The Test Incorporation Theory of Problem Solving (Preliminary Report)

Thomas G. Dietterich
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331

James S. Bennett
Teknowledge, Inc.
1850 Embarcadero Road
Palo Alto, California 94303

Abstract:

Test incorporation is a program transformation in which a generate-and-test problem solver is improved by moving information out of the test and into the generator. This paper sketches a theory of problem solving based on test incorporation. Two views of test incorporation are presented: (a) as a compile-time algorithm optimization and (b) as a run-time problem-solving method. The paper focuses on the latter, which is termed the "algebraic view," because it is often the case that test incorporations can be applied to "solve" for the desired answer thus eliminating any need to generate and test possible answers. The theory introduces an infinite tower of meta-level problem solvers, each of which has the task of improving the performance (via test incorporation) of all of the problem solvers "below" it. This infinite tower can neither be constructed in principle nor in practice, but it provides a kind of infinite-series expansion of any given problem solver. Several familiar AI methods are reconstructed as "residual" problem solvers—the remnants of the infinite tower of problem solvers after the meta-level problem solvers have done their tasks and been "compiled away." The test incorporation process is a means by which knowledge is effectively exploited to yield efficient performance. This observation leads naturally to a definition of "intelligence" as the ability to perform test incorporations. It is asserted that any intelligent system must have some ability to perform test incorporations, and test incorporation methods will be critical to the development of generally intelligent systems.

1 Introduction

Test incorporation is a program transformation that can be applied to generate-and-test problem solvers to improve their performance. The basic notion is familiar to all programmers. Rather than generating a set of candidates and then filtering them to eliminate non-solutions, it is often possible to modify the generator of candidates so that some of the conditions in the "test" are automatically satisfied. Ideally, *all* of the test information can be "incorporated" into the generator. The result is an algorithm that directly computes the solutions.

Test incorporation is a very attractive program transformation for several reasons. First, *it is incremental.* Hence, it allows us to gradually improve algorithms, and it can be applied to discover new algorithms (see Tappel, 1980). Second, *it is performance-aligned*—that is, it always improves (or at least does not degrade) the performance of the algorithm, and this performance improvement is easy to compute. Third, *it affects program modifiability.* As long as the test is a separate part of the problem solver, it is easy to modify the problem solver to solve a different problem. If the generator is generating a sufficiently large set of candidates, we can simply change the test to select different solutions. However, after test incorporation, the separation between generator and test is lost, and the problem solver is harder to modify (see Newell, 1969). Finally, *test incorporation is knowledge-aligned.* Test incorporation can be viewed as the transfer of knowledge out of the "test" and into the "generator." Hence, test incorporation is a fundamental way in which knowledge comes to be effectively applied in problem solving.

The purpose of this paper is to sketch a theory of AI problem solving methods based on the idea of test incorporation. We begin by presenting a simple example of the application of test incorporations to develop an efficient algorithm. This can be viewed as a kind of knowledge compiling process in which domain knowledge in the "test" is compiled into the "generator." Several issues arise here including the need to *factor* the generator and the test and the need to exploit additional domain knowledge during algorithm design. We will show that not all algorithms can be constructed via a sequence of test incorporations.

In order to extend test incorporation theory, we must shift our view of incorporation as a kind of compile-time program transformation and instead view it as a form of problem solving activity itself. This leads us to consider test incorporation as a kind of algebraic simplification of the problem solver. It permits us to derive a wider class of AI methods including means-ends analysis, data-driven algorithms, and goal-driven algorithms. However, it also raises the issue of how test incorporations are performed.

In the third section of the paper, we resolve this question by describing a meta-level Incorporation Problem Solver (IPS) whose task is to improve a base-level problem solver. The IPS searches a space of possible improved base-level problem solvers, looking for the most efficient one. Test incorporations provide a set of operators for moving around in this space. A straight-forward implementation of the IPS simply pursues a greedy (hill-climbing) strategy.

One concern about the IPS is that, although it is improving the base level problem solver, it may consume so much time itself that the overall problem solving process is slowed rather than accelerated. This concern leads us to consider a meta-IPS whose task is to improve the combined efficiency of the IPS and the base-level problem solver. The meta-IPS modifies the IPS so that it does not perform incorporations unless the cost of carrying out those incorporations is offset by the improved performance of the base-level problem solver.

In general, our analysis proposes an infinite tower of meta-level problem solvers, each of which has the task of improving all of the levels below it. In practice, this infinite tower is never constructed, but it provides us with a kind of "series expansion" of an ideal AI problem solving system. Particular systems can be viewed as "truncated" approximations to this infinite series.

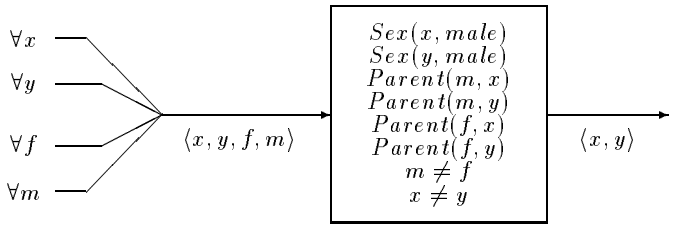

Figure 1: Naive generate-and-test algorithm for full brothers

Finally, this paper concludes with a discussion of the role of test incorporation theory in developing a theory of intelligence. Briefly, intelligent behavior has two fundamental requirements: knowledge and the means to effectively apply that knowledge. A generate-and-test problem solver contains in its test all of the knowledge needed to solve the problem. Intelligence consists of the process by which this knowledge is converted into effective action. In short, intelligence is the process of test incorporation.

