A Planning Problem Solver Based on Look-Ahead in Stochastic Game Trees

ERIK J. SANDEWALL

Uppsala University, Uppsala, Sweden*

ABSTRACT. This paper demonstrates how methods from game-playing programs can be applied to heuristic search problems. The game program look-ahead corresponds to a planning process for the heuristic search. Pruning methods, like the alpha-beta algorithm, carry over with small modifications. These techniques are then applied to the General Problem Solver, and an algorithm for a Planning Problem Solver is formulated. Sufficient conditions are given which guarantee that the Planning Problem Solver always sprouts the solution tree in a direction that minimizes the expected length of the remaining solution.

KEY WORDS AND PHRASES: problem-solving, planning, look-ahead, game against nature, stochastic

CR CATEGORIES: 3.64, 3.66

Introduction

Tree searching is an important component in many artificial intelligence programs. Two types of trees can be distinguished: (1) *game trees*, which occur when the computer is set to play a nonrandom game (e.g. chess or checkers); and (2) *gameless trees*, which occur in labyrinth searches; problem-solving as exemplified by the General Problem Solver (Newell 1960 [3], 1960 [4], 1961 [5]); heuristic proof-procedures (Gelernter 1959 [1], Slagle 1968 [10]); etc.

Much work has been done on game-playing, and some hard-core results are available. Most important, there is the alpha-beta method (for a concise description, see, e.g. Slagle 1967 [9]) through which some branches in the tree can be ignored ("pruned") with a corresponding reduction in search time. There is also much experience with learning in game-playing programs; see, e.g. Samuel 1959 [6], 1967 [7].

The applications that require gameless trees seem to be much more general and important than the board-game application. One would hope, therefore, that the results from the study of game trees can be carried over to gameless trees. However, no such extension has (to our knowledge) yet been performed.

The purpose of this paper is to demonstrate that one method involving gameless trees (i.e. the General Problem Solver) can be reformulated in terms of generalized game trees. This makes it possible to apply game-tree methods, e.g. the alpha-beta method. Using the game-tree interpretation, we then proceed to include a planning feature in the General Problem Solver and to formulate a Planning Problem Solver. In Section 7 we will prove that under certain given conditions, the Planning Problem Solver always sprouts the solution tree in a direction which minimizes the

* Computer Sciences Department. This research was supported by the (Swedish) Research Institute of National Defense (beställning 709885, 714257).

Journal of the Association for Computing Machinery, Vol. 16, No. 3, July 1969, pp. 364–382.

expected length of the remaining solution. In the final section, Section 8, the practical use of the Planning Problem Solver is discussed.

1. *The GPS Method*

The following description of the General Problem Solver (GPS) has been extracted from Newell 1960 [4], 1961 [5].

Consider a set P (possibly infinite) of *objects* and a finite set Q of *operators*. For each operator $q \in Q$, there is a set $d(q)$, the domain of q, for which $d(q) \subseteq P$. Also, for each $q \in Q$ and each $p \in d(q)$, $q(p)$ exists and is a member of P.

The purpose of the GPS is to solve the *transformation problem*, which can be phrased as follows. Given $p \in P$, $m \in P$, find a sequence $\langle q_1 , q_2 , \cdots, q_k \rangle$ (where each $q_i \in Q$) such that

$$q_k(q_{k-1}(\cdots q_2(q_1(p)) \cdots)) = m.$$

For a more concise formulation, we introduce the function k where $k(p)$ is the set of all objects p, $q(p)$, $q(q'(p))$, $\cdots$ for arbitrary q, q', etc. Formally,

$$k(p) = \{p\} \cup \{q(x) \mid q \in Q \wedge x \in k(p)\}.$$

The transformation problem can then (somewhat loosely) be formulated as: *identify m in $k(p)$*.

If nothing is known about the functions in Q, then this problem can only be approached through exhaustive search. The GPS method assumes and utilizes the following information:

(1) There is a finite set D of so-called *differences*.

(2) There is an (effectively computable) *difference recognition function*

$$r: \quad P \times P \to D.$$

In other words, if p_1 and p_2 are members of P, then $r(p_1 , p_2)$ exists and is a member of D.

(3) There is an *ordering* $\prec$ on the set D. If d_1 and d_2 are members of D, then $d_1 \prec d_2$ is taken to mean "the difference d_1 is more easily resolved than the difference d_2";

(4) There is an *operator selection function*

$$c: \quad D \to 2^Q.$$

In other words, if $d \in D$, then $c(d) \subseteq Q$.

