A Unified Framework for Knowledge-Lean and Knowledge-Rich Planning

Son Thanh To STO@CS.NMSU.EDU
Pat Langley PATRICK.W.LANGLEY@GMAIL.COM
Institute for the Study of Learning and Expertise, 2164 Staunton Court, Palo Alto, CA 94306 USA

Dongkyu Choi DONGKYUC@KU.EDU
Department of Aerospace Engineering, University of Kansas, Lawrence, KS 66045 USA

Abstract

The AI planning literature divides into two main paradigms, one focused on knowledge-lean methods that use only domain operators and another focused on knowledge-rich techniques that utilize hierarchical task networks or similar structures. In this paper, we present a unified framework that supports both approaches to planning, along with an implemented system that embodies its tenets. We demonstrate, on a variety of domains, that the system can handle problems stated in terms of goals, tasks, or their combination, and that it can generate plans using only operators, using hierarchical methods that decompose the problems entirely, or using an incomplete set of hierarchical methods that must be combined during planning. We report experiments that compare the efficiency of the system under these different conditions and the role of method filtering in making search tractable. In closing, we discuss other work on unifying the two paradigms and propose directions for future research.

1. Introduction

The ability to generate novel plans is one of the distinctive characteristics of human cognition. However, planning is hardly a uniform process, with people exhibiting great diversity in their behavior. This can range from extensive search on unfamiliar tasks for which they have little knowledge (Newell & Simon, 1972) to highly routine activity on problems for which they have extensive domain expertise (Larkin et al., 1980). Moreover, humans can switch between these modes of operation as the need arises, and they can even interleave them as appropriate.

The distinction between knowledge-lean and knowledge-rich approaches is reflected by two major paradigms in the AI planning literature. The first, sometimes called *first-principles planning*, relies on techniques that, starting only from domain operators, carry out extensive search to generate plans that achieve goals. This includes both older techniques that chain backward from the goal description and more recent ones that chain forward from the current state. The other main paradigm uses knowledge about task decompositions to constrain or even eliminate search. Work in this tradition typically relies on *hierarchical task networks* to guide the top-down construction of plans.

Each paradigm reflects an important facet of human planning, but, with only a few exceptions, traditional research papers have focused on one approach to the exclusion of the other. The field would benefit from a unified framework that handles both approaches as special cases. In addition to its theoretical value, such a unification would be useful in situations where some knowledge about task decompositions is available, but where this is not enough to solve problems by itself. We would like more flexible planning systems that exhibit humans' ability to combine domain knowledge with heuristic search as needed.

In this paper, we present a unified framework that supports both approaches to planning and that covers a superset of their problems, along with an implemented system that instantiates its assumptions. We demonstrate, on six domains, that the system can solve problems in which goals must be achieved, in which some task must be carried out, and in which both must be accomplished. We also show that it can generate plans with only operators, with complete hierarchical knowledge, and with an incomplete set of hierarchical methods. Using partial knowledge does not always improve search due to an increase in the branching factor, but we introduce a new technique that filters out irrelevant structures to mitigate this undesirable effect.

In the next section, we review the assumptions of these two approaches in more detail, including their strengths and weaknesses. After this, we provide a formal treatment of our new framework's structures and its processes for plan generation. We also describe UPS, an implemented system that incorporates and elaborates on the framework's central ideas. Then we make three claims about the system's planning behavior and report some experimental studies that support them. We conclude by reviewing other approaches to unifying the two paradigms, arguing that our framework makes a distinctive contribution, and discussing plans for future research in the area.

2. Two Planning Paradigms

Before we present our new framework, we should first review the two paradigms for planning that it brings together. We start by characterizing research on knowledge-lean approaches and then turn to knowledge-rich techniques. For the sake of simplicity, we restrict ourselves in this paper to 'classical' planning tasks in which the initial state is completely known and operators produce deterministic effects, assumptions that also hold for most studies of human problem solving. We also ignore other approaches to planning, such as case-based an SAT-based methods, which raise entirely different sets of issues.

2.1 Knowledge-Lean Planning via State-Space Search

The knowledge-lean paradigm defines a planning problem P as a tuple $\langle F, I, O, G \rangle$, where F is a set of propositions that describe the state of the world, I is the initial state, O is a set of operator instances that specify how to change the state, and G is a formula over F that describes the goal. Each operator $o \in O$ has an associated condition $cond(o)$ stated as a set of literals, a set of positive effects $add(o) \subseteq F$, and set of negative effects $del(o) \subseteq F$.[1] An operator instance o is applicable in a state s if $cond(o)$ is true in s, and its application in s produces a new state defined as $f(o, s) =$

1. A literal is either a proposition $p \in F$ or its negation $\neg p$. There are several languages for classical planning, including STRIPS, ADL, and PDDL. Here we adopt an extension of the STRIPS formalism.

$s \setminus del(o) \cup add(o)$. A solution to the problem P is a sequence of operators in O such that their application starting from the initial state I results in a state that satisfies the goal description G.

Researchers have developed a variety of approaches to such classical planning tasks. For some time, methods that chained backward from the goal (Kambhampati, 1997) dominated the field, but techniques that chain forward from the current state (Hoffmann, 2001) have been more popular for over a decade. One can view the latter methods as carrying out search through a tree in which nodes denote states, the start node corresponds to the initial state, and edges denote operator instances that transform states. A forward-chaining planner explores this space by repeatedly selecting a node, finding operators whose conditions match it, and using their effects to generate successor states. This process continues until the planner finds a state that matches the goal description.

State-space planning must address problems in which the number of states grows exponentially with path length. One way to mitigate this growth is the check for repeated states and ensure they are expanded only once. Another is to use heuristics, often stated as numeric evaluation functions, that rank candidate states or the operator instances that generate them, say in terms of their estimated distance to the goal. Many state-space systems combine such heuristics with a form of best-first search, which on each cycle selects the unexpanded state with the highest score. Heuristic guidance does not typically eliminate search, but it can reduce the effective branching factor enough to make the process tractable. Although not generally acknowledged, knowledge-lean planning methods have their origins in computational models of human problem solving, owing many of their basic assumptions to Newell, Shaw, and Simon's (1958, 1960) early work in the area.