2 An example of test incorporation

Consider the following problem. A problem solver contains a family tree database in which facts of the form $Parent(x, y)$ (x is a parent of y) and $Sex(x, s)$ (the sex of x is s) are stored. The task of the problem solver is to find all pairs of full brothers. In formal terms, the task is to find all pairs $\langle x, y \rangle$ such that

$$\exists m, f \; Sex(x, male) \wedge Sex(y, male) \wedge Parent(m, x) \wedge Parent(m, y) \wedge$$
$$Parent(f, x) \wedge Parent(f, y) \wedge m \neq f \wedge x \neq y$$

In other words, the program must find two distinct individuals (the mother m and father f) such that each is a parent of both x and y.

We can imagine a wide variety of algorithms for solving this problem. The most elementary algorithm (shown in Figure 1) would generate four-tuples of the form $\langle x, y, m, f \rangle$ and then verify that the indicated relationships hold. The simplicity (and inefficiency) of this algorithm leads us to call it the *naive* generate-and-test algorithm.

The naive generate-and-test algorithm uses virtually all of its knowledge about the problem as its "test." The generator simply enumerates all known four-tuples of objects. *Test incorporation* is the process of incorporating some of the test knowledge into the generator.

For example, suppose that our database representation is capable of generating the set of all x such that $Sex(x, male)$. In that case, we can incorporate this test into the generator and produce four-tuples $\langle x, y, m, f \rangle$ where each x is guaranteed to be a male (see Figure 2). We no longer need to test the sex of x in the test part. More importantly, there is a combinatorial time savings because we do not bother to construct four-tuples containing x values that are not males. Hence, we don't waste time testing any of the other properties (such as $Parent(f, y)$) on any of these tuples.

A more powerful kind of test incorporation can be performed by cascading generators. For example, suppose that the database is also capable of generating the set of all f such that $Parent(f, x)$ given a specific value for x. Then we can incorporate the $Parent(f, x)$ test into the generator by first generating all x such that $Sex(x, male)$ and then using these x values to generate all of the

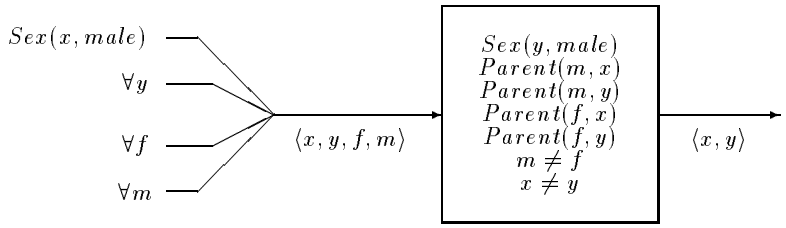

Figure 2: Incorporating $Sex(x, male)$ into the generator

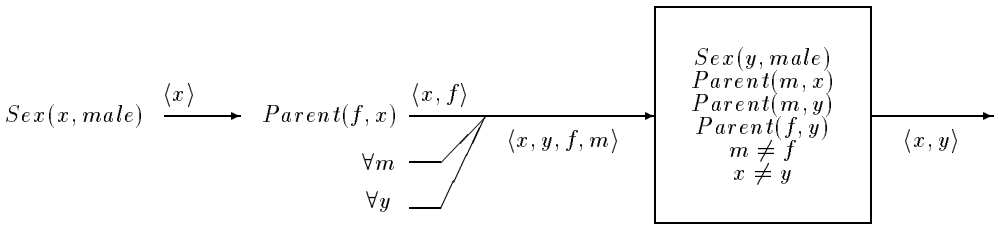

Figure 3: Cascading two generators

parents f of x (see Figure 3). This is a tremendous improvement because we are using parts of the "test" to incrementally construct the desired solution. This strategy of problem solving is sometimes called "seed growth," because we start with a seed (x) and use it to directly compute additional parts of the solution, gradually growing it into a complete four-tuple.[1]

If we can incorporate $Parent(f, x)$, we can certainly incorporate $Parent(m, x)$ in the same way to generate all tuples of the form $\langle x, f, m \rangle$ that automatically satisfy the properties $Sex(x, male) \wedge Parent(f, x) \wedge Parent(m, x)$. This produces the algorithm of Figure 4.

The next incorporation is very interesting. Whenever we want to generate pairs of distinct objects (i.e., f and m such that $m \neq f$) from a set, we can employ what might be called a *triangle*

[1] The island-driving technique developed in HEARSAY-II is another example of a seed growth algorithm.

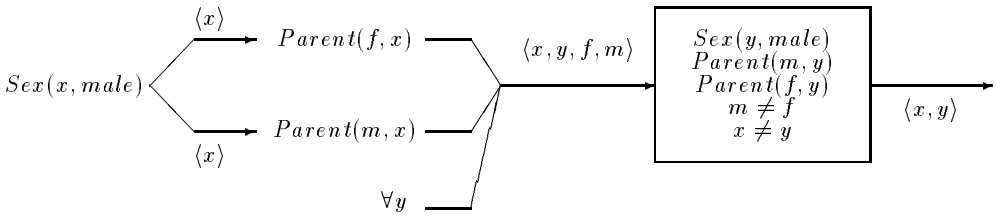

Figure 4: Algorithm that incorporates the ability to generate parents of x.

