Abstract

AI problem solvers have almost always been given a complete and correct axiomatization of their problem domain and of the operators available to change it. Here I discuss a paradigm for problem solving in which the problem solver initially is given only a list of available operators, with no indication as to the structure of the world or the behavior of the operators. Thus, to begin with it is "blind" and can only stagger about in the world tripping over things until it begins to understand what is going on. Eventually it will learn enough to solve problems in the world as well as if it the world had been explained to it initially. I call this paradigm naive problem solving. The difficulty of adequately formalizing all but the most constrained domains makes naive problem solving desirable.

I have implemented a naive problem solver that learns to stack blocks and to use an elevator. It learns by finding instances of "naive mathematical cliches" which are common mental models that are likely to be useful in any domain.

Keywords: learning; naive problem solving; naive mathematics; problem solving; planning; problem solving; semantic cliches.

A.I. Laboratory Working Papers are produced for internal circulation, and may contain information that is, for example, too preliminary or too detailed for formal publication. It is not intended that they should be considered papers to which reference can be made in the literature.
# Table of Contents

1. Naive Problem Solving .................................................. 1  
   1.1 Examples. .............................................................. 1  
   1.2 Interfaces between the world, the learner, and the planner. ....... 8  

2. The Learner ................................................................. 10  
   2.1 Semantic cliches ...................................................... 10  
   2.2 Finders for semantic cliches. ...................................... 10  
   2.3 Derived relations .................................................... 12  
   2.4 Learner implementation issues .................................... 13  

3. Naive Mathematics ....................................................... 14  
   3.1 Order and equivalence relations. .................................. 14  
   3.2 Functions ............................................................... 15  
   3.3 Stacks and queues .................................................... 16  
   3.4 Other naive mathematical concepts ................................ 17  

4. The Planner ................................................................. 18  

5. Future Work ............................................................... 20
1. Naive Problem Solving

AI problem solvers have almost always been given a complete and correct axiomatization of their problem domain and of the operators available to change it. Here I discuss a paradigm for problem solving in which the problem solver initially is given only a list of available operators, with no indication as to the structure of the world or the behavior of the operators. Thus, to begin with it is "blind" and can only stagger about in the world tripping over things until it begins to understand what is going on. Eventually it will learn enough to solve problems in the world as well as if it the world had been explained to it initially. I call this paradigm naive problem solving. The difficulty of adequately formalizing all but the most constrained domains makes naive problem solving desirable.

A naive problem solver consists of (at least) three parts: a learner, a planner, and a goal manager. The learner's job is to produce theories about the structure of the world and the behavior of the available operators by observing the world and the effects of actions on it. The planner's job is, given a goal, to construct a sequence of actions (applications of operators) that will bring about that goal, and then to perform those actions. If the planner does not know enough to produce such a sequence, it can ask the learner to find out more about the relevant parts of the world. The learner may propose experiments to test its theories; these experiments are in turn goals for planner to achieve. The goal manager chooses among goals produced by the learner and perhaps other parts of the mind. It is what we might call the "affective" or "deliberative" component of the problem solver.

This paper reports on a preliminary demonstrative implementation of a naive problem solver which learns to stack blocks. It also presents speculations about and plans for a more complete problem solver. The remainder of this chapter is spent on two example and a discussion of the overall modularity of a naive problem solver. The second chapter is devoted to the general learning strategies used, and the third to the a priori knowledge the learner is given. The fourth chapter explains the planning component of the problem solver, and the last discusses future extensions.

1.1 Examples

I will begin with two examples. These two examples are the only ones which the implementation has been shown to run. The examples are simplified here; they are given again in full detail in an appendix at the end of this paper.

The first example is learning to operate a very simple model of an elevator. Initially, a human "teacher" creates a new world of flavor up-down-world.

Human: (new-world! up-down-world)

This world informs the learner of some things that are true in it. An up-down-world has in it some "floors" which are represented by numbers. Some floors are below others.
The learner now has seen enough data that it wants to ask some questions.

Learner: Is (BELOW THREE THREE) true?
World: No.
Learner: Is (BELOW ONE ONE) true?
World: No.
Learner: Is (BELOW FOUR FOUR) true?
World: No.
Learner: Is (BELOW TWO TWO) true?
World: No.

Based on the available data, it appears that below is an antireflexive relation; that is, that for no \( x \) is it the case that (below \( x \ x \)).

Learner: BELOW seems to be antireflexive
Learner: Is (BELOW FOUR THREE) true?
World: No.
Learner: Is (BELOW THREE TWO) true?
World: No.
Learner: Is (BELOW TWO ONE) true?
World: No.

Below also seems to be antisymmetric; that is, for any \( x \) and \( y \), both (below \( x \ y \)) and (below \( y \ x \)) are never true.

Learner: Making BELOW antisymmetric
Learner: Is (BELOW ONE FOUR) true?
World: Yes.
Learner: Is (BELOW TWO FOUR) true?
World: Yes.
Learner: Is (BELOW ONE THREE) true?
World: Yes.

Based on these observations, it seems that below is also transitive.

Learner: BELOW seems to be a transitive relation.

These three conclusions together make below a partial order.
Learner: Below is an antireflexive partial order relation.
World: (Below Four Five)
World: (Below Five Six)
World: (Below Six Seven)
World: (Below Seven Eight)
World: (Below Eight Nine)
World: (Below Nine Ten)

This is all that the world tells the learner about floors. There is one additional fact that it supplies:

World: (On-Floor Robot Six)

The interpretation of this fact is illustrated by the following diagram:

The human teacher now supplies the problem solver with a goal: to get the robot to floor one.

Human: (goal (on-floor robot one))

First, the planner asks if goal is achieved vacuously (is already true):
Planner: Is (ON-FLOOR ROBOT ONE) true?  
World: No.

This being the case, it would be good to produce a plan for the goal. Unfortunately, all the planner knows about the operators of this world is that there are two, called up and down, and that neither takes any arguments. Since no plan can be produced, the problem solver will have to poke at the world with the operators randomly in hope that enough will be learned thereby to later produce a plan.