These functions should in the ideal case be interrelated as follows: if p and m are given objects and $q \in c(r(p, m))$, then there should exist some (easily retrievable) object p' in $k(p) \cap d(q)$ such that $d(q(p'), m) \prec d(p, m)$. In the simplest case, $p = p'$. Roughly speaking, if we take any member q of $c(r(p, m))$ and apply it to p (possibly after some preliminary transformations), then the result has taken us some way toward the "target" m.

For practically arising sets P and Q, this criterion can not always be satisfied. This is especially true as we need explicit, easily computable expressions for the functions r and c. Therefore we must be satisfied if the above ideal situation holds for most p, m, and q. The GPS includes a facility which takes care of the exceptions.

The GPS is a method for using these functions, r and c, to direct a search through

$k(p)$ to retrieve m. The method formulates three new problems (one of which is a generalization of the transformation problem) and gives routines for solving them. Let us first write out these three problems ("goals," in the vocabulary of Newell, et al.):

1. *Transformation problem* (generalized). Given $p \in P$, $M \subseteq P$, identify some member of M in $k(p)$.

Remark. This reduces to the original transformation problem if we select $M = \{m\}$.

2. *Reduction problem.* Given $p \in P$, $M \subseteq P$, identify some $p' \in k(p)$ such that $R(p', M) \prec R(p, M)$ where $R(x, Y) = \min_{y \in Y} r(x, y)$, with the obvious meaning of "min." It is assumed that R, or an approximation to R, is effectively computable for the sets M that can arise.

3. *Application problem.* Given $p \in P$, $q \in Q$, identify some member p' of $k(p) \cap d(q)$ and return $q(p')$ as the solution to the problem.

The GPS gives the following interdependent routines for solving these three problems:

1. *Transformation problem* from p to M. If $p \in M$, then the problem is trivial. Otherwise, find some solution to the reduction problem from p toward M. Then solve the transformation problem from p' to M.

2. *Reduction problem* from p toward M. Select some $q \in c(R(p, M))$ and find the solution to the application problem of q on p.

3. *Application problem* of q on p. Find a solution p' to the transformation problem from p to $d(q)$. Then $q(p')$ is the solution to the application problem.

These three routines call each other recursively, and recursion can only terminate in the degenerate case of the transformation problem. As we see, the GPS consists of two parts: (a) an *iterative part*, which says: to solve the transformation problem from p to M, select some q and solve the transformation problem from $q(p)$ to M instead; and (b) a *recursive part*, which says: in case $q(p)$ should be undefined, then obtain it from a solution to the transformation problem from p to $d(q)$.

In Sections 2–5 we are exclusively concerned with the iterative part, i.e. with the transformation and reduction problems. Only in Sections 6–7 is the recursive part considered.

To close our eyes to the application problem, we introduce the following very natural convention: If p is not a member of $d(q)$, then $q(p)$ denotes "the" solution to the application problem of q on p. This solution may not be unique but that will not cause any problems. To account for the case where there is *no* solution, we introduce one more object $p^0 \in P$ and decide that in such cases, $q(p) = p^0$. We also have $q(p^0) = p^0$ for all q.

2. *Look-Ahead in the GPS Method*

Our routine for solving a reduction problem from p to M says: select *some* q in $c(R(p, M))$ and $\cdots$. The purpose of this section is to discuss methods for choosing q. This gives us the first piece of the Planning Problem Solver.

In [5] (Newell 1961) this problem is managed by a heuristic rule which says, "Never try a subproblem if it is harder than one of its superproblems." In practice, this means that if $c(R(p, M)) = \{q_1, q_2, \cdots, q_k\}$, then their GPS evaluates $q_i(p)$ for all i and then identifies those q_i which satisfy the ideal,

$$R(q_i(p), M) \prec R(p, M).$$

All other q_i are dropped. Although it is not explicitly stated, one would assume that their GPS gives highest priority to the q_i which "minimizes" $R(q_i(p), M)$.

This method is "blind"; it does not perform any look-ahead. By contrast, a human problem-solver often makes a *plan* when he attempts to solve a complicated problem. He says, "First I will do this (I think it is relevant, and I think I can do it); then I will do that . . . ," etc. Only when the plan has been devised, does he set out to solve the actual problem. He follows his plan and revises it when its predictions do not fit in.