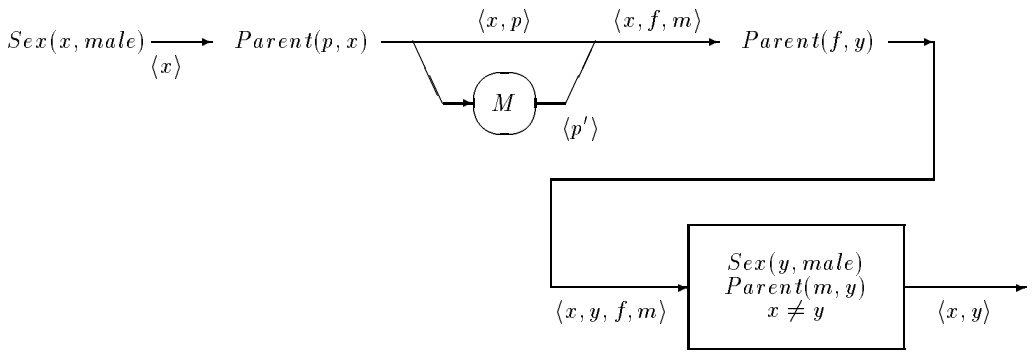

Figure 5: Final generator for finding all brothers

generator. We can incorporate the constraint $m \neq f$ by using only one generator of parents, $Parent(p, x)$, and remembering all of the parents that have been generated thus far. When each new parent p is generated, we consider all of the previously generated parents p' and produce pairs of the form $\langle p, p' \rangle$. Here is the code for this triangle generator:

```
memory := nil
for p such that parent(p,x) do
   for p' in memory do generate(<p,p'>)
   add p to memory
```

Because we only consider the parents generated thus far, we never generate pairs where $p = p'$. The name *triangle generator* is taken from the idea of generating only the lower-triangular region of a two-dimensional array.

There is only one remaining incorporation that we can perform to improve this algorithm—namely, to use the $Parent(f, y)$ generator to generate the remaining children of f. Once we have generated these children, the remaining three conjuncts, $Sex(y, male) \wedge Parent(m, y) \wedge x \neq y$ must be tested. There is no way to incorporate any of these into the generator at this point. Figure 5 shows the final algorithm that we have developed.

The kind of reasoning that we have been describing is familiar to most computer scientists. Several important AI systems have been designed around the notion of constraining a generator of possibilities. For example, the heart of the DENDRAL system (Lindsay, Buchanan, Feigenbaum & Lederberg, 1980) is CONGEN—a generator of all possible molecular structures consistent with a given set of constraints. If only the chemical formula of the molecule is given to CONGEN, it typically generates millions of possible structures. The secret to DENDRAL's success was to analyze the mass spectrum and extract constraints that could be incorporated into CONGEN.

In addition to using the idea of test incorporation to guide the development of particular AI systems, some researchers have attempted to formalize and implement general-purpose systems for performing test incorporation. The simplest example of this is the work on Prolog compilers, especially for parallel logic programs (Shapiro, 1986). A Prolog program can be viewed as a particular cascade of generators and tests. Some compilers consider re-arranging the order of generation and testing in order to improve performance. Related theoretical analyses of this problem include Simon & Kadane (1975) and Smith & Genesereth (1985). Cohen (1986) presents a system for compiling a more general class of logical queries. Mostow (1983a,b) describes a system for automatically

improving a heuristic search procedure via test incorporation. Tappel (1980) demonstrated how test incorporation can be employed to design new algorithms.

What is involved in the test incorporation process? There are three basic steps:

1. Identify some subpart of the test that is a candidate for incorporation.

2. Identify some subpart of the generator to which the necessary code can be added.

3. Perform the incorporation.

There are several things to note about this process. First, it requires access to the internal structure of the generator and the test. The traditional illustration of this is the example of the combination safe. Without knowledge of the combination, the only way to unlock a perfect combination safe is to generate all possible combinations and test each one. We say in this situation that the "test" is not *factorable*. The converse case, in which the generator is unfactorable, arises as well. Consider an inflexible program for displaying bulletin-board messages (i.e., the generator). The reader of these messages can recognize which ones are relevant (i.e., the reader is the test). However, the reader can't communicate this information to the bulletin-board program, because its internal structure isn't available for modification. The reader must step through the messages one-by-one until the relevant message appears.

When we say that the generator is factorable, we mean that it can be broken into subgenerators that can then be "rewired" to incorporate additional tests. Similarly, the test is factorable if it can be broken into subtests. Consider again Figure 4. At this point, the generator can be factored to isolate the subgenerator of all f such that $Parent(f, x)$. Similarly, the test includes the subtest $m \neq f$. The availability of this subgenerator and subtest permit the addition of a triangle generator that incorporates the subtest into the generator.

Mere factoring of the generator and the test is not sufficient, however, to guarantee that test incorporations will be correct. The second important point concerning test incorporation is that the correctness of incorporation steps often depends on knowledge *about* the generator. For example, the introduction of the triangle generator will only succeed if the basic generator for $Parent(p, x)$ does not produce duplicates. If it does produce a duplicate, then the triangle generator will produce a pair of the form $\langle p, p' \rangle$ where $p = p'$, and hence will commit an error.

