Machine Learning, 16, 11-36 (1994)
© 1994 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Acquisition of Children's Addition Strategies: A Model of Impasse-Free, Knowledge-Level Learning

RANDOLPH M. JONES rjones@eecs.umich.edu
Artificial Intelligence Laboratory, University of Michigan, 1101 Beal Avenue, Ann Arbor, MI 48109-2110

KURT VANLEHN vanlehn@cs.pitt.edu
Learning Research and Development Center, and Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260

Editor: Michael Pazzani

Abstract. When children learn to add, they count on their fingers, beginning with the simple SUM strategy and gradually developing the more sophisticated and efficient MIN strategy. The shift from SUM to MIN provides an ideal domain for the study of naturally occurring discovery processes in cognitive skill acquisition. The SUM-to-MIN transition poses a number of challenges for machine-learning systems that would model the phenomenon. First, in addition to the SUM and MIN strategies, Siegler and Jenkins (1989) found that children exhibit two transitional strategies, but not a strategy proposed by an earlier model. Second, they found that children do not invent the MIN strategy in response to impasses, or gaps in their knowledge. Rather, MIN develops spontaneously and gradually replaces earlier strategies. Third, intricate structural differences between the SUM and MIN strategies make it difficult, if not impossible, for standard, symbol-level machine-learning algorithms to model the transition. We present a computer model, called GIPS, that meets these challenges. GIPS combines a relatively simple algorithm for problem solving with a probabilistic learning algorithm that performs symbol-level and knowledge-level learning, both in the presence and absence of impasses. In addition, GIPS makes psychologically plausible demands on local processing and memory. Most importantly, the system successfully models the shift from SUM to MIN, as well as the two transitional strategies found by Siegler and Jenkins.

Keywords: cognitive simulation, impasse-free learning, probabilistic learning, induction, problem-solving strategies

1. Introduction

This research focuses on modeling naturally occurring discovery processes in cognitive skill acquisition. In particular, it provides an explanation of the well-known SUM-to-MIN transition that children exhibit when they are learning to add (Ashcraft, 1982, 1987; Groen & Parkman, 1972; Groen & Resnick, 1977; Kaye, Post, Hall, & Dineen, 1986; Siegler & Jenkins, 1989; Svenson, 1975). On the surface, this transition appears to be a case of *symbol-level* or *speed-up* learning (Dietterich, 1986). The SUM and MIN strategies are both correct and complete addition algorithms, but the MIN strategy is much faster. However, closer inspection reveals that the transition involves changes to the *structure* of the solution, which cannot be explained by conventional symbol-level learning methods. In addition, children appear to invent the MIN strategy spontaneously, rather than in response to any failures or impasses in problem solving. Thus, a successful model of the SUM-to-MIN transition must make dramatic changes in the strategies, and it must be able to do so without the benefit of impasses to drive learning.

Earlier work attests to the complexity of modeling this structurally intricate strategy shift. Neches (1987) was able to model the transition using a machine learner based on compiler optimization techniques, but the model required implausibly large amounts of extremely detailed information about both ongoing processes and related past experiences. Moreover, it predicted that subjects would briefly display certain strategies on their way to the MIN strategy, but these strategies were not observed in subsequent empirical work (Siegler & Jenkins, 1989). Other intermediate strategies were observed instead.

The research problem is to find a learning method (or methods) that can make the SUM-to-MIN transition, use only plausible amounts of computation and memory, and explain the observed intermediate strategies. In this paper, we concentrate on explaining the results of a longitudinal study carried out by Siegler and Jenkins (1989). They found that children invent the MIN strategy and two intermediate strategies independently, without any instruction on the new strategies. More importantly, Siegler and Jenkins discovered that the invention of the MIN strategy does not appear to be driven by failures or impasses in solving problems. Finally, we argue that learning the MIN strategy requires a form of knowledge-level learning (Dietterich, 1986) that introduces new, more efficient behavior, rather than simply tuning or composing old knowledge.

We describe a computational model, called GIPS (for General Inductive Problem Solver), that invents the MIN strategy as well as the correct transitional strategies. In addition, GIPS smoothly integrates a general problem-solving architecture with a simple, independently motivated learning algorithm. The learning algorithm applies a probabilistic concept learner to all of GIPS' major decision points, allowing it to combine impasse-driven, impasse-free, symbol-level, and knowledge-level learning in a single, uniform framework. Remarkably, the relatively simple problem-solving and learning algorithms interact so as to explain intricate strategy shifts that previous systems could not account for.

In the following section, we describe the SUM-to-MIN transition, and explain its complexities in detail. Next, we present the GIPS system, its representation of the addition domain, and its account of the SUM-to-MIN shift. The last section discusses GIPS' account and compares it to those offered by other models.

2. The SUM-to-MIN transition

When young children first learn to add two small numbers, they use the so-called SUM strategy. They create sets of objects to represent each addend, then count the objects in the union of the two sets. For example, suppose a child is asked, "What is 2 plus 3?" In order to solve this problem, the child says, "1, 2," while raising two fingers on the left hand; then "1, 2, 3," while raising three fingers on the right hand; then "1, 2, 3, 4, 5," while counting all the raised fingers. This is called the SUM strategy because its execution time is proportional to the sum of the two addends. Older children use a more efficient strategy, called the MIN strategy. In following this strategy, the child first announces the value of the larger addend, then counts onward from it. For instance, in order to solve $2 + 3$, the child would say, "3," then say, "4, 5," while raising two fingers on one hand. The execution time for the MIN strategy is proportional to the minimum of the two addends. Algorithms for the two strategies appear in Table 1.

Table 1. A comparison of the SUM and MIN strategies.

	Initial SUM strategy	MIN strategy
1.	Assign first addend to left hand;	Assign larger addend to left hand;
2.	Assign second addend to right hand;	Assign smaller addend to right hand;
3.	Let Counter be 0;	
4.	Loop	
5.	Raise finger on left hand;	
6.	Let Counter be Counter + 1;	
7.	Until Counter = left hand addend;	Let Counter be left hand addend;
8.	Let Counter be 0;	
9.	Loop	Loop
10.	Raise finger on right hand;	Raise finger on right hand;
11.	Let Counter be Counter + 1;	Let Counter be Counter + 1;
12.	Until Counter = right hand addend;	Until number of raised fingers = right hand addend;
13.	Let Counter be 0;	
14.	Loop	
15.	Mark raised finger;	
16.	Let Counter be Counter + 1;	
17.	Until number of marked fingers = number of raised fingers;	
18.	Let Answer be Counter;	Let Answer be Counter;

Although the SUM strategy is taught in school, the MIN strategy appears to be invented by the children themselves. The best evidence for this comes from a longitudinal study by Siegler and Jenkins (1989). They interviewed eight children weekly for 11 weeks, each time asking them to solve about 15 orally presented addition problems. After each problem, they asked the children how they got their answers. They also told each child whether the answer was correct, and gave the child a gold star if it was. Finally, they analyzed videotapes of the session and classified the child's behavior on each problem according to the strategy that the child used. As far as Siegler and Jenkins could determine, the only instruction that the subjects received during this period was their school's normal instruction on the SUM strategy. Nonetheless, seven of the eight children eventually began to use the MIN strategy. Moreover, the children appear to have discovered this strategy during the video-taped sessions. The tapes make it clear that they received no help from the experimenter, so the MIN strategy appears to have been invented by the subjects themselves. In addition, Siegler and Jenkins found two transitional counting strategies that the subjects used while proceeding from SUM to MIN. These are the SHORTCUT SUM strategy, in which a subject raises and counts fingers from one to the final sum across both hands, and the FIRST strategy, which is similar to MIN, except that the order for adding two addends is not determined by their relative sizes.

2.1. Impasse-free learning during strategy invention

A central issue for computational learning systems is deciding *when* to learn. A popular method is to learn when an *impasse* occurs, suggesting a hole in the system's knowledge base. The exact definition of "impasse" depends on the problem-solving architecture, but roughly speaking, an impasse occurs for a problem solver when it comes across a goal that cannot be achieved by any operator that is believed to be relevant to the task at hand. The essential idea of impasse-driven learning is to resolve the impasse somehow, then store the resulting experience in such a way that future impasses will be avoided or at least handled more efficiently. Many systems use impasse-driven learning, including LPARSIFAL (Berwick, 1985), OCCAM (Pazzani, Dyer & Flowers, 1986), SWALE (Schank, 1986), SOAR (Newell, 1990), SIERRA (VanLehn, 1990), and CASCADE (VanLehn & Jones, 1993; VanLehn, Jones & Chi, 1992). SOAR is perhaps the best-known impasse-driven learning system, but its definition of impasse is a bit idiosyncratic. It uses impasse-driven learning for *all* changes to memory. Because people automatically store a dense record of their on-going experiences (Tulving's episodic memory), a proper SOAR model must have impasses very frequently, perhaps several per second. Unlike SOAR, other models record their personal experiences with mechanisms that are separate from their impasse-driven learning mechanism. For them, an impasse corresponds to the subjective experience of getting stuck and knowing that you are stuck. In one detailed psychological study (VanLehn, 1991), this occurred about once every half hour. In this paper, we use "impasse" only for these higher level impasses.