Planner: I don't know how to achieve (ON-FLOOR ROBOT ONE); will frobnicate...
Frobnicator: action (DOWN)
World: (NOT (ON-FLOOR ROBOT SIX))
World: (ON-FLOOR ROBOT FIVE)

The planner keeps checking to see if the goal has been achieved accidentally.

Planner: Is (ON-FLOOR ROBOT ONE) true?  
World: No.  
Frobnicator: action (UP)
World: (NOT (ON-FLOOR ROBOT FIVE))
World: (ON-FLOOR ROBOT SIX)
Planner: Is (ON-FLOOR ROBOT ONE) true?  
World: No.  
Frobnicator: action (DOWN)
World: (NOT (ON-FLOOR ROBOT SIX))
World: (ON-FLOOR ROBOT FIVE)
Planner: Is (ON-FLOOR ROBOT ONE) true?  
World: No.  
Frobnicator: action (UP)
World: (NOT (ON-FLOOR ROBOT FIVE))
World: (ON-FLOOR ROBOT SIX)
Planner: Is (ON-FLOOR ROBOT ONE) true?  
World: No.  
Frobnicator: action (DOWN)
World: (NOT (ON-FLOOR ROBOT SIX))
World: (ON-FLOOR ROBOT FIVE)
Planner: Is (ON-FLOOR ROBOT ONE) true?  
World: No.  
Frobnicator: action (DOWN)
World: (NOT (ON-FLOOR ROBOT FIVE))
World: (ON-FLOOR ROBOT FOUR)
Planner: Is (ON-FLOOR ROBOT ONE) true?  
World: No.  
Frobnicator: action (UP)
World: (NOT (ON-FLOOR ROBOT FOUR))
World: (ON-FLOOR ROBOT FIVE)
The learner has now seen enough that it is willing to start guessing about the properties of the operators. It appears that up and down both just change the second argument to on-floor, and more over that up specifically increases the value of the argument in the below ordering and that down decreases it.

Learner: DOWN is a decrementing operator with respect to BELOW.
Learner: UP is an incrementing operator with respect to BELOW.

This is enough information for the planner to produce a plan to achieve the goal.

Planner: plan constructed.

The planner now executes the plan.

Planner: action (DOWN)
World: (NOT (ON-FLOOR ROBOT FIVE))
World: (ON-FLOOR ROBOT FOUR)
Planner: action (DOWN)
World: (NOT (ON-FLOOR ROBOT FOUR))
World: (ON-FLOOR ROBOT THREE)
Planner: action (DOWN)
World: (NOT (ON-FLOOR ROBOT THREE))
World: (ON-FLOOR ROBOT TWO)
Planner: action (DOWN)
World: (NOT (ON-FLOOR ROBOT TWO))
World: (ON-FLOOR ROBOT ONE)

The plan worked.

Planner: plan succeeded.

The second example is one of stacking blocks in the very simple world in which only one block can be on top of another. We begin by creating a puton-world, which has blocks in this initial configuration:
In this example, rather than immediately giving the problem solver a goal, the teacher takes some actions in the world himself. The learner "looks over the teacher's shoulder" observing the effects of these actions. I am doing this because in the blocks world most applications of puton are errors. The learner can deal with such errors perfectly well, but it makes so many that the example would be much longer. Also, this shows that the problem solver can learn by being shown as well as by experiment.

The learner is now in a position to guess about what puton is doing. It notices first that puton is essentially doing a stack pop: that is, that it removes the top element in an on chain; and also that it is a stack push: it adds a new element to an on chain. What is odd is that it is not either a simple push or pop; it is doing both things. These facts are duly noted.

The learner also notices that there is a dependency between the arguments to puton and the specific elements pushed and popped.
Learner: The first argument to PUTON is the item pushed in ON.
Learner: The second argument to PUTON is the end of the ON stack it pushes.
Learner: The first argument to PUTON is the end of the ON stack it pops.
Learner: The first argument to PUTON is the item popped in ON.

Upon further reflection, the mystery of the pushness and popness of puton is revealed: it is simply a composition of the two.

Learner: PUTON is a composition of a POP and a PUSH when acting on ON.

The human now sets up the classic "anomalous Sussman situation" from Sussman's PhD thesis [Sussman PhD]:

Human: (apply puton b x)
World: (NOT (ON B A))
World: (ON B X)

Human: (apply puton c a)
World: (NOT (ON C Z))
World: (ON C A)

And the Sussman anomaly problem is posed:

Human: (goals (on a b) (on b c))
Learner: Is (ON A B) true?
World: No.

Initial State

<table>
<thead>
<tr>
<th>C</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Z</td>
</tr>
</tbody>
</table>

TABLE

Goal State

<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
</tr>
<tr>
<td>C</td>
</tr>
</tbody>
</table>
8 Naive Problem Solving and Naive Mathematics

and solved:

Planner: Plan constructed.
Planner: action (PUTON C Z)
World: (NOT (ON C A))
World: (ON C Z)
Planner: action (PUTON B C)
World: (NOT (ON B X))
World: (ON B C)
Planner: action (PUTON A B)
World: (NOT (ON A TABLE))
World: (ON A B)
Learner: Is (ON A B) true?
World: Yes.
Learner: Is (ON B C) true?
World: Yes.
Planner: plan succeeded.

1.2 Interfaces between the world, the learner, and the planner

My model of perception is unrealistic. "Sense data" provided to the learner by the world are of the form

(relation . objects)

and

(not (relation . objects))

Both the relation and all the objects are lisp symbols. There are no compound objects or "function symbols". Relations must be of fixed arity. The relation of one proposition may be used as an object in another proposition. No symbol (except not) is given any a priori semantics.
Each time the problem solver takes some action, the learner is informed again by the world of some part of its state. Also, an application of an operator may "barf" — the arguments may be out of range or the operator may otherwise be ill-defined in the situation. The learner may draw tentative conclusions about the nature of the world and of operators from the changes it observes. In order to test these conclusions, it will propose experiments as goals to the goal manager. Experiments in general might involve a predicate on the world that must hold before the experiment can be run correctly and a set of side-effecting operators that should then be applied to the world when it is believed to be in that state. However, in the current implementation the only experiments are applications of the pseudo-operator ask, which asks the world if some proposition holds. ask is guaranteed to be side-effect free, but not to give an answer: the world can reply "I won't tell you". Both barfing and asking are artifacts of my non-theory of perception.

