Design of Representation-Changing Algorithms

Eugene Fink

February 1995

CMU-CS-95-120

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research is supported by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330. The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Wright Laboratory or the U. S. Government.

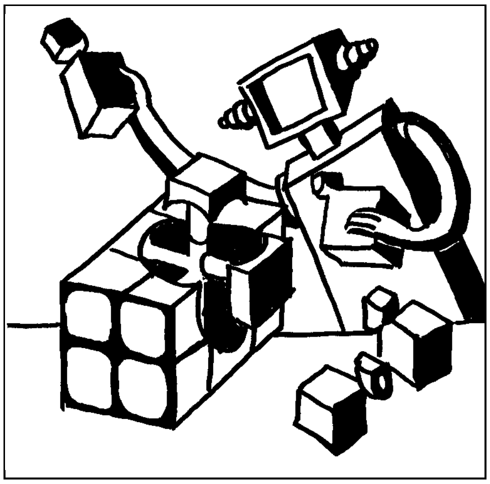
Keywords: problem solving, learning, problem reformulation.

Abstract

We explore methods for improving the performance of AI problem-solvers by automatically changing problem representations.

The performance of all problem-solving systems depends crucially on problem representation. The same problem may be easy or difficult to solve depending on the way we describe it. Researchers have designed a variety of learning algorithms that deduce important information from the description of the problem domain and use the deduced information to improve the representation. Examples of these representation improvements include generating abstraction hierarchies, replacing operators with macros, and decomposing complex problems into subproblems. There has, however, been little research on the common principles underlying representation-improving algorithms and the notion of useful representation changes has remained at an informal level.

We present preliminary results on a systematic approach to the design of algorithms for automatically improving representations. We identify the main desirable properties of such algorithms, present a framework for formally specifying these properties, and show how to implement a representation-improving algorithm based on the specification of its properties. We illustrate the use of this approach by developing novel algorithms that improve problem representations.



Contents

1 Introduction

The performance of all problem-solving systems depends crucially on problem representation. The same problem may be easy or difficult to solve depending on the way we describe it. Psychologists and AI researchers have accumulated much evidence of the importance of good representations for human problem-solvers [Newell and Simon, 1972; Simon *et al.*, 1985; Kaplan and Simon, 1990] and AI problem-solving systems [Newell, 1966; Amarel, 1968; Korf, 1980; Tamble *et al.*, 1990].

People often simplify problems by changing their description, which is a crucial skill for mathematicians [Polya, 1957], physicists [Qin and Simon, 1992; Larkin and Simon, 1987], economists [Tabachneck, 1992; Larkin and Simon, 1987], and experts in many other areas [Newell and Simon, 1972; Gentner and Stevens, 1983].

The purpose of our research is to automate the process of changing and improving problem representations. We report preliminary results in developing a general model of representation changes and systematic approach to the design of algorithms that automatically improve problem representations. We illustrate the use of this approach by designing novel representation-improving algorithms.

We begin by discussing the role of representation in problem solving (Sections 1.1), reviewing the previous work on representation changes (Section 1.2), and presenting an overview of the results described in the paper (Section 1.3).

1.1 Motivation: The role of representation

A *problem representation* in an AI problem-solving system is the input to the system. In most problem-solving systems, it includes the description of the operators in a problem domain, the initial and goal states of a problem, and possibly some other information, such as control rules and an abstraction hierarchy. The information given directly as an input is called *explicit*, and the information that can be deduced from the input is called *implicit*. Every problem representation leaves some information implicit.

Explicit representation of important information improves the performance. For example, we may improve the efficiency of a problem-solving system by encoding useful information about the domain in control rules [Minton, 1988] or an abstraction hierarchy [Knoblock, 1994]. On the other hand, explicit representation of irrelevant information decreases efficiency: if we do not mark such information as unimportant for the problem, the system attempts to use it, which takes extra computation and may lead the system to explore useless branches of the search tree. For example, if we add unnecessary operators to the domain description, and if these operators may (but need not) be used in solving a problem, the branching factor of search increases and the efficiency decreases.

Different problem-solving algorithms use different information about the domain and, therefore, perform efficiently with different representations [Stone *et al.*, 1994]. There is no "universal" representation that works well with all algorithms. The task of finding a good representation is usually left to the human user.

Many researchers have addressed the representation problem by designing learning algorithms that deduce important information from the domain description [Newell *et al.*, 1960;

Allen *et al.*, 1992; Carbonell, 1990], which includes learning control rules [Langley, 1983; Laird *et al.*, 1986; Minton, 1988; Veloso and Borrajo, 1994], generating abstraction hierarchies [Knoblock, 1994], replacing operators with macros [Korf, 1985; Mooney, 1988], and reusing past problem-solving cases [Carbonell, 1983; Hall, 1987; Veloso, 1994].

These learning algorithms, however, are themselves representation-dependent: they easily learn useful information with some problem descriptions, but become helpless with others. For example, the ALPINE algorithm for learning abstraction hierarchies [Knoblock, 1990] usually fails to generate a hierarchy when the domain contains unnecessary additional operators or the operator descriptions are too general [Knoblock, 1991a; Fink and Yang, 1992]; however, if the user selects a suitable domain description, ALPINE becomes very effective in reducing complexity of problem solving. Algorithms for learning control rules are often ineffective for a too general or too specific description of predicates and operators [Veloso and Borrajo, 1994], but again we may improve their performance by using a suitable representation.

To ensure the effectiveness of a learning algorithm in finding useful features of the problem domain, the human user needs to find a good representation for this algorithm. An important next step in AI research is to develop a system that automatically finds good representations.

1.2 Related work

The importance of good representations has long been recognized by experts in many areas, most notably mathematicians [Polya, 1957]. Simon, in collaboration with several other researchers, has analyzed the role of representation in mathematics [Larkin and Simon, 1987], physics [Larkin and Simon, 1981; Larkin and Simon, 1987; Qin and Simon, 1992], and economics [Tabachneck, 1992], and demonstrated that the use of good representations is essential for reasoning in these areas. He also showed that changes in a problem description may drastically affect the subjects' ability to solve the problem [Simon, 1975; Hayes and Simon, 1977; Simon *et al.*, 1985; Kaplan and Simon, 1990].

Kaplan and Simon studied representation changes by human subjects when solving the Mutilated-Checkerboard problem [Kaplan and Simon, 1990] and implemented a program that models the human reasoning on this problem [Kaplan, 1989].

Newell was first to discuss the role of representation in AI problem solving: he showed that the complexity of reasoning in some games and puzzles strongly depends on the representation [Newell, 1965; Newell, 1966]. Later, Newell with several other researchers implemented the Soar system [Laird *et al.*, 1987; Newell, 1992], capable of using different descriptions of a problem domain to facilitate problem solving and learning. Soar, however, does not generate new representations; the human user must provide all domain descriptions. A similar approach was used in the FERMI expert system [Larkin *et al.*, 1988], which automatically selects a representation for a given problem among several hand-coded representations.

Research on the automatic generation of new problem representations has been limited to several *special cases* of representation changes, including systems for decomposing a problem into subproblems [Newell *et al.*, 1960], generating abstraction hierarchies [Knoblock, 1994; Christensen, 1990; Bacchus and Yang, 1994], replacing operators with macros [Korf, 1985; Shell and Carbonell, 1989], replacing problems with similar simpler problems [Hibler, 1994], changing the search space by learning control rules [Minton, 1988; Etzioni, 1993; Pérez and Etzioni, 1992; Veloso and Borrajo, 1994], and reusing past problem-solving episodes in analogical reasoning [Carbonell, 1983; Veloso, 1994]. There has, however, been little research on the common principles underlying different types of representation-changers.

Researchers have developed theoretical frameworks for some special cases of representation changes, most notably generating abstraction hierarchies [Korf, 1987; Knoblock, 1991b; Knoblock *et al.*, 1991; Bacchus and Yang, 1992; Giunchiglia and Walsh, 1992], replacing operators with macros [Korf, 1985; Korf, 1987; Mooney, 1988], and learning control rules [Cohen, 1992].

A generalized model of representation changes was suggested by Korf, who formalized the concept of problem reformulation based on the notions of isomorphism and homomorphism of search spaces [Korf, 1980]. Korf's model, however, does not address "a method for evaluating the efficiency of a representation relative to a particular problem solver and heuristics to guide the search for an efficient representation for a problem" ([Korf, 1980], page 75), whereas the use of such heuristics is essential for developing an efficient representation-changing system.

To summarize, the results in representation changes in problem solving are still very limited. The main open problems include the formalization of methods used in designing representation-changing algorithms, the application of these methods to the design of new representation-changers, and the development of a unified theory of reasoning with different representations.

1.3 Overview of the paper

The review of the previous work shows that AI researchers have long recognized the importance of automatically improving representations and have implemented algorithms for several special cases of representation changes. There has, however, been little research on the general principles used in the design of representation-changing algorithms and methods for developing new algorithms. The results in developing a formal model of representation changes are also limited, and the notion of useful representation changes has remained at an informal level. Because of the lack of techniques and guidelines for the design of representation-changing algorithms, implementing a new representation-changer is usually a complex research problem.

In this paper, we present preliminary results on a systematic approach to the design of representation-changing algorithms. We identify the main desirable properties of such algorithms and describe a method for formally specifying these properties, which enables us to abstract the major decisions in the development of a representation-changer from implementational details. We show how to write a formal specification of the important properties of a representation-changing algorithm and how to use this specification to implement the algorithm. We illustrate the use of this approach by developing novel representation-changers.

