# The Test Incorporation Hypothesis and the Weak Methods

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

Thomas G. Dietterich[1]
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331

## Abstract:

Test incorporations are program transformations that improve the performance of generate-and-test procedures by moving information out of the "test" and into the "generator." The test information is said to be "incorporated" into the generator so that items produced by the generator are guaranteed to satisfy the incorporated test. This article proposes and investigates the hypothesis that a general theory of AI methods can be constructed using only test incorporations. Starting from an initial generate-and-test algorithm, we attempt to derive the weak methods of heuristic search, hill climbing, and avoiding duplicates via a series of test incorporations. The derivations show that test incorporations are very powerful but that occasionally other program reformulations are required. Nevertheless, we conclude that test incorporation provides a good foundation upon which to construct a general theory of methods.

Submitted to AAAI-86, Philadelphia, PA

[a]The authors have chosen to list their names in alphabetical order.

# The Test Incorporation Hypothesis and the Weak Methods

## 1 Background and Introduction

In his (1969) article on ill-structured problems, Allen Newell presented the vision of a comprehensive theory of AI problem-solving methods in which all methods could be seen as improvements on the fundamental method of generate and test. His main goal in pursuing such a theory was to identify and clarify the relationship between the *generality* of a method—its breadth of application to a range of problems—and its *power*—its efficiency in solving those problems. Such a theory would be valuable for many reasons. First, it might provide us with a general theory of compilation—that is, a theory of how inefficient problem-solving methods can be converted into more efficient ones. Second, it might be able to guide us toward the discovery of methods that are simultaneously general and efficient. In short, such a theory might enable us to construct a theory of intelligence.

In the years since Newell's article appeared, few researchers have taken up the quest for his theory of methods. In this paper we investigate one idea that might provide the seed for such a theory—*test incorporation.* Test incorporation is the process of improving a generate-and-test procedure by moving information out of the "test" and incorporating it into the "generator." In a "naive" generate-and-test procedure, the generator merely generates data objects that represent possible solutions to a problem. The "test" contains all of the knowledge necessary to recognize whether one of these data objects is in fact a solution. Test incorporation improves the naive generate-and-test procedure by modifying the generator so that the candidate solutions it generates are guaranteed to satisfy some portion of the "test." To obtain the most efficient procedures, this process of test incorporation can (in some cases) be continued until there is no "test" left—it has all been incorporated into the generator. Everything the generator produces is guaranteed to be a solution to the problem.

Examples of test incorporation can be found throughout computer science. Consider the problem of solving the equation $x+5 = 10$. A naive generate-and-test procedure would generate possible integer values for $x$ and test them by substituting them into the equation. A more clever approach applies an algebraic transformation to reformulate the equation (i.e., the test) so that it can be incorporated into the generator. In this case, we obtain the trivial generator that generates only the value 5. It is guaranteed to satisfy the test, so there is no need to even check it.

Some excellent work has been done on identifying particular kinds of test incorporations and building systems to perform them. Tappel's (1980) paper on algorithm design applied test incorporation to synthesize efficient combinatorial algorithms. Mostow (1983) demonstrated how to improve a heuristic search procedure by incorporating information extracted from the goal statement (i.e., the "test"). Analyses of appropriate methods for interleaving tests and generators has been done by Simon and Kadane (1975) and Smith and Genesereth (1985). Douglas Smith (1986) shows how to improve generate-and-test algorithms through the use of "sub-space generators." Amarel (1968, 1982, 1983) discusses methods for reformulating problem solvers so that efficiency-producing transformations can be applied.

Part of the attractiveness of generate-and-test problem solvers and of test incorporation is the ease with which they yield to a knowledge-level analysis (Newell, 1981). In a naive generate-and-test procedure, virtually all of the knowledge resides in the "test." The generator merely knows how to generate the elements of some search space. Test incorporation can be understood as moving knowledge from the test to the generator.

Another attractive aspect of the test incorporation paradigm is that it has a direct relationship to the tradeoff between generality and power. A naive generate-and-test procedure is very general in the sense that it can easily be modified to solve a different problem: if the generator is sufficiently broad, we need only change the test. Hence, the naive generator of integers that we used to solve $x + 5 = 10$ could also be applied to solve $x^3 - 4x^2 - 12x = 0$. As the knowledge in the test is incorporated into the generator, this flexibility is lost. As flexibility is lost, however, power is gained. After incorporation, the test will be invoked less often—useless work (of evaluating non-solutions) will be avoided.

To summarize, the notion of test incorporation appears to be a good point of departure for constructing a theory of methods that can explain both how knowledge becomes "compiled" into programs and how flexibility is lost because of this compilation process. The purpose of this paper is to pursue what we call the *test incorporation hypothesis*:

> **Test incorporation hypothesis:** All methods can be derived via a series of test incorporations from a naive generate-and-test method.