2.2 Hierarchical Planning via And/Or Search

There exist a number of approaches to knowledge-rich planning, but we will focus here on the dominant one, which utilizes hierarchical task networks (HTNs). As in knowledge-lean planning, this paradigm defines a state as a set of propositions and each operator describes the deterministic effects of an operator under given state conditions. However, a hierarchical task network also includes a set of methods, M, each of which specifies how to decompose a task into subtasks. Each method comprises a head, which denotes a task, a condition like that for an operator, and an ordered set of subtasks, each of which can be either compound or primitive. A compound task can be decomposed further, whereas a primitive task is associated with an operator having the same name.

The standard objective of an HTN planner is not to achieve a state that satsifies a goal description, but rather to carry out a high-level 'task'. A problem P is a tuple $\langle F, I, O, M, T \rangle$, where the state propositions F, the the initial state I, and the operator instances O are the same as for knowledge-lean planning. The difference lies with M, a set of hierarchical method instances that achieve tasks, and T, a top-level task that consists of a predicate and its arguments. A solution to the problem P is a sequence of operator instances in O that can be applied legally starting from the initial state I and that can be generated from T by expanding methods and submethods in M. The resulting plan includes not only the sequence of states and operators, but also the hierarchical decomposition tree that generated them.

Most HTN planners operate by decomposing tasks recursively into smaller subtasks using a form of And/Or search. For a given task, the planner chooses a method M whose conditions are satisfied and decomposes it into the subtasks that M specifies. The system repeats this process for

each subtask until it reaches primitive tasks associated with domain operators. The planner applies these to produce successor states, which it uses to determine whether methods considered later are applicable. Thus, it carries out a top-down, left-to-right And/Or search, invoking backtracking as needed, for an applicable sequence of operator instances, which it returns as the solution. Some HTN planners, such as SHOP (Nau, Cao, Lotem, & Muñoz-Avila, 2001), also use axioms to draw inferences about relations mentioned in conditions. Typically, such approaches require such little search that they neither check for duplicate states or use heuristics to guide choice. Moreover, they do not explicitly state goal information as part of the problem statement.

2.3 Tradeoffs Between the Paradigms

Both knowledge-lean and knowledge-rich approaches to planning have strengths and weaknesses. The former requires reasonably little knowledge engineering, as the developer need only specify representations for states, goals, and operators. However, the price is that, in many cases, the planner must carry out extensive state-space search that, even with heuristic guidance, does not scale well with the number of objects, the branching factor, or the solution length. Efficiency improvements in this paradigm have been due primarily to alterations in the search algorithms, optimizations in the component operations that support them, and better heuristics that guide their choices, which do not alter their underlying character.

In contrast, HTN planners are highly efficient and scale well with increases in problem complexity, sometimes reducing processing time exponentially over knowledge-lean techniques (Gupta & Nau, 1992; Slaney & Thibaux, 2001). However, this characteristic depends centrally on the availability of knowledge, stated in terms of hierarchical methods, for decomposing tasks into simpler ones. For most domains, this means the developer must carry out manual knowledge engineering, which is time consuming and which may introduce errors. Moreover, HTN planners assume this knowledge base is accurate and complete, which means they may not find legal plans when methods are missing or they may find incorrect plans when methods are inaccurate.

These observations suggest the desirability of a framework that unifies the two paradigms. This should take advantage of hierarchical knowledge when available to find plans efficiently, but it should also be able to fall back on state-space search when such knowledge is absent or when some methods have been omitted. The framework should also support both the goal-oriented planning of state-space approaches and the task-oriented planning of HTN approaches. Such a theory would account for both the knowledge-lean and knowledge-rich planning abilities observed in people, although our aim here is not to reproduce the details of human cognition.

3. A Unified Planning Framework

In response to these observations, we have developed a unified framework that incorporates elements from both the knowledge-lean and knowledge-rich approaches to planning. In this section, we describe the framework's representational structures, which are a superset of those assumed by the traditional approaches. Next we turn to its mechanisms, which we will can see operate over HTN methods when they are available but can resort to primitive operators when they are not. We argue that this technique is sound and complete, after which we report UPS, an implemented planning system that incorporates these ideas.

3.1 Representational Assumptions

Our framework's representation for states, problems, and knowledge subsumes those for knowledge-lean and HTN planners. A domain specification includes a set of propositions that describe states of the environment, operators for altering these propositions, and an optional set of hierarchical methods.[2] These methods may also incorporate a set of effects analogous to those in operators, although they may only partially specify the changes that result from application (e.g., in the case of recursive methods). The framework's specification of problems is broader than that for either state-space and HTN planning, in that it may include a goal description to be achieved, a task to be carried out, or both types of elements.

Formally, a problem P is a tuple $\langle F, I, O, M, G, T \rangle$, where F is a set of propositions, I is the initial state, O is a set of operators, M is a set of method instances, G is a formula over F that describes the goal, and T is a task. Each operator o has a condition $cond(o)$ that is a formula over F, an add set $add(o)$, and a delete set $del(o)$. Each method instance m in M has a condition $cond(m)$, an add set $add(m)$, a delete set $del(m)$, and an ordered set of subtasks $sub(t)$ that are associated with either operator instances in O or method instances in M. For example, the task $(travel\_by\_plane\ A\ C)$ might be associated with two method instances, one with subtasks $(book\_direct\ A\ C)(fly\ A\ C)$ and another with subtasks $(book\_via\ A\ B\ C)(fly\ A\ B)(fly\ B\ C)$.

If the task T is nonempty, then a solution to the problem is a sequence of operators associated with the primitive tasks obtained from a successful decomposition of T, with the resulting state satisfying the goal G. When no goal is given, then any state satisfies G, so we have a standard HTN planning problem. In contrast, when T is empty, then a sequence of operators is a solution if the resulting state satisfies the goal G, so we have a state-space planning problem. Table 1 presents methods and operators for a simple domain that involves travel planning, along with a problem that specifies a goal description but no task.