Because of the importance of impasse-driven learning in current models of intelligence, Siegler and Jenkins looked specifically for signs of impasses in their study. In particular, they designed some of the problems to cause impasses by making one of the addends very large (e.g., 23 + 1). They found that "The specific problems on which the children first used the MIN strategy were 2 + 5, 4 + 1, 3 + 1, 1 + 24, 5 + 2, and 4 + 3. These problems did not deviate from the characteristics of the overall set in any notable way" (p. 67). In fact, some of the children had earlier successfully solved exactly the same problem that they were working on when they discovered the MIN strategy. Although the large-addend problems did cause subjects who had already invented the MIN strategy to start using it more frequently, the problems did not cause those who had not invented the strategy to do so.

In addition, Siegler and Jenkins sought signs of impasses by examining solution times and errors in the vicinity of the discovery events. Solution times were longer than normal for the problems where the discovery occurred (a median of 17.8 seconds vs. overall median of 9.8 seconds) and for the problems immediately preceding the discovery trial (median 18 seconds). This might suggest some kind of impasse. However, the specific problems being worked on at those points were not particularly difficult. On the discovery trial, 71% of the problems involved addends that were both 5 or less and thus could each be stored on a single hand. This rate is almost identical to the rate of 72% for the set as a whole. Moreover, 88% of the problems encountered in the session prior to the discovery did not include a large addend. Using error rates as a measure of difficulty yielded a similar finding. Siegler and Jenkins report,

> Prior to discovering the min strategy, children had answered correctly 12 of the 16 problems that they had encountered within the session. This level of accuracy, 75%, was not substantially worse than the 85% correct answers that children generated across the entire practice set. Further, three of the four errors were generated by a single child; the other four children collectively made only one error on the 12 trials they encountered in the same session but before their discoveries. This, together with the fact that two other children used the min strategy for the first time on the

> first trial of a session, indicated that incorrect answers are not necessary to motivate discovery of a new strategy. However, the long solution times just prior to the discoveries do suggest a heightening of cognitive activity even without incorrect answers to motivate it. (p. 69)

The absence of impasses near the critical learning events presents a challenge for current theories of learning. However, Siegler and Jenkins suggest a reconciliation between their findings and the impasse-driven learning theories:

> Two types of strategy changes can be distinguished: changes in which the main difference between the strategies is in the answers themselves, and changes in which the main differences are not in the answers that are generated but rather in the efficiency with which answers are generated and/or the aesthetic appeal of the procedures. The first type of strategy change may occur primarily as a result of encountering impasses, but the second may typically occur for other reasons. (p. 104)

2.2. Symbol-level vs. knowledge-level learning

Dieterrich (1986) defines symbol-level learning as learning that improves the performance of a system, but does not increase the deductive closure of the system's knowledge. In contrast, learning at the knowledge level involves actually changing the system's knowledge base or domain theory, thus changing what the system can possibly deduce (given enough time). Most current problem-solving systems learn at the symbol level, achieving better performance by improving their search through a problem space. In general, this type of learning has taken one of two forms: search tuning and macro-operator formation. Search tuning involves methods for decreasing the average branching factor of the search via search-control rules (Minton, 1988), selection conditions on operators (Anderson, 1983; Mitchell, Utgoff, & Banerji, 1983), numerical strengths on operators (Langley, 1985), or similar methods. In contrast, macro-operators decrease the average depth of the space by composing the conditions and actions of operator sequences into individual operators (Anderson, 1983; Iba, 1989; Korf, 1985). Both of these forms of learning can greatly improve the quality of a system's search for a solution to a problem, and sometimes they can also improve the quality of the solution itself.

A close examination of the transition from SUM to MIN indicates that a model based strictly on symbol-level learning can explain some shifts, but has difficulties explaining others. Let us consider in turn the strategy differences that appear in Table 1. In lines 1 and 2, subjects learn to assign addends to their hands based on the addends' relative sizes. Provided this type of feature is included in its representation language, a symbol-level learner can easily discover the feature's relevance, based on failures in generating correct answers.

The procedure for representing an addend on the left hand (lines 4–7) in the SUM strategy is replaced by a single line in the MIN strategy, which simply asserts the output of the procedure. This shift could possibly be modeled with macro-operators, except that they would also force the appropriate number of fingers to be raised on the left hand. However, children can generate the correct counter value *without* raising fingers. This indicates that they can determine *which* output of the procedure is relevant to the task at hand. Macro-operators could not model this, because incrementing the counter is always paired with raising a finger in the SUM strategy. As we shall see later, our model predicts that children can identify the relevant output of the procedure by learning new preconditions on the operator that terminates the procedure.

The next difference between the two strategies appears in the procedure for representing the right-hand addend (lines 9–12). The only difference between the two procedures is the termination criterion for the loop. This is a somewhat simpler transition than the previous one, but it still causes problems for a symbol-level learner. The children appear to learn that recognizing the number of raised fingers on a hand is a better stopping criterion than using the value of the counter. This discovery occurs even though both criteria generate correct answers. However, recognizing the number of raised fingers serendipitously leads to a more efficient solution, because the counter no longer has to be zeroed in order to represent each addend (lines 8 and 9). This in turn makes the final loop (lines 14–17) unnecessary in the MIN strategy, because the correct answer is already available.

Once again, it is difficult to see how a symbol-level learner could account for this representation shift. Our model determines that the number of fingers raised on a hand is highly correlated with the value of the addend being represented. It eventually replaces the counter value as the loop termination criterion because it allows the loop to terminate faster. This transition requires the system to change its preconditions for the operator that terminates the loop. Thus it involves knowledge-level adjustment of the domain representation and cannot be explained simply in terms of knowledge tuning or the formation of macro-operators.

Our analysis of the differences between the SUM and MIN strategies, together with Siegler and Jenkins' findings, provide some strict criteria that a model of the SUM-to-MIN transition should meet. First, the model should proceed from usage of the SUM strategy to the MIN strategy without any outside instruction (other than feedback on the correctness of the answers). It should invent the same transitional strategies that Siegler and Jenkins found in their subjects. It also must account for the ability to invent new strategies even when there are no impasses to drive learning. Finally, the model must incorporate a mechanism for knowledge-level learning, so that it can adapt its representation of the task domain. GIPS, the model we describe in the next section, meets these criteria.

3. The General Inductive Problem Solver

GIPS is a problem solver that uses *flexible means-ends analysis* as its performance mechanism (Jones, 1993; Langley & Allen, 1991). Its learning mechanism is based on Schlimmer's (1987; Schlimmer & Granger, 1986a, 1986b) STAGGER system, which uses a probabilistic induction technique to learn concept descriptions from examples. GIPS uses its induction algorithm to learn search-control knowledge for its operators, assigning credit and blame in a manner similar to SAGE (Langley, 1985) and LEX (Mitchell, Utgoff, & Banerji, 1983). However, GIPS also uses probabilistic induction to learn new preconditions on its operators, thus modifying the descriptions of the operators themselves. Inductive modification of preconditions (as opposed to inductive modification of search-control knowledge)

Table 2. A GIPS operator to increment the value of a counter.

```
COUNT(?Hand, ?Initvalue, ?Finalvalue)
Preconditions:
   Hand(?Hand)
   Just-raised(?Hand)
   Counter-value(?Initvalue)
Add conditions:
   Counter-value(?Finalvalue)
Delete conditions:
   Counter-value(?Initvalue)
   Just-raised(?Hand)
Constraints:
   ?Finalvalue is ?Initvalue + 1
```

appears to be a new machine-learning technique. Although it could be risky, in that it seems capable of destroying the correctness of the operator set, we show that when properly controlled, it can produce correctness-preserving speed increases that standard techniques have not been able to produce. From a cognitive-modeling perspective, both learning about search control and learning new operator representations play crucial roles in the SUM-to-MIN transition.

3.1. Representation of the addition domain

In this section, we describe GIPS' representation of the task domain. GIPS describes the world as a set of relations between objects. In the domain of addition, these objects and relations include the numbers that are part of the problem, the state of the problem solver's "hands" while it is adding, and the value of a counter that the problem solver keeps "in its head." In addition, GIPS represents possible actions in the domain with operators that are similar in representation to those used by STRIPS (Fikes & Nilsson, 1971). Each includes a set of preconditions, add conditions, delete conditions, and possibly a set of constraints on the variable bindings.

As an example, consider the operator in Table 2, which increments the value of the counter. This operator has three variable parameters, `?Hand`, `?Initvalue`, and `?Final-value` (throughout this paper, an atom beginning with "?" represents a variable). The preconditions for the operator check the current value of the counter and make sure that the system has just raised a finger that needs to be counted. The constraint generates a final value for the counter by incrementing the initial value. GIPS' constraint mechanism allows constraints to propagate forwards or backwards, so this constraint can also compute the necessary initial value if it is given the final value as a goal. Finally, when the operator executes, it will delete the initial value of the counter and record the final value. In addition, it will delete the "just-raised" condition so that the finger will not be counted twice.