The goal manager maintains a list of goals that could be worked on. At every point in time, it chooses one goal to work on and passes this to the planner. Experiments are goals; other goals may come from elsewhere. In the current implementation, there are no experiments other than ask experiments, so there is no need for a goal-manager, and none has been implemented. This will have to change when experiments with side-effects or set-up are introduced.
2. The Learner

2.1 Semantic cliches

In the long run, it may be practical for a learner useful in a naive-problem solver to avoid preconceptions about the world entirely. Such a learner would use some powerful, very general mechanisms to understand the world. The old Cybernetic paradigm tried to build such systems. Gary Drescher's more recent genetic theory [Drescher SGAI] and Patrick Winston's analogy-based learning systems [Winston Analogy] look more promising. However, I believe that it is easier now to build learners that believe that any part of the world will fall into certain broad stereotypical patterns, which I call semantic cliches. These are very basic concepts that can be used in almost any domain. Examples of semantic cliches are transitivity, metrics, successive approximations, containment, attachment, enablement, paths, and ownership.

A general difficulty with reasoning systems is that they necessarily operate by "syntactically rubbing formulae together"; they operate "in the theory rather than in the model." Semantic cliches can be thought of a set of well-understood mental models with precompiled competence attached. If some theory can be found or forced to accept a semantic cliche as a model, then the syntactic work of formula rubbing can be short-circuited. Semantic cliches provide competence that, being efficient but not always applicable or very useful, is intermediate between weak methods such as analogy or theorem proving, which are inefficient but always applicable, and domain-specific methods, which are powerful but not general.

The idea of semantic cliches is not a radical departure. Piaget's program of Genetic Epistemology [Piaget Genetic Epistemology] is quite similar, as is the Contextual Dependency idea of Schank [Schank Semantics]. Minsky and Papert use the word "c-germ" for a closely related idea [Minsky Learning Meaning]. My use of the word "cliche" derives from its use by the Programmer's Apprentice group [Rich Programmer's Apprentice] to refer to powerful general ideas used in understanding programs.

2.2 Finders for semantic cliches

The learner I have implemented derives its power from its preconceptions about semantic cliches; it has no very new mechanisms for learning things. Each semantic cliche has associated with it one or more finders, which are agents that try to find examples of their cliche. Whenever the world supplies a "sense datum" to the learner, it is recorded in the world, model (described in a later section) for that moment in time. Then finders for applicable semantic cliches are applied to the datum. Because the finders are mostly independent and potentially numerous, it is best to imagine them operating in parallel: a finder is an autonomous agent that observes data and determines whether or not the data fit the semantic cliche for which it is responsible. However, this parallelism is not complete; not every finder is applied to every sense-datum. Finders are arranged in a frame-system, as in the Minsky Frames paper [Minsky Frames]. A frame-system is a network of recognizers that are activated by their neighbors. When a sense datum for a never-before-seen relation is encountered, a set of initial finders is applied to it. Depending on their findings, they selectively activate other finders that may be relevant. One finder may activate a second finder either because the first succeeded and the second is looking for a structure that is a
specialization of the first, or because the first failed, but noticed in so doing that the second was likely to succeed. The implemented frame system will be illustrated in a later section.

Relation finders provide a static understanding of a situation; operator finders provide understanding for the dynamic behavior of operators. Their implementation is similar: whenever an operator is applied, operator finders are informed of the application and the situation the world reported as resulting.

The operation of most finders is very simple. For example, there is a finder that looks to see whether a relation is symmetric or not. This finder simply looks at all instances of the relation to see if there are any examples of both \((r \times y)\) and \((\text{not } (r \times y))\). If there are none, and an arbitrary minimum number of data have been observed, it tentatively hypothesizes that the relation is symmetric. (This is called “indexing” the relation to the symmetry cliche.) The finder also suggests as experiments to determine for each known \((r \times y)\) whether \((\text{not } (r \times y))\) holds. It further sets up a new finder for the relation \(r\), the symmetry antagonist, whose job it will be to look for disconfirming evidence. This antagonist will for any new datum \((r \times y)\) observed, suggest as an experiment to determine the truth value of \((r \times y)\); and mark as such observed exceptions to the symmetry of \(r\). Antagonists are linked together in such a way that the frame system forms a sort of TMS. Part of the reason that finders are willing to guess on the basis of very little data is that wrong guesses and their consequences will be effectively withdrawn or invalidated by their antagonists.

Some finders are tolerant of exceptions. For example, an operator finder might tolerate “no-op” applications in which the operator simply didn’t do anything. It seems that this in particular is an exception that somehow “doesn’t matter much”; so much so that if frobbing a computer in some way fails to produce any result, one’s first impulse is to try to do the same thing again. More generally, operator finders may look to be sure that some expected result does always occur, but may be tolerant of additional side-effects. Such tolerance can be very important in planning. Suppose you have a chain you want broken. You have a stick of dynamite. If you are sufficiently desperate, it is useful to know that the dynamite will break the chain, though it is a good bet that it will have other effects. This example points out, though, that it is worth trying to produce a theory of what the exceptional side-effects are likely to be. This is handled by exception theories. If an indexing of a theory has exceptions, a parasitic exception theory is created that inherits from the original theory, but through a filter that admits only the marked exceptions. (I use the word “theory” to mean simply a unit or module of representation.) This exception theory is treated just like any other operator, relation, or object theory; finders are applied to it in the normal way, and so it can in turn be indexed. This process can be applied recursively.