3.2 Mechanisms for Generating Plans

We can now describe the processes that operate over these structures. The top-level mechanism carries out a form of forward-chaining state-space search that considers both HTN methods and primitive operators. Applying an operator instance involves generating a successor state based on the operator's specification. When applying an instance of a nonprimitive method, one uses And/Or search in an attempt to decompose the instance into a legal suplan. If this effort succeeds, the subplan's final state becomes the successor; if not, then one abandons the method instance. Each node in the search tree stores a sequence of operators that lead to it from the initial state, which has an associated empty sequence.

Figure 1 shows how the planning mechanism finds a solution to the problem in Table 1. Search starts from node 0, in which the agent is at place A, its car is at place C, and other propositions are static, in that no operator can change them. Applying operators and methods to this state produces four new states — 1, 2, 3, and 4 – that are placed in a queue ordered by their heuristic scores. Nodes 1 and 2 result from applicable operator instances — $(book\_via\ A\ B\ C)$ and $(book\_direct\ A\ B)$ — but the transitions to nodes 3 and 4 involve method instances — $(travel\_by\_plane\ A\ B)$ and $(travel\_by\_plane\ A\ C)$. To produce the last two nodes, the system must decompose their nonprim-

2. The framework also supports axioms and constraints, but we do not discuss them here for the sake of simplicity.

Table 1. (a) Domain knowledge for travel planning, including two HTN methods and four operators. Note that the same task can appear as the head of multiple methods, each of which specifies different ways to carry it out. (b) A travel planning problem that comprises an initial state and a goal description.

(a)	(travel_by_plane ?x ?y)	
	conditions	(at ?x) (place ?y) (not (= ?x ?y))
	effects	(at ?y)
	subtasks	(book_direct ?x ?y) (fly ?x ?y)
	(travel_by_plane ?x ?y)	
	conditions	(at ?x) (place ?y) (not (= ?x ?y))
	effects	(at ?y)
	subtasks	(book_via ?x ?z ?y) (fly ?x ?z) (fly ?z ?y)
	(book_direct ?x ?y)	
	conditions	(at ?x) (direct_flight ?x ?y) (not (flight_ready ?x ?y))
	effects	(flight_ready ?x ?y)
	(book_via ?x ?z ?y)	
	conditions	(at ?x) (direct_flight ?x ?z)
		(direct_flight ?z ?y) (not (flight_ready ?x ?y))
	effects	(flight_ready ?x ?z) (flight_ready ?z ?y)
	(fly ?x ?y)	
	conditions	(at ?x) (flight_ready ?x ?y)
	effects	(at ?y) (not (at ?x)) (not (flight_ready ?x ?y))
	(drive ?x ?y)	
	conditions	(at ?x) (have_car_at ?x) (place ?y) (not (= ?x ?y))
	effects	(at ?y) (not (at ?x)) (have_car_at ?y)
		(not (have_car_at ?x))
(b)	initial state	(at A) (have_car_at C) (place A) (place B)
		(direct_flight A B) (direct_flight B A) (place C)
		(direct_flight B C) (direct_flight C B) (place D)
	goal description	(at D)

itive methods into a sequence of operator instances. If this decomposition fails, then the planner abandons the associated transition. If it succeeds, the nonprimitive method instance is replaced by the unrolled sequence of operators and associated with the transition. In Figure 1, both nonprimitive methods are decomposed successfully, so a sequence of operator instances is stored in memory.

Suppose that, among the 'open' nodes, State 4 has the best score. This leads the planner to select it for expansion. There are two transitions from this state, one involving the operator instance $(book\_direct\ C\ D)$, which generates State 5, and the other involving $(drive\ C\ D)$, which produces State 6. The latter satisfies the goal description, so the planner halts its search and returns the operator sequence associated with this node, which it obtains by appending $(drive\ C\ D)$, which produced State 6, to the subplan already associated with State 4. This results in the plan $(book\_via\ A\ B\ C)(fly\ A\ B)(fly\ B\ C)(drive\ C\ D)$.