In this paper, we explore this hypothesis by focusing on the weak methods (e.g., hill climbing, heuristic search, etc.), since these are the methods most closely related to generate-and-test. If the test incorporation hypothesis is correct, then, at the very least, it should be possible to derive all of the weak methods via test incorporations from the method of generate-and-test.

The remainder of this paper presents derivations for several of the weak methods. These derivations demonstrate that the test incorporation hypothesis fails to hold for all of the weak methods. However, the nature of the failure suggests that a general theory of methods can still be obtained by augmenting test incorporation with other program transformations that reformulate the test without altering the generator.

## 2 Previous analysis of the weak methods

The weak methods have recently been the target of an alternative analysis by Laird and Newell (1983a, 1983b). They seek to develop an architecture in which the different weak methods can be expressed as a set of very closely related computer programs. They have developed a core computer program (the "universal weak method" or UWM) and a set of program extensions called "method increments." Each weak method can be obtained by combining the core computer program with one or more of the method increments. Hence, their analysis provides one possible approach to generating the weak methods.

The core program and the method increments are represented as sets of production rules in the SOAR architecture (Laird, 1984). The program and the method increments are combined simply by forming the union of the sets of production rules. This modularity is obtained by constraining the production rules to interact only by expressing "preferences" for the future actions of the architecture.

Does the universal weak method provide the generative theory of methods that we are searching for? The answer is no. The UWM and the SOAR architecture do not provide us with a set of operators that can be applied to convert weaker problem solvers into more powerful ones. Instead, they only provide us with an architecture for merging method increments. It is up to us to develop method increments that can combine with the UWM to yield powerful composite methods. Even this strategy cannot be pursued indefinitely. To quote Laird and Newell (p. 43), "At some point, ... the form of automatic assimilation required by a universal weak method fails as the knowledge about the task environment becomes sufficiently complex. We do not know where such a boundary lies ...."

Recent work by Laird, Rosenbloom, and Newell (in press) has extended the SOAR architecture to include *chunking*—a general mechanism for strengthening any search method that employs subgoaling. Chunking can be viewed as a simple kind of test incorporation in which knowledge made explicit during the processing of a subgoal is incorporated into the generator of future problem solving actions.

## 3 Notation and architectural assumptions

In order to describe our derivations of the weak methods, we need a notation for procedures. We will employ data flow diagrams containing four kinds of components: generators, tests, functions, and memories. These are connected by data flow links that carry single data items. There is no global scheduling. Each component executes as soon as it receives data items on all of its inputs. As it executes, the component can decide whether or not to consume each of these input data items. Since each data-flow link can hold only one data element at any time, this can be used to synchronize parallel processes. Briefly, each of the four components operates as follows.

A generator, in its simplest form, has no inputs and only one output. It generates elements over some domain and places them on its output. Except as noted below, we make no assumptions about the internal operation of our generators. They can best be modeled as doing random sampling with replacement from some domain set. Consequently, they do not generate the domain set in any particular order, and they do not produce any "stop signal" to indicate that they have exhausted the domain set. If a generator has an input, then each time an item arrives on that input, it may cause the set being generated to change. Some generators have a termination input that causes them to cease execution.

A test is a filter or switch. It has one input and one or more outputs. In the usual case, the test has two outputs: "success" and "failure." For each input data item, it determines whether the input satisfies some condition $T(x)$, and if it does, it is sent out the "success" output; otherwise, it is sent out the "failure" output. (In this paper, only success outputs are shown.)

A function has one or more inputs and one output. It computes its output as some function (in the mathematical sense) of its inputs.

A memory has two inputs and one output. One of the inputs causes data to be stored in the memory. The other input requests that data be retrieved and sent out the output. We also employ cumulative memories in which each input is added to a growing set of elements. The entire set of elements is produced in response to a retrieval request.

None of these components should be regarded as primitive. Each generator, test, function, and memory may in fact be constructed as a combination of more primitive generators, tests, functions, and memories. This regress terminates in the underlying hardware, which we will ignore in this paper.

Aside from the given inputs, there are no other inputs from the environment. In this paper, we do not discuss interaction with an external environment.