Although I am planning to deal with noise eventually, no provision is currently made for it in the design of finders. Also, the learner implicitly assumes that there is “no energy in the world”: that is, that all changes are immediate and direct results of actions that it has taken. Thus, for example, it will be confused by a ball continuing to roll down a slope after an operator has been applied to push it over the edge. I hope to address this limitation eventually also.
2.3 Derived relations

Some finders take actions other than just indexing. For example, there is a finder for reflexiveness of relations. Knowing that something is reflexive is not often in itself interesting or useful, but together with symmetry and transitivity it gives you an equivalence relation, which is often useful. Suppose that a relation is transitive and symmetric, but not reflexive. When the relation has been indexed as transitive and symmetric, the reflexiveness finder will notice that all the properties but reflexiveness hold. This is sad; a trivial bug in the relation prevents us from using it. So the finder creates a new theory of a derived relation, the reflexive closure of the original observable relation. This closure can be used for reasoning just as could the original relation. When a useful fact about the closure is discovered, this knowledge can generally be carried back to the original relation with minimal patching. (The current implementation creates such closures, but nothing currently reasons about them.)

In attempting to retrieve underlying semantics from syntax, there is a difficulty best explained by example: consider the assertions

\[
\begin{align*}
(green \ broccoli) \\
(broccoli-color \ green) \\
(color-of \ broccoli \ green) \\
(has \ color \ broccoli \ green)
\end{align*}
\]

All these may mean the same thing, and in context it ought to be possible to treat all the same way. This can be done by “currying” relations. A relation like color-of can be curried “down” at green to get the green relation, or at broccoli to get broccoli-color; or curried “up” to get (has color ...). All these derived relations can be reasoned about just as though they were presented by the world, and so which is actually observable is not as important as it would otherwise be.

Operator finders can spin off derived theories also. Temporal abstraction, a powerful technique derived from the Programmer’s Apprentice [Waters PhD], creates a derived function theory that is an abstraction of an operator theory. Temporal abstraction thus allows one to reason about some operators, which are formally functions from situations to situations, as though they were functions from objects to objects. This is a great improvement, partly because it halves the number of finders needed, but also because situations are large, complicated structures that are hard to reason about, whereas observable objects are simply atomic symbols.

It ought to be possible to temporally abstract any well-understood operator. There are some fairly complex issues involved though; for example, the abstraction of a push operator is a function from stacks to stacks; stacks are derived, not observable objects, and I do not yet explicitly represent derived (composite) objects. Thus I have only implemented a very simple scheme. This scheme temporally abstracts only setf operators. A setf operator is one that changes the state of a cell where a cell is a relation that holds of only one object at a time. For example, the up operator from the elevator example sets the cell (on-floor robot *) (this cell being a curry of the on-floor relation). In this case, one object is mapped to another by the temporal abstraction of the operator if the first is the state of the cell before an application of the operator, and the second is the state of the cell immediately afterward. Temporal abstraction is used in the elevator example to understand up and down. The extended example in the appendix shows this clearly.
2.4 Learner implementation issues

The learner maintains five basic types of datastructure: operator theories, and diachronic and synchronic object and relation theories. An operator theory contains a list of all applications of the represented operator: the arguments that it was applied to and the changes that were observed in the world as a result. An object theory contains facts about an object; a relation theory, facts about a relation. A diachronic object or relation theory contains facts about the object or relation that hold throughout time — for example that b1 is a block or that the transitive closure of on is a partial order; a synchronic theory for a given situation contains facts about an object or relation that hold in that situation — for example, that at t0, b1 is an extremal element in the on relation.

Currently, each finder amounts to one to two pages of Lisp code. This is annoying; it takes several hours to implement each. Since the behavior of all finders is fairly similar, I hope presently to use some sort of data-directed declarative or macro scheme to simplify the writing of finders.

My learner was initially built using an implementation of Doyle's SDL [Doyle PhD]. SDL, although powerful in expressibility, provides no means for accessing representations. To get information out of it, I ended up using an interesting generalization of virtual copy conceived by Phil Agre and myself [Agre Inclusion] and an exponentially slow theorem prover. This latter was much too inefficient to last. My implementation of SDL, together with papers on RLL [Greiner RLL], inspired a new, simple, modularized, classical pattern-matched database with procedural attachment, which has thus far been entirely adequate for my purposes.
3. Naive Mathematics

I have implemented some finders and plans for a system of related semantic cliches which I call collectively \textit{naive mathematics}, by analogy to "naive physics". Naive mathematics is an \textit{irrational reconstruction} of parts of mathematics. It is that of which mathematics is the formal sublimation. Mathematical semantic cliches are the central, core ones from which, perhaps, others are built. Most or all of the ideas of naive mathematics that I present will be familiar to you; what I have done is to develop ways that they can be used in common-sense reasoning.

Naive mathematics is different from formal mathematics in being rigorous but not formal. In its content, it differs mainly in its treatment of change. Formal mathematics models change almost exclusively with ideas from analysis, whereas naive mathematics, like the theory of computation and some logics, uses ideas drawn principally from algebra. The parts of naive mathematics I have explored are in fact principally concerned with finite algebraic structures.

The remainder of this chapter will sketch the parts of naive mathematics that I understand best; in particular those cliches for which I have implemented finders. Unfortunately, for all but a few of these cliches, I have only a finder, and no reasoning procedures; and so you must take on faith that they are worth looking for.

3.1 Order and equivalence relations

The implemented frame system for order and equivalence relations

Order and equivalence relations are dual to each other in a peculiar sort of way. Rigorously, (partial) order relations are antisymmetric, transitive, and either reflexive or antireflexive; equivalence relations are symmetric, transitive, and reflexive. In neither case is the reflexiveness really essential; and transitivity is expensive to check for. So the finders that are applied by default are those for symmetry and antisymmetry. If a relation is found to be symmetric or antisymmetric, the finder for transitivity is enabled. (Transitivity in isolation is not very useful; all sorts of "pathological" relations are transitive.) If a relation is found to be transitive, then the finders for reflexiveness and antireflexiveness are enabled. These finders will, if they fail, create the reflexive closure and antireflexive restriction of the relation. These derived relations are useful because they have essentially the same properties as the original relations, but are better behaved formally.
After all this, a relation may have been determined to be an equivalence or order relation. Equivalence relations are "the same thing as" (in a sense made precise by McAllester's symmetric set theory [McAllester SST]) partitions. Thus, in a relation found to be an equivalence, blocks of the induced partition can be reified. (This has not been implemented.)