Finally, the most important point to note about test incorporation is that not all algorithms can be derived simply by a sequence of test incorporations. Test incorporations may fail to produce an algorithm either because the test does not contain all relevant knowledge or because the test must be reformulated before it can be incorporated.

As an example of the first problem, consider what algorithm would result if we were able to incorporate all knowledge in the brothers test into the generator. No matter what sequence of incorporations we perform, we cannot produce the algorithm that simply generates the first two parents of x and uses only them. This is because the test does not contain all knowledge about the world, but only knowledge relevant to recognizing a solution. In particular, the test does not contain the knowledge that all people have exactly two parents. Instead, the test defines x and y to be full brothers if they share *at least* two parents. The knowledge that each person has only two parents can be incorporated into the algorithm to remove the triangle generator altogether and replace it with a generator that quits as soon as it has found two distinct parents.

As an example of the second problem, suppose that the database were represented in a different format in which there are entities called matings with the relations $Father(f, mat)$ (f is the father of mating mat), $Mother(m, mat)$ (m is the mother of mating mat), $Son(s, mat)$ (s is a son resulting from the mating mat) and $Daughter(d, mat)$ (d is a daughter resulting from the mating mat).

Suppose there are generators available for generating all matings and for generating all fathers, mothers, sons, or daughters in a given mating. If we are given only the original definition of full brothers in terms of $Parent$ and Sex, it will be impossible to perform any test incorporations without *reformulating* the test so that it corresponds to the available generators (see Amarel, 1968, 1983). If we had sufficient knowledge, we could reformulate the test to define x and y to be full brothers if there exists a mating mat such that $x \neq y \wedge Son(x, mat) \wedge Son(y, mat)$. It is easy to incorporate this test into the generators so that all matings are generated and then, for each mating, all sons are generated and distinct pairs of sons are produced using a triangle generator.

To conclude this section, let us summarize our results thus far. First, we have shown that test incorporation is a powerful program transformation, and that it can be applied incrementally to develop efficient algorithms. In addition to our example problem of computing full brothers, test incorporation has been applied to derive such algorithms as the Seive of Eratosthenes (Tappel, 1980), an efficient shortest-path algorithm (Tappel, 1980), the greedy algorithm for minimum spanning trees (Bennett & Dietterich, Forthcoming), quicksort (Smith, In Press), and the DENDRAL system (Bennett & Dietterich, Forthcoming).

Second, we have noted three important aspects of test incorporation: (a) test incorporation requires the ability to factor the generator and the test, (b) correctness of incorporations may rely on knowledge about the generator (such as order of generation or presence of duplicates) beyond simply knowing what set it generates, and (c) test incorporation is not capable of deriving all possible algorithms (without additional world knowledge and the ability to perform reformulations).

3 Run-time Test Incorporation

In the previous section, we considered test incorporation as a kind of compile-time optimization step. However, it is not always possible, at compile-time, to perform all of the beneficial test incorporations. This is because in many AI problem solvers, the complete "test" is not provided all at once, but instead parts of it are gradually revealed to the system.

One of the best examples of this is the candidate-elimination algorithm for concept learning (Mitchell, 1978). The naive generator in this task generates concept definitions from some space of possible concepts. The test checks to see if each candidate concept is consistent with all of the *possible* training instances (i.e., it classifies the positive instances as positive and the negative instances as negative). The main difficulty is that not all of the possible training instances are known to the program. Instead, training instances are presented to the program one at a time. Mitchell's version space algorithm is able to incorporate each of these "pieces" of the test into the generator by representing the space of possible concepts as an interval in a partially-ordered set and by moving the boundaries of this interval. At any point, the generator can be run, and it will generate only the concept definitions that are in the interval—that is, all of the concept definitions that would classify the *observed* training instances correctly.[2]

A closely related example is the game of "guess-the-number." In this game, the player must guess a number between 1 and 100. Whenever the player makes a guess, an oracle indicates whether the guess is correct, too low, or too high. A naive generate-and-test algorithm simply generates all possible numbers and depends on the oracle to test each one. However, a more clever algorithm guesses a number and then incorporates the oracle's answer into the generator by employing what Doug Smith (In press) calls a *subspace generator.* An example of a subspace generator is one that can generate all elements within an interval where the endpoints of the interval are inputs to the

[2]In practice, this generator is never run, and hence, it is never even implemented. However, it would not be difficult to write.

generator. The initial generator of all numbers between 1 and 100 can be implemented by giving the subspace generator the interval [1, 100]. Suppose the first generated guess is 20 and the oracle answers "too low." Then this answer is incorporated, at run time, into the generator by changing the interval to [21, 100]. This process continues until the interval is a single point.[3]

In both the version space algorithm and the guess-the-number algorithm, pieces of test information are made available over time. Efficient algorithms incorporate this test information into the generator immediately in the hope that eventually the generator will be eliminated altogether and the answer will be determined. We call this view of run-time test incorporation the *algebraic view*, by analogy with the process of solving a set of algebraic equations.