GIPS represents the addition domain with the 16 operators presented in Table 3. There are two particular operators, which we refer to as the ADDEND-REPRESENTED operators, that are involved in most of the strategy shifts. For future reference, the series of precon-

Table 3. Operators for the addition domain.

SELECT-HAND:	Select an addend to be counted on each hand. The left hand is always counted first.
COUNT-OUT-LEFTHAND:	Represent or count the left-hand addend.
COUNT-OUT-RIGHTHAND:	Represent or count the right-hand addend.
START-COUNT:	Keep track of the counter value while raising fingers.
START-RAISE:	Begin raising fingers in order to represent an addend.
RAISE-FINGER:	Raise a finger.
COUNT:	Count the last raised finger by incrementing the counter value.
LEFT-ADDEND-REPRESENTED:	Stop counting and raising fingers on the left hand.
RIGHT-ADDEND-REPRESENTED:	Stop counting and raising fingers on the right hand.
CLOBBER-COUNTER:	Set the counter value to zero.
COUNT-UP-BOTH-ADDENDS:	Make sure both addends have been counted together.
START-MARK-COUNT:	Keep a running count while marking raised fingers.
MARK-FINGER:	Mark a finger that has already been raised.
MARK-COUNT:	Count the last marked finger by incrementing the counter value.
END-MARK-COUNT:	Stop marking fingers on a hand.
DETERMINE-ANSWER:	Announce the answer.

Table 4. A series of preconditions for LEFT-ADDEND-REPRESENTED.

SUM strategy (a):
`Raising(LeftHand)`
`Assigned(LeftHand, ?Value)`
`Counter-value(?Value)`
	SUM strategy (b):
	`Raising(LeftHand)`
	`Assigned(LeftHand, ?Value)`
	`Counter-value(?Value)`
	`Raised-fingers(LeftHand, ?Value)`

SHORTCUT SUM strategy (c):
`Raising(LeftHand)`
`Assigned(LeftHand, ?Value)`
`Raised-fingers(LeftHand, ?Value)`
	MIN strategy (d):
	`Raising(LeftHand)`
	`Assigned(LeftHand, ?Value)`

ditions that the LEFT-ADDEND-REPRESENTED operator acquires in going from SUM to MIN appears in Table 4. For our study, we initialized GIPS' search-control and precondition knowledge for the 16 operators such that the system generates the SUM strategy on addition problems. We will discuss this initialization in more detail after presenting GIPS' performance algorithm and low-level knowledge representation.

3.2. *Performance algorithm*

As mentioned above, GIPS' problem-solving algorithm (see Table 5) is a form of flexible means-ends analysis, borrowed from the EUREKA system (Jones, 1993). As with standard means-ends analysis, the algorithm is based on trying to achieve a state change. The desired change is represented by a TRANSFORM, which is simply a pair consisting of the current state and some goals (an example appears in Table 6). In order to achieve

Table 5. GIPS' algorithm for solving problems.

```
TRANSFORM(CurState,Goals): Returns NewState
  If CurState satisfies Goals
     Then Return NewState as CurState
     Else Let OpSet be the ordered set of selected
                    operator instantiations;
          Let FailedOps be Nil;
          Loop for Operator in OpSet
              Let TempState be APPLY(CurState,Operator);
              If TempState is "Failed State"
                 Then push Operator onto FailedOps and continue loop;
              Let TempState be TRANSFORM(TempState,Goals);
              If TempState is "Failed State"
                 Then push Operator onto FailedOps and continue loop;
              Store (CurState,Goals) as a positive example for the
                selection concept of Operator;
              Store (CurState,Goals) as a negative example for the
                selection concept of each operator in FailedOps;
              Return NewState as Tempstate
          End loop;
          Return NewState as "Failed State";

APPLY(CurState,Op): Returns NewState
  Let P be PRECONDITIONS(Op);
  If CurState satisfies the execution concept of Op
     Then If the user says Op is executable
              Then Store CurState as a positive example for the
                     execution concept of Op;
                   Return NewState as EXECUTE(CurState,Op)
              Else Store CurState as a negative example for the
                     execution concept of Op;
  Let TempState be TRANSFORM(CurState,P);
  If TempState is "Failed State"
     Then Return NewState as "Failed State"
     Else Return NewState as APPLY(TempState,Op);
```

this transformation, GIPS selects an operator and attempts to apply it. If the operator's preconditions are met, GIPS executes it and the current state changes. If some of the preconditions are not met, a new TRANSFORM is created with the preconditions as the new goals. When this TRANSFORM is achieved, GIPS returns to the old TRANSFORM and attempts again to apply the operator. So far, this is simply a description of standard means-ends analysis.

The difference between standard and flexible means-ends analysis occurs in the selection of an operator to apply. Standard means-ends analysis requires that the actions of any selected operator directly address the goals of the TRANSFORM. In flexible means-ends analysis, operator selection is determined by a selection algorithm that can use any criteria to choose an operator. In order for the selection algorithm to be useful, it is usually under the direct control of the system's learning mechanism. In GIPS, operator selection is determined by *selection concepts*. Each operator is associated with an explicit concept that

Table 6. An example of a TRANSFORM for the addition domain.

Current State:	Goals:
On-Paper(First,Two)	Raising(LeftHand)
On-Paper(Second,Three)	Assigned(LeftHand,?Value)
Assigned(LeftHand,Three)	Counter-Value(?Value)
Counter-Value(Zero)	
Raised-Fingers(LeftHand,Zero)	

indicates when it should be selected. If the concept depends mostly on the current state of the TRANSFORM, then the operator will act like a forward-chaining inference rule and execute whenever the state is appropriate, regardless of the current goals. If the concept depends mostly on the goals of the TRANSFORM, then it will act like a backward-chaining inference rule. Typically, forward and backward operators intermingle during problem solving, yielding a psychologically plausible blend of goal-directed and opportunistic behavior.

In GIPS, each operator has a selection concept. The representation of a selection concept is similar to the representation of a TRANSFORM, consisting of a set of literals (predicates that may or may not be negated) describing the current state and goals. In addition, however, each literal in a selection concept has two numerical values associated with it: sufficiency and necessity. In order to evaluate the selection value of an operator, GIPS matches the literals against the current TRANSFORM. It determines the subset of literals that match (M) and fail to match (F), then calculates

$$\text{Selection Value} = \text{Odds}(C) \prod_{L \in M} S_L \prod_{L \in F} N_L,$$

where Odds(C) is the prior odds that the concept's operator is worth selecting, S_L is the sufficiency of the literal, L, with respect to the concept, and N_L is the necessity of L with respect to the concept. A sufficiency score that is much greater than 1 indicates that a literal is very sufficient for the selection concept. That is, if S_L is a high value, then the selection value will be high if the literal, L, appears in the current TRANSFORM. In contrast, a literal is very necessary if the necessity value is much *less* than 1. In other words, if N_L is low, it means that the selection value will likely be low *unless* L appears in the current TRANSFORM.

The above formula is used by STAGGER, Schlimmer's (1987) concept formation system, to estimate the odds that a given object is an instance of a particular concept. However, a major difference between STAGGER and GIPS is that STAGGER worked exclusively with propositional knowledge representations. In contrast, the literals in GIPS' concepts are general predicates that can also contain variables. This means that the relations in a given TRANSFORM will generally only *partially* match the relations in a concept, and the TRANSFORM may in fact match the concept in more than one way. In these cases, GIPS finds a number of partial matches and calculates a selection value for each one. Each of these matches in turn represents a different instantiation of the operator attached to the selection concept. Thus, the selection procedure typically returns a number of different instantiations of a number of different operators. When all the operator instantiations have been found

Table 7. The initial selection and execution concepts for LEFT-ADDEND-REPRESENTED.

Selection					
CURRENT STATE	S	N	GOALS	S	N
`Raising(LeftHand)`	1.03	0.98			
`Assigned(LeftHand,?Value4)`	1.03	0.98			
`Counter-Value(?Value4)`	1.03	0.98	`Counter-Value(?Value4)`	1.00	1.00
`Raised(LeftHand)`	0.01	1.01	`Raised(LeftHand)`	10.00	0.91
`Counted(LeftHand)`	0.71	1.01	`Counted(LeftHand)`	10.00	0.91
Execution					
CURRENT STATE	S	N			
`Assigned(LeftHand,?Value4)`	2.71	0.05			
`Raising(LeftHand)`	2.71	0.05			
`Counter-Value(?Value4)`	2.71	0.05			

and their selection values have been calculated, GIPS throws out all the instantiations with a selection value less than 1. The remaining instantiations are ordered according to their selection values.

