Advances in Cognitive Systems (2018) 251–270 Submitted 6/2018; published 12/2018

# Adaptive Search in a Hierarchical Problem-Solving Architecture

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

**Mike Barley** MBAR098@CS.AUCKLAND.AC.NZ
**Ben Meadows** BMEA011@AUCKLANDUNI.AC.NZ
Department of Computer Science, University of Auckland, Private Bag 92019, Auckland 1142 NZ

**Abstract**

The ability to solve novel problems is a distinguishing feature of human intelligence. This capacity has been replicated in both cognitive architectures and AI planning systems, but previous work has ignored its adaptive character. In this paper, we review HPS, a cognitive architecture that searches a space of hierarchical problem decompositions with parameters that support a variety of strategies. Moreover, decisions made by these strategic parameters may be conditioned on information available during search, such as the depth, branching factor, and progress toward the goal description. We examine three such parameters, one that decides whether to chain forward or backward when retrieving operator instances, one that determines how far to backtrack upon failure, and another that decides how deep to search before backtracking. In each case, we describe adaptive methods for making these decisions and report experiments which compare their performance with that for fixed strategies. In closing, we recount prior research on adaptive problem solving and propose some directions for future work in this understudied area.

## 1. Introduction and Motivation

The AI literature on problem solving and planning is littered with different techniques. For instance, early planning research emphasized methods that chained backward from goals, whereas current systems rely mainly on forward chaining. Best-first search is a widely adopted control scheme, but beam search and greedy techniques have also seen frequent use. Typically, authors champion one approach over others, but they seldom mention the obvious – that different methods may be preferable in different situations. There have been some publications on such tradeoffs (e.g., Langley, 1992), but studies of this sort are few and far between. Still, it seems likely that the relative effectiveness of strategies will vary and that problem solvers can benefit from the ability to adapt.

The psychological literature reports numerous examples of strategy shifts, but these focus on domain-level behavior (Siegler, 1989; Larkin et al., 1980). Simon and Reed (1976) observed strategy changes on Missionaries and Cannibals, but otherwise there appears to be no formal evidence for shifts in problem-solving styles on unfamiliar tasks, where domain knowledge cannot be used. However, it seems clear that novices use strategies like means-ends analysis on the Tower of Hanoi (Anderson, 1993) and ones like forward chaining in chess (de Groot, 1978). Similarly, people are quite capable of systematic search with backtracking on simple problems like Tic-Tac-Toe but resort to methods that sample action sequences, such as progressive deepening (de Groot, 1978), on more challenging games. We would like a computational account of such adaptation in problem solving, although our aim is to demonstrate the same types of flexibility observed in humans rather than fitting the findings from particular studies, which are rare in any case.

In this paper, we extend an existing architecture, HPS, to adapt its search strategies in response to the characteristics of problems it attempts to solve. We begin by reviewing HPS's hierarchical representation of problem solutions, the organization of its search tree, the architecture's basic decision cycle, and the role of strategic parameters in guiding behavior. We then focus in turn on three of these parameters. The first determines whether to retrieve candidate operators by chaining forward from the state or backward from goals, the second controls backtracking by selecting the node to visit when the current one is abandoned, and the third determines when such abandonment occurs. In each case, we describe an adaptive method that invokes a different strategy depending on problem features, along with experiments on its effectiveness during search. We conclude by discussing related work on adaptive problem solving and proposing ideas for additional research.

## 2. Review of the HPS Architecture

We have explored our ideas about adaptation within HPS, an architecture for *h*ierarchical *p*roblem-*s*olving (Langley et al. 2016). In this section we review the system's representation of candidate solutions, its organization of search trees, its cognitive cycle, and its parametric encoding for strategies. The latter lets the framework reproduce many distinct search methods, much as Soar (Laird et al., 1987), although the approach to specifying them is quite different. We must clarify how HPS operates before we can explain the ways in which we have incorporated adaptation.[1]

### 2.1 Problem Decompositions and Search Trees

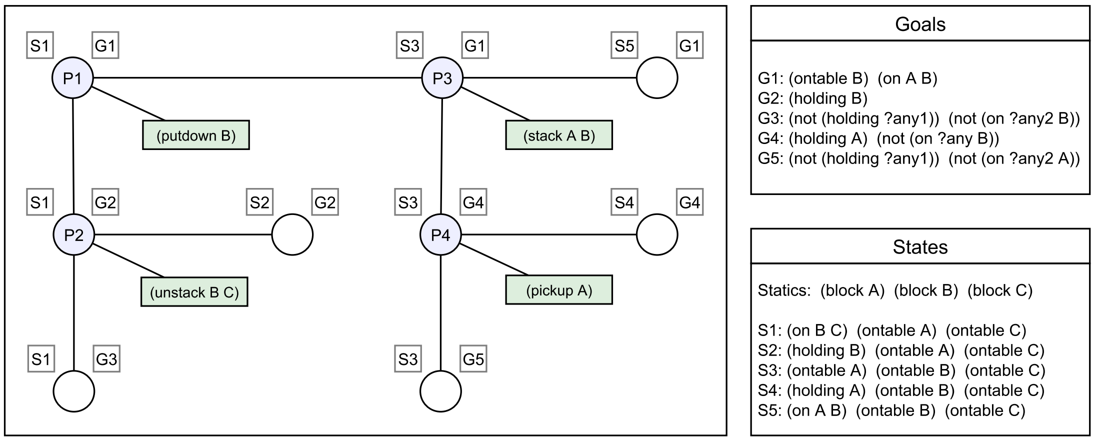
The HPS architecture encodes each candidate solution as a tree that decomposes a problem recursively into subproblems. Each element in such an AND tree has two main components:

- A *problem*, which includes a state and goal description. The former uses literals like *(on A B)* to specify a situation; the latter uses literals like *(not (on ?any A))* to denote a set of desired states.
- A *decomposition*, which breaks the problem into three distinct, ordered elements:
    - An *operator instance* *O* with a set of conditions, effects, and constraints on shared variables;
    - A *down subproblem* that one must solve before applying operator *O* to satisfy its conditions;
    - A *right subproblem* that one must solve after *O*'s application to achieve the parent's goals.

We refer to 'down' subproblems and 'right' subproblems because these terms reflect our graphical notation for problem decompositions, which we will examine shortly.

Such a decomposition tree has a hierarchical structure, so that each subproblem may be broken down further. A given tree maps onto a unique sequential plan, although a particular plan may be decomposed in different ways. Not all decomposition trees solve the top-level problem fully, so they are best viewed as partial plans. Special cases include a top-level problem statement, which

1. We will actually describe HPS/2, which differs slightly from its predecessor in its cognitive cycle and strategic parameters, but which still shares many of the earlier system's assumptions.



Goals

G1: (ontable B) (on A B)
G2: (holding B)
G3: (not (holding ?any1)) (not (on ?any2 B))
G4: (holding A) (not (on ?any B))
G5: (not (holding ?any1)) (not (on ?any2 A))

States

Statics: (block A) (block B) (block C)

S1: (on B C) (ontable A) (ontable C)
S2: (holding B) (ontable A) (ontable C)
S3: (ontable A) (ontable B) (ontable C)
S4: (holding A) (ontable B) (ontable C)
S5: (on A B) (ontable B) (ontable C)

*Figure 1.* A solution for problem P1 that decomposes it into four nontrivial subproblems (shaded circles) with associated operators (rectangles). Each problem node has an associated state (on its left) and goal description (on its right), whose labels refer to structures in the rightmost boxes. White circles denote trivial subproblems in which the goal description matches the state. The corresponding sequential plan has four steps: *(unstack B C)*, *(putdown B)*, *(pickup A)*, and *(stack A B)*.

has no operator or subproblems, and complete plans, which specify a sequence of operators that transform the initial state into one that meets the top-level goals.