We will formulate such a planning process in strict terms so that it can be performed by a computer. Our plan is set up not in terms of objects but in terms of *images*, which contain some of the information of the original objects. In an actual problem environment, each image might e.g. be a vector with Boolean components, which indicate whether certain features are present in the object. Accordingly, we assume a finite set $I = \{i^0, i^1, \cdots, i^n\}$ of images, and an *image extraction function* h: $P \rightarrow I$, which assigns an image $h(p)$ to each object p. The image i^0 is only the image of p^0, i.e.

$$p = p^0 \Leftrightarrow h(p) = i^0,$$

but for objects other than p^0, it is of course possible and normal that several objects share the same image.

In the ideal situation, $h(q(p))$ is uniquely determined from $h(p)$ for any p and q, so that the operators q can be considered as unambiguous operators on images. In practice, we have to be satisfied with less. We introduce a probability function v as follows: $v(i, q, i')$ is the probability that $h(q(p)) = i'$ if it is known that $h(p) = i$, while p itself is unknown. Obviously,

$$\sum_{j=0}^{n} v(i, q, i^j) = 1.$$

However, to retain the flavor of the ideal case, we assume that for most i and most q, there exists some j such that $v(i, q, i^j)$ is considerably larger than other $v(i, q, i')$.

With our extended notation from Section 1, $q(p)$ need not always be unambiguous. In such cases we assume that it is possible to assign probabilities to the various solutions, so that the function v is well-defined. Clearly, $v(i^0, q, i^0) = 1$ for all q.

Consider now the following game, which is equivalent to the transformation problem. Two players, A and B, move alternatingly. Initially, the board contains an image i. (There are no squares on the board.) A has the first move and puts an arbitrary operator q on the board. He exercises his own free will in selecting q. B picks up i and q and puts back another image i' on the board. B is assumed to have no free will; the probability that he will put back a given i' is exactly $v(i, q, i')$. When B has moved, A puts another operator on the board and the game continues. Any image or operator can be used any number of times. A wins the game when a member of a prescribed set $L \subseteq I$ comes on the board; he loses when i^0 comes on the board. B is impartial and does not try to help or hinder A.

This game will be called the *lottery game from i to L*. It is completely defined by the quintuple $\langle I, Q, i^0, L, v\rangle$. We assume that A has complete knowledge of the quintuple and ask how he should make his moves to maximize his chances of reaching L, i.e. winning the game, or to reach L as quickly as possible.

Clearly, the tree-search techniques usually applied to nonrandom games can be applied to the lottery game. We assign the value 0 to each branch in the tree which ends in a loss for A and the value 1 to each branch which gives a win for A.

Alternatively, if A is willing to trade a higher chance of winning for a slightly lower chance of winning more quickly, we should assign something like n^{-1} or e^{-n} to an n-step winning branch. Or we can generalize to e^{-hn}, where h is a measure of A's haste.

As with ordinary games, it is impossible to search the tree to its end; in fact, the tree is usually infinite in some branches. A's expectation of a position must therefore be estimated through look-ahead to a certain depth, where a static evaluation function is applied. Extrapolation from the experience with game-playing programs would indicate that such look-ahead pays (i.e. that we are badly off if we apply the static evaluation function immediately), but this has to be tested in experiments.

The technique of searching a lottery tree (i.e. the tree of a lottery game) is studied more closely in the next few sections. Let us now outline the iterative part of the PPS and relate it to the GPS. For simplicity, we assume that D, r, and c for the GPS are given and use them for the PPS.

We select I as $D + \{i^0, i^*\}$ where D is the set of differences, i^0 is the ficticious image for failure, and i^* is an image for success (corresponding to the zero difference). The set D is very apt to be used as images, especially as Newell 1960 [4] suggests that differences will have structure.

The image recognition function h is defined as $h(p) = R(p, M)$, which includes i^* if $p \in M$. The static evaluation function, s, can be selected as we find suitable, but we must have

$$d_1 \prec d_2 \equiv s(d_1) > s(d_2).$$

The target set L is of course defined as $\{i^*\}$. Finally, we assume that somebody has given us a probability function v. The PPS then solves the GPS problems in the following way:

Reduction problem from p toward M. Perform look-ahead in the lottery tree from $h(p)$ to $H(M)$. Select q as the best first move on the basis of this look-ahead and determine $q(p)$.