GIPS differs from standard means-ends systems in one more important way. In standard problem-solving systems, each operator has a set of *preconditions*, which are used in two ways. First, they determine when the operator can execute. Second, they dictate which subgoals should be set up via means-ends analysis when an operator is not yet executable. GIPS uses the preconditions to set up subgoals, but it does not use them to determine the executability of the operators. Rather, each operator has an associated *execution concept* that dictates when the system will try to execute it. GIPS' execution concepts are similar in form to selection concepts, except they contain literals describing the current state but not the current goals.

As mentioned previously, GIPS' initial selection and execution concepts were set up to generate the SUM strategy for addition. The literals of each operator's selection concept were set to the preconditions and the goals that the operator could satisfy. The necessity and sufficiency of these literals were set so that they would be retrieved in either a backward-chaining or forward-chaining fashion, depending on the role of the operator in the domain. For example, pure backward-chaining operators had each of their goal literals set with high sufficiency. Forward-chaining operators had each of their current state literals set with high necessity. Finally, each operator had an initial set of preconditions, and the execution concept for each operator was initialized to the literals occurring in the preconditions, each with high necessity.

As an example, the initial preconditions for LEFT-ADDEND-REPRESENTED appear in Table 4(a), with its initial selection and execution concepts in Table 7. An examination of Table 7 shows that LEFT-ADDEND-REPRESENTED is likely to be selected when the current TRANSFORM's goals include `Raised(LeftHand)` or `Counted(LeftHand)` (high S value), *unless* these literals also appear in the TRANSFORM's current state (low S value). The operator is set to execute only when all three of `Assigned(LeftHand, ?Value4)`, `Raising(LeftHand)`, and `Counter-Value (?Value4)` are matched by literals in the current TRANSFORM's state description (medium S value and low N value).

From our description of GIPS' general problem-solving algorithm, it is clear that there are exactly two types of choice that the system has to make while solving problems: the choice of which operator to apply next, and the choice of whether to execute an operator or subgoal on its preconditions. It is appealing to apply a uniform learning and decision mechanism in a system's performance algorithm, so GIPS uses its probabilistic concept-matching and learning mechanism for both of these decision points. Other problem solvers include additional types of decisions. For example, PRODIGY (Minton, 1988) makes an explicit decision for which subgoal to work on next, whereas that decision is implicit in GIPS' operator-selection decision. Our experiences indicate that a STAGGER-like algorithm can be used at any type of decision point, providing the same kinds of learning benefits to each. Thus, if we decided to have GIPS make an explicit choice about which subgoal to work on next, we would also use the concept-matching algorithm for that, enabling the system to learn and improve its behavior for choosing subgoals as well. In the following section, we discuss how execution and selection concepts change with experience. More importantly, we explain how changes in the execution concepts directly lead to representation changes in the operator preconditions.

3.3. *Learning in GIPS*

GIPS adjusts its selection concepts on the basis of its successes and failures while solving problems. When a TRANSFORM is finally solved, GIPS adjusts the sufficiency and necessity values of the successful operator so that the operator will be rated even higher the next time a similar TRANSFORM occurs. For each operator that initiated a failure path (i.e., it took the first step off a TRANSFORM's solution path), GIPS adjusts the values in its selection concept so that it will receive a lower value next time. Note that GIPS considers every TRANSFORM to be a "problem," so it can learn about any particular TRANSFORM even if it doesn't lie on the solution path to some global problem. In order to do this kind of learning, GIPS must store the current solution path and every operator that led off it. However, as soon as each individual TRANSFORM in a problem is finished, and the updating is completed, that portion of the solution path is discarded.

This method of assignment for credit and blame is similar to the method used by other problem-solving systems that include concept-learning mechanisms (Langley, 1985; Mitchell, Utgoff, & Banerji, 1983). These systems (and GIPS) can easily assign credit and blame, because they backtrack until they find a solution to the current problem. Then, each decision that leads off the final solution path is classified as a bad decision (a negative example), and each decision that lies on the final solution path is classified as a good decision (a positive example).

However, GIPS differs from these previous systems in the details of its concept-learning algorithm. GIPS computes the sufficiency and necessity scores for literals in a concept (S_L and N_L) with the following equations:

$$S_L = \frac{P(L \text{ matches } I \mid I \in C)}{P(L \text{ matches } I \mid I \notin C)},$$

$$N_L = \frac{P(L \text{ does not match } I \mid I \in C)}{P(L \text{ does not match } I \mid I \notin C)},$$

where $I \in C$ means that TRANSFORM instance, I, is a positive example of the concept, C, and "L matches I" means that literal, L, of the concept is matched by some literal in I. Thus, sufficiency and necessity for each literal are determined by four conditional probabilities.

GIPS learns by updating its estimates of these conditional probabilities. For each literal in a selection concept, GIPS records four values: t, the total number of examples (positive *and* negative) that the system has stored into this selection concept; p, the number of those that were positive examples; l, the total number of times this literal has been matched by any (positive or negative) example; and c, the number of times the literal has been matched in a positive example. In precise form, the conditional probabilities are estimated by

$$P(L \text{ matches } I \mid I \in C) = \frac{c}{p},$$

$$P(L \text{ matches } I \mid I \notin C) = \frac{l-c}{t-p},$$

$$P(L \text{ does not match } I \mid I \in C) = \frac{p-c}{p},$$

$$P(L \text{ does not match } I \mid I \notin C) = \frac{t+c-p-l}{t-p}.$$

As indicated in the algorithm in Table 5, GIPS learns by storing an instance (the literals describing the state and goals of the current TRANSFORM) as a positive or negative example of an operator's selection concept (depending on whether the operator led to a solution or a failed search path). Every time the system stores a TRANSFORM as a positive or negative example, it matches the literals in the TRANSFORM to the literals in the selection concept. If there are any literals in the new instance that do not already appear in the selection concept, they are added into the selection concept's representation. Finally, GIPS increments the appropriate counts for each literal: always incrementing t, incrementing p if the instance is a positive example, incrementing l if the literal is matched by a literal in the instance, and incrementing c if the instance is a positive example *and* the literal is matched. For the interested reader, Schlimmer (1987; Schlimmer & Granger, 1986a, 1986b) provides excellent, detailed descriptions of the learning algorithm and its behavior in classification tasks.

We have so far described how GIPS updates its selection concepts. These concepts determine when operators are selected to achieve a TRANSFORM, so they represent search-control knowledge. As we have mentioned, the system also must adapt its execution concepts. The conditional probabilities are updated identically to selection concepts. However, the assignment of credit and blame is a bit different.

Assignment of credit and blame for execution concepts can be computed in a manner similar to credit and blame for selection concepts. When GIPS thinks that a particular operator should execute, it blindly carries on and eventually generates an answer. However, it is possible that the answer will be wrong, indicating that the operator should not have executed (i.e., GIPS' execution concept for that operator is wrong). If GIPS is allowed to backtrack until it eventually generates the correct answer, it can precisely determine which situations should be stored as negative and positive instances of the execution concept, just as with selection concepts. We discovered that in some instances, when GIPS unfortunately generated multiple "bad" execution concepts, backtracking could take quite a while before the system would generate a correct answer and do the appropriate learning. We finessed this problem by giving the system immediate feedback on whether it would generate a correct answer when it attempted to execute operators.

Unfortunately, this does not tell the whole story on credit and blame assignment. In Siegler and Jenkins' study, they awarded each subject a gold star after the subject gave a correct answer, but they did not force the subjects to keep working on the problems until they could give a correct answer, as we do with GIPS. For a strict model of this experiment, we would give GIPS feedback after it generates a complete solution, and not force the system to backtrack. However, if GIPS is not allowed to backtrack, it must incorporate an incremental credit-assignment algorithm, such as the bucket-brigade algorithm (Holland, Holyoak, Nisbett, & Thagard, 1986). In our study, we were more concerned with the order of acquired strategies than the speed of acquisition, so we did not implement such an algorithm in the current version of GIPS. We are convinced that a more realistic credit-assignment algorithm would slow down learning, but would not disturb the order of strategy acquisition. However, future research with GIPS should certainly address this issue.

The final aspect of learning in GIPS involves changing the preconditions on operators. When GIPS successfully predicts that an operator should execute, but the probabilistic execution concept does not agree with the current preconditions of the operator, the system changes the preconditions appropriately. Operator preconditions in GIPS contain only the literals from the execution concept that GIPS has decided are very necessary. This symbolic representation is used to post subgoals when the system wants to apply an operator that it believes cannot yet be executed. Recall that a TRANSFORM includes literals representing the current goals, and these are matched against selection concepts for the operators. Thus, changing operator preconditions can lead directly to subsequent changes in the selection of operators while solving problems.

Logically, GIPS should include in its preconditions for an operator exactly the literals that are highly necessary (i.e., have very low values for N_L). In problem-solving terms, all the literals in the preconditions of an operator should be true (or matched) for the operator to be executable. Thus, it should add literals from the execution concept that have low N_L values to the preconditions, and it should drop literals that do not have low N_L values from the preconditions. However, in order to speed up GIPS' learning, we have adopted a heuristic approach for each of these cases.

