Machine Learning, 15, 69–117 (1994)
© 1994 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

On-Line Learning from Search Failures

NEERAJ BHATNAGAR (BHATNAGA@INFERENCE.COM)
Inference Corporation, 550 N. Continental Blvd., El Segundo, CA 90245

JACK MOSTOW (MOSTOW@CS.CMU.EDU)
Carnegie Mellon University, Robotics Institute, 215 Cyert Hall, 4910 Forbes Avenue, Pittsburgh, PA 15213

Editor: Paul Rosenbloom

Abstract. Learning by explaining failures and avoiding similar ones thereafter is an attractive way to speed up problem solving. However, previous methods for explanation-based learning from failure can take too long to detect failures, explain them, or test the learned rules. This expense is especially critical for **adaptive search**, in which control knowledge acquired while solving an individual problem instance must be learned quickly enough to speed up its solution.

We present an adaptive search technique that speeds up state-space search by learning heuristic censors while searching. The censors speed up search by pruning away more and more of the space until a solution is found in the pruned space. Censors are learned by explaining dead ends and other search failures. To learn quickly, the technique overgeneralizes by assuming that certain constraints are **preservable**, i.e., remain true along at least one solution path. A recovery mechanism detects violations of this assumption and selectively relaxes learned censors. The technique, implemented in an adaptive problem solver named FAILSAFE-2, learns useful heuristics that cannot be learned by other reported methods.

We present experimental evidence that FAILSAFE-2 is **effective** (learns useful rules, even in recursive domains where PRODIGY and STATIC do not), **adaptive** (learns fast enough to pay off even within a single problem), and **general** (speeds up diverse problem solvers, even initially strong ones).

Keywords. explanation-based learning, learning from failure, on-line learning, adaptive search, preservable constraints, learning control knowledge for state space search, FAILSAFE-2, PRODIGY, STATIC, SOAR

1. Introduction

This article identifies some limitations of previous methods for explanation-based learning from failure, proposes an approach for overcoming them, describes the implementation of this approach in an adaptive state-space problem solver named FAILSAFE-2, and presents empirical evaluations of its effectiveness. The work described here falls within an especially active area of research on explanation-based learning: automating the acquisition of search control knowledge for problem solvers. In principle, a problem solver should be able to speed itself up over time by learning from its own experience. In particular, given a theory of its own problem-solving architecture and environment, it should be able to explain its own successes and failures, and extract from them general rules for controlling its problem solving more efficiently in the future. In practice, this goal has remained somewhat elusive—although there are a number of problem solvers that use explanation-based learning to acquire search control knowledge, the vision of autonomous problem solvers that learn to solve complex practical problems in realistic domains has yet to be fully realized.

The work reported here is one step towards that ambitious vision; it focuses on an attempt to improve one type of learning—learning from failure, broadly defined as lack of progress toward the goal.

There are some compelling reasons to believe that learning from failure will be especially important in complex domains. First, in complex domains it is not sufficient to learn only from successes, even partial successes like achieved subgoals, because they may be too difficult to obtain. Without effective control knowledge, an autonomous problem solver may exhaust its resources even before it reaches its first partial success. Without this partial success, the problem solver cannot start to learn the control knowledge it needs. Partial successes may be few and far between in complex domains, especially difficult ones characterized by a low density of solutions in the space of candidates explored by problem solvers before they have acquired effective control knowledge. In contrast, states that do not satisfy the goal are plentiful and easy to come by, and thus should provide a much more readily available source of data to learn from. Second, in the absence of effective control knowledge, many or most of a problem solver's actions are just plain stupid (i.e., lead to obvious failures) or irrelevant (i.e., do not get any closer to the goal). In fact, Kibler and Morris (1983) showed that for some standard tasks, a few simple rules for avoiding "stupid" actions were sufficient to eliminate most of the search. If such actions are the primary source of search inefficiency, learning to avoid them should speed up the problem solver dramatically. FAILSAFE-2 detects such actions while solving a problem, explains them, and learns heuristics to avoid similar actions thereafter, whether during the same problem it is in the middle of solving, or in later ones.

The general idea of using explanation-based learning from failure to acquire control knowledge for a problem solver has been employed before, notably in PRODIGY (Minton, 1988a, 1988b, 1990). However, PRODIGY suffers from three basic problems. First, it waits to learn from failures until a dead end (or similar search failure) is reached during problem solving. In complex problems, even a dead end may occur too late. Second, it explains failures by exhaustively examining the alternatives. Such conservative learning can be too expensive to do while problem solving. Third, if the explanation involves recursion, it learns too many rules of relatively low utility (Etzioni, 1990a). The ULS system (Chase et al. 1989) addresses the last problem by transforming learned operator preference rules into approximate versions that are cheaper to test, but at this point the expense of detecting and explaining failures has already been incurred. The STATIC system (Etzioni, 1990a) reduces the cost of failure detection by compiling control rules *before* problem solving begins, but finesses the other problems simply by refraining from trying to learn about failures whose explanations involve recursion.

The FAILSAFE-2 system reported here attempts to alleviate all three problems:

1. To avoid exhaustive searches while solving complex problems, FAILSAFE-2 uses various heuristics to detect failure early on.
2. To avoid constructing complex, exhaustive explanations of detected failures, FAILSAFE-2 builds quick, incomplete explanations.
3. To avoid the expense of testing complex rules, FAILSAFE-2 learns simple, overgeneral rules that sometimes cause it to exclude *all* solutions to a problem. When this situation arises, FAILSAFE-2's problem solver must overrule the recommendations of its learner.

FAILSAFE-2's heuristic approach to explanation-based learning of control knowledge evolved in the context of *adaptive search*—learning in the course of solving an individual problem instance, so that information accumulated during search can be used to guide subsequent search more and more accurately. An important example of adaptive search is *design iteration*, in which information gleaned from an unacceptable design is used to guide subsequent exploration of the design space. Although design continues to serve as a motivating example for us, it involves many other complicating factors (Mostow, 1985; Tong, 1987a, 1990). Therefore, we started with the simpler case of state-space search, though we hope that some of the insights that emerge from this simpler case may eventually prove applicable to design. Although some work has been done on learning search control knowledge in the context of design (Steier, 1987; Tong, 1987b, 1988; Tong & Franklin, 1989; Mostow, 1989; Mostow et al., 1989; Norton & Kelly, 1988; Mahadevan, 1989), it has not focused on online learning from failure.

In particular, we were interested in learning from search failures (such as dead ends), since they are an abundant form of data in solving complex problems. We wanted to use explanation-based learning to generalize from such failures so as to extract as much useful control knowledge from them as possible. With explanation-based learning, we hoped that the learned control knowledge would apply throughout the search space for the same and subsequent problems, and thereby serve to prevent many similar failures. Thus an initially inefficient search would become progressively more focused: each failure encountered along the way would cause another portion of the search space to be pruned away. By dynamically reducing the search space, adaptive search would, we hoped, make it feasible to solve otherwise intractable problems. Our initial implementation of this approach in FAILSAFE-2's predecessor, called FAILSAFE, yielded some encouraging preliminary results (Mostow & Bhatnagar, 1987), but left some important issues unresolved, as we shall soon explain.

The adaptive search context imposes some strong requirements on the explanation-based approach to learning from failure. First, the cost of failure detection is a lower bound on the cost of the overall search; failures must be detected quickly to avoid an intractable search before they are found, since search is blind before learning occurs. To take a simple example, suppose the problem solver uses forward chaining to solve a blocks world problem involving *n* blocks. Before it learns any control knowledge to guide its search, it will explore arbitrary sequences of moves. These sequences can grow exponentially long before reaching a search failure in the form of a dead end where every move leads to an already-seen state.[1] Therefore, an efficient method for learning from search failures must be able to detect them long before a dead end is reached.

Second, failures must be explained quickly, since the adaptive search context dictates that the cost of learning must be amortized over a single problem. The learner cannot afford to spend a long time learning good rules; instead, it must quickly learn rules that may be imperfect. Moreover, the explanation must be based on the information available when the failure is detected; the learner cannot wait to do a retrospective analysis of the completed search. That is, the learner must operate *on-line*.

Third, the learned rules must be efficient to use. Of course, this "utility" requirement also applies in the non-adaptive case, where knowledge learned from one problem is used only on future problems. However, the use of imperfect learning in the adaptive context complicates the issue of utility, in that the problem solver must be able to tolerate erroneous learned rules well enough to recover from their ill effects. (This problem can be avoided in non-adaptive learning contexts, where it is computationally feasible to learn correct rules off-line.) Moreover, to prevent those effects from recurring, the learner must be able to correct erroneous rules efficiently and dynamically, subject to the same limitations on the time and information available for learning them in the first place.

This article discusses how FAILSAFE-2's problem solving and learning components address these requirements.

1.1. Dynamic adjustment of search control knowledge in FAILSAFE-2

FAILSAFE-2 performs tasks that consist of finding a legal (but not necessarily optimal) solution path from a given initial state to a state that satisfies a given goal. Its purpose is to find such a path quickly. Learning speeds up the search by constraining the generation of paths so that paths that are likely to fail are not generated.

Accordingly, FAILSAFE-2 searches alternately at two different levels, as shown in figure 1. At the base level, the problem solver performs forward-chaining search in a problem space defined by an initial state, a conjunctive set of goals, a set of operators, and a knowledge base of search control rules. *Censors* prune proposed operators. Other rules order goals. The object of this search is to find a path from the initial state to a goal state.

At the meta-level, the learner tunes the search control knowledge by adding and modifying search control rules. The object of this tuning is to speed up the base-level search by converging on search control knowledge that defines a base-level space small enough to search quickly, yet large enough to contain a solution path. The idea is to prune away more and more of the space until a solution is found.

FAILSAFE-2 treats state-space search as an enumeration of allowed paths until a solution path is found (Mostow, 1983). We now classify some different types of constraints on such paths.

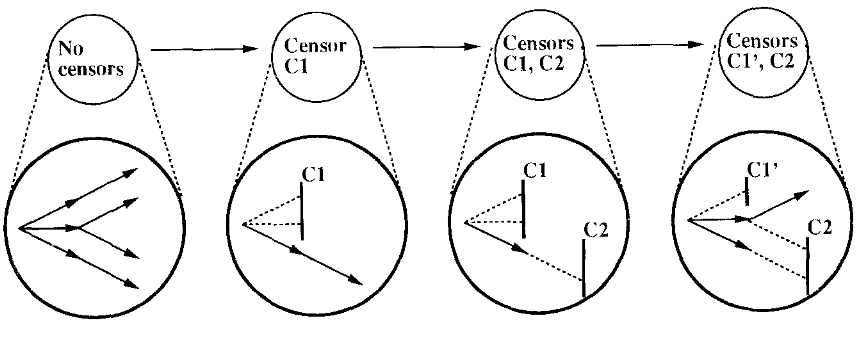


Figure 1. FAILSAFE-2's two levels of search.

1.2. Types of constraints on paths

Consider a constraint on the states occurring on a path. We shall say that such a constraint holds for a path if it holds for *every* state in that path. We are particularly interested in constraints on *solution paths*, which we define as all sequences of legal operator applications leading from a given state I to any state that satisfies the given goal G. For example, the goal G might be to color block A green, where in the initial state I, block A is red and there is some green paint.

We classify such constraints into three types, as depicted by the three diagrams in figure 2. Each diagram shows an initial state, a goal, and some possible paths through the state space. The goal is drawn larger to indicate that it may be satisfied by more than one state. A constraint splits the state space into one region where it is satisfied and another where it is not.

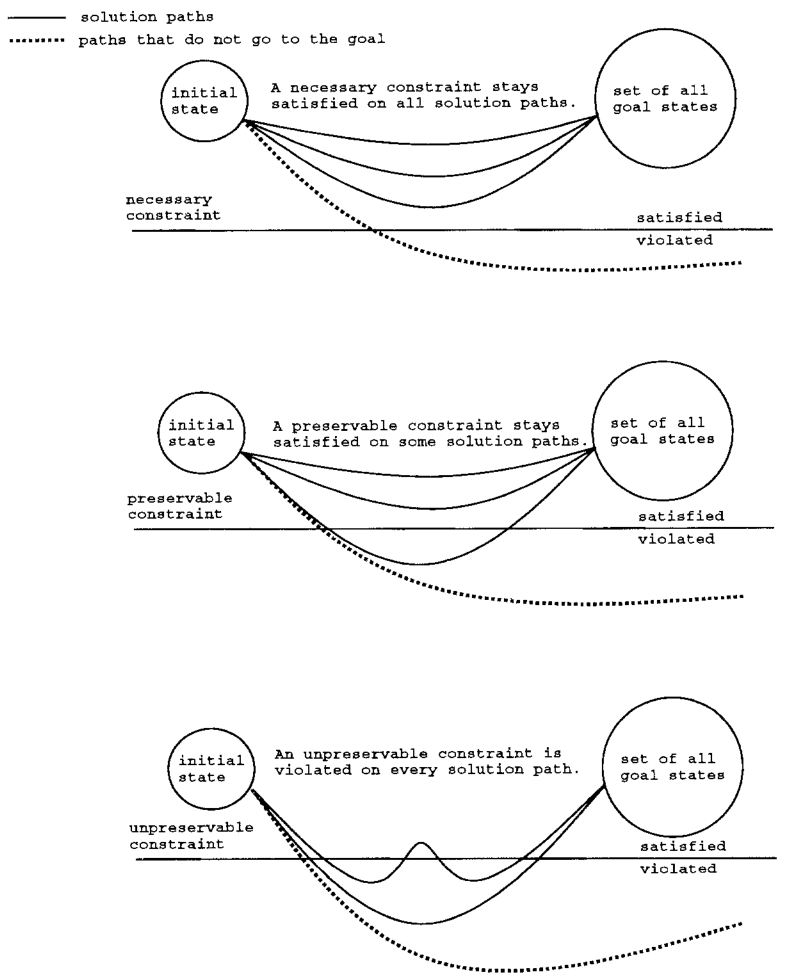


Figure 2. Three types of constraints on states.

A *necessary* constraint holds for *all* solution paths and, therefore, should be preserved on the paths generated by the problem solver. That is, if a necessary constraint is violated, the problem solver *must* backtrack. "Block *A* exists" is a necessary constraint in our example, assuming we have no operator to create it.

A *preservable* constraint holds for *some*, but not necessarily all solution paths. If a preservable constraint is violated, the problem solver *may* backtrack, but is not forced to, since other paths on which it is not violated also lead to solutions. "Blue paint is available" is a preservable constraint in our example, since there are paths to the goal through states that violate this constraint.

An *unpreservable* constraint does not hold for *any* solution path, and must temporarily be violated in order to reach the goal. If the problem solver backtracks every time an unpreservable constraint is violated, it will cut off all solution paths. "Block *A* is dry" is an unpreservable constraint, since painting it will get it wet.

This intuitive description is not quite precise, since the type of a constraint actually depends on the operators available to the problem solver, and on the other constraints that the problem solver is currently imposing. More formally, let *S* be a set of constraints on the generated paths. We say that with respect to a given initial state, goal, and operators, the set *S* is

- *necessary* if every constraint in *S* stays satisfied along *all* solution paths
- *preservable* if every constraint in *S* stays satisfied along at least *one* solution path
- *unpreservable* if at least one constraint in *S* is violated on *every* solution path

FAILSAFE-2 learns preservable constraints, backtracks whenever they are violated, and recovers from unpreservable constraints. Thus, the objective of learning in FAILSAFE-2 is to converge to a preservable set of constraints. For adaptive search in the context of a single problem, the purpose of learning is to converge to a set of constraints that is preservable for the given operators, initial state, and goal. If the problem solver is to be applied to a number of problems with different initial states and goals but the same operators, the purpose of learning is to converge to a set of constraints that is preservable for every possible initial state and goal, or at least for those that will be encountered in the problem set.

FAILSAFE-2 extends previous work on FOO (Mostow, 1983) and FAILSAFE (Mostow & Bhatnagar, 1987). FOO (Mostow, 1983) used the idea of necessary constraints to derive heuristics for pruning the search. The derivations were interactive; at each step, the user selected a transformation from a catalog, and FOO applied it. FAILSAFE automated the derivation of such heuristics by using explanation-based learning to explain constraint violations and acquire rules for avoiding similar violations thereafter. However, it lacked a mechanism for dealing with unpreservable constraints. By allowing incomplete explanations of failures, FAILSAFE-2 generalizes the concept of necessary constraints (that *must* be preserved) to that of preservable constraints (that *may* be preserved). Furthermore, by allowing relaxation of learned censors when they give incorrect results, FAILSAFE-2 provides a mechanism to recover when the constraints turn out to be unpreservable.

In order to describe FAILSAFE-2's learning method, we must first provide some additional detail on its problem solver and how it interacts with the learner. We next describe learning mechanism and some refinements to it. We then present experimental evaluations of its effectiveness. Finally, we relate FAILSAFE-2 to other work and assess its significance.

2. Architecture of FAILSAFE-2

We now specify FAILSAFE-2's architecture in terms of its decomposition into two interacting components—the problem solver and the learner—and the decomposition in turn of the problem solver into a fixed framework and the variable components that fit into it.

2.1. *FAILSAFE-2's problem solver*

The forward-chaining problem solver takes as input a state-space problem specified by an initial state and a goal (both expressed as conjunctions of ground literals), a set of STRIPS operators, and possibly some control knowledge, and produces as output a solution path leading from the initial state to a goal state via a sequence of operator applications.

