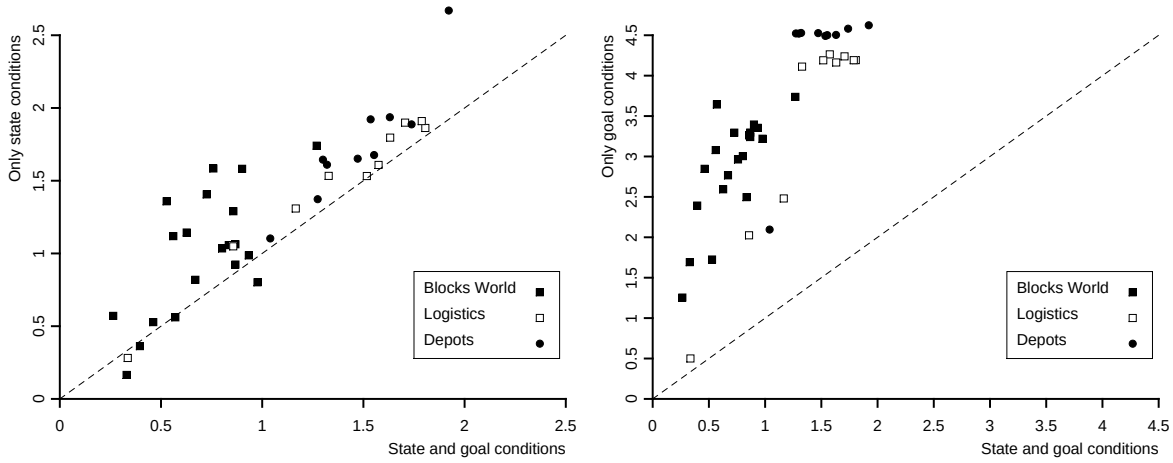

Figure 1. Scatter plots that compare the effort by different knowledge bases to solve problems in three domains. The left graph shows the CPU time taken by a complete HPD program vs. the time needed by one that lacked goal conditions. The right graph shows the time for a full HPD program and one that lacked dynamic state conditions. Points above the diagonal denote problems in which the complete knowledge knowledge base was more efficient. Each point is averaged across 30 separate runs and shown on logarithmic scales.

the `:subproblems` field. Another variant retained dynamic state conditions but omitted goal conditions, while the final version kept goal conditions but not dynamic state conditions. We expected that search, and thus time to find solutions, would increase in all three variants, but we did not know if state or goal conditions would be more important. For each domain, we told HPD to try no more than 20,000 decompositions and to only consider solution paths with 20 steps or fewer. We ran the system on the 40 problems described above, averaging across 30 runs in each case.

Figure 1 presents the main results of this comparison as scatter plots. The graph on the left compares the CPU time, in logarithmic scale, for the full HPD knowledge base in each domain with one that omitted goal conditions. On nearly every problem, the unmodified programs fared better, although the lesioned variants still kept search down to a reasonable level. The graph on the right compares the problem-solving times for full knowledge bases with those for HPD programs that retained the goal (unless) conditions but not dynamic state conditions. The differences here are much greater, so much that on some tasks the system failed to find a solution before exceeding its allotted number of decompositions. This indicates that state conditions, which constrain methods' bindings, limited search more than goal conditions, which constrain goal orderings. The fourth variant, which elided both types of conditions, did substantially worse than the others, as predicted.

In addition, we recorded the number of decompositions tried during each run. We have not graphed these results because they were correlated highly ($r = 0.997$) with the observed CPU times for solved problems, which suggests that the cost of matching methods was not a factor. We also tracked the length of solutions found by the architecture with all four variations. This dependent variable was nearly unaffected by removal of state or goal conditions, except when the system failed to find a solution because it reached the limit placed on decompositions. In other words, the length of solutions remained the same even when substantial search was required to find them.