First, consider the criterion for adding a new literal to the preconditions of an operator. Again, GIPS should ideally consider this action for any literal with a low value for N_L. An examination of the equation for N_L shows that it decreases as $P(L$ does not match $I \mid I \notin C)$ increases. Learning about necessity poses some difficulties, because GIPS can increase its estimate of $P(L$ does not match $I \mid I \notin C)$ only when it predicts that the operator associated with C should execute, but the user tells the system that it made an incorrect prediction (an error of commission). However, GIPS is generally conservative in attempting to execute operators, so this type of event is relatively rare. Thus, GIPS takes a long time to learn that any new literal is necessary for execution. To overcome this difficulty, we allow GIPS to use a different criterion for adding preconditions. Rather than looking for literals that are very necessary, it looks for literals that are somewhat sufficient (i.e., have relatively high values for S_L). Mathematically speaking, sufficiency is not a valid predictor of new preconditions, but it does have some heuristic value, because literals that are very necessary are also always somewhat sufficient (if not very sufficient). This heuristic can encourage GIPS to make errors of commission, and thus learn whether the new literal really is necessary to the execution concept.

Now let us consider the case of dropping a literal from the preconditions of an operator when its value for N_L becomes too big. Again looking at the equation for N_L, we see that N_L increases as $P(L$ does not match $I \mid I \in C)$ increases. This corresponds to the case where GIPS correctly predicts that the operator associated with C should execute, but L does not appear. Intuitively, this means that L is not necessary for the operator to execute, so we have some evidence that it should be removed from the preconditions. As far as the system is concerned, it is learning that there is no reason to subgoal on a literal if it is not actually a necessary precondition for the operator.

A "proper" implementation of the STAGGER algorithm for this case would increment the estimate for $P(L$ does not match $I \mid I \in C)$ slightly every time the operator successfully executes "early." Thus, it would slowly gather evidence that L is not necessary, and it would eventually delete L from the preconditions of the operator. The algorithm requires substantial evidence before it will do this, because it must learn that L really is unnecessary and that the system has not simply encountered a noisy instance. In order to speed up GIPS' learning of preconditions, we assume that the feedback on operator execution from the user is always correct (i.e., not noisy). This allows us to bias GIPS to increase $P(L$ does not match $I \mid I \in C)$ by a *large* value for this type of instance rather than just by a small increment. Thus, GIPS can drop a literal, L, from its preconditions for an operator the *first time* it successfully executes that operator without the presence of L, rather than waiting for a large number of confirming experiences.

4. Strategy acquisition in the addition domain

This section presents GIPS' behavior through a series of different strategies for adding numbers. These strategy shifts arise from the learning algorithm incorporated into the system, and they correspond well with the shifts observed by Siegler and Jenkins. Siegler and Jenkins classified their subjects' behavior into eight strategies, of which four were based on counting (the others involved various kinds of recognition and guessing, which GIPS does not model). In this section, we describe each of the four counting strategies in the order in which they generally appear. However, it is important to note that children always intermingle their strategies, sometimes even on a trial-by-trial basis. We will discuss the issue of strategy variability in the following section.

4.1. The Sum strategy

GIPS' initial strategy for addition is the SUM strategy. To better follow GIPS' behavior, we have provided a trace of the sum strategy for the problem $2 + 3$ in Table 8. In this trace, we have omitted calls to the `Transform` function, only listing when operators are selected to apply and when they actually execute. The first thing the system does is assign an addend to each hand. For example, when adding 2 and 3, the system may assign the number 2 to the left hand and the number 3 to the right hand. However, in this strategy the order of the addends does not make a difference, so it could just as easily have switched them.

Next, the system begins its procedure of raising and counting a set of fingers on each hand. To accomplish this task, the ADDEND-REPRESENTED operators use a counter to determine when an addend is finished being represented on each hand (see Table 4(a)). For example, the preconditions of LEFT-ADDEND-REPRESENTED demand that the system be raising fingers on the left hand, and that the value of the counter be equal to the value of the left-hand addend. These preconditions are set up as subgoals, causing the selection of the START-RAISE and START-COUNT operators, which initialize the forward-chaining procedure of raising and counting fingers one at a time. These operators execute alternately until LEFT-ADDEND-REPRESENTED can execute, when the correct number of fingers have been counted on the left hand.

After the left hand has been counted, the CLOBBER-COUNTER operator immediately executes. This operator executes when all the fingers of a hand have been raised along with a running count. Its effects are to zero the value of the counter to prepare it for the next hand, and to mark the current hand as *uncounted*, because the counter's value has been changed. This entire procedure then repeats with the right hand.

After both hands have been counted, DETERMINE-ANSWER checks whether it can execute. It can only execute if both hands are marked as *counted*, but CLOBBER-COUNTER has caused this to be false. Therefore, the system again attempts to count up fingers on each hand, this time marking fingers that are already raised. For this procedure, no CLOBBER-COUNTER is necessary, because the number of raised fingers (rather than the value of the counter) is used to terminate the count for each hand. Finally, after each hand has been counted for the second time, GIPS announces the answer.

As the system repeatedly solves addition problems, it continuously updates the execution concepts for the two ADDEND-REPRESENTED operators. After a while, these two concepts encode several regularities that are always true when these operators execute. For example, there are always two addends in the problem description, and the number of "marked" fingers is always zero. Most importantly, however, the concepts encode the number of raised fingers as always equal to the counter value (which in turn is equal to the goal value for counting an addend). Literals representing this fact eventually get added into the preconditions for the ADDEND-REPRESENTED operators (see Table 4(b)). This action alone does not change the system's outward behavior, but it proves important for later strategies.

Table 8. GIPS' initial SUM strategy for addition.

```
Apply SELECT-HAND
      (LeftHand, Two, RightHand, Three)
Execute SELECT-HAND
      (LeftHand, Two, RightHand, Three)
Apply DETERMINE-ANSWER (?Answer)
  Apply COUNT-OUT-LEFTHAND (?Value70)
    Apply LEFT-ADDEND-REPRESENTED
          (?Value70)
      Apply START-RAISE (LeftHand)
      Execute START-RAISE (LeftHand)
      Apply START-COUNT (LeftHand)
      Execute START-COUNT (LeftHand)
      Apply RAISE-FINGER (LeftHand, 1)
      Execute RAISE-FINGER (LeftHand, 1)
      Apply COUNT (1)
      Execute COUNT (1)
      Apply RAISE-FINGER (LeftHand, 2)
      Execute RAISE-FINGER (LeftHand, 2)
      Apply COUNT (2)
      Execute COUNT (2)
    Execute LEFT-ADDEND-REPRESENTED (2)
  Execute COUNT-OUT-LEFTHAND (2)
  Apply CLOBBER-COUNTER
  Execute CLOBBER-COUNTER
  Apply COUNT-OUT-RIGHTHAND (?Value120)
    Apply RIGHT-ADDEND-REPRESENTED
          (?Value120)
      Apply START-RAISE (RightHand)
      Execute START-RAISE (RightHand)
      Apply START-COUNT (RightHand)
      Execute START-COUNT (RightHand)
      Apply RAISE-FINGER (RightHand, 1)
      Execute RAISE-FINGER (RightHand, 1)
      Apply COUNT (1)
      Execute COUNT (1)
      Apply RAISE-FINGER (RightHand, 2)
      Execute RAISE-FINGER (RightHand, 2)
      Apply COUNT (2)
      Execute COUNT (2)
      Apply RAISE-FINGER (RightHand, 3)
      Execute RAISE-FINGER (RightHand, 3)
      Apply COUNT (3)
      Execute COUNT (3)
    Execute RIGHT-ADDEND-REPRESENTED (3)
  Execute COUNT-OUT-RIGHTHAND (3)
  Apply COUNT-UP-BOTH-ADDENDS (?Answer)
    Apply CLOBBER-COUNTER
    Execute CLOBBER-COUNTER
    Apply COUNT-OUT-LEFTHAND (?Value158)
      Apply END-MARK-COUNT
            (LeftHand, ?Value158)
        Apply START-MARK-COUNT (LeftHand)
        Execute START-MARK-COUNT (LeftHand)
        Apply MARK-FINGER (LeftHand, 1)
        Execute MARK-FINGER (LeftHand, 1)
        Apply MARK-COUNT (1)
        Execute MARK-COUNT (1)
        Apply MARK-FINGER (LeftHand, 2)
        Execute MARK-FINGER (LeftHand, 2)
        Apply MARK-COUNT (2)
        Execute MARK-COUNT (2)
      Execute END-MARK-COUNT (LeftHand, 2)
    Execute COUNT-OUT-LEFTHAND (2)
    Apply COUNT-OUT-RIGHTHAND (?Value208)
      Apply END-MARK-COUNT
            (RightHand, ?Value208)
        Apply START-MARK-COUNT (RightHand)
        Execute START-MARK-COUNT (RightHand)
        Apply MARK-FINGER (RightHand, 1)
        Execute MARK-FINGER (RightHand, 1)
        Apply MARK-COUNT (3)
        Execute MARK-COUNT (3)
        Apply MARK-FINGER (RightHand, 2)
        Execute MARK-FINGER (RightHand, 2)
        Apply MARK-COUNT (4)
        Execute MARK-COUNT (4)
        Apply MARK-FINGER (RightHand, 3)
        Execute MARK-FINGER (RightHand, 3)
        Apply MARK-COUNT (5)
        Execute MARK-COUNT (5)
      Execute END-MARK-COUNT (RightHand, 5)
    Execute COUNT-OUT-RIGHTHAND
          (RightHand, 5)
  Execute COUNT-UP-BOTH -ADDENDS (5)
Execute DETERMINE-ANSWER (5)
```