Remark. We have used the obvious notation $H(M) = \{h(m) \mid m \in M\}$. This function H will be used again.

Transformation problem from p to M. If $p \in M$, return p and terminate. Otherwise, find some solution p' to the reduction problem from p toward M and memorize the look-ahead tree that was then constructed. It is called our *plan*. Then solve the transformation problem from p' to M, using and updating the same plan. Restriction: If we later discover that the static estimates in the original plan were bad, and that another q seems to be more favorable, then back up in the solution tree and use that other q instead.

Application problem. Not considered until Section 6.

The precise formulation, including exact rules for when and how to back up, is given in Section 5. This discussion was only to give the main ideas.

By comparison, the GPS routine essentially performs static evaluation on the B player's nodes in the tree to select the best first move for the A player. In very rough terms, it could therefore be characterized as a one-ply look-ahead method.

3. *The Value and Best First Move of a Lottery Tree*

Our PPS routine for the reduction problem assumes that we can select the best first move in the lottery game from $h(p)$ to $H(M)$. In this section, we strictly define what this means. The definition comes via the definition of the *expectation* of a lottery tree.

Two sets, I and Q, are given as before. A *node* is a nonempty, finite sequence of the form $\langle i_1 , q_1 , i_2 , \cdots \rangle$. It is called an *A-node* if its last component is a member of I and a *B-node* otherwise. If a node t_1 is obtained from a node t_2 by deleting the last component, then t_1 is the *predecessor* of t_2, and t_2 is a *successor* of t_1. The successor of a node t which is obtained by adding x is written $f_x(t)$; the set of all successors is written $F(t)$. The last element x of $f_x(t)$ is called the *face* of $f_x(t)$, and $f_x(t)$ is said to *display* x.

Let T be a set of nodes. A node in T is called a *root* in T iff its predecessor is not a member of T. T is called a *branch* iff it has only one root, and a *tree* iff it is a branch whose root has only one component.

A branch is *complete* iff it contains all successors of all its nodes. It follows that every complete branch is *infinite*, i.e. has infinitely many members. A branch T' whose root is t' is a *subbranch* of a branch T iff it is the largest possible subset of T which is a branch with t' as root. (Thus a subbranch of a complete branch is always complete.) A branch T' is a *stump* of a branch T iff it is a subset of T and has the same root as T.

The *depth* of a node is one less than its number of components. Thus the root of a tree has depth zero.

Let T be a branch. A node in T is *terminal* iff none of its successors is a member of T. T is said to be *nicely pruned* iff every node is either a terminal A-node or all its successors are members of T. In particular, every complete branch is nicely pruned.

Let T be a finite, nicely pruned branch. T is of *depth* k iff all its terminals are of depth k.

A *path* is a sequence of nodes $\langle t_1 , t_2 , \cdots \rangle$ where each t_k is a successor of t_{k-1}.

We assume that L is a fixed subset of I and that i^0 is a member of $I - L$. Also, we assume a *static evaluation function* s which assigns values between 0 and 1 to each member of I, and a *probable successor function* v as in the preceding section. These two functions are generalized to A-nodes as follows:

$$s(\langle i_1 , q_1 , i_2 , \cdots , i_k \rangle) = s(i_k),$$

$$v(\langle i_1 , q_1 , i_2 , \cdots , i_k , q_k , i_{k+1} \rangle) = v(i_k , q_k , i_{k+1}).$$

Finally, we assume a real, monotonic function g of a real variable satisfying $g(0) = 1$, $g(\infty) = 0$. (Occasionally, we also consider the case where $g(x) \equiv 1$.)

Let T be a finite, nicely pruned branch. The *expectation* $e(t)$ of a node $t \in T$ is defined by the first applicable case in the following table:

$g(k)$	if t of depth k displays a member of L
0	if t displays i^0
$s(t)g(k)$	if t of depth k is a terminal node
$\sup_{t' \in F(t)} e(t')$ or equivalently, $\sup_{q \in Q} e(f_q(t))$	if t is an A-node
$\sum_{t' \in F(t)} v(t')e(t')$ or $\sum_{i \in I} v(f_i(t))e(f_i(t))$	if t is a B-node

To indicate that the expectation has been computed on T, we sometimes write $e_T(t)$.

The *expectation* $E(T)$ of T is the expectation $e(t_0)$ of the root t_0 of T. If t is a nonterminal A-node, and $\bar{q}$ maximizes $e(f_q(t))$, then $\bar{q}$ is a *best move* to t. A best move to t_0 is called a *best first move* in T.