## 4 The Naive Generate-and-Test Methods

Our goal is to derive the weak methods via a series of test incorporations applied to a single "naive" generate-and-test procedure. However, it is not possible to develop a single generate-and-test procedure than can solve all problems. Instead, there appear to be three broad classes of problems, each with its own "naive" generate-and-test method. The three problem classes are (a) *find one* problems, in which the goal is to find a single data item that is a solution to the problem; (b) *find all* problems, in which the goal is to find all legal solutions to the problem; and (c) *find best* problems, in which the goal is to find the best solution according to some optimality criterion.

$$\begin{tikzcd} \boxed{G} \arrow[r, "c"] & \boxed{T} \arrow[r, "c"] & {} \end{tikzcd}$$

Figure 1: Naive generate-and-test method for *find one* problems

$$\begin{tikzcd} \boxed{G} \arrow[r, "S"] & \boxed{T} \arrow[r, "S"] & {} \end{tikzcd}$$

Figure 2: Naive generate-and-test method for *find all* problems

For each of these classes, we want to construct a naive generate-and-test procedure that satisfies the following four properties. First, it must have the form of a single generator connected to a single test. Second, the generator should only produce candidates that could constitute entire solutions to the problem. Third, the only knowledge in the generator should be knowledge of how to generate elements from the domain set. Fourth, the only knowledge in the test should be knowledge of how to recognize legal solutions.

There are two reasons for stating these criteria. First, we want to limit the complexity of the naive generate-and-test procedure. If there were no constraints placed on the procedure (i.e., if other components were permitted), our test incorporation hypothesis could be trivially satisfied. Second, we want to preserve the knowledge level analysis of the generate-and-test procedure by (a) placing limits on the communication between the generator and the test and (b) keeping the control of the procedure very simple. Communication must be limited, because it permits knowledge to flow between the test and the generator, thus defeating the knowledge level analysis. Control must also be kept simple, because complex control can itself embody significant amounts of knowledge. We want a generate-and-test problem solver that is so simple that as soon as the generator produces a solution acceptable to the test, the procedure can halt. All additional complexity should be introduced as the result of test incorporations.

For *find one* problems, it is easy to find a naive generate-and-test procedure that satisfies the properties given above. Figure 1 shows a data flow diagram for *find one* generate-and-test procedures. The generator produces candidate solutions until the test is satisfied. The paradigmatic example of a *find one* problem is the task of opening a combination safe. If the safe is well designed, the only way to open it is to generate and test all possible combinations until one succeeds. The generator in such situations is the person trying to open the safe, and the test is the safe itself.

To construct naive generate-and-test procedures for the other two classes of problems, we must convert them into *find one* problems. Figure 2 shows the flow diagram for *find all* problems. Each time the generator runs, it produces a *set* of candidate solutions. The test must determine (a) whether each member of the set is indeed a solution ($T_{sol}$) and (b) whether the set contains all possible solutions ($T_{done}$). If such a set of all legal solutions is produced, the test will place it on its success output and the problem solver will halt. This procedure satisfies all three of our properties.

We have not found it possible to construct a *find best* method that satisfies all four of the properties mentioned above. We have had to compromise on property two—namely, that each item produced by the generator could serve as the complete solution to the problem. For *find best* problems, the complete solution is simply a single element from the solution space. Suppose we used the generator from Figure 1 to generate the individual elements in the space. In that case, the

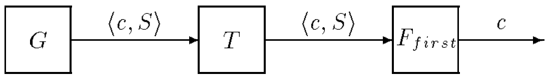
$$\begin{tikzcd} \boxed{G} \arrow[r, "\langle c,S\rangle"] & \boxed{T} \arrow[r, "\langle c,S\rangle"] & \boxed{F_{first}} \arrow[r, "c"] & {} \end{tikzcd}$$

Figure 3: Naive generate-and-test method for *find best* problems

test would need to generate for itself all of the other items in the solution space and check to see if the generated item was the optimum. But such a test knows too much—it knows how to generate the entire space as well as how to recognize a solution. There is no need for a separate generator.

To resolve this problem, we have chosen to let the generator provide the test with some information in addition to the candidate "best" item. Figure 3 shows our naive *find best* procedure. The generator produces ordered pairs of the form $\langle c, S\rangle$ where $c$ is a candidate for the best solution, and $S$ is a set of candidates that the generator is proposing as the set of all possible solutions. The test must determine (a) whether each member of $S$ is indeed a solution ($T_{sol}$), (b) whether $S$ contains all possible solutions to the problem ($T_{done}$), and (c) whether $c$ is the best solution in $S$ ($T_{best}$). When a candidate solution satisfies all three of these tests, then the $c$ portion of the ordered pair is produced as the answer. This is the purpose of the selector function $F_{first}$ in the figure.

## 5 Derivation of the Weak Methods

Having described the naive generate-and-test methods for each problem class, we can now derive the various weak methods familiar to researchers in artificial intelligence. Space limits prevent us from showing derivations for all of the weak methods. Instead, we show (a) how to obtain the familiar weak methods for *find all* and *find best* problems, (b) how to derive heuristic search, (c) how to combine heuristic search with the *find best* method to yield simple hill climbing, and (d) how to derive the weak method of generate-and-test avoiding duplicates.