The problem solver has two main components—the *state selector* and the *operator generator.* The state selector is repeatedly invoked to choose which open state to expand next (starting with the initial state). The operator generator then chooses which previously untried operator to apply. If the resulting state has been generated previously, it is ignored, and the remaining operators are tried until one of them generates a new state, which is then added to the set of open states. Once a state has been completely expanded, it is marked *closed* to prevent its being selected again.

Associated with each state are a *current goal* to achieve and zero or more *pending goals* to achieve thereafter. When the current goal is achieved, it is marked as *protected,* and the problem solver chooses an unsatisfied pending goal as the new current goal for the resulting state. (The problem-solving architecture does not automatically prevent the problem solver from undoing goals marked as protected, but learned control rules may have that effect.)

The problem solver uses two types of learned search control knowledge. *Goal-ordering rules* constrain the selection of which goal to pick as the next current goal. These rules are used to order the unsatisfied goals in a state as soon as the state is generated. *Censors* constrain the selection of which operator to apply to the current state. Censors are applied when a state is expanded: if the problem solver tries to apply an operator to a state, and the preconditions of the operator are satisfied but applying it would violate a censor, FAILSAFE-2 marks the state as *suspended* so that it can be reconsidered if no solution is found otherwise.[2]

Depending on what state selector and operator generator are used, FAILSAFE-2 can perform different types of state-space search, including depth-first, breadth-first, heuristic best-first, A*, and means-ends analysis. For example, depth-first search is performed by choosing the most recently visited open state and applying the "next" applicable operator to it. To limit search depth, the state selector can be modified not to select any state deeper than some specified limit.[3] To make operator generation sensitive to the current goal, the operator selector is replaced with one that uses means-ends analysis to find operators that could achieve the goal, and operator subgoaling to identify operators that not only lead towards the goal but also can be applied in the current state.

Notice that FAILSAFE-2 does not itself learn any control knowledge to improve the internal process of operator generation; it only learns rules for selecting goals and censoring operators. However, the operator generator can make use of control knowledge acquired by other means. In particular, in some of our experiments, FAILSAFE-2's operator generator consisted of PRODIGY (Minton, 1988a) guided by the control rules produced by STATIC (Etzioni, 1990b). Thus FAILSAFE-2's operator generators span a spectrum ranging from weak (blind forward chaining) to strong (knowledge-guided means-ends analysis).

2.2. Interactions between the problem solver and the learner

In FAILSAFE-2 the problem solver and the learner have two types of interactions, which occur when the search appears *under-constrained* or *over-constrained.* In perfectly constrained search, the problem solver proceeds straight to the goal without backtracking. In under-constrained search, it traverses many unsuccessful paths before finding a solution path. In over-constrained search, the problem solver is constrained so much that all solutions to the problem are blocked. We now specify the interface between the problem solver and the learner by describing these two types of interactions, but we postpone until section 3 the internals of the learner itself.

2.2.1. Interactions when the search is under-constrained

When the search is under-constrained, the problem solver hits too many dead ends or explores blind alleys without reaching the goal. In the case of under-constrained search, the learner attempts to learn a new censor so as to constrain the search. FAILSAFE-2 treats the following four situations as symptoms of under-constrained search:

1. **No operators apply:** A "dead end" state is reached in which no operator is applicable, or where all applicable operators lead back to already visited states. A perfectly constrained problem solver will not generate such states.
2. **All operators censored:** All applicable operators in a state are censored by previously learned censors. Assuming that the learned censors are preservable, a path to the goal exists elsewhere in the search space. Thus, reaching such states also indicates that the search is under-constrained.
3. **A protected goal is violated:** Applying an operator in a state leads to the violation of a goal protected in that state. Assuming that the goals are serializable, it should be possible to solve the problem without violating a protected goal.
4. **Waited too long to learn:** Whenever a specified number of new states is generated without achieving the current goal, FAILSAFE-2 treats it as one more indicator of under-constrained search.[4] The purpose of this heuristic, called "forced learning," is to escape from long blind alleys that contain neither goal states nor dead ends. Forced learning resembles a conventional search depth limit in its effect, but differs in that it does not sacrifice completeness, for reasons to be made clear later.

The problem solver continues to search until any of these symptoms occurs, at which point it declares a failure and hands over control to the learner. When the learner is invoked, it is given the state in which failure has just been declared, along with the path that led to it from the initial state—but no other branches of the search, since they may not have been generated yet. This limitation reflects the requirement that the learner operate on-line, in contrast to systems that wait to learn from a subtree until after they finish exploring it, at which point they analyze the entire explored part.

The learner's task is to explain why the current state fails to satisfy the current goal, assign blame for this failure to one of the operators on the path to the current state, learn an appropriate censor on this operator, and instruct the problem solver to resume search at the state where this operator was applied—much as dependency-directed backtracking (Stallman & Sussman, 1977) backtracks to the state blamed for a failure, except that FAILSAFE-2 generalizes from failures. If the censor returned by the learner is new, it is added to the set of censors and takes effect immediately.

The learner cannot always perform this task in full. If it can explain why the state doesn't satisfy the goal, but cannot identify an operator to blame, it assumes that the operators on the given path are *irrelevant* to the current goal, that is, do not contribute toward achieving it. FAILSAFE-2 then learns an *irrelevancy censor* (as described in section 4.2) and backtracks chronologically, i.e., resumes search at the parent of the state where failure was declared. If the learner cannot even explain the failure, it does not learn any censor, and just resorts to chronological backtracking.

The four indicators of under-constrained search in FAILSAFE-2 are closely related to impasses in SOAR. Like impasses in SOAR, these indicators force FAILSAFE-2 to change its current task in favor of a new one with the intention of improving the overall problem-solving performance. The first two of the above four interactions have exact counterparts in the SOAR architecture. The third interaction is detected via rules in some SOAR-based systems, but is not part of the architecture. The fourth interaction is heuristic in nature and is novel to FAILSAFE-2.

2.2.2. *Interactions when the search is over-constrained*

The other type of interaction between the problem solver and the learner occurs when the search is *over-constrained.* Over-constrained search prunes away all solution paths. It is manifested when the problem solver fails to continue making progress. In the case of over-constrained search, the learner tries to "guess" a good censor to relax. If relaxing this censor indeed leads to achieving the goal, the learner then specializes the censor to avoid repeating the same mistake.

FAILSAFE-2 treats the following two situations as symptoms of over-constrained search:

1. **No states to expand:** This extreme indicator of over-constrained search occurs when the problem solver runs out of states to expand. That is, the open list becomes empty, but the problem is still unsolved.
2. **Waited too long to relax:** If the problem solver generates a specified number of states without achieving the current goal, it declares the search to be over-constrained.[5] The purpose of this heuristic, called "forced relaxation," is to escape from exhaustive search of a space rendered barren by overgeneral censors.

These two situations also resemble impasses in SOAR. While the first situation has an exact counterpart in SOAR, the second is heuristic in nature and is novel to FAILSAFE-2. Whenever either of these situations occurs, FAILSAFE-2 presumes that at least one of the suspended states was wrongly suspended, and could lead to a solution by relaxing the right censor(s). At this point, FAILSAFE-2 heuristically selects a state *st* and an operator *op* such that operator *op* is blocked by some censor *C* in the suspended state *st*. The problem solver resumes search by overruling *C* and applying *op* to *st*. This process, called *censor relaxation*, employs the following heuristics:

- For *st*, select the suspended state with the largest number of true subgoals, on the assumption that it is the closest to a complete solution.
- In case of a tie, select the state at the shallowest depth, so as to encourage shorter solutions to problems.
- If more than one operator is censored in *st*, then return the least recently censored one, so as to relax operators in more or less the same order they were generated.

Relaxing a censor in a state does not deactivate the censor everywhere, just for a single operator in that state. If applying operator *op* in state *st* generates a new state, the new state is added to the open list as if it were generated during normal problem solving. Otherwise, censor relaxation is repeated until either a new state is generated or the learner runs out of suspended states, in which case it simply returns control to the problem solver. If there are no suspended states and no open states, the search space is exhausted, and FAILSAFE-2 declares the problem unsolvable.

How does learning affect FAILSAFE-2's completeness as a problem solver? Since censors merely suspend operators rather than prune them permanently, in theory FAILSAFE-2's censor relaxation mechanism preserves completeness relative to the uncensored operator generator.[6] In practice, FAILSAFE-2 exhausts the search space only for relatively small problems. For most larger problems, it either finds a solution or else stops when it reaches a resource limit on total search cost, as we will explain in section 5.

Since FAILSAFE-2's censor relaxation heuristics are not infallible, relaxing the chosen censor may or may not lead to a solution. If it does not, the space will remain overconstrained, leading to additional relaxation. However, when relaxing a censor does eventually lead to achieving a goal, FAILSAFE-2 infers that the censor was overgeneral and calls the learner to specialize it.

Whenever the problem solver achieves its current goal, it invokes the learner to see if any censors should be specialized. It passes to the learner the subsequence of steps during which it was working on the goal it just achieved. If any steps along this path were initially censored, the learner assumes the censors were overgeneral, since relaxing them led to achieving the goal. The learner therefore specializes the censors to prevent them from blocking progress in similar future cases. FAILSAFE-2's processes of relaxation and specialization of censors can be related to the process of recovery from incorrect knowledge in SOAR (Laird, 1988). SOAR's recovery process enables it to recover from many different types of incorrect knowledge, namely, when an incorrect object is preferred over the correct object, when a correct object is incorrectly rejected, and when an incorrect object is inappropriately required. FAILSAFE-2's overgeneral censors resemble the second of the above three types of incorrect control knowledge in SOAR. The recovery mechanism in SOAR is different from the recovery mechanism in FAILSAFE-2. While FAILSAFE-2 specializes the overgeneral censors, SOAR attempts to learn new chunks that mask the undesirable effects of the incorrect chunks.

3. FAILSAFE-2's learner

Having completed our decription of the problem solver and its interface to the learner, we now explain how the learner actually works. When the learner is invoked, it is given the current set of censors, the current state, and the path that led to it, including the goals at each point on the path. In addition to this dynamic information, the learner also makes use of the following static knowledge, which depends on the problem domain but does not change during problem solving. The *domain theory* represents static aspects of the state or problem, e.g., (*block A*), (*connects door*12 *room*1 *room*2). It also includes the *operator definitions*, which describe the problem-solving operators for the domain in a STRIPS-like representation conducive to goal regression.

Although the domain theory may suffice to explain success, it may not explain failures. Therefore, a domain-specific *impossibility theory* is used to explain why a given state fails to satisfy the current goal. This theory specifies pairs A and B such that A is a possible goal, B is a conjunctive condition, and A and B cannot hold simultaneously.[7] The explanation takes the form $current\text{-}goal(A) \wedge B$, where B is true in the current state. For the blocks world, FAILSAFE-2 is given 13 such rules, for example.

$$failure(?State) \leftarrow current\text{-}goal(on\ ?X\ ?Y) \wedge (on\ ?X\ ?Z) \wedge (\neq\ ?Y\ ?Z)$$

This rule can be paraphrased as "?X is on the wrong block." It reflects the fact that it is impossible (in standard blocks world) for block ?X to be simultaneously on block ?Y and on some other block ?Z. Here, variable names start with ?, and the right-hand side of the rule implicitly refers to the current state, ?*State*.

It should be emphasized that FAILSAFE-2 does *not* use the impossibility theory to *detect* failures, but only to *explain* failures that have already been declared by the heuristics used to detect under-constrained search. Thus the impossibility rules implicitly presume that such a failure has already occurred. Otherwise, they would cause FAILSAFE-2 to declare failure at virtually every step, merely because the current goal was not satisfied yet.

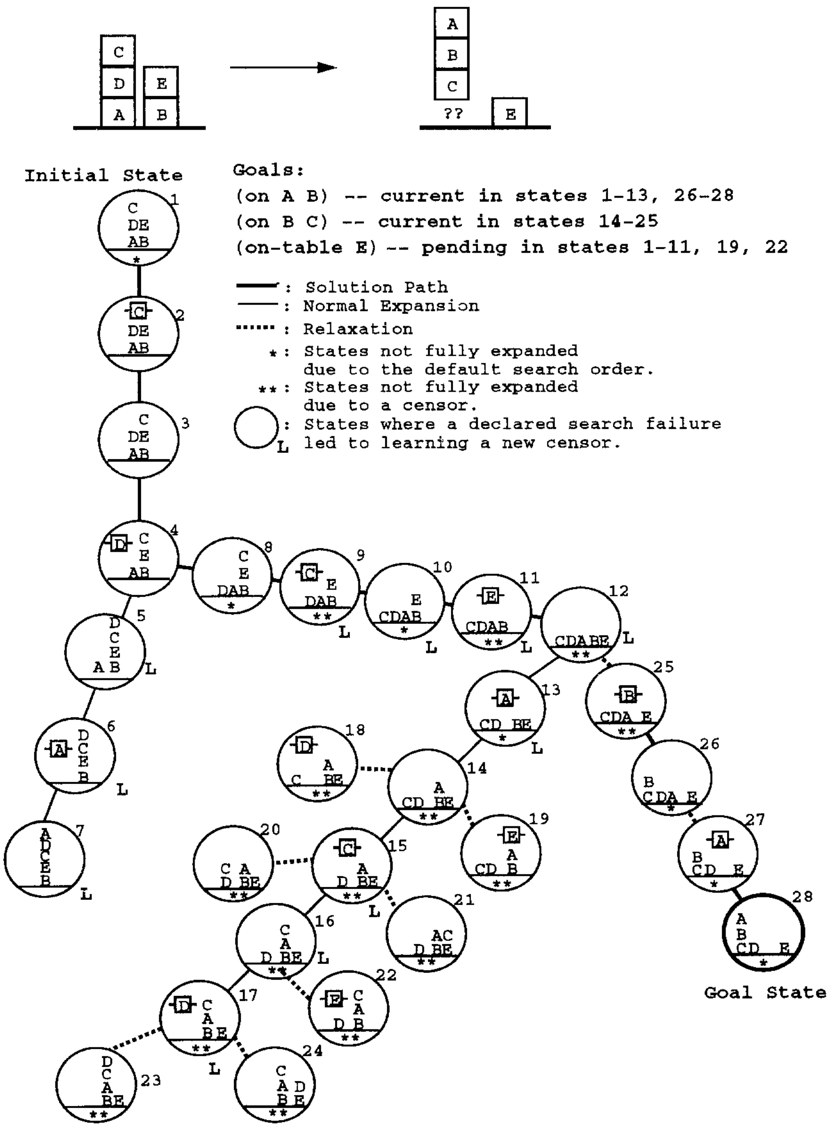
Appendix I lists the domain-specific impossibility theories given to FAILSAFE-2 for some of the experiments reported here; the rest are given by Bhatnagar (1992). Each theory also includes the following general rule:

$$failure(?State) \leftarrow protected\text{-}in\text{-}parent(?G) \wedge not(?G)$$

This rule specifies that a failed state may be explained by the violation of a goal marked as protected in the previous state.

Although the impossibility theory is provided as an input to FAILSAFE-2, it might be possible to extract it automatically from a set of STRIPS operators by using the technique described by Siklossy and Dowson (1977) to discover conditions that cannot hold simultaneously in any legal state. For example, inspection of the operator definitions reveals that every operator that adds the condition (*holding X*) deletes the condition (*clear X*). Thus, no state can satisfy both conditions, assuming the initial state does not.[8]

We shall now describe some episodes of FAILSAFE-2's learning behavior in the course of solving the blocks world problem shown at the top of Figure 3, starting with no control knowledge. The figure also shows the state-space search tree explored by FAILSAFE-2



Goals:
(on A B) -- current in states 1-13, 26-28
(on B C) -- current in states 14-25
(on-table E) -- pending in states 1-11, 19, 22

━ : Solution Path
— : Normal Expansion
······ : Relaxation
* : States not fully expanded due to the default search order.
** : States not fully expanded due to a censor.
◯L : States where a declared search failure led to learning a new censor.

Figure 3. Solution trace of an example problem.

in the course of solving this problem. In this trace, the state selector and operator generator perform simple depth-first search. In each state, the operator generator considers the four blocks world operators *stack*, *unstack*, *pick-up*, and *put-down*, in that order, instantiating them with successive blocks in alphabetical order. The operator generator tries all instantiations of a single operator before going to the next operator. The censors that are learned while solving this problem have the effect of dynamically modifying this initial default search order by deferring censored operators, thereby taking only 28 states to reach the goal; without learning, the process would take 187 states.

In the course of solving this problem, FAILSAFE-2 learns 11 censors and 1 goal-ordering rule. Out of these 11 censors, one gets specialized when it is found to be overgeneral. These censors prevent the application of 56 distinct operators in different states. FAILSAFE-2 overrules 9 of these 56 instances of censoring in order to reach the goal (in other words, there are 56 instances of censoring and 9 instances of censor relaxation). Only 2 of these 9 censor relaxations are useful in reaching the goal. If the censor relaxation mechanism correctly guessed these two instances in the first two attempts, FAILSAFE-2 could solve this problem after visiting 21, instead of 28, states.