Consider, for example, the following pair of equations:

$$x + 2y = 7$$

$$x - y = 1$$

A naive generate-and-test problem solver might generate pairs of integers $\langle x, y \rangle$, substitute them into the equations, and check whether they are solutions. However, by performing test incorporations, we can completely solve the problem. From the second equation, we know $x = 1 + y$, which we can substitute into the first equation to obtain $1 + 3y = 7$. This changes the problem solver so that it (a) generates possible values for y, (b) tests them to see if $1 + 3y = 7$, and then (c) computes x from $1 + y$, and produces the solution $\langle x, y \rangle$. Hence, the original two equations been incorporated into the generator, but there is now a generate-and-test algorithm *within* the generator that can be improved. By solving the equation $1 + 3y = 7$ for y, we obtain $y = 2$. Hence, the final algorithm performs no search whatsoever, it simply produces $y = 2$ and then computes $x = 3$.

From the algebraic perspective, every problem solver is analogous to a system of equations. Test incorporation is a kind of algebraic operator that allows us to simplify this system of equations and solve for the answer. The system of equations can't be solved, of course, unless all of the equations are known. At compile time, in cases like guess-the-number, not all of the "equations" (i.e., pieces of test information) are known. The run-time inputs to the problem solver can be viewed as additional equations that are being made available. When enough equations are known, the solution can be determined. From this perspective, we see that the distinction between run-time and compile-time is simply based on *when* the "equations" are made available.

Sometimes this algebraic view requires us to stretch our intuition. Consider the MYCIN system, for example. According to the algebraic view, the naive generate-and-test algorithm for MYCIN would operate by generating pairs of the form $\langle patientdescription, disease \rangle$ and testing them to see if the patient description matched the patient data supplied as input to the program at run time. A much better approach, of course, is to wait until the patient data is given to the program and then compute the associated disease. This way the patient data can be incorporated into the generator of plausible diseases.

The algebraic view also provides an explanation of the effectiveness of methods such as means-ends analysis. Consider the naive generate-and-test algorithm for planning a trip from Palo Alto, California to Cambridge, Massachusetts. It generates possible sequences of actions and then tests them to see if they get us from Palo Alto to Cambridge efficiently. Means-ends analysis operates by examining the goal and incorporating into the generator a step such as $Fly(SFO, BOS)$. The remaining tasks are to generate sequences of actions that get us from Palo Alto to San Francisco Airport (SFO) and sequences of actions that get us from Boston Airport (BOS) to Cambridge. The

[3]Notice that these incorporations are only possible because the oracle gives more information than simply whether the guess is correct. The answers "too high" and "too low" provide some insight into the test, and thus provide material for incorporation. The test is partially factored.

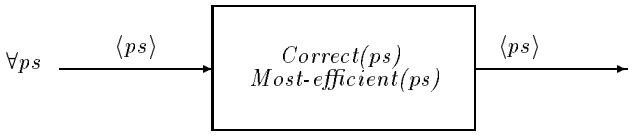

Figure 6: A naive generate-and-test IPS

incorporation of these goals into the generator takes into account not only the correctness of the plan (i.e., it gets us to the proper destination) but also the efficiency of the plan (i.e., it chooses flying rather than walking).

From these examples, we can see that the algebraic view of test incorporation—in which it is a form of run-time problem solving rather than simply a compile-time optimization—provides derivations for data-driven methods, goal-driven methods, and programs that accept input from the environment. Data and goals are both treated as pieces of test information that are incorporated at run-time. One question we have *not* dealt with, however, is *who is selecting and performing these incorporations?* The next section explores this question.

4 The Incorporation Problem Solver

In many of the examples we have discussed thus far (especially the full brothers example), test incorporations were identified and applied by the programmer as he or she designed an efficient algorithm. This section describes how a meta-level *Incorporation Problem Solver* (or IPS) could automate this process.

The task of the IPS is to improve the performance of the base-level problem solver by applying test incorporations (and possibly other transformations as well). This improvement in problem-solving efficiency must preserve the correctness of the algorithm. To have some notion of correctness, the IPS must have some kind of specification of the desired behavior. One simple form of specification is to provide the IPS with a naive generate-and-test algorithm as its initial base-level problem solver. Any improvement that produces the same outputs as this generate-and-test procedure will be considered to be correct. However, the IPS may also find it useful to maintain more abstract, partial descriptions of the program that it is modifying. Indeed, it is reasonable for the IPS to be given a formal specification of the desired algorithm as well as a particular implementation.

There are many different ways that we might try to implement an IPS. Perhaps the most general would be to use a generate-and-test algorithm (see Figure 6). The naive generator for the IPS generates all possible algorithms and tests each of them to see if (a) it is equivalent to the original (or desired) algorithm and (b) it is the most efficient such algorithm. Unfortunately, this test cannot be performed, because part (a) involves solving the halting problem. Hence, *a truly general IPS cannot be implemented.*

Fortunately, many reasonable approximations of the IPS *can* be constructed. One obvious candidate is an incremental, greedy IPS based on test incorporation. This IPS would operate iteratively as follows. First, it would identify a generator and a test within the current base-level problem solver. Then it would analyze the generator and the test to identify possible incorporations that could be performed. It would apply the best of these incorporations and repeat until no more incorporations could be performed. This greedy IPS takes advantage of the attractive properties of test incorporations—namely, that they are incremental and always lead to improved performance.