### 5.1 Deriving the standard *find all* and *find best* weak methods

The naive methods described in the previous section are somewhat unorthodox. The standard approach to solving *find all* and *find best* problems is to accumulate individual solutions in a memory, and, in the case of *find best* problems, keep track of the best solution generated. In this section, we show how these more familiar procedures can be derived through a series of incorporations applied to the naive methods of the previous section.

All incorporations require some ability to analyze the internal structure of the generator and the test. In our naive *find all* method, the test can be subdivided into two subtests $T_{sol}$ (which identifies legal solutions) and $T_{done}$ (which determines when all solutions have been found). The generator, which in Figure 2 produces entire sets of candidate solutions, usually contains a subgenerator ($G_{el}$) that generates individual elements from the domain. When these substructures are available, it is possible to rearrange the problem solver and incorporate $T_{sol}$ into the generator. The results are shown in Figure 4. Elements generated by $G_{el}$ are immediately filtered by $T_{sol}$. This vastly improves the procedure. Next these solutions are accumulated in a memory, $M_{sols}$. As each element enters $M_{sols}$, it also causes $M_{sols}$ to produce as output all of the solutions that have accumulated to this point. This set $S$ of elements is passed to $T_{done}$. This gives us the familiar method for finding all the solutions to a problem, which we will call $G_{all}$.

To obtain the familiar *find best* procedure, we first divide the test into three subtests: $T_{sol}$,

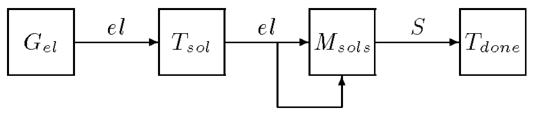

Figure 4: The *find all* procedure after incorporating $T_{sol}$

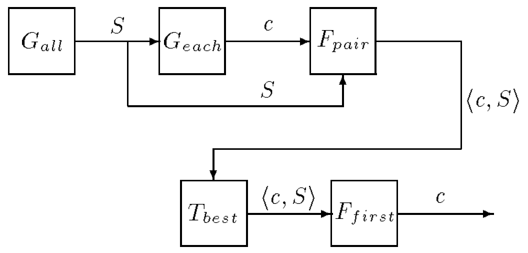

Figure 5: The *find best* method after incorporating $T_{sol}$ and $T_{done}$

$T_{done}$, and $T_{best}$. Then we incorporate $T_{sol}$ and $T_{done}$ into the generator to yield $G_{all}$ as described above. When $G_{all}$ produces the set $S$ of all possible solutions, $S$ is passed to $G_{each}$. This generator takes a set as an input and generates each of its elements in turn. These elements and the original set are both passed to the function $F_{pair}$, which constructs the ordered pairs that are output by the modified generator. These ordered pairs, since they already satisfy $T_{sol}$ and $T_{done}$, need only be checked by $T_{best}$ to find the best solution. The resulting procedure is shown in Figure 5.

This *find best* procedure is still quite inefficient, because it doesn't take any advantage of the internal structure of $T_{best}$. For some optimization problems, however, this is the best that can be done. Consider, for example, the problem of finding the point in a set of points in $n$ dimensional space that is nearest the center-of-mass of the set. The computation of the center-of-mass can be performed incrementally. However, the problem of finding the point nearest to this center-of-mass requires explicit storage of the $S$ set ($M_{sols}$) and some (possibly clever) enumeration of this set analogous to $G_{each}$.

For many optimization problems, however, the internal structure of $T_{best}$ involves the repeated application of a comparative test, $T_{better}$. $T_{better}$ determines whether one candidate solution is better than another. If $T_{better}$ is available, we can incorporate it immediately after $T_{sol}$. Figure 6 shows this improved version of the *find best* method. $G_{all}$ is expanded to show $T_{sol}$, $M_{sols}$, and $T_{done}$. The memory $M_{bsf}$ stores the best solution that we have seen so far. $F_{gate}$ is a simple function that passes $c$ through unchanged. The output of $T_{done}$ acts as a gating signal to release $c$. We have not shown all of the fetch signals going into $M_{sols}$ and $M_{bsf}$.

This incorporation is an example of the finite differencing transformation studied by Paige (Paige and Koenig, 1982). One form of finite differencing converts a function $F(S)$ over a set $S$ into a new function $F'(el)$, where $el$ is a new element being added to $S$. In Figure 5, the combination of $G_{each}$, $F_{pair}$, $T_{best}$, and $F_{first}$ can be viewed as a function for computing the best element of $S$. In Figure 6, the combination of $T_{better}$ and $M_{bsf}$ constitutes the finite difference of this function with respect to the changes in $S$ that occur at $M_{sols}$.