Figure 1 presents a successful decomposition of a Blocks World task that involves a top-level problem (P1), three nontrivial subproblems (P2, P3, and P4), their selected operators, and the associated state and goal descriptions. P2 and P4 are 'down' subproblems that must be solved before applying the operators *(putdown B)* and *(stack A B)*, respectively, whereas P3 is a 'right' subproblem that remains after *(putdown B)* has been applied. This hierarchical decomposition maps onto the sequential plan *[(unstack B C), (putdown B), (pickup A), (stack A B)]*. The figure also shows five unlabeled terminal nodes; these are trivial subproblems in that their associated states satisfy their goal descriptions, and thus require no further decomposition.

This sample decomposition is similar to those produced by means-ends analysis (Jones & Langley, 2005; Newell et al. 1960), but HPS is not limited to such structures. In particular, it can also encode fully right-branching trees that reflect solutions found through forward chaining and fully down-branching trees that would result from regression planning (McDermott, 1991). Different problem-solving strategies produce different types of decompositions, which will be important for some forms of adaptation. The example solution is also similar to plans generated from hierarchical task networks (Nau et al., 2003), but HPS decompositions have a more constrained structure that involves an operator, a down subproblem, and a right subproblem, and, as we will see shortly, they do not depend on the use of domain-specific knowledge for their construction.

Naturally, HPS must search through a space of such hierarchical plans and it must somehow organize these alternatives. The system stores candidates in an OR tree like that in Figure 2. Each node denotes a partial or complete decomposition of the top-level problem into subproblems, with every child introducing one new operator beyond those in its parent. For instance, the root *N1*

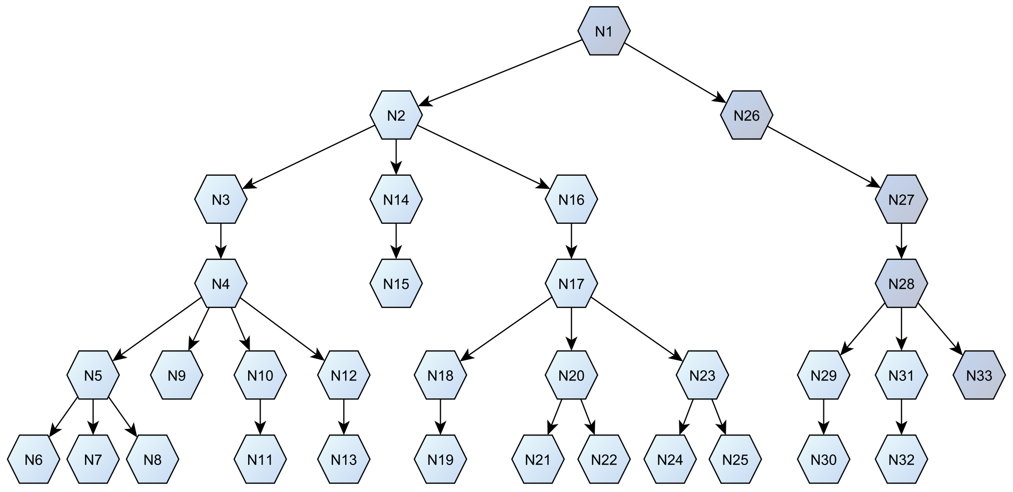


*Figure 2.* An HPS search tree produced by a combination of means-ends analysis and depth-first search with a depth limit of five. Numbers reflect the order in which nodes were generated and *N33* refers to the hierarchical solution in Figure 1. Each node expands on its parent by introducing a new operator instance and associated subproblems. Terminal nodes *N9* and *N15* denote partial plans that HPS rejected due to goal loops.

includes the top-level problem *P1* from Figure 1, *N26* introduces the operator *(putdown B)* and adds the down subproblem *P2*, while *N27* adds the operator *(unstack B C)* and the subproblem *P3*. Node *N28* elaborates its parent by introducing *(stack A B)* and, because its conditions are not met, subproblem *P4*. The lowest node on the shaded path, *N33*, adds the operator *(pickup A)*, which enables *(stack A B)* and solves both *P3* and *P1*. This node corresponds to the the four-step hierarchical plan shown in Figure 1. Alternative children specify different elaborations on their parent's partial solution. The numbers on nodes reflects their order of creation, which clarifies that HPS generates children one at a time, rather than 'expanding' a node to produce all children at once.

## 2.2 Problem Solving in HPS

The structures just described are central to HPS's problem-solving process, which involves search through a space of alternative problem decompositions in an effort to transform the initial state into one that satisfies the goal description. At the outset, the root node contains a single problem with only state and goal descriptions. The architecture decomposes this problem recursively into subproblems in different ways, adding new nodes to its search tree, with each child elaborating its parent by introducing one operator instance. This process continues until it finds enough solutions to satisfy its success criterion or until it abandons the effort.

A key notion here is the *focus* problem, which is a task or subtask in the hierarchical decomposition that has not yet been solved. This may be a down subproblem *D* or a right subproblem *R* of a problem *P*, but the latter can become the focus only after HPS has solved the former. A child in the search tree always extends its parent by decomposing the focus subproblem. This may introduce a new subproblem that becomes the focus in the child. When the system solves a given focus problem, it pops back to its parent $P$ and either shifts to $P$'s right subproblem or, if $P$ has also been solved, shifts to its parent. For instance, *P1* in Figure 1 is the focus problem of node *N1* in Figure 2, *P2* is the focus of node *N26*, *P3* is the focus of *N27*, and *P4* is the focus of *N28*. All problems and subproblems in the final node, *N33*, are solved, which returns the focus to *P1*.

The problem-solving architecture operates in discrete cycles, each of which applies one of nine meta-level rules that advance the search process:

1. If there are *enough problem solutions*, no options remain, or resources are exhausted, then halt and return the solutions that have been found.
2. If the top-level problem for current node $N$ is *solved* (e.g., satisfies all the goals), then add $N$ to the solved list and *select* another node in the search tree.
3. If current node $N$ is *unacceptable* (e.g., it is too deep or has a loop), has *enough children*, or has *enough failed retrievals*, then add it to the closed list and *select* a new node in the search tree.
4. If focus problem $F$ has no associated operator, then attempt to retrieve one by *filtering* candidates, *evaluating* any that remain, and *selecting* an operator $O$ from this set.
5. If retrieval has produced an operator $O$ for the focus problem $F$ of node $N$, then create a child $C$ of $N$ with focus $F$ and operator $O$, *assign a score* to $C$, and *select* a new node (possibly $C$).
6. If focus problem $F$ has an unapplied operator $O$ whose conditions match the current state $S$, then apply $O$ to $S$ to generate a new current state for $F$.
7. If focus problem $F$ has an unapplied operator $O$ whose conditions $C$ do not match current state $S$, then create a down subproblem $D$ with state $S$ and goals $C$, and make $D$ the focus.
8. If focus problem $F$ with goals $G$ has an operator that produced state $S$, and if $S$ does not satisfy $G$, then create a right subproblem $R$ with state $S$ and goals $G$, and make $R$ the focus.
9. If focus problem $F$ with goals $G$ has an operator that produced state $S$, and if $S$ satisfies $G$, then mark $F$ as solved and (unless it is the top-level problem) shift the focus to its parent.

This iterative procedure decides if the current node is acceptable or unacceptable, shifts to different ones when it finds a solution or encounters obstacles, retrieves operators used to generate nodes that elaborate their parents, and processes the focus problem to apply operators, create subproblems, and shift the focus. The content stored at a given node changes over the course of problem solving, as HPS applies operators and populates problems with the resulting states, but such steps do not produce new nodes in the search tree. Taken together, these activities search through a space of problem decompositions that may transform the initial state into one that satisfies the goal description.