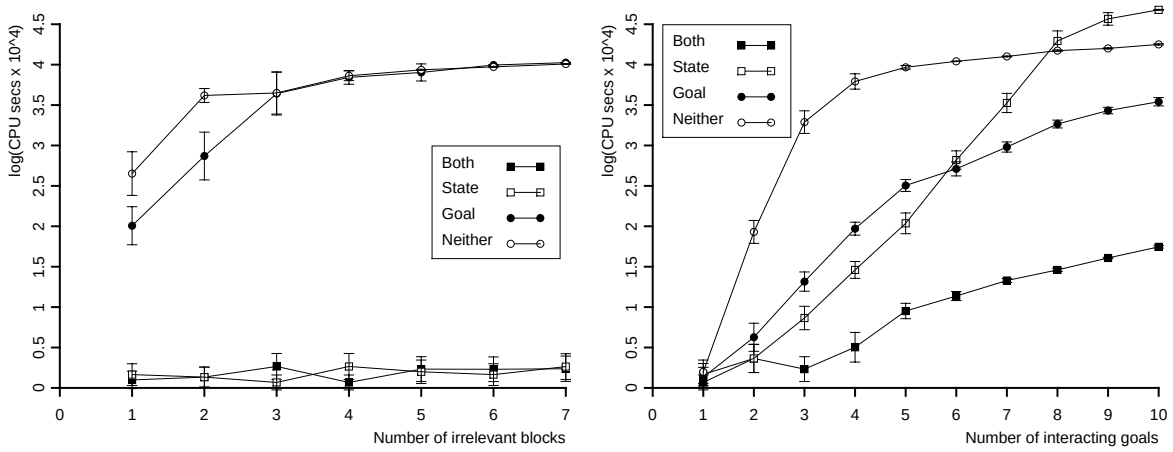

Figure 2. Scaling curves that show CPU times to solve problems in the Blocks World as a function of (left) the number of irrelevant blocks and (right) the number of interacting goals. Each graph includes a curve for the HPD program from Table 3, the same methods with only state conditions, with only goal conditions, and with neither of them. Each point is an average over 30 runs, with CPU times shown on a logarithmic scale and with error bars denoting 95 percent confidence intervals. The higher curves level off because search fails when HPD reaches the limit on plan length, thus bounding the time spent on problem solving.

5.3 Scaling Studies of HPD's Problem Solving

The previous comparisons revealed the relative importance of state vs. goal conditions on a sample of problems in three domains, but not their influence on scaling behavior, which is critical to combinatorial tasks. To understand this important issue, we devised two additional experiments, both involving the Blocks World. In the first study, we held constant the number of goals and varied the number of blocks that were irrelevant to problem solutions. We hypothesized that state conditions would keep search under control as we increased the number of objects, since they contsrain variable binding in methods and thus which subproblems are created. Figure 2 (left) shows results that are consistent with this prediction. The flat curve for methods with state conditions but no goal conditions is indistinguishable from that for the full HPD program in Table 3, implying that no search arose. In contrast, the plot for methods with goal constraints but no state conditions is no different from the HPD variant with neither of them; in both cases, search grows with the number of objects.

In the second study, we varied the number of interacting goals in problems that involved building a tower from blocks on the table. This also meant varying the number of objects, but we predicted that goal conditions would be more helpful in scaling with problem complexity, since they impose constraints on goal orderings, while state conditions would prove less useful. The graphs in Figure 2 (right) only largely agree with these expectations. The presence of goal conditions aids HPD slightly more than state conditions until six interacting goals, when the former starts to mitigate search substantially more than the latter. Note that processing time increases even for the full HPD program from Table 3, despite the fact that it requires no search, because the problems with more goals have longer solutions. However, the growth in solution time is slightly more than linear, which suggests that the cost of matching methods may also be increasing slowly.

6. Links to Earlier Research

The theory of hierarchical problem networks borrows ideas from the previous literature, but it also makes important new contributions. One of the framework's core tenets is that procedural knowledge is organized into modular, hierarchical methods. This assumption has been adopted widely, from early research on mathematical reasoning (Slagle, 1963) to the entire paradigm of logic programming (Lloyd, 1984). The closest links are to hierarchical task networks (Nau et al., 2003), hierarchical goal networks (Shivashankar et al, 2012), and the ICARUS architecture (Langley & Choi, 2006). However, the current theory is distinctive in that methods break down *problems* – sets of goals – rather than tasks or individual goals, and that each method includes a domain operator. This focus on problems lets HPNs specify explicit constraints on goal orderings that are not possible in other hierarchical frameworks, at least without considerable extensions.

