# EASe: Integrating Search with Learned Episodes

**David Ruby**
druby@ics.uci.edu
**Dennis Kibler**
kibler@ics.uci.edu

Technical Report 92–30

May 15, 1992

This research was supported by National Science Foundation grants number IRI-9001756 and IRI-9143332.

# EASe: Integrating Search with Learned Episodes

David Ruby
druby@ics.uci.edu

Dennis Kibler
kibler@ics.uci.edu

*Department of Information & Computer Science*
*University of California, Irvine*
*Irvine, CA 92717 U.S.A.*
*(714) 856-7557*

Keywords: Learning, problem solving, weak methods, episodes, macro-operators, cases.

## Abstract

Weak methods are insufficient to solve complex problems. Constrained weak methods, like hill-climbing, search too little of the problem space. Unconstrained weak methods, like breadth-first search, are intractable. Fortunately, through the integration of multiple weak methods more powerful problem solvers can be created. We demonstrate that augmenting a weak constrained search method with episodes provides a tractable method for solving a large class of problems. We demonstrate that these episodes can be generated using an unconstrained weak method while solving simple problems from a domain. We provide an analytical model of our approach and empirical results from the logic synthesis domain of VLSI design as well as the classic tile-sliding domain.

# 1 INTRODUCTION

Search has proven to be an excellent metaphor for understanding and generating intelligent behavior. Researchers have discovered several different types of fundamental search methods. These weak methods form a core understanding of how to search within a problem space. Unfortunately, these weak methods alone cannot solve complex problems.

Weak methods fall into two general classes, constrained and unconstrained. Constrained weak methods, like hill-climbing or beam-search, remain largely tractable while searching within a problem space. But, they search so little of the problem space that they seldom solve difficult problems. The search conducted by an unconstrained weak method like breadth-first search or iterative deepening (Korf, 1985a) is assured to include a solution, but is often intractable.

Case-based search (Lehnert & Bradtke, 1988; Ruby & Kibler, 1988) demonstrates how cases can constrain search within a complex problem space, keeping it tractable. Solving problems by reusing previous cases is an approach that is unaffected by the branching factor of the problem or the depth of a solution. Its major limitation is the need for enough cases to cover the problem space.

Each weak method can only solve a small class of problems. What is needed is a way to enlarge this class by integrating these existing methods.

## 1.1 Importance of Integrating Methods

The potential power from integrating the right approaches is understood within the problem-solving community. SOAR (Laird, Newell, & Rosenbloom, 1987), for example, provides a general problem-solving framework that includes a mechanism for integrating multiple methods. Multiple problem spaces in SOAR allows it to use several different weak methods while solving a problem. What is not understood about problem solving is which methods should be integrated and how they should be organized.

What we want is an integrated problem solver that tractably finds solutions to problems in complex domains. One way to accomplish this is to begin with an intractable unconstrained method and learn to improve it, speedup learning. An alternative is to begin with a tractable constrained method and to increase its capabilities. This is the approach we take. Thus, we must integrate methods that are tractable. In addition, the methods must complement one another so that their integration provides a more powerful approach than either of the them alone. The question remains, though, which particular tractable methods do we integrate.

Problem decomposition is a classic approach to problem solving. Breaking a large problem into simpler subproblems is almost always useful. This approach is tractable and makes a good choice for integrating with other search methods. Hill-climbing is tractable and makes another good choice for integration. It uses a global evaluation function and is thus complemented well by a technique that uses local knowledge. Cases provide a representation for local knowledge. Case-based problem solving is tractable as long as the number of

cases needed remains low and few modifications are required to reuse them. It too makes a good choice for a tractable method. The complementary nature of these methods make them excellent choices for integration.

The cases needed for case-based problem solving must somehow be learned. An unconstrained weak method must be used to generate these cases to ensure that any one needed is learnable. Since these methods are generally intractable, an additional technique is needed to make learning the required cases tractable. When people learn to solve problems in a new domain they usually begin by solving *simple* ones. After learning to solve these simple problems they transfer their knowledge to larger and more complex ones. These simple problems provide them with the opportunity to acquire the knowledge needed to solve the various other problems in the domain. We use an approach similar to this to make learning tractable for our problem solver.

Simple problems are characterized by a smaller branching factor and depth of solution than the more complex ones from the same domain, thus they have a smaller problem space. In these smaller problem spaces even an intractable method like iterative deepening can find a solution. So simple problems can essentially provide a means for making an intractable search method tractable. The solutions found can then be used by a case-based method while solving more complex problems where the problem space is too large for unconstrained search.

These then are the basic tools that we bring together to form our problem solver: (1) goal decomposition to decompose a problem into simpler subproblems, (2) hill-climbing for tractably searching through much of the problem space, (3) case-based problem solving for handling the exceptions to hill-climbing, and (4) iterative deepening for solving the simple problems needed to learn the required cases.

### 1.2 Importance of Evaluation

Before examining our approach in more detail there is one important question to answer. How are we to evaluate our problem solver? Using a critical theory of problem solving we could evaluate it, but no agreed upon theory exists. Given this inability to properly evaluate it theoretically, an alternative is an empirical evaluation. Probably the best way to empirically evaluate a problem solver is against problems of known difficulty. A set of scaled benchmarks can measure the power of a problem solver. Unfortunately, there is no agreed upon measure of problem difficulty and no benchmark set of scaled problems.

Minton (1988) with his work on Prodigy demonstrated that the difficulty of problems within a domain can be roughly determined. For example, in the blocks world the number of blocks in a problem is a rough measure of their difficulty. These rough measures when combined with large sets of random problems provide a fairly accurate measure of the average difficulty of a set of problems. Using these rough measures it is possible to generate a set of scaled benchmark problems within a domain. These scaled problems then provide

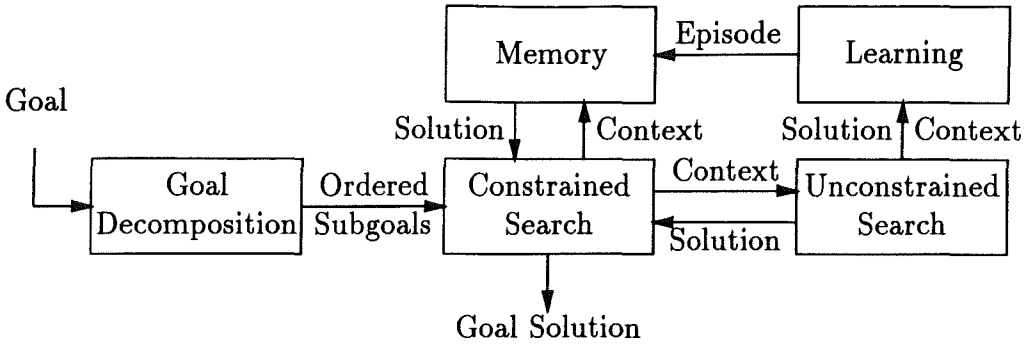


Figure 1: Overview of EASe

a means for evaluating a problem solver. The use of multiple domains for evaluation adds further validity to the results.

The evaluation required of the solutions produced by a problem solver can vary across domains. In some cases, any solution that meets the goal is equally acceptable. In other cases, the quality of the final solution is important. In domains where the quality is important, comparisons with the best alternative for solving the same type of problem is also needed. Not only must learning be compared against not learning within the same approach, but also against an independent quality metric. In addition, since our goal is to discover approaches for solving problems that currently require human abilities, the domains on which we choose to evaluate our approach are those that are also difficult for people. This adds a further test of the power of our approach.

Finally, the best form of evaluation is an analytical one. An analysis of the power of an approach that can tell us when it will and will not work is by far the best means for evaluating it. Unfortunately, such strong results are seldom possible. We will provide analytical results for a simplified model of our approach. We use these analytical results to qualitatively explain our empirical results. This helps in both understanding and evaluating our approach.

In the following sections we will describe our approach, which we call *Episode Augmented Search* or EASE. We provide empirical evidence from the classic tile-sliding domain, and the logic synthesis task of VLSI design. We provide an analytical model for explaining the behavior of our approach, and provide a description of the mechanisms that come together to provide the power for EASe.

## 2 AUGMENTING SEARCH WITH EPISODES

EASe consists of five general components: (1) goal decomposition, (2) constrained search, (3) memory, (4) unconstrained search, and (5) learning. Each of these components operates largely independently from the others. The architecture of EASe is outlined in Figure 1.