This control framework has much in common with those adopted by means-ends architectures like GPS (Newell et al., 1960), PRODIGY (Carbonell et al., 1990), Eureka (Jones & Langley, 2005), and ICARUS (Langley et al., 2009). One important difference is that HPS does not include a decision step that selects a goal on which to focus; goals can influence operator selection, but this is not an architecture-level commitment. Thus, although HPS shares with means-ends systems the idea of search through a space of problem decompositions, the architecture can define these spaces in far more flexible ways than its intellectual precursors.[2]

2. We should mention two other systems — PRL (Marsella & Schmidt, 1993) and SteppingStone (Ruby & Kibler, 1993) — that search a space of problem decompositions, but, like HTN planners, rely on domain-specific knowledge.

*Table 1.* Descriptions of the 12 parameters that HPS uses to specify its problem-solving strategy and the meta-level control rules in which they appear. Each parameter can take on different settings from the specified defaults to produce distinct search behaviors.

| *Parameter description* | *Control rule* | *Default setting* |
|---|---|---|
| When has the search process found enough solutions? | 1 | one solution |
| When is the current node an acceptable solution? | 2 | goals satisfied |
| Which node should one select on finding a solution? | 2 | parent of node |
| When is the current node unacceptable? | 3 | loops, depth $> 10$ |
| When does the current node have enough children? | 3 | 30 children |
| When has operator retrieval at a node failed enough times? | 3 | 10 failures |
| Which node should one select on rejecting a node? | 3 | parent of node |
| How should operator retrieval filter candidates? | 4 | conditions, unselected |
| How should operator retrieval evaluate candidates? | 4 | constant |
| How should operator retrieval select among candidates? | 4 | best score |
| How should one evaluate nodes in the search tree? | 5 | constant |
| Which node should one select on computing a node's score? | 5 | current node |

## 2.3 Modulating Problem-Solving Behavior

We have characterized the search process in generic terms, without specifying how decisions are made, but naturally the architecture must select among alternatives at each choice point. To this end, HPS incorporates 12 *strategic parameters* that determine the details of its search behavior. Table 1 provides brief descriptions of these parameters, some that involve Boolean tests, others that specify how to select among nodes and operators, and still others that indicate how to score alternatives. As the table indicates, each parameter is associated with one step in the problem-solving cycle and influences decisions made at that stage. The italicized terms in the nine meta-level rules just presented also mark the loci of these parameters.

We will not review all of these decision points in detail, as they are not our emphasis in this paper. Instead, we will focus on three parameters that lend themselves to adaptation and that play central roles in experiments reported later. These choice points include:

- *Operator filtering*. When attempting to retrieve an operator instance for the current node's focus subproblem, HPS calls on a parameter that determines which candidates to consider. Two basic options include retrieving operators whose conditions match the current state and ones whose effects would achieve at least one goal. The first setting leads to forward-chaining behavior that produces right-branching problem decompositions. The second results in means-ends analysis that creates down subproblems when selected operators are inapplicable and right subproblems after they have been applied. This parameter option led to the hierarchical solution in Figure 1.
- *Node selection on failure*. When HPS decides that the current node in the search tree is unacceptable, it invokes a parameter that selects a new node and thus controls backtracking. One natural scheme is to shift attention to the current node's parent, while another is to return back to the root node. The first alternative leads to depth-first search, which produced the node ordering shown in Figure 2. The second option results in iterative sampling (Langley, 1992), a search

method that repeatedly samples the problem space using a greedy technique. The latter behavior is similar to progressive deepening, which de Groot (1978) observed in human chess play.

- *Node unacceptability*. When HPS visits a node in the search tree, it calls on a third parameter that decides whether the node is unacceptable in some way. Two straightforward options here are rejecting a node when its position in the search tree exceeds a depth limit and, in settings where operators have some cost, when the total expense is greater than some threshold. The search tree shown in Figure 2 resulted from setting this parameter to a depth limit of five, which led the system to reject nonsolution nodes *N6*, *N7*, *N8*, and others at that level.

These examples illustrate how HPS uses parameters to support a variety of fixed problem-solving strategies. The approach is similar in spirit to how parameters in the PRISM environment (Langley, 1983) let it specify many distinct production system architectures. However, there is no inherent reason why parameters must describe fixed behaviors and, as we will see shortly, they can also support ones that respond adaptively to the solver's situation.

The strategic flexibility of HPS has implications for traditional notions of soundness and completeness. The architecture never applies a selected operator until it achieves a state that satisfies its conditions, so any hierarchical decomposition that achieves the top-level goal description will be sound in that its operator sequence transforms the initial state into the target specification. This even holds for less stringent solution criteria, such as those needed for partial-satisfaction planning, which consider achieving a subset of top-level goals as success. The guarantees for completeness are much weaker. We can ensure that HPS will find all possible solutions to a given problem, at least those constrained by termination criteria like a depth limit, but only with particular parameter settings. For instance, the value for 'enough solutions' must be set to infinity, as must the parameter for 'enough children'. In addition, operator filtering must always return both all candidates whose conditions match the current state and ones with effects that would achieve one or more goals. Of course, human problem solvers seldom find all possible solutions and their answers may not always be sound, and we view HPS's ability to reproduce such behavior not as a bug but as a feature.

## 2.4 Theoretical Postulates and Implementation Details

The HPS architecture incorporates a number of linked theoretical tenets. The framework builds on two standard assumptions, both due to Newell and Simon (1976): the *physical symbol system* hypothesis – that intelligence relies on representing and manipulating organized symbol structures, and the *heuristic search* hypothesis – that it involves guided search through problem spaces. However, HPS introduces four additional postulates:

- Candidate solutions are structured as recursive decompositions of problems, each with an associated operator, down subproblem, and right subproblem.
- Such candidates are organized as nodes in a search tree where each child elaborates on its parent by introducing one operator and any subproblems that result.
- Problem solving involves search through a space of alternative decompositions that aim to transform the initial state into one that satisfies the goal description.
- Details of this search behavior, including evaluation and selection of operators and nodes, as well as criteria for success and failure, are determined by strategic parameters.

Thus, HPS retains the key ideas of means-ends problem solving without its commitment to chaining only off operators that achieve goals. The framework replaces this control scheme with parametric options, with traditional means-ends analysis being a special case. Here we add another postulate: *the search process adapts parameter settings in response to characteristics of the problem being solved*. Although each of these ideas has appeared elsewhere, HPS is the first architecture that combines them into unified theory.

We have implemented these theoretical ideas in Common Lisp. The processing cycle outlined above relies on iteration through a sequence of conditional statements, with HPS carrying out the actions associated with the first satisfied alternative. Both conditions and actions used in the cognitive cycle refer to strategic parameters, which support considerable variety within a single problem-solving architecture. Each setting for a parameter corresponds to an executable Lisp function that returns a result appropriate for later processing. These functions inspect, create, and modify information about nodes, problems, states, goals, and operators, but they are domain independent in the sense that they do not refer to domain predicates. Moreover, the parameter settings are intrinsically composable, so that different combinations of them can reproduce many distinct strategies.

Parametric functions carry out only local computations to ensure tractability. For instance, typical options for node selection return the parent of the current node or the root. The most expensive alternative, used to produce best-first search, finds the highest-scoring node on the open list. The cost of operator retrieval is linear in the branching factor for each focus problem. Some of the adaptive options reported later involve extra calculations, but these are minor and remain local. As a result, HPS avoids the substantial overhead found in early approaches to meta-level control (e.g., Genesereth et al., 1981). Like many planning systems, HPS generates a set of possible operator instances at the outset of problem solving. The implementation uses type information about entities to constrain this process and depth-limited chaining on operator's effects to eliminate impossible combinations. Grounding operators lets the system rely primarily on equality tests during operator retrieval rather than on relational pattern matching, which can be substantially more expensive.