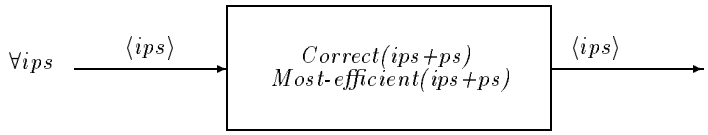

Figure 7: Naive generate-and-test version of the meta-IPS

Programs similar to this IPS have already been developed. Elaine Kant's (1979) LIBRA system generates alternative algorithm refinements within the PSI automatic programming system, analyzes and compares their efficiencies, and selects the best refinement. David Smith's (1985) greedy algorithm for ordering of conjunctive queries performs a similar analysis. At each step, it considers which conjunct of the query should be processed (i.e., incorporated) next. It chooses the conjunct with the fewest estimated number of solutions. Since (in Smith's architecture) each conjunct serves simultaneously as a generator (of values for unbound variables) and as a test (of the values of bound variables), the effect is to apply the most informative tests as soon as possible and delay running generators until they are highly constrained by bound input variables.

Each of these IPS implementations is quite general, in that each performs significant deliberation at run-time. However, in order to understand the specific kinds of run-time incorporation involved in algorithms like the guess-the-number game or means-ends analysis, we must consider very special purpose IPS implementations.

In the guess-the-number algorithm, the IPS performs no deliberation at all. It simply waits for the next answer from the oracle and incorporates it into the subspace generator's current interval. If any other information were made available about the answer—such as "the number is even"—it could not be exploited by this specialized IPS. Similarly, in means-ends analysis, the procedure followed by the IPS is also fixed. It maintains a stack of goals and consults the operator-difference table to determine which incorporation to make at each step.

In general, we can view all of these various IPS implementations as improvements upon the (unimplementable) generate-and-test IPS implementation. This leads us to ask whether some form of IPS could be applied to improve *itself*. What kinds of knowledge are being incorporated into the IPS?

5 The Meta-Incorporation Problem Solver

The task of a meta-IPS is to improve the combined performance of the IPS and the base-level problem solver by modifying the IPS. It can be viewed as searching a space of possible IPSes looking for one that, when executed with the base-level problem solver, solves the problem most efficiently. Hence, a naive generate-and-test version of the meta-IPS generates all possible incorporation problem solvers and tests each one to see if (a) it produces base-level problem solvers that are correct and (b) the combined cost of running it and then running the base-level problem solver that it produces is minimized. Figure 7 shows a diagram of this naive generate-and-test meta-IPS.

Of course, this naive algorithm is unimplementable. In practice, the meta-IPS can employ test incorporations to improve the IPS. These test incorporations will move test information out of the IPS's test and incorporate it into the IPS's generator of possible base-level problem solvers. Recall from Figure 6 that the test of the IPS contains knowledge about the problem specification so that the IPS can verify that its improved base-level problem solvers are correct. Hence, the meta-IPS

can incorporate knowledge about the specific problem (or class of problems) into the IPS. The result is an IPS that can only incorporate test knowledge relevant to particular classes of problems.

Consider the guess-the-number example again. The initial problem specification consists of a generator of the numbers from 1 to 100 and a test (the oracle) that can give the outputs "yes," "too high," and "too low." An incremental IPS of the kind we have been discussing would allow the base-level problem solver to make a guess (e.g., 20), accept the answer from the oracle ("too low"), generate all possible (base level) test incorporations, and check them to see if they could incorporate this specific answer. The meta-IPS can improve this incremental IPS by incorporating the knowledge (from the IPS's test) that only certain kinds of test information will be available to the IPS—namely, the three possible answers from the oracle. Hence, it will change the IPS so that it is only capable of incorporating these three answers. A good meta-IPS could figure out that a sub-space generator for integer intervals could be used to incorporate each of the possible oracle answers. Hence, it would modify the IPS so that it will incorporate this kind of generator into the base-level problem solver. The result is the algorithm that we described earlier for playing guess-the-number.

Hence, we see that the meta-IPS operates by moving test knowledge out of the IPS's test and into its generator of possible incorporations. This results in a version of the IPS customized to incorporating particular kinds of tests into particular kinds of generators. Similar kinds of meta-IPS reasoning could produce the specialized IPS that constitutes the version space algorithm and the specialized IPS that constitutes means-ends analysis.

Although we have yet to find an example that requires it, there is no reason to stop with only two meta-levels. We can imagine an infinite tower of meta-level problem solvers, each of which has the task of improving the combined performance of all of the problem solvers below it. The familiar algorithms that we use from day to day can be viewed as the "residue" that remains after the IPS, the meta-IPS, and even the meta-meta-IPS (ad infinitum) have performed their reasoning and been "compiled away." The infinite tower of meta-level problem solvers would be fully general and fully flexible. Any aspect of the problem specification could be changed, and the infinite tower would incorporate those changes into its strategy for incorporating changes...into the base level problem solver. The tower is unimplementable, of course, both because of its infinite regress and also because each level (except the lowest) must solve the halting problem. The value of this infinite tower is that it provides a kind of infinite series expansion that describes the implementation decisions that resulted in the final algorithm. Any real tower of IPSes will need to be truncated at some level—thus becoming a finite approximation to the infinite series.

An interesting consequence of this view of the meta-IPS is that it resolves the problem of the tradeoff between meta-level and base-level problem solving. The naive IPS is a compulsive optimizer. It will continue to search for the most efficient base-level problem solver, even if that search ends up taking more time than it would have taken to simply execute the unimproved base-level program. This problem is common in all meta-level architectures. The costs of doing the meta-level reasoning may overwhelm the benefits (see Smith, 1985; Rosenschein & Singh, 1983).

