19)

FAILURE-DIRECTED REFORMULATION

by

PUSHPINDER SINGH

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREES OF

BACHELOR OF SCIENCE IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE AND MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

March 10, 1998

Signature of Author [illegible]
Department of Electrical Engineering and Computer Science
March 8, 1998

Certified by [illegible]
Marvin Lee Minsky
Toshiba Professor of Media Arts and Sciences
Thesis Supervisor

Accepted by [illegible]
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Eng.

Failure-Directed Reformulation

by
Pushpinder Singh

Submitted to the
Department of Electrical Engineering and Computer Science

March 10, 1998

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Electrical Engineering and Computer Science
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

In this thesis we explore the domain of ordinary, common-sense problem solving. We study the heuristic method of *reformulation*, that is, of changing the representation of the problem. We are interested in reformulations that cause a substantial change in viewpoint, one that is likely to be different enough from the original viewpoint that we have a good chance of getting unstuck. We present a simple way of organizing representations so that reformulation can occur quickly if a dead-end is encountered while applying current method of solution. When an impasse is encountered, the problem is blamed on a specific aspect of the executing problem solving system, and in particular, an aspect of the representation it is employing. And because we organize representations in *difference networks*, it is then possible to select a better representation by following a *difference pointer*. We call this technique *failure-directed reformulation*. We demonstrate these ideas on a particular problem with a small test system possessing more than one representation, to illustrate the working of the control.

Thesis Supervisor: Marvin Lee Minsky
Title: Toshiba Professor of Media Arts and Sciences

Acknowledgements

I would like to thank the following people for helping me finish this thesis:

My family
Mahender, Kulwant, Raminder, and Vindi
for their love and support

My colleagues
Nick Cassimatis and Tim Chklovski
for always being available to discuss issues and ideas

My advisor and mentor
Marvin Minsky
Who taught me to think about my own thinking
Whose enchanting vision of the mind as a Society of Agents
is the reason I came to MIT in the first place
Most of these ideas are either direct implementations of ideas from Minsky's many
writings, or had their roots there

Oliver Selfridge
Who prodded me on for months and months, and gave me many ideas and insights
His example, for how to write and how to approach problems, was essential

Betty Lou McClanahan
For her encouragement and support
And for fending off the department after my missing too many deadlines to count

My many friends and colleagues in and out of MIT for
Discussion
Criticism
Support
Advice
Love

Table of Contents

1 Introduction

1.1 Preamble

Workers in the field of artificial intelligence have been tremendously creative over the years. They have produced a treasure trove of ideas in their quest to produce a human-level intelligence: a vast range of algorithms, heuristics, neural circuits, representations and other devices. The products of AI span the space of computational processes. Despite all attempts to the contrary, there does not seem to be any way to characterize all such mechanisms in terms of a simple unified theory, one that lets us see them all as parameterizations of some ideal.

We believe that this observation has not been taken seriously within the field. Researchers continue to try to build unified theories of intelligence, and those attempts continue to fail. Ideologies are built around particular tools such as neural networks, genetic algorithms, or statistical estimation. The thrust is uniformly to try to use one of these tools to solve all known hard problems, and any suggestion that the tool might not be up to the task is scorned.

Such behavior is given a well-known and familiar justification—Occam's razor, that given some phenomenon, the simplest theory that explains it is the best one. And we agree with Occam's razor itself. But we do not agree with the aesthetic principle in which it takes its most common form, that all theories should have roughly the number of components and interactions that have been discovered in physics, our most successful science. This principle is rarely stated explicitly, but it explains why the theories that are most popular in AI are all so simple, reflecting, as Minsky wrote in [MIN90], "theoretically neat, but conceptually impoverished ideological positions."

So in our research we take the opposite approach. Rather than try to find a single mechanism capable of solving all problems, we try to match types of problems to types of solutions. We are interested in how to build *heterogenous* AI systems, ones that integrate a diverse variety of mechanisms. Our goal is to achieve robustness and flexibility by building systems that are *resourceful*, that know so many different ways to solve problems that they hardly ever get stuck. We envision that in the future, computers will no longer suffer from their present-day rigidity and inflexibility, and will instead come built-in with millions of special purpose problem solving agents that give them ways to deal with virtually any problem they encounter.

To achieve this goal, we must explore how to organize large collections of agents into societies that exhibit high levels of resourcefulness and versatility. We concentrate on agent systems because they are the most sophisticated control structures known in computer science, as they allow an infinite variety of problem solving "personalities". Each agent may have a different set of methods, strategies, heuristics, knowledge, representations and all the other things that distinguish a particular way of thinking about things. And if one agent can't solve a problem, we can switch to another one with different way of looking at it.

1.2 Multiple Representations

The general problem of how to select or construct an appropriate agent to solve a given problem is difficult because agents have so many dimensions of diversity—their particular control structures, heuristics, representations, etc. In this thesis we focus on the dimension of *representation*. There are many ways to think about representations:

- as particular simplifications of the world
- as descriptions of prototypical things
- as data structures used by computational processes
- as languages for describing the world
- as collections of features or attributes of a thing or environment
- as problem spaces, such as the search space of chess
- as models of some domain, such as the blocks-world
- as ways of computing particular predicates or functions
- as collections of default assumptions and rules

- as theories of particular domains, such as Newtonian physics
- as simulated internal models of real external things
- as inputs to mechanisms like neural networks or situated-action systems
- as data structures with a simple semantics, like neural networks or logic

These viewpoints are not entirely mutually exclusive, and there are many others we have not listed. It seems that every AI worker has a different view on what representations are and how they ought to be used. This makes sense, as different representations are suited for different purposes. For our purposes, we will take the view that representations are data structures that let us express in machines the many kinds of knowledge needed to cope with the wide range of problem situations that one normally encounters in the world. The structures and types of the representations we might use have been deeply studied, especially for the problem of representing our commonsense world; good references are [LEN90] and [DAV90]. But less studied is the crucial issue of how to *select* an appropriate representation for the problem at hand. In this thesis we explore one aspect of that issue: How do we change representations if the current one starts failing us?

1.3 Failure-Directed Reformulation

We operate under the assumption that much of ordinary problem solving occurs using simple and schematic representations akin to those used in classical AI microworlds. We do this because, while no one has managed to build AI systems that exhibit high-grade performance in complex real-world domains, it has often been possible to build systems with simple representation that offer good performance in simple domains. However, in making this assumption we face the limitations of simple representations. At some point the representation will fail to account for some essential detail of the situation, make an inaccurate assumption about the world, restrict from consideration a possible method of solution, or impose some other limitation of viewpoint that in simplifying the problem makes it difficult or impossible to find a good solution.

Therefore it is essential that we study the heuristic method of *reformulation*, that is, of changing the representation of the problem. We are interested in reformulations that cause a substantial change in viewpoint, one that is likely to be different enough from the original viewpoint that we have a good chance of getting unstuck. In this thesis we present a simple way of organizing representations so that reformulation can occur quickly if a dead-end is encountered while applying the current method of solution. When an impasse is encountered, the problem is blamed on a specific aspect of the executing problem solving system, and in particular, an aspect of the representation it is employing. And because we organize representations in a *difference networks*, it is then possible to select a better representation by following a *difference pointer*. We call this technique *failure-directed reformulation*.

We demonstrate these ideas on a particular problem with a small test system possessing more than one representation, to illustrate the working of the control. The problem is a very ordinary one—how do you fit a large cardboard box through a small door? The solution is found by a change of representation: First we conceive of the box as a geometric solid, and from this perspective one way to fit the box through the door is to re-orient it. But the box turns out to be too big, so this fails. We then change our viewpoint to conceive of the box as a folded-up sheet of cardboard. Aha! From this new perspective, it is obvious that we should unfold the box first and then slip it through the door. We explore how such a change of viewpoint might take place.

1.3 Related Work

- **Marvin Minsky**. This work is based heavily on the ideas about frames, frame systems, and matching in Marvin Minsky's influential essay "A Framework for Representing Knowledge" [MIN75]; and additionally, on the ideas about agents, reformulation, differences, level bands, A-brains and B-brains in his seminal *The Society of Mind* [MIN85].

- **Newell, Shaw, & Simon.** Another important influence was Newell, Shaw, and

Simon's early description of the General Problem Solver [NEW59], which demonstrated the profound problem solving heuristic of eliminating differences, and to us served as a model for how to do a piece of AI research. We were particularly impressed by the vision shown in their little-known paper "A Variety of Intelligent Learning in a GPS" [NEW60], which described perhaps the first AI system that could be said to have an A-brain and a B-brain.

- **Roger Schank**. Our work is in many ways related to that of Roger Schank and his students, especially **Kris Hammond** and **Ashwin Ram**. We drew much from their notion of failure-directed learning in case-based systems, as described in [SCH82], [HAM90], and [RAM94].

- **Gerald Sussman**. His 1972 Ph.D. thesis on debugging [SUS72] described perhaps the first AI system that operated by recognizing problem solving failures in order to help direct the process of recovering from the failure. The system we present in this thesis is strongly motivated by this idea, although it differs from his approach in essential way: following a problem solving failure, instead of using diagnosis information to repair the failed procedure, we use it to change the active representation, so that we can find some other way to solve the problem.

- **Douglas Hofstaedter**. We were very inspired by Douglas Hofstaedter's various projects on treating pattern recognition as an analogy-making and reformulation process. No one has written as eloquently as he on the fundamental importance of reformulation in problem solving, e.g. [HOF95]

1.4 Overall Organization

This thesis is organized into 5 chapters beyond this introduction:

Chapter 2 We discuss why we need multiple representations, and describe a particular representation known as a *frame*. Systems of frames can be put together to build arbitrarily detailed representations of situations.

Chapter 3 We describe a new kind of problem solver that combines the well-known GPS system with a novel way of organizing solution methods that we call a *means-end hierarchy*. This problem solver is particularly dependent on a good choice of representation in order to find good solutions.

Chapter 4 Building on the ideas from chapter 3, we describe the heuristic of *failure-directed reformulation*. The heuristic is to change representations by associating how the current solution method failed with differences between the current representation and potential reformulations. This is done by organizing frames in a *difference network*.

Chapter 5 We demonstrate these ideas on a small test system. The system is described in some detail, to show how the ideas from the previous chapters can be implemented.

Chapter 6 We examine some important issues we did not touch upon in the previous chapters, and discuss the future of this approach.

2 Multiple Representations

We will begin by discussing why problem-solving systems need many representations and the ability to select an appropriate one for the problem at hand.

2.1 Why choosing a good representation is important

A *representation* is any kind of data structure that can be used as a substitute for something else, for certain purposes, just as a map can be used for certain purposes as a substitute for a city. Representations are used by problem solving systems to describe the problem situation, preferably in a simple way, by making explicit only those aspects of the situation relevant to their purposes.