Starting at the initial state, the problem solver adopts the first goal (*on A B*) as the current goal (in the absence of any goal-ordering rules that might dictate otherwise) and attempts to achieve it by proceeding depth-first as specified above.[9] The first failure is encountered at state 7. This state is a dead end, since the only operator applicable in this state, namely (*unstack A D*), takes the problem solver back to a previously generated state, namely state 6. Since a dead end is a symptom of under-constrained search (as explained in section 2.2.1), the problem solver declares a failure at state 7, invokes the learner, and learns a censor as described in section 3.1. Subsequent failures (and attendant learning episodes) occur at state 6 and then at state 5, and the problem solver backs up to state 4 and generates 8 through 17, marking as suspended (as indicated by double asterisks) the states where censors prune moves. The problem solver now repeatedly encounters a symptom of over-constrained search—absence of uncensored moves—and accordingly relaxes censors in states 14 through 17 and then 12, allowing it to generate states 18 through 26. At this point, the search once again becomes over-constrained, but the problem solver relaxes a censor in state 26 and goes on to achieve the goal at state 28. Sections 3.1 through 4.5 use selected episodes from this problem-solving trace to illustrate FAILSAFE-2's various learning methods, albeit ordered logically rather than chronologically.

3.1. Learning censors to avoid failures

FAILSAFE-2 learns a censor by 1) explaining a search failure, 2) generalizing the failure condition, 3) assigning blame to the step that caused the condition, and 4) regressing the cause of failure through the operator used in the step.

The learner uses standard explanation-based generalization (Mitchell et al. 1986; Dejong & Mooney, 1986) to explain and generalize the failure, taking as its target concept the definition of failure specified by the impossibility theory. First the learner uses this theory to explain why state 7 does not satisfy the current goal (*on A B*). To explain a failure, the learner chooses an applicable rule from its theory of impossibility; if more than one applies, it chooses one of them at random.[10] In this example, the learner uses the rule presented earlier, and attributes the failure to the fact that block A is on the wrong block, namely, D instead of B as specified in the current goal. We call this explanation the *failure condition*. Actually, the EBG process returns two versions of this explanation, respectively called the *specific* and *general failure conditions*. In our example, the specific failure condition is *current-goal*(*on A B*)$\wedge$(*on A D*)$\wedge$($\neq$ *B D*); the general condition is *current-goal*(*on* ?*X* ?*Y*)$\wedge$(*on* ?*X* ?*Z*)$\wedge$($\neq$?*Y* ?*Z*).

Notice that a more comprehensive explanation of failure would not merely explain why the failure state does not satisfy the current goal, but also why the failure cannot be repaired, i.e., why no operator sequence can reach the goal from that state. PRODIGY and STATIC build such explanations by using richer definitions of failure as their target concepts; consequently, they need to construct more expensive proofs. FAILSAFE-2 avoids this additional expense by assuming that the constraint violated by the failure is *preservable* on at least one solution path. This assumption frees it to build an incomplete explanation that ignores alternative operator sequences. This shorter proof is expressed solely in terms of the path leading directly to the failure. Thus FAILSAFE-2 reduces the expense of EBG, not by modifying EBG itself, but by giving it a simpler target concept.[11]

The next step in learning is to assign blame to the step that caused the failure. The learner simply scans backward from the failure state along the path that led up to it, and blames the (most recent) step that made the specific failure condition become true. In this example, it happens to be the step leading from state 6 to state 7 via the operator (*stack A D*), so it blames that operator. In general, the blamed step may precede the failure by a number of steps, since the rules for *explaining* failure differ from the cirteria for *detecting* it. For example, if the current goal had been (*on B C*) instead of (*on A B*), the failure would still have been declared at state 7, but might been blamed on putting the wrong block on top of C, namely D instead of B, in going from state 4 to state 5.

Finally, the learner regresses the generalized failure condition through the general definition of the blamed operator to derive a new censor on the operator. The general definition of (*stack* ?*X* ?Z) is

$\underline{\text{pre}}$ and $\underline{\text{delete}}$ conditions: (*holding* ?*X*)$\wedge$(*clear* ?Z)
$\underline{\text{add}}$ list: (*on* ?*X* ?Z)$\wedge$(*clear* ?*X*)$\wedge$(*hand-empty*)

The standard regression method (Nilsson, 1980) is used, except that the meta-predicate *current-goal* regresses to itself; that is, regressing a condition of the form *current-goal*(G) back through an operator leaves the condition unchanged, since applying the operator did not change the goal. Thus the new censor will apply whenever the following regressed generalized failure condition is true:

current-goal(*on* ?*X* ?*Y*)$\wedge$(*holding* ?*X*)$\wedge$(*clear* ?Z)$\wedge$($\neq$?Z ?*Y*)

Since the second and third conjuncts are already part of the operator precondition for (*stack* ?*X* ?Z), the learned censor takes effect only when the current goal is to stack block ?*X* on some block other than ?Z. It can be paraphrased as "if the goal is to put a block on another, then don't stack it on a wrong block."

Censors learned by FAILSAFE-2 become effective as soon as they are learned, and usually prevent the same failure from occurring again. However, FAILSAFE-2 can sometimes learn the same censor more than once. During the relaxation process, FAILSAFE-2 often chooses a wrong censor to relax. Relaxing a wrong censor may cause the repetition of a failure that occurred previously and may consequently cause an existing censor to be relearned. Therefore, FAILSAFE-2 matches the learned censor against its list of existing ones and adds it only if it appears genuinely new.[12]

After creating a censor, the learner instructs the problem solver to resume search at the state where the blamed operator was applied, in this case state 6. As it turns out, efforts to further expand states 6 and 5 (in that order) fail because no more operators apply in those states; the failures lead to two new censors, not described here. The problem solver marks states 5, 6, and 7 *closed* because all operators applicable in them have been considered, and continues the search at state 4, whose further expansion creates state 8.

The new censor prevents the problem solver from stacking block ?*X* on the wrong block ?*Z* whenever the goal is to put block ?*X* on block ?*Y*. This censor is preservable in the sense that at least one solution path never violates it. That is because in the blocks world it is always possible to put block ?*X* on the table rather than stack it on a block other than its destination. However, it is also overgeneral in the sense that it prunes some valid solution paths. The overgenerality stems from using a simplistic explanation of failure. Sometimes this overgenerality causes FAILSAFE-2 to eliminate *all* solution paths, as we shall now see.

3.2. *Learning a bad censor*

FAILSAFE-2 sometimes learns *bad* censors that cut off all paths to the goal. An instance of learning a bad censor occurs at state 13 of the example trace shown in figure 3. At state 13 the current goal is (*on A B*). The learning occurs after the problem solver backtracks from state 14. The problem solver declares a failure at state 13 because all remaining operators are either censored or lead back to a previous state. In particular, stacking block *A* on any block other than *B* violates the censor learned at state 7, and putting it down on the table leads back to state 12.

The learner explains this failure using the following impossibility rule:

$$\textit{failure}(?\textit{State}) \leftarrow \textit{current-goal}(\textit{on}\ ?X\ ?Y) \wedge (\textit{holding}\ ?X)$$

Thus it attributes the failure to achieve the current goal (*on A B*) to the fact that block *A* is in the hand instead of on the table. It therefore blames the step where (*holding A*) became true, namely, the operator (*pick-up A*) applied in state 12.

The resulting censor blocks the operator (*pick-up* ?*X*) whenever the current goal is (*on* ?*X* ?*Y*). That is, it prohibits picking up a block from the table in order to stack it on another block. This is a bad censor—one that can prune all paths to the goal.

Indeed, after some additional search, FAILSAFE-2 eventually runs out of states to expand, and relaxes censors in states 14, 15, 16, 17, and 12 in that order, following the heuristics described in section 2.2.2. These relaxations generate states 18 to 25, but learned censors prevent further expansion of all these states except state 25. Here a newly learned goal-ordering rule puts subgoal (*on B C*) first, and it is achieved in state 26. The desired operator (*pick-up A*) is now blocked by another bad censor, leaving no open states and thereby causing the problem solver to detect over-constrained search. The relaxation heuristics select state 26 because it is the shallowest state with two satisfied subgoals. This relaxation enables the problem solver to reach the goal at state 28.

Whenever the current subgoal in a state is satisfied in the state that results by applying an operator to it, FAILSAFE-2 sees it as an opportunity to specialize some overgeneral censor that might have hindered progress toward the subgoal. Such a situation occurs in states 27 and 28 where the current subgoal in state 27, (*on A B*), becomes true in state 28. As a result, FAILSAFE-2 invokes the learner at state 28 to check if a censor needs to be specialized. The specialization mechanism notices that at state 26, the operator (*pick-up A*) was incorrectly censored. We now explain how the offending censor is specialized to prevent this mistake from recurring.

3.3. Specialization of bad censors

Whenever the problem solver achieves a subgoal, it invokes the learner to see if any censors need to be specialized. In our example, the learner is invoked when the subgoal (*on A B*) is achieved at state 28. Although in our example, state 28 happens to be a goal state for the entire problem, in general the learner is invoked as soon as any subgoal is achieved, not just the entire goal. This ability to learn from partial successes, not just complete ones, is important for adaptive learning.

The problem solver passes to the learner the subpath in which the subgoal was the current goal. In our example, the subpath starts at state 26, where the current goal became (*on A B*); in the preceding state, the current goal was (*on B C*), thanks to a learned goal-ordering rule to be described in section 4.4.

If any step along this subpath was censored and then relaxed, the responsible censor is specialized to keep it from blocking similar subpaths in the future. In our example, the step (*pick-up A*) was censored by the bad censor described in section 3.2.

To specialize a censor, the learner regresses the current goal through the subpath that achieved it, and makes the resulting condition an exception to the censor's precondition. Both the goal and the operator sequence are generalized in order to yield a broader exception. In our example, the subpath consists of the two-step sequence

[(*pick-up A*),(*stack A B*)]

Its regressed precondition is

(*clear A*)∧(*clear B*)∧(*on-table A*)∧(*hand-empty*)

Thus the generalized sequence is

[(*pick-up ?X*),(*stack ?X ?Y*)]

The learned exception to the censor occurs when

$$(\textit{clear } ?X) \wedge (\textit{clear } ?Y) \wedge (\textit{on-table } ?X) \wedge (\textit{hand-empty})$$

The operator precondition for (*pick-up* ?*X*) already includes all but the second of these conditions. Thus the learned exception simplifies to (*clear* ?*Y*), and its negation is added to the censor's precondition. The resulting specialized version of the censor blocks the operator (*pick-up* ?*X*) only when the goal is (*on* ?*X* ?*Y*) and (*not*(*clear* ?*Y*)) holds. It can be paraphrased as "Don't pick up a block until its destination is clear." Unlike the initial, overgeneral version of the censor, the specialized version is preservable, since picking up a block from the table in order to stack it on a second block is useless until the second block is clear.

It may be clarifying to point out how the specialization process applies—or does not apply—to the other two goals in our example. When the problem solver achieves the goal (*on B C*) at state 26, it passes to the learner the subsequence during which (*on B C*) was the current goal. However, this subsequence begins at state 25, since the preceding state 12 has a different current goal. Since no censors were relaxed in the subsequence, there are none to specialize. As for the third goal, (*on-table E*), it is achieved as a side effect of the transition from state 11 to state 12. Since it is not the current goal, achieving it does not invoke the learner.

4. Refinements to FAILSAFE-2's learner

Besides the basic mechanisms described so far for learning, relaxing, and correcting censors, FAILSAFE-2 learns to take shortcuts to achieve goals; to avoid irrelevant or premature operators; and to reorder goals so as to protect preconditions that may help achieve pending goals. We now illustrate these refinements to FAILSAFE-2's learner, concluding with a summary of both the basic mechanisms and the refinements. For further details, see Bhatnagar (1992).

4.1. Learning macros from exceptions

Section 3.3 described how FAILSAFE-2 corrects an overgeneral censor when it finds an operator sequence that achieves the current goal. The weakest precondition of this sequence is recorded as a general exception to the censor. Although specializing the censor permits instances of this exception thereafter, it is a rather indirect way to learn from success. If an exception holds, it indicates that the entire operator sequence can be applied and moreover will achieve the current goal. Therefore, FAILSAFE-2 also learns the exception in the form of a macro for achieving the goal directly.

In our example, FAILSAFE-2 makes the generalized two-step sequence leading to state 28 into a macro

[(*pick-up* ?*X*),(*stack* ?*X* ?*Y*)]

for achieving the goal (*on* ?*X* ?*Y*), with the precondition (*holding* ?*X*)∧(*on* ?*Z* ?*Y*)∧(*clear* ?Z). Using this operator sequence is a shortcut in the sense that when it is applied, it eliminates some operator generation and selection. However, FAILSAFE-2 does not eliminate the operator applications themselves. Doing so would require computing and storing the macro's net effect, namely,

precondition: (*on-table* ?*X*)∧(clear ?*X*)∧(*clear* ?*Y*)∧(*hands-empty*)
delete: (*on-table* ?*X*)∧(*clear* ?*Y*)
add: (*on* ?*X* ?*Y*)

Whenever the exception arises, FAILSAFE-2 opportunistically applies the whole macro to achieve the current goal. Thus, suppose a similar situation were to arise at some future point, where the current goal was (*on A B*), the operator generator had just proposed (*pick-up A*), and the problem solver was checking its censors. It would find the learned macro indexed under the specialized censor, and apply not only (*pick-up A*) but the rest of the macro as well, thereby achieving the current goal.

Although macro learning is not novel, our version of it differs from previous methods in its selective learning and use of macros. Indiscriminate macro learning can degrade problem-solving performance due to the cost of testing the applicability conditions of too many learned macros (Minton, 1985). Past research has investigated heuristics for restricting when to learn macros (Iba, 1989) and which learned macros to retain (Minton, 1990). It should be emphasized that FAILSAFE-2 learns macros only as exceptions to overgeneral censors, tests their applicability conditions only when the original overgeneral censor would have applied, and does not apply all macros in all states, just the ones whose perconditions are fortuitously found to hold while testing the censors, and which are therefore known to achieve the current goal. Due to this opportunistic use of macros, FAILSAFE-2 can achieve some subgoals with negligible extra cost.

4.2. Learning censors to avoid irrelevant actions when blame assignment fails

FAILSAFE-2's method for learning from a failure relies on being able to blame the particular operator that made the failure condition true. In cases where this condition has been true starting from the initial state, blame assignment fails, and FAILSAFE-2 resorts to chronological backtracking. However, it does try to determine why the operator just prior to the failure was *irrelevant*, that is, did not prevent the failure. FAILSAFE-2 learns censors to avoid such irrelevant actions.

An episode of learning such a censor occurs when FAILSAFE-2 backtracks to state 5, where the goal is (*on A B*). The problem solver declares failure here because the only untried operator, namely (*unstack D C*), leads to an already existing state, namely state 4. The learner explains that state 5 does not satisfy the current goal because block *A* is on the table instead of on block *B*. That is, the specific failure condition is

current-goal(*on A B*)∧(*on-table A*)

However, the learner is unable to assign blame, since A has been on the table right from the start, without being put there by any step. It therefore tries to determine why the operator applied just before the failure, in this case (*stack D C*) at state 4, was irrelevant.

An operator is irrelevant to a goal if it does nothing to achieve it. In the blocks world domain, for example, picking up a block from the table does not help in achieving the goal (*on-table ?block*). Similarly, in Minton's scheduling world, painting an object does not help put a hole in it. A planner can improve its efficiency by learning to ignore such irrelevant operators.

FAILSAFE-2's learner uses a simple heuristic cirterion of irrelevance: it tries to identify the conditions under which an operator does not *directly* influence a failure condition, and learns not to apply that operator when these conditions hold. We say that an operator *directly influences* the failure condition of a goal if (under some bindings) it achieves that goal, deletes a positive conjunct occurring in the failure condition, or adds a negated conjunct.

In the example here, the failure condition is

$$\textit{current-goal}(\textit{on } A\ B) \wedge (\textit{on-table } A)$$

The operator (*stack ?X ?Y*) can achieve the goal (*on A B*) only when the variable *?X* is bound to A and the variable *?Y* is bound to B, and it can never delete the condition (*on-table A*). So FAILSAFE-2 learns a censor that rejects (*stack ?X ?Y*) when the current goal is (*on ?A ?B*) and block *?A* is on the table, unless *?X* = *?A* and *?Y* = *?B*. This censor eliminates any application of *stack* that is irrelevant to the generalized failure condition, i.e., does not achieve the current goal (*on ?A ?B*). The censor is enforced only when block *?A* is on the table—at which point it is not ready to stack, since (*holding ?X*) is a precondition of the operator (*stack ?X ?Y*). Thus this censor has the effect of preventing *any* use of *stack* when the block one really wants to stack is still on the table. Since *stack* is indeed irrelevant to the current goal in that case, the censor has a desirable effect.

In general, if the failure condition has the form $\textit{current-goal}(P) \wedge Q$, FAILSAFE-2 learns a censor that rejects the operator whenever the current goal is P, condition Q holds, and the operator is not directly relevant to the failure condition, that is, it neither immediately achieves P nor falsifies Q.