An actual PPS process operates in terms of finite branches and selects best first moves according to this definition. For theoretical purposes, we also need a definition of the best first move in a complete branch. For this we must first define the expectation of a complete branch and prove that it exists.

Let T be a complete branch whose root is of depth J, and let $T_1 \subseteq T_2 \subseteq \cdots$ be an infinite sequence of stumps of T where each T_j is of depth $J + 2j$, if the common root of T, T_1, T_2, etc. is an A-node, and $J - 1 + 2j$ otherwise. This guarantees that all terminals are A-nodes; so each T_j is nicely pruned. Let one arbitrary static evaluation function s be given, and let also s_0 and s_1 be two alternative static evaluation functions, whose values are identically zero and identically one, respectively. Finally, let $E_0(T_j)$, $E(T_j)$, and $E_1(T_j)$ denote the expectation of T_j which is obtained by using s_0, s, and s_1, respectively.

With these assumptions, the following results hold for all j.

LEMMA 3.1. *Let e_1 and e_2 be two different expectation functions, derived for the same function v, but possibly for different s and T. If t does not display i^0 or an L, and*

$$(x)\ e_1(f_x(t)) < e_2(f_x(t)),$$

then

$$e_1(t) < e_2(t),$$

and similarly for the relations $\leq$, $>$, $\geq$, and $=$.

PROOF. The proof is immediate.

LEMMA 3.2. *For each subbranch T_j' of T_j,*

$$E_0(T_j') \leq E(T_j') \leq E_1(T_j').$$

PROOF. The proof is by recursive application of Lemma 3.1.

LEMMA 3.3. *If subbranches T_j' of T_j and T_{j+1}' of T_{j+1} have the same root, then $E_0(T_j') \leq E_0(T_{j+1}')$.*

PROOF. The proof is by recursive application of Lemma 3.1. In the first step of recursion (i.e. when T_j' consists only of a terminal), inequality can occur as some terminal of T_{j+1}' displays a member of L.

LEMMA 3.4. *With the assumptions of Lemma 3.3, $E_1(T_j') \geq E_1(T_{j+1}')$.*

PROOF. The proof is as for Lemma 3.3. Inequality can occur as some terminal of T_{j+1}' displays i^0.

These lemmas are used for only the special case where $T_j' = T_j$.

LEMMA 3.5. $E_1(T_j) - E_0(T_j) \leq g(2j)$.

PROOF. The proof is by recursive application of Lemma 3.1.

THEOREM 3.6. *The following limits exist and are equal as j tends to infinity:*

$$\lim_j E_0(T_j) = \lim_j E(T_j) = \lim_j E_1(T_j)$$

PROOF. The proof is immediate.

This common limit is called the *expectation of* T and is denoted $E(T)$.

For use in Section 4, it is nice to have

LEMMA 3.7. *If T is a complete branch, T' a finite stump of T, and $\bar{j}$ and $\bar{k}$ the minimum viz. maximum depth of terminals of T', then*

$$E_0(T_j) \leq E_0(T') \leq E_0(T_k), \quad E_1(T_j) \geq E_1(T') \geq E_1(T_k),$$

where T_j is of depth $\bar{j}$ and T_k of depth $\bar{k}$.

PROOF. The proof is by recursive application of Lemmas 3.3 and 3.4.

COROLLARY 3.8. *If T is a complete branch, and $T_1, T_2, \cdots$ is an infinite sequence of finite stumps of T whose minimum terminal depth tends to infinity, then*

$$\lim_j E(T_j) = E(T).$$

In Section 4, we look for finite stumps T' such that $E(T')$ approximates $E(T)$. Corollary 3.8 gives us full freedom in the selection of T'.

Let T be a complete branch, let t be an arbitrary node in T, and let T' be the subbranch of T which has t as root. The *ultimate expectation* $E(t)$ of t is defined as $E(T')$. It follows that the expressions for $e(t)$ given earlier apply for $E(t)$ as well if E is inserted for e throughout. (However, $E(t)$ is not computable from these expressions.) In particular, if the root t_0 of T is an A-node, then

$$E(T) = E(t_0) = \sup_{q \in Q} E(f_q(t_0)).$$

A $\bar{q}$ in Q for which the supremum is attained is said to be a *best first move* in T. This is what we need for the reduction routine in our PPS.