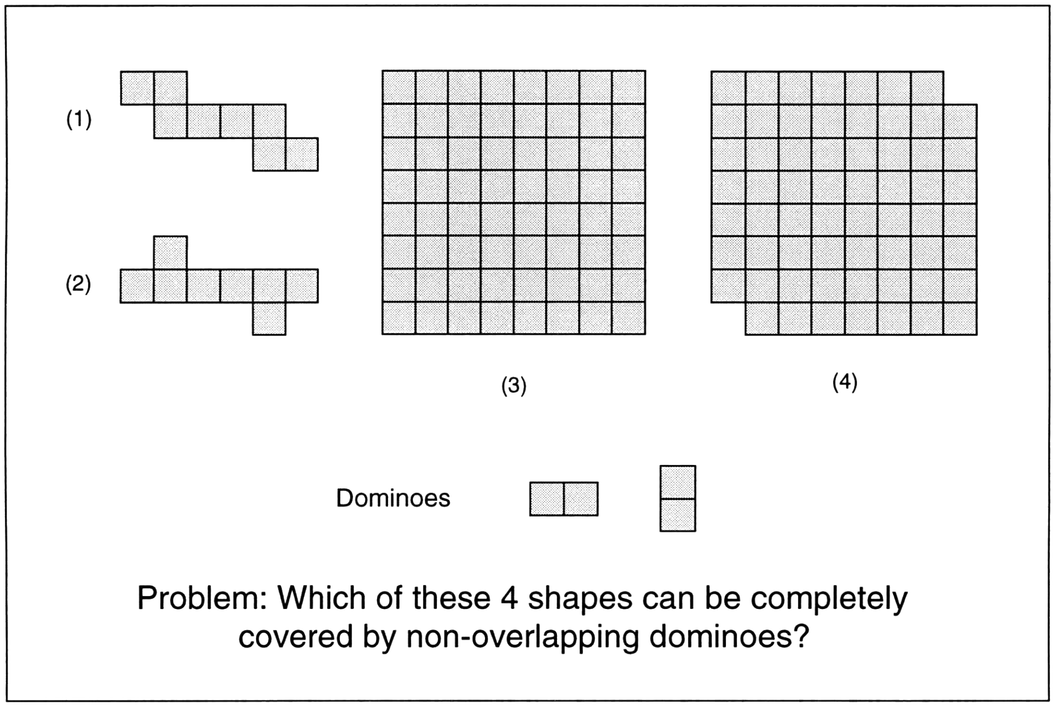
While representations take many different forms, in practice artificial intelligence programs are provided with a particular, fixed representation of the problem situation by their designers. Most of the design and computational effort is then focused on how to use that representation to solve some class of problems. In search-based methods, this effort takes the form of a search through possible sequences of worldly actions or mental inferences, leading to the construction of a plan of action or chain of reasoning that solves the problem; chess programs work this way, using a representation that reflects the positions of the pieces on the chessboard. In rule-based and case-based methods, the effort is in matching and adapting known solutions; the representation then indexes and describes a large collection of solutions or old experiences that apply to the present problem. In connectionist systems like neural networks, much effort goes into training them to exhibit useful behavior, where their training data is provided in terms of a particular representation.

It is well-known that choosing the right representation is important for efficient problem solving, because there is an enormous amount one can say about even the most trivial situation. Consider the simple situation "Mary gave John a present". I might focus on Mary's hair color, John's appreciation of the present, the physical motion of the present as it changed hands, the transfer in ownership of the present, or a vast range of other possibilities. It is essential to try to discard irrelevant details, so that everything about the world need not be considered while solving a problem. Search-based methods may search a very long time if the representation determines a problem space that is too large, and may not find a solution at all if that space was poorly chosen and does not contain a solution. Rule-based and case-based methods may find it very hard to retrieve a good solution if the wrong aspects of the current problem have been emphasized for determining similarities and differences from known problems. A neural network may need to be very large, and require an extremely large set of training examples, if the concept it is learning is not an easy function of its input representation.

The aspects of the world to be included in a representation should be chosen according to the purposes for which they are to be used. Consider describing a chessboard. For the purpose of *determining the best move for white*, the positions of the pieces are crucial descriptors. For the purpose of *the manufacture of the board*, the appearance of the wood and the kind of wood may be all that matters. Another way to see why purposes are important is if we make an analogy between a representation and a map. For the purpose of finding a way to drive from Boston to Montreal, it would be useful to have a highway map of the Canada-New England area to show us what routes are available, what paths lead towards and away from our destination, and what the distances are between the important landmarks. A map of the whole of North America might not depict potentially important local roads, and a map of Mexico alone would be of no use at all!

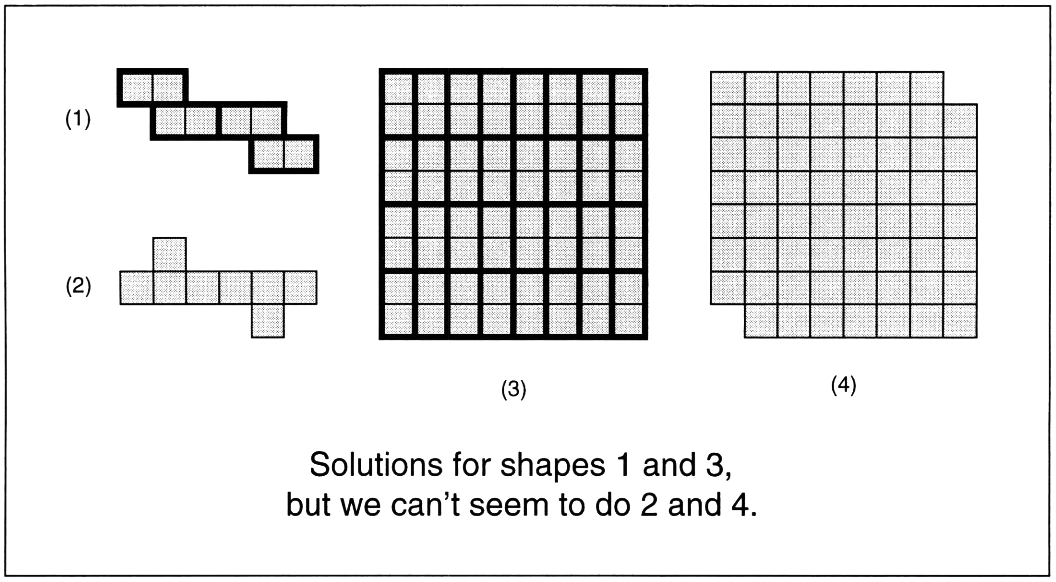
2.2 Example: Covering Shapes with Dominoes

Here is a little mathematical problem, due to Oliver Selfridge, that will illustrate some of these arguments:


(1)
(2)
(3)
(4)
Dominoes
Problem: Which of these 4 shapes can be completely covered by non-overlapping dominoes?

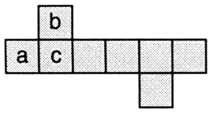
So the question arises: Why can't we do shapes 2 and 4? Better yet, how do we *prove* that we can't do those shapes? It turns out that two completely different representations are needed to prove the impossibility for those two shapes.

The first representation is basically a spatial one, where the method is to add dominoes until the shape is covered. This works well for shapes 1 and 3, as shown in the following diagram:



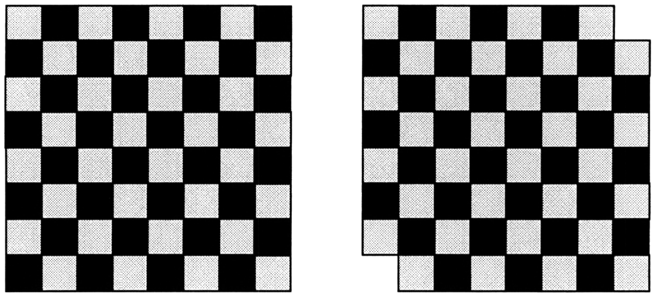
(1)
(2)
(3)
(4)
Solutions for shapes 1 and 3, but we can't seem to do 2 and 4.

And also on shape 2, we quickly reach an answer. Consider the squares labeled a, b, c at the left of shape 2:



It is clear that we can have either domino a+b, or a+c, but not both. So shape 2 can't be done.

But for shape 4, such a representation would be hopeless: try it! We need a completely different representation. It is this: in shapes 3 and 4 we are dealing with a checkerboard. Color every other square black, like a real checkerboard:



Look at shape 4. It has 32 white squares, and only 30 black squares. Since each domino has just one white square and one black square—well! Notice that this technique wouldn't work on shapes 1 and 2, because they both have the same number of black and white squares.

2.3 Hard problems require many representations

For simple problems, a single fixed representation is often effective. But for hard problems, choosing the right representation may not be easy to do in advance, for it may not be possible to anticipate the nature of the subproblems that will be encountered. While it is important that a representation discard irrelevant details, for difficult problems it may not be obvious at the outset which details are relevant and which are irrelevant.

The usual response to this is to try to give the problem solving system a representation that describes all aspects of the situation that *might* be useful. The trouble with this is we lose the crucial advantage of using a representation—to make explicit only the most important aspects of the situation. We drown the program in so many details that the problem becomes impossibly confusing. Further, while it is widely recognized that choosing a good representation is important, it is rarely acknowledged that every representation is deficient with respect in *some way* with respect to *some* problem. As we noted in the introduction, at some point the representation will fail to account for some essential detail of the situation, make an inaccurate assumption about the world, restrict from consideration a possible method of solution, or impose some other limitation of viewpoint that in simplifying the problem makes it hard to find a good solution.

We suggest an obvious solution to these problems: the programmer should not choose a fixed representation in advance, but should give the system many representations, and the capacity to *reformulate*, to change representations as work on the problem progresses. We believe it is worth a great deal of effort to find good ways to select representations, as choosing the right representation can simplify a problem to the point of making the solution obvious. One might even go so far as to say that the particular problem-solving algorithm we use is hardly relevant, as compared to the importance of first choosing a good representation in terms of which to cast the problem.

But how can we implement this idea?

2.4 Frames as a unit of representation

In order to build a system that can choose a good representation given a selection of many, we need to be able to represent representations themselves! That is, we need to be able to treat representations as things that can be individually selected and unselected, so that the system can choose among them in describing the world. To do this we will use a representational unit called a *frame*, first described in [MIN75]. With frames we can avoid having to use a single large representation, and can instead use many specialized frames, each representing a particular aspect of the situation. We can achieve a coverage of reality as broad or as detailed as we would like, depending on how many frames we choose to apply, but by restricting our consideration at any moment to only a small subset of these frames we retain the effectiveness of a simple representation.

What is a frame, exactly? A frame is a structure that is a model of a typical instance of something, where by a "model" we mean a device that can be used to answer questions about that thing. While there is no agreed upon way of building and using frames, they are usually implemented with at least the following three elements[1]:

- **Terminals:** A frame has a set of terminals, which are points to which other structures can be attached, often other frames but in general any other sort of representation. Terminals indicate that whatever is attached to them is an attribute, a structural component, or plays a certain kind of role in the context of

[1] We should note that since 1975 Minsky has expanded greatly on the frame idea. It is now perhaps better seen in terms of his newer ideas: K-lines, polynemes, pronomes, and level bands. These devices allow a more flexible way of building representations of things, roles, parts, and defaults, and frames are only one way to use them. In this thesis we do not use these newer ideas, partly because we feel we have not developed a sufficient understanding of how to use them, and partly because it is sometimes better to start with simple ideas and understand them well, and only later elaborate them to make them more powerful.

the thing being represented by the frame. Terminals are usually implemented in computers as variables containing pointers or symbols that refer to other data structures.

- **Default Attachments:** Frame terminals are never empty or left dangling. There is always at least a *default attachment*, which represents the usual or typical values those terminals take on. These are considered *weak* attachments, as they are overridden by any structure later attached to those terminals. Default attachments distinguish frames from other kinds of representations, because they are what make a frame a representation of a "typical instance" of something, rather than just a collection of attributes.

- **Markers:** For each terminal there may be associated *markers* that constrain what can be attached to that terminal. How markers are implemented varies from system to system. One way to build terminal markers is by applying processes that signal a problem when important differences are detected between the default attachment and what is actually attached. We will not be using markers in this thesis.

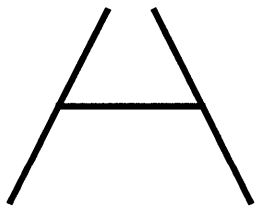
For example, the following frame could represent the event described in the sentence "Alice drove from Boston to Montreal":

TERMINALS	ATTACHMENTS
Driver	Alice
Origin	Boston
Destination	Montreal
Difference	Location
Vehicle	Car
Action	Driving

So we see that we can "ask a question" of a frame by accessing the frame terminals. Whatever attachment we find there can be seen as the answer to that question. If we access the **Vehicle** terminal in the frame above, we will find **Car**, which is perhaps represented as a frame itself with its own set of terminals describing the car's cost, brand, size, etc. In general, one frame is not sufficient to describe a situation because there are usually important things in the situation that deserve to be separately represented. Therefore frames are normally put together in *systems*, collections of frames that are connected to one another in order to describe something that no single frame could. The frame above would be a part of a larger system if the various attachments – **Car**, **Driving**, etc. – were themselves represented using frames.