Table 9. GIPS' SHORTCUT SUM strategy.

```
Apply SELECT-HAND
(LeftHand, Two, RightHand, Three)
Execute SELECT-HAND
(LeftHand, Two, RightHand, Three)
Apply DETERMINE-ANSWER (?Answer)
  Apply COUNT-OUT-LEFTHAND (?Value988)
    Apply LEFT-ADDEND-REPRESENTED
                    (?Value988)
      Apply START-RAISE (LeftHand)
      Execute START-RAISE (LeftHand)
      Apply RAISE-FINGER (LeftHand, 1)
      Execute RAISE-FINGER (LeftHand, 1)
      Apply COUNT (1)
      Execute COUNT (1)
      Apply RAISE-FINGER (LeftHand, 2)
      Execute RAISE-FINGER (LeftHand, 2)
      Apply COUNT (2)
      Execute COUNT (2)
    Execute LEFT-ADDEND-REPRESENTED (2)
  Execute COUNT-OUT-LEFTHAND (2)
  Apply COUNT-OUT-RIGHTHAND (?Value1022)
  Apply RIGHT-ADDEND-REPRESENTED
                    (?Value1022)
      Apply START-RAISE (RightHand)
      Execute START-RAISE (RightHand)
      Apply RAISE-FINGER (RightHand, 1)
      Execute RAISE-FINGER (RightHand, 1)
      Apply COUNT (3)
      Execute COUNT (3)
      Apply RAISE-FINGER (RightHand, 2)
      Execute RAISE-FINGER (RightHand, 2)
      Apply COUNT (4)
      Execute COUNT (4)
      Apply RAISE-FINGER (RightHand, 3)
      Execute RAISE-FINGER (RightHand, 3)
      Apply COUNT (5)
      Execute COUNT (5)
    Execute RIGHT-ADDEND-REPRESENTED (3)
  Execute COUNT-OUT-RIGHTHAND (3)
  Apply COUNT-UP-BOTH-ADDENDS (?Answer)
  Execute COUNT-UP-BOTH-ADDENDS (5)
Execute DETERMINE-ANSWER (5)
```

4.2. The SHORTCUT SUM *strategy*

After the new preconditions have been added and a number of addition problems have been solved, the new literals in GIPS' execution concepts for LEFT-ADDEND-REPRESENTED and RIGHT-ADDEND-REPRESENTED become so strong that GIPS decides that the operators should execute when the number of fingers raised on a hand is equal to the goal value even though the system has not yet incremented its count for the last finger. It turns out that the system can successfully solve the addition problem even if it executes this operator prematurely, so it deletes the condition that the current counter value must be equal to the goal value in the preconditions of the ADDEND-REPRESENTED operators (see Table 4(c)).

This change has a direct effect on GIPS' behavior (see Table 9). When attempting to apply LEFT-ADDEND-REPRESENTED, the value of the counter no longer appears in the preconditions, so it is not posted as a subgoal. This means that the START-COUNT operator is no longer selected. Thus, a running count is still kept while raising fingers, but the counter is not marked for use as the termination criterion. This means that CLOBBER-COUNTER will not execute, and that leads to two changes in strategy. First, the counter is not reset to zero after counting the left hand, and counting continues from the left hand's final value. Second, the hands are not marked as *uncounted*, so there is no need to count up the raised fingers again after the two hands have initially been counted. This behavior corresponds to the SHORTCUT SUM strategy, which was invented by all eight of Siegler and Jenkins' subjects.

This representation assumes that the student can determine without counting when the number of raised fingers on either hand is equal to the goal value. For example, in adding 2 + 3, just after saying, "five," and raising a third finger on the right hand, the subject must *see* that the number of fingers on the right hand is equal to 3, the number of fingers that they intended to raise. Before they began their study, Siegler and Jenkins tested their subjects' ability to recognize small numbers of objects without counting, and all the subjects could perform the skill adequately. Thus, we represent it in GIPS as a primitive skill, or a simple literal.

4.3. The SHORTCUT MIN strategy

The next shift leads to an intermediate strategy between SHORTCUT SUM and MIN, which we call SHORTCUT MIN. Although Siegler and Jenkins do not classify SHORTCUT MIN as a distinct strategy from SHORTCUT SUM, they do note (p. 119) that some of their subjects begin to switch addends during SHORTCUT SUM so that they start counting with the larger addend on the left hand, rather than just picking whichever addend appears first in the problem. GIPS also accounts for this behavior.

An important feature of the SHORTCUT SUM strategy is that the problem solver's counter value is not equal to the number of fingers being raised on the right hand (i.e., the second hand). We hypothesize that this causes interference and subsequent failure. Such interference would not occur with the left hand, because the number of raised fingers in the SHORTCUT SUM strategy is always equal to the value of the counter for that hand. Unfortunately, interference is a phenomenon that GIPS does not yet model, so we were forced to simulate its effects. We assumed that interference between the value of the counter and the number of fingers raised on the right hand would cause a child to become confused and fail to solve the current problem. This behavior is confirmed by Siegler and Jenkins: "when children used the shortcut-sum approach, they were considerably more accurate on problems where the first addend was larger than on ones where the second addend was" (p. 71).

We simulated this process by causing GIPS to fail sometimes during the SHORTCUT SUM strategy when it decided to count the larger addend on its right hand. These failures caused the system to update its selection concept for the SELECT-HAND operator. Thus, GIPS learned to prefer assigning the larger of the two addends to its left hand (information on the relative sizes of the addends was explicitly included in the state representation). Note that the learning algorithm does not require a model of interference to make this strategy shift. It simply records the fact that failure is somewhat correlated with assigning a larger addend to the right hand.

4.4. The MIN strategy

The final strategy shift occurs in a similar manner to the shift from SUM to SHORTCUT SUM. At this point, GIPS has attempted to execute the ADDEND-REPRESENTED operators at various times and has been given feedback each time as to whether it would be able to solve the current problem if it executed the operator at that time. Thus, it is slowly learning a "good" concept for when the ADDEND-REPRESENTED operators are executable. One of the things that proves to be true every time these operators execute is that the goal value for counting out a hand is equal to the addend assigned to that hand.

Table 10. GIPS' MIN strategy.

```
Apply SELECT-HAND
   (LeftHand, Three,
     RightHand, Two)
Execute SELECT-HAND
       (LeftHand, Three,
         RightHand, Two)
Apply DETERMINE-ANSWER (?Answer)
  Apply COUNT-OUT-LEFTHAND
                    (?Value1498)
   Apply LEFT-ADDEND-REPRESENTED
                    (?Value1498)
    Apply START-RAISE
                  (LeftHand)
    Execute START-RAISE
                  (LeftHand)
 Execute LEFT-ADDEND-REPRESENTED(3)
 Execute COUNT-OUT-LEFTHAND (3)
 Apply COUNT-OUT-RIGHTHAND
                     (?Value1516)
   Apply RIGHT-ADDEND-REPRESENTED
                     (?Value1516)
      Apply START-RAISE (RightHand)
      Execute START-RAISE (RightHand)
      Apply RAISE-FINGER
                  (RightHand, 1)
      Execute RAISE-FINGER
                  (RightHand, 1)
      Apply COUNT (4)
      Execute COUNT (4)
      Apply RAISE-FINGER
                  (RightHand, 2)
      Execute RAISE-FINGER
                  (RightHand, 2)
      Apply COUNT (5)
      Execute COUNT (5)
     Execute RIGHT-ADDEND-REPRESENTED(3)
    Execute COUNT-OUT-RIGHTHAND (3)
    Apply COUNT-UP-BOTH-ADDENDS
                          (?Answer)
    Execute COUNT-UP-BOTH-ADDENDS (5)
  Execute DETERMINE-ANSWER (5)
```

Eventually, the system attempts to execute the LEFT-ADDEND-REPRESENTED operator without having raised any fingers at all (see Table 10). When it succeeds in doing this, it deletes the precondition that the number of fingers raised on the hand be equal to the goal value (see Table 4(d)). The system has learned that it can simply start counting from the goal value for the left hand rather than starting from zero. Note that this behavior could not be generated directly from the initial SUM strategy, because it requires the counter to be used for counting the total sum, so it cannot aid in representing the right-hand addend. As with LEFT-ADDEND-REPRESENTED, GIPS also attempts to execute the RIGHT-ADDEND-REPRESENTED operator early, but this leads to failure. Thus, the system begins to exhibit the MIN strategy, in which the largest number (the left-hand number) is simply announced and used to continue counting the smaller number as in the SHORTCUT MIN strategy.