The presentation of our results is organized in four sections. In Section 2, we give three examples of representation changes that drastically improve the efficiency of problem solving. In Section 3, we outline a general model of representation-changing algorithms and present a systematic approach to the design of such algorithms. To illustrate the use of this approach, we develop and analyze a novel representation-changing algorithm, which improves the efficiency of problem solving by removing unnecessary operators from a domain description. In Section 4, we show the application of our approach to the design of three other representation-changers. Finally, in Section 5, we summarize the results and discuss some research problems on automatic representation changes.

2 Examples of representation changes

For every problem-solving system and almost every problem, even a simple one, we can find a representation that makes the problem very hard or even unsolvable. These hard representations may not be cumbersome or artificial: a natural representation of a problem is often inappropriate for problem solving. The user must find a representation that ensures efficiency, which may be a difficult task: first, the user may not be sufficiently familiar with the system to determine which representation is appropriate; second, finding a good representation is often a complex creative task, which may involve extensive search.

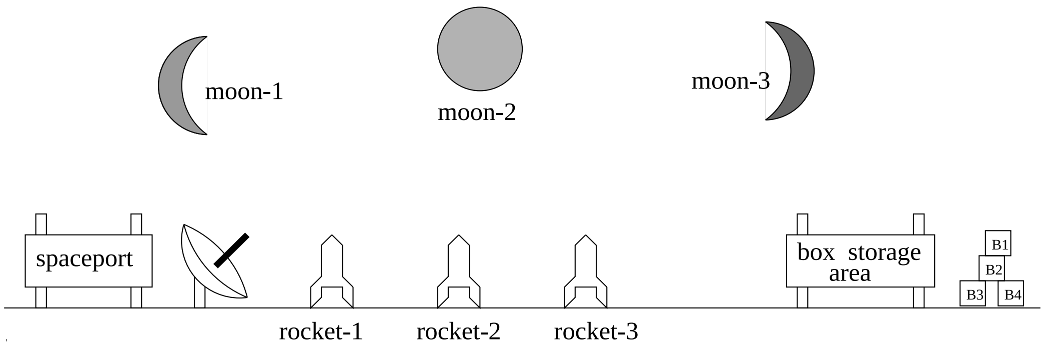
We present three examples of situations where changing the representation is essential for efficient performance of the PRODIGY system. First, we show two examples of simple problems that are very hard for PRODIGY, even though their representation is natural for the user; but after we change the descriptions of these problems, PRODIGY solves them efficiently (Sections 2.1 and 2.2). Then, we describe a complex problem that can be considerably simplified by finding a good representation (Section 2.3).

2.1 Three-rocket transportation

Consider a planning domain with a planet, three moons, three rockets, and several boxes (see Figure 1a) [Stone and Veloso, 1994]. Initially, all boxes and all three rockets are on the planet. A rocket can carry any number of boxes to any moon. After a rocket has arrived to its destination, it *cannot* be refueled and used again; thus, each rocket can be launched only once. The task of a problem-solving system is to find a plan for delivering certain boxes to certain moons. (We do not care about the resulting locations of rockets, as long as every box has reached its proper destination.) For example, if we want to send box-1 and box-2 to moon-1 and box-3 to moon-3, we can achieve the goal with the following plan:

> Load box-1 and box-2 into rocket-1 and box-3 into rocket-2.
> Send rocket-1 to moon-1 and rocket-2 to moon-3.
> Unload both rockets.

To describe a current state of the domain, we have to specify the locations of each rocket and each box, which can be done with three predicates: (at <rocket> <place>), (at <box> <place>), and (in <box> <rocket>), where the "<..>" brackets denote variables; for example, <rocket> is a variable that denotes an arbitrary rocket. We obtain literals describing a state of the domain by substituting specific constants for variables; for example, the literal (at rocket-1 planet) means that rocket-1 is on the planet and (in box-1 rocket-1) means that box-1 is inside rocket-1. The basic operations in the domain, called *operators*, are described by their *preconditions* and *effects*. All preconditions of



(a) Three-Rocket Transportation domain.

```
Operator LOAD                || Operator UNLOAD               || Operator FLY
Parameters:                  || Parameters:                   || Parameters:
  <box> <rocket> <place>     ||   <box> <rocket> <place>      ||   <rocket> <moon>
Preconds:                    || Preconds:                     || Preconds:
  (at <box> <place>)         ||   (in <box> <rocket>)         ||   (at <rocket> planet)
  (at <rocket> <place>)      ||   (at <rocket> <place>)       || Effects:
Effects:                     || Effects:                      ||   Del: (at <rocket> planet)
  Del: (at <box> <place>)    ||   Del: (in <box> <rocket>)    ||   Add: (at <rocket> <moon>)
  Add: (in <box> <rocket>)   ||   Add: (at <box> <place>)     ||
```

(b) Initial representation of the operations in the domain.

```
Operator                        || Operator                        || Operator
  FLY-ROCKET-1-TO-MOON-1        ||   FLY-ROCKET-2-TO-MOON-2        ||   FLY-ROCKET-3-TO-MOON-3
Preconds:                       || Preconds:                       || Preconds:
  (at rocket-1 planet)          ||   (at rocket-2 planet)          ||   (at rocket-3 planet)
Effects:                        || Effects:                        || Effects:
  Del: (at rocket-1 planet)     ||   Del: (at rocket-2 planet)     ||   Del: (at rocket-3 planet)
  Add: (at rocket-1 moon-1)     ||   Add: (at rocket-2 moon-2)     ||   Add: (at rocket-3 moon-3)
```

(c) Search-saving representation of the `fly` operation.

Figure 1: Changing the representation of the Three-Rocket Transportation domain to reduce PRODIGY's search: problem solving with the initial representation requires an extensive search, whereas the new representation of the `fly` operation enables PRODIGY to solve transportation problems with little search.

an operator must hold before the execution of the operator (that is, the preconditions are conjunctive); the effects are the results of the execution.

The user may describe the operations in the Three-Rocket domain by the three operators shown in Figure 1(b). This description, however, makes the problem hard for PRODIGY: the system tries to use the same rocket for transporting boxes to different moons. PRODIGY performs a long search to discover that each rocket can go to only one moon [Stone and Veloso, 1994]. If we increase the number of moons and rockets, the problem-solving time grows exponentially. We can use control rules to reduce the search, but the Three-Rocket domain requires a complex set of rules, which are difficult to hand-code or learn automatically.

We can, however, improve the efficiency of PRODIGY by replacing the `fly` operator with three more specific operators, shown in Figure 1(c). These new operators encode explicitly the knowledge that each rocket flies to only one moon. The use of these operators allows PRODIGY to solve three-rocket transportation problems almost without search. Thus, we have removed some actions from the domain, leaving a subset of actions sufficient for solving all transportation problems.

The new domain description works only for three rockets and three moons, whereas the original description worked for any number of rockets and moons; thus, we have improved efficiency by limiting the set of problems that we can solve.

2.2 Tower of Hanoi

We now show an example of a representation change that enables the ALPINE algorithm [Knoblock, 1994] to generate an abstraction hierarchy, which reduces PRODIGY's search.

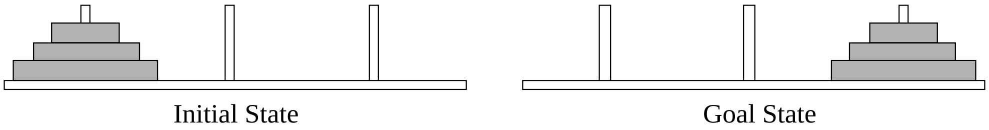
We consider the Tower-of-Hanoi puzzle with three disks (Figure 2a). The user may describe the states of this puzzle with a single predicate, `(on <disk> <peg>)`, and the operations with the operator shown in Figure 2(b). Most problem-solvers find the solution to the puzzle by an extensive search; the search time grows exponentially with the length of a solution plan. The search could be reduced by using an abstraction hierarchy of predicates [Knoblock, 1991a], but the problem description in Figure 2(b) does not allow us to generate a hierarchy of predicates, since this description contains only one predicate.

We may remedy the situation by replacing the predicate `(on <disk> <peg>)` with three more specific predicates, `(on small <peg>)`, `(on medium <peg>)`, and `(on large <peg>)`, which specify, respectively, the positions of the small, medium, and large disk. Using these three predicates, we generate a new description of the operations, shown in Figure 2(c). Given this new representation, the ALPINE algorithm generates the three-level abstraction hierarchy shown in Figure 2(d), which reduces the complexity of PRODIGY's search from exponential to linear in the length of a solution.

We again improved efficiency by limiting the set of problems that we can solve: the new description works only for three disks, whereas the old one worked for any number of disks.