There are a number of useful specializations of partial order relations. First, there are total orders. The finder for total orders is enabled by the success of the finder for partial orders, and checks trichotomy directly. Some orders are dense or everywhere discrete; there are finders for both of these. It is often useful to try to make any order behave as though it were dense. A finder exists that notes all extreme elements in orders.

### 3.2 Functions

There is a finder that is always enabled that checks whether a relation is a function. Functions may be one-to-one; there is a finder for this. Several interesting specializations of one-to-one functions are also looked for. A one-to-one function which has a single connected component when considered as a graph I call a **chain**. Chains are further specialized as **cycles** and **threads**. A cycle is a chain on a finite set of elements that is a total function; this can happen only when \( \text{cycle}(x) = x \); \( f \) is the cycle and \( n \) is the cardinality of the domain. A thread is a finite non-cyclic chain; the temporal abstractions of up and down are both threads, as the learner discovers in the elevator example in the appendix. The transitive closure of a non-cyclic chain is an order relation, and all everywhere-discrete order relation has an underlying non-cyclic chain. Much reasoning can be transferred between chains and orders thus.

Hoffstadter [Hoffstadter GEB] describes a phenomenon called a **strange loop**. This is the collapse of an order relation into a cycle. The order relation antagonist notes such strange loops.

There is a finder for monotonic increasing and decreasing functions. Of course, for it to be applicable, there must be known orders on the domain and range of the function. **Up** and **down** are both monotonic increasing operators, as the learner observes in the appendix.

There are finders for increment and decrement functions (so that \( f(x) \) is always greater than (or less than) or equal to \( x \). These are used in the elevator example: they index the temporal abstractions of up and down.
Composition of functions can be used in two ways: the problem solver ought to define for itself composite derived ("macro") operators that have useful properties; and it ought to be able to recognize a provided operator as a composition of operations that are cliches. Only the latter has been implemented. The learner finds compositions of two cliches by looking at an operator or function (say puton) and seeing if it has two indexings (say to push and pop) each of which has exceptions, but such that all the exceptions to each is fully explained by the other — in other words, that the exception theory for push is indexed as pop, and vice versa, and that the exception theories do not have recursive exceptions.

3.3 Stacks and queues

There are finders for the push and pop operations. Since you can push or pop either end of a thread, these can be used to understand both stacks (which have a push and pop at the same end) and queues (for which the pop is at the opposite end from the push). These finders are used in the blocks world example to understand puton.

When it is determined that an operator is a push or a pop, finders are activated for the two arguments to those operators. The finder for the pushed or popped element argument simply looks at each argument to the operator and sees if it is consistently the case that that element is the object pushed or popped. Finding the stack argument is a little more subtle: stacks are derived objects, so an operator can not refer to one by name. Various reference schemes are imaginable; in the case of puton, the stacks are referred to by a metonymy, that is, a metaphor in which a (typically distinguished) part of something is used to refer to the whole. For puton, the stack popped is that whose top is the popped item, and the stack pushed is that whose top is the block on which the other is put.

In general, inputs to an operator can be passed by side-effect as well as by argument. For example, consider a blocks world with operators pickup and putdown, in which pickup asserts holding of a block and putdown negates the holding assertion. Here putdown is passed its pushed-element input through the cell holding. I have a scheme for dealing with this that is not yet implemented.

It is reasonable to ask whether pushing and popping can be considered naive mathematical structures. In general, in this paper I will not consider the question of what things ought to be considered semantic cliches; the answer must be tied up with the question of where cliches come from. However, if it is thought unreasonable that push and pop are naive, we can achieve the same effect by the use of more general (and so more likely cliched) structures: set addition and deletion. Push and pop respectively add and delete elements from a set (the thread). They do so at distinguished points: the ends of the thread. It is plausible that very general finders for set addition and deletion could be written that would look especially for their effects at points that are distinguished with respect to other structures on the set. So for example, we need only specify that the ends of threads are "interesting" (which they are for a number of other purposes); then effectively pushes and pops can be found by looking for set adds and deletes at those points.
3.4 Other naive mathematical concepts

To give some feel for the scope of (the algebraic part of) naive mathematics, I will give here an enumeration of some vocabulary items that should eventually have finders, plans, and other reasoning procedures developed for them. None of these things have been implemented.

Naive mathematics seeks to reduce analytic and topological notions to (preferably finite) algebraic ones. Algebraic structures are generally easier to compute with. Most uses of analytic notions in common sense depend only on certain properties of the real numbers. Many uses are only for the order properties. Some depend on density. And many operators on orders obey some of the field axioms, for example. Ideas from topology will probably be be useful in understanding space.

Binary oppositions, discrete orderings, and continuua (dense orderings viewed analytically) are all interconvertible: there are cliched ways of reformulating arguments from one of these views on something to another. This is related to Forbus' qprop idea [Forbus QP] which reduces real-valued functions to finite algebraic structures.

Restrictions and extensions of relations. Representations of relations as binary functions and as set-valued functions.

Connected components of a graph.

Trees and hierarchies.

Lattices. Intervals.

Onto and bijective functions. Morphisms and congruences.

Identity elements; inverse elements; fixed-points. Nilpotent, idempotent, associative, and commutative functions; traditional algebraic structures: groups, semigroups, rings, and so forth.

Product spaces.

Arithmetic.