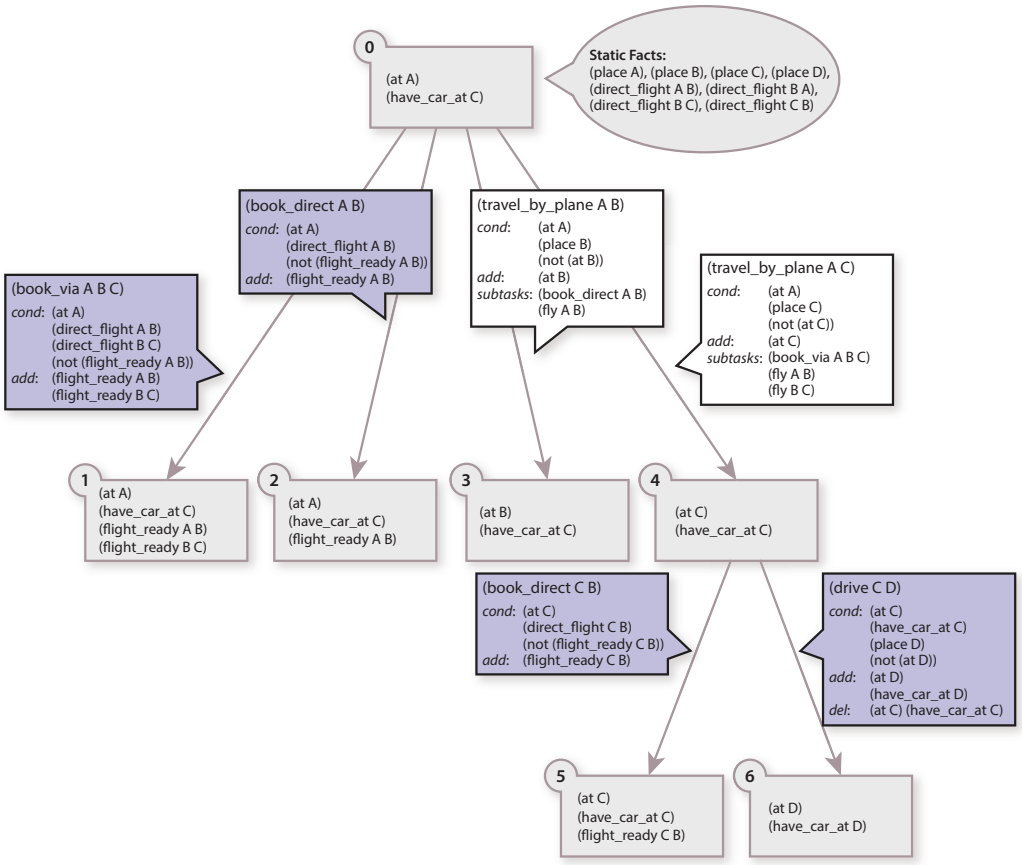

Figure 1. Illustration of the unified framework's behavior on an example that involves both hierarchical methods (unshaded) and primitive operators (shaded), and that specifies a goal but no required task.

We should also describe in more detail the And/Or search used to apply a nonprimitive method. Consider the method instance $(travel\_by\_plane\ A\ C)$ from State 0 to State 4, which has two decompositions, $(book\_direct\ A\ C)(fly\ A\ C)$ and $(book\_via\ A\ B\ C)(fly\ A\ B)(fly\ B\ C)$. The planner considers each sequence in turn to determine whether they produce legal subplans. For the first alternative, the initial operator instance, $(book\_direct\ A\ C)$, does not match in State 0, as there is no direct flight from A to C. As a result, the planner abandons this path and instead tries the second decomposition, $(book\_via\ A\ B\ C)(fly\ A\ B)(fly\ B\ C)$. The first operator, $(book\_via\ A\ B\ C)$, matches in State 0 and, on application, produces (ignoring static literals) the internal State $0.1 = \{(have\_car\_at\ C)(at\ A)\ (flight\_ready\ A\ B)(flight\_ready\ B\ C)\}$.

The next operator instance, $(fly\ A\ B)$, is applicable in this state and leads to State $0.2 = \{(at\ B)\ (have\_car\_at\ C)\ (flight\_ready\ B\ C)\}$. The final step in the decomposition, $(fly\ B\ C)$, matches in this state and, on application, generates State $0.3 = \{(at\ C)(have\_car\_at\ C)\}$. Having reached the sequence's end, the planner returns this internal state, which it renames State 4, along

with its associated subplan, $(book\_via\ A\ B\ C)(fly\ A\ B)(fly\ B\ C)$. This example assumes that fly is a primitive operator. If it were not, then the planner would attempt to decompose $(fly\ A\ B)$ before proceeding with $(fly\ B\ C)$. In other words, decomposition proceeds in order of application so that, at any step during the process, the planner has a complete description of the current state and considers a subtask only if all its left siblings have already been decomposed successfully. This ensures that each subplan returned from the decomposition process is feasible in the current state.

3.3 Soundness and Completeness

We maintain that this unified approach to plan generation is sound and complete, both of which are easy to demonstrate. For the first criterion, we note that any generated plan will, if executed, produce a state that satisfies the goal description. This follows because decomposition of a task proceeds from left to right, in the order that methods and operators would be executed, which means that the state generated after each primitive operator application is correct. Because the planner halts when it reaches a state that satisfies the goal, any sequence of states produced in this manner will also be correct. As in the two paradigms that our framework unifies, this assumes that the operators and methods are themselves accurate.

Completeness follows from the planner's reliance on a variant of forward search that guards against loops. The process starts from the initial state and keeps applying methods or operators until it reaches the goal or until there are no more untried options available, at least when sufficient computational resources are present. This ensures that, if there exists a sequence of operator instances that achieve the goal, the search process will be able to find it when given enough time. The planning mechanism's use of hierarchical methods may speed the discovery of this solution, but, even if they prove misleading, they will not keep it from eventual success, making the approach complete provided the methods contain accurate operators.

3.4 UPS: A Unified Planning System

We have instantiated the approach just described in UPS, a planning system implemented in Common LISP. The program assumes an input language similar to STRIPS in which the condition of an operator or method is a set of generalized literals. A goal description may include conjunctions, disjunctions, negation, and universal or existential quantifiers, while a task description comprises some task name and its associated arguments. Domain knowledge may also include axioms that imply some relations when conjunctions of others hold.

UPS' planning mechanism differs slightly from that presented earlier. The system does not attempt to decompose a nonprimitive transition immediately upon generation. Instead, it associates a temporary state with the method instance based on its instantiated effects. This may be incomplete, but it serves as a placeholder during decisions about which state to expand. Only when UPS selects such a state to expand does it attempt to decompose the method instance that produced it. If the decomposition process succeeds and the actual state the subplan produces has not been expanded, it replaces the temporary state with the actual state and expands it. If not, then it eliminates the transition from the search tree. This strategy avoids excessive decompositions of transitions that hold no promise for reaching a goal state.

Adding method transitions to the forward-chaining state-space search helps shorten search path to a goal state, but it also may increase the branching factor of the search considerably. For that reason, we incorporated a technique into UPS that filters HTN methods for relevance. This module returns for consideration only those method instances that produce at least one 'useful' proposition. The system defines a literal as useful if it is either a goal or one of the conditions of an operator that produces some goal. Also, if one applicable method instance produces a superset of the useful literals that another one produces, then it retains only the first candidate for expansion during search.

UPS also incorporates a numeric heuristic to guide the best-first search process. This favors states that satisfy more elements in the goal formulae. To compute a state's score, the system converts the goal formula into conjunctive normal form, that is, a conjunction of disjunctions of literals, with each disjunction corresponding to a goal element. In most cases, such an element is a single literal. The basic version of this heuristic simply counts the number of unmatched goal elements. However, when two or more states tie on this metric, UPS invokes a more sophisticated calculation. In this case, for each unsatisfied goal element g, it sums three subscores: if g is negated, the number of literals that make it false; if an axiom defines the predicate g, the number of literals in its antecedent that are unsatisfied; for each operator or method with g as an effect, the minimum of the number of literals unsatisfied in their conditions. This heuristic provides useful search guidance even when UPS only has access to primitive operators, but it is especially important when hierarchical methods are available. In such situations, it favors states produced by higher-level methods, which take larger steps through the space and reduce the effective depth of search.