2.3 Interurban transportation

We next describe a complex planning problem that can be simplified by decomposing it into subproblems. We consider a transportation domain with four buses and an airplane, which



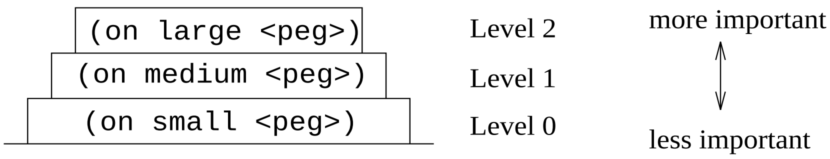
(a) Tower-of-Hanoi puzzle.

```
Operator MOVE
Parameters: <disk> <from-peg> <to-peg>
Preconds: forall <other-disk>
            if (smaller <other-disk> <disk>)
              then ((not (on <other-disk> <from-peg>))
                    and (not (on <other-disk> <to-peg>)))
Effects:
  Del: (on <disk> <from-peg>)
  Add: (on <disk> <to-peg>)
```

(b) Initial representation of the operations in the puzzle.

```
Operator MOVE-SMALL         || Operator MOVE-MEDIUM        || Operator MOVE-LARGE
Parameters:                 || Parameters:                 || Parameters:
 <from-peg> <to-peg>        ||  <from-peg> <to-peg>        ||  <from-peg> <to-peg>
Preconds:                   || Preconds:                   || Preconds:
 (on small <from-peg>)      ||  (on medium <from-peg>)     ||  (on large <from-peg>)
Effects:                    ||  (not (on small <to-peg>))  ||  (not (on small <to-peg>))
 Del: (on small <from-peg>) ||  (not (on small <from-peg>))||  (not (on small <from-peg>))
 Add: (on small <to-peg>)   || Effects:                    ||  (not (on medium <to-peg>))
                            ||  Del: (on medium <from-peg>)||  (not (on medium <from-peg>))
                            ||  Add: (on medium <to-peg>)  || Effects:
                                                           ||  Del: (on large <from-peg>)
                                                           ||  Add: (on large <to-peg>)
```

(c) Representation that enables ALPINE to generate an abstraction hierarchy.



(d) Abstraction hierarchy of predicates generated by ALPINE.

Figure 2: Changing the representation of the Tower-of-Hanoi puzzle to generate an abstraction hierarchy: the ALPINE algorithm fails to generate a hierarchy for the initial representation, but it generates a three-level hierarchy for the new representation.

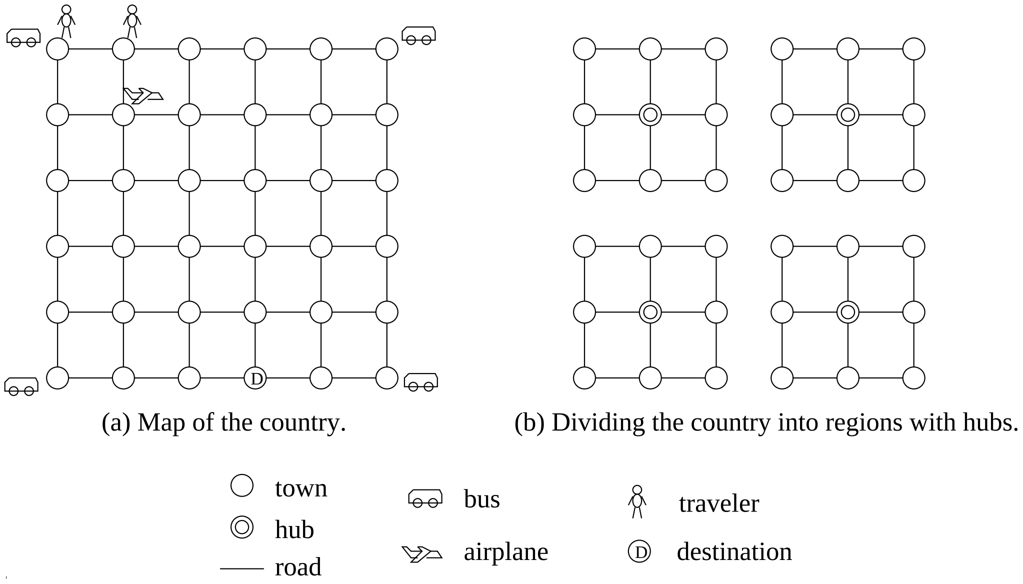


Figure 3: Changing the representation of the Interurban Transportation domain: we simplify planning problems by dividing the country into regions with hubs; the airplane flies only between hubs, whereas the buses deliver people to and from hubs within regions.

move between towns of a small country. The country contains thirty-six towns, connected by roads as shown in Figure 3(a). The buses travel along roads, whereas the airplane can fly between any two towns.

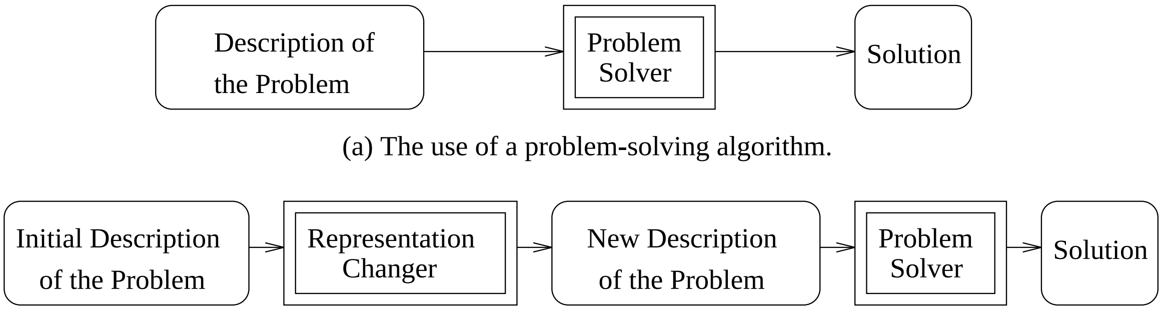
We determine the quality of a plan in this domain by the total cost of operations in the plan. We define the cost of a bus ride between adjacent towns as 1 and the cost of a flight between any two towns as 2. The cost of a plan is the sum of the costs of all its operators.

For example, suppose that the initial state is as shown in Figure 3(a) and we have to deliver the two travelers to the town marked by "D" (for "destination"). We may solve this problem as follows: the bus takes both people to the airplane, and then the airplane takes them to their destination. The cost of this plan is 4, since it involves two bus rides between adjacent towns and one flight.

Now suppose that we have the list of people who need to travel during some ten-day period. For each traveler, the list shows her initial location, destination, and the day of her travel. For each day of the ten-day period, we must design a plan for transporting the travelers to their destinations.

Finding a near-optimal transportation plan is a complex task, especially if we must transport people to many different towns. We may, however, simplify the problem by decomposing it into subproblems: we divide the country into four regions, as shown in Figure 3(b), and use the central town of each region as a hub. The airplane flies only between hubs, whereas the buses deliver people to and from hubs within regions.

The use of hubs enables a problem-solver to find a solution without much search. If most problems involve travel from and to multiple, randomly selected towns, the solutions based on this representation are usually close to optimal.



(a) The use of a problem-solving algorithm.

(b) Changing the representation before applying a problem-solving algorithm.

Figure 4: Representation changes in problem solving: a representation-changing algorithm generates a new description of the problem and then a problem-solving algorithm uses the new description to solve the problem.

3 Foundations of the systematic approach

We have shown that the use of "good" representations may significantly improve the efficiency of problem solving. We now present a general model of representation-improving algorithms and a systematic approach to the design of such algorithms. We describe main classes of representation changes (Section 3.1), give an example of a representation-changing algorithm, called *Operator-Remover* (Section 3.2), and use this example to illustrate our model and methods for developing representation-changers (Section 3.3). We show that the *Operator-Remover* algorithm is able to automatically perform the representation change in the Three-Rocket domain, described in Section 2.1.

3.1 Representation changes in problem solving

We begin by discussing the role of representation changes in problem solving and listing several classes of representation changes.

In Figure 4(a), we illustrate the use of a problem-solving algorithm: the algorithm inputs a description of a problem and outputs a solution to the problem. The input must satisfy certain semantic and syntactic rules, called the *input language*.

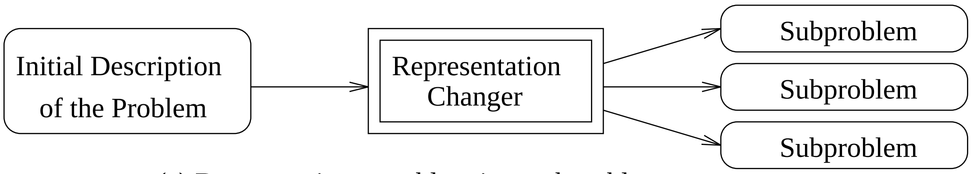
If the description of a problem does not obey the rules of the input language, we must "translate" it into the input language before using the algorithm (Figure 4b) [Hayes and Simon, 1974; Hayes and Simon, 1976]. When the description obeys the rules of the input language, we may still want to change it in order to simplify the problem.

The process of converting the initial description of a problem into an input to a problem-solving algorithm is called a *representation change*. It may serve two goals:

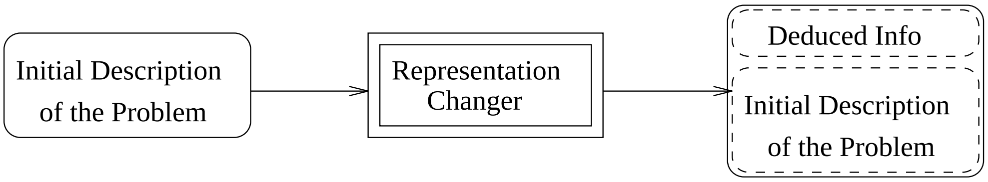
(1) translating the problem description to the input language of the problem-solver

(2) and simplifying the problem in order to improve the efficiency of problem solving.

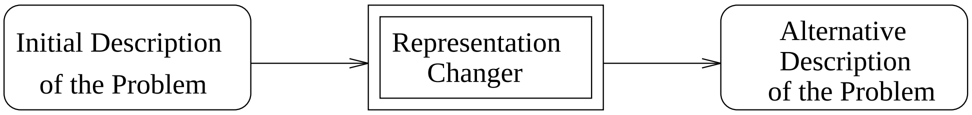
We will concentrate on the efficiency-improving representation changes. We distinguish two main classes of representation changes: decomposing a problem into subproblems and reformulating a problem.



(a) Decomposing a problem into subproblems.