4. *Practical Computation of Expectation*

In practice, we must of course approximate $E(t)$ with $e(t)$ computed on some suitable finite stump when we select the best first move of a lottery tree. Because of this approximation, some disturbances occur in the problem-solving process; that is the topic of Section 5. The present section is devoted to methods for selecting the stump which yields the best approximation for given computational resources. The process of cutting off branches to obtain a good stump is referred to as *pruning*.

Adaptive depth. Stumps T_j of depth $2j$, which we used for theoretical purposes in Section 3, are likely to be bad choices from the efficiency point of view. Consider some B-node t and its successor A-node t', which satisfies $v(t') \approx 0$. Such t' should be quite common. Clearly, $e(t')$ affects $e(t)$ very little, and the expectation of the total tree even less. It can therefore be computed with a very crude method, e.g. as $s(t')$. More radically, it can be approximated with the weighted average of the more probable successors of t. Formally, we select an $n' < n$ (where n is the number of images) and use the approximation

$$e(t) = \sum_{j=1}^{n} v(t_j)e(t_j) \approx \sum_{j=1}^{n'} v(t_j)e(t_j) / \sum_{j=1}^{n'} v(t_j)$$

where t_j is the jth successor of t; $v(t_j) > v(t_k)$ for each $j \leq n'$ and each $k > n'$.

Obviously, we do not have to specify the stump first and evaluate it afterward. Instead, we can start out on the recursive, infinite process for computing $E(t)$ and decide to prune whenever we encounter a node t' for which $E(t') \approx 0$. This pruning method can be generalized.

Pruning on probability. In principle, we desire that each $v(t')$ be close to either 0 or 1. In practice, we have to live with medium-sized probabilities as well. For them, the above pruning criterion should not be applied. However, it is natural to prune paths which contain several successive medium-sized probabilities.

To do this in a systematic fashion, we select a *pruning level*, which should be a small positive number. As we work down recursively into the lottery tree, we keep track of the access probabilities for the various branches and prune as this probability falls below the pruning level. This scheme clearly contains the previous simple pruning scheme as a special case.

The pruning on probability scheme must be supplemented with a *pruning on maximum depth* scheme, which says that whatever happens, we will not dig deeper than nodes whose depth is K units larger than the depth of the root, for a fixed K. Together, these are called schemes for *forward pruning*, in contradistinction to the following scheme.

Backward pruning. This is a generalization of the alpha-beta method (cf., e.g. Slagle 1967 [9]). Consider the evaluation of $e(\bar{t})$ for a nonterminal, not-in-L, not-i^0 A-node $\bar{t}$. $E(\bar{t})$ is defined as the supremum over some $e(t_{(j)})$, $j = 1, 2, \cdots, r$. Under favorable circumstances, it is not necessary to evaluate all $e(t_{(j)})$, at least not completely, to find the value of $e(\bar{t})$. Suppose we already have the partial result

$$\alpha = \sup(e(t_{(1)}), e(t_{(2)}), \cdots, e(t_{(k)}))$$

for some $k < r$ and intend to compute $e(t)$ for $t = t_{(k+1)}$. (We have selected our notation so that t is a B-node, just as in the discussion of adaptive depth, above.) If we have access to an upper bound on $e(t)$ and know that $e(t) < \alpha$, then the exact value of $e(t)$ is clearly irrelevant. Such an upper bound may be obtained in two ways:

(a) *before* the recursive evaluation of $e(t)$ has been commenced. For example, we can make a quick estimate using some static evaluation function which applies to B-nodes, and ignore $e(t)$ if the estimate is considerably smaller than α;

(b) *during* the recursive evaluation of $e(t)$. Let us define

$$y_m = \sum_{j=1}^{m} e(t_j),$$

$$z_m = \sum_{j=1}^{m} v(t_j)\ e(t_j),$$

where t_j is the jth successor of t, as before. We recall that n is the number of members of i and conclude that $y_n = 1$ and $z_n = e(t)$. It follows that

$$e(t) = z_m + \sum_{j=m+1}^{n} v(t_j)\ e(t_j) \leq z_m + \sum_{j=m+1}^{n} v(t_j) = z_m + 1 - y_m .$$

Therefore, as we compute the successive z_m for $m = 1, 2, \cdots, n$, we keep track of the corresponding y_m. (This is a small extra effort.) At each step we check whether $z_m + 1 - y_m \leq \alpha$ and terminate if this is the case. This is our backward[1] pruning method.