2.5 Using Frames

Frames are selected using some sort of *matching process* that can find a good frame to describe the current situation or object of interest. We are not particular concerned with the details of such processes here. However, an important observation is that often a thing may reasonably be represented using any one of a number of frames. Consider the following figure, due to Oliver Selfridge:



Is this an 'A' or an 'H'? The question has no best answer, since we can choose to see it either way, that is, we could choose to assign either an 'A'-frame or an 'H'-frame to our sensory representations[2]. Selecting a frame therefore amounts to choosing a way to

[2] We do not deal here with problem of *perception*, so that we can focus on higher level ideas. Normally, frames would be activated because they provided a useful description of the world as seen through the system's sensors. Activating a frame could be thought of as turning on all the necessary perceptual

interpret a situation. In the same way (and to anticipate the example we will use in chapter 5) when one looks at a cardboard box, one might see it as a container, a geometric solid cube, a folded-up sheet of cardboard, or any number of other interpretations. However, some of those interpretations may be more typical or consistent with what we know is true about the box than others.

When we apply a matching process to try to match the current situation to a known frame, it is unlikely that there exists a frame that matches the situation *exactly*. This is especially true if the known frames are defined with many terminals or if the matching process is recursive and matches subframe terminals also—for then we compare so many features that there is almost certainly a difference at *some* terminal. In practice we must consider certain differences as unimportant in order for matching to be possible at all. In general, we can use any frame to describe anything, if we consider enough differences to be unimportant!

For example, in certain situations it could be useful to see the cardboard box as a geometric solid of a certain shape and size, say if we are moving the box from one place to another without regard to its contents. We might have had experiences moving a "geometric solid" object in the past, but to apply what we learned about manipulating that object to the present situation, we would need to regard as unimportant the blatant difference between the present situation and the past ones: that the cardboard box is hollow, while the past objects were not. We can do this because we know that, for the

---

processing to compute its descriptors. But going from sensory representations to the kinds of high-level representations we are dealing with here surely involves a great variety of complex pattern recognition machinery and all kinds of intermediate representations. The problem is tremendously hard, and so far there has been little success in building flexible and reliable perceptual systems. We strongly suspect that using multiple representations will make such systems far more robust. For even if it is difficult to produce one kind of representation of an object or situation from sensory data, it may be easy to produce another! We believe that much of the unreliability of present perceptually-oriented AI systems can be attributed to fixing a single representation to serve as the interface between the world and higher levels of the system.

purpose of manipulating something, the solid-hollow distinction usually doesn't matter.

But we must always be prepared to change representations if the current one begins to fail us. If we decided we were done using the cardboard box and wanted to store it away, the geometric solid interpretation might prevent us from seeing good places to put it. For this new problem, it would be far more useful to see the cardboard box as a folded-up sheet of cardboard, which would allow us to think about how to unfold it and how we might slip it into an unobtrusive location at the back of our closet.

How do we choose which frame we should switch to, given that we have decided to change representations? That will be the subject of chapter four. But first, in chapter three, we will describe a simple problem solving system that uses frames to represent the problem situation. We shall see that the choice of frame greatly affects whether a good solution can be found.

3 Means-End Hierarchies

In this chapter we will describe a simple problem solving system that uses the frame representation introduced in chapter two. We shall see that the choice of frame the solver uses to describe the problem situation greatly affects whether a good solution can be found. In chapter four we will discuss how to augment this system so that it can change frames in an informed manner when an applied solution fails.

3.1 What is a problem?

Problem solving is about achieving goals, about transforming some aspect of our situation into a more desirable state. A powerful way to think about this process is as *eliminating the important differences* between the present situation and some ideal situation in which our goals are met. Let's say we want to go somewhere. Our goal might be outside, very distant, or on the other side of a high fence. Eliminating or reducing any of these differences gets us closer to the goal, possibly in the literal sense of spatial distance, but certainly in the more subjective sense of "reducing the difficulty of the problem". This was the key insight used in one of the earliest AI programs, the General Problem Solver (GPS) by Newell, Simon, and Shaw [NEW56][3]. GPS solved problems by

(a) recognizing important differences and

(b) taking actions to eliminate those differences.

Let us refer to any problem solver that works this way as one that employs *difference-driven control.* This is essentially the form of control used in simple control systems, which attempt to solve problems by reducing a measured error between some variable in

[3] Curiously, the idea of eliminating differences was abandoned shortly after GPS, possibly because Newell and Simon came to regard it as "just another problem solving heuristic".

world and a “set point” that determines the ideal value of that variable. John McCarthy has referred to GPS as applying “symbolic negative feedback”. This is appropriate as historically difference-driven methodology derives from the application and analysis of negative feedback in assorted engineering applications.

The GPS program implemented the “recognize difference, take action” behavior in a very simple way, by applying a set of rules. For instance, for the purpose of “going to the store” GPS might be programmed with the following rules:

- if the store is across the street, walk across the street
- if the store is on the other side of a high wall, then go around the wall
- if the store is very distant, take a taxi there

The condition on the left describes a difference we want to reduce, and the action on the right describes a way of eliminating that difference. GPS was “reactive” in the sense that it responded directly to the difference between the situation and the goal, and did no advanced planning. Yet it could solve quite tricky problems in logic and other well-defined puzzles.

Unlike most modern programs, GPS was usually given several ways to attack any given type of problem. For instance, to change an object’s location, one might try to apply any of the methods ‘push’, ‘pull’, ‘throw’, or ‘carry’. If one method failed to reduce the difference, another method could be tried. Therefore it is convenient to think of a GPS program as a table associating problem types with method types, or a *difference-method table*. An example table is shown in Figure 3.1.

But difference-method tables have a serious limitation: if there are many methods that are known to eliminate a particular difference, there is no way to select between those methods so that only the most appropriate ones are considered. For while we may know a variety of ways to make a certain kind of change, some of those ways may not apply in our present situation, others may require too many resources, and still others may have

Figure 3.1: Difference-Method Table

	Difference Types			
	Change of Location	**Decrease in Height**	**Change of Knowing**	**Increase in Sortedness**
Method Types	push	squeeze	telephone	bubblesort
	carry	fold	yell	quicksort
	throw	disassemble	speak	mergesort
	pull	rotate	write letter	insertionsort

costly side-effects. Further, while there may only be a few methods for each problem type, such as pushing, throwing, or carrying as ways of moving something, there are many more *variations* on such methods. For throwing a baseball is not like throwing a balloon, pushing a bureau is not like pushing a person, and carrying a cat is not like carrying a glass of water.

In the original GPS system, the different methods were simply considered one after the other, as they were given fixed priorities. But if we have hundreds of method variations rather than just a few general methods, this could be a very expensive proposition. Newell and Simon later proposed a way of automatically refining the difference-table by applying learning [NEW60], but despite the potential power of the learning approach they described[4], it was fundamentally hampered by the limitations of the difference-

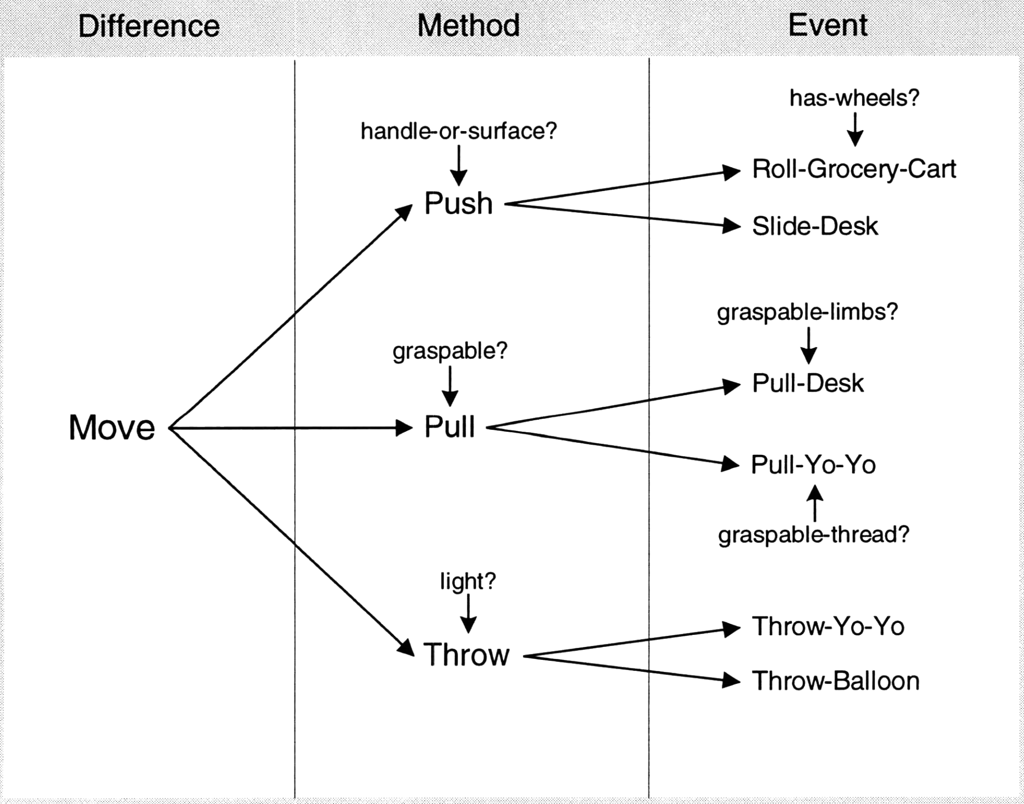
[4] They applied a meta-GPS to "diagonalize" the difference-method table of another GPS!

method table as a representation.

3.2 Means-End Hierarchies

We offer a simple augmentation to the difference-method table idea that helps deal with the selection problem between the many potentially useful methods. Instead of using a table, we use something more like a tree. We propose that every major type of difference sits at the top of a tree of methods for reducing that difference, and that this is a useful way to organize "how-to" knowledge in a problem solving system. We will call such a structure a **Means-End Hierarchy**. One such hierarchy is depicted schematically in Figure 3.2, describing various solutions to the problem of moving an object from one

Figure 3.2: Means-End Hierarchy for Moving Something



place to another.

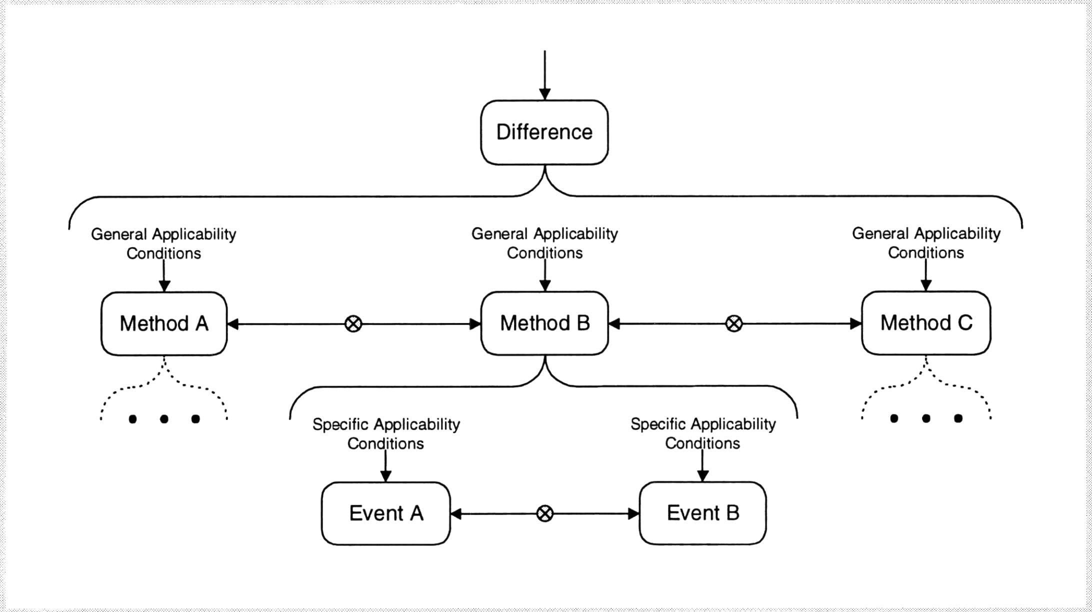
The most important feature of a means-end hierarchy is that there are many means to achieve the given end of eliminating the topmost difference. The network branches out to a collection of different methods, derived from different experiences where we made something change its location. The hierarchy depicted above is deceptive in its sparseness–there may be dozens of variations on every type of solution method.