(b) Enhancing a problem description:
deducing some information and adding it to the initial description.



(c) Replacing a problem description:
adding deduced information and discarding parts of the old information.

Figure 5: Main classes of representation changes in problem solving.

Decomposing a problem into subproblems.
We can often decompose a hard problem into several simpler subproblems (Figure 5a) [Polya, 1957], as we did in the Interurban-Transportation example (Section 2.3). To perform such a decomposition, we need some approximate measure of the difficulty of problems in the domain. Subproblems may interact with each other, in which case certain subproblems must be solved before others. Abstraction problem solving is a typical example of such a situation: the first subproblem is finding a solution on the abstract level and the other subproblems are refining parts of the abstract solution on the concrete level.

Reformulating a problem.
Converting the initial description of a problem into a new description, used as an input to a problem-solver, is called a *problem reformulation*. We used problem reformulations in the Three-Rocket and Tower-of-Hanoi example (Sections 2.1 and 2.2). We distinguish two types of problem reformulation, *enhancing* a problem description and *replacing* an old description with a new one.

- *Enhancing a problem description* is deducing some information that was implicit in the initial description of a problem and adding it to the description (Figure 5b). For example, we may add control rules, macro operators, or a library of problem-solving episodes for analogical reasoning. We view it as a problem reformulation because the use of the deduced information may completely change the behavior of a problem-solver.

- *Replacing a problem description* is adding information deduced from the initial description *and* discarding parts of the old information (Figure 5c). For example, we may remove unnecessary instances of operators from a domain description, as we did in the Three-Rocket example (Section 2.1).

We judge the quality of a new representation by the following three factors:

Efficiency: Problem solving with the new representation should be efficient for most of the frequently encountered problems. We measure efficiency by the average running time.

Near-Completeness: We must ensure that most solvable problems remain solvable in the new representation. We measure this factor by the percentage of solvable problems that become unsolvable after changing the representation.

Near-Optimality: We must ensure that the change of representation preserves optimal or near-optimal (satisficing) solutions to most problems and the problem-solving algorithm is able to find near-optimal solutions. We define the optimal solution as the solution with the lowest total cost of operators. We measure the optimality factor by the largest increase in the cost of solutions generated by the algorithm.

The design of representation-changing algorithms involves decisions on the trade-offs between these factors: we allow a "reasonable" loss of completeness and optimality in order to improve efficiency.

3.2 Example of a representation-changing algorithm

We describe a novel representation-changing algorithm, called *Operator-Remover*, which detects operators that can be removed from a problem domain without impairing our ability to solve problems. The algorithm removes these unnecessary operators, thus reducing the branching factor of search by a problem-solver and, therefore, improves the efficiency of the problem-solver.

The *Operator-Remover* algorithm is able to perform automatically the representation change in the Three-Rocket domain, described in Section 2.1: the algorithm determines that all transportation problems can be solved by sending `rocket-1` to `moon-1`, `rocket-2` to `moon-2`, and `rocket-3` to `moon-3`, and removes the operators for the other flights from the domain description. This representation change significantly reduces the complexity of search by the PRODIGY problem-solver.

In terms of Section 3.1, we classify *Operator-Remover* as an algorithm that *replaces a problem description* (see Figure 5c) by discarding unnecessary operators from the description.

The main problem in designing an algorithm for removing unnecessary operators is to guarantee *near-completeness* and *near-optimality* of problem solving. We must ensure that the reduced set of operators, generated by the algorithm, allows us to find near-optimal (satisficing) solutions to most problems in the domain.

An improper selection of operators may result in generating non-optimal solutions. For example, suppose that we try to solve the transportation problem shown in Figure 3(a) without the use of the airplane. The cost of the bus transportation of the two travelers to the town marked by "D" is 8, whereas the cost of the optimal plan for this problem is 4. In this example, the ratio of the cost of the solution based on the reduced set of operators and the cost of the optimal solution is $8/4 = 2$. This ratio is called the *cost increase* of problem solving with the reduced set of operators.

To address the problem of preserving near-optimal solutions, we use a learning algorithm that (1) tests whether the reduced set of operators preserves the completeness of problem solving and (2) estimates the maximal cost increase. The algorithm generates solutions to example problems based on the reduced set of operators and compares them with optimal solutions. If some problem cannot be solved with the reduced set of operators or the cost increase is larger than a certain constant C, the algorithm concludes that the reduced set of operators is not sufficient and adds more operators from the initial domain description to this set. Informally, this algorithm can be described as follows:

1. Generate an initial reduced set of operators.
2. Ask the user for the maximal allowed cost increase, C.
3. Repeat "several" times:
   (a) Generate an example problem.
   (b) Find an optimal solution, using all operators in the problem domain.
   (c) Try to find a solution with operators from the reduced set,
   such that the cost of this solution is at most $C \cdot Cost(Optimal\text{-}Solution)$.
   (d) If such a solution is not found,
   then add a new operator to the reduced set of operators.

To complete this algorithm, we must provide procedures for generating an initial reduced set of operators and for selecting a new operator when the current reduced set of operators is not sufficient for problem solving. We discussed some methods for accomplishing these tasks in [Fink and Yang, 1994a].

The input to *Operator-Remover* must include the set of operators in a problem domain. We may also provide the information about possible initial and goal states of problems, which will make the algorithm more effective. The algorithm uses the restrictions on the initial and goal states in generating example problems. For instance, *Operator-Remover* simplifies the description of the Three-Rocket domain only if we specify that the positions of rockets are never a part of the goal. (If we do not specify restrictions on goals, the algorithm assumes by default that all states can be goals.) This additional information about possible problems is called an *optional* input to the algorithm.

We next show the upper bound on the probability that the use of the reduced set of operators results in the loss of completeness or in finding too costly solutions. Suppose that we use the problem-solving algorithm with the reduced set of operators generated by *Operator-Remover*. Let δ denote the probability that, given a randomly selected solvable problem, the algorithm fails to find a solution the cost of which is at most C times larger than the cost of an optimal solution. In other words, δ is the probability that the problem-solver does not find a solution or finds a solution that is too costly. Let n denote the number of operators in the problem domain and m denote the number of example problems considered by the *Operator-Remover* algorithm. If $m > n^2$, then the "failure" probability δ is bounded by the following inequality:

$$\delta \leq n^2/m.$$

We present the derivation of this inequality in Appendix A.

This relationship between the number of examples and the failure probability is called the *sample complexity* of the learning algorithm. The inequality shows that the failure probability can be made arbitrarily small by increasing the number of training examples.

3.3 Specification of representation-changing algorithms

We have described the *Operator-Remover* algorithm, which removes operators from a domain description in order to reduce the branching factor of search while preserving completeness and near-optimality of problem solving. We say that removing operators is the *type* of the representation change performed by *Operator-Remover* and that reducing the branching factor while preserving completeness and near-optimality is the *purpose* of the representation change.

When implementing a representation-changing algorithm, we must decide on the *type and purpose* of the representation change performed by the algorithm. Restricting representation change to a specific type limits the space of alternative representations explored by the algorithm, whereas specifying the purpose shows exactly in which way the algorithm improves the representation. We must also determine which other algorithms the representation-changer calls as subroutines and identify the features of a problem domain that must be included into the representation-changer's input.

These top-level decisions form a *specification* of a representation-changing algorithm. Specifications allow us to (1) formally describe the desired properties of a representation-changing algorithm before implementing the algorithm and (2) separate these properties from specific methods used in the implementation. In Section 4, we will illustrate the use of specifications in the design of representation-changers.

When developing a specification of a representation-changing algorithm, we must make sure that the specification describes a useful representation change and that we can implement a simple and efficient algorithm that satisfies the specification. We compose specifications from the following five parts:

Type of representation change. When designing a representation-changer, we first select a type of representation change, such as removing operators (as in *Operator-Remover*) and replacing predicates with more specific ones (as in the Tower-of-Hanoi example in Section 2.2). We give some examples of types of representation changes in Figure 6.

Purpose of representation change. A representation-changing algorithm should find a representation that improves the *efficiency* of problem solving, while preserving *near-completeness* and *near-optimality* (Section 3.1). We quantitatively define these factors as follows:

- *Efficiency:* We use some easily-computable objective function to estimate the efficiency of problem solving with a new representation. For example, in the *Operator-Remover* algorithm, we use the number of operators in the reduced set as an objective function for estimating the problem-solving efficiency: the fewer operators, the more efficient problem solving.

Generating an abstraction hierarchy: Decomposing the set of predicates in a domain description into several subsets, according to the "importance" of predicates [Sacerdoti, 1974].

Replacing operators with macros: Replacing some operators in a domain description with macros constructed from these operators [Fikes *et al.*, 1972].

Removing operators: Deleting unnecessary operators from a domain description (see the Three-Rocket example in Section 2.1).

Generating more specific predicates: Replacing a predicate in a domain description with several more specific predicates, which together describe the same set of ground literals (see the Tower-of-Hanoi example in Section 2.2).

Selecting primary effects of operators: Choosing certain "important" effects of operators and then using operators only for achieving their important effects (see Section 4.2).

Figure 6: Examples of types of representation changes for improving the efficiency of problem solving.

Alternatively, we may estimate the efficiency of problem solving by measuring the running time of the problem-solver on several example problems. This method is more accurate, but it is much less efficient.

- *Near-Completeness:* We measure the completeness loss by the percentage of solvable problems that become unsolvable after changing the representation. In *Operator-Remover*, the percentage of problems that may become unsolvable is bounded by n^2/m, where n is the number of operators in the domain and m is the number of example problems considered by the algorithm (see Section 3.2).