The above deduction of the backward pruning method guards pessimistically

[1] The reason we call it "backward" is that pruning is performed when we are already "sitting" on the branch that we intend to cut. On the other hand, if $e(t)$ is suppressed after an estimate of type (a), then we clearly have still another case of forward pruning.

against the possibility that all $e(t_j)$ for $j = m + 1, \cdots, n$ will equal 1. Of course, we can do better with estimates, as discussed under "adaptive depth."

The backward pruning method is a generalization of the alpha-beta method for ordinary game trees. This is immediately seen if we decide that v is not actually random but always gives $v(f_i(t)) = 1$ for the i which minimizes $e(f_i(t))$. Our lottery game then degenerates into an ordinary idealized game, and our pruning method goes into the ordinary alpha-beta method. (More precisely, it goes into the "alpha" part of the alpha-beta method, which operates symmetrically on both players. Such symmetry is not possible for lottery games.)

The alpha-beta method is very efficient for reducing the search effort in ordinary game trees. Unfortunately, we can not expect that the generalized method will be quite as good. With the pessimistic estimate of $e(t_j)$ as $+1$, it can be used only when $y_m \approx 1$, which means that the remaining $v(t_j)$ are close to zero, which means we would have skipped the corresponding $e(t_j)$ anyway with forward pruning. To make the method useful, better estimates of $e(t_j)$ are needed. But the use of improved estimates always means taking a certain risk of underestimating $e(t_j)$.

5. *The Planning Problem Solver*

The purpose of this section is to give a precise definition of the Planning Problem Solver (PPS) method and to explain how it works.

We use the same notation as in previous sections. Let a transformation problem from p to M be given and consider first the following happy state of affairs, where the job of the problem solver is trivial:

(1) $(i)(q)(\exists i')\ v(i, q, i') = 1$;
(2) $(p)(q)(i)\ v(h(p), q, i) = 1 \equiv h(q(p)) = i$;
(3) $(p)\ [h(p) \in H(M)] \supset [p \in M]$;
(4) $E(t)$ is known for all t in T, the lottery tree from $h(p)$ to $H(M)$.

It is easily proved that with these assumptions, and for each node t in T, we have $E(t) = g(2k)$ where k is the length of the shortest solution that goes through t. A shortest solution can therefore easily be retrieved by the following routine: Start in the root of T, select q as the best first move, set $p := q(p)$, advance two steps in T; select a new q, etc.

Condition (3) is easy to fulfill in practical cases by merely including the truth value of $p \in M$ in the feature vector $h(p)$. But conditions (1) and (4) never hold in practice, especially not together.

The failure of condition (1) to hold has the following effect. If our attempts to solve the transformation problem by use of the above routine leads us to a node $\langle \cdots, h(p) \rangle$, then the expectation of this node will *change*. Earlier it had been defined using v, but as we actually apply an operator q on p and become able to inspect $h(q(p))$, the expectation of $\langle \cdots, h(p), q \rangle$ is reset to the expectation of its "factual" successor $\langle \cdots, h(p), q, h(q(p)) \rangle$. But $\langle \cdots, h(p), q \rangle$ had been selected as the *best* successor of $\langle \cdots, h(p) \rangle$; so the expectation of the latter will also change. In fact, this change backs up to the root of the tree. Most important, it may change our opinion about what would have been the best move in a previous step.

The failure of condition (4) to hold may be "overcome" by approximating $E(t)$ with $e_{T'}(t)$ where T' is a finite stump of T. But then we may be forced to (in fact,

we will prefer to) extend T' during the problem-solving process. This causes the same back-up changes of expectation as above.

Unlike "ordinary" games, the problem-solving lottery game permits the A player ($\sim$ us) to back up in the game and select a move other than the one we previously chose. It is a crucial question when and how to use that right. The PPS determines the answer (which we prove to be a good answer) by studying the back-up of expectation change. To express this strictly, we need more notation.

A *planning node* is a finite sequence of the type $\langle i_1 , q_1 , i_2 , \cdots \rangle$; a *solution node* is a sequence of the type $\langle p_1 , q_1 , p_2 , \cdots , p_k \rangle$ where each p_{j+1} is a solution to the application problem of q_j on p_j. Planning nodes are characterized by the same concepts as before (A-node, B-node, face, display, etc.), and wherever applicable, these concepts are extended to solution nodes. The only exception is that the predecessor and successor of a solution node are obtained by removing, viz. adding, two components at the end of the sequence. If s is a solution node, then $f_q(s)$ is obtained by adding q and $q(p_k)$ at the end.