This definition of irrelevance is shortsighted; just because an operator does not *immediately* prevent a failure does not always make it truly irrelevant. For example, an operator that does not directly achieve a goal might achieve a precondition of another operator that does. Ignoring such indirect influences sometimes causes FAILSAFE-2 to learn unpreservable censors. Unfortunately, calculating such influences precisely (as opposed to estimating them conservatively, as in ABSOLVER (Mostow & Prieditis, 1989)) can be expensive. Instead, FAILSAFE-2 does fast, incomplete reasoning to learn censors that may turn out to be unpreservable; if they do, it eventually specializes them.

FAILSAFE-2's approximate method to determine irrelevance can be compared with a more conservative method employed in the ALPINE system (Knoblock, 1990). Given the description of a domain, ALPINE builds a hierarchy of abstraction spaces for the domain. The algorithm employed by ALPINE guarantees that the goals belonging to higher levels of abstraction can be solved first and then held invariant while the remaining goals are solved. This means that the operators required to solve the goals in lower abstraction levels are, in some sense, irrelevant to the goals in the higher abstraction levels. The main difference between FAILSAFE-2 and ALPINE is that while FAILSAFE-2 attempts to learn conditions under which an operator is irrelevant to a goal, ALPINE attempts to classify the goals into different abstraction levels such that the goals in the lower levels are irrelevant to the goals at the higher levels.

4.3. Enhancing explanations of failure

The example in section 3.2 of learning a bad censor illustrates the negative consequences of a simplistic explanation of failure. The failure condition *current-goal(on A B)*$\wedge$*(holding A)* at state 13 merely says why the state where failure was declared does not satisfy the goal. It does not say why the problem solver cannot find a path leading from the failed state to the goal. Such an explanation is overgeneral in that it ignores the reasons why alternative operators cannot be applied. In cases where they can, a rule based on this overgeneral explanation may prune a potentially successful step.

In contrast, a complete explanation of failure explains a failed state (at least a dead end) by showing why none of the known operators can be applied in that state, and why there is no other path from the blamed step to the goal. But such an explanation is overspecific in that it covers operators that would not lead toward the goal even if they did apply. Moreover, it can be intractable to construct, since it must show why exhaustive search would fail.

The learner therefore uses a compromise approach that falls between the above two extremes: it explains why *one* operator does not lead to the goal. It selects a *direct* operator, defined as one capable of achieving the current goal in one step when it is applicable. The learner *enhances* the simplistic explanation of failure described in section 3.1 by appending an explanation of why this direct operator cannot be applied. That is, it appends all the unsatisfied preconditions of the direct operator. The process of enhancement can be viewed as incorporation of some means-ends analysis into forward search.

The enhanced explanation is still incomplete, in two respects. First, it only covers one direct operator; if there is more than one, the learner chooses one at random. Thus the explanation is "horizontally incomplete" in that it omits alternative operators. By "horizontally incomplete" we mean that the explanation omits some branches of the proof tree. Second, it ignores paths that would achieve the goal in more than one step. Thus the explanation is "vertically incomplete" in that it only looks ahead one step beyond the current state. By "vertically incomplete" we mean that some branches of the proof tree are not expanded beyond a certain depth in this case 1.

The enhancement feature can be switched on or off in order to measure its usefulness, as reported in section 5. It was turned on for the trace in figure 3, but to simplify the presentation we have until now presented explanations in unenhanced form. We now illustrate how the failure condition at state 6 is enhanced. (We will consider state 13 again shortly.)

For the goal *(on A B)*, there is only one direct operator, namely *(stack A B)*, whose preconditions are *(holding A)*$\wedge$*(clear B)*. In state 6, block *A* is being held, but block *B* is buried under other blocks, so the failure is blamed on block *B* not being clear. The learner therefore *enhances* the failure condition by appending the negation of the unsatisfied precondition:

current-goal(on A B)$\wedge$*(holding A)*$\wedge$*(not(clear B))*

As before, the failure is blamed on the operator (*pick-up A*) that made the failure condition true, and the generalized failure condition is regressed through the blamed operator to create a new censor. However, the censor learned from the enhanced explanation has a more discriminating precondition than the bad censor described in section 3.2. Like the bad censor, it prevents the application of (*pick-up ?X*) when the current goal is (*on ?X ?Y*)—but *only* when the block ?*Y* is not clear. This censor can be paraphrased as "If you want to put block ?*X* on block ?*Y*, don't pick ?*X* up from the table until ?*Y* is clear."

Why doesn't enhancement prevent learning a bad censor at state 13? The unenhanced failure condition is the same as at state 6:

$$\textit{current-goal}(\textit{on } A\ B) \wedge (\textit{holding } A)$$

However, at state 13, the direct operator (*stack A B*) has no unsatisfied preconditions, and consequently enhancement leaves the failure condition unchanged.

In the example at state 6, enhancement of the explanation enables FAILSAFE-2 to learn a censor that prevents premature application of an operator, that is, application of an operator before it can achieve the current goal. We next see how enhanced explanations also enable FAILSAFE-2 to learn better goal orderings.

4.4. Learning rules to reorder goals

A deeper analysis of failure in our example has to do with a faulty goal ordering: the problem solver is trying to achieve (*on A B*) before (*on B C*).[13] Even if block *B* were clear, stacking *A* on *B* would still lead to failure, because (*on A B*) would have to be undone in order to achieve (*on B C*). We now explain how at state 17, FAILSAFE-2 learns a rule to reorder these two goals.

The problem solver declares failure at state 17 because the previously learned censors prevent stacking block *D* on either *C* or *E*. The learner explains that state 17 does not satisfy the current goal (*on B C*) because *B* is on the table and not on block *C*:

$$\textit{current-goal}(\textit{on } B\ C) \wedge (\textit{on-table } B)$$

Enhancement chooses (*stack B C*) as the (only) direct operator for the current goal.

The learner now checks to see if any unsatisfied precondition of the direct operator contradicts any protected goal. In state 17, there is only one unsatisfied precondition, namely (*holding B*), and only one protected goal, namely (*on A B*). A precondition *P* is considered to *contradict* a goal *G* if the impossibility theory contains a rule of the form

$$\textit{failure}(?\textit{State}) \leftarrow \textit{current-goal}(G) \wedge P$$

or

$$\textit{failure}(?\textit{State}) \leftarrow \textit{current-goal}(G) \wedge P \wedge Q$$

In this example, the impossibility theory for blocks world does indeed contain the rule

$$\textit{failure}(?\textit{State}) \Leftarrow \textit{current-goal}(\textit{on}\ ?X\ ?Y) \wedge (\textit{holding}?Y)$$

When the learner finds such a contradiction, it learns a goal-ordering rule to defer the protected goal until after the current goal. In this example, the learned rule says to achieve (*on* ?*X* ?*Y*) after (*on* ?*Y* ?Z). Since this rule is derived from an impossibility axiom of the first form above, it applies unconditionally to any new state where there are two such goals yet to be satisfied. In contrast, goal-ordering rules derived from impossibility axioms of the second form apply only when the condition *Q* holds.

Goal-ordering rules are used by the problem solver to order the pending goals in a newly created state before selecting which goal to pursue next. Thus if the goals in a given domain are not serializable, learned goal-ordering rules could theoretically create a deadlock in the form of a cycle in the overall ordering. FAILSAFE-2 therefore relies on the user to declare whether the domain's goals are serializable, and it refrains from learning goal-ordering rules in non-serializable domains.

In general, FAILSAFE-2 learns goal-ordering rules of the form "if *condition* then achieve *goal* 1 before *goal* 2." Such rules are used by the problem solver to order the pending goals before selecting which one to achieve next. In the simple example above, *goal* 1 is (*on* ?*B* ?*C*), *goal* 2 is (*on* ?*A* ?*B*), and *condition* is identically true. However, in principle the *condition* can refer to the current state and to protected and pending goals. (There is no current goal yet at the point where such rules are used.)

4.5. Learning to protect useful preconditions

One more type of learning made possible by enhancement of failure conditions extends beyond protecting goals to protecting preconditions of operators for achieving goals. This type of learning occurs when a failure condition is enhanced to include an unsatisfied precondition *P* of a direct operator for the current goal *G*, and a previous operator *O* is blamed for deleting that precondition.

An example of this situation arises when failure is declared at state 15. At this point the current goal is (*on B C*), which has only one direct operator, namely (*stack B C*), both of whose preconditions are unsatisfied. The first precondition, (*holding B*), has never been true, but the second precondition, (*clear C*), has just been deleted by applying (*pick-up C*) to state 14. The learner therefore learns a censor on (*pick-up* ?*Y*) to prevent it in the future from deleting the precondition (*clear* ?*Y*) of (*stack* ?*X* ?*Y*) whenever the current goal is (*on* ?*X* ?*Y*). That is, if the current goal is to put block ?*X* on block ?*Y*, and block ?*Y* is already clear, then do not pick it up from the table.

In general, the learner will learn a censor that prevents operator *O* from deleting *P* when the goal is *G*. The censor helps keep open the possibility of using the direct operator to achieve *G*. In this example, *G* was already the current goal when operator *O* deleted *P*. If the problem solver had been pursuing some earlier goal at that point, the learned censor would instead apply when *G* is a pending goal.

In either case, the idea here is to incorporate some lookahead so as to keep open the possibility of achieving goal *G*. The method assumes that goal *G* will be achieved by the direct operator—or at least it tries to protect that option. There may be multiple operators that achieve *G*, but one is enough. On the other hand, for each precondition *P* of the direct operator, and each operator *O* that can delete *P*, it may in principle be necessary to learn a separate censor. However, such censors are learned only when *P* turns out to be unsatisfied in practice.

FAILSAFE-2's methods for learning to reorder goals and to protect preconditions of direct operators approximate PRODIGY's method for learning from goal interactions (Minton, 1988a). A key difference is that while PRODIGY learns by building exhaustive proofs of various types of goal interactions, FAILSAFE-2 takes a heuristic approach and learns from incomplete proofs. By protecting useful preconditions, FAILSAFE-2's censors provide the problem solving operators with lookahead capabilities similar to (but weaker in scope than) the ones available to the *block-preventing patching* oerators described by Voigt and Tong (1989). These operators are equipped with lookahead components that attempt to predict and avoid long-term negative consequences of applying the operators.

4.6. Summary

We now summarize FAILSAFE-2's learning methods in terms of the conditions under which each one is used, the control knowledge it produces, and the simplifying assumptions on which it is based.

- If the search appears under-constrained, declare a failure and explain why the state fails to satisfy the current goal. Ignore reasons why the problem solver cannot reach the goal from that state.
- If the search appears over-constrained, heuristically choose a suspended state and censor to relax in that state, and apply the censored operator. The heuristic assumes that states with the most satisfied goals are likeliest (but not certain) to be closest to the goal.
- When explaining a failure, enhance the explanation to show why some (randomly chosen) direct operator for the goal cannot be applied. Ignore alternative operators that might lead to the goal, as well as alternative instantiations of the operator that could lead to the goal in more than one step.
- When a failure is blamed on an operator, censor the operator to avoid similar failures. Ignore alternative paths to the goal.
- When a failure cannot be blamed on an operator, censor the last operator to avoid actions not directly relevant to the current goal. Ignore indirect relevance.
- When the current goal is achieved by relaxing a censor, derive a generalized exception to the censor from the successful subpath, specialize the censor to exclude the exception, and learn a macro to achieve the goal when it applies. Ignore alternative paths that might have achieved the goal without relaxing the censor.
- When a protected goal contradicts a precondition of a direct operator, learn a goal-ordering rule to defer the protected goal until after the current goal. Ignore alternative paths to the goal, and assume the two goals are serializable.

- When a failure is explained by an unsatisfied precondition of a direct operator, censor the operator to prevent its premature application. Ignore its possible indirect contribution toward achieving the goal.
- When a failure is blamed on clobbering a protected goal, censor the responsible operator to protect the goal. Assume serializability of goals.
- When a failure is blamed on deleting a precondition of a direct operator for a current or pending goal, censor the responsible operator to protect the precondition. Ignore alternative operators that could achieve the goal.

5. Experimental results

We carried out experiments to answer some questions about FAILSAFE-2's effectiveness. Does the learned knowledge lead to performance improvement in diverse domains? How does FAILSAFE-2's performance compare with PRODIGY's? Is it really adaptive? Can FAILSAFE-2's learning method improve performance of diverse problem solving methods? Do its extra features really help? We classify these questions under four different hypotheses:

- **Effectiveness hypothesis:** The effectiveness hypothesis asserts that by exploiting the preservability of path contraints, FAILSAFE-2 can improve the performance of satisfying state-space search in diverse domains. This hypothesis predicts *across-task transfer* of learned knowledge from a set of training problems to a separate set of test problems.
- **Adaptiveness hypothesis:** FAILSAFE-2 interleaves learning with problem solving by declaring failures early and not waiting to learn until exhaustion of possibilities. The adaptiveness hypothesis asserts that by interleaving learning with problem solving, FAILSAFE-2 can solve more problems than without learning, given the same computational resources. This hypothesis predicts *within-task transfer* of learned knowledge in the course of solving individual problems.
- **Generality hypothesis:** Various problem-solving methods perform search by applying problem-solving operators in the forward direction. The generality hypothesis asserts that FAILSAFE-2 can improve performance of diverse problem solvers by selectively pruning the forward operators they generate.
- **Internal features hypothesis:** FAILSAFE-2 uses various refinements to its basic learning method. It uses exceptions to the learned censors as macros (section 4.1), learns irrelevancy censors and enhances the failure conditions (sections 4.2 and 4.3), and uses the forced learning and forced relaxation heuristics (section 2.2.1). The internal features hypothesis asserts that these refinements indeed increase FAILSAFE-2's effectiveness.

The following four sections report the results of experiments conducted to these these hypotheses.

5.1. Testing the effectiveness hypothesis

We tested FAILSAFE-2's effectiveness in six different easy-to-model domains, including the three domains of PRODIGY (Minton, 1988a)—the standard blocks world, the scheduling world, and the STRIPS robot world. In addition to these three domains, we also tested FAILSAFE-2 in the standard 8-Puzzle domain, the augmented blocks world domain (Etzioni, 1990b),[14] and the Think-A-Dot puzzle domain[15] (Prieditis, 1990; Bhatnagar, 1992). We chose the last three domains because the operator definitions in these domains involve recursion, and PRODIGY reportedly (Etzioni, 1990b) cannot acquire effective control rules for such domains.

To test FAILSAFE-2's effectiveness in a domain, we compared its performance in *before-learning* and *after-learning* execution modes. In the *before-learning* mode, FAILSAFE-2 solves problems from a *test* set of problems without any learned control knowledge. In the *after-learning* mode, FAILSAFE-2 solves problems from the same *test* set of problems but uses control knowledge learned previously in a separate *training* mode. In both before-learning and after-learning modes, FAILSAFE-2 solves problems without learning any new control knowledge. In the training mode, FAILSAFE-2 solves problems from a different *training* set of problems. In this mode, FAILSAFE-2 begins to work on the first training problem without any learned control knowledge. As it solves the problems, it learns new censors and goal-ordering rules and uses the learned censors and rules. It also specializes the censors that are found to be overgeneral. After the computational resources assigned to a problem are exhausted or the problem is solved, FAILSAFE-2 transfers the learned censors and goal-ordering rules to the next problem, and continues the process until all training problems are attempted. At the end of the training run, FAILSAFE-2 returns the learned control knowlege for use in the *after-learning* mode.

In all the effectiveness experiments, FAILSAFE-2 used depth-first forward search as the default search method. Our experiments measured FAILSAFE-2's problem-solving performance, both in terms of the number of visited states while solving a given problem and in terms of problem-solving time. While solving a problem, FAILSAFE-2 was assigned fixed resource limits (maximum CPU time allowed on each problem, and maximum number of states visited). We used the number of problems solved within the assigned resource limit as an additional measure of performance. As Segre et al. (1991) point out, comparisons based on these measures may be sensitive to the values of the resource limits, and inconclusive when some of the problems are not solved within the assigned resource limits.

Figure 4 shows FAILSAFE-2's performance in the blocks world domain. The horizontal axis of the graph in the figure represents the total number of problems attempted. The vertical axes, respectively, show the cumulative states visited and CPU time taken in solving these problems. In this experiment, the training problem set consisted of the 50 even-numbered problems from the problem set used by PRODIGY (Minton, 1988a). The test problem set consisted of the remaining 50 problems. In the training mode, FAILSAFE-2 learned 53 censors. As a result of learning, FAILSAFE-2 solved all 50 test problems after expanding 519 states in 259 CPU seconds. Before learning, it took at least 22,721 states and 8872 CPU seconds—not counting the states and time that it would have taken to solve the five problems left unsolved within the given resource limits.

As a benchmark, we compared FAILSAFE-2's after-learning performance with PRODIGY+STATIC's. By PRODIGY+STATIC, we actually mean an instantiation of FAILSAFE-2 in which PRODIGY generates operators using the control rules learned by STATIC. In this instantiation, FAILSAFE-2 expands states by using PRODIGY+STATIC as a subroutine to generate forward operators. FAILSAFE-2 passes the description of the state that is to be expanded along with the list of unsatisfied subgoals in this state to PRODIGY+STATIC. When PRODIGY+STATIC finds an applicable forward operator, instead of applying this operator itself, PRODIGY+STATIC returns the operator to FAILSAFE-2. FAILSAFE-2 then applies the returned operator in the forward direction. If PRODIGY+STATIC fails to find a forward operator, it informs FAILSAFE-2, and FAILSAFE-2 assumes that there are no more applicable operators in the state. This configuration is not quite as efficient as running PRODIGY itself (since the goal stack gets recomputed each time PRODIGY is invoked). Therefore, our CPU times do not provide an exact comparison with PRODIGY itself. Rather, they should be interpreted as a reality check to make sure that the comparisons based on number of states reasonably reflect the actual time costs. By this criterion, FAILSAFE-2's after-learning performance was comparable to PRODIGY+STATIC's. To solve the same 50 problems, PRODIGY+STATIC took 570 states and 181 CPU seconds.