One further incorporation can be performed to eliminate $T_{done}$. Many generators over finite sets produce the elements systematically in such a way that it is easy to tell when the generator has produced all possible elements. Hence, the generator can signal when $T_{done}$ is satisfied. Figure 7

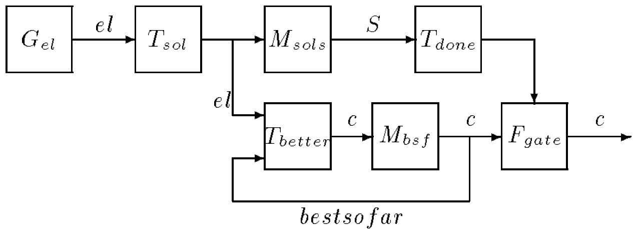

Figure 6: Improved *find best* method incorporating $T_{better}$

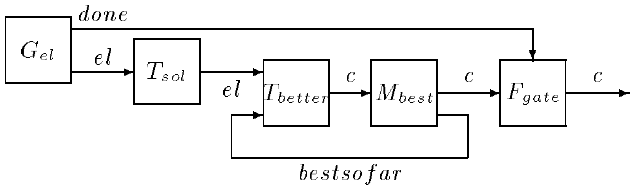

Figure 7: *Find best* method exploiting the done signal from $G_{el}$

shows the *find best* procedure with this incorporation performed.

These incorporations demonstrate the power of test incorporation and lend support to the test incorporation hypotheses. Each of the standard *find best* and *find all* methods was obtained via a series of test incorporations applied to a naive generate-and-test method.

## 5.2 Derivation of Heuristic Search

So far, we have only considered generators, such as $G_{el}$, that generate whole solution candidates. However, virtually all AI search methods work by incrementally constructing and evaluating partial solutions. Game-playing programs, for example, do not generate entire games and then test to see if they have won. Instead, they generate a sequence of moves by analyzing the board position at each stage in the game and selecting moves from that position that appear promising. Hence, in order to derive the weak method of heuristic search, we must examine the internal structure of the generator, $G_{el}$, and find opportunities there for incorporating tests.

Figure 8 shows the internal structure of an incremental path-extending generator similar to one employed by Mostow (1983a). $M_{path}$ is a cumulative memory for the set of partial paths that have already been generated. Each trip around the loop, the generator $G_{path}$ starts generating all of the elements in $M_{path}$ until one of them satisfies the path test $T_{path}$. While $G_{path}$ and $T_{path}$ are selecting a path, $G_{step}$ and $T_{step}$ are selecting a step $e$ to concatenate onto the end of the path in order to extend it. Once both of these are selected, $T_{appl}$ determines whether the selected extension $e$ can be applied to the selected path $p$. If not, a new path and a new extension will be selected. If so, the extension is applied to the path by $F_{apply}$ to produce a new path $p'$. At this point, $T_{comp}$ determines whether $p'$ is a complete path. If so, it is produced as the output of this composite generator. In any case, it is merged into the remaining paths in $M_{path}$.[1]

This path-extending generator can be made more efficient by modifying $G_{step}$ so that it takes the output of $T_{path}$, the chosen path, as an input and incorporates $T_{appl}$ (see Figure 9). In other

[1]In problem space terminology, a partial path corresponds to a state, and a path extension corresponds to an operator that can be applied to the state to yield a new state.

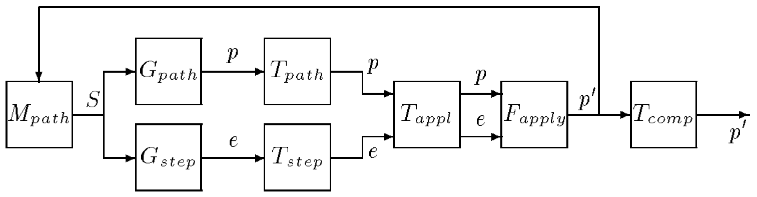

Figure 8: A path-extending generator

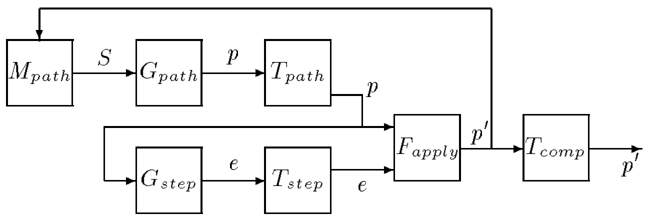

Figure 9: The path-extending generator after incorporating $T_{appl}$

words, $G_{step}$ will only generate extensions that can be applied to the chosen path. We will consider this more efficient generator as the starting point for developing the heuristic search method.