EASe uses a generalization of the problem-solving architecture developed for SteppingStone (Ruby & Kibler, 1989; Ruby & Kibler, 1991). The previous work with this architecture was dependent upon the use of steppingstones

as the representation for the learned knowledge. EASe removes this dependency and demonstrates that the representation of the learned knowledge is not as important as the general problem-solving architecture. In the following subsections we will describe the functionality of each component of EASe.

### 2.1 Goal Decomposition

EASe assumes a state-space representation for problems that consist of a goal, a set of operators, and an initial state. The goal consists of the conjunction of a set of subgoals. EASe decomposes the problem of solving the goal to that of solving each of the individual subgoals and protecting them once solved. The subgoals are ordered so that when solved successively, the likelihood a solved subgoal will need undoing is reduced. Note in Figure 1 how problem solving begins with the goal decomposition component decomposing a goal into an ordered sequence of subgoals.

As problem solving proceeds, the protection of solved subgoals increasingly constrains the remaining search. The protection of some subgoals constrain search more than others. For example, solving the center tile of an 8-puzzle problem constrains the moves of a state much more than solving a corner tile. Previous work on ordering subgoals (Cheng & Irani, 1989) have focussed on determining required orderings of subgoals. In many domains there are no required subgoal orderings, but some orderings are still preferable to others. For example, generally the more constrained search becomes, the less likely it will contain a solution. A good ordering of the subgoals is one that keeps the state as unconstrained as possible as each subgoal is solved. Finding such an ordering requires some measure of the degree of constraint in a state.

We developed the domain independent heuristic *openness* (Ruby & Kibler, 1989) to measure the degree of constraint within a state or partial state. It operates similarly to an idea loosely outlined by Korf (1985b) for ordering subgoals. Openness estimates the likelihood of solving a subgoal given a set of solved subgoals are protected.

We begin by defining openness over a single unsolved goal and a set of solved goals. Let $G_S$ denote a set of solved goals, $G_U$ a set of unsolved goals, and $g$ a single goal in $G_U$. The openness of $g$ with respect to $G_S$ is defined as:

$$\text{Openness}(G_S, g) = \text{Number of operators for moving from a state where goals } G_S \text{ are solved and } g \text{ is not, to a state where } G_S \text{ and } g \text{ are both solved.}$$

Openness can be computed in several different ways. For STRIPS style operators, the Openness$(G_S, g)$ is computed by counting the number of operators for solving $g$ whose preconditions are not contradicted by the assertion of goals $G_S$. For domains with invertible operators, we count the number of operators for undoing $g$ that also maintain $G_S$ given a state where $g$ and goals $G_S$ are solved. It any case, openness estimates the likelihood of solving goal $g$ given goals $G_S$ are solved and protected. It is only an estimate since it ignores interactions arising when solving preconditions of the operators. The openness

of the set $G_U$ of unsolved goals with respect to the set $G_S$ of solved goals is now defined as:

$$\text{Openness}(G_S, G_U) = \sum_{g \in G_U} \text{Openness}(G_S, g).$$

Given a goal set $G$ consisting of $g_1, g_2, \ldots, g_n$, the desired ordering maximizes the sum:

$$\sum_{i=1}^{n-1} \text{Openness}(g_1 \ldots g_i, g_{i+1} \ldots g_n).$$

Rather than finding a subgoal sequence that maximizes this sum, we instead find one with a simple hill-climbing search. We begin by assuming all of the subgoals are solved and test which subgoal if unsolved would most *open* the state. This is selected as the final subgoal in the sequence. The next subgoal is chosen in the same way until all of the subgoals are included in the sequence. Currently subgoal orderings are not learned. The subgoal ordering is found for each problem with this hill-climbing search. The cost of this search is then included in the cost of solving the problem. These orderings could be learned to further improve problem solving performance.

### 2.2 Constrained Search

The constrained search component takes as input an ordered set of subgoals. It then attempts to solve each subgoal successively. Once solved, a subgoal is protected, so that constrained search cannot undo it. Constrained search attempts to solve each of these subgoals with hill-climbing.

Each subgoal is assumed to have a measure for evaluating its closeness to being solved. For example, in the 8-puzzle domain a subgoal consists of getting a particular tile into its goal position. In this domain, the Manhattan distance a tile is from its goal position serves as its measure of completion. For those subgoals that are either true or false, improvement is possible only by solving them.

When hill-climbing for a particular subgoal, only its distance from completion is used. This differs from standard hill-climbing where the metric used is the sum of the distances of each of the subgoals of the problem. When hill-climbing cannot solve a subgoal without undoing a previously solved one, an impasse occurs. When and only when an impasse occurs is the memory component called.

Constrained search uses operator subgoaling. Operators may or may not have preconditions. If an operator for improving upon the current subgoal has preconditions, EASe is recursively called on them. The new context will include all of the protected subgoals of the previous levels. Memory and learning may be called at any level of this recursion.

Allowing operator subgoaling is dangerous since it can make constrained search intractable. We choose to allow it because it provides a means for defining operators that are much more natural to the problem. For example, in the tile-sliding domain it allows defining operators for moving any tile, instead of just the blank. To ensure that it remains tractable we limit the

CONTEXT

| CURRENT | PROTECTED |
|---|---|
| position-1-3(blank) | position-1-1(1) |
| | position-1-2(2) |
| | position-2-3(3) |

STATE

| | | |
|---|---|---|
| 1 | 2 | 8 |
| 4 | | 3 |
| 5 | 7 | 6 |

Figure 2: Example Context and State from 8-puzzle

depth of the recursion to be linear in the size of the problem. This ensures that subgoaling will not continue indefinitely, and ensures the tractability of constrained search.

### 2.3 Memory of Problem Solving Episodes

The memory component of EASe takes as input a context and produces as output a solution to the impasse. A context consists of the subgoal currently being solved and the currently protected subgoals. Figure 2 gives an example context from the 8-puzzle along with the current state. In this example, the final goal is to have the tiles in the standard numeric order. Currently, tile 1 and tile 2 have been placed into their final position. The final position for tile 3 is the upper right corner, `position-1-3(3)`[1]. Tile 3 is protected in its current location, `position-2-3(3)`, since it is a precondition for moving the tile up from there to its goal position. The other precondition for the move up is that the blank be in the destination position, `position-1-3(blank)`. Once the blank is correctly placed all of the preconditions for the move up will be satisfied. Their protection will be removed, and the 3 tile will be moved up into its final position and protected. Unfortunately, the subgoal `position-1-3(blank)` cannot be solved while maintaining the protected subgoals. To resolve this impasse EASe looks to its memory of problem solving knowledge.

Memory consists of a set of problem solving episodes. An episode is a piece of a problem solving case where reducing the current subgoal required undoing protected subgoals. It encodes an exception to the problem solving conducted by the constrained problem solver. Episodes act as exceptions to the constrained problem solver much as Golding & Rosenbloom (1991) use cases to encode exceptions to their rule-based system.

PET also learned episodes for improving upon a subgoal (Kibler & Porter, 1983). Like EASe, PET only learned episodes when problem solving required going against the standard measure of improvement. The episodes of PET, though, consisted of a group of rules. These rules operated as a loosely packed

[1]Note that `position-X-Y(Z)` refers to tile Z in row X and column Y

CONTEXT

| CURRENT | PROTECTED |
|---|---|
| position-1-3(W) | position-1-1(X) |
| | position-1-2(Y) |
| | position-2-3(Z) |

IMPASSE SOLUTION

| | | |
|---|---|---|
| (1) left | (2) up | (3) right |
| (4) right | (5) down | (6) left |
| (7) up | (8) left | (9) down |
| (10) right | (11) right | (12) up |

Figure 3: Example Episode from Memory

macro-operator for improving upon a subgoal. The episodes of EASe are represented and used in a much different way.

Each episode consists of a context and the impasse solution. The impasse solution of an episode generated improvement on the current subgoal when it was learned. The protected subgoals of an episode's context are those that were protected when it was learned that were undone and resolved by the impasse solution. The episode is a piece of the entire previous problem solving case. Only those pieces of a case where an impasse was resolved are learned. The rest of the case is discarded. The learned episodes are then indexed by their context. Unlike the rules of PET that are tried on every problem solving cycle, an episode is only indexed when an impasse occurs. Figure 3 gives an example episode from the 8-puzzle domain. The moves listed in the impasse solution for this episode refer to the direction the blank is moved.

If the current context matches the context of an episode from memory, its impasse solution is returned to the constrained search method. For the current context to match the context of an episode, the current subgoal of the episode must bind to the current subgoal of the inputted context. The protected subgoals of the episode must also bind to some subset of the protected subgoals of the inputted context.