## 3. Empirical Studies of Adaptive Problem Solving

We are interested in whether adaptive versions of HPS's parameters can enable more effective problem solving than ones with fixed methods. To this end, we implemented new settings for operator retrieval, node selection after failure, and node abandonment that condition their decisions on problem characteristics. In this section, we describe these adaptive elements and report experiments that compare their behavior to fixed strategies. However, first we must review the domains used in these studies and present our general experimental design.

Our aim was to demonstrate that the HPS framework supports interesting forms of adaptation, and that such adaptive strategies have the potential to produce more effective search than their fixed analogs. However, we will not claim that our adaptive techniques are the only ones possible or the best, and, as we discuss later, others have explored alternative approaches to adaptive search. Moreover, even negative results would be informative, as they can clarify the conditions under which adaptation is useful and when fixed strategies are preferable. Our work's main theoretical contribution lies in providing a framework that supports adaptive problem solving and identifying three decision points that have potential to benefit from such meta-level control.

## 3.1 Domains and Experimental Design

Our experiments draw on four problem-solving domains: the Blocks World, Logistics planning, inferring Kinship relations, and the Five Puzzle. The first domain involves changing one configuration of blocks that rest on a table into another configuration that satisfies a goal description. Logistics planning requires one to transport packages from initial to target locations using operators for loading and unloading trucks and airplanes, along with driving and flying these vehicles. Both are classic planning tasks that will be familiar to researchers in that subfield. In the Kinship domain, one infers relations like *uncle*, *grandparent*, and *ancestor* from primitive ones like *parent* and *male*. This involves purely monotonic inference, as the 'operators' are datalog rules that only add literals to the state. The Five Puzzle involves sliding numbered tiles from their initial arrangement to a target configuration. As in the Eight Puzzle, there are nonmonotonic operators for moving a title up, down, left, and right, but only five tiles and six cells are present.

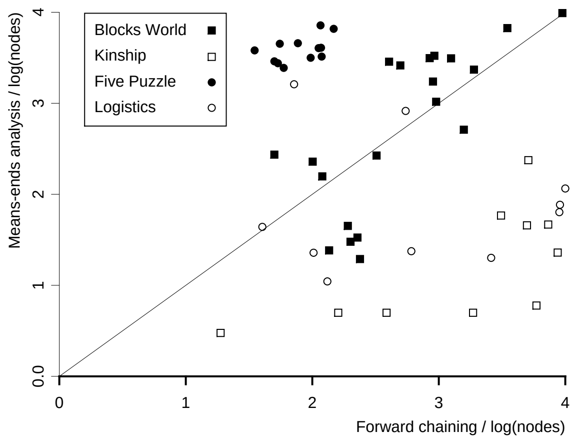
For each domain, we presented HPS with multiple tasks of varying complexity,[3] as measured by the number of steps in the minimal solution. We examined its problem-solving behavior with both fixed and adaptive strategies, making this our primary independent variable. On each run, we measured the number of nodes generated during search before finding a solution and the associated CPU time. Because node selection in HPS relies on local computation, such as finding the parent or root, we expected run time to correlate well with the node count, so we generally report only the latter. We limited the search process 10,000 nodes, recording this number when the system failed to find a solution. Unless stated otherwise, we used the default parameter settings from Table 1. In the absence of heuristic criteria, HPS carries out 'uninformed' search and selects among candidate operators at random, so we ran the system 20 times on each task and averaged the results.

## 3.2 Fixed and Adaptive Operator Retrieval

As we have noted, one of HPS's core parameters determines how the system retrieves candidate operators for use in decomposing the current focus problem. In previous work, we had implemented two fixed schemes for this aspect of decision making. The first considers only operator instances whose conditions match the current state. This produces the familiar strategy of forward-chaining search, which leads HPS to create a succession of right subproblems. The second considers only operator instances whose effects would achieve at least one currently unsatisfied goal. This gives the backward-chaining strategy that underlies traditional means-ends analysis. When the current state does not satisfy the operator's conditions, it leads HPS to introduce a down subproblem, although the system may also create a right subproblem after operator application.

Human problem solvers appear to use both retrieval methods (Anderson, 1993; de Groot, 1978), and it seemed likely that goal-driven chaining would be more effective on some tasks and state-driven chaining on others. We tested this prediction by running the two strategies on problems from the four domains. The left graph in Figure 3 presents the results of this experiment in terms of a scatter plot, with forward chaining on the *x* axis and means-ends analysis on the *y* axis. Each point denotes the average number of nodes generated for both strategies, which means that goal-driven retrieval was more effective for those below the diagonal line and state-driven retrieval fared better for

3. These included 20 Blocks World tasks with solutions of four to ten operators, ten Kinship problems needing two to eight inference steps, 12 Five Puzzle tasks with four to eight moves, and ten Logistics tasks with three to ten steps.



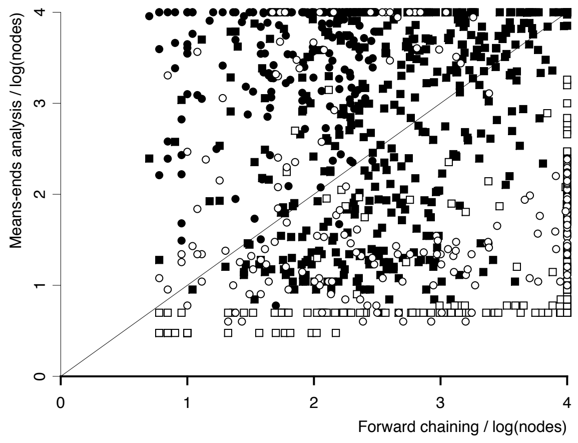


*Figure 3.* Scatter plots that compare the number of nodes generated by HPS using state-driven operator retrieval (forward chaining) and goal-driven retrieval (means-ends analysis) on problems from four domains. The left graph shows the mean number of nodes, in logarithmic scale, generated for each problem; the right graph shows the numbers for each separate run. Points below the diagonal denote problems in which goal-driven operator retrieval generated fewer nodes than did state-driven retrieval. Items at the *x* and *y* boundaries indicate problems that a method failed to solve given the nodes allocated.

those above it. The graph shows both measures in log scale to retain visibility for simpler problems. The trends are clear: neither approach is uniformly superior to the other, with the preferable method depending on the task. Two clear results were that backward chaining always did better on Kinship problems, while forward chaining was always more efficient on the Five Puzzle, although behavior on the Blocks World and Logistics tasks was less regular. Standard errors were high, but adding error boxes to the graphs would make them unreadable; the right graph in Figure 3 shows pairs of values from indivdual runs that support these conclusions despite the considerable variation.

These results suggest that an adaptive method might outperform the uniform use of either strategy in isolation. A natural criterion for deciding whether to chain off operator conditions or effects is the branching factor for each option. If goal-driven backward chaining produces fewer alternatives than does state-driven forward search, then the first scheme should reduce the search needed to find a solution and vice versa. For instance, this held for the Kinship domain, where the forward branching factor was 34 and the backward one was never greater than four. We implemented an adaptive method for operator retrieval in HPS that works along these lines. For each node in the search tree, it examines operator instances that result from chaining forward from the state and chaining backward from unsatisfied goals, then selects an operator from the smaller set, falling back on state-driven retrieval when ties occur.