The meta-IPS solves this problem by restraining the IPS. Because the meta-IPS seeks to minimize the *combined* cost of the IPS and the base-level problem solver, it will always ensure that the cost of running the IPS is offset by the improvements to the base-level problem solver. When this is not the case, the meta-IPS will "compile-away" the IPS and simply run the base-level problem-solver.

To see how this works, consider the familiar algorithm for conducting a binary search of a sorted array of integers. The problem to be solved is "Does the array A contain integer i?" This is a case where the complete "test" (i.e., the array of integers) is known. The naive generate-and-test algorithm simply generates possible indexes j and checks whether $A[j] = i$. When the naive

IPS begins to analyze this problem, one thing that it might do is to completely inspect the test (i.e., the array A) and determine the right answer. It can then incorporate that answer into the generator producing a trivial base-level problem solver. However, the meta-IPS will not permit this to happen. After all, the IPS cannot search the array A any faster than the base-level problem solver can, so the combined base-level plus IPS problem solver has not been improved.

Instead, the meta-IPS will consider the costs and benefits each time the IPS inspects a cell of A. Each inspection of a cell yields the information "correct," "too high," or "too low." This means that the meta-IPS can modify the IPS so that it treats the array A as if it were the oracle in guess-the-number. It will adopt the subspace generator for intervals and incorporate the information that is learned from each inspected cell. Because the meta-IPS wants to minimize the total problem-solving time, it will consider ways to minimize the number of required incorporation steps. The number of incorporation steps is proportional to the size of the remaining interval of possible integers. Hence, the number of incorporation steps can be minimized by minimizing the size of this interval. The meta-IPS will therefore constrain the IPS to consider only the cell from A that is in the middle of the interval. This produces the binary search algorithm. When the IPS is run, it will completely solve the problem by analyzing carefully selected parts of the test and incorporating them into the base-level subspace generator until only one integer remains. This specialized IPS produces the same result (i.e., the trivial generator that only produces the answer) as the naive IPS, but it does so much more efficiently.

6 Summary

We began this paper by describing a general kind of optimizing transformation: test incorporation. We showed that test incorporation is very general, and that it can be used to derived a wide variety of algorithms. Furthermore, test incorporation provides a knowledge-level account of the derivation of these algorithms, showing at each stage what knowledge was exploited in order to justify the derivation.

We then shifted our perspective and considered test incorporation as a kind of algebraic operation, in which the base-level problem solver is progressively simplified until only the answer remains. This perspective allowed us to derive a broader class of algorithms including means-ends analysis and the candidate-elimination algorithm.

Next, we considered the requirements for a problem solver that would automate the test incorporation process. We showed that a fully general incorporation problem solver (IPS) could not be implemented, but that several interesting specialized IPSes could be developed. Our definition of the IPS as a meta-level problem solver that improves the base-level problem solver had the consequence that the IPS does not worry about its own efficiency.

To address this shortcoming, we introduced the meta-IPS (and by extension, an infinite tower of meta-level improvement problem solvers). The meta-IPS is concerned with improving the combined performance of the IPS and the base-level problem solver. We showed that the meta-IPS could accomplish this by developing specialized IPSes that only pursue specific incorporations and that perform incremental cost/benefit analyses.

7 Concluding Remarks

This paper has been concerned primarily with a theory of problem solving and algorithm construction. We would now like to consider the implications of test incorporation theory for artificial intelligence.

One of the long standing goals of artificial intelligence is to develop very general, yet powerful (i.e., efficient) problem solvers. Intelligent systems—for example people—are remarkable not only because they possess a large store of knowledge about the world but also because they are able to apply this knowledge so effectively in such a diverse set of circumstances. Furthermore, people are able to acquire new knowledge and integrate it into their existing problem solving routines.

In his (1969) article on ill-structured problems, Allen Newell discussed the widely perceived tradeoff between generality and power. He characterized the weak methods, such as generate-and-test, as being very inefficient but broadly applicable and the strong methods, such as linear programming, as being very efficient but highly specialized. The DENDRAL system (Buchanan and Feigenbaum, 1978) and the vast array of expert systems that have followed it have demonstrated that it is possible to engineer these highly efficient and specialized methods for a wide variety of domains.

Let us consider the theories of intelligence that are implicit in the methodology of expert systems and in Newell's (1969) paper. The work on expert systems describes intelligent systems as possessing a large collection of highly specialized methods, each applicable to a different task. This "big switch" theory of intelligence says that a system is more intelligent if it has more of these specialized methods.

Newell's paper takes a slightly more general perspective. He says that intelligent systems possess a range of methods of varying generality and power. In addition to the many specialized "expert" methods, people also possess the weak methods and many intermediate methods as well. Systems are more intelligent to the extent that they have a broader range of methods—strong methods as well as weak methods.