When the memory component returns an impasse solution to constrained search it is applied again. If its application produces an improvement upon the current subgoal as well as maintaining all of the protected subgoals, then problem solving continues. If the new state generated does not meet all of those constraints, then the changes are undone and other episodes from memory matching the current context are tried. During testing, when memory no longer has any additional episodes that are relevant to the current impasse, problem solving ends. This ensures that problem solving will remain tractable. When learning is enabled, like during training, unconstrained search is called after the memory component fails.

### 2.4 Learning New Episodes with Unconstrained Search

Unconstrained search is only called during training, since in the general case it is intractable. It stays tractable during training only because simple problems are used. These simple problems provide an opportunity for EASe to learn its needed episodes.

Unconstrained search takes as input a context, like the memory component. It then searches the problem space for a sequence of moves that generate improvement upon the current subgoal while ensuring that all protected subgoals that are undone are resolved. For this we use iterative deepening depth-first search (Korf, 1985a).

If an impasse solution is found, unconstrained search passes it and the context to the learning component. It also passes the impasse solution back to constrained search so problem solving can continue. If unconstrained search cannot solve an impasse, it passes a null solution to constrained search, which then has no recourse but to end problem solving.

The learning component takes as input a context and an impasse solution. The learning component uses the solution and context to determine what part of the current context is relevant to the episode. Figure 4 illustrates an example context and solution from the 8-puzzle.

The learning component determines the relevant portions of the context by replaying the impasse solution. After replaying the sequence of moves, the learner determines which of the protected subgoals were undone during the solution. Only these subgoals are included in the context for the episode in memory. To make the indexing of the episodes explicit, constants are then variablized. This can lead to overgeneralization, but can be tolerated since it is only used to index the episode. Figure 3 illustrates the episode that is generated from the context and solution of Figure 4.

The assumption that the episodes learned on one problem will transfer to other ones is integral to EASe. Langley (Langley, 1985) outlined three forms of transference possible with learned problem-solving knowledge. The simplest transference is *scaling up*. This allows applying knowledge learned on simple problems to more complex ones of the same type. It is this type of transference that we are using to make learning tractable by enabling us to learn only on simple problems. The second form of transference outlined by Langley allows knowledge learned on one type of problem to apply to other types. We make use of this transference as well by learning episodes for subproblems. As long as the learned subproblems recur in problems, regardless of their type, the knowledge will transfer. Langley actually outlines a third type of transferance as well, *learning by analogy*. Although clearly important, we do not currently make use of it.

## 3 PROBLEM SOLVING WITH EASE

To demonstrate our approach scales to problems with large search spaces and many subgoal interactions we chose to test it on a classic problem-solving domain. The classic 8-puzzle and 15-puzzle are well known in the problem-

CONTEXT

| CURRENT | PROTECTED |
|---|---|
| position-1-3(blank) | position-1-1(1) |
| | position-1-2(2) |
| | position-2-3(3) |

IMPASSE SOLUTION AND STATES GENERATED

(1) left

| | | |
|---|---|---|
| 1 | 2 | 8 |
| | 4 | 3 |
| 5 | 7 | 6 |

(2) up

| | | |
|---|---|---|
| | 2 | 8 |
| 1 | 4 | 3 |
| 5 | 7 | 6 |

(3) right

| | | |
|---|---|---|
| 2 | | 8 |
| 1 | 4 | 3 |
| 5 | 7 | 6 |

(4) right

| | | |
|---|---|---|
| 2 | 8 | |
| 1 | 4 | 3 |
| 5 | 7 | 6 |

(5) down

| | | |
|---|---|---|
| 2 | 8 | 3 |
| 1 | 4 | |
| 5 | 7 | 6 |

(6) left

| | | |
|---|---|---|
| 2 | 8 | 3 |
| 1 | | 4 |
| 5 | 7 | 6 |

(7) up

| | | |
|---|---|---|
| 2 | | 3 |
| 1 | 8 | 4 |
| 5 | 7 | 6 |

(8) left

| | | |
|---|---|---|
| | 2 | 3 |
| 1 | 8 | 4 |
| 5 | 7 | 6 |

(9) down

| | | |
|---|---|---|
| 1 | 2 | 3 |
| | 8 | 4 |
| 5 | 7 | 6 |

(10) right

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 8 | | 4 |
| 5 | 7 | 6 |

(11) right

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 8 | 4 | |
| 5 | 7 | 6 |

(12) up

| | | |
|---|---|---|
| 1 | 2 | |
| 8 | 4 | 3 |
| 5 | 7 | 6 |

Figure 4: Example Context and Solution from 8-puzzle

solving literature. We chose to use a generalization of these domains to test our approach.

We define the general tile-sliding problem to consist of moving from a random initial tile-sliding state to some random achievable final state. A state consists of an NxN matrix of tiles. Operators are defined as moving a tile or a blank either left, right, up or down. The precondition for an operator moving a tile is that the new location for the tile be occupied by a blank. Operators for moving the blank have no precondition.

The goal for an NxN tile-sliding problem consists of $N^2-1$ subgoals defining the final position for each of the tiles. The total number of states are $N^2!$. From each of these, there are $N^2!/2$ reachable states. Therefore, there are $N^2! * (N^2!/2)$ different solvable problems in a NxN tile-sliding domain. The large problem space and strongly interacting subgoals of this domain makes it difficult.

Earlier learning research with this domain (Ruby & Kibler, 1989; Korf, 1985b; Iba, 1989) operated only on tile-sliding problems with a fixed goal. Korf (1985b) describes how any approach for solving the fixed goal problem can also solve a subset of the random goal problems. If a path from the random start state and final state can be found to the fixed goal, these solutions can be combined to form one for the entire problem. By inverting the path from the random goal to the fixed goal and attaching it to the end of the path from the start state to the fixed goal, a path is created from the start state to the random final state. This approach, though, only works on problems where both the start state and random final state are reachable from the fixed goal. In addition, using this approach to solve the random goal problem is not the same as learning the knowledge needed to solve it.

These earlier approaches were also limited in the generality of their knowledge. Only Iba's system, Maclearn, applied its knowledge learned on one size of problem to any other size. Maclearn's limitation was that it always tried all of its knowledge. Since it learns macro-operators, each newly learned one increases the branching factor of the problem. Since Maclearn used best-first search, an intractable unconstrained method, the potential benefit of these new operators were offset by their cost. Given these limitations the largest problem solved by any of these earlier methods was the 6x6 35-puzzle. EASe does not have any of these limitations.

### 3.1 Indexing an Episode from Memory

For an episode learned on an 8-puzzle to apply to a 15-puzzle or 24-puzzle it must be indexed when it can be embedded within one of these larger problems. This is possible with many representations. The state representation we chose was that of an NxN matrix. A fixed offset for the row and column when indexing allows matching within a state. These offsets are bound when matching the current subgoal with the current subgoal for the context of an episode in memory. The effects of this offset are on indexing only. No modification to the impasse solution is made.

CONTEXT

| CURRENT | PROTECTED |
|---|---|
| position-1-4(blank) | position-1-1(1) |
| | position-1-2(2) |
| | position-1-3(3) |
| | position-2-4(4) |

STATE

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 7 |
| 15 | 5 | | 4 |
| 10 | 12 | 9 | 13 |
| 8 | 6 | 14 | 11 |

Figure 5: Impasse Example from the 15-puzzle

An example will better illustrate how the indexing takes place. Figure 5 gives an example impasse from a 15-puzzle problem. Here the goal is again to get the tiles in numeric order. Currently, tiles 1, 2, and 3 are solved and protected. To move tile 4 into its goal position requires a move up from its current location. The precondition for this move requires that tile 4 remain in its current position and that the blank be in the upper right corner, `position-1-4(blank)`.

In this example, the blank cannot be hill-climbed from its current position to its goal position while protecting all of the solved subgoals. Figure 3 illustrates an episode from memory indexed for this impasse. The episode is indexed by binding the current subgoal, `position-1-4(blank)`, to the current subgoal of the episode, `position-1-3(W)`. This binding results in `W` being bound to the blank with a row offset of 0 and a column offset of $-1$. Since all of the solved subgoals of the episode also bind to currently solved subgoals using this same offset, the impasse solution of the episode is returned to constrained search. In this example, this episode will successfully solve the impasse.