4.5. *The* FIRST *strategy*

The only other counting strategy identified by Siegler and Jenkins is the FIRST strategy. It was used on only six trials, all by the same subject. FIRST is similar to the MIN strategy, except that it does not assign the larger addend to the left hand. Rather, it starts with whichever addend is presented first, and continues counting with the second. In GIPS, this strategy follows from the SHORTCUT SUM strategy when the system does not encounter interference, and thus does not learn about ordering the addends. While using the FIRST strategy, the system can still eventually generate the MIN strategy through the same type of failure-driven learning that leads from SHORTCUT SUM to SHORTCUT MIN.

5. Summary and analysis

Both the SUM strategy and the MIN strategy have three main subgoals: to represent the first addend, to represent the second addend, and to count the union of the representations. The SUM-to-MIN *transition involves three independent modifications to the* SUM *strategy*:

1. The process of representing the addends is run in parallel with counting up the union. In the SUM strategy, representing the two addends must be completed before counting up the union begins.

2. The order of addends is made conditional on their sizes so that the larger addend is represented by the easier process. That is, any interference that occurs with the second addend is more likely to occur if the addend is large. Clearly, this strategy change must take place after the first one, because there is no interference when representing an addend and counting the union take place separately.

3. The subgoal of representing one addend changes from explicitly constructing a set of objects to simply saying the addend.

The GIPS account for each of these transitions is as follows. The first transition is caused by a combination of correlational learning of preconditions and search-control learning. Initially, GIPS represents an addend on a hand by raising fingers and counting until the counter's value is equal to the addend value. In two steps, the system learns that recognizing the number of fingers raised on a hand is a better stopping criterion than the value of the counter. When the value of the counter disappears as a subgoal of representing an addend, it is still free to be used to count the union of objects.

Because the counter is no longer required for representing addends, the question arises of why the system should continue to count as it raises fingers on each hand. This is an example of impasse-free search-control learning. The COUNT operator is responsible for incrementing the oral counter. Initially, it is selected only when the subgoal of counting up an addend is present. Eventually, correlated relations that are present in the current state (e.g., that a finger has just been raised) come to dominate the selection concept, and the operator becomes a forward-chaining operator. Basically, people have developed the habit of counting whenever they raise a finger even if that count doesn't serve any direct purpose. The combination of this habit with learning of new preconditions causes GIPS serendipitously to achieve the goal of counting the union as it deliberately represents the second addend.

Although GIPS can learn this habit, we gave COUNT a forward-chaining selection concept in this experiment because there is no direct evidence that the subjects did not learn this behavior even *before* they learned the SUM strategy (e.g., when they learned to count on their hands). However, GIPS predicts that subjects that do not have this habit could generate a fifth strategy, which we call the LAYOUT SUM strategy. In this strategy (which GIPS successfully generates), we would expect to see subjects silently represent an addend on each hand and then count up the union of fingers aloud. Siegler and Jenkins did not explicitly report observing this strategy, but it is possible that they included this behavior as a form of the SUM strategy.

The second transition from SUM to MIN is caused by failure-driven, search-control learning. Given two apparently equivalent methods, persistent errors in one of them causes the other one eventually to dominate. Because the errors are correlated with the relative sizes of the addends, GIPS learns that this is the relevant attribute upon which to base its decision of which hands to represent addends on.

The third transition once again involves impasse-free correlational learning at the knowledge level. GIPS keeps track of which literals in the situation are correlated with the final achievement of the goal of representing the first addend. Eventually, it considers these correlated literals to be just as essential as the originally specified preconditions. It eventually discovers that the originally specified preconditions can be ignored as long as the correlated literals are achieved. This summary makes clear that impasse-free, correlational learning of operator preconditions is crucial to the GIPS account for the first and third transitions. Ordinary symbol-level learning can handle the second.

6. Discussion

The GIPS analysis helps clarify several important, general issues about strategy change. Siegler and Jenkins observe that, "Not one child adopted a strategy that could be classified as indicating a lack of understanding of the goals of addition" (p. 107). In this respect, the subjects are similar to those of Gelman and Gallistel (1978), who found that very young children would invent correct strategies for counting a set of objects even when unusual constraints were placed on them to thwart their normal strategy. Gelman and Gallistel explain this remarkable competence by hypothesizing that children possess innate (or at least predetermined) principles of numerousity. Although linguists had earlier proposed the existence of innate constraints on language development, Gelman and Gallistel provided the first empirical evidence of innate constraints on non-linguistic development. This set off a heated debate in the developmental community. Siegler and Jenkins (p. 115) suggest that such constraints may exist on the development of addition strategies, but they do not give a specific list.

The initial knowledge given to GIPS does not involve any explicit principles of addition or counting. It is merely a set of operators and selection preferences that happen to generate the correct answers. It is able to avoid developing bad strategies because of the feedback it receives while solving problems. GIPS occasionally attempts to execute an operator in situations that would produce a wrong answer. If it were not told that the execution was wrong, it would develop wrong strategies. Thus, GIPS suggests one possible account for learning without the hypothesized innate constraints.

However, as we have discussed, the feedback we provided to GIPS did not correspond exactly to the type of feedback that children usually receive. In the Siegler and Jenkins study, students were told after each trial whether they got the problem right, and then presented with a new problem. GIPS assumes that it can precisely assign blame to bad operators by receiving early feedback from the user or by searching until a correct solution can be found. A more proper model of Siegler and Jenkins' feedback would require the incorporation of an incremental credit-assignment algorithm. On the other hand, if GIPS *were* supplied with knowledge about constraints on numbers and arithmetic, it would be able to assign credit and blame as it solved problems (Ohlsson & Rees, 1991), rather than after receiving its "gold star" for each problem. Such constraints would generate behavior similar to GIPS' current implementation, providing the same type of immediate feedback that the user provided when GIPS attempted to execute operators inappropriately.

A common misconception about discovery is that the newly discovered strategy, concept, or idea instantly and totally supplants its predecessor. In all protocol-based studies of discovery (e.g., Kuhn, Amsel, & O'Laughlin, 1988; Siegler & Jenkins, 1989; VanLehn, 1991), the transition between the old strategy and the new one is gradual. For instance, Siegler and Jenkins (p. 73) report, "In the first five sessions after children discovered the min strategy, they used the strategy on only 12% of the trials in which they used any of the three counting strategies." We have not tried to model the gradual transition to the use of the MIN strategy with GIPS because doing it right would require implementing several memory-based strategies. However, it is clear that the probabilistic nature of GIPS' selection and execution concepts would tend to predict a gradual transition.

Starting in the eighth week of the study, Siegler and Jenkins began including "impasse problems," such as 2 + 23. They had hoped that these would encourage discovery of the MIN strategy, but they did not, for only one child first used the MIN strategy on an impasse problem. However, children who had already discovered the MIN strategy began to use it much more frequently on the impasse problems and even on the non-impasse problems that followed the eighth week. GIPS would tend to do the same thing if it were given impasse problems. The larger addend would invite errors during the SHORTCUT, SUM and FIRST strategies, which would lower the values of their selection concepts. The inclusion of impasse problems would not affect the error rate of the MIN strategy, so it would gradually become the preferred strategy for all counting trials.

Siegler and Jenkins noticed that some children seemed consciously aware that they had invented a new strategy in that they could explain it on the first trial where they used it, and some even recognized that it was a "smart answer," in the words of one child. Other children denied using the MIN strategy even when the videotape showed that they had used it. For instance, one child said, "I never counted ... I knew it ... I just blobbed around." Siegler and Jenkins divided children into those who seemed conscious of the strategy and those who did not, and measured the frequency of their subsequent usage of the MIN strategy. The high awareness group used the MIN strategy on about 60% of the trials where they used any counting strategy. The low awareness group used the MIN strategy on less than 10% of the trials. This suggests that being aware of a newly discovered strategy facilitates subsequent usage of it.

This finding cannot be modeled by GIPS because GIPS has no way to distinguish a strategy that can be explained from one that is inaccessible to consciousness. However, the finding could probably be modeled by combining GIPS with a system that uses an analytical learning algorithm. The basic idea is simple. GIPS would discover a new strategy just as it does now, and a trace of the strategy's actions would remain in memory. This trace would be used as an example that is explained by the analytical system. (Siegler and

Jenkins asked subjects after each problem to explain how they got their answer—a sort of mandatory reflection.) If enough of the trace can be recalled for the explanation to succeed, it annotates the steps in the trace and perhaps the operators whose executions produced the steps. These elaborations make it easier to retrieve the modified operators from memory, and they may help in assigning credit and blame, thus speeding the adjustment of the preconditions, selection, and execution concepts. These influences increase the usage of the new strategy on subsequent problems.