- *Near-Optimality:* We measure the optimality loss by the increase in the cost of solutions after changing the representation. In *Operator-Remover*, this cost increase is bounded by the user-specified maximal cost increase, C.

To summarize, we view the purpose of a representation change as maximizing a specific objective function, while preserving near-completeness and near-optimality, which is the central idea of the approach. This formal specification of the purpose shows exactly in which way we improve the representation and enables us to evaluate the results of a representation change.

The use of other algorithms. We specify here the problem-solving and learning algorithms that the representation-changer uses as subroutines and the purpose of using these algorithms. For example, *Operator-Remover* calls the PRODIGY problem-solver to solve the example problems used in learning.

Required input. We list here the parts of the domain description that we must include into the representation-changer's input. For example, the required input to the *Operator-Remover* algorithm includes the description of operators in the problem domain.

Type of representation change: Removing operators.

Purpose of representation change: Minimizing the number of remaining operators, while ensuring that problem solving is complete and the cost increase is within the user-specified bound C.

The use of other algorithms: A problem-solver, used in checking completeness and estimating the cost increase of problem solving with the reduced set of operators.

Required input: Description of the operators; maximal cost increase.

Optional input: Restrictions on the possible initial and goal states.

Figure 7: Specification of the *Operator-Remover* representation-changing algorithm.

Optional input. Finally, we specify the additional information about the problem domain that may be used by the representation-changer. If the user inputs this information, the representation-changer uses it to generate a better representation; in the absence of such information, the algorithm uses some default values. The optional input may include useful knowledge about the properties of the problem domain, such as control rules and a list of primary effects of operators, and restrictions on the types of problems that we will solve with a new representation. For example, we may specify restrictions on the initial and goal states of problems as an optional input to *Operator-Remover*; if we do not provide this information, the algorithm assumes by default that problems may have any initial and goal states.

We summarize the specification of the *Operator-Remover* algorithms in Figure 7.

4 Examples of designing representation-changers

We have described a method for specifying the important properties of representation-changing algorithms, which enables us to abstract these important properties from implementational details. We now illustrate the application of this method to the design of three representation-changers, *Instantiator*, *Prim-Tweak*, and *Margie*.

Instantiator is a simple search algorithm (Section 4.1), which improves the effectiveness of the ALPINE abstraction-generator by replacing some predicates in the domain description with more specific predicates. *Instantiator* is able to perform the representation change in the Tower-of-Hanoi domain, described in Section 2.2.

Prim-Tweak and *Margie* are more complex algorithms, which improve problem representations by selecting primary effects of operators. We have implemented these algorithms in collaboration with Yang [Fink and Yang, 1992; Fink and Yang, 1993]. *Prim-Tweak* reduces the branching factor of search (Section 4.2), whereas *Margie* improves the effectiveness of the ALPINE abstraction-generator (Section 4.3)[1].

[1]The *Margie* representation-changer (pronounced *mär'gē*) is not related to the Margie (*mär'jē*) system for parsing and paraphrasing simple stories in English, implemented by Schank, Goldman, Rieger, and Riesbeck.

4.1 Instantiator: Increasing the number of abstraction levels

We design a novel representation-changer that improves the quality of ALPINE's abstraction hierarchies.

The ALPINE abstraction-generator is a very fast, polynomial-time algorithm that generates an abstraction hierarchy of predicates [Knoblock, 1994]. The importance of every predicate in a problem domain is represented by a natural number: the larger the number, the more important the predicate. Hierarchies generated by ALPINE are based on the following two constraints, which usually lead to a high efficiency of abstraction problem solving [Knoblock *et al.*, 1991]:

For every operator Op in a problem domain,
(1) if eff_1 and eff_2 are effects of Op,
then $Importance(\mathit{eff}_1) = Importance(\mathit{eff}_2)$;
(2) if eff is an effect of Op and $prec$ is a precondition of Op,
then $Importance(\mathit{eff}) \geq Importance(prec)$.

For example, consider the three-disk Tower-of-Hanoi domain represented by three predicates, `(on small <peg>)`, `(on medium <peg>)`, and `(on large <peg>)`, as shown in Figure 2(c). In Figure 2(d), we show an abstraction hierarchy of predicates in this domain that satisfies ALPINE's constraints.

If the user describes a problem domain by a small number of general predicates, the ALPINE algorithm usually fails to generate an abstraction hierarchy, since the algorithm cannot distribute special cases of a general predicate between several levels of abstraction. We encountered this problem in the Tower-of-Hanoi domain described by a single predicate, `(on <disk> <peg>)` (see Figure 2b). ALPINE cannot distribute this single predicate between several levels of abstraction and, thus, it cannot generate a multi-level abstraction hierarchy.

We may avoid the problem of too general predicates by instantiating all variables in the domain description, thus replacing each general predicate with a large number of fully instantiated predicates and each operator with a number of fully instantiated operators. In complex domains, however, the full instantiation leads to a combinatorial explosion in the number of instantiated predicates and operators.

Knoblock proposed a more sophisticated technique for replacing general predicates with specific ones, in which the user divides the values of variables into classes [Knoblock, 1991a]. For example, in the Tower-of-Hanoi domain, the user may indicate that the disks are divided into three classes, `Small`, `Medium`, and `Large`, whereas all pegs are of the same class. ALPINE uses this information to generate three specific predicates, `on-small`, `on-medium`, and `on-large`, and then generates more specific operators by incorporating these more specific predicates into the operator description. This approach, however, requires the user's help. Besides, a large number of classes in a complex domain may still lead to a combinatorial explosion.

To avoid the explosion, the algorithm must automatically pinpoint the variables whose instantiation leads to increasing the number of abstraction levels, and instantiate only these variables. For example, in the Tower-of-Hanoi domain, the algorithm must instantiate `<disk>` and leave `<peg>` uninstantiated.

Type of representation change: Generating more specific predicates.

Purpose of representation change: Maximizing the number of levels in the abstraction hierarchy generated by ALPINE.

The use of other algorithms: ALPINE, used in generating an abstraction hierarchy based on more specific predicates.

Required input: Description of the operators; the possible values of the variables in the domain description.

Optional input: Predicates describing unchangeable relations between the values of variables.

Instantiator algorithm:
For every variable v in the domain description:
1. For every operator Op that contains v,
   instantiate v in the description of Op with all its possible values,
   thus replacing Op with a set of more specific operators.
2. Call ALPINE to generate abstraction for the resulting domain description.
3. If the number of levels in this new abstraction hierarchy is larger
   than that in the hierarchy before instantiating v,
   then leave v instantiated in the domain description,
   else go back to the domain description with uninstantiated v.

Figure 8: Increasing the number of levels in an abstraction hierarchy by generating more specific predicates: a specification of a representation-changer and a greedy algorithm that satisfies the specification.

To perform the instantiation, the algorithm must know all possible values of the variables. For example, it must know that the possible values of `<disk>` are `small`, `medium`, and `large`. The algorithm should also use predicates that describe unchangeable relations between the values of variables, such as `(smaller small medium)`, which constrain possible instantiations of variables in the description of operators. Using these constraints, the algorithm generates instantiated operators, which enables ALPINE to use less restrictions in generating an abstraction hierarchy and, thus, produce a hierarchy with more levels.

We can now express the requirements to an algorithm for generating more specific predicates as a specification of a representation-changer; we show the specification at the top of Figure 8. We do not include the preservation of completeness and optimality into the purpose of representation change, because the generation of more specific predicates cannot compromise completeness or optimality of problem solving.

We next design a greedy algorithm, called *Instantiator*, that satisfies this specification. An informal description of the *Instantiator* algorithm is shown at the bottom of Figure 8. The algorithm tries to instantiate different variables, one by one, and calls ALPINE to determine whether instantiating a variable leads to an increase in the number of abstraction levels. If the instantiation does not increase the number of levels, the algorithm leaves the variable uninstantiated and tries to instantiate another variable.

For example, suppose that we apply the *Instantiator* algorithm to the Tower-of-Hanoi domain described with a single predicate, (on <disk> <peg>). The algorithm will notice that instantiating the variable <disk> results in generating a three-level abstraction hierarchy, whereas instantiating the variable <peg> does not increase the number of abstraction levels. The algorithm will thus instantiate <disk> and generate the three-level hierarchy shown in Figure 2(d).

Note that the *Instantiator* algorithm does not try to instantiate several variables at once and, therefore, it fails to find a good instantiation if the only way to increase the number of abstraction levels is to instantiate several variables together. We may avoid this problem by modifying the algorithm in such a way that it considers instantiations of couples and triples of variables when instantiating single variables does not improve the abstraction hierarchy.

4.2 Prim-Tweak: Selecting primary effects of operators

We now show the use of specifications in designing a learning algorithm that reduces the complexity of search by selecting *primary* effects of operators.

The idea of problem solving with primary effects is to select "important" effects among the effects of each operator. A problem-solver inserts new operators into a plan only when it needs to achieve one of the primary effects of the operator. The use of primary effects reduces the branching factor of search and may result in an exponential reduction of running time.