Now that we have examined the internal structure of this path-extending generator, let us consider what incorporations can be performed when we use it in place of $G_{el}$ in Figure 6. The key to deriving the various forms of heuristic search is to incorporate parts of $T_{sol}$ and $T_{better}$ as heuristic preferences in $T_{path}$ and $T_{step}$. For example, if $T_{path}$ always chooses the path most recently added to $M_{path}$, then we obtain the weak method of depth-first search. If $T_{step}$ prefers steps that reduce differences between the current state and the goal state, then we obtain means-ends analysis (MEA). In this case, part of $T_{sol}$ (the knowledge of the goal) has been incorporated into $T_{step}$. Mostow (1983a, 1983b) demonstrates a system that takes the overall test, $T_{sol}$, and incorporates all or part of it into $T_{path}$ and $T_{step}$. His system also incorporates knowledge from $T_{sol}$ into the initial state of $M_{path}$ and into an additional test placed on the output line between $F_{apply}$ and $M_{path}$.

## 5.3 Derivation of Hill-climbing

Another method that exploits the structure of a path-extending generator is the method of simple hill climbing (SHC). At each point in the search space, SHC searches for a path extension that will improve the evaluation of the current state. As soon as such a path extension is found, it is taken. If no such path extension can be found, then the search terminates and returns the current state. In other words, the procedure climbs upward to the top of a hill, but it doesn't necessarily follow the steepest path. Simple hill climbing will only succeed in finding a global optimum if two properties are satisfied. First, there must be exactly one hill (i.e., the space is unimodal), and second, mesas and ledges (i.e., regions of adjacent points with equal evaluations) must not exist.

When we attempt to derive SHC from the *find best* procedure of Figure 3, we encounter difficulties. The initial test for SHC contains the three subtests $T_{sol}$ (which filters out illegal points), $T_{best}(c, S) \equiv \forall x \in S \; x \neq c \supset T_{better}(c, x)$ (which defines global optimum), and $T_{done}$ (which deter-

$$\begin{tikzcd}\boxed{G_{el}} \arrow[r, "c"] & \boxed{T_{sol}} \arrow[r, "c"] & \boxed{T_{hill}} \arrow[r, "c"] & {}\end{tikzcd}$$

Figure 10: An initial program for Simple Hill Climbing

mines when all possible points have been generated). None of these subtests says anything about unimodality, mesas, or ledges. The unimodality property can be represented as

$$P_{uni} \equiv \forall s[\forall op\ T_{appl}(op, s) \supset T_{better}(s, F_{apply}(op, s))] \supset T_{best}(s, S).$$

This says that local optimality (in which every operator *op* applied to $s$ yields an inferior state) implies global optimality. The absence of mesas and ledges can be represented as

$$P_{mesa} \equiv \forall s, s', op\ F_{apply}(op, s) = s' \supset [T_{better}(s, s') \vee T_{better}(s', s)].$$

This says that if two states $s$ and $s'$ are adjacent, then one is better than the other.

In previous sections, we have seen that test incorporation can require decomposing and reformulating the generator and the test. However, in the case of SHC, no reformulation of the generator or of the test alone can produce the $P_{uni}$ and $P_{mesa}$ properties. This is because these properties arise from the interaction between the structure of the generator and the structure of the test. $P_{uni}$ and $P_{mesa}$ each refer to parts of the generator (i.e., $T_{appl}$ and $F_{apply}$) and parts of the test (i.e., $T_{better}$). Hence, simple hill climbing provides a counterexample to the test incorporation hypothesis.

Further improvement of the *find best* procedure from Figure 3 requires making explicit $P_{uni}$ and $P_{mesa}$. If these properties are made explicit, then the naive problem solver can be reformulated to have a different test containing two subtests, $T_{sol}(c)$ and $T_{hill}(c)$. $T_{hill}(c)$ is defined as $\forall op\ T_{appl}(op, c) \supset T_{better}(c, F_{apply}(op, c))$. Notice that neither test requires $S$ in order to determine whether $c$ is optimal. Hence, we can improve the search procedure as shown in Figure 10. This procedure simply generates elements from the space and checks each one to see if it is a local maximum.

Now suppose that $G_{el}$ is a path-extending generator of the kind shown in Figure 8. There are two incorporations that can usually be done. First, $T_{sol}$ can be incorporated into $G_{step}$ so that every path (state) generated by the incremental generator is a legal point in the space. Second, as mentioned above, $T_{appl}$ can be incorporated into $G_{step}$ so that only operators applicable to the path selected by $T_{path}$ are generated. These two incorporations produce a more efficient generator of all possible points in the space.