### 3.2 Generality of the Learned Episodes

In testing the capability of our system in the tile-sliding domain we were interested in investigating several issues. First, whether or not a small set of knowledge exists with enough generality to allow solving all problems. Second, whether or not this knowledge can be learned by training only on simple problems. These two issues question whether the transference assumption mentioned in section 2.4 holds in this domain. The final issue is whether or not the knowledge needed can be acquired tractably with iterative deepening search. To explore these issues a series of tests were conducted.

We began by training EASe in the tile-sliding domain. In training our system we developed an approach that we call convergence training. In it we train the system on one size of problem until it solves some number of them successively without learning anything new. This is the convergence test. Once passed, the size of the problem is increased and training continues. For our

| Problem Size | Problems Solved | Episodes Learned | Episode Length | | Search Required | |
|---|---|---|---|---|---|---|
| | | | Min | Max | Min | Max |
| 3x3 | 88 | 21 | 3 | 15 | 12 | 6831 |
| 4x4 | 98 | 10 | 3 | 20 | 23 | 17,342,971 |

Table 1: Tile-Sliding Learning Results

experiments we used 50 as the number of problems for the convergence test.

Each of the training problems were generated by first beginning with a state where all of the tiles were in numeric order. A new initial state was generated from this state by randomly applying some number of permutations. A permutation swaps any two tiles in a state. A random walk was then taken from this new initial state to generate the new goal state. For the 3x3 and 4x4 training problems 1000 permutations were applied to generate the initial state, and a random walk of 1000 moves used to generate the goal state. Note that a new initial state and final state was generated for each problem.

For the tile-sliding domain, EASe was trained on the 3x3 sized 8-puzzle and the 4x4 sized 15-puzzle. After this, when training on larger problems no further learning occurred. The episodes learned while training on these smaller problems proved sufficient to solve all additional ones. Table 1 gives information on the episodes learned. In total, 31 episodes were learned after training on 186 problems. Most of these were learned on the simpler 8-puzzle. The episodes learned varied in length from 3 moves to 20 moves. The cost to find the longest episode was more than 17 million nodes.

All of the episodes learned involved the placement of the blank as the current subgoal. The episodes varied in the number of protected subgoals that needed to be undone for further progress. The simplest episodes did not require undoing any protected subgoals. These episodes moved a blank around a protected tile. These were needed because the blank cannot be hill-climbed from behind a tile to in front of a tile. The most complex episodes required undoing as many as five protected subgoals.

To test the power of our approach before and after learning we conducted a series of tests with problems of varying size. We generated 10 random problems each from the 3x3, 4x4, 5x5, 10x10, 15x15, and 20x20 sized problems. We generated the problems as before, using random permutation for the initial state and a random walk for the final state. For the 3x3 and 4x4 puzzles, 2000 permutations and 2000 moves were used. For the 5x5, 10x10, 15x15, and 20x20 problems 10000 permutations and 10000 moves were used.

We then tested EASe on these problems with and without the learned episodes. When testing without the learned episodes, we allowed EASe to fall back on unconstrained search but did not allow it to learn. This allowed us to determine the amount of reduction in search provided by the learned episodes. For comparison, iterative-deepening A* (IDA*) using the Manhattan distance

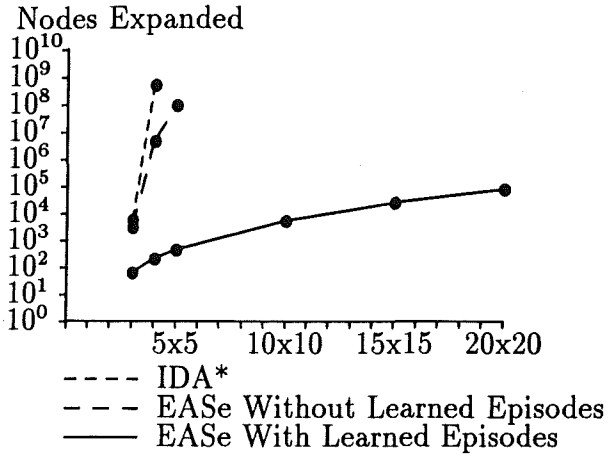


Figure 6: Performance Improvement from Learning

metric was also tested on the same problems. Note, that this comparison is somewhat unfair since IDA* finds optimal solutions. Still, it provides a nice base from which to compare the results. The results are plotted in Figure 6.

With the 31 episodes learned, EASe solved all of the problems from each of domains tested ranging from the 3x3 size to the 20x20 size. This demonstrates that for this domain the knowledge learned on the simpler 3x3 and 4x4 problems was sufficient for solving all additional problems. It is this characteristic of the domain that keeps it tractable. Without the learned episodes, only problems up to the 5x5 size were solvable. This clearly demonstrates that learning not only speeds up problem solving, but also increases the number of problems solvable.

The rate of growth in the search required for EASe without learning was still much smaller than that of IDA*. This is somewhat surprising given that EASe relied upon iterative deepening depth-first search to solve the impasses. This reduction was due solely to the effects of the subgoal decomposition and the use of hill-climbing for the simpler parts of the problem. This subgoal decomposition reduced the length of the subgoal solution, thus reducing the amount of search required. Note that with the 8-puzzle there is little difference between the performance without episodes and IDA*. The difference in the performance is only encountered when scaling to larger problems.

## 4 OPTIMIZATION WITH EASE

Our approach towards problem solving is well suited to optimization. In fact, we view all problem solving as an incremental process of taking varying sized steps towards completion. With an optimization problem, though, it may not be possible to actually generate the optimal solution. The goal for these problems is merely to find the best solution possible.

Finding optimal solutions to design optimization problems is in most cases intractable. It was our hypothesis, though, that a small set of knowledge would allow generating good solutions. We were interested in discovering whether or

not learning on simple problems would again provide us with this knowledge. To test EASe's capabilities with optimization problems we chose the logic synthesis and optimization task of VLSI design. This is a classic optimization task that is difficult both for people and computational approaches. Several application systems have been developed for this task and a wide variety of benchmark problems exist.

### 4.1 Logic Synthesis and Optimization

In logic synthesis, a functional specification of a circuit is mapped into combinational logic using a library of available components. These components are taken from a technology-specific library. The components available in a library as well as the performance characteristics of the components will vary depending upon the technology and particular manufacturer chosen. The synthesized circuit can be optimized for a variety of constraints. In testing our approach we chose to optimize the circuits for their critical path delay.

Each component in a library computes some boolean function. It has some number of inputs and one output. To compute the delay of a circuit requires a model of the delay of each component. Earlier work with this domain (Ruby & Kibler, 1991) used a simplified component model that consisted of a single delay value for each component. We elaborate this model to include for each input to a component its rise delay, fanout rise delay, fall delay, fanout fall delay, and load. This level of detail is necessary to operate on the benchmark problems developed in the logic synthesis research community.

A functional specification for a problem consists of a set of inputs, a set of outputs, and a definition for each of the outputs as a boolean function of some of the inputs and other outputs. The boolean operations allowed are `and`, `or`, and `not`. Operators in this domain map boolean expressions into the component for computing it. Other operators map components into their boolean expression. An operator for removing two successive inverters is also included. Although these operators are procedural, our approach operates as easily with them as it does STRIPS style operators.

The goal of a problem is then represented as the conjunction of two subgoals: (1) find a circuit that is realizable with components from the selected library, `realizable` and (2) optimize the circuit for the critical path delay time, `optimize-cp`. Hill-climbing finds a circuit that is realizable. Initially, the system has no knowledge of any operators for improving the critical path delay. It must learn how to optimize for the critical path delay time.

Figure 7 gives an example impasse from the logic synthesis domain. The initial state of this problem was the Boolean equation $f = a \wedge b \wedge c$. In the current impasse state, the subgoal `realizable` is solved and protected. The current subgoal is to optimize for the critical path delay, `optimize-cp`. Unfortunately, it is not possible for constrained search to improve upon the subgoal `optimize-cp` without undoing the protected subgoal `realizable`.