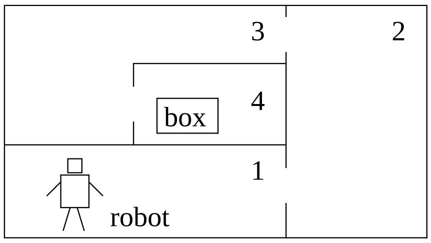
For example, consider a planning domain with a robot, four rooms, and a box, shown in Figure 9. The robot may go between two rooms connected by a door and it may carry the box. If two rooms are separated by a wall, the robot may break through the wall to create a new door. We may select the following primary effects in this domain:

Operators	**Primary Effects**
(go <from-room> <to-room>)	(robot-in <to-room>)
(carry-box <from-room> <to-room>)	(box-in <to-room>)
(break <from-room> <to-room>)	(door <from-room> <to-room>)

Suppose that the initial state is as shown in Figure 9 and the robot must go to room-3. The robot may achieve this goal by breaking through the wall between room-1 and room-3; however, since changing the location of the robot is *not* a primary effect of the break operator, a problem-solver will not consider this possibility. Instead, it will find the plan "(go room-1 room-2), (go room-2 room-3)."

The number of primary effects determines the branching factor of search: the fewer primary effects, the smaller the branching factor. Therefore, the search complexity decreases with decreasing the number of primary effects.

On the other hand, selecting primary effects involves the same completeness and optimality problem as removing operators (Section 3.2): we must ensure that the use of primary effects does not prevent a problem-solver from finding solutions to solvable problems and that the cost increase is never larger than a user-specified value C. We may verify the completeness of planning with selected primary effects and estimate the cost increase by testing a problem-solving algorithm on a set of example problems, using the same technique as in the *Operator-Remover* algorithm (Section 3.2).




```
Operator GO                         ||  Operator CARRY-BOX                 ||  Operator BREAK
Parameters:                         ||  Parameters:                        ||  Parameters:
  <from-room> <to-room>             ||    <from-room> <to-room>            ||    <from-room> <to-room>
Preconds:                           ||  Preconds:                          ||  Preconds:
  (robot-in <from-room>)            ||    (box-in <from-room>)             ||    (robot-in <from-room>)
  (door <from-room> <to-room>)      ||    (robot-in <from-room>)           ||    (adjacent <from-room> <to-room>)
Effects:                            ||    (door <from-room> <to-room>)     ||  Effects:
  Del: (robot-in <from-room>)       ||  Effects:                           ||    Del: (robot-in <from-room>)
  Add: (robot-in <to-room>)         ||    Del: (box-in <from-room>)        ||    Add: (robot-in <to-room>)
Cost: 2                             ||         (robot-in <from-room>)      ||         (door <from-room> <to-room>)
                                    ||    Add: (box-in <to-room>)          ||  Cost: 4
                                    ||         (robot-in <to-room>)        ||
                                    ||  Cost: 3                            ||
```

Figure 9: Operators of the robot domain: the robot can go between two rooms connected by a door, carry the box, and break through a wall to create a new door.

To summarize, we are interested in minimizing the number of primary effects of operators, while ensuring that problem solving with primary effects is complete and the cost increase is no larger than the user-specified bound. We express these requirements as a specification of a representation-changing algorithm, shown at the top of Figure 10.

We next design an algorithm, called *Prim-Tweak*, that satisfies the specification. We use the same learning method as in *Operator-Remover*. An informal description of the *Prim-Tweak* learner is shown at the bottom of Figure 10. The learner takes an initial selection of primary effects, specified by the user or generated by a simple heuristical algorithm [Fink and Yang, 1993], and selects more primary effects to make sure that problem solving is complete and the cost increase is at most C. This goal is achieved by ensuring that, for every operator Op, the problem-solver can always find a plan, with the cost at most $C \cdot cost(Op)$, that achieves all side effects of Op: the learner generates a number of states that satisfy the preconditions of Op and checks whether the side effects of Op can be achieved in all these states.

For example, suppose that we apply this learning algorithm to the robot domain, and the initial selection of primary effects is as shown above. The learner is likely to notice that the effect `(robot-in room-4)` of the operator `(break room-1 room-4)` cannot be efficiently achieved when using primary effects: a problem-solver that uses primary effects will generate the plan "`(go room-1 room-2), (go room-2 room-3), (go room-3 room-4)`," the cost of which is 6, whereas the cost of the `break` operator is only 4; thus, the cost increase for this problem is $6/4 = 1.5$. If the user-specified bound C on the value of a cost increase is less than 1.5, the algorithm will make the `robot-in` effect a primary effect of the `break` operator.

In [Fink and Yang, 1994a] we presented a formal description of *Prim-Tweak*, described

Specification:

Type of representation change: Selecting primary effects of operators.

Purpose of representation change: Minimizing the number of primary effects, while ensuring that problem solving is complete and the cost increase is within the user-specified bound C.

The use of other algorithms: A problem-solver, used in checking completeness and estimating the cost increase of problem solving with primary effects.

Required input: Description of the operators; maximal cost increase.

Optional input: Restrictions on the possible states of the domain; primary effects selected by the user.

Prim-Tweak algorithm:

1. Generate an initial selection of primary effects.
2. Ask the user for the maximal allowed cost increase, C.
3. For every operator Op in the domain, repeat "several" times:
   (a) Pick at random a state S that satisfies the preconditions of Op.
   (b) Run a problem-solver using primary effects selected so far to find a plan $\mathcal{P}$ such that
      ◦ $\mathcal{P}$ achieves the side effects of Op from the initial state S
      ◦ and the cost of $\mathcal{P}$ is at most $C \cdot cost(Op)$.
   (c) If such a plan is not found, then make one of the side effects of Op be a primary effect.

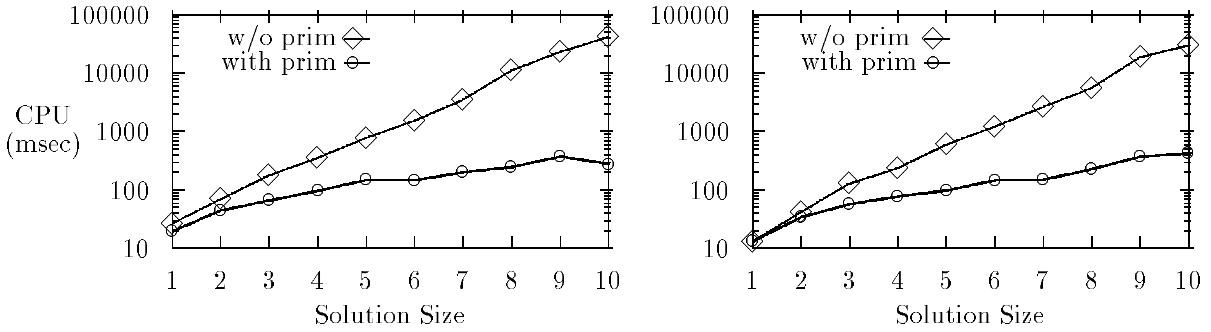
Figure 10: Reducing the branching factor of search by selecting primary effects: a specification of a representation-changer and a learning algorithm that satisfies the specification.

methods for generating random legal states satisfying the preconditions of a given operator, and estimated the number of different states that must be considered for every operator to ensure a high probability of completeness and optimality. We demonstrated that the sample complexity of this learning algorithm is linear; that is, the probability of losing completeness decreases linearly with an increase in the number of states considered for each operator.

We have demonstrated experimentally that *Prim-Tweak* considerably improves the efficiency of the ABTWEAK planning system in many different domains [Fink and Yang, 1994a]. In Figure 11, we present the results of testing *Prim-Tweak* in two artificial domains. The graphs show the running time of the ABTWEAK planner without primary effects ("w/o prim") and with the use of primary effects selected by *Prim-Tweak* ("with prim") for problems with different optimal-solution sizes. (Note that the time scale is logarithmic—the use of primary effects improves efficiency by two orders of magnitude.)

4.3 Margie: Primary effects in learning abstraction hierarchies

We next apply our approach to developing a representation-changer that improves the effectiveness of the ALPINE abstraction-generator [Knoblock, 1994] by selecting primary effects of operators. The ALPINE algorithm can use the knowledge of primary effects in generating an abstraction hierarchy, and we may improve the quality of the hierarchy by finding the



All operators have the same cost.

Operators have different costs.

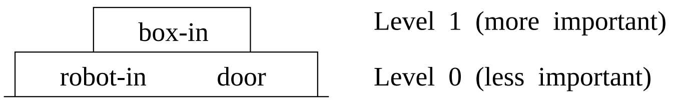
Figure 11: Experiments on problem solving without primary effects ("w/o prim") and with the primary effects selected by the *Prim-Tweak* representation-changer ("with prim"): the use of primary effects improves efficiency by two orders of magnitude.

"right" primary effects.

If ALPINE has the information about the primary effects of operators, it generates an abstraction hierarchy based on the following constraints, which are less restrictive then the constraints shown in Section 4.1:

> For every operator *Op* in a problem domain,
> (1a) if $prim_1$ and $prim_2$ are primary effects of *Op*,
> then $Importance(prim_1) = Importance(prim_2)$;
> (1b) if *prim* is a primary effect of *Op* and *side* is a side effect of *Op*,
> then $Importance(prim) \geq Importance(side)$;
> (2) if *prim* is a primary effect of *Op* and *prec* is a precondition of *Op*,
> then $Importance(prim) \geq Importance(prec)$.

For example, consider the robot domain with the primary effects selected at the beginning of Section 4.2. The following hierarchy of predicates in this domain satisfies ALPINE's constraints:



On the other hand, if we do not specify primary effects in the robot domain, the ALPINE algorithm uses the constraints of Section 4.1 and fails to generate an abstraction hierarchy: when the algorithm imposes its constraints on the importance levels of predicates, all predicates collapse into a single-level hierarchy.

We are interested in selecting primary effects of operators that maximize the number of levels in the hierarchy generated by ALPINE, while ensuring that the use of primary effects does not violate completeness and the cost increase is within a user-specified bound C. We express these requirements as a specification of a representation-changing algorithm, shown at the top of Figure 12.