Each method has a certain domain of applicability, the conditions under which applying that method causes the change described by the topmost difference. For instance, we may only be able to apply the **throw** method if the object we are throwing is light. This knowledge is reflected at each node in the tree as a list of conditions that must be true for the more specific methods underneath that node to be considered.

Because applying a method may be a significant investment, we want to try to find the method that is most likely to work. The particular heuristic we use is to find the method whose domain of applicability matches most specifically what we believe is true about the current situation. Because checking these conditions could be an expensive proposition, and because a rich means-end hierarchy may index hundreds of methods, finding an appropriate solution to the current problem cannot be done efficiently by an undirected search of the tree of methods. So to find a solution we apply the general control structure[5] shown in Figure 3.3.

[5] This general form of control structure has been known for a long time. It was used by Niko Tinbergen in the 1940s to explain the behavior he observed in the Stickleback fish. The biggest difference from the usual use of this structure and from ours is that in our case the different sub-behaviors in the network are different ways to solve the *same* sort of problem.

Figure 3.3: General Control Structure


Difference
General Applicability Conditions
General Applicability Conditions
General Applicability Conditions
Method A
Method B
Method C
Specific Applicability Conditions
Specific Applicability Conditions
Event A
Event B

Once we have recognized the topmost difference, we proceed to find an appropriate solution method. The selection process is basically like that of a decision-tree. We start at the highest level of the tree and select the method at that level which best matches the current situation. For our purposes, we will assume that an exact match is required, and that the hierarchy has been carefully structured so that it is unlikely multiple matches occur. Once we have selected a method, we then consider the next level of methods underneath that one, and repeat the match-selection. This process is repeated until we reach the bottom of the tree, where we find a specific solution method that we then activate. For larger networks, this search process would probably need to involve constraints flowing from bottom-up as well, and external controls of various sorts. But this simple control structure is sufficient for our present purposes.

3.3 Example: Moving a Heavy Desk

Let us consider as an example the problem of moving a heavy desk to the other side of the room. We quickly see that the important difference is one of location, and let us say this causes us to activate the **move** means-end hierarchy shown in Figure 3.2. We must first make a decision about whether we want to **push**, **pull**, or **throw** the desk. Let us assume that we represent the heavy desk with a frame that has the following terminal assignments:

TERMINALS	**ATTACHMENTS**
Weight	Heavy
Has-Handle?	No
Has-Wheels?	No

The method **throw** requires that the object be light, whereas **pull** requires something to grasp onto. Hence we choose **push**, for which there are no major differences at this level. We now move down to the next level under **push**. Here we must choose between **roll-grocery-cart** and **slide-desk**. The method **roll-grocery-cart** requires wheels, so we try **slide-desk**, which works.

3.4 Means-End Hierarchies and Level Bands

The means-end hierarchy is a simplified version of Minsky's "level band" idea, as described in [MIN85]. The level band idea is that a description of something can be divided into many partial descriptions at many levels of abstraction. These partial descriptions may range from being highly abstract *functional* descriptions of how that thing might be used, to highly detailed *structural* descriptions of how that thing might be recognized. By bridging structure and function this way, we can organize what we know about things so that we can quickly find the ideal description of our situation to solve our current problem.

We do not implement here the full level band idea, which would allow bottom-up constraint as well as more powerful matching processes such as ring-closing (also described in MIN85.) But we do use the important subidea of a general-to-specific organization of solution descriptions. In a means-end hierarchy, the highest level is a type of problem, like changing the location of an object, quenching one's thirst, or communicating something to someone. At the middle levels are types of solutions these problems. One can move something by pushing it, pulling it, throwing it—or if it is person, by asking it. At the bottom levels are specific solutions to problems, perhaps particular remembered experiences; one might have moved a desk before by sliding it along the floor, or have pulled a wagon down the street, or have asked a friend to come into our home—all ways of moving different kinds of things.

In summary, a means-end hierarchy can be thought of as a rule that is triggered by recognizing the topmost difference, and that triggers a search for an appropriate method for reducing or eliminating that difference. While this may seem a simple idea, we believe it underlies a great deal of intelligence[6], that is:

For every type of problem, we know many different ways to solve it!

3.5 Why the representation matters so much

In chapter two we discussed the importance of choosing good representations. How do those arguments apply to the augmented GPS we described in this chapter? We see from the example in section 3.3 that had the representation of the desk been slightly different, say, if we assumed that the desk was light, we might have tried throwing it instead of pushing it. The particular frame system used to represent the desk controls how we traverse the means-end hierarchy, as it determines what aspects of the desk we consider

[6] This thesis must seem to be about very uncreative machines, since nowhere do we do any constructive creation of fundamentally new solutions to problems! But we believe that far more of our intelligence is due to matching and adapting existing solutions than it is to search-based construction of new solutions—any search that occurs happens implicitly in the matching process, as new options are considered and the priorities among our options change. People have very definite styles and ways of doing hard things, like speaking and writing and programming. This can only be a consequence of everyone constructing their own elaborate agencies for doing these things.

at all and the values they take on, by assignment and by default.

So we see that the problem solving system we describe here depends on the representation for the reason described back in section 2.1: Rule-based and case-based methods may find it very hard to retrieve a good solution if the wrong aspects of the current problem have been emphasized for determining similarities and differences from known problems. If the right frame system is chosen to represent the situation, the system will descend the means-end hierarchy to a good solution. But if we have locked on to the wrong representation, we will not consider solutions that do not make sense under that interpretation. In general, if a good representation is chosen, likely-to-work solutions will be considered and not-likely-to-work solutions will be shut out.

Often, if we are stuck, a minor change to our representation will suffice to get us unstuck, like noticing that a desk has a protrusion we might grasp, allowing us to pull it. But sometimes a more drastic change is necessary. Take the dominoes-covering problem from chapter two: it did not occur to us to consider counting the squares until the reformulation occurred, which caused us to see the board as a *checkerboard* rather than simply a *grid*. Until then, we could not consider the solution that worked.

In the next chapter we discuss some ideas about how to change representations in this way. We focus in particular on this question: if we find a method and it fails, how do we choose another method? The simplest solution is to backtrack in a best-first search, or even better, backtrack to those branch points at which we made decisions with the greatest uncertainty, in the hopes of finding the "next best" path. The trouble with these sorts of approaches is that deciding on things like uncertainty may not be of great help if our assumptions have forced us into an unproductive part of the search space, e.g. in the dominoes problem. For as long as our assumptions are essentially wrong, we will keep trying nearby variations and never get unstuck.

How then can we change our assumptions? One way is to have an external system that searches over all *representations*, and re-runs the means-end hierarchy for each one. This means applying all known frames to the current situation. Then we will try many new paths down the means-end hierarchy. The trouble is that we may know hundreds of frames for which we have some reason to believe might be a better way to look at things—and if we are willing to try such a large search, why not go one step further and simply search the whole means-end hierarchy?

In the next chapter we discuss this problem further, and offer a better solution. To avoid an undirected search over representations, we organize representations in a configuration known as a *difference network.* An external system then decides how to change representations by associating how the current method failed with differences between the current representation and potential reformulations.

4 Failure-Directed Reformulation

We have discussed how to build a problem solver that operates by matching the current problem to known problem types for which solution methods are available. The trouble arose of what to do when those solutions failed to work. In this section we discuss a method for recovering from failure by reformulating the situation, with the hope that by finding another way of looking at the situation we might be able to find a better solution. We call our approach *failure-directed reformulation*, because it uses the nature of the failure to direct the selection of a new representation of the situation. Changing representation is treated as taking a kind of “mental action”, one that attempts recovery from the failure by making a change not in the world, but in the problem solver itself.

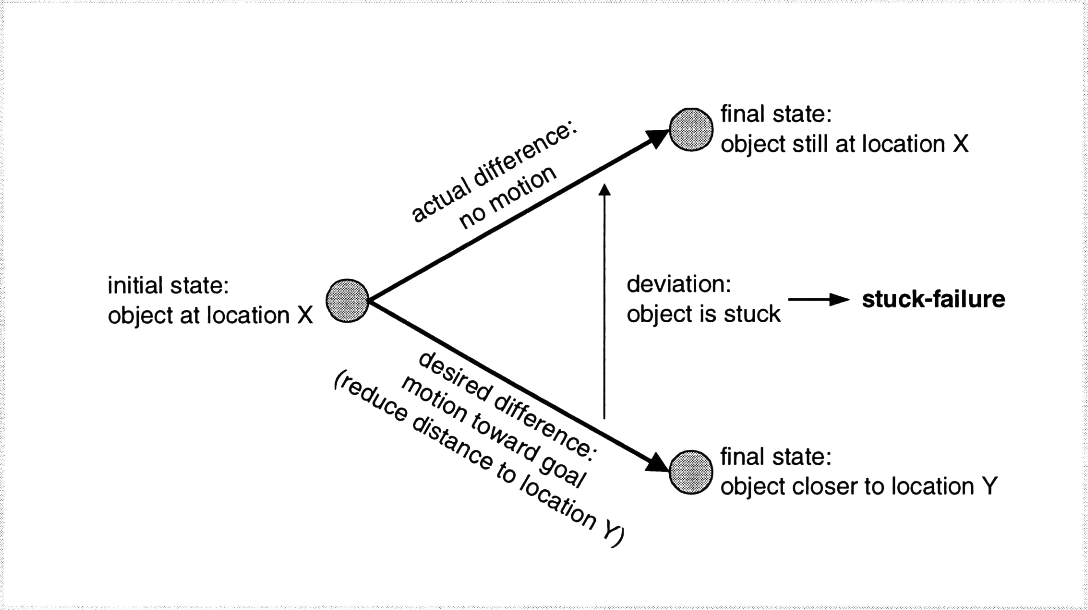
4.1 Expectation failures

Before we can deal with a failure, we must first detect it. It is not easy, in general, to detect when things are starting to go wrong. When deep into a hard problem, it can be difficult to distinguish progress from regress. How do we know if a given chess move will assure our victory, or will be the cause of our defeat? Sometimes we may not be able to tell directly that we have reached a dead-end, but we can detect unproductive behavior patterns in the problem solver, such as considering the same possibilities over and over, or producing nothing but mediocre, unlikely-to-work options. Detecting signs of failure may be so difficult that it may be important for a problem solver to have vast arrays of knowledge about what *not* to do, so that it does not even begin to consider methods of solution that are likely to fail in serious ways.

To simplify things somewhat, we will concern ourselves solely with a particularly easy-to-detect kind of failure, which we will call an *expectation failure*. An expectation failure occurs when we fail to properly predict the outcome of an action or solution method. For example: suppose that you are hungry, and seeing a basket of fruit on a nearby table, grab an apple from it and bite into it—only to hurt your teeth on the plastic apple! We expected to be able to bite into the apple, but encountered an expectation failure because the "bite" action did not proceed as expected[7].

We take a simple approach to detecting expectation failures. We have, for every solution method, a representation of its usual or expected course of action, and ways to recognize serious deviations from that course. For the general method **move**, the representation might let us describe the usual event "object is moving", and serious deviations like "object is not moving" or "object is moving too slowly". We classify such negative deviations into failure types. For example, if we try to **move** something and we detect

Figure 4.1: Move Monitor



[7] As an example of a failure that is not an easy-to-detect discrete event like failing to bite into a plastic fruit, consider sorting a large list. There are many different sorting techniques, like bubblesort, insertion sort, quicksort, etc.—and they all work, in the sense that they can successfully sort of list of any length. What matters is how long the technique takes, and that depends on the nature of the list to be sorted. Quicksort takes as long to sort a list that is already fully sorted as to sort a random one. Insertion sort knows a sorted list at once, but on a random one it takes far longer than Quicksort. So sometimes "failure" might be better thought of as exceeding some cost, rather than avoiding a discrete circumstance.

that the object isn't moving, we might classify this failure as a **stuck-failure**. This idea is summarized in Figure 4.1.