Sets and multisets.
4. The Planner

The planner is the least interesting part of the problem solver. It exists mainly to demonstrate that the learner is doing something useful (supplying it with theories about the world and about operators) and to allow the learner to do experiments. Because I am not principally interested in planning, but rather in learning, I decided to use off-the-shelf planning technology for the naive problem solver. The state of the art in planning seems to be Sacerdoti’s NOAH [Sacerdoti PhD Book], and so I implemented something very similar to that. There are some minor extensions that had to be made; there are also features of NOAH not needed to solve simple blocks-world problems that I haven’t implemented.

The planner is able to deal only with goals that are conjunctions of observable relation tuples. To plan for such a goal, it first partitions the the tuples by relation, and plans for each relation separately. For each relation, the planner looks to see what operators are known to side-effect that relation (for example, puton modifies on). For each such operator, the planner looks in the operator theory to see what cliches the operator has been indexed to by the learner. For each such indexing, the planner looks in the theory of the cliche to see what plans can be used for that cliche. One of these plans must be selected to achieve the goals involving the relation. This can be a complex problem which ought to be handled by a meta-level deliberative problem solver, as in [Doyle PhD], but currently simply the first applicable plan retrieved is used. These plans themselves, as in NOAH, are represented as Lisp procedures which build networks that represent partially ordered sets of actions. The learner also supplies indexings on operator theories that explain the use of arguments to the planner.

Currently the only implemented plans are the two used in the examples in this paper. The first plan, used in the elevator example, uses an incrementing operator and a decrementing operator to set the state of a cell whose possible contents are related by an order (below in this case). The second plan, used in the blocks world example, uses a push and a pop to build an arbitrary thread. An interesting feature of planning in the blocks world is that puton serves as both a push and a pop. Since the planner is good at dealing with unwanted side-effects, this is not a problem.

When the problem solver is given a goal for which the planner fails to produce a plan, it instead frobnicates. Frobnication is application of operators to the world not to directly achieve a specific provided goal, but rather in order to learn enough about some part of the world that a goal can in fact be planned for. Highly complicated and specific strategies for frobnication can be imagined. It is important, for example, to only frobnicate with parts of world that are likely to be relevant to achieving the goal at hand; various more or less complex forms of locality can be supposed to be taken into account in determining a frobnication strategy. Again, it is useless to frobnicate something you already understand. And so forth. However, the implemented frobnication strategy is very simple: to learn to achieve a goal involving the relation \( r \), apply operators selected at random to objects selected at random from those known to participate in \( r \).

The frobnicator is given a complete list of the operators available in the world and their arities. This may be unrealistic; it is not clear that human infants know what operators they may apply to the world. Indeed, biofeedback experiments in which adults learn to control their blood pressure or skin temperature suggest that most people never learn the complete set of “primitive”
operators available.
5. Future Work

The learner as currently implemented has innumerable deficiencies and incompletenesses. These need to be fixed. In addition, I have plans for several major extensions. Several of these are concerned with side-effects, hidden state variables, and debugging.

Common sense reasoning is based on many shallow, overlapping, incomplete, partially correct theories of the world. Most AI projects have attempted to give computers deep, correct models of their domains. But it is only expert knowledge that has this form. The understanding of even the simple blocks world demonstrated by the problem solver in the puton world example is very incomplete. For example, it will fail if you give it a goal of the form (on x table) because table is not the top of a stack. I believe that the reason that the blocks world has been traditionally hard to understand is because it is just complex. It is easy to give a complete, correct axiomatization of it, but that does not amount to an understanding. For example, consider that putting something on the table amounts to making a new stack. This is not predicted by anyone's blocks world axioms, yet is vital in blocks world planning. Perhaps twenty such odd facts about the blocks world are needed to fully understand it. I hope to give a full account of that understanding in a future publication.
Appendix: The Examples in Full

These are very nearly verbatim dribble files of the problem solver at work. The odd order in which the learner does things is due to its peculiar frame-system control structure.

Human: (new-world! up-down-world)
World: (BELOW ONE TWO)
World: (BELOW TWO THREE)
World: (BELOW THREE FOUR)
Learner: Is (BELOW THREE THREE) true?
World: No.
Learner: Is (BELOW FOUR TWO) true?
World: No.
Learner: So far, BELOW is functional at its first place

Of course, below isn't functional. But the learner doesn't yet have any evidence to the contrary. I never got around to implementing a functionality antagonist, so this delusion persists.

Learner: Is (BELOW ONE FOUR) true?
World: Yes.
Learner: Is (BELOW ONE ONE) true?
World: No.
Learner: Creating the reflexive closure of BELOW
Learner: Is (BELOW FOUR FOUR) true?
World: No.
Learner: Is (BELOW TWO TWO) true?
World: No.
Learner: Making BELOW antireflexive
Learner: Is (BELOW FOUR ONE) true?
World: No.
Learner: Is (BELOW FOUR THREE) true?
World: No.
Learner: Is (BELOW THREE TWO) true?
World: No.
Learner: Is (BELOW THREE ONE) true?
World: No.
Learner: Making BELOW antisymmetric
Learner: Is (BELOW TWO FOUR) true?
World: Yes.
Learner: Is (BELOW ONE THREE) true?
World: Yes.
Learner: BELOW seems to be a transitive relation.
Learner: BELOW is an antireflexive partial order relation.
Learner: BELOW is an incrementing function with respect to BELOW.
Learner: BELOW is a monotonic non-decreasing function with respect to BELOW and BELOW.
World: (BELOW FOUR FIVE)
Note in the following how the reflexiveness antagonist keeps asking dumb questions.