The sequence of states shown in Figure 7 lead from the impasse state to one where the critical path delay is improved. This new circuit is found with

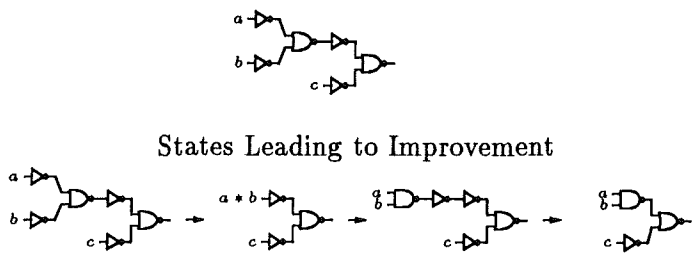


Figure 7: Example Impasse from Logic Synthesis

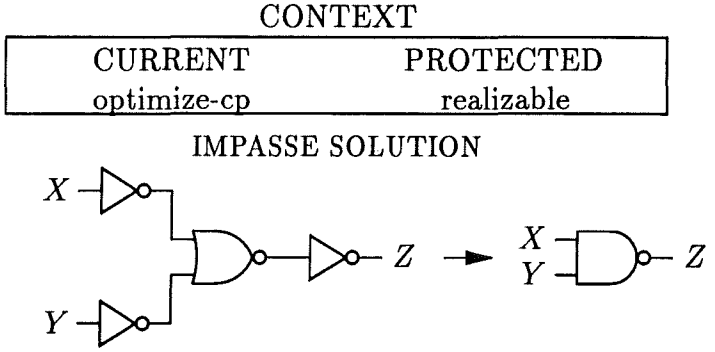


Figure 8: Example Episode from Logic Synthesis

the unconstrained method of iterative deepening. These states show that only part of the state needs to be made unrealizable for improvement on the critical path delay. It is only those parts of the state involved in the protected subgoal that needed to be undone, the subgoal `realizable`, that need to be included in the episode learned. After editing out the portions of the state not involved with the impasse, we end up with the episode shown in Figure 8. The impasse solution shown is the replacement rule representing the sequence of moves used to improve upon the subgoal `optimize-cp`. The variables X, Y, and Z can be bound to any subcircuit for indexing. Whenever the impasse solution is applied again, the bound left hand side is replaced in the circuit by the bound right hand side of the rule.

When indexing episodes from memory, they are ordered by the amount they can improve upon their current subgoal. Those likely to generate the largest improvement are tried first. These improvement values are computed when the episode is acquired and are not revised during problem solving.

### 4.2 Episode Augmented Search for Logic Synthesis

To demonstrate EASe could learn optimization knowledge we conducted a series of experiments. A component library was created with a few of the common components available from the MCNC benchmark library.[2] The com-

[2]Several libraries are available for anonymous ftp from "mcnc.org". The components used in the experiments were a subset of those from

| Problem Size | Problems Solved | Episodes Learned | Episode Length Min | Episode Length Max | Search Required Min | Search Required Max |
|---|---|---|---|---|---|---|
| 2-inputs | 73 | 3 | 1 | 4 | 1 | 16 |
| 3-inputs | 59 | 4 | 3 | 6 | 26 | 6179 |
| 4-inputs | 91 | 13 | 3 | 8 | 147 | 25754 |
| 5-inputs | 125 | 10 | 4 | 7 | 239 | 24680 |

Table 2: Logic Synthesis Learning Results

ponents included were: (1) 2-input nand, (2) 2-input nor, (3) 3-input nand, (4) 3-input nor, (5) inverter, (6) aoi21 ($\neg(a*b+c)$), and (7) oai21 ($\neg((a+b)*c)$).

We trained the system on a subset of random boolean functions with 2 to 5 inputs using the boolean connectives of *and*, *or*, and *not*. Functions generated consisted of $n$ terms connected by $n-1$ binary boolean operations, where $n$ was the number of inputs. Each term consisted of either one of the $n$ inputs or one of the inputs inverted. Each input was used in only one term. Since there were two choices for each boolean operation there are on order $2^{n-1}$ possible functions of this form with $n$ inputs. For each of these functions each of the inputs may or may not have been inverted, making the total number of possible functions with n-inputs on the order of $2^{n-1} * 2^n$ or $2^{2n-1}$.

We trained the system again using convergence training, with 50 as the number of problems for the convergence test. We trained on problems ranging from 2-inputs to 5-inputs. A maximum of 26000 nodes were allowed for unconstrained search during training. Table 2 gives the results of this training.

In total, 30 episodes were learned after training on over 348 problems. Most of the episodes were learned on the larger training problems. The length of learned episodes varied from 1 move to 8 moves with the maximum amount of search requiring over 25000 nodes. Note that the maximum length of a learned episode when training on problems with 4-inputs is greater than that when training on those with 5-input problems. This was due to growth in the branching factor given a constant maximum search of 26000 nodes.

Table 3 gives the impasse solutions from all of the episodes learned. The table is organized with the left hand side of the replacement rule given in the *before* column and the right hand side given in the *after* column. The terms X, Y, and Z can be bound to any subcircuit. When reapplied, the bound subcircuit in the *before* column is replaced by respective subcircuit in the *after* column.

The rules are organized by the format of their *before* subcircuit. The first group all involve subcircuits that consist of a single component. The second group consist of a single inverted component. Most of the rules, 22 out of 30, map a single component along with some inverters into some new configuration

the pub/benchmark/LGSynth89/wkslibrary/lib2.mis2lib library.

| Id | Before | After |
|---|---|---|
| 1 | aoi21(X,Y,Z) | nor2(nor2(not(X),not(Y)),Z) |
| 2 | aoi21(X,Y,Z) | nor2(not(nand2(X,Y)),Z) |
| 3 | nor3(X,Y,Z) | nor2(not(nor2(X,Y)),Z) |
| 4 | not(nand3(X,Y,Z)) | nor2(nand2(X,Y),not(Z)) |
| 5 | not(aoi21(X,Y,Z)) | nor2(nor2(X,Y),not(Z)) |
| 6 | not(oai21(X,Y,Z)) | aoi21(not(X),not(Y),not(Z)) |
| 7 | not(nand2(X,Y)) | nor2(not(X),not(Y)) |
| 8 | not(nor2(X,Y)) | nand2(not(X),not(Y)) |
| 9 | not(nor3(X,Y,Z)) | nand3(not(X),not(Y),not(Z)) |
| 10 | oai21(not(X),Y,Z) | nand2(nand2(X,not(Y)),Z) |
| 11 | aoi21(not(X),Y,Z) | nor2(nor2(X,not(Y),Z)) |
| 12 | aoi21(not(not(X)),not(Y),Z) | nor2(nor2(not(X),Y),Z) |
| 13 | nor3(not(X),Y,Z) | nor2(nand2(X,not(Y),Z)) |
| 14 | nor2(not(X),not(Y)) | not(nand2(X,Y)) |
| 15 | oai21(not(X),not(Y),Z) | nand2(nand2(X,Y),Z) |
| 16 | nand3(X,not(Y),not(Z)) | nand2(nor2(Y,Z),X) |
| 17 | nor3(X,not(Y),not(Z)) | nor2(nand2(Y,Z),X) |
| 18 | not(oai21(X,Y,not(Z))) | aoi21(not(X),not(Y),Z) |
| 19 | not(nor2(not(X),Y)) | nand2(X,not(Y)) |
| 20 | not(nand2(X,not(Y))) | nor2(not(X),Y) |
| 21 | not(nand3(not(X),Y,Z)) | nor3(X,not(Y),not(Z)) |
| 22 | not(oai21(not(X),Y,not(Z))) | aoi21(X,not(Y),Z) |
| 23 | nand2(X,nand2(Y,Z)) | oai21(not(Y),not(Z),X) |
| 24 | nand2(X,nor2(Y,Z)) | nand3(not(Y),not(Z),X) |
| 25 | nor2(X,nor2(Y,Z)) | aoi21(not(Y),not(Z),X) |
| 26 | nand2(nand2(X,not(Y),Z) | oai21(not(X),Y,Z) |
| 27 | nor2(X,not(nand2(Y,Z))) | aoi21(Y,Z,X) |
| 28 | aoi21(W,X,not(nor2(Y,Z))) | nor3(Y,Z,not(nand2(W,X))) |
| 29 | not(oai21(nand2(W,not(X),Y,Z))) | aoi21(nor2(not(W),X),not(Y),not(Z)) |
| 30 | not(not(X)) | X |

Table 3: Learned Impasse Solutions from Logic Synthesis

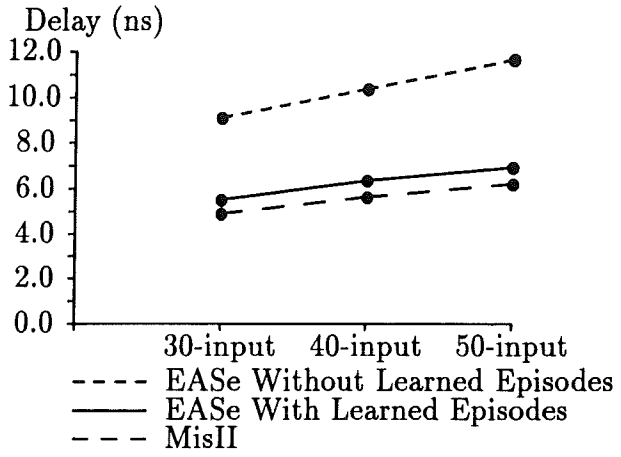


Figure 9: Average Performance on Random Problems

of components. The remaining replacement rules involve two components or the removal of two successive inverters.