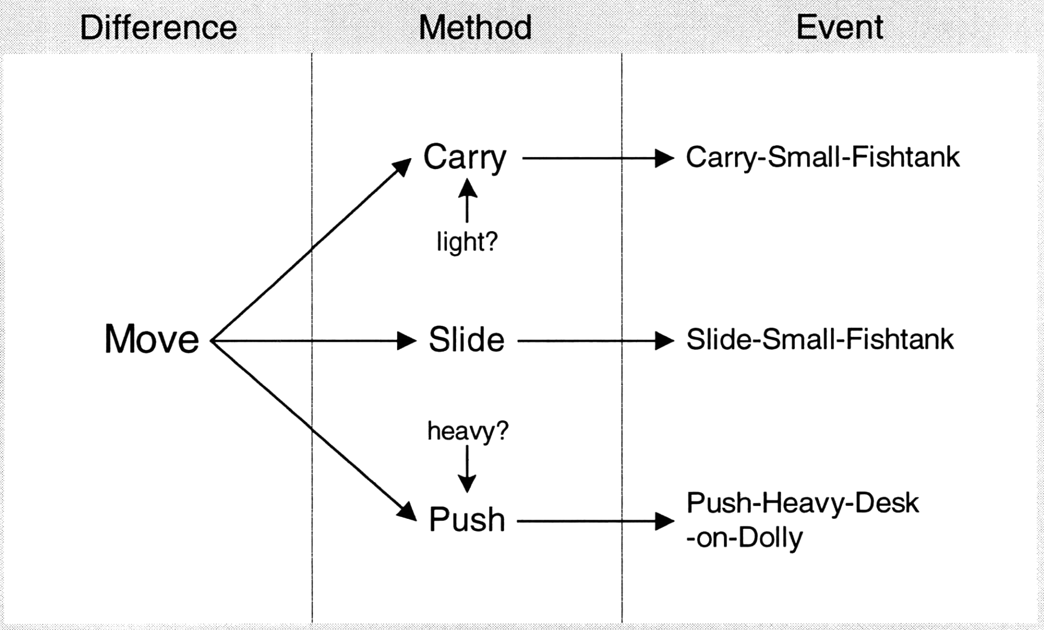
So to recognize an expectation failure, we associate with each solution method a monitoring process. This process runs in parallel with the solution method, and watches to see if the situation is proceeding according to expectations. If the situation plays out in a different way, then the process attempts to classify the difference between the expectation and what actually happens. Some differences are considered serious, and if such a difference is recognized, the process signals that the method has failed in a particular way. This signal then triggers another process that decides, based on the failure type, on a method for recovering from the expected failure.

4.2 Many ways to recover from expectation failures

Once we have recognized that an expectation failure has occurred, how can we recover from the trouble? There are many ways to do so, and we propose that these different methods of recovery can themselves be selected in a difference-driven way. In particular, we will apply the means-end hierarchy idea from chapter 3 to the problem of how to recover from an expectation failure! In the terminology of *The Society of Mind*, we propose to use a *B-brain* agency, one concerned not with problems in the outside world, but with problems in another agency, an *A-brain*, that is in turn concerned with problems in the outside world. The "difference" reduced by the B-brain hierarchy is whatever caused the failure signal. One way to think about this difference is as the "2nd derivative of progress". If we think of the first derivative of progress as describing the process of getting closer to the goal, then the second derivative would describe deviations in this process, such as stopping or slowing down. In that case, we would have an expectation failure and should attempt to re-evaluate our strategy.

To be more concrete about all this, consider the following example. Suppose we are trying to move a very large fishtank that weighs several hundred pounds. Perhaps we have something like the means-end hierarchy of methods of Figure 4.2 available to us,

Figure 4.2: Means-End Hierarchy for Fishtank Example



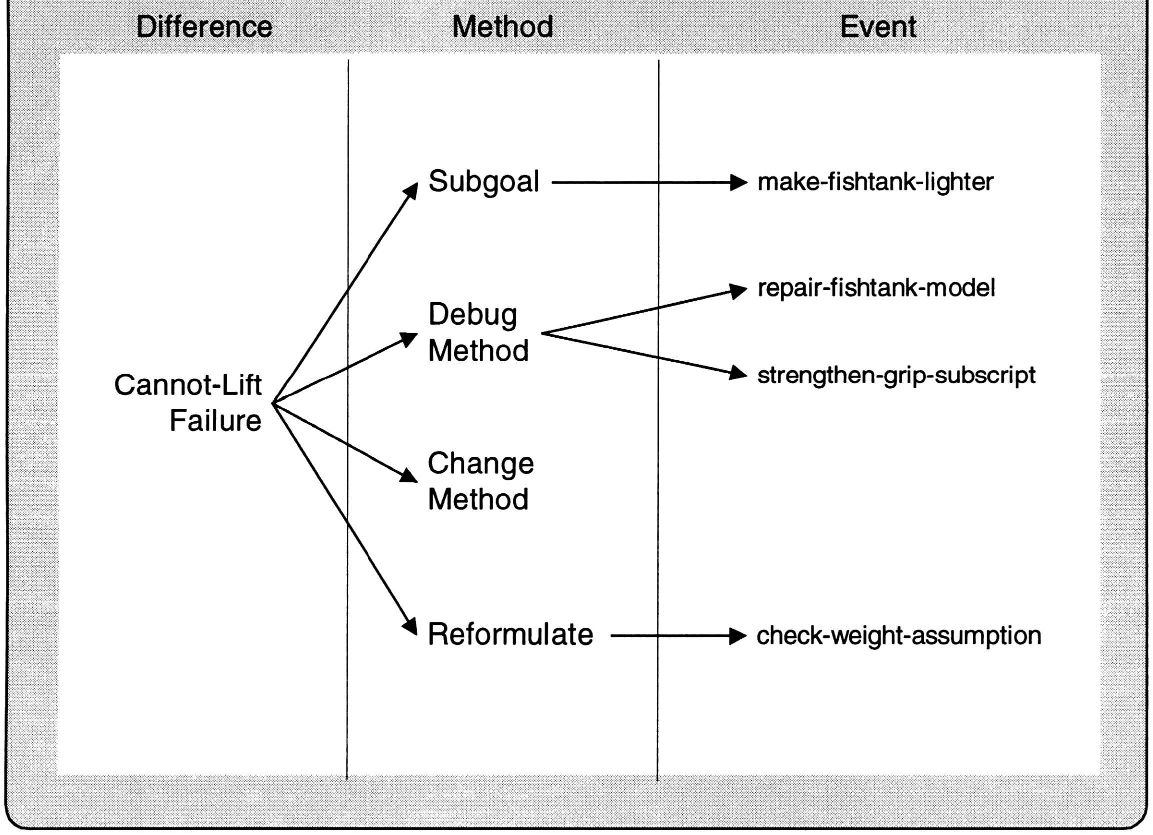
drawn from our past experiences. There are two representations, **small-fishtank** and **heavy-desk**, and let us say that we have initially matched to **small-fishtank**. The first thing we might do is try to apply **carry-small-fishtank**, a method that instructs you to pick up the fishtank and carry it just like it was a small fishtank. We try this and fail, raising a **cannot-lift** failure signal. What should we try next? To deal with this failure, assume we have available to us the recovery options shown in Figure 4.3: subgoaling, learning, changing method, and reformulating. Let us consider each of these in turn.

- **Subgoaling**. This is the most usual way to recover from failures in an AI system. We try to bring the world more in line with the conditions required by the method we just tried to apply, by treating those conditions as subgoals to be achieved. The **carry-small-fishtank** method requires that the fishtank be light, so we adopt the subgoal of making the fishtank light. One way to achieve this subgoal is to empty the fishtank before trying to move it. While that might work, perhaps we rule it out because there is no place to put the water and the fish.

The next three recovery methods assume the problem is not in the world, but in *us*. That is, they try to make a change in our models or our approach, rather than continue to make the current approach work.

- **Learning**. We can try and repair the failed method by learning from the experience. In particular, we assume there is a bug in the applied method, so we try to debug the failure and then try the method again. For example, if we had dropped the fishtank, we might decide to modify **carry-small-fishtank** so it required that we first get a good grip on the fishtank before trying to move it. This is the kind of recovery method used in Gerald Sussman's HACKER program [SUS72], which knew many ways to debug broken programs.

Figure 4.3: Hierarchy for Dealing with "Cannot-Lift" Expectation Failure



- **Changing method**. We can declare trying to make this method work a lost cause, and find another method of solution. So rather than continuing to try to make **carry-small-fishtank** work, we can re-run the A-brain means-end network with the failed method suppressed. Perhaps we might try **slide-small-fishtank** instead. In fact, we might even re-run the means-end hierarchy without suppressing the current method. For things, both in the world and in the problem solver, have no doubt changed somewhat since the failed method was initiated, and so the frame system describing the situation might now represents things a little differently. In that case, re-running the hierarchy might lead to the selection of a different and better method.

This suggests a new possibility—why not *actively* change our representation? If we could find a substantially different viewpoint that sufficiently fit the constraints of the present situation, a viewpoint different enough from the current one, then we might have a good chance of getting unstuck. This leads us to consider the final method of recovery:

- **Reformulating**. Perhaps we were looking at the situation in the wrong way, one that caused us to select a method that wouldn't work. In this example, we thought of the fishtank as a variation on a small fishtank, but applying the method that worked for a small fishtank failed for the actual fishtank, which was big and heavy. So perhaps the best thing to do in this situation is to change our representation of the fishtank, from thinking about the fishtank as a **small-fishtank** to thinking about it as a **heavy-desk**. Re-running the means-end hierarchy then causes us to try **push-heavy-desk-on-dolly**, drawn from an experience moving a heavy desk using a dolly.

In general, such changes of representation can open up new paths of action, remove constraints on the way we apply actions, and change our perspective, so that it no longer makes as much sense to do what we were doing and so forces us to consider something new, as well as causing many other changes in our problem solving approach.

4.3 Failure-Directed Reformulation

Finally, after laying much foundation, we can present the main idea of the thesis, the notion of *failure-directed reformation.* In the fishtank example, there were only two representations, and so when we decided to reformulate it was obvious which new representation we should try. But let us suppose that there were dozens of potential reformulations instead. Is there a non-arbitrary way of choosing a new representation? Why not think of the fishtank as a **cardboard-box** or as a **glass-window**? Can we find some basis for guiding this selection, so that we can go from thinking of the fishtank as a **small-fishtank** to the fruitful reformulation of thinking of it as a **heavy-desk**?

To help guide reformulation when there are many potential representations, we connect frames into an organization known as a *difference network*, first described in Patrick Winston's Ph.D. thesis [WIN70]. In any substantial frame-based system there will be sufficiently many frames that fast access to them will require that they be organized into some sort of retrieval network. In a difference network, we organize frames so that they are linked by descriptions of the differences between them. Not all frame pairs are necessarily linked, and not all the differences between any pair need be described. For example, the **small-fishtank** frame and the **heavy-desk** frame might be linked by something like the following structure:

small-fishtank $\rightarrow$ **"difference-in-weight: heavier"** $\rightarrow$ **heavy-desk**

The key advantage of the difference network organization of frames is that it is optimized for retrieving frames that differ from a particular one in some particular way. This has the following advantage for reformulation:

> *If we can lay the blame for a problem-solving failure on an aspect of the representation we were using, we can try to find a new representation that doesn't suffer from the deficiency that caused the failure, i.e. that is* ***different*** *from the current one in that one aspect.*

However, what aspect of the representation should we consider at fault? Consider some of the many dimensions of a representation:

- the types of attributes of what they represent
- the capabilities of the methods that use them
- the kinds of problems they are helpful in dealing with
- the assumptions they make about what they represent

All of these properties are useful in that they might determine the difference between representation that lets us find a good solution and one that does not. However, we focus here on the last dimension listed: the assumptions they make about what they represent.