To conclude our discussion, we address a competing model of the SUM-to-MIN transition. Neches' (1987) HPM is designed to effect strategy changes whenever it detects an opportunity for improving the efficiency of a procedure. To achieve this, HPM stores a complete trace of its processing, and constantly monitors this memory with heuristics, such as "If a sub-procedure produces an output, but no other sub-procedure receives that result by the time the overall procedure finishes, then modify the overall procedure to eliminate the superfluous computation." Neches demonstrated that this heuristic and two others sufficed for changing the SUM strategy into the MIN strategy.

HPM had to produce two transitional strategies before it could get to MIN. Siegler and Jenkins sought evidence for these transitional strategies in their data. One of the strategies (the FIRST strategy) occurred six times, all in the protocol of one subject. Moreover, all of these instances occurred *after* the subject invented the MIN strategy, whereas HPM must invent it *before* it can get to MIN. The second transitional strategy predicted by Neches did not appear at all. These unfulfilled predictions cast doubt on the HPM model.

Another problem with HPM's account is that it requires the storage of an entire search tree over several problem-solving attempts. In contrast, GIPS only stores the trace of the current solution attempt and discards it after learning. As mentioned previously, it may be useful to explore the use of a different credit-assignment method in GIPS, such as the bucket-brigade algorithm (Holland et al., 1986). Such an algorithm could allow the system to avoid storing *any* of the solution path. As Neches noted, the HPM model "assumes the relative accessibility of extremely detailed information about both ongoing process and related past experiences. How can this be reconciled with known limitations on the ability to report this information?" (p. 213). Although HPM is computationally sufficient to produce the SUM-to-MIN transition, it makes dubious empirical and mnemonic assumptions.

To summarize, GIPS achieves its main research objective, providing a computational account of the several strategy shifts observed during the SUM-to-MIN transition. It uses plausible local processes, rather than the global optimization processes of Neches' HPM. In addition, GIPS uses modest amounts of storage, in contrast to HPM, which stores complete solution traces for indefinite periods. Most importantly, GIPS produces all and only the transitional strategies observed in the Siegler and Jenkins study. It predicts an additional possible strategy, but does not require it to occur before the invention of the MIN strategy.

The GIPS analysis solves a number of puzzles raised by Siegler and Jenkins' study. These include the source of the various strategies that appear in the SUM-to-MIN transition and their order of appearance, as well as the ability to make significant strategy shifts without impasse-driven learning. GIPS also suggests a role for innate knowledge of the principles of addition in the ability to avoid inventing bad strategies, although this depends on the specific type of feedback given to the system. Thus, GIPS provides a plausible, computationally sufficient account of the discovery of the MIN strategy. However, Siegler and Jenkins produced a second set of findings on the gradual increase in usage of the newly discovered strategy. We have not yet tried to model these findings, but GIPS seems to provide an appropriate framework for doing so.

Finally, the SUM-to-MIN transition does not appear to be explainable by conventional, symbol-level learning mechanisms. Rather, some of the important shifts require changes to the representation of the domain. GIPS models these changes by altering preconditions on some of its operators. Adjusting operator preconditions is somewhat dangerous, because it can allow the system to corrupt a previously correct domain theory, but GIPS demonstrates that such a mechanism can generate useful behavior shifts when controlled by feedback on its decisions.

Acknowledgments

This research benefited from discussions with Jeff Schlimmer and Bob Siegler. Comments from Clare Bates Congdon and the reviewers improved the technical quality of the paper. The research was supported in part by contract N00014-88-K-0080 from the Office of Naval Research, Cognitive Sciences Division, and a postdoctoral training grant from the Department of Health and Human Services.

References

Anderson, J.R. (1983). *The architecture of cognition.* Cambridge, MA: Harvard University Press.

Ashcraft, M.H. (1982). The development of mental arithmetic: A chronometric approach. *Developmental Review, 2*, 213–236.

Ashcraft, M.H. (1987). Children's knowledge of simple arithmetic: A developmental model and simulation. In C.J. Brainerd, R. Kail, and J. Bisanz (Eds.), *Formal methods in developmental psychology.* New York: Springer-Verlag.

Berwick, R. (1985). *The acquisition of syntactic knowledge.* Cambridge, MA: MIT Press.

Dietterich, T.G. (1986). Learning at the knowledge level. *Machine Learning, 1*, 287–316.

Fikes, R.E., and Nilsson, N.J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence, 2*, 189–208.

Gelman, R., and Gallistel, C.R. (1978). *The child's understanding of number.* Cambridge, MA: Harvard University Press.

Groen, G.J., and Parkman, J.M. (1972). A chronometric analysis of simple addition. *Psychological Review, 79*, 329–343.

Groen, G.J., and Resnick, L.B. (1977). Can preschool children invent addition algorithms? *Journal of Educational Psychology, 69*, 645–652.

Holland, J.H., Holyoak, K.J., Nisbett, R.E., and Thagard, P.R. (1986). *Induction: Processes of inference, learning, and discovery.* Cambridge, MA: MIT Press.

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

Jones, R.M. (1993). Problem solving via analogical retrieval and analogical search control. In A. L. Meyrowitz and S. Chipman (Eds.), *Foundations of knowledge acquisition: Machine learning.* Boston: Kluwer Academic.

Kaye, D.B., Post, T.A., Hall, V.C., and Dineen, J.T. (1986). The emergence of information retrieval strategies in numerical cognition: A development study. *Cognition and Instruction, 3*, 137–166.

Korf, R.E. (1985). Macro-operators: A weak method for learning. *Artificial Intelligence, 26*, 35–77.

Kuhn, D., Amsel, E., and O'Laughlin, M. (1988). *The development of scientific thinking skills*. New York: Academic Press.

Langley, P. (1985). Learning to search: From weak methods to domain-specific heuristics. *Cognitive Science, 9*, 217–260.

Langley, P. and Allen, J.A. (1991). The acquisition of human planning expertise. In L.A. Birnbaum and G.C. Collins (Eds.), *Machine Learning: Proceedings of the Eighth International Workshop* (pp. 80–84). Los Altos, CA: Morgan Kaufmann.

Minton, S. (1988). *Learning effective search control knowledge: An explanation-based approach*. Boston: Kluwer Academic.

Mitchell, T.M., Utgoff, P.E., and Banerji, R. (1983). Learning by experimentation: Acquiring and refining problem-solving heuristics. In R.S. Michalski, J.G. Carbonell, and T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach*. Los Altos, CA: Morgan Kaufmann.

Neches, R. (1987). Learning through incremental refinement of procedures. In D. Klahr, P. Langley, and R. Neches (Eds.), *Production system models of learning and development*. Cambridge, MA: MIT Press.

Newell, A. (1990). *Unified theories of cognition: The William James lectures*. Cambridge, MA: Harvard University Press.

Ohlsson, S. and Rees, E. (1991). The function of conceptual understanding in learning of arithmetic procedures. *Cognition and Instruction, 8*, 103–179.

Pazzani, M., Dyer, M., and Flowers, M. (1986). The role of prior causal theories in generalization. *Proceedings of the Fifth National Conference on Artificial Intelligence* (pp. 545–550). Los Altos, CA: Morgan Kaufmann.

Schank, R. (1986). *Explanation patterns: Understanding mechanically and creatively*. Hillsdale, NJ: Lawrence Erlbaum.

Schlimmer, J.C. (1987). Incremental adjustment of representations for learning. *Proceedings of the Fourth International Workshop on Machine Learning* (pp. 79–90), Los Altos, CA: Morgan Kaufmann.

Schlimmer, J.C. and Granger, R.H., Jr. (1986a). Beyond incremental processing: Tracking concept drift. *Proceedings of the Fifth National Conference on Artificial Intelligence* (pp. 502–507). Los Altos, CA: Morgan Kaufmann.

Schlimmer, J.C. and Granger, R.H., Jr. (1986b). Incremental learning from noisy data. *Machine Learning, 1*, 317–354.

Siegler, R.S. and Jenkins, E. (1989). *How children discover new strategies*. Hillsdale, NJ: Lawrence Erlbaum.

Svenson, O. (1975). Analysis of time required by children for simple additions. *Acta Psychologica, 39*, 289–302.

VanLehn, K. (1990). *Mind bugs: The origins of procedural misconceptions*. Cambridge, MA: MIT Press.

VanLehn, K. (1991). Rule acquisition events in the discovery of problem solving strategies. *Cognitive Science, 15*, 1–47.

VanLehn, K., and Jones, R.M. (1993). Integration of explanation-based learning of correctness and analogical search control. In S. Minton and P. Langley (Eds.), *Planning, scheduling and learning*. Los Altos, CA: Morgan Kaufmann.

VanLehn, K., Jones, R.M., and Chi, M.T.H. (1992). A model of the self-explanation effect. *Journal of the Learning Sciences, 2(1)*, 1–60.

Received April 2, 1992
Final Manuscript January 18, 1994