Now we arrive at the incorporations crucial to hill climbing. The first incorporation changes $M_{path}$ from a cumulative memory to a memory for a single path. It also inserts $T_{better}$ after $F_{apply}$ to check if the newly derived state is better than the previous one. The new state is only stored in $M_{path}$ if it is better than the previous state. These changes do not incorporate all of $T_{hill}$. They only incorporate the necessary condition that if two states $s$ and $s'$ are adjacent and $s$ is known to be better than $s'$, then $s'$ cannot be the optimum. Figure 11 shows the resulting data-flow graph.

The second important incorporation completely eliminates the need to test $T_{hill}$. This incorporation can only be performed if $G_{step}$ is capable of producing a stop signal when it has generated all possible applicable steps. In such cases, this means that none of the steps generated by $G_{step}$ were acceptable to $T_{step}$ or $T_{better}$; hence, all steps applicable to $p$ lead to states that are worse than $p$, and hence, $p$ is the optimum. To carry out the incorporation, we connect the stop signal from

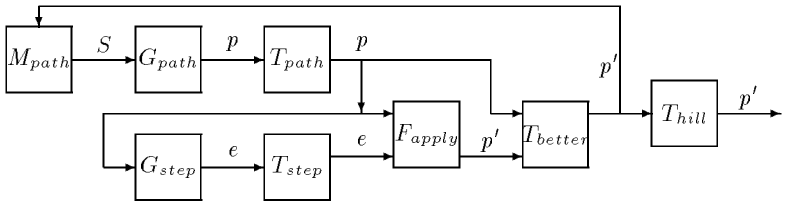

Figure 11: Improved hill climbing

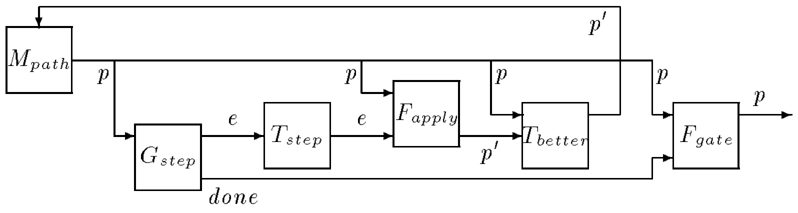

Figure 12: Final version of Simple Hill Climbing

$G_{step}$ in such a way that it gates $p$ to the output. Figure 12 shows the traditional procedure for simple hill climbing.

From this analysis of SHC, we can see that the test incorporation hypothesis is not correct. In order to derive all of the weak methods, test incorporation needs to be augmented by other transformations that are able to reformulate the generate-and-test problem solver so that further test incorporations can be applied. We discuss this problem in more detail below.

## 5.4 Generate-and-Test Avoiding Duplicates

The final weak method that we wish to derive is the simple method of avoiding the generation of duplicate candidates. As mentioned above, we do not assume that our generators are irredundant—they may produce duplicates. For any problem in which the test is expensive to apply, it is worthwhile to modify the generator so that it does not produce duplicates. This can be done, for example, by recording in a memory ($T_{fail}$) all items that have failed the test and passing new items out to the test only if they have not been previously rejected (see Figure 13).

At first glance, the decision to avoid duplicates appears to be another example of a program improvement that does not involve test incorporation. No additional test is being satisfied as a result

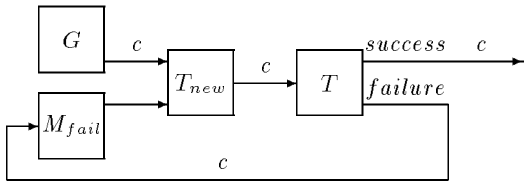

Figure 13: Generate-and-test avoiding duplicates

of this improvement to the program. However, if we consider what happens when the procedure is executed, we see that information is flowing from the test back to the generator—information about what items have failed to pass the test. This information is incorporated into the generator so that it doesn't make the same mistake in the future. We call this *run-time* test incorporation—that is, incorporation of test information revealed during the execution of the problem solver.

## 6 Discussion

The preceding derivations demonstrate that test incorporation provides a unifying framework for many kinds of program optimizations. The general procedure for performing an incorporation involves (a) analyzing the internal structure of the test to find a subtest $T_{sub}$ to incorporate, (b) analyzing the internal structure of the generator to find a place in which $T_{sub}$ can be incorporated, and (c) carrying out the incorporation. Steps (a) and (b) may involve sophisticated reasoning. In order to carry out this analysis, the programmer must have access to the internal structure of the generator and the test. Indeed, in the combination safe example, the safe cracker does not have access to the internal structure of the test (i.e., the safe). An exhaustive generate-and-test method must be used, and no test incorporations are possible (except avoiding duplicate combinations). The converse case, in which the generator is inaccessible, arises as well. Consider an inflexible program for displaying bulletin-board messages (i.e., the generator). The reader of these messages can recognize which messages 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.