So how might reformulation happen in the fishtank example, given that our different representations are organized in a difference network? Let us say that we cannot move the fishtank by picking it up, and so we decide that something is wrong with how we are representing the fishtank. If we consider our past experiences, we find that sometimes we cannot lift something because we have assumed something wrong about thing's weight, i.e. an incorrect assignment to its **weight** terminal. So perhaps the reason we could not lift the fishtank is because we mistakenly assumed it was light enough to easily manipulate. So the way we then reformulate is by asking the difference network to retrieve a new frame that differs from the current one in that it does not make this mistaken assumption. If all goes well, this will invoke a new representation that works better, in this case the **heavy-desk** representation. If we adopt this new representation, re-running the means-end hierarchy causes us to try **push-heavy-desk-on-dolly**, and the problem is solved.

In chapter 5 we describe a test system that demonstrates these ideas. We discuss them with greater specificity, and show one way of implementing the difference network organization described above.

5 Problem: Moving a Big Box through a Small Door

5.1 The Problem

We tested the heuristic of failure-directed reformulation on a simple, illustrative problem, inspired by an ordinary, real-world problem that we all learn many ways to solve. The problem and a solution are described in this imaginary episode:

> *"You are six years old. Your family is moving to another city, and everyone is helping with the packing. Your mother asks you to fetch a cardboard box from the next room to put books in. You run to the next room and find one box left. It's a big one, almost as big as you are! You pick it up and head for the door—only to discover that the box doesn't fit. You reorient it a bit and try again, but still no success. You repeat this a few more times, with increasing frustration. Suddenly, you get an idea—unfold the box! You pull clumsily at it until it comes apart, and slip the now flat box through the door. Now if you can only figure out how to fold it up again..."*

What might have happened in your mind for you to have considered unfolding the box? We posit that, at first, you might have been representing the cardboard box as a geometric solid, a three-dimensional rigid object. But then, when you couldn't fit the box through the door, you changed your viewpoint by re-representing it as a folded-up sheet of paper, a two-dimensional flexible object. With that new viewpoint, an obvious thing to do was to unfold the box.

5.2 The Microworld

We built a simple microworld in which to simulate the above problem. We have moved down one dimension: the microworld is two-dimensional, consisting of two rooms connected by an open aperture. The cardboard box is modeled as a U-shaped line. The

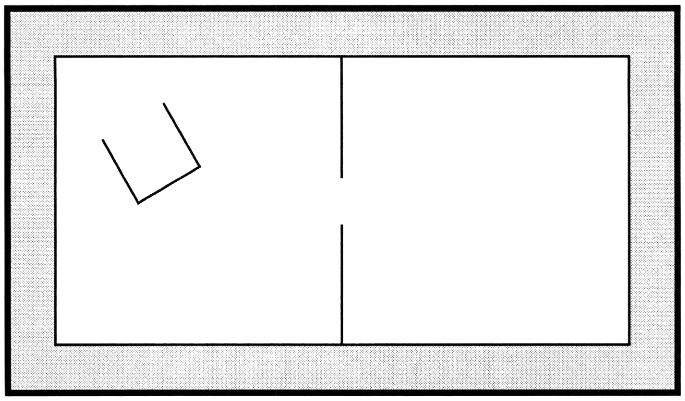


Figure 5.1: The Microworld

world appears as shown in Figure 5.1. We built a problem solving agent that operates on this microworld. The agent can take a few simple, discrete actions on the box:

- move up, down, left, right
- rotate clockwise, counterclockwise
- fold, unfold

The agent has a few simple sensory representations of the microworld:

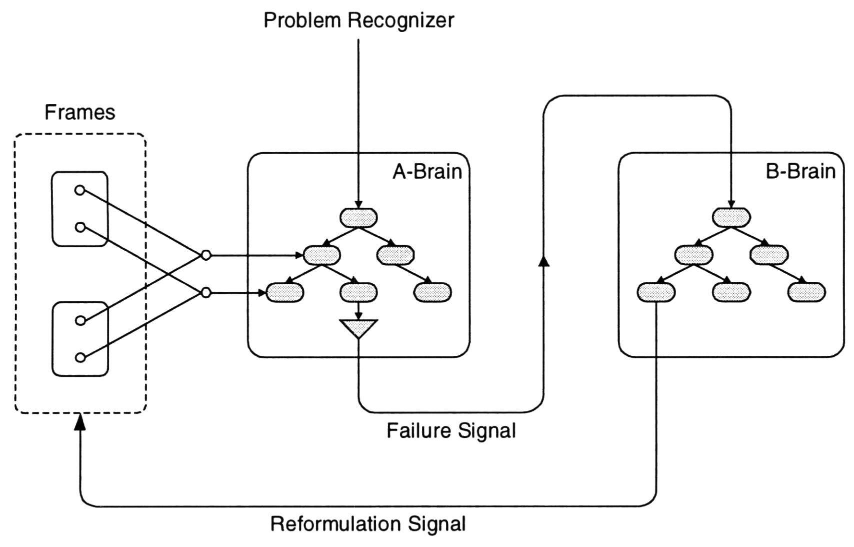
- position of box
- orientation of box
- horizontal and vertical size of the box
- position of aperture
- vertical size of aperture

The agent also receives a signal if the 'rotate' action fails to change the vertical size of the box (which is used by the failure-detection mechanism described further on.)

5.3 How the system solves the problem

The system that solves the above problem is built from two means-end hierarchies that are connected in the general arrangement shown in Figure 5.2. The precise meaning of the various symbols will become clear as we explain how the system solves the problem.

Figure 5.2: Overall Architecture



Let us assume that we begin by representing the box as a **rigid-cube**, which is described by the following frame:

TERMINALS	**ATTACHMENTS**
Dimensions	3d
Flexibility	Rigid
Orientation	Left-Tilted

The system then solves the problem in the following steps:

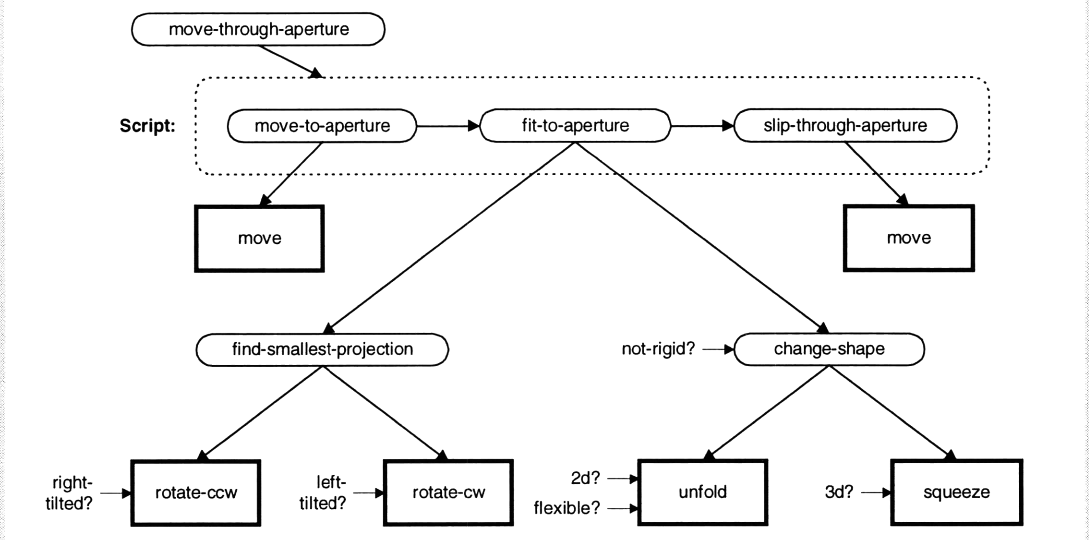
1) First, express the desired goal

The goal is expressed as a desired state of some representation of the world. In this case, the goal might be expressed as the proposition **box-is-on-other-side-of-aperture**.

2) Compute differences

In the first step we expressed the goal, but not the *problem*. The problem is to reduce the

Figure 5.3: A-Brain + "Fit-to-Aperture" Hierarchy



difference between the current situation and the goal. In this case, the problem is to move the cardboard box through the aperture, which matches to a known problem type: **move-object-through-aperture**.

3) Select a method to eliminate the difference

We have activated the problem type **move-object-through-aperture**, which activates the A-brain mechanism shown in Figure 5.3. We will assume that we know only one way to solve this problem, that is, by activating a script that activates the following agents in sequence:

(a) **move-to-aperture**. Move the object in front of the aperture.

(b) **fit-to-aperture**. Make the object's projection against the aperture fit within the aperture.

(c) **slip-through-aperture**. Move the object directly through the aperture.

Steps (a) and (c) are the subproblems of moving the box from one point to another, and are straightforward since there is nothing in the box's way by the time they are activated. Of course, to solve (c), slipping the box through the aperture, we must first solve (b), fitting the box to the aperture. As shown in Figure 5.3, **fit-to-aperture** is a problem type that sits on top of means-end hierarchy of methods for making an object smaller than some aperture. The means-end hierarchy provides two general approaches to solving this "make smaller" problem:

(a) find-smallest-projection: Try re-orienting the object until you find its smallest projection against the aperture.

(b) change-shape: Try changing the shape of the object so that its projection against the aperture is smaller than the aperture.

We have represented the cardboard box using a **rigid-cube** frame, whose **flexibility** terminal has the attachment **rigid** and whose **orientation** terminal has the attachment **left-tilted**. Under these assumptions, traversing the **fit-to-aperture** means-end hierarchy will lead to activating the **find-smallest-projection** agent, and then the **rotate-clockwise** agent. So the first way we try to solve the problem is by rotating the box clockwise until its projection becomes smaller than the aperture.

4) Detect failure

The **rotate-clockwise** agent eventually turns the box so it is no longer tilted, minimizing the size of its projection against the aperture. Unfortunately, as we see in Figure 5.4, this is still too large! We must be able to detect this failure so that we can deal with it. To do this we have scripts associated with the **rotate-clockwise** agent that describe its usual course of action and ways in which it might deviate from that course:

(a) usual-event: The object's projection against the aperture gets smaller until it is smaller than the aperture.

(b) cannot-make-smaller-failure: The object's projection against the aperture reaches its minimum while remaining larger than the aperture.