This decision about how to proceed is local to the focus subproblem. If the number of choices in one direction is always greater than in the other, the system will mimic strict forward search or pure means-ends analysis. However, if the ratio of branching factors varies enough, then the strategy will alternate between forward and backward chaining, as one direction or the other becomes more constrained. In these cases, the system should examine, on average, fewer nodes than either fixed strategy. Taken together, these observations suggests a testable conjecture:

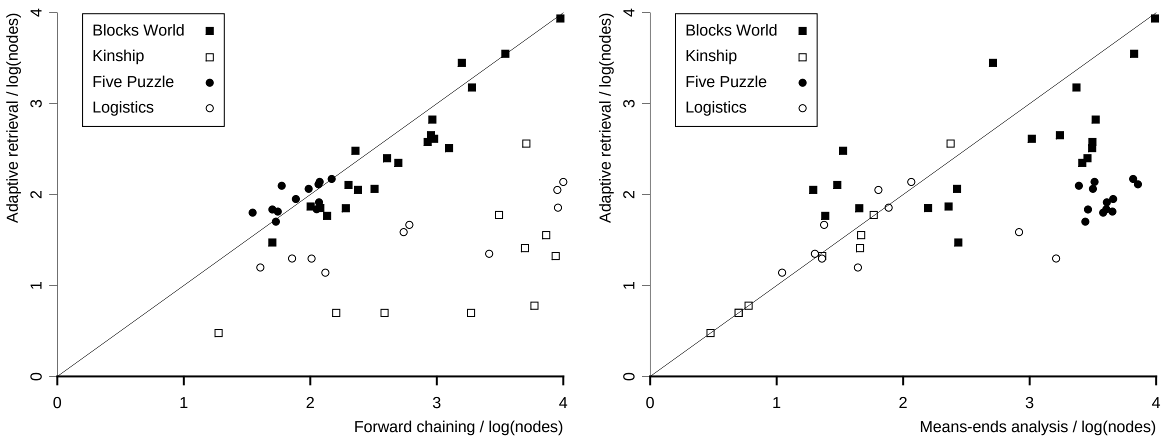


*Figure 4.* Scatter plots that compare the number of nodes in logarithmic scale generated with (left) adaptive operator retrieval vs. state-driven retrieval (forward chaining) and with (right) adaptive operator retrieval vs. goal-driven retrieval (means-ends analysis). Points below the diagonal denote problems in which adaptive retrieval created fewer nodes during the search process.

- *Hypothesis: Adaptive operator retrieval will require no more, and sometimes less, search to find a solution for a given problem than fixed state-driven or goal-driven retrieval.*

To test this prediction, we ran HPS with this adaptive method on the same problems as used in the first study. The scatter plots in Figure 4 compare the search behavior of adaptive retrieval with forward chaining (left) and means-ends analysis (right). As before, each point represents an average over 20 runs, with the value on the *x* axis denoting the nodes generated by a fixed strategy and the value on *y* axis giving the nodes for the adaptive one.

The results are quite encouraging. Adaptive operator retrieval explores about the same number of nodes as forward chaining on the Blocks World and Five Puzzle, but it takes substantially less effort on all Kinship tasks, which we know cause difficulty for forward-chaining search, and on most Logistics problems. The benefits over means-ends analysis are weaker but still clear cut. The two strategies do nearly the same on Kinship problems, and on many Blocks World and Logistics tasks, but adaptive retrieval requires less search on some of the latter and on all Five Puzzle problems. We should examine more closely the four Blocks World tasks that fall above the diagonal, but the results are generally positive and support our hypothesis. The correlation between between nodes generated and CPU time was 0.87 on 3,120 distinct runs, confirming our expectations on this front.

Note that our approach to adaptive operator retrieval differs from standard bidirectional search techniques (e.g., Lippi et al., 2012; Holte et al., 2016), which typically assume fully specified goal states and which rely on 'state-space' methods. HPS instead carries out a form of 'plan-space' search (Kambhampati, 1997), with the choice on each step being whether to elaborate a hierarchical plan by adding a down subproblem or a right subproblem. The system decides not the direction in which to search, but rather the direction in which to chain when considering operators it should use to refine a partial solution. We discuss the essential differences between bidirectional methods and our approach at greater length in the section on related research.

## 3.3 Fixed and Adaptive Backtracking

We have seen that another key parameter in HPS determines how the system backtracks when it decides the current node is unacceptable. One alternative simply makes the failed node's parent the new focus of attention, which produces traditional depth-first search. Another option instead returns to the root node and continues search from there. This leads to *iterative sampling*, a strategy that carries out repeated greedy search until it finds a solution or exhausts its resources. This method is interesting because it is similar to progressive deepening, which de Groot (1978) has observed in human chess players. Langley (1992) reported formal analyses and experimental studies that identified the situations in which iterative sampling will, on average, explore fewer nodes than depth-first search. Briefly, the latter does worse when the cost of making incorrect choices early on is high, which holds when solutions are clustered in a few regions of the search tree. The cost of errors for iterative sampling is independent of the depth at which they occur.

We decided to see whether similar results held on the Blocks World, Logistics, Kinship, and Five Puzzle domains, so we ran HPS with both node-selection settings on the same problems as earlier. We set operator retrieval to adaptive chaining and used the default settings from Table 1 for other system parameters. The leftmost graph in Figure 5 shows the results in a scatter plot, with depth-first search on the *x* axis and iterative sampling on the *y* axis, using the number of generated nodes, in logarithmic scale, as the performance measure. On many problems, the two backtracking methods behaved comparably, but iterative sampling outperformed depth-first search substantially on many Blocks World and a few Kinship tasks, while the opposite held for only one Blocks World problem. We have designed synthetic domains that reverse this trend, but our sample tasks do not appear to have problem spaces with the same characteristics.

These results suggested that adaptive backtracking may not be as productive an arena as operator retrieval, but it still seemed worth exploring. Langley's (1992) analysis took into account the branching factor $b$, the solution depth $d$, and the number of solutions $s$, from which he calculated the expected number of nodes generated by each strategy before finding a solution. We will not review the results in detail, as HPS makes somewhat different assumptions. In particular, the analysis assumed that iterative sampling has no memory of previous passes and so might regenerate nodes. In contrast, HPS retains nodes in its search tree and, although it may revisit them, it does not recreate them. Moreover, it does not create all possible children of a node at once, as in most heuristic search systems, but rather, like humans, generates them one at a time.

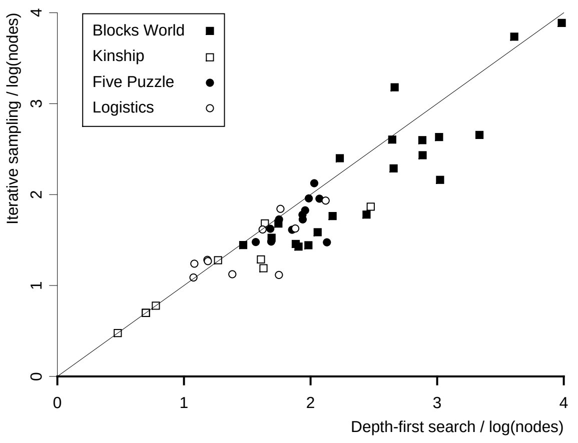
Given a problem with branching factor $b$, search depth $d > 0$, and $s$ solutions, we can calculate the expected number of nodes generated by depth-first search recursively as

$$E_{dfs}(b, d, s) = I(b, d, s) \cdot N(b, d-1) + E_{dfs}(b, d-1, s) + 1 \ ,$$

where $I(b, d, s) = max(0, b - \sqrt[d]{s})/(\sqrt[d]{s} + 1))$ is the expected number of incorrect choices and $N(b, d) = b^d + N(b, d-1)$ is the number of nodes in the subtree it will consider exhaustively when this occurs. We can compute the expected number of nodes generated by iterative sampling as

$$E_{is}(b, d, s) = [(d+1) \cdot J(b, d, s)] + E_{is}(b, d-1, s) + 1 \ ,$$

where $J(b, d, s) = (d+1) \cdot b^d/s$ is the expected number of samples taken before this method finds a solution path. We can use these expected values to control which node HPS selects upon failure,