Specification:

Type of representation change: Selecting primary effects of operators.

Purpose of representation change: Maximizing the number of levels in the abstraction hierarchy generated by ALPINE, while ensuring that problem solving is complete and the cost increase is within the user-specified bound C.

The use of other algorithms: ALPINE, used in generating abstraction hierarchies based on the selected primary effects; *Prim-Tweak*, used in selecting additional primary effects to ensure that problem solving is complete and the cost increase is within the user-specified bound.

Required input: Description of the operators; maximal cost increase.

Optional input: Restrictions on the possible states of the domain.

Margie algorithm:

1. Mark all effects of all operators as side effects.
2. For every predicate p in the problem domain,
   if there are operators that achieve p,
   select one of these operators and make p its primary effect;
      select this operator in such a way as to maximize the number of levels
      in the hierarchy generated by ALPINE for the current primary effects.
3. For every operator Op that does not have primary effects yet,
   make one of the effects of Op primary;
      select this effect in such a way as to maximize the number of levels
      in the hierarchy generated by ALPINE for the current primary effects.
4. Call *Prim-Tweak* to select additional primary effects
   in order to ensure that the cost increase is at most C.

Figure 12: Increasing the number of levels in an abstraction hierarchy by selecting primary effects of operators: a specification of a representation-changer and a greedy algorithm that satisfies the specification.

We have designed a greedy algorithm, called *Margie*, that satisfies this specification [Fink and Yang, 1992; Fink and Yang, 1994b]. We give an informal description of this algorithm at the bottom of Figure 12. The *Margie* algorithm considers different selections of primary effects and calls ALPINE to generate abstraction hierarchies based on these selections. *Margie* determines the selection that results in the largest number of abstraction levels and then calls the *Prim-Tweak* algorithm to select additional primary effects in order to ensure that the cost increase is at most C.

5 Conclusions and open problems

We have described a systematic approach to the design and analysis of representation-changing algorithms, based on the formal specification of the important properties of these algorithms. We demonstrated that the use of specifications simplifies the task of designing representation-changing algorithms and evaluating their performance. When developing a representation-changer, we first formalize its desirable properties and then implement a learning or search algorithm with these properties.

We now discuss some other research problems on automatic representation changes, which we plan to address. We follow Simon's view of representations as "data structures and programs operating on them" ([Larkin and Simon, 1987], page 67). In an AI system, the programs are problem-solving algorithms; the data structures are the inputs of these algorithms, which may include operators, control rules, macros, libraries of past problem-solving episodes, etc.

A representation-changing system should perform two tasks:

(1) select a problem-solving algorithm (from a certain library of problem-solvers)

(2) and change the domain description to improve the performance of this algorithm.

We therefore plan to build a system that consists of three main parts: (1) a library of problem-solving algorithms; (2) a library of representation-changing algorithms; and (3) a top-level decision procedure that selects algorithms appropriate for a given problem. The top-level procedure will first choose a problem-solving algorithm and then maximize the efficiency of this algorithm by applying appropriate representation-changers to the initial description of the problem.

We have so far worked on the development of a library of representation-changers. Our research goals include the development of a formal model of specifications, methods for designing specifications of useful representation changes, and techniques for implementing representation-changing algorithms according to specifications. We plan to apply these techniques to designing algorithms capable of performing complex representation changes, such as the representation change in the Interurban-Transportation example (Section 2.3).

Acknowledgements

I gratefully acknowledge the valuable help of Herbert Simon, Jaime Carbonell, and Manuela Veloso, who guided me in my research and writing. I am grateful to Nevin Heintze, who encouraged me to write this paper. I also owe thanks to Henry Rowley, Yury Smirnov, and Alicia Pérez for their comments and suggestions.

The *Prim-Tweak* and *Margie* algorithms (Sections 4.2 and 4.3) were developed in collaboration with Qiang Yang. The introductory illustration is by Evgenia Nayberg.

A Sample complexity of Operator-Remover

We analyze the probability that the reduced set of operators selected by the *Operator-Remover* algorithm is *not* sufficient for problem solving. We denote the number of operators in the problem domain by n and the number of example problems available to *Operator-Remover* by m. We draw these example problems at random, using some probability distribution μ of solvable problems in the domain.

Suppose that *Operator-Remover* has generated a reduced set of operators for some user-specified cost increase C. Suppose further that we have randomly selected a solvable problem, using the distribution μ, and we try to solve this problem with the reduced set of operators. We define δ as the probability that, given a randomly selected solvable problem, we *cannot* find a solution the cost of which is at most C time larger than the cost of an optimal solution. We call δ the *failure probability* of problem solving with the reduced set of operators.

We are interested in finding an estimate of δ in terms of n and m; this estimate is called the *sample complexity* of learning. We will show that our estimate does not depend on the distribution μ, which means that the learning algorithm is *distribution-free*.

To simplify the derivation, we consider a modified version of the *Operator-Remover* algorithm, shown in Figure 13. This new version differs from the original algorithm by the italicized line: the new algorithm terminates when it successfully solves $\frac{m}{n}$ consecutive example problems without adding new operators to the reduced set. Since the algorithm can add at most n operators to the initial reduced set, it considers at most m problems.

We define δ' as the failure probability of problem solving with the operators selected by the modified algorithm. Since the modified algorithm considers at most as many examples as the original *Operator-Remover* and the failure probability monotonically decreases with the number of considered examples, we conclude that $\delta' \geq \delta$.

We now derive an upper bound for δ', in terms of n and m. We begin by dividing the possible outcomes of learning into the following n cases:

Case$_1$: the reduced set contains *one* operator;
Case$_2$: the reduced set contains *two* operators;
...
Case$_n$: the reduced set contains *all* n operators.

We denote the probability of Case$_i$ by Prob(Case$_i$) and the failure probability in Case$_i$ by δ_i. We can now express the overall failure probability δ' as follows:

$$\delta' = \sum_{i=1}^{n} \text{Prob}(\text{Case}_i) \cdot \delta_i.$$

The algorithm terminates only when it solves $\frac{m}{n}$ consecutive problems without adding new operators. If the reduced set already contains i operators, then the probability that the algorithm terminates without adding more operators is $(1-\delta_i)^{\frac{m}{n}}$. Therefore, the probability that the algorithm terminates with exactly i operators in the reduced set is *at most* $(1-\delta_i)^{\frac{m}{n}}$; that is,

$$\text{Prob}(\text{Case}_i) \leq (1-\delta_i)^{\frac{m}{n}}.$$

1. Generate an initial reduced set of operators.
2. Ask the user for the maximal allowed cost increase, C.
3. `Repeat`:
   (a) Generate an example problem.
   (b) Find an optimal solution, using all operators in the problem domain.
   (c) Try to find a solution with operators from the reduced set, such that the cost of this solution is at most $C \cdot$ *Cost*(*Optimal-Solution*).
   (d) If such a solution is not found, then add a new operator to the reduced set of operators.

   `Until` *the algorithm solves* $\frac{m}{n}$ *consecutive example problems without adding new operators to the reduced set of operators.*

Figure 13: Modified version of the *Operator-Remover* algorithm, used in the derivation of the sample complexity of learning.

Substituting this bound into the expression for δ', we get the following inequality:

$$\delta' \leq \sum_{i=1}^{n} (1-\delta_i)^{\frac{m}{n}} \cdot \delta_i.$$

To find an upper bound for the sum on the right-hand side of the inequality, we observe that, for $0 \leq x \leq 1$, the function $(1-x)^{\frac{m}{n}} \cdot x$ reaches its maximum at the point $x = \frac{1}{\frac{m}{n}+1}$, which can be verified by taking the derivative of this function and finding the points where the derivative is 0. Therefore, for $0 \leq x \leq 1$, we have:

$$(1-x)^{\frac{m}{n}} \cdot x \leq (1 - \frac{1}{\frac{m}{n}+1})^{\frac{m}{n}} \cdot \frac{1}{\frac{m}{n}+1} \leq \frac{1}{\frac{m}{n}+1}.$$

We next observe that, for every i, the value of δ_i is between 0 and 1, and, therefore, we have $(1-\delta_i)^{\frac{m}{n}} \cdot \delta_i \leq \frac{1}{\frac{m}{n}+1}$. Substituting this bound into the inequality for δ', we get

$$\delta' \leq \sum_{i=1}^{n} \frac{1}{\frac{m}{n}+1} = \frac{n}{\frac{m}{n}+1} = \frac{n^2}{m+n} \leq \frac{n^2}{m}.$$

Since the failure probability δ of the original *Operator-Remover* algorithm is no larger than δ', we may use the same expression as an upper bound for δ:

$$\delta \leq \frac{n^2}{m}.$$

References

[Allen *et al.*, 1992] John Allen, Pat Langley, and Stan Matwin. Knowledge and regularity in planning. In *Proceedings of the AAAI 1992 Spring Symposium on Computational Considerations in Supporting Incremental Modification and Reuse*, pages 7–12, 1992.

[Amarel, 1968] Saul Amarel. On representations of problems of reasoning about actions. In Donald Michie, editor, *Machine Intelligence 3*, pages 131–171. American Elsevier Publishers, 1968.

[Bacchus and Yang, 1992] Fahiem Bacchus and Qiang Yang. The expected value of hierarchical problem-solving. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, 1992.

[Bacchus and Yang, 1994] Fahiem Bacchus and Qiang Yang. Downward refinement and the efficiency of hierarchical problem solving. *Artificial Intelligence*, 71(1):43–100, 1994.