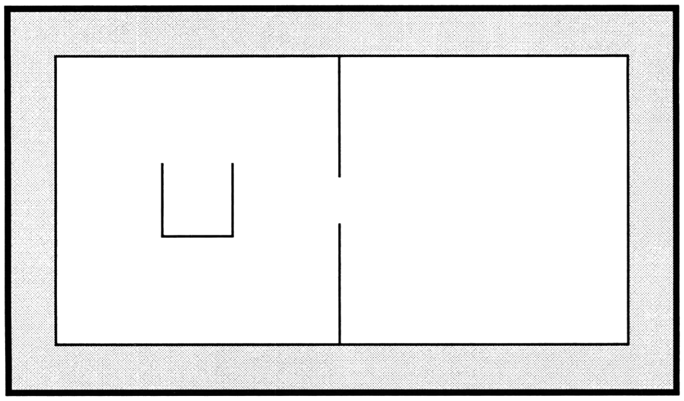


Figure 5.4: The Re-Oriented Box

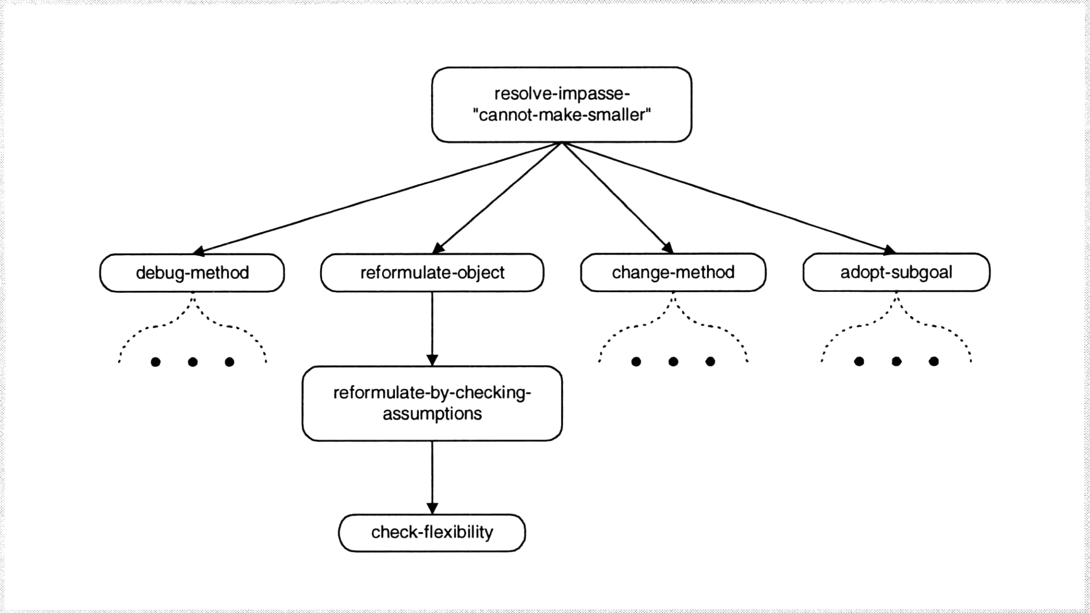
Script (a) describes the usual, successful application of the **rotate-clockwise** method as a means of fitting an object to an aperture. Script (b) describes one way that such an application might fail, described in terms of a *difference from script (a).* In fact, because **cannot-make-smaller-failure** is the only failure type we define, this failure situation is recognized as having taken place if the script **usual-event** hasn't taken place! In our simple program, **usual-event** is not even represented as a script, but instead is regarded as equivalent to the property **getting-smaller?**, which is updated by the microworld simulator after a rotate method is applied (this is the "rotate fails" signal mentioned at the end of section 5.2).

What this amounts to in the context of the example is this: When the **rotate-clockwise** method is applied, the box's projection against the door begins to get smaller. This means that the **getting-smaller?** property is true in the simulator. As we noted above, this means that this is the **usual-event**. When the minimum projection size is reached, the **getting-smaller?** property turns false, which triggers **cannot-make-smaller-failure**. We thus simultaneously detect and classify the failure (since there is only one type of failure here!)

5) Recover from failure by reformulation

What can we do to recover from the failure of our method? When the failure is detected and classified, we activate a B-brain means-end hierarchy to deal with the problem,

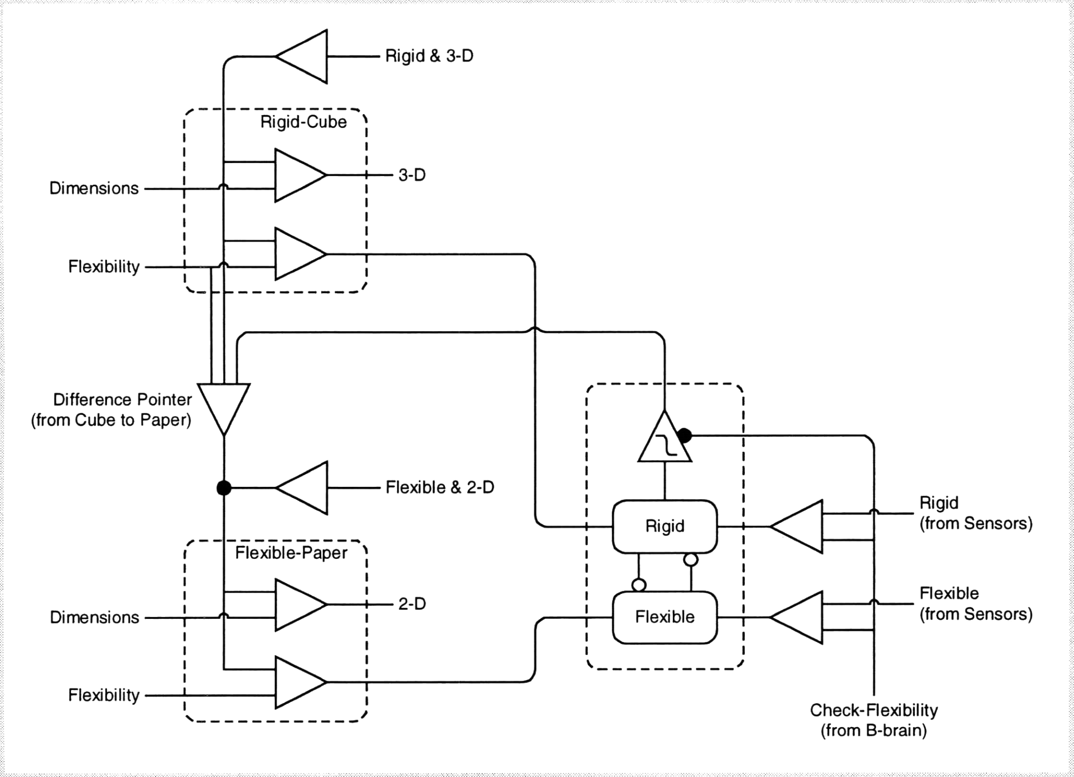
Figure 5.5: B-brain Hierarchy



shown in Figure 5.5. Although we have depicted several possible methods of failure recovery—reformulation, change of method, debugging, and subgoaling—we focus here on the technique of reformulation (which is the only method of the four that our program actually implements.) We therefore do not address the recovery method selection problem here.

The reason the A-brain failed is because it assumed that the box was rigid when it is really was not, and this mistaken assumption is what caused us to get stuck with a **cannot-make-smaller** failure. Therefore the B-brain means-end hierarchy activates **check-flexibility**, which causes the A-brain to verify its rigidity assumption by checking against the sensors. Since our system has no actual sensors, these are hardwired to say that the box is **flexible**, something we might determine in reality by managing to bend the cardboard a little bit. This triggers the following sequence of events, referring to Figure 5.6. We assume that the **rigid-cube** frame is active. When **check-flexibility** runs, it causes the flexibility agency to change state from rigid to flexible.

Figure 5.6: Difference Network of Cube and Paper Frames



This change is detected, and activates a difference-pointer to cause **flexible-paper** to be activated instead, which has the following terminal assignments:

TERMINALS	**ATTACHMENTS**
Dimensions	2d
Flexibility	Flexible

So by checking flexibility we can trigger a change in frame. In general there will be many difference pointers, and we need some way to manage the decision of which new frame to change to, but we do not address that issue here.

6) Select a new method

Now that we have changed representation, we re-run the A-brain means-end hierarchy. This time, we follow a different path down the hierarchy, leading us to consider the

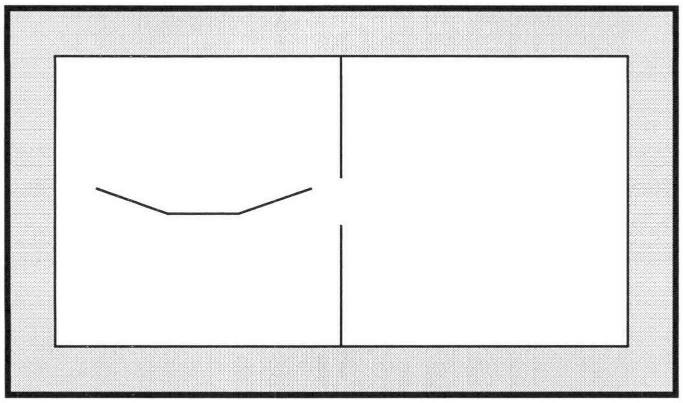


Figure 5.7: Unfolded Box

unfold action. Unfolding the box makes it smaller than the door (see Figure 5.7), allowing us to slip it successfully through the aperture.

Problem solved!

6 Discussion

6.1 Summary

We have argued that using many representations is better than using just one, because then we can always operate in a simple context and can avoid having to consider everything we know at all times. We described a problem solving system capable of applying different representations to a problem at different times. This agent operated using two basic heuristics:

(a) difference-driven application of solution methods given a representation, and

(b) failure-directed transitioning to another representation if the current one becomes ineffective.

We demonstrated these ideas by implementing the system and having it solve a simple test problem.

6.2 Important points we made

In addition to defending our thesis about failure-directed reformulation as an approach to changing representation, we did a number of other things that we believe are of importance:

A. We described a way of organizing how-to knowledge.

Problem solvers are normally thought of as goal-achieving machines. But it is rarely recognized that solutions are best indexed not by the goals they achieve, but rather the important *differences* they make. We described a way of organizing solution methods according to this insight, using a structure we called a *means-end hierarchy*. We proposed that for every important type of problem, there exists a means-end hierarchy that indexes a vast collection of known solutions to that problem.

We see means-end hierarchies as the simplest way to implement *level bands*, the strategy outlined in *The Society of Mind* for relating knowledge about the function of things to knowledge about the structures of those things. In the future we plan a tighter function-structure integration that incorporates causal models of how the function derives from the structure. Then the match-selection of solution methods could be done with more thoughtfulness than simply descending a decision-tree by rote. Also, we will build a better matching system, one that allows for bottom-up constraint on the matching process as well as easy integration with perceptual systems.

B. We made the idea of diversity centrally important

The problem solver we described had several ways to solve both the problem of how to fit a box through a door and the problem of how to get unstuck in dealing with that first problem. The means-end hierarchy idea allowed us to organize these many methods so that they could be efficiently selected between. Generally, we assumed that for every problem we could program several solutions to it. We believe this simple idea has been greatly underappreciated in computer science, where the emphasis has been placed on algorithms, which are by definition a single simple mechanism for solving a problem. But many algorithms are known for most problems, and it is generally the case that no one algorithm is best under all circumstances. Which is better, bubblesort or quicksort? If we want to insert just one item into an already sorted list, bubblesort is faster! In artificial intelligence as well, almost all systems are programmed with essentially one way to deal with problems when they arise—usually by ignoring the problem and simply continuing to produce subgoals. But by using the means-end hierarchy to decide how to recover from impasses, we could program our problem solver with many different ways to recover from failures, such as changing our strategy or reformulating our description of the situation. Thus we offered one solution to the problem of how to integrate many simple mechanisms to produce a flexible and robust society of mechanisms.

C. We built a "societal" AI system

We think of a *societal* AI system as one where the subfunctions are served by goal-directed problem solvers. Present AI systems can be thought of as being made of the simplest agents—usually there is a fixed algorithm for matching, planning, credit assignment, etc. But if those subfunctions were themselves served by AI systems, then the overall system could be much more flexible, powerful, and capable of self-improvement. Our problem solver had two separate goal-directed problem solvers, one for the A-brain and one for the B-brain. There were several difficult subfunctions that need to be served: credit assignment, frame matching, and frame switching. Consider the credit assignment problem of determining what aspect of the society should be blamed if something goes wrong during while applying a solution method. There is no limit to how difficult this problem could become in a large society of agents. It may require playing back the events in mental simulation, which requires careful management of memory resources, or it may involve making sophisticated analogies to find situations where things went wrong in a similar way in the past. There has been much AI research on diagnosis and debugging, and this seems to be a problem domain as difficult as any other.

In general, we believe that the kinds of mechanisms described in here are on the level of a programming language for building higher levels of functionality. To build AI systems we will need components like matching systems and case-based problem solvers—but these form the ingredients for larger structures of which we know very little about! That is, we believe that the kinds of problem solving systems produced so far in AI are not the final product, but only ways to build agents of even bigger systems. And if this is the case, we have an additional justification for the means-end hierarchy as a problem solving method. After all, if we want to build larger problem solvers out of simpler ones, the simpler ones must be very fast—i.e., no expensive searches. If they perform the subfunctions, they must be quick for the overall operation to be quick.