The overall planning strategy differs from that used in the General Problem Solver (Newell, Shaw, & Simon, 1960) and STRIPS (Fikes, Hart, & Nilsson, 1972), which considered operator instances only if their application would achieve one of the currently unsatisfied goals. Analyses revealed that these early systems could not solve certain classes of problems, such as the Sussman anomaly. In contrast, UPS uses goal information to guide forward-chaining search, but it can still backtrack if its heuristic measure leads it down fruitless paths. In this sense, it comes closer the Jones and Langley's (2005) notion of *flexible means-ends analysis*.

4. Experimental Evaluation

Although our new framework is theoretically attractive, we must still demonstrate that it supports both knowledge-lean and knowledge-rich planning in the manner we have described. In this section, we state explicit hypotheses about UPS, the implemented system that embodies our theoretical postulates, and report experiments designed to test those claims. We propose four distinct hypotheses about the program's behavior:

- **Goal-driven and task-driven planning**. UPS generates valid plans given only goal descriptions, only task specifications, or both forms of problem statement.
- **Knowledge-lean and knowledge-rich planning**. UPS generates valid plans given only primitive operators, HTN methods sufficient to solve problems, or incomplete HTN knowledge.
- **Benefits of domain knowledge**. The availability of hierarchical methods – both complete and partial – makes the planning process more efficient.
- **Benefits of filtering in search**. Filtering hierarchical methods for relevance during their use in state-space search makes the planning process more efficient.

The first two claims relate directly to the functionality that we designed UPS to support, but they still deserve to be demonstrated on a variety of problems from a number of domains. The last two points focus instead on efficiency of the planning process. We adopt three measures for this purpose: the number of problems solved within a given time, the number of nodes expanded during search, and the run time in CPU milliseconds.[3]

To ensure generality of our results, we tested UPS on problems from six distinct planning domains of varying difficulty. The domains included the Blocks World (with problems involving from six to 20 blocks), Gripper (from ten to 30 balls and from two to 30 grippers), Logistics (from four places and four packages to six locations and ten packages), Floortile (from nine to 30 tiles and from two to three robots), Tetris (from 12 to 28 positions and from one to four squares), and Visitall (from 100 to 400 distinct locations). These domains have been studied using knowledge-lean planning, but they also lend themselves to hierarchical decomposition. For each domain, we created two HTN knowledge bases, one sufficient to solve each problem instance in a single high-level 'step' and another that required combining multiple methods or operators. The same hierarchical knowledge was available for all tasks in a domain, even though some was irrelevant to each problem.

4.1 Demonstrations of Basic Functionality

Our initial studies aimed to show that UPS implements the unified framework described earlier. The first set of runs focused on the first hypothesis, that it successfully produces valid plans for problems (if they have a solution) when given only goal descriptions, when provided with only task specifications, and when given both types of structure. For each condition, we ran UPS on ten randomly selected problems from each the six domains, providing it with a maximum of 150 CPU seconds per problem to complete its runs. In each case, the system found a solution that satisfied the goal description, that carried out the task, or that did both, as required in the problem statement. We also tested UPS against a small set of tasks that had no solution and found that it either determined no solution existed or gave up when it exceeded the time limit.

Another set of runs tested the second hypothesis – that UPS successes finds valid plans when only primitive operators are available, when it has access to a 'complete' set of HTN methods, and when it has incomplete HTN knowledge, that is, when it has available only a subset of the methods needed to solve a given problem by decomposition alone. Here we again ran the system, in each condition, on ten randomly chosen problems from each of the six domains mentioned above. In each case we provided goal descriptions rather than tasks to allow comparison with the operator-only condition. As before, we gave UPS at most 150 CPU seconds per problem to complete its runs, and it found solutions to each of these planning tasks in the allotted time.

As we will see shortly, the time needed varied widely across both different tasks and different levels of knowledge, but the system's ability to operate with and without HTN knowledge demonstrates the basic functionality for which we were aiming. Most planning times ranged from a few milliseconds to a few seconds, although a small set required more than ten seconds. We also compared UPS on Floortile, Tetris, and Visitall problems from the 2014 International Planning Competition, finding that it solves a comparable number of problems as the entrants, even without hi-

3. Although our approach to planning is complete, we make no guarantees about the optimality of generated plans, which is not an aim of either our research or most other work on cognitive systems (Langley, 2012).

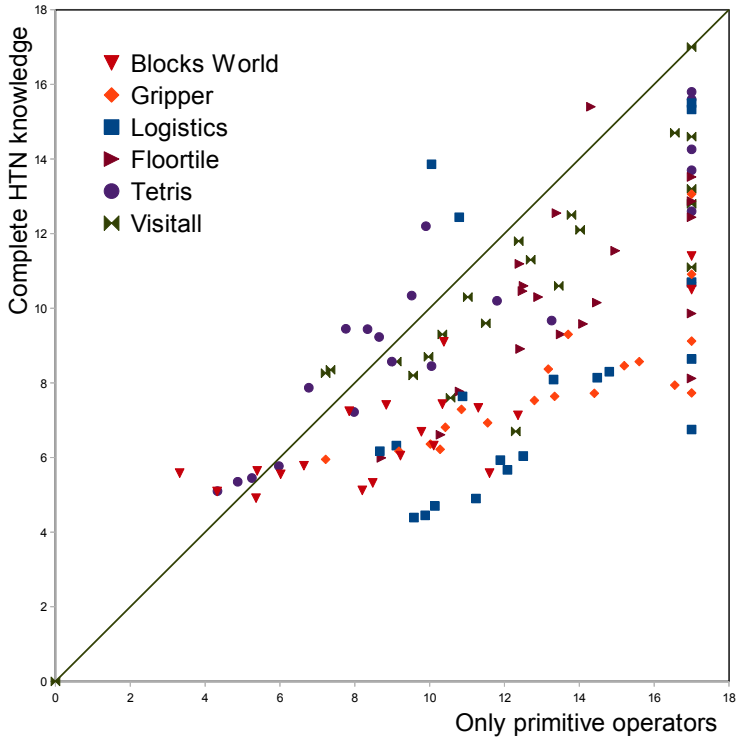

Figure 2. Comparison of UPS run times with primitive operators vs. complete HTN knowledge. Points below the diagonal denote problems on which the operator-only condition took longer, while ones at the far right exceeded the time allocated in this condition. Axes report run times as logarithms of CPU milliseconds.