In the scheduling world, the training and test sets also consisted of 50 problems each.[16] In this domain, FAILSAFE-2's after-learning performance was substantially better than its before-learning performance. As a result of learning, FAILSAFE-2 could solve 44 of the 50 test problems after expanding 1779 states in 6099 CPU seconds. Before learning it could solve only 24 problems after exploring 13,566 states in 23,157 CPU seconds.

FAILSAFE-2's after-learning performance was worse than PRODIGY+STATIC's. Like FAILSAFE-2, PRODIGY+STATIC also solved 44 (not the same ones) out of the 50 test problems, but took only 195 states and 175 CPU seconds. The performance graphs in this domain were qualitatively similar to the performance graphs in the blocks world domain shown in Figure 4. To save space, we do not present these graphs here, but they are shown in Bhatnagar (1992).

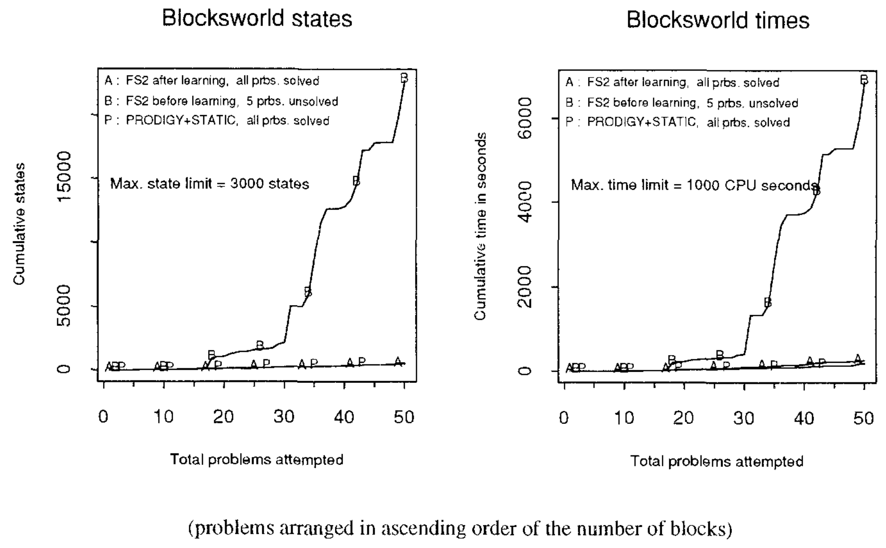


Figure 4. FAILSAFE-2's performance improvement in the blocks world domain. (Note: In this and subsequent figures, FAILSAFE-2 is abbreviated as FS2.)

We now present results from a domain where FAILSAFE-2 failed to improve the performance of the forward-chaining problem solver. Figure 5 shows the results from the STRIPS robot world domain. In this domain, we again used the problems from PRODIGY's problem set with the even-numbered problems for training and the odd-numbered ones for testing. In the training run, FAILSAFE-2 learned 283 censors. These censors were then used to solve the test problems. Though FAILSAFE-2 solved eight more problems in the after-learning mode than in the before-learning mode, the performance improvement appears to be small. We used Student's t distribution (Smith, 1992) to find the confidence level of the hypothesis that FAILSAFE-2's after-learning performance was better than its before-learning performance in terms of the number of visited states and the CPU time taken in solving the problem. These confidence levels were only 0.80 and 0.63, respectively. These low confidence levels indicate that the after-learning performance was not substantially better than the before-learning performance.[17]

In this domain, PRODIGY+STATIC performed far better than FAILSAFE-2. In the after-learning case, FAILSAFE-2 spent 15,795 CPU seconds and visited 2743 states in attempting the test problems, whereas PRODIGY+STATIC solved these problems after spending only 1000 CPU seconds and visiting 878 states.

Why did FAILSAFE-2 not perform as well as PRODIGY+STATIC in the STRIPS robot world? In this domain, achieving certain subgoals requires applying relatively long sequences of operators—for example, the key to a particular room may have to be fetched from a distant room. In such situations, a means-ends analysis problem solver like PRODIGY's can be expected to perform better than a forward-chaining problem solver like the one used by FAILSAFE-2 in this experiment.

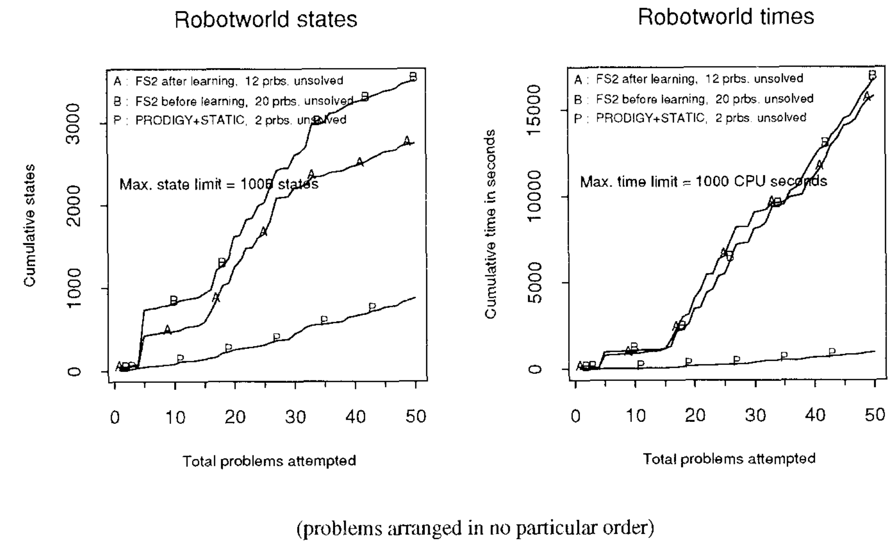
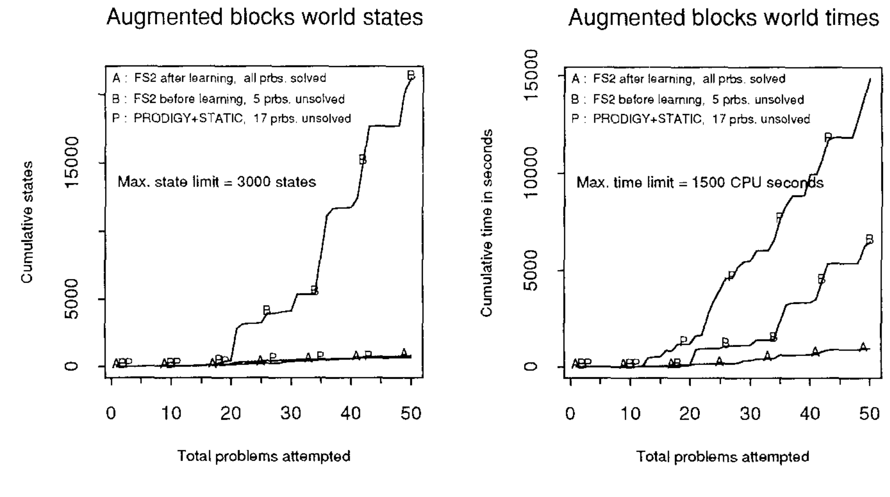


Figure 5. FAILSAFE-2's performance improvement in the STRIPS robot world domain.

We shall now present results from three domains in which operator definitions involve recursion. We conjecture that recursion in operator definitions hampers not only the explanation-based learning process but also the backward means-ends analysis search process. Thus by exploring the search space in the forward direction, declaring failures early without exhaustive search, and building incomplete explanations of them, FAILSAFE-2 should achieve better performance in many recursive domains than other learning problem solvers.

Figure 6 shows FAILSAFE-2's effectiveness in the augmented blocks world. In this experiment, the training and test sets used were identical to the problem sets used in the blocks world domain. During the training run, FAILSAFE-2 learned 115 censors. In the after-learning case, FAILSAFE-2 solved all the test problems well within the given resource limit of 1500 CPU seconds per problem. In the before-learning case, it failed to solve five problems. On the same set of test problems, PRODIGY+STATIC's performance was worse than FAILSAFE-2's after-learning performance. In fact, in this experiment PRODIGY+STATIC's performance was even worse than FAILSAFE-2's before-learning performance, both in terms of the number of problems that it could not solve (17 versus 5) and in terms of problem solving time (14,843 versus 6483 CPU seconds). In the graphs of figure 6, PRODIGY+STATIC seems to take nearly as few states as FAILSAFE-2 after learning. However, the low number of states explored by PRODIGY+STATIC in the augmented blocks world is misleading. In this domain, the backward-chaining process of PRODIGY falls into long (or infinite) calculations and, as a result, exhausts the maximum allowed problem-solving time without expanding many states.

It is instructive to compare the relative performance of the forward-chaining and means-ends analysis methods in the blocks world and augmented blocks world domains (see figures 4 and 6). Due to the additional recursive problem solving operator, PRODIGY+STATIC



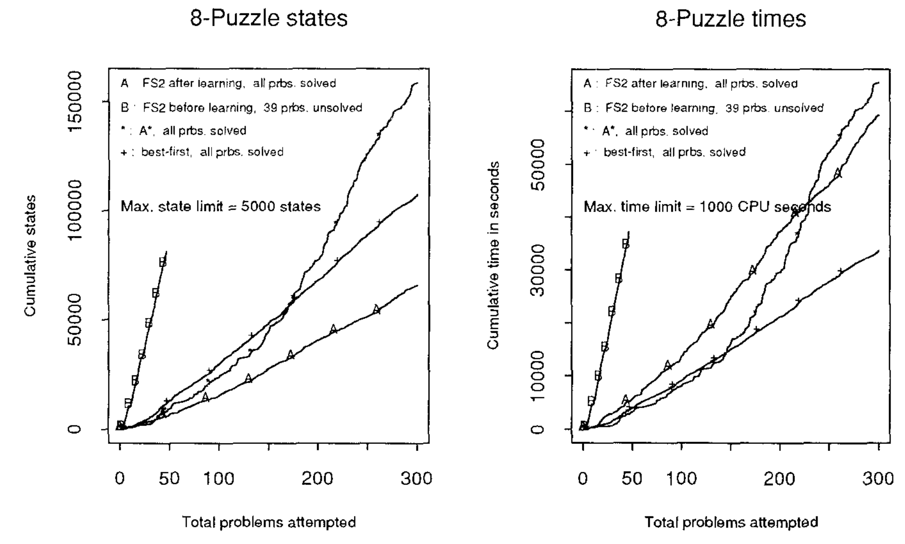
(problems arranged in ascending order of the number of blocks)

Figure 6. FAILSAFE-2's performance improvement in the augmented blocks world domain.

took at least 82 times more CPU time in solving the same 50 test problems in augmented blocks world than in standard blocks world (14,843 versus 181 CPU seconds). In contrast, FAILSAFE-2 took only 3.5 times more CPU time in augmented blocks world than in standard blocks world on the same 50 problems (921 versus 259 CPU seconds). This observation supports the hypothesis that the additional recursive operator slowed down the backward search process of PRODIGY+STATIC much more than it slowed down the forward search process of FAILSAFE-2.

In the 8-Puzzle domain, we used 50 training problems and 300 test problems. In the training run, FAILSAFE-2 learned 109 censors. These censors were then used to solve the 300 test problems. In this domain, we compared FAILSAFE-2's after-learning performance with its before-learning performance and with that of two heuristic methods. The first heuristic method (A*) uses the Manhattan Distance between a state and the goal, added to the depth of the state (distance from the initial state) to select which state to expand next. Thanks to the admissibility of the Manhattan Distance heuristic, this method is guaranteed to find the shortest possible solution to the problem. The second heuristic method uses the Manhattan Distance of a state from the goal state as a heuristic evaluation function and does a best-first search by always selecting the state with the smallest Manhattan Distance to the goal without regard to its depth. This method sacrifices admissibility to reduce total search cost.

Figure 7 shows these results. In the less interesting before-learning case, FAILSAFE-2 could solve only 8 of the first 47 test problems. We stopped this experiment at the 48^{th} problem because the performance in the remaining problems would have been equally bad. Comparison of the after-learning case with the other two cases is more interesting. In the



(problems arranged in ascending order of the Manhattan Distance from goal)

Figure 7. FAILSAFE-2's performance improvement in the 8-Puzzle domain.

after-learning case, FAILSAFE-2 performed better than both versions of the Manhattan Distance heuristic in terms of the number of visited states (65,712 states for FAILSAFE-2 versus 159,011 for the A* version and 107,534 for the best-first version). As expected, the best-first version performs better than the A* version both in terms of time and states. In terms of time, FAILSAFE-2's performance was comparable to the A* version (the confidence level for the hypothesis that FAILSAFE-2 took less time than A* was only 0.82), but was worse than the best-first version.

FAILSAFE-2 also improved the performance of the forward-chaining problem solver in the Think-A-Dot domain. In this domain, the test-problem set consisted of 150 problems. In the before-learning case, FAILSAFE-2 could solve only 9 of the first 30 test problems. We stopped the experiment after 30 problems were attempted. On the average, FAILSAFE-2 spent 991 CPU seconds and visited 1805 states while attempting each of these 30 problems. In the after-learning case, it solved all 150 of the test problems. On the average, it spent 54 CPU seconds and visited 61 states in solving these problems.

In the Think-A-Dot domain, we could not compare FAILSAFE-2's performance with any other method because we are not aware of any heuristic or method to solve the Think-A-Dot problem effectively. The ABSOLVER system, which discovers heuristics by applying various abstracting transformations (Prieditis, 1990), did not learn any effective heuristics to solve problems in this domain. Further details about the experiments in this domain are available in Bhatnagar (1992).

In two of the three PRODIGY domains, FAILSAFE-2's performance was comparable to PRODIGY+STATIC's (but worse). In the third, FAILSAFE-2 actually performed far worse than PRODIGY+STATIC. FAILSAFE-2 outperformed PRODIGY+STATIC only in the domains with recursive operators. A basic question to ask is whether FAILSAFE-2 outperforms PRODIGY+STATIC in any way other than because of the difference between forward and backward search. As the above experiments show, FAILSAFE-2's after-learning performance is far better than its before-learning performance in the domains with recursive operators. So only a part of FAILSAFE-2's success in these domains can be attributed to the forward search mechanism. FAILSAFE-2's success (in recursive and non-recursive domains) to a great extent comes from the learning mechanism that it employs. In addition, unlike PRODIGY, FAILSAFE-2 also shows adaptive learning behavior, which we will now discuss.

5.2. Testing the adaptiveness hypothesis

The adaptiveness hypothesis asserts that by interleaving learning with problem solving, FAILSAFE-2 can solve more problems than without learning, given the same computational resources. This requires that the cost of learning should not be prohibitive with respect to the cost of problem solving. To test if FAILSAFE-2 satisfies this property, we compared FAILSAFE-2's performance in *adaptive*, *before-learning*, and *after-learning* modes. In *adaptive* mode, FAILSAFE-2 begins to solve every new problem with no learned control knowledge, but learns and uses the learned knowledge as it solves the problem. After the problem is solved, FAILSAFE-2 does *not* pass the learned knowledge to the next problem. The *before-learning* and *after-learning* modes are as described in section 5.1.

We tested the adaptiveness hypothesis in the blocks world, the scheduling world, and the 8-Puzzle domains. Figure 8 shows the results of the adaptiveness experiment in the blocks world. To solve the 50 test problems in this domain, FAILSAFE-2 expanded only 13% as many states in the adaptive mode as in the before-learning mode (2870 versus 22,672 states). Similarly, it took only 30% as much CPU time in the adaptive mode as in the before-learning mode (2113 versus 6856 CPU seconds). Also, FAILSAFE-2 solved three more problems in the adaptive mode than in the before-learning mode (one unsolved problem versus four unsolved problems). As can be expected, FAILSAFE-2's adaptive mode performance was worse than its after-learning performance. In the after-learning mode, it expanded 18% as many states (519 versus 2870 states) and took 12% as much CPU time (259 versus 2113 CPU seconds) as in adaptive mode.

The adaptiveness results in the scheduling world and 8-Puzzle domains are qualitatively similar to the results in the blocks world domain. These results are described in detail by Bhatnagar (1992). It should be noted that these adaptiveness results are for the forward-chaining problem solver. They indicate that FAILSAFE-2's learning method can be used adaptively for a weak method with a high branching factor. The adaptiveness hypothesis is less likely to hold when the forward-branching factor is low to begin with. Similarly, the adaptiveness hypothesis cannot be expected to hold in the domains where FAILSAFE-2's after-learning performance was not any better than its before-learning performance (such as the STRIPS robot world domain).

5.3. Testing the generality hypothesis

The generality hypothesis states that by selectively pruning forward operators, FAILSAFE-2 can improve the performance of diverse problem solvers. We conducted the following experiments to test whether FAILSAFE-2 can improve the performance of methods more efficient than depth-first forward chaining.