[Carbonell, 1983] Jaime G. Carbonell. Learning by analogy: Formulating and generalizing plans from past experience. In Ryszard S. Michalski, Jaime G. Carbonell, and Tom M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*. Tioga Publishers, Palo Alto, CA, 1983.

[Carbonell, 1990] Jaime G. Carbonell, editor. *Machine Learning: Paradigms and Methods*. MIT Press, Boston, MA, 1990.

[Christensen, 1990] Jens Christensen. A hierarchical planner that generates its own abstraction hierarchies. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 1004–1009, 1990.

[Cohen, 1992] William W. Cohen. Using distribution-free learning theory to analyze solution-path caching mechanisms. *Computational Intelligence*, 8(2):336–375, 1992.

[Etzioni, 1993] Oren Etzioni. Acquiring search control knowledge via static analysis. *Artificial Intelligence*, 62(2):255–301, 1993.

[Fikes *et al.*, 1972] Richard E. Fikes, P. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3(4), 1972.

[Fink and Yang, 1992] Eugene Fink and Qiang Yang. Automatically abstracting effects of operators. In *Proceedings of the First International Conference on AI Planning Systems*, pages 243–251, 1992.

[Fink and Yang, 1993] Eugene Fink and Qiang Yang. Characterizing and automatically finding primary effects in planning. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 1374–1379, 1993.

[Fink and Yang, 1994a] Eugene Fink and Qiang Yang. Automatically selecting and using primary effects in planning: Theory and experiments. Technical Report CMU-CS-94-206, School of Computer Science, Carnegie Mellon University, 1994.

[Fink and Yang, 1994b] Eugene Fink and Qiang Yang. Search reduction in planning with primary effects. In *Proceedings of the Workshop on Theory Reformulation and Abstraction*, pages 39–55, 1994.

[Gentner and Stevens, 1983] Dedre Gentner and Albert L. Stevens, editors. *Mental Models*, Hillside, NJ, 1983. Lawrence Erlbaum Associates.

[Giunchiglia and Walsh, 1992] Fausto Giunchiglia and Toby Walsh. A theory of abstraction. *Artificial Intelligence*, 57:323–389, 1992.

[Hall, 1987] Rogers P. Hall. Understanding analogical reasoning: Computational approaches. *Artificial Intelligence*, 39:39–120, 1987.

[Hayes and Simon, 1974] John R. Hayes and Herbert A. Simon. Understanding written problem instructions. In L. W. Gregg, editor, *Knowledge and Cognition*, pages 167–200. Lawrence Erlbaum Associates, Potomac, MD, 1974.

[Hayes and Simon, 1976] John R. Hayes and Herbert A. Simon. The understanding process: Problem isomorphs. *Cognitive Psychology*, 8:165–190, 1976.

[Hayes and Simon, 1977] John R. Hayes and Herbert A. Simon. Psychological difference among problem isomorphs. In N. J. Castellan, D. B. Pisoni, and G. R. Potts, editors, *Cognitive Theory*. Lawrence Erlbaum Associates, Hillside, NJ, 1977.

[Hibler, 1994] David Hibler. Implicit abstraction by thought experiments. In *Proceedings of the Workshop on Theory Reformulation and Abstraction*, pages 9–26, 1994.

[Kaplan and Simon, 1990] Craig A. Kaplan and Herbert A. Simon. In search of insight. *Cognitive Psychology*, 22:374–419, 1990.

[Kaplan, 1989] Craig A. Kaplan. SWITCH: A simulation of representational change in the Mutilated Checkerboard problem. Technical Report C.I.P. 477, Department of Psychology, Carnegie Mellon University, 1989.

[Knoblock *et al.*, 1991] Craig A. Knoblock, Josh Tenenberg, and Qiang Yang. Characterizing abstraction hierarchies for planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 692–697, 1991.

[Knoblock, 1990] Craig A. Knoblock. Learning abstraction hierarchies for problem solving. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 923–928, 1990.

[Knoblock, 1991a] Craig A. Knoblock. *Automatically Generating Abstractions for Problem Solving.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1991. Technical Report CMU-CS-91-120.

[Knoblock, 1991b] Craig A. Knoblock. Search reduction in hierarchical problem solving. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 686–691, 1991.

[Knoblock, 1994] Craig A. Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68, 1994.

[Korf, 1980] Richard E. Korf. Toward a model of representation changes. *Artificial Intelligence*, 14:41–78, 1980.

[Korf, 1985] Richard E. Korf. Macro-operators: A week method for learning. *Artificial Intelligence*, 25:35–77, 1985.

[Korf, 1987] Richard E. Korf. Planning as search: A quantitative approach. *Artificial Intelligence*, 33:65–88, 1987.

[Laird *et al.*, 1986] John E. Laird, Paul S. Rosenbloom, and Allen Newell. Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning*, 1:11–46, 1986.

[Laird *et al.*, 1987] John E. Laird, Allen Newell, and Paul S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33:1–64, 1987.

[Langley, 1983] Pat Langley. Learning effective search heuristics. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pages 419–421, 1983.

[Larkin and Simon, 1981] Jill Larkin and Herbert A. Simon. Learning through growth of skill in mental modeling. In *Proceedings of the Third Annual Conference of the Cognitive Science Society*, 1981.

[Larkin and Simon, 1987] Jill H. Larkin and Herbert A. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11:65–99, 1987.

[Larkin *et al.*, 1988] Jill H. Larkin, Frederick Reif, Jaime G. Carbonell, and Angela Gugliotta. FERMI: A flexible expert reasoner with multi-domain inferencing. *Cognitive Psychology*, 12:101–138, 1988.

[Minton, 1988] Steven Minton. *Learning Effective Search Control Knowledge: An Explanation-Based Approach.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1988. Technical Report CMU-CS-88-133.

[Mooney, 1988] Raymond J. Mooney. Generalizing the order of operators in macro-operators. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 270–283, San Mateo, CA, 1988. Morgan Kaufmann.

[Newell and Simon, 1972] Allen Newell and Herbert A. Simon. *Human Problem Solving.* Prentice Hall, Englewood Cliffs, NJ, 1972.

[Newell *et al.*, 1960] Allen Newell, J. C. Shaw, and Herbert A. Simon. A variety of intelligent learning in a general problem solver. In Marshall C. Yovits, editor, *International Tracts in Computer Science and Technology and Their Applications*, volume 2: Self-Organizing Systems, pages 153–189. Pergamon Press, New York, NY, 1960.

[Newell, 1965] Allen Newell. Limitations of the current stock of ideas about problem solving. In A. Kent and O. Tualbee, editors, *Electronic Information Handling*. Spartan Books, Washington, DC, 1965.

[Newell, 1966] Allen Newell. On the representations of problems. In *Computer Science Research Reviews*. Carnegie Institute of Technology, Pittsburgh, PA, 1966.

[Newell, 1992] Allen Newell. Unified theories of cognition and the role of Soar. In J. A. Michon and A. Akyürek, editors, *Soar: A Cognitive Architecture in Perspective*, pages 25–79. Kluwer Academic Publishers, Netherlands, 1992.

[Pérez and Etzioni, 1992] M. Alicia Pérez and Oren Etzioni. DYNAMIC: A new role for training problems in EBL. In D. Sleeman and P. Edwards, editors, *Proceedings of the Ninth International Conference on Machine Learning*, San Mateo, CA, 1992. Morgan Kaufmann.

[Polya, 1957] George Polya. *How to Solve It.* Doubleday, Garden City, NY, second edition, 1957.

[Qin and Simon, 1992] Yulin Qin and Herbert A. Simon. Imagery and mental models in problem solving. In N. Hari Narayanan, editor, *Proceedings of the AAAI 1992 Spring Symposium on Reasoning with Diagrammatic Representations*, Palo Alto, CA, 1992. Stanford University.

[Sacerdoti, 1974] Earl D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135, 1974.

[Shell and Carbonell, 1989] Peter Shell and Jaime G. Carbonell. Towards a general framework for composing disjunctive and iterative macro-operators. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 1989.

[Simon *et al.*, 1985] Herbert A. Simon, K. Kotovsky, and J. R. Hayes. Why are some problems hard? Evidence from the Tower of Hanoi. *Cognitive Psychology*, 17:248–294, 1985.

[Simon, 1975] Herbert A. Simon. The functional equivalence of problem solving skills. *Cognitive Psychology*, 7:268–288, 1975.

[Stone and Veloso, 1994] Peter Stone and Manuela M. Veloso. Learning to solve complex planning problems: Finding useful auxiliary problems. In *Proceedings of the AAAI 1994 Fall Symposium on Planning and Learning*, pages 137–141, 1994.

[Stone *et al.*, 1994] Peter Stone, Manuela M. Veloso, and Jim Blythe. The need for different domain-independent heuristics. In *Proceedings of the Second International Conference on AI Planning Systems*, pages 164–169, 1994.

[Tabachneck, 1992] Hermina J. M. Tabachneck. *Computational Differences in Mental Representations: Effects of Mode of Data Presentation on Reasoning and Understanding.* PhD thesis, Department of Psychology, Carnegie Mellon University, 1992.

[Tamble *et al.*, 1990] Milind Tamble, Allen Newell, and Paul S. Rosenbloom. The problem of expensive chunks and its solution by restricting expressiveness. *Machine Learning*, 5:299–348, 1990.

[Veloso and Borrajo, 1994] Manuela M. Veloso and Daniel Borrajo. Learning strategy knowledge incrementally. In *Proceedings of the Sixth International Conference on Tools with Artificial Intelligence*, pages 484–490, New Orleans, LA, 1994.

[Veloso, 1994] Manuela M. Veloso. *Planning and Learning by Analogical Reasoning.* Springer Verlag, 1994.