erarchical knowledge, given the same time limits of 30 CPU minutes. In particular, it solved four Floortile problems, seven Tetris tasks and eight Visitall problems, contrasting with 6.46, 6.36, and 10.37 solved on average by the Competition entrants.

4.2 Benefits of Hierarchical Knowledge

We also carried out an experiment to test the hypothesis that hierarchical knowledge benefits planning efficiency. Here we ran UPS under three conditions: with only primitive operators, with HTN methods that were sufficient to solve problems in a single 'step' through decomposition, and with an incomplete set of HTN methods that could reduce the length of solution paths but not to one step. We provided UPS with 20 problems each from the six planning domains, giving a total of 120 problems of varying difficulty; these involved only goal descriptions, as task-oriented problems are ill defined for the operator-only condition. As before, we gave the system 150 CPU seconds to find a solution, in each case recording the overall run time and number of nodes expanded during search.

Figure 2 presents a scatter plot that graphs CPU time taken in the operator-only condition, for each planning task, against the time needed with 'complete' HTN knowledge.[4] As expected, UPS

4. We adopt scatter plots because they provide more detail about individual problems than tables with mean values and because they offer direct comparison between two experimental conditions on a dependent measure of interest.

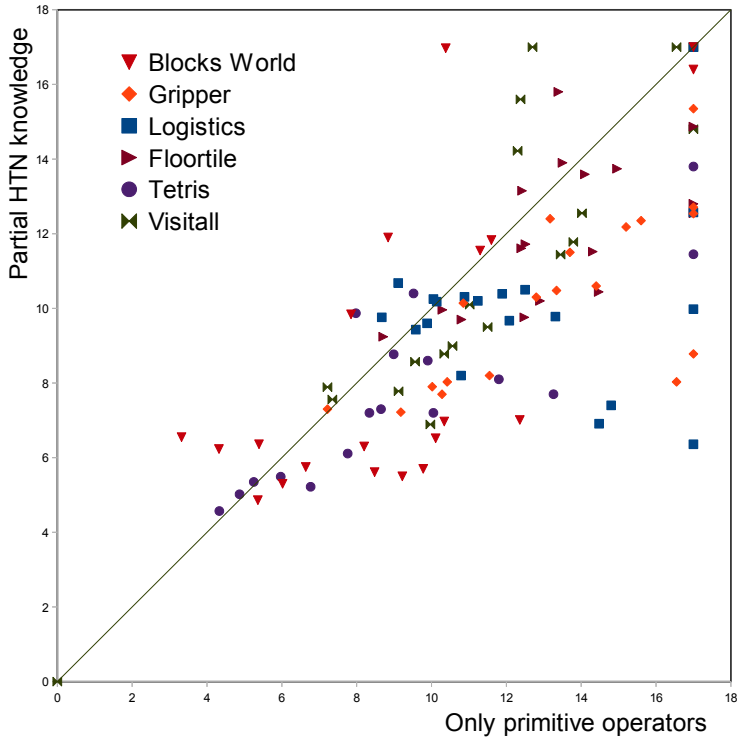

Figure 3. Comparison of UPS run times with primitive operators vs. incomplete HTN knowledge. Points below the diagonal denote problems on which the operator-only condition took longer. Axes report run times as logarithms of CPU milliseconds.

solves most tasks more rapidly in the latter condition because, in each case, it has a high-level method that achieves the goal description in one step. Similar results hold for the number of nodes expanded during search. With such HTN knowledge, the system solved all problems in the time given, while it solved only 94 out of 120 tasks without it, as indicated by points at the far right of the graph. Knowledge-lean planning was faster in a few cases, but these involved simple problems in which the overhead of examining all operators and methods exceeded the cost of what little search was needed in the operator-only condition.

Figure 3 shows a similar graph that maps time for the operator-only condition against that needed for the situation in which an incomplete set of HTN methods was available. The results here reveal that knowledge speeds planning on most of problems, but that it slows plan generation on others. In some cases, partial knowledge even led to failure within the CPU time allocated, although it still fared better overall. With incomplete knowledge, UPS solved 105 out of 120 problems, while it succeeded on only 94 with no HTN methods, as indicated by points at the top of the graph. The Gripper, Logistics, and Tetris domains benefit most from partial HTN knowledge because the heuristic scores for problem states appear to increase roughly monotonically with distance from the goal, much as some in work on macro-operators.