There are certainly other approaches to specifying such restrictions. Problem-solving architectures like Soar (Laird, 2012) and Prodigy (Minton, 1988) incorporate control rules that state when to select, reject, or prefer certain goal orderings. Similarly, some extensions to hierarchical task networks, such as SHOP3 (Goldman & Kuter, 2019), include second-order inference rules that describe relations between states and goals, which can then appear in methods' conditions. Another scheme relies on some variety of temporal logic to encode such ordering constraints (e.g., Lin, Kuter, & Sirin, 2008) and eliminate alternatives that violate them. Hierarchical problem networks differ from these approaches by augmenting the formalism for describing methods themselves to incorporate this content, providing a compact and intuitive notation for procedural knowledge that does not require inference rules to constrain goal orderings.

Another central idea is that problem solving involves a process of recursive decomposition that produces hierarchical solutions. This assumption also appears in many earlier efforts on HTN and HGN planning, as does the theory's reliance on cognitive cycles that match, select, and apply rules. The key difference is that HPN solvers produce not a sequence of subtasks or subgoals, but rather a sequence of *subproblems*, each of which includes an operator instance. This approach lets a small set of general methods handle a broader class of problems than HTNs or HGNs, which require different decomposition rules for problems with different numbers of goals. Moreover, the backward-chaining character of HPN interpretation, combined with the use of goal conditions, is more akin to a knowledge-based version of partial-order planning (Kambhampati, 1997) than to other techniques for recursive decomposition.

Researchers have reported other systems that decompose problems into subproblems, but these have seldom incorporated procedural knowledge beyond the level of primitive operators. Newell, Shaw, and Simon's (1960) General Problem Solver introduced means-ends analysis, which uses operators in this manner, and successors like EUREKA (Jones & Langley, 2005) and HPS (Langley, Barley, & Meadows, 2018) invoke flexible variations on the same theme. However, these systems emphasized search through a space of possible decompositions, rather than drawing on domain knowledge to eliminate search. A closer analog to HPD is Marsella and Schmidt's (1993) REAPPR, which used decomposition rules to break problems into subproblems but did not associate an operator with each one. However, our framework does share this idea with Geib's (2016) lexicalized HTNs, in which each method includes an operator and in which each subtree of a hierarchical plan incorporates at least one operator instance.

Finally, the theory offers an effective way to specify general procedures that an interpreter like HPD can use to solve a broad range of problems without search. This supports constructs associated with traditional programming languages, including arguments, conditional and iterative statements, subroutines, and recursion. At the same time, the framework supports search through a problem space when the knowledge available is not enough to eliminate it entirely. Again, HTNs and cognitive architectures that incorporate search-control rules have similar capabilities, but they emphasize search and treat search-free procedures as a special case. HPNs reverse this focus, coming closer to Miller et al.'s 1960) original notion of plans as deterministic procedures. Most important, HPN programs obeys strong constraints, in that their structure follows directly from operators and their effects, making them elegant, straightforward to write, and potentially easy to learn.

7. Concluding Remarks

In this paper, we introduced a new representation for procedural expertise – *hierarchical problem networks* – that specifies how to decompose problems (sets of goals) into subproblems. HPN methods can include both state conditions, which constrain bindings on variables, and goal conditions, which constrain the order in which goals are tackled. We also described HPD, a problem-solving architecture that provides an HPN syntax and an interpreter that generates hierarchical plans. Given sufficient knowledge, the interpreter operates much like that for a procedural programming language, but has the ability to fall back on search when this content is incomplete. Finally, we reported empirical results with HPD on three planning domains. These showed that simple HPNs suffice to solve many problems without search, identified the contributions of state and goal conditions when search arises, and clarified how these knowledge sources influence scaling to problem complexity.