- In the 8-Puzzle domain, we tried to improve the performance of the best-first heuristic method described in section 5.1. The method uses the Manhattan Distance of a state from the goal as the heuristic evaluation function. The base problem solver in this case always selects the state with the minimum Manhattan Distance for expansion and applies the first applicable forward operator to this state.
- In the standard blocks world and STRIPS robot world, we tried to improve the performance of PRODIGY+STATIC. The base problem solver in this cases uses PRODIGY+STATIC for forward operator generation and selects the most recently visited state for expansion.

Figure 9 shows the cumulative states and CPU time taken in the 8-Puzzle domain before and after learning. In this case we used the same training and test problem sets as in the earlier 8-Puzzle experiment. In the training run, FAILSAFE-2 (using a best-first operator generator) learned 96 censors (the forward-chaining version had learned 109 censors on the same set of training problems). In this experiment the number of visited states after learning was only half of the number before learning (54,244 versus 107,534). However,

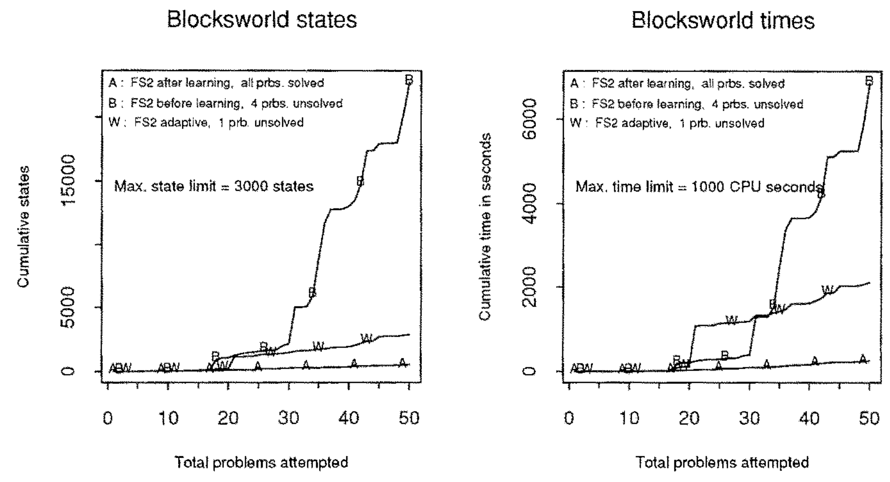


Figure 8. FAILSAFE-2's adaptive behavior in the blocks world domain.

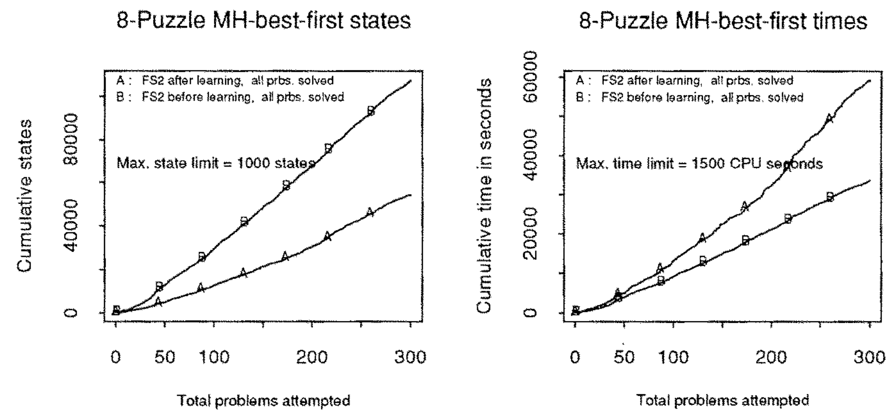


Figure 9. The effect of FAILSAFE-2's learning on the performance of a best-first method in the 8-Puzzle domain.

due to the increased cost of testing the operators against the censors, learning did not improve the problem-solving time of the best-first problem solver. The CPU time after learning was about twice the CPU time before learning.

We confronted some more challenging issues while experimenting with PRODIGY+STATIC. PRODIGY+STATIC achieves a remarkably low forward-branching factor in Minton's three domains. For example, in solving the 50 blocks world test problems of the effectiveness experiment (section 5.1), PRODIGY+STATIC visited a total of 570 states. Out of these states, 466 fell on the solution paths found. Thus the branching factor of PRODIGY+STATIC in blocks world was very close to 1.

In these domains, FAILSAFE-2 failed to improve the performance of PRODIGY+STATIC when it used all of the learned censors. To find a subset of most reliable (probably preservable) and most useful (frequently applicable) censors, we first trained FAILSAFE-2 on 50 training problems and acquired a set of censors. Then we specialized these censors by running FAILSAFE-2 in a *specialization mode* on the same training problems. In this mode, FAILSAFE-2 acquires no new censors but specializes the existing censors when they give incorrect results. The purpose of this additional run was to increase the reliability of the learned censors. Instead of using all censors in the after-learning mode, we used the 25 most frequently applicable censors that had never been specialized in training and specialization modes. That is, the specialization mode was used to identify—and eliminate—censors having exceptions.

Figure 10 shows the results of these experiments in the blocks world. The training problem set was the same as in the effectiveness experiment. The test problem set consisted of 100 problems. Out of these, the first 50 were the 50 odd-numbered problems of Minton's original problem set. To increase the problem set size and the complexity of the problems, we added 50 more problems to this set. These problems were generated using the method outlined in Minton's thesis but had a larger number of blocks (from 12 to 20 blocks). On the 100 test problems, FAILSAFE-2 improved the performance of PRODIGY+STATIC by about 41% in terms of the visited states (1695 versus 2898 states). An improvement of 41% in terms of visited states is statistically significant, and surprisingly large given that the original branching factor of search was almost 1. FAILSAFE-2 achieved this improvement by finding shorter solutions to the problems. The average solution length after learning was only 49% of the average solution length before learning (average solution length reduced from 53 to 26). In this experiment, FAILSAFE-2 showed no significant improvement in terms of the problem-solving time.

We also conducted similar experiments in the STRIPS robot world. In this domain we also generated a relatively more complex set of training and test problems. The problems in this set were generated by adding random rooms, doors, and objects to the original problems.

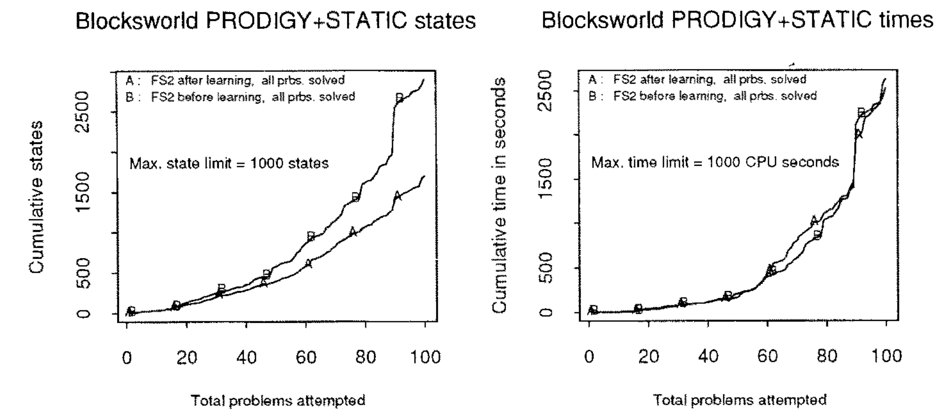


Figure 10. FAILSAFE-2's performance improvement of PRODIGY+STATIC for harder blocks world problems.

Additionally, in all of these problems the number of visited states was at least 1.25 times the length of the solution found in the before-learning case (the branching factor still was almost 1). Out of 194 such problems, we randomly chose 50 for training and specialization, and used the remaining 144 for testing.

Figure 11 shows FAILSAFE-2's performance improvement in this domain. FAILSAFE-2 achieved a moderate 11% savings in the visited states, 16% savings in the solution lengths, and 14% savings in CPU time (confidence level 0.92).

In Minton's scheduling world, PRODIGY+STATIC's forward branching factor was also close to 1. Our efforts to generate more complex problems in this domain resulted in generating problems that had subgoals that cannot be achieved in a linear order (Sussman, 1973) and so were not solvable by PRODIGY.

Thus FAILSAFE-2 was able to make some slight improvements to initially strong problem solvers in domains where they already performed close to optimally. To achieve greater improvements would require either focusing on harder problems (where before-learning performance leaves more room for improvement) or improving the operator generation process itself.

Since we were only interested in satisfying state-space search, in our experiments we did not pay much attention to the lengths of the solutions found by FAILSAFE-2. Is it possible that FAILSAFE-2 reduces problem-solving time and the number of visited states by sacrificing the quality of solutions? To answer this question, we compared the average length of the solutions found by FAILSAFE-2 with the length of the solutions found by the competing methods on identical problems. Such a comparison would be valid only in the cases when both FAILSAFE-2 and the competing method solved most of the problems. The only instance where FAILSAFE-2 found significantly longer solutions than a competing *learning* method occurred in the scheduling world effectiveness experiment. In this domain, the average lengths of the solutions found by FAILSAFE-2 and PRODIGY+STATIC were 3.8 and 2.0, respectively. In contrast, in the blocks world effectiveness experiment

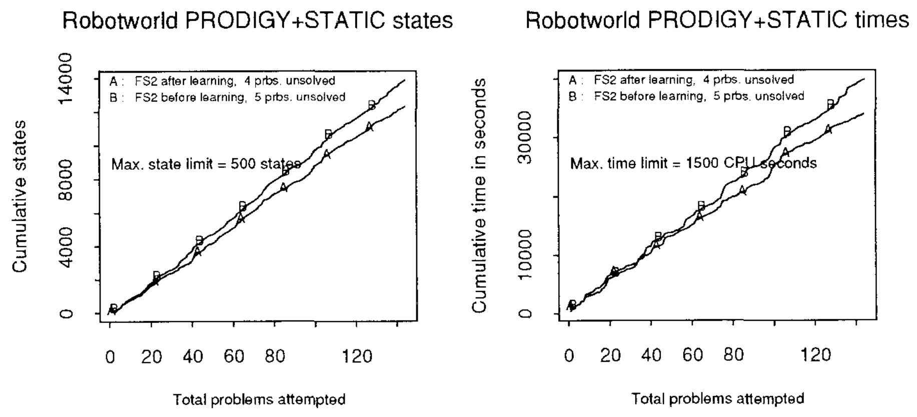


Figure 11. FAILSAFE-2 improves the performance of PRODIGY+STATIC in STRIPS robot world.

FAILSAFE-2 actually found shorter solutions than PRODIGY+STATIC. In this experiment the average length of the solutions found by FAILSAFE-2 and PRODIGY+STATIC were 7.2 and 9.3, respectively. However, in the 8-Puzzle effectiveness experiment FAILSAFE-2 found significantly longer solutions than the competing *heuristic* methods. In this experiment the average lengths of the solution found by heuristic A*, heuristic best-first, and FAILSAFE-2 were 18.9, 46.7, and 60.1, respectively. In summary, FAILSAFE-2's solution quality was lower only in some domains, and hence does not account in general for its improvement over other methods.

5.4. Testing the internal features hypothesis

We also conducted experiments to test the effectivenes of our refinements to FAILSAFE-2's basic learning method: learning macros when exceptions to rules are found, learning irrelevancy censors, enhancing explanations, and forced learning and relaxation. We now report the results of these experiments.

5.4.1. Testing that use of exceptions as macros helps

Exceptions to censors in FAILSAFE-2 are stored as macros (as described in section 4.1). If while testing an operator against a censor an exception holds, FAILSAFE-2 not only applies the operator but "opportunistically" applies the whole macro. Does this opportunistic use of macros give any power to FAILSAFE-2? Does FAILSAFE-2 get all its power from such macros? To answer these questions, we tested FAILSAFE-2 in the blocks world and 8-Puzzle domains. In these experiments FAILSAFE-2 used the control knowledge learned in the previous training runs (reported in section 5.1) but solved the problems with and without using the learned censors and macros. The test problems were the first 65 problems of the test set used in the generality experiments described in section 5.3; the more complex last 15 problems were included to highlight the differences at the end of the curves.

Figure 12 shows the results of these experiments in the blocks world domain. In the "censors and macros" mode, FAILSAFE-2 used all the control knowledge available to it. So it is identical to the after-learning mode described earlier. In the "no censors, macros" mode, FAILSAFE-2 used the macros when they applied but went ahead and applied operators even if they were censored by one of the censors. In the "censors, no macros" mode, FAILSAFE-2 used the censors but did not use the exceptions as macros.[18] In the "no censors, no macros" mode, FAILSAFE-2 did not use censors or macros. So this mode was almost like before-learning mode, except that it let FAILSAFE-2 use the learned goal-ordering rules.

In the blocks world, FAILSAFE-2's performance was best in the "censors and macros" mode. In this mode, its performance was 3.5 and 7 times better (in terms of states and CPU time, respectively) than the "censors, no macros" mode, supporting the hypothesis that using macros as exceptions helps. FAILSAFE-2's performance in "censors and macros" mode was 25 and 16 times better than in "no censors, macros" mode, confirming the hypothesis that even though FAILSAFE-2 is helped by the macros, it does not get all its

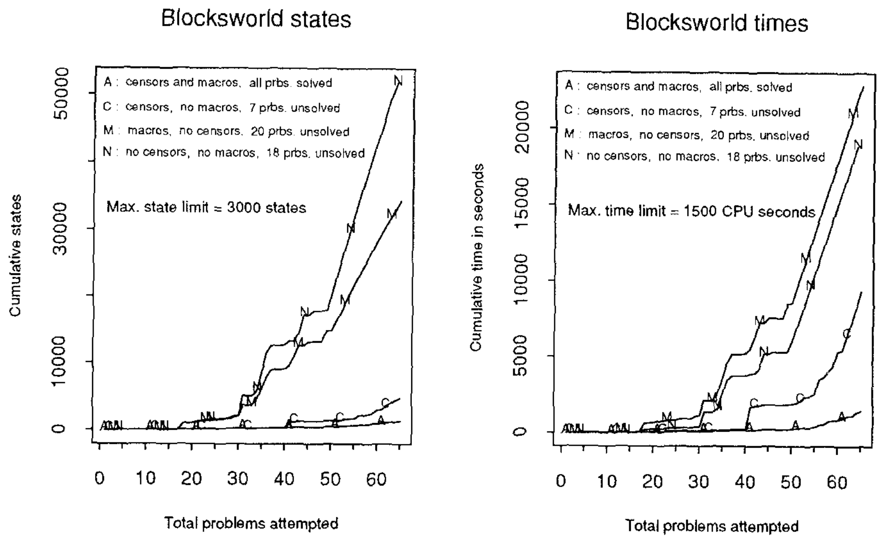


Figure 12. Effectiveness of macros in the blocks world domain.

power from them. There was very little difference between FAILSAFE-2's performance in "no censors, macros" and "no censors, no macros" modes. This fact indicates that macros without censors contribute little, if any, power to FAILSAFE-2. In the 8-Puzzle domain, we got qualitatively similar results. The details of this experiment in the 8-Puzzle domain can be found in the work of Bhatnagar (1992).

5.4.2. Testing that irrelevancy censors and enhancement of failure conditions helps

To evaluate the effectiveness of learning irrelevancy censors and enhancing failure conditions, we trained FAILSAFE-2 on the same 50 blocks world training problems, but in four different training modes—with and without learning irrelevancy censors, and with and without enhancement of failure conditions. As a result of this training, we got four different sets of learned control knowledge, one each for the above four training modes. We solved the same 50 test problems using the above four sets of control knowledge and compared the after-learning performance. Figure 13 shows the cumulative number of states and total time over these problems. Control knowledge learned with both irrelevancy and enhancement was the most effective. The control knowledge learned without irrelevancy and without enhancement was the least effective. The other two cases fell in between. For this set of 50 problems, enhancement of failure conditions alone improved the performance of FAILSAFE-2 more than learning irrelevancy censors alone. This is understandable, because in the blocks world problems, goal-ordering rules play quite a significant role, and enhancement makes learning goal-ordering rules possible.

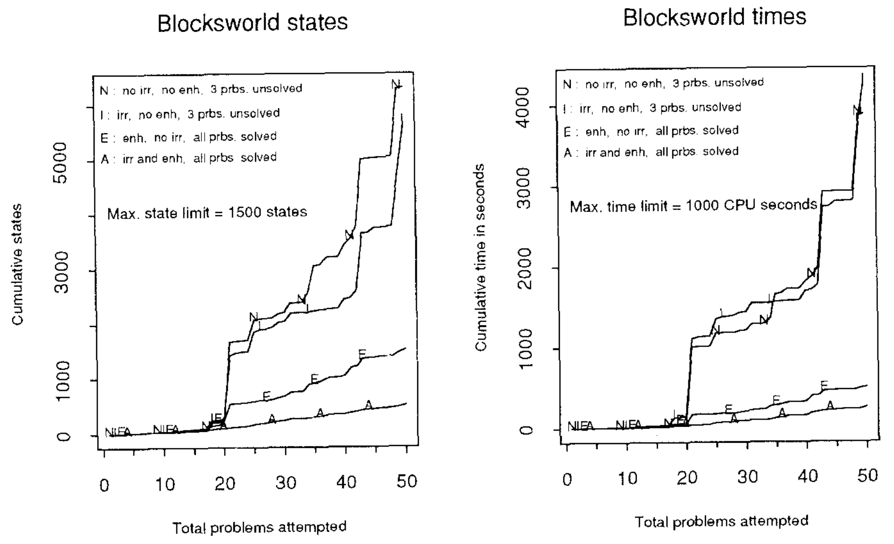


Figure 13. Learning irrelevancy censors and enhancing explanations of failures help in the blocks world.