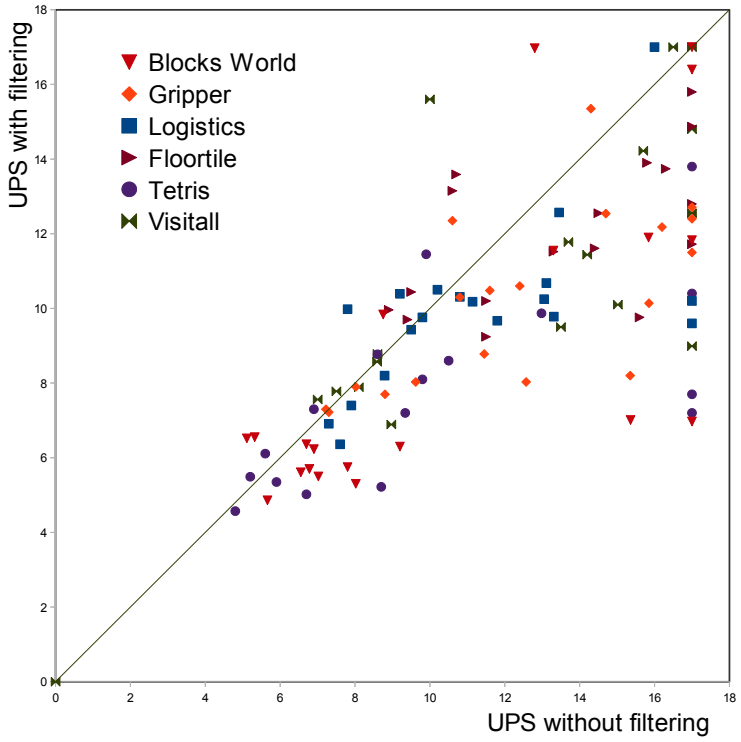

Figure 4. Comparison of UPS run times with and without filtering of methods enabled when only partial HTN knowledge was available. Points below the diagonal denote problems on which the version without filtering took longer. Axes report run times as logarithms of CPU milliseconds.

These mixed findings are reminiscent of the *utility problem* in Minton's (1988) work on learning search-control rules in PRODIGY. He found that adding knowledge indiscriminately, even though it was clearly relevant to some problems, led to longer solution times on average. However, his explanation revolved around the cost of matching control knowledge and the probability of its usefulness. Our experimental analysis instead reveals an increase, on some problems, in the number of nodes expanded during search. This suggests that the presence of hierarchical methods slows planning on some tasks because it increases the effective branching factor, even with use of the filtering technique. This behavior comes closer to that reported by Iba (1989) in his studies of macro-operators.

4.3 Benefits of Filtering / Summary

Our final experiment addressed the fourth hypothesis about the effectiveness of UPS' filtering technique, which we designed to reduce the branching factor during forward-chaining search over HTN methods. To this end, we carried out a lesion study in which we removed this module and compared planning efficiency with and without its use. As expected, we found that, when the system had access to only partial HTN knowledge, its inclusion reduced substantially both the CPU time required to find solutions and the number of nodes expanded. Figure 4 shows this effect graphically, plotting

the planning time for UPS with filtering disabled against that needed when it was utilized. Removing this technique decreased, from 105 to 89, the number of problems solved in the allotted time, as indicated by the many points near the right border. We also observed that, on most tasks, only a few method instances were selected for expansion when UPS used filtering, while it considered all method instances on each node expansion when it did not.

In summary, our experiments have shown that UPS implements our unified approach to planning in an effective manner. The system can solve planning tasks in which it must achieve goals, in which it must carry out tasks, or in which it must accomplish both; it can also operate using only primitive operators, take advantage of complete HTN knowledge, and utilize incomplete HTN knowledge. In most cases, behavior in the second condition is superior to that in the first, but access to partial HTN methods is less consistently beneficial, as it increases the branching factor during planning. The filtering technique typically reduces search and CPU time, since it helps select promising HTN methods when only partial knowledge is available. We consider these results encouraging, but we should replicate them on additional domains and understand better the utility problem that sometimes occurs with partial HTN knowledge.

5. Related Research

Our approach to planning incorporates ideas from the two traditions that we have already reviewed, but we also should discuss other work that combines techniques from knowledge-lean and knowledge-rich planning. Perhaps the best-known approach along these lines utilizes *macro-operators* that let a planner take large steps through the problem space. Fikes, Hart, and Nilsson (1972) reported an early method for learning macro-operators, while Iba (1989) described a more selective scheme for acquiring such structures. Both these and similar approaches combined primitive operators with macro-operators during planning, although the latter were limited to fixed sequences of steps, making them closer to our partial HTN condition. Shavlik's (1990) approach to recursive macro-operators comes closer to our approach, although it operated within the situation calculus. McIlraith and Fadel (2002) also use this framework to compile macro-operators with conditional effects, but they do not combine them with primitive operators during planning.

Kambhampati, Mali, and Srivastava (1998) proposed an early framework that unifies ideas from knowledge-lean and knowledge-rich planning. Their work also supported planning with only primitive operators, complete hierarchical knowledge, and partial HTN methods. However, their framework searches through a plan space, rather than the state space that ours adopts. Their scheme also favored high-level methods over primitive operators when they produce the same effects, although UPS produces this bias naturally from its goal-oriented heuristic function. Finally, Kambhampati et al. do not appear to have instantiated their theoretical framework as a running system, as we have done in our own research.

Gerevini et al. (2008) describe DUET, another implemented system that combines first-principles and HTN planning.[5] As in our framework, their approach can handle problems stated in terms of goals, tasks, or their combination, and it can generate plans using primitive operators, full HTN knowledge, or a partial set of HTN methods. A key element in UPS is the association of effects

5. Shivashankar et al.'s (2013) GoDeL system seems less relevant, in that it combines state-space planning with knowledge about goal decompositions, but not with traditional HTN methods.

with hierarchical methods; Gerevini et al. achieve a similar function by attaching effects to 'abstract actions' that correspond to nonprimitive tasks. Both systems invoke an HTN planner to ensure proposed methods expand legally and adapt a standard first-principles planner to combine steps when necessary. However, UPS utilizes forward-chaining search for the latter purpose, whereas DUET draws on LPG, a repair-based planner that carries out randomized local search, which means it is not complete. We believe our framework is more intuitive and elegant, but the two systems share many of their core ideas.

Another closely related line of research involves the ICARUS architecture (Li, Stracuzzi, & Langley, 2012), which includes a problem-solving module that can operate over both primitive skills (analogous to operators) and nonprimitive ones (analogous to HTN methods). However, it carries out a form of means-ends analysis that chains backward from goals, which contrasts with UPS' forward-chaining approach. Wilkins' (1988) SIPE system also appears to support both knowledge-lean and knowledge-rich planning, although papers on this system emphasize the latter. Two other problem-solving architectures, Soar (Laird et al., 1987; Rosenbloom et al., 1990) and PRODIGY (Carbonell et al., 1990) have this ability, but they encode domain knowledge as finer-grained control rules, not as HTN methods, which come closer to ICARUS' hierarchical skills.