Learner: Is (BELOW FIVE FIVE) true?
World: No.
World: (BELOW FIVE SIX)
Learner: Is (BELOW SIX SIX) true?
World: No.
World: (BELOW SIX SEVEN)
Learner: Is (BELOW SEVEN SEVEN) true?
World: No.
World: (BELOW SEVEN EIGHT)
Learner: Is (BELOW EIGHT EIGHT) true?
World: No.
World: (BELOW EIGHT NINE)
Learner: Is (BELOW NINE NINE) true?
World: No.
World: (BELOW NINE TEN)
Learner: Is (BELOW TEN TEN) true?
World: No.
World: (ON-FLOOR ROBOT SIX)
Human: (goal (on-floor robot one))
Learner: Is (ON-FLOOR ROBOT ONE) true?
World: No.
Planner: I don't know how to achieve (ON-FLOOR ROBOT ONE);
    will frobnicate...
Frobnicator: action (DOWN)
World: (NOT (ON-FLOOR ROBOT SIX))
World: (ON-FLOOR ROBOT FIVE)
Learner: Is (ON-FLOOR ROBOT ONE) true?
World: No.
Frobnicator: action (UP)
World: (NOT (ON-FLOOR ROBOT FIVE))
World: (ON-FLOOR ROBOT SIX)
Learner: Is (ON-FLOOR ROBOT ONE) true?
World: No.
Frobnicator: action (DOWN)
World: (NOT (ON-FLOOR ROBOT SIX))
World: (ON-FLOOR ROBOT FIVE)
Learner: Is (ON-FLOOR ROBOT ONE) true?
World: No.
Frobnicator: action (UP)
World: (NOT (ON-FLOOR ROBOT FIVE))
World: (ON-FLOOR ROBOT SIX)
Learner: Is (ON-FLOOR ROBOT ONE) true?
World: No.
Frobnicator: action (DOWN)
World: (NOT (ON-FLOOR ROBOT SIX))
World: (ON-FLOOR ROBOT FIVE)
Learner: Is (ON-FLOOR ROBOT ONE) true?
World: No.
Frobnicator: action (DOWN)
World: (NOT (ON-FLOOR ROBOT SIX))
World: (ON-FLOOR ROBOT FIVE)
Learner: Is (ON-FLOOR ROBOT ONE) true?
World: No.
Frobnicator: action (DOWN)
World: (NOT (ON-FLOOR ROBOT FIVE))
World: (ON-FLOOR ROBOT FOUR)
Learner: Is (ON-FLOOR ROBOT ONE) true?
World: No.
Frobnicator: action (UP)
World: (NOT (ON-FLOOR ROBOT FOUR))
World: (ON-FLOOR ROBOT FIVE)
Learner: Is (ON-FLOOR ROBOT ONE) true?
World: No.
Frobnicator: action (DOWN)
World: (NOT (ON-FLOOR ROBOT FIVE))
World: (ON-FLOOR ROBOT FOUR)
Learner: Is (ON-FLOOR ROBOT ONE) true?
World: No.
Frobnicator: action (DOWN)
Learner: Looks like DOWN mungs ON-FLOOR in its second place only.
Learner: Looks like DOWN mungs ON-FLOOR at a unique second place curry.
The entire expression in vertical bars is the gensymmed name for the temporal abstraction of down.

The entire expression in vertical bars is the gensymmed name for the temporal abstraction of down.

Learner: Creating |DOWN ON-FLOOR 2 CURRY-TEMPORAL-ABSTRACTION|
     as the temporal abstraction of DOWN at 2
Learner: So far, |DOWN ON-FLOOR 2 CURRY-TEMPORAL-ABSTRACTION|
     is functional at its first place
Learner: So far, |DOWN ON-FLOOR 2 CURRY-TEMPORAL-ABSTRACTION|
     is functional at its second place
Learner: Creating the reflexive closure of
     |DOWN ON-FLOOR 2 CURRY-TEMPORAL-ABSTRACTION|
Learner: Making |DOWN ON-FLOOR 2 CURRY-TEMPORAL-ABSTRACTION| antireflexive
Learner: Making |DOWN ON-FLOOR 2 CURRY-TEMPORAL-ABSTRACTION| antisymmetric
Learner: |DOWN ON-FLOOR 2 CURRY-TEMPORAL-ABSTRACTION|
     is a decrementing function with respect to BELOW.
Learner: \(\text{\text{DOWN ON-FLOOR 2 CURRY-TEMPORAL-ABSTRACTION}}\) is a monotonic non-decreasing function with respect to \text{BELOW} and \text{BELOW}.
Learner: Looks like \(\text{\text{DOWN ON-FLOOR 2 CURRY-TEMPORAL-ABSTRACTION}}\) is one-to-one at its second place
Learner: Looks like \(\text{\text{DOWN ON-FLOOR 2 CURRY-TEMPORAL-ABSTRACTION}}\) is one-to-one at its first place
Learner: Looks like \(\text{\text{DOWN ON-FLOOR 2 CURRY-TEMPORAL-ABSTRACTION}}\) is a chain.
Learner: \(\text{\text{DOWN ON-FLOOR 2 CURRY-TEMPORAL-ABSTRACTION}}\) is a thread, rather than a cycle.
World: \(\text{(NOT (ON-FLOOR ROBOT FOUR))}\)
World: \(\text{(ON-FLOOR ROBOT THREE)}\)
Learner: Is \(\text{(ON-FLOOR ROBOT ONE)}\) true?
World: No.
Frobnicator: action (DOWN)
World: \(\text{(NOT (ON-FLOOR ROBOT THREE))}\)
World: \(\text{(ON-FLOOR ROBOT TWO)}\)
Learner: Is \(\text{(ON-FLOOR ROBOT ONE)}\) true?
World: No.
Frobnicator: action (UP)
World: \(\text{(NOT (ON-FLOOR ROBOT TWO))}\)
World: \(\text{(ON-FLOOR ROBOT THREE)}\)
Learner: Is \(\text{(ON-FLOOR ROBOT ONE)}\) true?
World: No.
Frobnicator: action (DOWN)
World: \(\text{(NOT (ON-FLOOR ROBOT THREE))}\)
World: \(\text{(ON-FLOOR ROBOT TWO)}\)
Learner: Is \(\text{(ON-FLOOR ROBOT ONE)}\) true?
World: No.
Frobnicator: action (UP)
World: \(\text{(NOT (ON-FLOOR ROBOT TWO))}\)
World: \(\text{(ON-FLOOR ROBOT THREE)}\)
Learner: Is \(\text{(ON-FLOOR ROBOT ONE)}\) true?
World: No.
Frobnicator: action (DOWN)
World: \(\text{(NOT (ON-FLOOR ROBOT THREE))}\)
World: \(\text{(ON-FLOOR ROBOT TWO)}\)
Learner: Looks like UP mungs ON-FLOOR in its second place only.
Learner: Looks like UP mungs ON-FLOOR at a unique second place curry.
Learner: Creating \(\text{\text{UP ON-FLOOR 2 CURRY-TEMPORAL-ABSTRACTION}}\) as the temporal abstraction of UP at 2
Learner: So far, \(\text{\text{UP ON-FLOOR 2 CURRY-TEMPORAL-ABSTRACTION}}\) is functional at its first place
Learner: So far, \(\text{\text{UP ON-FLOOR 2 CURRY-TEMPORAL-ABSTRACTION}}\) is functional at its second place
Learner: Creating the reflexive closure of \(\text{\text{UP ON-FLOOR 2 CURRY-TEMPORAL-ABSTRACTION}}\)
Learner: Making \(\text{\text{UP ON-FLOOR 2 CURRY-TEMPORAL-ABSTRACTION}}\) antireflexive
Learner: Making \( \text{[UP ON-FLOOR 2 CURRY-TEMPORAL-ABSTRACTION]} \) antisymmetric
Learner: \( \text{[UP ON-FLOOR 2 CURRY-TEMPORAL-ABSTRACTION]} \) is an incrementing function with respect to BELOW.
Learner: \( \text{[UP ON-FLOOR 2 CURRY-TEMPORAL-ABSTRACTION]} \) is a monotonic non-decreasing function with respect to BELOW and BELOW.
Learner: Looks like \( \text{[UP ON-FLOOR 2 CURRY-TEMPORAL-ABSTRACTION]} \) is one-to-one at its second place
Learner: Looks like \( \text{[UP ON-FLOOR 2 CURRY-TEMPORAL-ABSTRACTION]} \) is one-to-one at its first place
Learner: Is \( \text{(ON-FLOOR ROBOT ONE)} \) true?
World: No.
Planner: plan constructed.
Learner: Is \( \text{(ON-FLOOR ROBOT ONE)} \) true?
World: No.
Planner: action (DOWN)
World: \( \text{(NOT (ON-FLOOR ROBOT TWO))} \)
World: \( \text{(ON-FLOOR ROBOT ONE)} \)
Learner: Is \( \text{(ON-FLOOR ROBOT ONE)} \) true?
World: Yes.
Learner: Is \( \text{(ON-FLOOR ROBOT ONE)} \) true?
World: Yes.
Planner: plan succeeded.