5.4.3. Testing sensitivity to heuristic parameter values

In a different experiment, we evaluated FAILSAFE-2's sensitivity to the parameters used by the "forced learning" and "forced relaxation" heuristics (section 2.2.1). In this experiment we measured FAILSAFE-2's performance in the blocks world domain for different values of these parameters by independently varying how often forced learning and forced relaxation are performed. The details of this experiment appear in the study by Bhatnagar (1992). It indicated that FAILSAFE-2's performance does indeed benefit from these heuristics, but does not critically depend on the precise values chosen for the parameters.

6. Related work

We now compare FAILSAFE-2 with other *published problem solvers that learn censors by explaining failures they encounter in the process of synthesizing plans.* It is important to point out that there are other settings for failure-driven learning, such as repairing an existing plan (Hammond, 1986; Gupta, 1987), or refining an incomplete or incorrect domain knowledge base. Likewise, there are other forms of search control knowledge worth learning besides censors, other useful sources of information besides failures, and other effective learning methods that do not involve explanation. All this less directly related work lies outside the scope of this article; for pointers to some of it, see Bhatnagar (1992).

FAILSAFE-2 is most closely related to PRODIGY (Minton, 1990) (and its descendants ULS (Chase et al., 1989) and STATIC (Etzioni, 1991)) in its explanation-based learning approach to the acquisition of search control knowledge for a problem solver. We now discuss three key differences from them—the type of knowledge acquired, the technique used to learn it. and the effect of recursion.

Type of knowledge learned: Insofar as FAILSAFE-2 constitutes an attempt to overcome some limitations of PRODIGY and STATIC, it makes sense to compare the censors they learn. In particular, it is natural to ask what kinds of rules FAILSAFE-2 learns that they do not, and vice versa. The kinds of rules each system can learn—and use—depend on the kinds of control choices it can make. Both FAILSAFE-2 and PRODIGY can be viewed in terms of modules that select a subgoal to pursue, a state to expand, and an operator to apply, though not necessarily in that order (FAILSAFE-2 chooses the state before the goal). The key difference has to do with operators. FAILSAFE-2 learns censors that filter the *output* of the given operator generator by pruning (or at least suspending) the *forward* application of operators to the current state if they seem unlikely (or unnecessary) to reach the goal. In contrast, PRODIGY (and STATIC) seek to speed up the built-in means-ends analysis search that generates operators in the first place, so they learn rules that control the *backward* application of operators to the current goal.

In comparing these systems, it would be nice to test how well each one performs when it uses the control rules learned by the other. Unfortunately, such tests are meaningless or impossible. Since the purpose of FAILSAFE-2's censors is to prune operators that might not reach the current goal from the current state, it would be useless to apply them to PRODIGY. Not only do they apply to operators in the wrong direction (forward rather than backward), but they would be superfluous: since PRODIGY back-chains from the current goal, every path generated is guaranteed to reach it—the question instead is whether it can be extended backward to reach the current state. Conversely, since FAILSAFE-2 treats operator generation as a black box, its architecture provides no slot for plugging in PRODIGY's rules directly. However, PRODIGY itself can be plugged in as the operator generator, using *any* set of PRODIGY-type rules—including those produced by STATIC. As section 5 reported, this scheme was used to test FAILSAFE-2's ability to improve further on the control knowledge produced by STATIC.

Learning method: Like PRODIGY, FAILSAFE-2 uses explanation-based learning to generalize failures (and successes) into search control rules. However, FAILSAFE-2 uses a simpler target concept that allows it to overgeneralize by ignoring alternatives to the path that led to failure, as well as the reasons this path cannot be extended to reach the goal. Overgeneralization enables FAILSAFE-2 to learn censors that enforce constraints that are preservable but not necessary. For example, one such censor avoids stacking block ?*X* on block ?*Y* when the goal is to put ?*X* on some other block. The typical effect of this censor is to put block ?*X* on the table instead. While it is always *possible* in the blocks world to put a held block on the table, it is not always *necessary*. Since PRODIGY and STATIC do not overgeneralize, we believe they cannot learn this simple but useful censor (or its backward analogue). Of course, overgeneralization imposes the need for a recovery mechanism, but our empirical results suggest that this cost is outweighed by the benefits of search reduction.

Another apparent benefit of overgeneralization is the simplicity of the learned rules, at least in the domains where we have tested FAILSAFE-2. Each censor has about as many preconditions as the single impossibility rule from which it is derived, plus a few more for each learned exception. In contrast, PRODIGY's rules are often pages long before they subjected to compression analysis to simplify them. Although the two systems have different problem solving architectures and learning methods, we conjecture that the relative simplicity of FAILSAFE-2's censors is due primarily to its simple, overgeneral target concept for failure.

Effect of recursion: Finally, FAILSAFE-2's ability to overgeneralize often enables it to alleviate the complexity caused by recursive problem-solving operators. Figure 14 abstractly depicts how PRODIGY, STATIC, and FAILSAFE-2 explain the failure of some goal *A* that depends on *B* and *C*, where *B* involves recursion and *C* does not. Assume that *A* fails if both nodes *B* and *C* fail. For example, *B* and *C* might correspond to the only two operators for achieving goal *A*; thus *A* cannot be achieved if neither operator applies.

The three systems react differently to *B*. STATIC looks at *B* and *C*, notices that *B* involves recursion, and does not prove or learn anything. PRODIGY builds possibly many

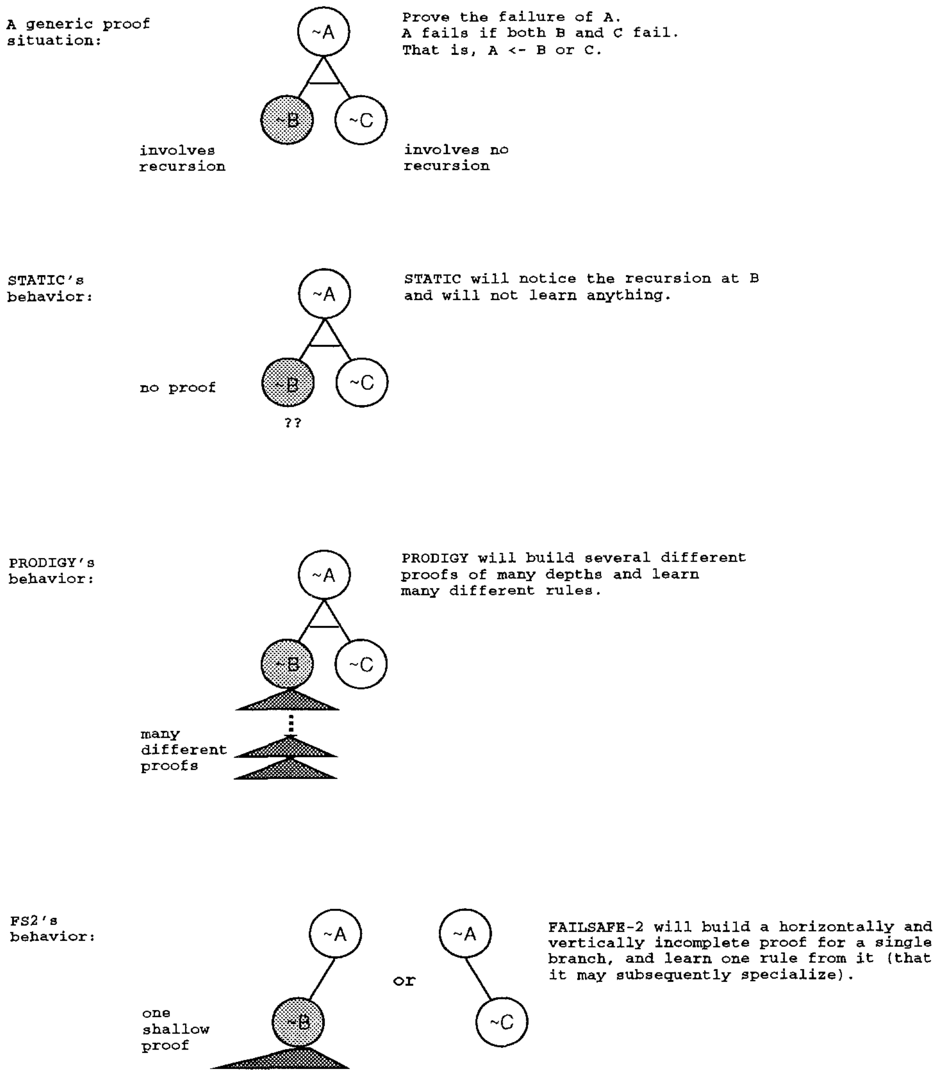

A generic proof situation:
~A
~B
~C
Prove the failure of A.
A fails if both B and C fail.
That is, A <- B or C.
involves recursion
involves no recursion
STATIC's behavior:
~A
~B
~C
no proof
??
STATIC will notice the recursion at B and will not learn anything.
PRODIGY's behavior:
~A
~B
~C
PRODIGY will build several different proofs of many depths and learn many different rules.
many different proofs
FS2's behavior:
~A
~B
or
~A
~C
one shallow proof
FAILSAFE-2 will build a horizontally and vertically incomplete proof for a single branch, and learn one rule from it (that it may subsequently specialize).

Figure 14. Three systems' responses to failures involving recursion.

proofs of *B* for different depths of recursion, combines each of them with the proof of failure of *C*, and learns many rules. Some of these rules may eventually be rejected due to their low utility.

FAILSAFE-2, in contrast, builds a simple explanation that is both horizontally and vertically incomplete, as defined in section 4.3. The horizontal incompleteness stems from considering only the operator that the problem solver happened to try; the learned censor will restrict that operator. The vertical incompleteness stems from ignoring why applying this operator could not be extended to reach the goal by applying subsequent additional operators. FAILSAFE-2 can tolerate an overgeneral censor as long as it is preservable, i.e., the goal can still be achieved by the other operator. If FAILSAFE-2 learns censors on both operators, it may have to relax and specialize one of them. In the worst case, it learns an indefinitely large number of exceptions for the operator that involves recursion.

FAILSAFE-2 can be compared with SOAR (Laird et al., 1986a, 1986b; Rosenbloom & Laird, 1986; Laird et al., 1987; Steier, 1987; Laird, 1988; Unruh & Rosenbloom, 1989), both in terms of their respective problem-solving architectures, and in terms of what knowledge they learn. First, FAILSAFE-2's flexible architecture allows its learning method to be applied to a variety of problem solvers. This aspect of FAILSAFE-2's architecture is closely related to the universal subgoaling idea first employed in SOAR and has been partly motivated by it. Similarly, as we mentioned earlier, various interactions between FAILSAFE-2's learner and problem solver can be compared with impasses in SOAR. Much of FAILSAFE-2's problem-solving behavior could be incorporated in SOAR by giving appropriate pieces of knowledge to SOAR.

Second, many ideas about learning from search failures have been tried in SOAR. However, we are not aware of any published studies of domain-independent methods in SOAR for learning from the sort of "failure by exhaustion" that occurs when no further operators can be applied to a given state. Although in principle SOAR can learn from this type of failure, in practice it is usually kept from doing so, since the resulting rules are either too specific to be useful, or else too general to be valid. FAILSAFE-2 uses various domain-independent heuristics to detect such failures, construct (overgeneral) explanations for them, and recover in those cases where the learned rules are so overgeneral as to be unpreservable. FAILSAFE-2's overgeneralization of failures can be compared with the abstraction techniques used by Unruh and Rosenbloom (1989). While FAILSAFE-2 overgeneralizes failures by including only the failed preconditions of an operator (and ignoring the others) in its explanations, Unruh and Rosenbloom's implementation overgeneralizes success by ignoring the preconditions that fail to hold. FAILSAFE-2's mechanism to recover when censors turn out to be overgeneral can be compared with the process of recovery from incorrect knowledge in SOAR (Laird, 1988). As we mentioned earlier, while FAILSAFE-2 specializes the overgeneral censors, SOAR attempts to learn new chunks that mask the undesirable effects of the incorrect chunks.

7. Conclusions

In this article, we presented FAILSAFE-2, a system that performs adaptive search by learning on-line from its failures. We demonstrated FAILSAFE-2's performance improvement in several experimental domains, supporting our key hypotheses:

- *Effectiveness:* FAILSAFE-2 learns useful rules, even in three recursive domains. In five of the six domains we studied, these rules sped up an initially weak forward-chaining problem solver to a performance level comparable to or better than competing problem solvers guided by strong heuristics (such as Manhattan distance, or the rules that STATIC generates for PRODIGY).
- *Adaptiveness:* FAILSAFE-2 learns fast enough for learning to pay off, even in the course of solving a single problem (at least when the problem solver is an initially weak forward chainer).
- *Generality:* FAILSAFE-2 can improve diverse problem solvers. It was even able to speed up an initially strong problem solver (PRODIGY+STATIC), at least when FAILSAFE-2 was modified to retain only its most reliable learned rules.

We also described a number of refinements to the basic method, and presented empirical evidence of their effectiveness.

The key contribution of FAILSAFE-2 is its use of *preservable* constraints to prune search. The use of these constraints allows FAILSAFE-2 to learn search control rules that cannot be learned by previous systems like PRODIGY and STATIC that refrain from overgeneralization. FAILSAFE-2's ability to leap to overgeneral conclusions (and recover later if necessary) enables it to detect search failures early, generalize from them quickly, and avoid future ones cheaply. This approach should improve on more conservative previous methods in domains where there are strong preservable constraints, especially when simple censors based on such constraints can avoid the recursive explanation problem reported by Etzioni (1990a).

However, the goal of this research is not to replace previous methods, but to improve them. Further research is needed to better understand FAILSAFE-2's advantages and disadvantages relative to the PRODIGY family of problem solvers, in order to identify how best to combine their methods. In particular, what is the analogue of preservable censors for PRODIGY-like problem solvers that apply operators in the backwards direction, and when is it useful to learn them?

An important lesson to draw from FAILSAFE-2 (and many other AI systems) is to identify what types of knowledge are responsible for their effectiveness, or could make them more effective if they were available. FAILSAFE-2's results suggest that an impossibility theory can be a useful form of knowledge for pruning search. Thus one direction for future work is to generate or discover such theories automatically, perhaps by extending the approach described by Siklossy and Dowson (1977) to eliminate the underlying assumptions mentioned in section 3. Conversely, the weakness of FAILSAFE-2's general heuristics for detecting failures suggests the usefulness of strengthening them to exploit available domain knowledge, or of discovering new heuritics for detecting failures as early as possible.

Progress on these and other questions may lead towards adaptive problem solvers that learn sufficiently fast and effectively from their exploration of novel search spaces to converge quickly to solutions even in complex domains.

Appendix I. Impossibility theory given to FAILSAFE-2 for both blocks word domains

The following definitions of failure are given to FAILSAFE-2 for both the standard blocks world and the modified blocks world domains. (For the other domains, please see Bhatnagar (1992).)

Block ?*X* should be on block ?*Y* but is on the table:

current-goal((*on* ?*X* ?*Y*))∧(*on-table* ?*X*)

Block ?*X* is on the wrong block:

current-goal((*on* ?*X* ?*Y*))∧(*on* ?*X* ?*Z*)∧?*Y* ≠?*Z*

Block ?*X* should be on the table but is being held:

current-goal((*on-table* ?*X*))∧(*holding* ?*X*)

Block ?*X* should be on the table but is on another block:

current-goal((*on-table* ?*X*))∧(*on* ?*X* ?*Y*)

The wrong block is on block ?*Y*:

current-goal((*on* ?*X* ?*Y*))∧(*on* ?*Z* ?*Y*)∧?*Z*≠?*X*

Block ?*X* should be on block ?*Y* but is being held:

current-goal((*on* ?*X* ?*Y*))∧(*holding* ?*X*)

The hand should be empty but is holding a block:

current-goal((*hand-empty*))∧(*holding* ?*X*)

The wrong block is being held:

current-goal((*holding* ?*X*))∧(*holding* ?*Y*)∧?*X*≠?*Y*

Block ?*X* should be in the hand but is on the table:

current-goal((*holding* ?*X*))∧(*on-table* ?*X*)

Block ?*X* should be in the hand but is on another block:

current-goal((*holding* ?*X*))∧(*on* ?*X* ?*Y*)

Appendix II. Control rules learned by FAILSAFE-2 in the blocks world

To give a better flavor of the rules that FAILSAFE-2 learns, we shall now paraphrase the rules that FAILSAFE-2 learned in the (standard) blocks world in the effectiveness experiment described in section 5.1. For a listing of the censors learned in all six domains, please see Bhatnagar (1992).

II.1. Censors learned by FAILSAFE-2 in the blocks world

If the goal is to clear a block, and a future goal is to stack a second block on some other block, then do not unstack the second block:

if (*CLEAR* ?*X*) is the goal then censor operator (*UNSTACK* ?*Y* ?*Z*) provided

((*PENDING-GOAL* (*ON* ?*Y* ?*A*)))

(For clarity, we include the precise definition of this censor; for brevity, we omit them below.)