While test incorporation provides a fruitful tool for analyzing the weak methods, it is also clear that test incorporation cannot, by itself, provide a general theory of AI methods. There are two major problems.

First, the SHC example shows that the test incorporation hypothesis is false. Not all of the weak methods can be derived via test incorporation. The difficulty is that the generator and the test in the naive generate-and-test procedure are very decoupled. All that the test knows about the generator is that it produces candidate data items in a certain format. The generator knows nothing about the test. This decoupling is deliberate—we wanted the test to be the repository for virtually all knowledge about the problem so that the naive generate-and-test procedure would be very general and flexible. However, we can see in the SHC case that sometimes the successful incorporation of test knowledge requires that the test first be reformulated to take into account the internal structure of the generator. This reformulation is not, itself, a test incorporation.

The second failing of test incorporation as a general theory of methods is that it does not provide a complete account of the tradeoff between between generality and power. It is certainly true that test incorporations reduce the generality of programs. However, there are other determinants of program generality—namely, the degree to which the generator and the test can each be factored into weakly interacting subcomponents. This ability to be factored is critical to effective test incorporation. The incremental path-extension generator, for example, is much more flexible than a random generator that samples with replacement from some set. Many different kinds of test information can be incorporated into the path-extension generator. Indeed, all of the basic search methods (depth-first search, breadth-first search, A*, beam search, etc.) can be obtained via simple incorporations to the path-extension generator. To develop a general theory of methods, we need to acquire some understanding of the kinds of generators that can serve as the targets of incorporations. Similarly, we need to study the kinds of test information that can be incorporated.

Despite these difficulties, test incorporation transformations provide an important foundation for building a theory of methods. There are several directions that we are pursuing in our attempts to construct such a theory.

We have conducted preliminary test incorporation analyses of some expert systems (e.g., DENDRAL, MYCIN, and EL). The naive generate-and-test versions of these systems include large amounts of knowledge in their tests. The problem-solving strength of these systems—in comparison to the weak methods—derives from the fact that this knowledge has been incorporated into their (often implicit) generators (see Amarel, 1982). These incorporations are made during system development by the knowledge engineer and the expert.

We are also pursuing the design of an automated "incorporation problem solver" (IPS) that performs test incorporations along the lines already investigated by Tappel and Mostow. The IPS is itself a *find best* procedure that attempts to find the most efficient version of a given generate-and-test program (c.f. Kant, 1976). This perspective raises the possibility of applying the IPS to itself (Kahn, 1983).

The final direction we are pursuing concerns run-time incorporation. Most of the incorporations discussed in this paper have required that all the knowledge in the test is available for incorporation before problem-solving begins. Often, however, the knowledge required to apply incorporations does not become available until problem solving is under way. The derivation of avoid duplicates points out the importance of incorporations that occur at run-time. The avoid duplicates procedure can be viewed as a combination of a simple generate-and-test procedure and a very simple IPS that performs the run-time incorporations.

If we consider intelligence to be the ability to apply knowledge effectively, then a system is more intelligent to the extent that it is able to perform test incorporation. Test incorporation converts knowledge from an ineffective, static form to a more effective, generative form. Our preliminary analyses indicate that intelligent problem-solving is characterized both by the procedures employed to solve problems directly and by the techniques applied to incorporate knowledge into those procedures.

## 7 Acknowledgments

The authors wish to thank Bruce Porter, Nicholas Flann, and Dan Corpron for critical readings of the text. This research was supported in part by the National Science Foundation under grant numbers IST-8519926 and DMC-8514949.

## 8 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. 1982. Expert behavior and problem representations. Rep. No. CBM-TR-126, Department of Computer Science, Rutgers University.

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.

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.

Laird, J. E. 1984. Universal subgoaling. Rep. No. CMU-CS-84-129. Doctoral Dissertation, Department of Computer Science, Carnegie-Mellon University.

Laird, J. E., and Newell, A. 1983a. A universal weak method. Rep. No. CMU-CS-83-141, Department of Computer Science, Carnegie-Mellon University.

Laird, J. E., and Newell, A. 1983b. A universal weak method: summary of results. *Proceedings of IJCAI-83*, Los Altos: Morgan-Kaufman. 771–773.

Laird, J. E., Rosenbloom, P. S., and Newell, A. In press. Chunking in Soar: The anatomy of a general learning mechanism. To appear in *Machine Learning.*

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.

Paige, R., and Koenig, S. 1982. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems.* 4 (3) 402–454.

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, 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.