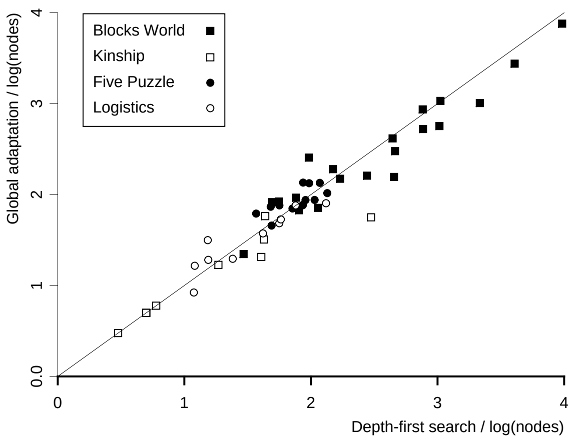


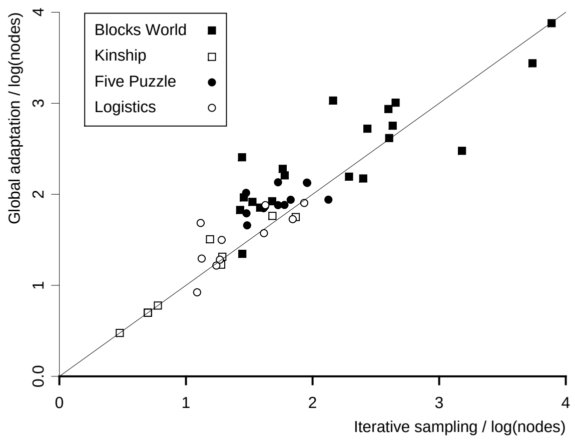
*Figure 5.* Scatter plots that compare the average number of nodes, in logarithmic scale, that HPS generates (left) using fixed strategies for iterative sampling vs. depth-first search and (right) using globally adaptive backtracking vs. depth-first search. Points below the diagonal denote problems in which the first alternative in each case explored fewer nodes than its competitor.

as when it detects a loop or reaches a depth limit, but its use requires estimates for $b$, $d$, and $s$. The first is trivial, as the problem solver knows the number of operator choices at the current node. We can estimate depth, the steps to reach a solution, as the number of unsatisfied goals from the current problem's goal description. We can estimate the number of solutions by finding all ways that entities in the state can match the goal description. Thus, given the goal description *((on ?x B) (on B ?y))* and three blocks – *A*, *B*, and *C* – there are two ways to satisfy the former – *((on A B) (on B C))* and *((on C B) (on B A))*. This ignores the possibility that multiple paths can produce a given solution, and also that some candidate solutions may be unreachable, but it has some merit.

There are two distinct ways to incorporate this idea into HPS. The most obvious involves estimating $b$, $d$, and $s$ for the top-level task, selecting depth-first search or iterative sampling, and then using this consistently during problem solving. A more sophisticated approach makes completely local decisions. Note that we can predict the number of nodes generated to find a solution not only for the top-level problem, but for any subproblem. This means that, when the current node fails, we can examine its parent *P* and, if depth-first search at this level appears better than iterative sampling, we select *P* as the new node. Otherwise we shift to *P*'s parent and repeat the test there, continuing until we reach the root or an ancestor that favors a depth-first strategy. We refer to this approach as *local* adaptation of backtracking to contrast it with the *global* version described earlier.

We implemented both of these conditional strategies in HPS as new settings for the parameter that controls node selection after failure. We expected that global adaptation would be more effective than the fixed backtracking strategies, and that local adaptation would make even better choices, giving us the conjecture:

- *Hypothesis: Globally adaptive backtracking will require no more, and sometimes less, search to find solutions than fixed depth-first search or iterative sampling. Moreover, locally adaptive backtracking will require no more, and sometimes less, search than global adaptation.*



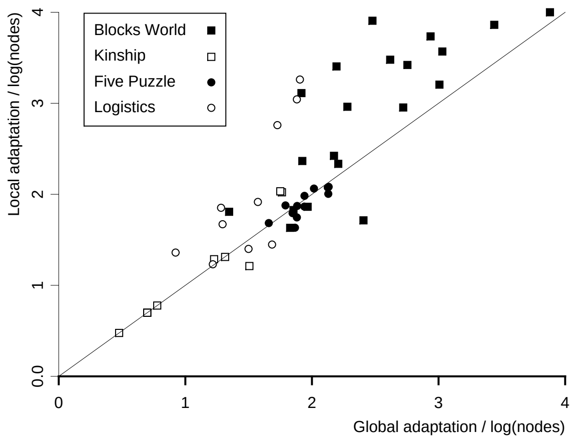


*Figure 6.* Scatter plots that compare the average number of nodes, in logarithmic scale, generated by (left) globally adaptive backtracking vs. iterative sampling and by (right) local adaptation vs. global adaptation. Points below the diagonal correspond to problems in which adaptive the first alternative in each case created fewer nodes than its competitor.

We tested both predictions on the same domains and problems that we used earlier. We ran HPS on each task with four distinct settings for backtracking: depth-first search, iterative sampling, global adaptation, and local adaption. We set the parameter for operator retrieval to use adaptive chaining and set all other architectural parameters to the defaults in Table 1. As before, we gave the problem solver a depth limit of ten and told it to terminate after considering 10,000 nodes, running it 20 times on each task. Figure 5 and 6 present the average results for each experimental comparison.

The studies reveal no clear benefits of adaptive backtracking. In fact, Figure 5 shows that fixed iterative sampling does better in relation to depth-first search than does global adaptation, and the left graph in Figure 6 clarifies that the latter is itself dominated even more by iterative sampling. Since global adaptation simply selects which of the two fixed backtracking methods to apply on a given problem, the results imply that it often makes the incorrect choice. The right graph in Figure 6 indicates the situation is even worse for local adaptation, which produces substantially more search than the global variety on many Blocks World and Logistics problems. The reasons for these disappointing results are unclear, but one plausible explanation is that HPS's estimate for the depth $d$ or the number of solutions $s$ is inaccurate. Both alternatives seem likely, as the method currently used does not consider that each goal may require multiple steps or that different sequences of operators may produce the same goal state. We can track down the culprit with synthetic domains that let us vary systematically the branching factor, solution depth, and number of solutions. Inspection of choice points where alternative methods make different backtracking decisions should clarify why the adaptive techniques do no better than depth-first search or iterative sampling.

## 3.4 Fixed and Adaptive Termination

A third important HPS parameter controls when the architecture should abandon a node in the search tree rather than elaborate it further. We will refer to this as the *termination* criterion, which is distinct from another system-level switch that determines when to halt problem solving entirely.[4] The classic criterion for such node termination is that the search tree has reached a maximum depth. A common generalization of this idea assigns costs to each operator and halts when the total cost of a node exceeds a threshold. Both are instances of fixed problem-solving strategies, and it seems natural to ask whether adaptive termination schemes would be more effective at reducing search.

We have explored an approach based on *rate of progress*. This requires some measure of each node's quality, such as the degree to which it satisfies the goal description. For example, given the Blocks World goals *((on a b) (on b c) (on c d))*, a node which produces a state that satisfies the goal *(on c d)* but not the others might receive a score of 1, whereas a node that satisfies all three goals might have 3 as its score. Other measures are possible; the important point is that node values should increase with the number of goals they satisfy. Given such a metric for node quality, we can measure progress toward the goal description as the rate of change in scores between a node in the search tree and the root node from which search started. For instance, a node that achieves one goal per level of depth should have a higher rate than one that achieves a goal every two levels.

The elements we need for such a progress measure include a goal-oriented score $R$ for the root, an analogous score $C$ for the current node $N$, and the depth $D$ of node $N$. Given these, one reasonable definition for progress $P$ is $(C - R + 1) / (D + 1)$. Let us return to the earlier goal description, assume that the node score is the number of goals satisfied, and consider two nodes at depth 3 in the search tree. At one extreme is a node $N1$ whose operators have produced a state that matches all three goals; this has a progress value $P = (3 - 0 + 1) / (3 + 1) = 1$. At the other extreme is a node $N2$ that does not satisfy any of the goals, so that $P = (0 - 0 + 1) / (3 + 1) = 1/4$. The deeper that search proceeds without achieving any goals, the lower this rate would become.