6. Concluding Remarks

In this paper, we presented a unified framework for knowledge-lean and knowledge-rich planning that carries out state-space search using a combination of primitive operators and hierarchical methods. We provided an abstract statement of the framework and we described UPS, an implemented system that incorporates its main ideas, along with a goal-directed heuristic and a filtering technique to guide search. We reported empirical studies that demonstrated UPS' ability to handle problems stated in terms of goals, tasks, or their combination, as well as the capacity to generate plans from operators alone, from complete hierarchical methods, and from partial hierachies. The experiments also showed that planning with hierarchical methods is generally more rapid than with operators alone. Planning with a partial set of methods can produce a variant of the utility problem, but filtering methods for relevance can mitigate this effect. We discussed other approaches to unifying the two paradigms, but none have incorporated all the features that ours provides.

Despite the progress to date, there remain a number of avenues we should explore in future research. We should carry out additional studies that clarify when hierarchical knowledge makes UPS more efficient and when it hinders it. This could in turn suggest changes to the system that let it take advantage of knowledge about task decompositions without suffering its drawbacks. We should also extend the framework to support operators and methods that describe quantitative effects and thus are relevant to activity in physical domains. Planning over such continuous representations has been studied in the context of knowledge-lean techniques, but it has received little attention in the knowledge-rich paradigm.

Although our planning framework unifies two paradigms that have traditionally been separate, it still remains less flexible than humans, who utilize different planning strategies, such as forward and backward chaining, in different contexts. We should explore ways to encode strategic control not in the architecture but in domain-independent knowledge structures (Laird et al., 1987; Langley

et al., 2014). Ideally, the next version of UPS should not only exhibit such flexibility in its control of search, but also adapt its strategies as the need arises. Finally, we should develop methods for learning new hierarchical methods from solutions found during problem solving (Li et al., 2012), letting UPS shift automatically from knowledge-lean to knowledge-rich planning as it gains experience with a given domain.

Acknowledgements

This research was supported by Grant No. N00014-10-1-0487 from the Office of Naval Research. We thank Mike Barley, Subbarao Kambhampati, Ugur Kuter, Hector Muñoz-Avila, Dana Nau, and Michael Stilman for useful discussions about alternative approaches to planning, and we thank Charlotte Worsfold for help with finalizing the paper.

References

Carbonell, J. G., Knoblock, C. A., & Minton, S. (1990). Prodigy: An integrated architecture for planning and learning. In K. VanLehn (Ed.), *Architectures for intelligence*. Hillsdale, NJ: Lawrence Erlbaum.

Fikes, R. E., Hart, P. E., & Nilsson, N. J. (1972). Learning and executing generalized robot plans. *Artificial Intelligence*, *3*, 251–288.

Gerevini, A., Kuter, U., Nau, D., Saetti, A., & Waisbrot, N. (2008). Combining domain-independent planning and HTN planning: The Duet planner. *Proceedings of the Eighteenth European Conference on Artificial Intelligence* (pp. 573–577). Patras, Greece.

Gupta, N., & Nau, D. S. (1992). On the complexity of blocks-world planning. *Artificial Intelligence*, *56*, 223–254.

Hoffmann, J. (2001). FF: The Fast-Forward planning system. *AI Magazine*, *22*, 57–62.

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.

Kambhampati, S., Mali, A., & Srivastava, B. (1998). Hybrid planning for partially hierarchical domains. *Proceedings of the Fifteenth National Conference on Artificial Intelligence* (pp. 882–888). Madison, WI: AAAI Press.

Laird, J. E., Newell, A., & Rosenbloom, P. S. (1987). Soar: An architecture for general intelligence. *Artificial Intelligence*, *33*, 1–64.

Langley, P. (2012). The cognitive systems paradigm. *Advances in Cognitive Systems*, *1*, 3–13.

Langley, P., Pearce, C., Barley, M., & Emery, M. (2014). Bounded rationality in problem solving: Guiding search with domain-independent heuristics. *Mind and Society*, *13*, 83–95.

Larkin, J. H., McDermott, J., Simon, D. P., & Simon, H. A. (1980). Expert and novice performance in solving physics problems. *Science*, *208*, 1335–1342.

Li, N., Stracuzzi, D. J., & Langley, P. (2012). Improving acquisition of teleoreactive logic programs through representation extension. *Advances in Cognitive Systems*, *1*, 109–126.

Minton, S. (1988). Quantitative results concerning the utility of explanation-based learning. *Proceedings of the Fifth National Conference on Artificial Intelligence* (pp. 564–569). Philadelphia: AAAI Press.

McIlraith, S. A., & Fadel, R. (2002). Planning with complex actions. *Proceedings of the Ninth International Workshop on Non-Monotonic Reasoning* (pp. 356–364). Toulouse, France.

Nau, D. S., Cao, Y., Lotem, A., & Muñoz-Avila, A. (2001). The SHOP planning system. *AI Magazine*, *22*, 91–94.

Newell, A., Shaw, J. C., & Simon, H. A. (1958). Elements of a theory of human problem solving. *Psychological Review*, *65*, 151–166.

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, Paris.

Newell, A., & Simon, H. A. (1972). *Human problem solving* Englewood Cliffs, NJ: Prentice-Hall.

Rosenbloom, P. S. Lee, S., & Unruh, A. (1990). Responding to impasses in memory-driven behavior: A framework for planning. *Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control* (pp. 109–114). San Diego: Morgan Kaufmann.

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

Shivashankar, V., Alford, R., Kuter, U., & Nau, D. (2013). The GoDeL planning system: A more perfect union of domain-independent planning and hierarchical planning. *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence* (pp. 2380–2386). Beijing: AAAI Press.

Slaney, J., & Thiebaux, S. (2001). Blocks world revisited. *Artificial Intelligence*, *125*, 119–153.

Wilkins, D. (1988). *Practical planning: Extending the classical AI planning paradigm*. San Mateo, CA: Morgan Kaufmann.