If the goal is to clear a block and you cannot clear it by putting it on the table, then do not pick up other blocks.

If the goal is to clear a block and you cannot clear it by stacking it on a clear block, then do not pick up other blocks.

If the goal is to clear a block, and a future goal is to stack a second block on some other block, then do not put the second block on the table.

If the goal is to clear a block, and a future goal is to stack a second block on some other block, then do not stack the second block on a wrong block.

If the goal is to clear a block but it is under some block, then do not unstack any other block.

If the goal is to clear a block and you are not holding it, then do not apply the stack operator.

If the goal is to clear a block, then do not put other blocks on the table.

If the goal is to hold a block, then do not put other blocks on the table.

If the goal is to hold a block, then do not put other blocks on the table. (FAILSAFE-2 learned two exceptions to this censor.)

If the goal is to hold a block, then reject the stack opeator. (FAILSAFE-2 learned one exception to this censor.)

If the goal is to hold a block and it is on the table, then do not put other blocks on the table.

If the goal is to hold a block, then do not put a block on the table if a future goal is to put the second block on some other block.

If the goal is to hold a block, then do not pick it up if there is a pending goal to pick up some other block. (FAILSAFE-2 learned one exception to this censor.)

If the goal is to hold a block, then do not put a block on the table.

If the goal is to hold a block, then reject the operator stack. (FAILSAFE-2 learned one exception to this censor.)

If the goal is to hold a block, then do not stack A on B if a future goal is to put A on C.

If the goal is to stack a block on another, then reject the operator put-down. (FAILSAFE-2 learned two exceptions to this censor.)

If the goal is to put a block on another that is not clear, then reject the operator pick-up.

If the goal is to stack a block on another, then reject other instances of stack.

If the goal is to stack a block on another, then do not put down the second block.

If the goal is to stack a block on another, then reject other instances of stack.

If the goal is to stack on another, then do not put down the second block if a future goal is to put it on another block. (FAILSAFE-2 learned two exceptions to this censor.)

If the goal is to stack a block on another, then do not stack a wrong block on the second block. (FAILSAFE-2 learned one exception to this censor.)

If the goal is to stack a block on another, then do not pick up a block distinct from the first block.

Do not stack a block one block on another if a different block must go on the second block. (FAILSAFE-2 learned one exception to this censor.)

If the goal is to stack a block on another, then do not stack a wrong block on the second block. (FAILSAFE-2 learned one exception to this censor.)

If the goal is to put A on B, do not pick up A. (FAILSAFE-2 learned one exception to this censor.)

If a future goal is to put a block on the table, then do not put it on some other block. (FAILSAFE-2 learned one exception to this censor.)

If the goal is to put a block on another, then do not put it on a wrong block. (FAILSAFE-2 learned two exceptions to this censor.)

If the goal is to put a block on another, then reject the other instances of the operator stack. (FAILSAFE-2 learned two exceptions to this censor.)

If the goal is to put a block on another, then do not pick up a block that is distinct from the first block.

If the goal is to put a block on another, then do not stack a wrong block on the second block. (FAILSAFE-2 learned one exception to this censor.)

If the goal is to put a block on another, and the first block is on a wrong block, then reject the stack operator for other pairs of blocks. (FAILSAFE-2 learned one exception to this censor.)

If the goal is to put a block on another, and the first block is on the table, then reject the stack operator for other pairs of blocks.

If the goal is to put a block on another, then reject the other instances of the operator stack.

If the goal is to put a block on another, then do not unstack a block from this block. (FAILSAFE-2 learned three exceptions to this censor.)

If the goal is to put a block on another, then do not unstack it from another block. (FAILSAFE-2 learned one exception to this censor.)

If the goal is to put a block on the table, then do not put other blocks on the table. (FAILSAFE-2 learned six exceptions to this censor.)

If the goal is to put a block on the table, then do not put a block on the table if a future goal is to put the second goal on the some block. (FAILSAFE-2 learned one exception to this censor.)

If the goal is to put a block on the table, and there is a pending goal to put another block on the table, do not stack the other block.

If the goal is to put a block on the table, then do not use the stack operator.

If the goal is to put a block on the table, then do not pick up any block.

If (clear X) is a protected goal, then do not put anything on X.

If (clear X) is a protected goal, then do not unstack X from where it is.

If (on X Y) is a protected goal, then do not unstack X from Y.

If (on-table X) is protected, then do not pick up X.

If (clear X) is protected, then do not pick up X.

If (holding X) is protected, then do not put X down.

If *arm-empty* is a protected goal, then reject the operator unstack.

II.2. Goal-ordering rules learned by FAILSAFE-2 in the blocks world

Stack blocks before emptying the arm:

(*prefer-goal* (*on* ?*x* ?*y*) (*arm-empty*))

Stack lower blocks first:

(*prefer-goal* (*on* ?*x* ?*y*) (*on* ?*z* ?*x*))

Stack a block on another before clearing it:

(*prefer-goal* (*on* ?*x* ?*y*) (*clear* ?*x*))

Put a block on the table before stacking something on it:

(*prefer-goal* (*on-table* ?*x*) (*on* ?*y* ?*x*))

Finish all stacking goals before holding something:

(*prefer-goal* (*on* ?*x* ?*y*) (*holding* ?*z*))

Acknowledgments

This work originated at Rutgers University, where it was supported in part by the National Science Foundation (NSF) under Grant Numbers DMC-8610507 and IRI-9017121, in part by the Defense Advanced Research Projects Agency (DARPA) under Contract Number N00014-85-K-0116, and in part by the Center for Computer Aids to Industrial Productivity (CAIP), an Advanced Technology Center of the New Jersey Commission on Science and Technology, at Rutgers University, Piscataway, New Jersey. Subsequent work was performed while the first author was a member of the Learning and Knowledge Acquisition Group at Siemens Corporate Research, Inc., in Princeton, New Jersey. The opinions expressed in this report are those of the authors and do not reflect any policies, either expressed or implied, of any granting agency.

Portions of this article appeared in our AAAI90 paper (Bhatnagar & Mostow, 1990). Additional portions are adapted from the first author's dissertation (Bhatnagar, 1992).

We thank Andrea Danyluk, Steven Minton, Stellan Ohlsson, and Prasad Tadepalli for commenting on drafts of the AAAI90 paper, and Oren Etzioni for additionally commenting on our initial journal submission. Thanks are also due to Alex Borgida, Tom Ellman, John Laird, Joan Lucas, G.D. Mostow, Don Smith, Lou Steinberg, Chris Tong, and many others for their assistance. Special thanks go to dissertation committee members Lou Steinberg, Haym Hirsh, and Steve Minton for their contributions to both research and presentation. Last but by no means least, this article benefited considerably from comments by three anonymous reviewers and the editor, Paul Rosenbloom.

Notes

1. To demonstrate the existence of such sequences, split the n blocks into two piles of $n/2$ blocks. Suppose a move sequence builds each pile into a tower, tears it down, builds it into a new tower with a different ordering of blocks, and so forth. There are $(n/2)!$ possible towers for a pile. By alternating between the two piles, it is possible to build them all without ever repeating the same total configuration of blocks. Any such sequence of moves will be exponentially long in n.
2. Marking a state as suspended does not immediately interrupt its expansion; it just records the fact that at least one move from the state was censored and so may be reconsidered later.
3. Actually, depth-limited search is accomplished by a somewhat different mechanism, to be described shortly.
4. In the experiments reported here, this limit was set to 10. Section 5.4.3 discusses FAILSAFE-2's sensitivity to this parameter.
5. In the experiments reported here, this limit was set to 15. Section 5.4.3 discusses FAILSAFE-2's sensitivity to this parameter.
6. However, if the operator generator is sensitive to the current or pending goals, goal-ordering rules may alter the generated space—and FAILSAFE-2 cannot modify bad goal-ordering rules.
7. FAILSAFE-2's impossibility theory rules roughly resemble the goal-negation axioms and the constraints on legal states used by STATIC (Etzioni, 1990b).
8. As a reviewer pointed out, this technique relies both on the closed-world assumption (no additional operators) and on the STRIPS assumption (all effects noted in an operator).
9. The term *attempt* is a bit misleading here, since the operator generator for blind depth-first search pays no attention to the current goal.
10. This policy neutralizes any bias in the ordering of the rules, and also allows FAILSAFE-2 to learn diverse rules from similar failures.
11. Actually, the learner constructs a slightly more complex explanation of failure, but for clarity of exposition, we defer this complication until section 4.3.
12. Limitations of this matching process occasionally produce "duplicate" (logically equivalent) censors. Eliminating this duplication would make FAILSAFE-2 slightly more efficient.
13. Although the simple operator generator in this example is not directly sensitive to the current and pending goals, the learned censors are.
14. The augmented blocks world is the same as the standard blocks world, but has an additional problem-solving operator that allows the problem solver to grasp a block that is second from the top of a tower.
15. The Think-A-Dot puzzle consists of a two-dimensional grid of interconnected chutes with gates. The gates in the chutes can point either to the left or to the right. A problem-solving operator in this puzzle consists of dropping a marble down a chute. Every time a marble passes through a gate, the direction of the gate is inverted. A typical problem in this domain consists of achieving a given goal configuration of the gates starting from a given initial configuration.
16. These problems were derived from the original 100 scheduling world problems used by PRODIGY. In the original problem set, 39 of 100 problems could not be solved by PRODIGY due to some missing information. To increase the number of solvable problems, we added some of the missing information to them. The details of these additions are given by Bhatnagar (1992).

17. Confidence levels close to 1 indicate that the hypothesis is well supported by the data. Low confidence levels contradict the hypothesis. Throughout our experiments, whenever we compared two sets of data, we calculated the corresponding confidence levels. In all our claims about performance improvement, the confidence levels were high (0.95 or better). In this article we shall mention only those confidence levels that were less than 0.95.
18. Exceptions to overgeneral censors were still learned, but were not used as macros.

References

Bhatnagar, N. (1992). *On-line learning from search failures.* Doctoral dissertation, Rutgers University Computer Science Department, New Brunswick, NJ.

Bhatnagar, N. & Mostow, J. (1990). Adaptive search by explanation-based learning of heuristic censors. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI90).* Boston, MA: AAAI. (Available as Rutgers AI/Design Project Working Paper Number 169.)

Chase, M., Zweben, M., Piazza, R., Burger, J., Maglio, P., & Hirsh, H. (1989). Approximating learned search control knowledge. In *Proceedings of the Sixth International Workshop on Machine Learning.* Ithaca, NY: Morgan Kaufmann.

Dejong, G., & Mooney, R. (1986). Explanation-based generalizaton: an alternative view. *Machine Learning, 1(2),* 145-176.

Etzioni, O. (1990). Why Prodigy/EBL works. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI90).* Boston, MA: AAAI.

Etzioni, O. (1990). *A structural theory of explanation-based learning.* Doctoral dissertation, Carnegie-Mellon University, Pittsburgh, PA.

Etzioni, O. (1991). STATIC: A problem-space compiler for PRODIGY. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI90).* Boston, MA: AAAI.

Gupta, A. (1987). Explanation-based failure recovery. *AAAI87.* Seattle, WA: AAAI.

Hammond, K.J. (1986). Learning to anticipate and avoid planning problems through the explanation of failures. *AAAI86.* Philadelphia, PA: American Association for Artificial Intelligence.

Iba, G.A. (1989). A heuristic approach to the discovery of macro-operators. *Machine Learning, 3(4),* 285-317.

Kibler, D. & Morris, P. (1983). Don't be stupid. *IJCAI83.* Karlsruhe, Germany.

Knoblock, C.A. (1990). Learning abstraction hierarchies for problem solving. In *Proceedings of the Eight National Conference on Artificial Intelligence (AAAI90).* Boston, MA: AAAI.

Laird, J.E., (August 1988). Recovery from incorrect knowledge in SOAR. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI88).* St. Paul, MN: AAAI.

Laird, J.E. Rosenbloom, P.S., & Newell, A. (1986). Chunking in Soar: the anatomy of a general learning mechanism. *Machine Learning, 1(1),* 11-46.

Laird, J.E., Rosenbloom, P.S., & Newell, A. (1986). Overgeneralization during knowledge compilation in Soar. *Workshop on Knowledge Compilation.* Oregon State University.

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

Mahadevan, S. (1989). *An apprentice-based approach to learning problem-solving knowledge.* Doctoral dissertation, Rutgers University Computer Science Department, New Brunswick, NJ. Rutgers Computer Science Technical Report Number ML-TR-30.

Minton, S. (1985). Selectively generalizing plans for problem-solving. *Proceedings IJCAI85.* Los Angeles, CA.

Minton, S. (1988). *Learning effective search control knowledge: An explanation-based approach.* Doctoral dissertation, Carnegie-Mellon University Computer Science Department, Pittsburgh, PA.

Minton, S. (1988). Quantitative results concerning the utility of explanation-based learning. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI88).* St. Paul, MN: AAAI.

Minton, S. (March 1990). Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence, 42(2-3),* 363-391. (Revised version of AAAI88 paper.)

Mitchell, T.M., Keller, R.M., & Kedar-Cabelli, S.T. (1986). Explanation-based generalization: a unifying view. *Machine Learning, 1(1),* 47-80.

Mostow, D.J. (1983). Machine transformation of advice into a heuristic search procedure. In J.G. Carbonell, R.S. Michalski, & T.M. Mitchell (Eds.), *Machine learning.* Palo Alto, CA: Tioga.

Mostow, J. (1985). Toward better models of the design process. *AI Magazine, 6(1)*, 44–57.

Mostow, J. (1989). Design by derivational analogy: issues in the automated replay of design plans. *Artificial Intelligence, 40(1–3)*, 119–184. In J. Carbonell (Ed.), Special Volume on Machine Learning. Reprinted as *Machine learning: Paradigms and methods*, Cambridge, MA: MIT Press, 1990.

Mostow, J. & Bhatnagar, (1987). Failsafe—a floor planner that uses EBG to learn from its failures. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI87).* Milan, Italy: Morgan Kaufmann.

Mostow, J., & Prieditis, A.E. (1989). Discovering admissible heuristics by abstracting and optimizing: a transformational approach. In *Proceedings of the Eleventh Joint International Conference on Artificial Intelligence.* Detroit, MI.

Mostow, J., Barley, M., & Weinrich, T. (1989). Automated reuse of design plans, *International Journal for Artificial Intelligence in Engineering, 4(4)*, 181–196.

Nilsson, N. (1980). *Principles of artificial intelligence.* Los Altos, CA: Morgan Kaufmann.

Norton, S., & Kelly, K. (1988). Learning preference rules for a VLSI design problem-solver. *The 4th IEEE Conference on AI Applications.* San Diego, CA: IEEE (Available as Rutgers AI/VLSI Project Working Paper No. 66.)

Prieditis, A. (1990). *Discovering admissible heuristics by abstraction and speedup: A transformational approach.* Doctoral dissertation, Rutgers University Computer Science Department, New Brunswick, NJ.

Rosenbloom, P.S., & Laird, J.E. (1986). Mapping explanation-base generalization onto Soar. *Proceedings AAAI86.* Philadelphia, PA: AAAI.

Segre, A., Elkan, C., & Russell, A. (1991). A critical look at experimental evaluations of EBL. *Machine Learning, 6(4).*

Siklossy, L., & Dowson, C. (1977). The role of preprocessing in problem solving systems. *IJCAI-5.* Cambridge, MA.

Smith, Gary. (1992). *Statistical reasoning.* Boston, MA: Allyn and Bacon, Inc.

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

Steier, D.M. (1987). Cypress-Soar: A case study in search and learning in algorithm design. In *Proceedings IJCAI-87.* Milan, Italy.

Sussman, G.J. (1973). *A computational model of skill acquisition.* Doctoral dissertation, Massachusetts Institute of Technology, Cambridge, MA.

Tong, C. (July 1987). Toward an engineering science of knowledge-based design. *International Journal of Artificial Intelligence in Engineering, 2(3).* In special issue on AI in Engineering Design. (Also available as Rutgers AI/VLSI Project Working Paper Number 49.)

Tong, C. (1987). Learning justifiable design evaluation functions: A decomposable, multi-agent learning problem (Rutgers AI/VLSI Project Working Paper No. 48). Rutgers University, New Brunswick, NJ.

Tong, C. (1988). *Knowledge-based circuit design.* Doctoral dissertation, Stanford University Computer Science Department, Stanford, CA.

Tong, C. (1990). Knowledge-based design as an engineering science: The Rutgers AI/Design Project. In *Proceedings of the AI in Engineering Conference.* Boston, MA. Invited paper. (Also appears as Rutgers AI/Design Project Working Paper Number 167).

Tong, C. & Franklin, P. (1989). Tuning a knowledge base of refinement rules to create good circuit designs. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence.* Detroit, MI.

Unruh, A., & Rosenbloom, P.S. (1989). Abstraction in problem solving and learning. In *Proceedings of the Eleventh Joint International Conference on Artificial Intelligence*, Detroit, MI.

Voigt, K., & Tong, C. (1989). Automating the construction of patchers that satisfy global constraints. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence.* Detroit, MI.

Received August 1990
Accepted May 1993
Final Manuscript May 28, 1993