The fundamental shortcoming of both of these theories of intelligence is that they do not explain learning and adaptation. Where do all of these expert methods come from? Surely physicians are not born with MYCIN's knowledge of blood diseases or DENDRAL's knowledge of organic chemistry! Test incorporation shows promise of providing a better theory of intelligence that explains how these strong methods are developed. According to this theory, intelligence is not the ability to perform a task effectively. It is instead *the ability to exploit knowledge* effectively by integrating it into the problem solver—in short, it is the ability to perform test incorporation. Specialized, strong methods are developed by incorporating knowledge into weak methods—particularly into the weakest method of all: generate-and-test. A system is more intelligent to the extent that it is able to incorporate a broader range of knowledge into its problem solving apparatus more efficiently.[4]

This theory of intelligent behavior draws a distinction between knowledge and intelligence. 'Knowledge' is defined (as in Newell's (1981) Knowledge Level paper) to be the ability to compute the right answer without regard to how that computation occurs (i.e., how much space or time is consumed). According to this definition, even a naive generate-and-test algorithm is knowledgeable. However, it is not intelligent, because it is unable to exploit its knowledge to solve problems efficiently.

In summary, the fundamental implication of test incorporation theory for AI is that intelligent systems must possess the ability to perform test incorporations. It is the ability to accept new knowledge and exploit it to improve problem-solving efficiency that is the mark of an intelligent system. An immediate corollary is that, because it is impossible to develop a fully general IPS, no fully intelligent system can be constructed. All systems will have some kinds of input knowledge that they will be unable to exploit. Finally, according to test incorporation theory, an important area in which to focus our research efforts is to identify and analyze new test incorporation methods.

[4]Even this definition of intelligence may be too restricted, because it ignores the question of where the "input" knowledge comes from that the system is trying to exploit. We are talking here only of "symbol level learning," not "knowledge level learning." See Dietterich, 1986.

This requires us to develop a better understanding of the properties of digital computers and other kinds of computing devices, since test incorporation methods succeed by exploiting the properties of these devices. It may be the case that digital computers do not provide a good target for test incorporations and that other computing devices must be developed. In any case, test incorporation theory shows promise of providing us with guidance in our attempts to understand and reproduce the mechanisms of intelligence.

8 Acknowledgments

This research was supported in part by the National Science Foundation under grant numbers IST-8519926 and DMC-8514949. The authors thank Nicholas Flann for a critical reading of earlier drafts of this paper.

9 References

Amarel, S. 1968. On the representation of problems of reasoning about actions. In Michie (ed.), *Machine Intelligence 3*, U. of Edinburgh Press.

Amarel, S. 1983. Program synthesis as a theory formation task—problem representations and solution methods. Rep. No. CBM-TR-135, Department of Computer Science, Rutgers University.

Bennett, J. and Dietterich, T. Forthcoming. Applying test incorporation to derive efficient algorithms.

Buchanan, B. G. and Feigenbaum, E. A. 1978. Dendral and Meta-Dendral: Their applications dimension. *Artificial Intelligence,* 11, 5–24.

Cohen, D. 1986. Automatic compilation of logical specifications into efficient programs. In *Proceedings of AAAI-86*, Los Altos: Morgan-Kaufmann. 20–25.

Dietterich, T. G., 1986. Learning at the Knowledge Level. *Machine Learning,* 1 (3).

Kahn, K. M. 1983. A partial evaluator of Lisp written in Prolog. UPMAIL memo, Department of Computing Science, Uppsala University.

Kant, E. 1979. Efficiency considerations in program synthesis: A knowledge-based approach. Doctoral dissertation. Rep. No. STAN-CS-79-755. Department of Computer Science, Stanford University.

Lindsay, R., Buchanan, B., Feigenbaum, E., and Lederberg, J. 1980. *Applications of Artificial Intelligence to Organic Chemistry: The Dendral Project.* New York: McGraw-Hill.

Mostow, D. J. 1983a. Machine transformation of advice into a heuristic search procedure. In *Machine Learning*, Michalski, R. S., Carbonell, J. G., and Mitchell, T. M., (eds.), Palo Alto: Tioga. 367–404.

Mostow, D. J. 1983b. A problem-solver for making advice operational. In *Proceedings of AAAI-83*, Los Altos: Morgan-Kaufmann. 279–83.

Newell, A. 1969. Heuristic programming: ill-structured problems, in *Progress in Operations Research,* Arnofsky, J., (ed.), New York: Wiley. 363–414.

Newell, A. 1981. The Knowledge Level. *AI Magazine* 2 (2) 1–20.

Rosenschein, J., and Singh, V., 1983. The utility of meta-level effort. Rep. No. HPP-83-20. Department of Computer Science, Stanford University.

Shapiro, E. 1986. (ed.) *Third International Conference on Logic Programming,* Lecture Notes In Computer Science No. 225. Berlin: Springer Verlag. 25–83.

Simon, H. A., and Kadane, J. B. 1975. Optimal problem-solving search: all-or-none solutions. *Artificial Intelligence.* 6 (3) 235–247.

Smith, David E., and Genesereth, M. R. 1985. Ordering conjunctive queries. *Artificial Intelligence*, 26 (2) 171–216.

Smith, David E. 1985. *Controlling Inference,* Doctoral Dissertation, Department of Computer Science, Stanford University.

Smith, Douglas R. In press. On the design of generate-and-test algorithms: subspace generators.

Tappel, S. 1980. Some algorithm design methods. In *Proceedings of AAAI-80*, Stanford, California. 64–67.

Tarjan, R. E. 1975. Efficiency of a good but not linear set union algorithm, *J. Association for Computing Machinery,* 22, 215–225.