So perhaps means-end hierarchies or something like them will form the functional substrate of the intelligent machines of the future. We believe that designing "functional sketches" of cognitive architectures and finding ways to build them by piecing together smaller goal-directed problem solvers is the next big challenge in artificial intelligence.[8]

D. We provided a justification for using microworlds

There has been a great deal of criticism of microworlds. The general complaint is that microworlds are so simple that ideas developed in them fail to scale up to harder problems: learning algorithms are always presented with the right features, rendering their task obvious; planning systems are given complete models of their operators, and so are never surprised by their actions having unexpected effects; reasoning systems assume that the world contains only a well-defined set of objects, and so do not have to take into account contingencies generated by sources they have not considered; etc. As such, microworlds are no longer trusted as a research tool, and many researchers now entirely avoid using microworlds in their research.

But we believe just the opposite—the simplicity of microworlds is not their weakness, but their strength! We think that, very often, the microworld-level is the scale at which AI ideas *ought* to be developed. The solution to the scaling-up problem is to organize our conceptual microworlds—representations—into societies, so that when problems arise, we can switch to more appropriate and reliable representations and mechanisms. In other words, don't even *try* to solve problems in difficult domains, not directly. Instead, build systems that (a) can solve problems in myriad simpler domains, and (b) can solve the problem of factoring difficult problems into simpler ones that can be solved by analogy in known simple domains. This is one of the central insights of Marvin Minsky's *Society of Mind* theory, and we believe problem-solving systems should be organized following its inspiration, into networks of agents that use a variety of representations. Any particular agent has only a simple viewpoint on the world, but *societies* of agents can solve problems far harder than any individual agent could.

---

[8] Of course, while one could always have every subfunctions served by a sophisticated AI system, this recursion must bottom out somewhere. Certainly at some point agents must get simpler, and less smart, if we want to be able to build these things.

6.3 Future work

To make this discussion happen, we had to defer many issues for later research:

A. Explore activating many representations in parallel

An important topic we did not touch on at all is having many representations operating in *parallel*, where we do not transition between frames, but instead have several frames active at once and we only change their relative priorities. Perhaps it would have been better to solve the box-through-door problem by seeing the box as *both* a geometric solid and a flexible sheet at the same time. We assumed that one representation was active at a time, and so we did not face the difficulties or see the advantages of simultaneously activating many representations. Here are three advantages:

(1) Speed. Problems can be solved faster with N representations than with one—but how much faster? We suspect that in some cases we can actually do much than a factor of N, because if we try not to deactivate representations we do not lose state, and so do not have to reconstruct lines of reasoning or have to repeat negative experiences.

(2) Eases activation. Another advantage of having many representations active simultaneously is that it makes it easier to activate new ones! To use a new representation we must first be able to compute some of the values at its terminals. If more representations are active, the more paths and intermediate results are available for computing those values, and so for using new representations.

(3) Yet another advantage of multiple representations is that any particular one is bound to be broken in some respect, or incomplete in its knowledge. Bringing many different representations to bear on the same problem is a powerful way to build robust problem solvers.

The main disadvantage of having multiple representations running at the same time is the need to share limited resources. We expect that managing large agent systems will require many mechanisms for making sure that representations are available and computed when they are needed, and that goals are scheduled keeping resource allocation issues in mind. Marvin Minsky has suggested that one of the main purposes of human emotions is to play such a resource management role [MIN98].

B. Incorporate anticipation and planning

In any exploration of problem solving, the issue of planning is an important consideration. However, the problem solver we described is essentially reactive on differences and default assumptions. While this would explain the speed of much of human thought despite its slow hardware, it does not explain how people can anticipate problems and plan ahead as they do. Anticipation might be implemented by a simulation system that could look ahead a few steps, perhaps always on the lookout for obstacles or dangerous situations. It is important to try and see failures before they happen.

How can we implement planning in the context of difference-driven control? This might not be difficult, if we note that the direction of processing is determined largely by the way we conceive of the present situation. We believe that many of the traditional heuristics of commonsense planning—abstraction, nonlinearity, etc.—can be treated as special cases of the general reformulation idea. We propose that many of the abilities of modern-day planning systems can be reproduced by the careful selection of and transfer of control between representations:

Hierarchical planning. Most abstract representations can be chosen first.
Nonlinear planning. Representations can be interleaved.
Replanning. Alternative trajectories can be selected.
Means-end planning. Natural behavior of our problem solver.
Change of representation. We described one approach in this thesis.

This allows us to use the difference-directed control strategy on all levels. We do not include here the more constructive forms of planning where we piece together trajectories between imagined situations, but even those might be implemented by applying these sorts of approaches between *imagined* situations—which requires that our system know how to propose good intermediate situations, be able to manipulate those situations in simulation, and be able to keep track of the various pieces of the solution. In general, we put forth the suggestion that casting planning as incremental, directed reformulation is a better way of looking at it than usual "searching a state-space" view of it.

C. Explore B-brain control over aspects of society other than representation

In this system the B-brain selected a new representation on detecting a failure in the A-brain. The other recovery methods—learning, changing method, and subgoaling—were not implemented. In the future we plan to fill out the rest of the failure recovery hierarchy, not only with those three methods but with some of the many more recovery methods that exist, such as adapting an old solution, or trying to better understand the problem domain. As one example of how we might do this, let us briefly consider how we might implement the "changing method" technique. How do we decide which method to try next? Perhaps we might connect the many methods we know in a big difference network that relates them by their many strengths and weaknesses. This would make it far easier to select a *better* method when the current one breaks. For example, perhaps **move-by-sliding** worked before in situations where **pickup-and-move** did not, making it a good candidate to consider next. We might compare methods in various systematic ways: If we try to open a stuck door by using a wooden stick and the stick breaks, try to find a *stronger* stick, like a metal rod. If we run one kind of optimization algorithm and it runs out of memory, we can try another that requires *less memory*, etc.

D. Learning

In this thesis we gave no space to the problems of learning and adaptation. This is because we believe that any such work should be preceded by an understanding of the structure of what is to be learned, and how it can be represented. But now that we have a first theory of how to represent societal problem solving processes, a next step is to consider how to build mechanisms for modifying and growing such societies. This will surely involve many different kinds of learning programs for making changes at difference levels and in different places in the society. For the means-end hierarchy organization alone, we could learn many kinds of things:

- how to best balance the hierarchy for quick decisions
- models of the individual solution methods
- how to incorporate new method into the hierarchy
- problem types and failure types
- associating differences with solution methods

And then there is the possibility of learning new representation. This would require machinery for learning scripts, procedures, causal models, and many other kinds of things. In addition, learning itself is a problem that we ought to be able to apply our full intelligence to, something almost no machine learning systems do today.

E. Implementing in neural networks

We believe this theory will not be difficult to implement in neural networks, circuits of primitive elements like perceptrons and K-lines (which are essentially wires that can activate a set of agents). As we showed in Figure 5.6, frames are easy to implement using such elements. The means-end hierarchy would be simple to implement as well. Selecting a hierarchy is done by computing a difference between the actual and an ideal situation. This can be done by first activating a K-line that represents the ideal goal state, and then quickly activating our representation of the actual situation. If agents are made to be difference-sensitive, this will automatically compute the desired difference, which can then be matched against known means-end hierarchies. Activating the children nodes of a hierarchy can be done using K-lines, and checking if their conditions are valid can be done using simple matching operations, which can be done with networks of perceptrons.

AI workers who study so-called “connectionist” mechanisms like feedforward neural networks often complain that symbolic structures like frames are neurally implausible. They object that the kind of easy manipulation of symbols that frame-based AI systems seem to depend on could not occur in a slowly changing network of neurons. But frames as we implemented them are never "manipulated" in the sense of list structures being arbitrarily consed together and copied. Frames are just activated and deactivated, and attached and detached from one another. Minsky has argued that attachment can be performed without significant changes in computational structure using a mechanism he called frame arrays, which assumes that many connections are already present, and so it's simply a matter of a performing a competitive selection between the attachments. Furthermore, by using bundles of neural fibers as connection buses, we can easily build systems that are capable of symbolically referring to any agent connected to the bus. Even the "slowly changing network" requirement is not a serious restriction. If we do make large changes, such as we do when we produce long-term memories, (a) this might still be just a matter of connecting a few wires and (b) in humans this takes about a half an hour, so slow operations are allowed. Learning a few bits a second may be fast enough if we are selective about those bits!

6.4 Conclusions

In this thesis we explored one way to build a problem solver that employs many representations. There are undoubtedly many other approaches to building such problem solvers, both for ones that use frames and for ones that use the other kinds of representations we listed in the introduction. Nevertheless, we hope that with our simple approach we were able to make clear two points:

1. *Robustness and flexibility comes from having many ways to approach problems.*

2. *Representations should be thought of as tools, and (from point 1) we should be able to apply different ones to any given problem.*

We advocate that AI researchers should commit greater effort to understanding how the particular functions that interest them can be achieved by different computational structures. While this goal runs contrary to the usual one of trying to find the best way to solve a problem, as we have argued it may be impossible to anticipate the best way to solve a *hard* problem. So we must try to understand how the different ways of solving a problem are related, so that we can choose among them as we learn more about the situation.

When those different methods use different representations, we must try to understand the relative strengths and weaknesses between those representations. For even if the representation we are using fails to account for some essential detail of the situation, makes an inaccurate assumption about the world, restricts from consideration a possible method of solution, or imposes some other limitation of viewpoint that in simplifying the problem makes it difficult or impossible to find a good solution, if we can recognize the fault and use it to find a *better* representation then we may still find success.

We believe that AI systems should be designed with these points in mind. In order to make progress on the hard problems of artificial intelligence, we must accept that robustness and flexibility often requires incorporating high levels of diversity into our systems. We must accept that our science is a different kind of science, where we should be biased far less towards looking for simple unified theories, and far more towards making sure we have *enough* theories to explain that diverse variety of phenomena we call "intelligence".

7 References

[DAV90] Ernest Davis. *Representations of Commonsense Knowledge.* Morgan Kaufmann: San Mateo, CA. 1990.

[HAM90] Kristian Hammond. Case-Based Planning: A Framework for Planning from Experience. Cognitive Science, 14(3), pp. 385-443, July 1990.

[HOF95] Douglas Hofstadter. *Fluid Concepts and Creative Analogies: Computer Models of the Fundamental Mechanisms of Thought.* Basic Books: New York, NY. 1995.

[LEN90] Douglas Lenat and R. V. Guha. *Building Large Knowledge-Based Systems.* Addison-Wesley: Reading, MA. 1990.

[MIN75] Marvin Minsky. A Framework for Representing Knowledge. The Psychology of Computer Vision, ed. Patrick Henry Winston. McGraw Hill, 1975.

[MIN85] Marvin Minsky. *The Society of Mind.* Simon and Schuster: New York, NY. 1985.

[MIN90] Marvin Minsky. "Logical vs. Analogical or Symbolic vs. Connectionist or Neat vs. Scruffy", in Artificial Intelligence at MIT., Expanding Frontiers, Patrick H. Winston (Ed.), Vol 1, MIT Press, 1990. Reprinted in AI Magazine, 1991.

[MIN98] Marvin Minsky. *The Emotion Machine.* Forthcoming.

[NEW59] Alan Newell, J. C. Shaw and Herbert Simon. Report on a general problem-solving program for a computer. Information processing: proc. intl. conf. on information processing (IFIP 60), pp. 256-264, 1959.

[NEW60] Alan Newell, J. C. Shaw and Herbert Simon. A Variety of Intelligent Learning in a GPS. "Self-Organizing Systems", M.T. Yovitts and S. Cameron, Eds. Pergamon Press, New York: NY. 1960.

[RAM94] Ashwin Ram and Michael Cox. Introspective reasoning using meta-explanations for multistrategy learning. In R. Michalski & G. Tecuci (Eds.), *Machine learning: A multistrategy approach IV* (pp. 349-377). Morgan Kaufmann: San Mateo, CA. 1994.

[SCH82] Roger Schank. *Dynamic memory: A theory of reminding and learning in computers and people.* Cambridge Univ. Press: Cambridge, MA. 1982.

[SUS75] Gerald Sussman. *A computer model of skill acquisition.* New York: American Elsevier. 1975.

[WIN70] Patrick Winston. *Learning structural descriptions from examples.* MIT Department of Electrical Engineering Thesis. Ph.D. 1970.