In many ways learning in this domain was simpler than in the tile-sliding domain. The maximum search required, length of learned episodes, and even the number of learned episodes are all less than that required for the tile-sliding domain. What was difficult about this domain was the complex representation required for the problem space. The size and complexity of the program needed to represent and process the logic synthesis problems were orders of magnitude greater than for the tile-sliding domain. Because of the complexity of the state and operator representations required for the logic synthesis problems, it was not possible to search as much in this problem space as we did for the tile-sliding domain.

The knowledge learned in this domain is not complete like it was in the tile-sliding domain. If we were to search more of the problem space we would learn more episodes. EASe does not require complete knowledge but merely uses its knowledge to best effect. It uses its knowledge to generate solutions and assumes those generated are sufficient. Thus, problem solving remains tractable even in an intractable domain like this one. But we must evaluate the quality of the solutions found.

To test the quality of solutions generated by EASe a series of tests were conducted. We tested EASe on ten random problems each from a series of problems sizes. The sizes included were 30-input, 40-input, and 50-input random boolean functions. The results of these experiments are plotted in Figure 9. The average critical path delay of circuits generated by the system without learning, using goal-decomposition and hill-climbing alone, are also plotted in Figure 9 to illustrate how much using the learned episodes improved the quality of the solutions.

In order to judge the difficulty of these problems and the quality of the solutions generated we decided to compare our results to an existing application

system. We chose the MisII system (Brayton, Rudell, Sangiovanni-Vincentelli, Wang, 1987) because it is the standard for comparison in the logic synthesis research community. This system actually consists of a package of utility programs. These utility programs can be combined in different ways for different modes of operation. These modes are defined by *scripts*. Two different scripts were used for testing. The first was a default script included with the system. The second script was the simplest one possible. The simplest script returned the best results for the random problems, so it was the one used for testing. The results returned from our experiments with MisII are also plotted in Figure 9. Note, that although the MisII system had capabilities that allowed it to generate solutions not in the search space of our approach, EASe still generated circuits with critical path delay times that averaged within 13 percent of those generated by MisII.

### 4.3 Augmenting MisII with Episodes

The knowledge acquired by EASe is different from that usually encoded in an algorithm for solving a class of problems. A general algorithm encodes a core of knowledge for handling problems in the domain. This knowledge is similar to that used by the constrained problem solver in EASe. Learned episodes encode exceptions to the approach followed by the constrained problem solver. Specific episodes can also encode exceptions to a general algorithm.

To demonstrate that EASe could use the episodes learned to improve upon the solutions generated by an application system, a second experiment was conducted. For this second experiment we used the same library of components and goal specification as in the previous experiment. In this case, though, the input to EASe was a circuit that had already been optimized by MisII, not a functional specification.

Instead of using random problems for this experiment, we used benchmark problems from the logic synthesis community. These problems are designed to test logic synthesis application systems. We decided that these problems would present the best means for evaluating our approach as well. The set of problems chosen are available along with the libraries of components for anonymous ftp[3]. There are 64 problems of various sizes in the set chosen.

The problems in this set varied considerably in difficulty. We decided to test our approach on a subset of the problems, based on their size. The MisII package provides several measures for the size of a problem. The two used for the benchmark problems were: (1) the number of literals in the sum-of-product representation, and (2) the number of nodes in the network. Given these two measures, the problems fell into roughly 3 classes. The first class consisted of problems where the number of nodes in the network was less than 100 and the number of literals was less than 1000. This was the largest class and consisted of 39 problems. The second class consisted of problems where either the number of nodes was between 100 and 1000, or the number of literals

[3]The problems chosen are available for anonymous ftp from "mcnc.org" in the directory pub/benchmark/LGSynth89/mlexamples.

| Problem Name | Number of Nodes | Number of Literals | Critical Path Delay (ns) MisII | Critical Path Delay (ns) MisII+EASe | Improvement (%) |
|---|---|---|---|---|---|
| cm163a | 16 | 69 | 6.30 | 4.40 | 30.2 |
| cm42a | 13 | 35 | 4.17 | 3.10 | 25.8 |
| cht | 36 | 374 | 7.62 | 5.72 | 25.0 |
| sct | 40 | 236 | 5.93 | 4.87 | 17.9 |
| cm150a | 16 | 92 | 6.67 | 5.49 | 17.6 |
| ttt2 | 67 | 719 | 10.0 | 8.3 | 17.0 |
| pm1 | 31 | 98 | 3.48 | 2.94 | 15.6 |
| cm82a | 6 | 28 | 4.48 | 3.82 | 14.8 |
| pcler8 | 24 | 102 | 9.77 | 8.32 | 14.8 |
| cm85a | 24 | 68 | 6.49 | 5.64 | 13.1 |
| cmb | 14 | 69 | 5.41 | 4.76 | 11.9 |
| 9symml | 44 | 278 | 11.53 | 10.29 | 10.8 |
| x2 | 12 | 74 | 4.80 | 4.31 | 10.2 |
| ldd | 31 | 173 | 8.12 | 7.31 | 10.1 |
| lal | 71 | 258 | 5.61 | 5.05 | 9.9 |
| apex7 | 59 | 352 | 11.30 | 10.22 | 9.5 |
| pcle | 16 | 78 | 7.10 | 6.43 | 9.4 |
| f51m | 16 | 327 | 6.54 | 5.93 | 9.3 |
| cm162a | 19 | 74 | 6.61 | 6.02 | 9.0 |
| count | 47 | 174 | 13.45 | 12.30 | 8.6 |
| c8 | 48 | 363 | 5.48 | 5.03 | 8.2 |
| z4ml | 8 | 256 | 5.36 | 4.95 | 7.6 |
| cu | 23 | 98 | 4.83 | 4.50 | 6.7 |
| cm138a | 9 | 35 | 4.78 | 4.47 | 6.6 |
| myadder | 49 | 305 | 22.71 | 21.35 | 6.0 |
| comp | 55 | 200 | 14.63 | 13.77 | 5.9 |
| unreg | 32 | 144 | 3.66 | 3.48 | 4.7 |
| frg1 | 3 | 792 | 8.78 | 8.50 | 3.2 |
| parity | 15 | 60 | 7.91 | 7.71 | 2.6 |
| cc | 33 | 110 | 5.48 | 5.41 | 1.3 |
| C17 | 6 | 12 | 2.05 | 2.05 | 0.0 |
| b1 | 12 | 19 | 2.97 | 2.97 | 0.0 |
| majority | 12 | 21 | 3.01 | 3.01 | 0.0 |
| cm152a | 1 | 32 | 4.41 | 4.41 | 0.0 |
| cm151a | 9 | 45 | 5.74 | 5.74 | 0.0 |
| tcon | 16 | 56 | 1.62 | 1.62 | 0.0 |
| decod | 18 | 68 | 3.92 | 3.92 | 0.0 |
| mux | 6 | 142 | 8.02 | 8.02 | 0.0 |
| Average | 25.18 | 169.4 | 6.86 | 6.21 | 9.0 |

Table 4: EASe with Learned Episodes

was between 1000 and 10000. The third class consisted of a single problem with 14533 literals.

The first of these classes of problems were used to test EASe. Unfortunately, these measures of problem size are not perfect measures of the size of the circuit generated. One problem from the first class turned out to be much more difficult than all of the rest. It could not be completed within a few cpu hours, so it was removed from the test set.

When testing MisII on these problems we again used the two scripts (simplest and default) described earlier. For these problems there was no single best script for all of the problems chosen. Each problem was optimized by MisII using both scripts with the best results taken as the optimized circuit. EASe was then used to further improve these optimized circuits. Table 4 shows that significant improvements were possible on these benchmark problems. The improvements ranged from 30 percent to no improvement. The average improvement EASe generated on these 38 benchmark problems was 9.0%.