Human: \( \text{(new-worldl puton-world)} \)
World: \( \text{(ON A B)} \)
World: \( \text{(ON B TABLE)} \)
World: \( \text{(ON C TABLE)} \)
Learner: So far, ON is functional at its second place
Learner: Is \( \text{(ON C C)} \) true?
World: No.
Learner: Creating the reflexive closure of ON
Learner: Is \( \text{(ON TABLE TABLE)} \) true?
World: No.
Learner: Is \( \text{(ON B B)} \) true?
World: No.
Learner: Is \( \text{(ON A A)} \) true?
World: No.
Learner: Making ON antireflexive
Learner: Is \( \text{(ON TABLE C)} \) true?
World: No.
Learner: Making ON antisymmetric
Learner: Is \( \text{(ON A TABLE)} \) true?
World: No.
World: \( \text{(ON X TABLE)} \)
Learner: Is \( \text{(ON X X)} \) true?
World: No.
World: \( \text{(ON Z TABLE)} \)
Learner: Is \( \text{(ON Z Z)} \) true?
World: No.

Human: (apply puton a table)
World: (NOT (ON A B))
World: (ON A TABLE)

Human: (apply puton b a)
World: (NOT (ON B TABLE))
World: (ON B A)

Human: (apply puton c b)
World: (NOT (ON C TABLE))
World: (ON C B)

Human: (apply puton x c)
World: (NOT (ON X TABLE))
World: (ON X C)

Human: (apply puton x c)
Learner: Creating a theory of exceptions to PUTON being a PUSH on ON
Learner: PUTON pops the top of ON
Learner: PUTON pushes onto the top of ON
Learner: Creating a theory of exceptions to PUTON being a POP on ON
Learner: PUTON pushes onto the top of ON
Learner: The second argument to PUTON is the end of the ON stack it pushes.
Learner: The first argument to PUTON is the item pushed in ON.
Learner: PUTON pops the top of ON
World: Barf! (PUTON X C) -- application invalid now.

Human: (apply puton x table)
Learner: The first argument to PUTON is the end of the ON stack it pops.
Learner: The first argument to PUTON is the item popped in ON.
Learner: PUTON is a composition of a POP and a PUSH when acting on ON.
World: (NOT (ON X C))
World: (ON X TABLE)

Human: (apply puton c z)
World: (NOT (ON C B))
World: (ON C Z)

Human: (apply puton b x)
World: (NOT (ON B A))
World: (ON B X)

Human: (apply puton c a)
World: (NOT (ON C Z))
World: (ON C A)
Human: (goals (on a b) (on b c))
Learner: Is (ON A B) true?
World: No.
Planner: Plan constructed.
Planner: action (PUTON C Z)
World: (NOT (ON C A))
World: (ON C Z)
Planner: action (PUTON B C)
World: (NOT (ON B X))
World: (ON B C)
Planner: action (PUTON A B)
World: (NOT (ON A TABLE))
World: (ON A B)
Learner: Is (ON A B) true?
World: Yes.
Learner: Is (ON B C) true?
World: Yes.
Planner: plan succeeded.
Acknowledgments

The theory described here benefitted from discussions with many people, including many whom I will probably not remember to include here. Phil Agre, John Batali, Anne Chapman, Gary Drescher, Carl Feynman, Ken Forbus, Ken Haase, Danny Hillis, Dan Huttenlocher, David McAllester, Chuck Rich, and Hazel Rovno contributed especially.

Phil Agre read a draft of this paper and provided many useful suggestions.
References