HPS can use such a measure of progress to decide when to abandon further search. Rather than backtracking when exceeding a specified depth, it would instead halt on dropping below an acceptable level of progress. The need to specify a threshold remains, but such a strategy is adaptive in the sense that it will explore the search space to different depths depending on their promise. This suggests another conjecture:

- *Hypothesis: Progress-bounded search will require no more, and sometimes less, effort to find a solution for a given problem than depth-bounded search. Moreover, the advantage of adaptive termination will increase with the fixed depth limit.*

This involves a somewhat different form of modulation than those we examined earlier, as it does not involve shifting among fixed strategies. Nevertheless, the approach alters the depth of search depending on how well each path appears to be doing, which arguably is a form of adaptation.

We tested this hypothesis on the same domains and problems as those in our previous experiments. We ran HPS with three settings, one that used a depth limit of ten as the termination criterion, another with a fixed depth limit of 14, and a third that abandoned nodes when their rate of progress fell below 0.15. We configured the system to use adaptive chaining and globally adaptive backtracking, but we kept settings for other strategic parameters at the defaults in Table 1. Again, we told the problem solver to continue search until it found a solution or considered 10,000 nodes. We ran each version of HPS 20 times per problem, recorded the number of nodes generated, and computed the

4. Even best-first search methods, which do not backtrack in the traditional sense, require some termination criterion.

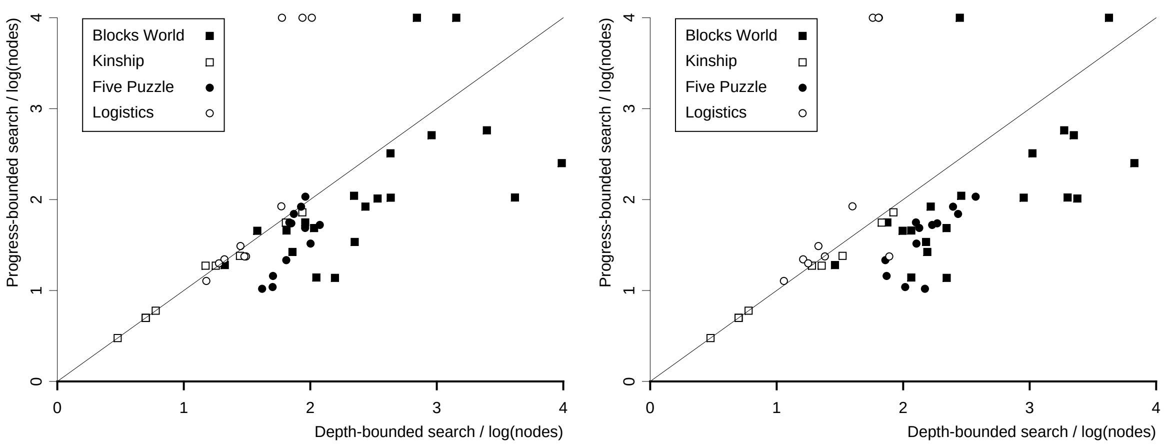


*Figure 7.* Scatter plots that compare the number of search nodes, in logarithmic scale, generated by progress-bounded and depth-bounded termination strategies. The left and right graphs show results when the fixed depth limit was ten and 14, respectively. Points below the diagonal denote indicate in which adaptive termination created fewer nodes.

mean for each strategy-problem pair. Figure 7 gives the results of this comparison, again presented in the form of scatter plots for average system behavior on each problem.

The left graph reveals that, as predicted, the adaptive mechanism typically requires less search before finding a solution than the fixed method. Progress-bounded termination generates substantially fewer nodes than depth-bounded halting on most Block World and Five Puzzle tasks, but there was no benefit on the Kinship or Logistics domains. The first was presumably because adaptive chaining was already so effective on the inference tasks that there remained little search, while the second was probably because Logistics included at most two goals, making progress a poor criterion. On two Blocks World and three Logistics problems, shown at the top of the graph, the adaptive method failed to find a solution within 10,000 nodes. Inspection revealed that these required multiple operator applications before achieving any goals, which in turn produced lower progress scores. This led HPS to abandon search at a shallower depth than was necessary to achieve the top-level goal description. This buttresses our earlier point that progress-bounded search still depends on a threshold that influences behavior, although it usually appears far less sensitive to this number than does the depth-bounded approach.

The right graph in Figure 7 shows that the benefits of adaptive termination over depth-bounded problem solving becomes somewhat greater as one increases the fixed depth limit. As before, there is little difference on Kinship and most Logistics problems, but the results for most Blocks World and Five Puzzle tasks on the scatter plot are slightly further below the diagonal. The absolute changes appear small, but remember that the graphs use logarithmic scale, so they offer some additional evidence that progress-bounded termination lets HPS mitigate the exponential character of problem solving by keeping the system from searching deeper than needed. In summary, the experiment generally supported our hypothesis that adaptive termination, at least this particular variety, leads to less effort than traditional depth-limited alternatives.

## 4. Related Research

The HPS framework is certainly not the first to support multiple problem-solving strategies. For instance, Soar (Laird et al., 1987) demonstrated the ability to mimic many different weak methods with generic control rules, and FLECS (Veloso & Stone, 1995) extended PRODIGY to support both backward and forward chaining. But the main topic of this paper is *adaptation* in problem solving, which has received far less attention. There has been substantial work on learning in this setting, much in the context of cognitive architectures. However, the emphasis has been on acquiring search-control rules or macro-operators that reduce effective branching factor or depth of solutions. In contrast, we are concerned with adaptation in response to generic problem characteristics that, in principle, should reduce effort independently of whether learning occurs. Anderson's (1990) analysis of problem solving comes closest to our approach, as it showed how changing probabilities of success in the forward and backward direction can produce strategy shifts, but it assumed estimation over multiple solution attempts rather than our immediate form of adaptation.

We should also consider work that relates to our specific adaptation methods. Our approach to selecting forward or backward chaining resembles techniques for bidirectional search, but there are important differences. Most of these systems (e.g., Lippi et al., 2012; Holte et al., 2016) assume a fully specified goal state, whereas HPS can solve problems with abstract goals. A few techniques (e.g., Torralba et al., 2014) use binary decision diagrams to encode abstract goals, but even they carry out search through a state space, as opposed to HPS's traversal of a plan space. This means they must deal with two separate search trees, each with its own frontier, to ensure that a forward and backward path eventually meet. Felner et al. (2010) report a bidirectional method that retains a single frontier of state-goal pairs and that favors the direction with a lower branching factor, making it closer to HPS than other approaches. However, their system operates over fully specified states and only considers backward operators whose entire effects match the goal state. This makes it closer to a combination of forward-chaining search and regression planning, whereas our extension adaptively combines forward search and means-ends analysis.

HPS's mechanisms for adaptive backtracking, especially the one that makes local decisions, is similar in spirit to classic techniques for dependency-directed backtracking (Stallman & Sussman, 1977) and dynamic backtracking (Ginsberg, 1993). But these methods focused on constraint-satisfaction tasks rather than planning problems, and they reasoned about specific choices made during search to select an ancestor node, rather than examining high-level problem statistics. There is potential for incorporating such reasoning into HPS's backtracking strategy, but this would involve a very different approach from the one we have described, which supports a continuum between depth-first search and iterative sampling. There has been less related work on adaptive termination criteria. The closest is Bhatnagar and Mostow's (1994) FAILSAFE-II, which learned control rules from failures it encountered during problem solving. This system declared a node as failed when search exceeded a specified limit without achieving a goal or encountering other signs of activity. As a result, it explored the space to different depths, although it did not use our definition of progress.