## 5 MODEL OF EASE

In this section we present an analysis of the effectiveness of EASe. The analysis requires strong assumptions about the domain and the problem-solving process. In particular, we will model the weak methods as bounded breadth-first or hill-climbing search. Our analysis will compare the situation where EASe has no memory to that where it has a complete memory. A complete memory is one with sufficient coverage such that the unconstrained problem solver is never needed. We will also demonstrate the significance of training on simple problems. We will explore two different models to demonstrate the generality of the results. These models differ in their choice for the constrained search method. The first model used bounded breadth-first search, while the second uses hill-climbing search.

### 5.1 Breadth-First Model

In our first model, we assume that constrained search does a breadth-first search to depth $d$ and unconstrained search does one to depth $l$. We assume that the branching factor is the constant $b$. We also assume all problems can be solved without the use of memory.

Before any learning takes place, problem solving will oscillate between constrained and unconstrained search. Suppose $c$ subproblems are solved by the constrained search, while unconstrained search solves $u$ subproblems. The solution cost for the $c$ subproblems solved by constrained search is $c * b^d$. Since the $u$ subproblems solved by unconstrained search must first be tried by constrained search their total solution cost is $u * b^d + u * b^l$. The total computation cost for the model without memory is thus bounded above by:

$$(c + u) * b^d + u * b^l. \tag{1}$$

Now suppose that sufficient learning takes place such that the uncon-

strained problem solver need never be called. We define $b_m$ as the number of episodes that will match an impasse and assume that it is the same for all impasses. Since all but the last of the episodes that match might fail to resolve the impasse, all episodes that match may need trying. The resulting search cost is bounded above by:

$$(c+u) * b^d + u * b_m. \tag{2}$$

Roughly speaking, the effect of memory is to replace the term $b^l$ by the term $b_m$. This demonstrates that as long as $b_m$ is not large an appropriate memory of past difficulties yields an exponential decrease in computation cost. Actually, the factor $b_m$ is not as important as the likelihood that an indexed episode will succeed in solving an impasse. In particular, if $s$ is the likelihood that a matched subgoal sequence from memory will succeed, then $b_m$ should be replaced by $1/s$.

### 5.2 Hill-Climbing Model

Another model of constrained search is hill-climbing. As above, let us first assume that when memoryless, the unconstrained method is sufficient to solve problems. Let $m$ be the length of the solution to a subgoal found using hill-climbing. Also let $l$ be the length of the solution to a subgoal found by the unconstrained problem solver. In this case equation 1 becomes:

$$(c+u) * m * b + u * b^l. \tag{3}$$

Now, after learning is complete, a bound on the computational cost is:

$$(c+u) * m * b + u * b_m. \tag{4}$$

As before, the factor $b_m$ can be replaced by $1/s$ where $s$ is the likelihood of success. In any case, the major effect is to replace $b^l$ by the factor $1/s$. In this case, a complete memory transforms an exponential search to a linear one.

### 5.3 Training on Simple Problems

From the previous analysis it appears that learning only speeds up problem solving. Problems must be solvable without learning if the needed episodes are to be learned. Fortunately, this is not the case. What is missing from the analysis is the significance of training on simple problems. In particular, it is unreasonable to assume that the branching factor is constant over all problems from the simplest to the most difficult.

Assume that unconstrained search is allowed to expand approximately $b^l$ nodes, rather than being bounded to search no deeper than $l$. This is a more realistic constraint, since it is computation time that is usually bounded. This means for problems with a larger branching factor, only a search to a lesser depth is possible.

Assume also a domain with a range of problems of different difficulties, where the branching factor of the easiest problems is $\sqrt{b}$ and the most difficult

is $b^2$. Given that the length of the impasse solutions is $l$, then learning on the easiest problems will only require expanding on order $b^{l/2}$ nodes. More interesting is that without learning, those problems with branching factor greater than $b$ are unsolvable given the search bound. Solving the most difficult problems without learning requires expanding on order of $b^{2*l}$ nodes, or $b^l$ times as many nodes as that allowed. It is only by learning these episodes on simpler problems and reusing them that the more difficult problems can be solved at all. Note that in many domains, like logic synthesis, there is no bound on problem difficulty. In these cases, the savings from learning will also grow without bound.

## 6 POWER OF EASE

The power of our approach stems from three sources. First, it decomposes a problem into simpler subproblems. Second, it learns and reuses previous problem solving episodes for difficult subproblems. Third, it effectively integrates these basic sources. We will now examine the value of these techniques.

### 6.1 Problem Decomposition

Simplifying a problem by decomposing it into simpler problems is a fundamental approach to problem solving. Problem decomposition simplifies a difficult problem into a set of subproblems. We take advantage of two different approaches to breaking a problem into subproblems: (1) breaking conjunctions, and (2) hill-climbing.

When operating on a problem that consists of a conjunction of subgoals, we begin by decomposing the initial problem into that of solving each of the individual conjuncts. These conjuncts are ordered using the openness heuristic. Once solved, a subgoal is protected. It is initially assumed that the ordered conjuncts, or subgoals, can be solved in order without undoing them once solved. Although all of the subgoals of a problem cannot usually be solved under this assumption, some usually can.

Hill-climbing is also used to decompose a problem. We decompose the problem of solving a subgoal to that of reducing its difference from completion. Thus a single subgoal is decomposed into several simpler subproblems each of which moves the subgoal closer to completion. Since we assume that any subproblem solution can be combined to form the final solution, these subproblems are independent. This is an effective means of decomposing a problem into simpler independent parts.

Decomposing a problem into simpler subproblems provides our approach with a great deal of power. Decomposing a problem into independent parts is known to greatly reduce the difficulty of a problem (Korf, 1987). Given a problem with branching factor $B$ and solution depth $D$ would require unconstrained search to expand on order $B^D$ nodes. If this problem can be reduced to $n$ independent subproblems each with depth $D/n$, the search tree is reduced to $n$ search trees each with size on the order of $B^{D/n}$ nodes. This reduction is possible as long as the subgoals are treated as though they were independent.

The techniques used by EASe to break difficult problems down into subproblems are relatively simple. Many additionaly techniques have been developed, such as abstraction (Knoblock, Tenenberg, Yang, 1991) and problem reformulation (Flann, 1989; Riddle, 1990). EASe could use these methods or any other ones to further assist in decomposing a problem. The use of additional problem decomposition methods in EASe is an interesting area for future research.

### 6.2 Memory of Previous Cases

The use of previous cases to solve problems is a well known technique (Lehnert, 1987; Hammond, 1990; Alterman, 1988). One of its advantages is that it can solve a problem regardless of its difficulty as long as it has solved a similar problem before. Its difficulties arise when trying to operate on random problems from domains with large problem spaces. Rajamoney & Lee (1991) demonstrated that case-based problem solving could be combined with other methods to solve novel problems from complex domains. It is through integration with other methods that we too use case-based problem solving.

To better understand case-based problem solving we begin by characterizing the search it requires. Given a problem it must first search for a relevant previous case. Once a relevant previous case is found, a search through a space of case modifications must be conducted to allow the case to operate on the current problem. Thus, case-based problem solving moves a search dependent upon the depth and branching factor of the problem to one dependent upon the number of cases in memory and the number of modifications required.

We use case-based problem solving to solve the difficult portions of a problem. Since we do not allow any modifications to the case, it remains tractable as long as the number of them required is not too large. Our approach depends upon generating the necessary cases with unconstrained search on simple problems. The cost of finding the cases is kept tractable by learning them on simple problems. We keep the number of cases required low by only learning them when constrained search fails to solve a subproblem.

### 6.3 Integrating Multiple Methods

It is through the integration of methods that our approach derives its real power. Each of the methods individually cannot solve difficult problems. This alone is an interesting result. It demonstrates why the evaluation of basic methods on complex problems is not always feasible. Only through their integration do they scale up to difficult problems.

SOAR provides us with an architecture for integrating methods through multiple problem spaces. It allows the use of multiple problem spaces by switching between them when an impasse occurs. For SOAR, an impasse occurs whenever a decision cannot be made in the current problem space. We use this same basic approach to integrate our weak methods. Constrained search is used until it is unable to make further progress, at which point an impasse occurs. The problem space then changes to a case-based problem solver with the goal of improving upon the impasse. Here episodes are indexed until

either improvement is made or all relevant episodes are tried. If that too fails, unconstrained search is tried and if successful a new episode learned. Learning in EASe, like SOAR, leads to less changes between problem spaces. After learning, EASe no longer needs unconstrained search.