Despite this progress, we have only started to explore the potential of hierarchical problem networks for knowledge-based planning. In future work, we will extend the theory, and its implementation in the HPD architecture, to incorporate additional cognitive abilities by:

- Including a heuristic that favors methods with fewer unsatisfied goals in their initial subproblem, which should mitigate the current reliance on state conditions to reduce search and thus improve scaling to more difficult planning tasks;
- Extending hierarchical plans to allow OR branches for nondeterministic outcomes, which seems necessary to support information-gathering operators that are core to diagnostic procedures, like those common in mechanical and medical settings;
- Allowing methods and plans to contain durative operators with temporal constraints on start and end times, as needed for procedures that carry out actions in parallel, like those needed to prepare meals and operate vehicles; and
- Integrating HPN-guided problem solving with partial-order planning to take advantage of decomposition rules when such domain knowledge is present but to fall back on search using primitive operators when it is not available.

This last extension may hold the key to learning HPN methods. Recall that each method's head corresponds to an effect of an operator that appears as one of its subproblems, so one need only identify the state and goal conditions for each method. We hypothesize that one can determine goal conditions from orderings on operators found during partial-order planning and that one can acquire

state conditions with a variety of explanation-based learning which analyzes causal dependencies in problem solutions (Minton, 1988). Extending the framework to create HPN methods automatically from experience would remove the main bottleneck to making knowledge-guided planning preferable to the currently dominant knowledge-lean schemes.

Acknowledgements

This research was supported by Grant N00014-20-1-2643 from the Office of Naval Research, which is not responsible for its contents. We thank Gary Borschart, Sue Felshin, and Boris Katz for constructive discussions that influenced the ideas we have reported here.

References

Clocksin, W. F. & Mellish, C. S. (1981). *Programming in Prolog*. Berlin, Germany: Springer-Verlag.

Geib, G. (2016). Lexicalized reasoning about actions. *Advances in Cognitive Systems*, *4*, 187–206.

Goldman, R. P., & Kuter, U. (2019). Hierarchical task network planning in Common Lisp: The case of SHOP3. *Proceedings of the Twelfth European Lisp Symposium* (pp. 73–80). Genova, Italy.

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

Jones, R. M. & Langley, P. (2005). A constrained architecture for learning and problem solving. *Computational Intelligence*, *21*, 480–502.

Kambhampati, S. (1997). Refinement planning as a unifying framework for plan synthesis. *AI Magazine*, *18*, 67–97.

Laird, J. E. (2012). *The Soar cognitive architecture*. Cambridge, MA: MIT Press.

Langley, P. (2018). Theories and models in cognitive systems research. *Advances in Cognitive Systems*, *6*, 3–16.

Langley, P., Barley, M., & Meadows, B. (2018). Adaptive search in a hierarchical problem-solving architecture. *Advances in Cognitive Systems*, *6*, 251–270.

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

Langley, P., Laird, J. E., & Rogers, S. (2009). Cognitive architectures: Research issues and challenges. *Cognitive Systems Research*, *10*, 141–160.

Langley, P., Shrobe, H. E., & Katz, B. (in press). A cognitive task analysis of rapid procedure acquisition from instructional documents. *Advances in Cognitive Systems*.

Lin, N., Kuter, U., & Sirin, E. (2008). Service composition with user preferences. *Proceedings of the Fifth Annual European Semantic Web Conference* (pp. 629–643). Tenerife, Spain: Springer.

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

Neches, R., Langley, P., & Klahr, D. (1987). Learning, development, and production systems. In D. Klahr, P. Langley, & R. Neches (Eds.), *Production system models of learning and development*. Cambridge, MA: MIT Press.

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

Miller, G. A., Galanter, E., & Pribram, K. A. (1960). *Plans and the structure of behavior*. New York: Holt, Rhinehart, & Winston.

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

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

Newell, A., Shaw, J. C., & Simon, H. A. (1960). Report on a general problem-solving program for a computer. *Proceedings of the International Conference on Information Processing* (pp. 256–264). UNESCO House, France: UNESCO.

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

Shell, P., & Carbonell, J. G. (1989). Towards a general framework for composing disjunctive and iterative macro-operators. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 596–602). Detroit, MI: Morgan Kaufmann.

Shivashankar, V., Kuter, U., Nau, D., & Alford, R. (2012). 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.

Slagle, J. R. (1963). A heuristic program that solves symbolic integration problems in freshman calculus. *Journal of the ACM*, *10*, 507–520.

Veloso, M. M., & Carbonell, J. G. (1993). Derivational analogy in PRODIGY: Automating case acquisition, storage, and utilization. *Machine Learning*, *10*, 249–278.