Portfolio methods for planning (e.g., Cenamor et al., 2016; Gerevini et al., 2009) also support a form of adaptive problem solving, but again the resemblance is misleading. These invoke a number of different search techiques in parallel and return the first solution found. In contrast, HPS carries out a single search through its problem space but adapts its strategy to the problem or subproblem at hand. Techniques for automated configuration or tuning of problem solvers (e.g., Fawcett et al., 2011) are much closer, as they select a specific combination of parameters in a general framework. However, these methods base their tuning on behavior over a set of training problems, whereas HPS adapts its strategies to the current problem, without need for training cases.

In summary, the AI literature contains few examples of adaptive problem solving of the type reported here. There is anecdotal evidence that humans alter their strategies in response to problem characteristics. People use means-ends analysis on some tasks but resort to forward chaining on others, they adopt systematic exploration and repeated sampling in different settings, and they carry out search through problem spaces to variable depths. However, apart from Simon and Reed's (1976) early documentation of strategy shifts in novice behavior on puzzles, the psychology literature is mute on this important phenomenon, and the role of adaptation in problem solving merits far greater attention from both research communities.

## 5. Concluding Remarks

In the preceding pages, we described HPS, an architecture for problem solving that searches a space of decompositons to transform an initial state into another that satisfies a goal description. We saw that the system includes parameters whose settings determine its search strategy, but also that these elements are fixed. In response, we developed adaptive methods for the parameters that control operator retrieval, backtracking when a node fails, and when to abandon a node. Experiments with adaptive retrieval on four domains showed that it often produced less search than either state-directed forward chaining or goal-directed means-ends analysis. Studies of adaptive backtracking were less promising, with both global and local adaptation doing nearly the same as fixed depth-first search and iterative sampling. The results for adaptive termination were stronger, with progress-bounded search not only more effective than depth-bounded halting, but with increased depth limits heightening the effect. These results are positive enough to encourage further work on a topic that, as we have seen, has received little attention in either AI or cognitive psychology.

There are many avenues for additional research. Our highest priority should be understanding the reasons that adaptive backtracking is sometimes ineffective, but we should also replicate our positive results on additional domains. These should include both synthetic classes of task that allow systematic control and more established ones that are familiar to the AI planning and search communities. We should also examine problems that have been studied by cognitive psychologists to strengthen our links to their literature. In addition, the compositional character of HPS's strategic parameters suggests that we examine interactions between methods for operator retrieval and backtracking. In some cases, adaptive backtracking appears to be more effective with backward chaining than forward search, and we should explain this behavior and look for similar phenomena. Moreover, we should study more fully variants of HPS that can adapt simultaneously on multiple fronts, looking for both synergies and interference in their joint effects on problem solving.

Finally, we should explore adaptive variations on other parameters. One candidate is responsible for selecting a node after finding a problem solution, which plays a role when multiple answers are desired. Here meta-level criteria like a desire for solution diversity might affect how far to backtrack after success. Another is the parameter that controls selection of operators after they have been filtered for relevance, which can introduce more or less randomness into the search process. A third is the parameter responsible for determining a node's acceptability as a solution. Upon encountering especially difficult problems, an augmented HPS might weaken this criterion, effectively lowering the system's aspiration level to enable satisficing. These extensions may lead to additional insights about the adaptive character of problem solving in cognitive systems.

## Acknowledgements

This research was supported by Grant N00014-15-1-2517 from the Office of Naval Research, which is not responsible for its contents. We thank Christopher MacLellan, Miranda Emery, Chris Pearce, Yu Bai, and Charlotte Worsfold for their contributions to early versions of the HPS architecture, and to Stephanie Sage for invaluable discussions about the framework.

## References

Anderson, J. R. (1990). *The adaptive character of thought*. Hillsdale, NJ: Lawrence Erlbaum.

Anderson, J. R. (1993). Problem solving and learning. *American Psychologist*, *48*, 35–44.

Bhatnagar, N. & Mostow, J. (1994). On-line learning from search failures. *Machine learning*, *15*, 69–117.

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.

Cenamor, I., De La Rosa, T., & Fernández, F. (2016). The IBaCoP planning system: Instance-based configured portfolios. *Journal of Artificial Intelligence Research*, *56*, 657–691.

de Groot, A. D. (1978.) *Thought and choice in chess* (2nd ed.). The Hague: Mouton Publishers.

Fawcett, C., Helmert, M., Hoos, H., Karpas, E., Röger, G., & Seipp, J. (2011). FD-autotune: Domain-specific configuration using fast downward. *Proceedings of the ICAPS-2011 Workshop on Planning and Learning* (pp. 13–17). Freiburg, Germany: AAAI Press.

Felner, A., Moldenhauer, C., Sturtevant, N. R., & Schaeffer, J. (2010). Single-frontier bidirectional search. *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence* (pp. 59–64). Atlanta, GA: AAAI Press.

Genesereth, M. R., Greiner, R., Smith, D. E. (1981). *MRS manual* (Technical Report HPP-80-24). Department of Computer Science, Stanford University, Stanford, CA.

Gerevini, A., Saetti, A., & Vallati, M. (2009). An automatically configurable portfolio-based planner with macro-actions: PbP. *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling* (pp. 350–353). Thessaloniki, Greece: AAAI Press.

Ginsberg, M. L. (1993). Dynamic backtracking. *Journal of Artificial Intelligence Research*, *1*, 25–46.

Holte, R. C., Felner, A., Sharon, G., & Sturtevant, N. R. (2016). Bidirectional search that is guaranteed to meet in the middle. *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence* (pp. 3411–3417). Phoenix, AZ: AAAI Press.

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

Langley, P. (1983). Exploring the space of cognitive architectures. *Behavior Research Methods and Instrumentation*, *15*, 289–299.

Langley, P. (1992). Systematic and nonsystematic search strategies. *Proceedings of the First International Conference on Artificial Intelligence Planning Systems* (pp. 145–152). College Park, MD: Morgan Kaufmann.

Langley, P. Choi, D., & Rogers, S. (2009). Acquisition of hierarchical reactive skills in a unified cognitive architecture. *Cognitive Systems Research*, *10*, 316–332.

Langley, P., Pearce, C., Bai, Y., Barley, M., & Worsfold, C. (2016). Variations on a theory of problem solving. *Proceedings of the Fourth Annual Conference on Cognitive Systems*. Evanston, IL: Cognitive Systems Foundation.

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

Lippi, M., Ernandes, M., & Felner, A. (2012). Efficient single frontier bidirectional search. *Proceedings of the Fifth International Symposium on Combinatorial Search* (pp. 49–56). Niagara Falls, ON: AAAI Press.

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.

Torralba, A., López, C. L., & Borrajo, D. (2016). Abstraction heuristics for symbolic bidirectional search. *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence* (pp. 3272–3278). New York: IJCAI.

McDermott, D. (1991). Regression planning. *International Journal of Intelligent Systems*, *6*, 357–416.

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.

Newell, A., & Simon, H. A. (1976). Computer science as empirical inquiry: Symbols and search. *Communications of the ACM*, *19*, 113–126.

Siegler, R. (1989). Hazards of mental chronometry: An example from children's subtraction. *Journal of Educational Psychology*, *81*, 497–506.

Simon, H. A., & Reed, S. K. (1976). Modeling strategy shifts in a problem-solving task. *Cognitive Psychology*, *8*, 86–97.

Stallman, R. M., & Sussman, G. J. (1977). Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, *9*, 135–196.

Veloso, M., & Stone, P. (1995). FLECS: Planning with a flexible commitment strategy. *Journal of Artificial Intelligence Research*, *3*, 25–52.