EASe differs from SOAR in several ways, though. SOAR can dynamically choose a new problem space in response to an impasse. The problem spaces in EASe are predefined and static. EASe also learns episodes instead of chunks and indexes its knowledge differently. SOAR learns to avoid impasses. EASe continues to encounter impasses, but learns how to solve those encountered. Impasses signal EASe that memory needs to be tried. Still, using the basic SOAR concepts of multiple problem spaces and impasses to integrate the *right* weak methods into the *right* architecture we created an approach capable of solving complex and difficult problems.

## 7 RELATED WORK

To better understand our approach we compare it to other learning problem solvers. Korf (1985b) showed that if macros are learned for solving each subgoal of a problem and these macros depend only on the value of that subgoal and the previously solved subgoals, problems can be solved easily. Unfortunately this approach only works if a complete set of these macros can be learned. This requires that the problem state space be *operator decomposable*, with states represented by a vector of discrete state variables. We use a related approach but adopt a heuristic view. Our approach does not require complete knowledge, since it uses search to solve some of the subgoals. In addition, we add the use of hill-climbing to further decompose difficult problems.

Morris (Minton, 1985) learned macro-operators for a STRIPS style problem solver. Like our approach, it demonstrated that being selective about what to learn can improve performance. It demonstrated performance gains over MACROPS (Fikes, et al., 1972) by learning less. Still, each learned macro increased the branching factor of the problem. Unlike our approach, which reluctantly tries its new knowledge, Morris always tries its new knowledge first. In addition, Morris used a single search mechanism, means-ends analysis. We take advantage of two different search mechanisms, using the more costly to provide a source of knowledge for the less costly.

Maclearn (Iba, 1989) was also selective about what it learned. It learned macro-operators for a best-first algorithm that satisfied its peak-to-peak heuristic. A peak occurs when reaching a state where all of the surrounding states have worse heuristic values. While Maclearn attempts to learn how to move from one peak to another peak, EASe attempts only to learn to make it over difficult peaks, depending upon its constrained problem solver to move it towards the next peak. Rather than learn a macro-operator that must be tried along with all of the other operators, EASe learns an episode. These episodes are only indexed after all of the domain operators fail to lead to improvement.

In our earlier work with SteppingStone (Ruby & Kibler, 1989; Ruby & Kibler, 1991) we developed the basic architecture used by EASe. Stepping-

Stone, though, used a different representation for its problem solving knowledge. It learned an abstract plan for solving an impasse, rather than the impasse solution itself. This plan was represented as a sequence of subgoals, or steppingstones. EASe demonstrates that the architecture developed with SteppingStone can use other components and still generate a powerful problem solver.

## 8 CONCLUSIONS

The goal of this research has been to find and understand methods for solving difficult problems. Most well understood methods are weak and unable to solve complex problems. With our research we have demonstrated that when these weak methods are integrated it is possible to create a powerful problem solver. Our goal was a tractable problem solver for complex problems, so we combined the tractable and complementary methods of hill-climbing (with its global knowledge) and case-based problem solving (with its local knowledge). We made iterative deepening tractable for learning by using it only to solve simple problems.

Solving complex problems not only requires a good problem solver, it also requires simplifying the problem as much as possible. We used problem decomposition to simplify difficult problems. Goals consisting of the conjunction of subgoals were split into their component subgoals and ordered. We used hill-climbing to further decompose problems into episodes that improve upon the solution.

We demonstrated our approach on random problems from the tile-sliding domain. We found that with a small memory of episodes we were able to solve all random problems in this domain. We found a similar result with the logic synthesis and optimization task of VLSI design. Here we found that we did almost as well as the standard application system in the domain. In addition, we showed we could actually improve upon the solutions generated by the application system.

Finally, we provide analytical results with a model of our approach to explain our empirical results. These analytical results demonstrate how an exponential process is reducible to a linear one. These analytical results help us to better understand our approach and the results achieved.

These results have demonstrated our approach will work whenever a problem can be decomposed into: (1) simple subproblems, and (2) difficult subproblems that recur. The simple subproblems are defined as those solvable by weak methods. The difficult subproblems must not only recur, but also be solvable. It may not be possible to find the solution to them in general, but must be possible in at least idealized training examples. If unsolvable by automated methods, an expert can also generate the cases needed. As long as a small number of episodes can cover all of the difficult subproblems, this approach will be successful.

We now intend to extend our research in two directions. First, given our success with optimization problems from logic synthesis, we are now interested

in exploring scheduling optimization problems. Scheduling problems occur in domains ranging from manufacturing to software engineering. They pose a difficult challenge to current computational methods. Success in this domain offers the potential for important contributions not only to machine learning but to scheduling applications areas as well. The second direction we intend to take this research is towards a better formalization of our results. Our research has demonstrated the power and generality of EASe empirically, but our analytical work has not yet fully formalized these results. A more complete formalization will yield a better understanding of when the problem solving of EASe will succeed.

**Acknowledgments**

Thanks go to Karl Schwamb, Steve Hampson, Piew Datta, Steve Morris, Yousri El Fattah, Margaret Elliott, Caroline Ehrlich and the rest of UCI Machine Learning group for discussions and comments on earlier drafts of this paper.

**References**


Alterman, R. (1988). Adaptive Planning. *Cognitive Science, 12*, 393–421.

Bradtke, S., & Lehnert, W. (1988). Some experiments with case-based search. *Proceedings of the Seventh National Conference on Artificial Intelligence* (pp. 133–138). St. Paul, MN: Morgan Kaufmann.

Cheng, J., & Irani, K. B. (1989). Ordering problem subgoals. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 931–936). Detroit, MI: Morgan Kaufmann.

Flann, N. S. (1989). Learning appropriate abstractions for planning in formation problems. *Proceedings of the Sixth International Workshop on Machine Learning* (pp. 235–239). Ithaca, NY: Morgan Kaufmann.

Golding, A. R. & Rosenbloom, P. S. (1991). Improving rule-based systems through case-based reasoning. *Proceedings of the Ninth National Conference on Artificial Intelligence* (pp. 22–27). Anaheim, CA: AAAI Press.

Hammond, K, J. (1990). Explaining and repairing plans that fail. *Artificial Intelligence, 45*, 173–228.

Iba, G. A. (1989). A Heuristic Approach to the Discovery of Macro-operators. *Machine Learning, 3*, 285–317.

Kibler, D., & Porter, B. (1983). Episodic Learning. *Proceedings of the National Conference on Artificial Intelligence* (pp. 191–196). Washington, D.C.: Morgan Kaufmann.

Knoblock, C. A., & Tenenberg, J. D., & Yang, Q. (1991). *Proceedings of the Ninth National Conference on Artificial Intelligence* (pp. 692–697). Anaheim, CA: AAAI Press.

Korf, R. E. (1985a). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence, 27*, 97–109.

Korf, R. E. (1985b). *Learning to Solve Problems by Searching for Macro-Operators.* Boston, MA: Pittman Advanced Publishing Program.

Korf, R. E. (1987). Planning as search: A quantitative approach. *Artificial Intelligence, 33*, 65–88.

Laird, J. E., Newell, A., & Rosenbloom, P. S. (1987). SOAR: An Architecture for General Intelligence. *Artificial Intelligence, 33*, 1–64.

Langley, P. (1985). Learning to Search: From Weak Methods to Domain-Specific Heuristics. *Cognitive Science, 9*, 217–260.

Lehnert, W. G. (1987). Case-based problem solving with a large knowledge base of learned cases. *Proceedings of the Sixth National Conference on Artificial Intelligence.* Seattle: Morgan Kaufmann.

Minton, S. (1985). Selectively generalizing plans for problem solving. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* (pp. 596–600). Los Angeles, CA: Morgan Kaufmann.

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

Riddle, P. (1990). Automating problem reformulation. In D. P. Benjamin (Ed.), *Change of Representation and Inductive Bias.* San Mateo, CA: Morgan Kaurmann.

Ruby, D., & Kibler, D. (1988). Exploration of case-based problem solving. *Proceedings of the Case-Based Reasoning Workshop* (pp. 345–356). Clearwater, FL: Morgan Kaufmann.

Ruby, D., & Kibler, D. (1989). Learning subgoal sequences for planning. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 609–614). Detroit, MI: Morgan Kaufmann.

Ruby, D., & Kibler, D. (1991). SteppingStone: An empirical and Analytical Evaluation. *Proceedings of the Ninth National Conference on Artificial Intelligence* (pp. 527–532). Anaheim, CA: AAAI Press.

Rajamoney, S. A., & Lee, H. (1991). *Prototype-based reasoning: An integrated approach to solving large novel problems. Proceedings of the Ninth National Conference on Artificial Intelligence* (pp. 34–39). Anaheim, CA: AAAI Press.