A planning A-node t and a solution node s written on the above forms are said to *correspond* iff they have equal depth, their q_j agree, and $i_j = h(p_j)$ for each j.

A *solution tree* is a finite set of solution nodes. A *planning tree* is a nicely pruned tree of planning nodes. A planning tree T is a *plan for* a solution tree S iff every node in S corresponds to some nonterminal node in T. If this is the case, we have the following additional concepts. A B-node in T is *passé* iff it has a successor which corresponds to a node in S. That successor, which is by necessity unique, is called the *factual* successor.

The set of corresponding A-nodes and passé B-nodes constitute a tree, which is a stump of T. It is called the *base* of T. If t is an A-node in the base, but $f_q(t)$ is not in the base, then $f_q(t)$ is a *bud*.

The *expectation* of a planning node is defined as before, with the following two exceptions:

(1) The expectation of an attempted node is defined as the expectation of its factual successor.

(2) If $\langle i_1 , q_1 , \cdots , i_J , q_J \rangle$ is a bud, then the depth of its (direct or indirect) successor

$$\langle i_1 , q_1 , \cdots , i_J , q_J , \cdots , i_k \rangle$$

is defined as $2(k - J)$. (The depth of a node is used in the definition of expectation.)

We have as an immediate result

COROLLARY 5.1. $E(T) = \sup E(b)$ *where* b *ranges over the buds of* T.

A bud which maximizes $E(b)$ is called a *best bud* or a *focus*.

With this vocabulary, we are now ready for the definition of the iterative PPS.

Let a transformation problem from p to M be given, together with heuristic information on the form of I, h, v, and s; *cf.* above. The *iterative PPS* is the method of solving the transformation problem by the following steps:

1. Define $S = \{\langle p \rangle\}$, and T equal to a finite stump of the lottery tree from $h(p)$ to $H(M)$, large enough to be a plan for S.

2. Perform the following routine iteratively, until some member of S displays a member of M: (2a) select a focus $f_q(t)$ of T; (2b) if s is the node in S which cor-

responds to t, add $f_q(s)$ to S; (2c) add a suitable number of nodes to T, at least so many that it is still a plan for the extended S.

Notice that as $f_q(s)$ is added to S, the focus becomes passé, gets its factual successor, and thereby gets a new expectation.

Intuitively speaking, the expectation $e(t)$ of a node is a measure of the expected length of solutions to the transformation problem. (This point is discussed strictly in Section 7.) When the PPS selects the best bud in the planning tree, it therefore actually minimizes (or at least thinks it minimizes) the expected length of the solution. We can now see the reason for the above redefinition of the depth of a node: the PPS is interested in minimizing the *remaining* path to the solution, not the total path.

The PPS has been defined in terms of the best buds. We can relate this to the back-up-and-try-another-solution way of thinking, described at the beginning of this section, by the following observation. Let b be the focus, $e^- \leq e(b)$ be the expectation of the second-best bud, and $f_i(b)$ the factual successor of b after sprouting has occurred. If $e(f_i(b)) > e^-$, then the PPS must select a successor of $f_i(b)$ as its next focus (i.e. proceed on the same path), and if $e(f_i(b)) < e^-$, it must select the old second-best bud as its new focus (i.e. back up in T and in S).

This concludes our description of the iterative PPS.

6. *The Recursive PPS*

During actual performance of an iterative PPS process, according to Section 5, we may get caught in a long and cumbersome evaluation of $q(p)$ for nontrivial application problems. In this section we extend the PPS so that such complications can be predicted and planned for.

The obvious technique for retrieving nontrivial $q(p)$ is to initialize a new, iterative PPS process, this time from p to $d(q)$. The subprocess needs its own solution tree, planning tree, static evaluation function, etc.

In the simplest case, we run the subprocess to its end, and return to the top-level PPS result in hand. However, we might have started this subprocess with a very high expectation, only to discover that it was not as easy as we thought. In such cases we would like to interrupt the subprocess and try some other bud in the main process.

One way to do exactly that would be the following. As we proceed in the subprocess, the expectation of the subprocess planning tree, T', changes dynamically. (It is equal to the expectation of the best bud.) In particular, it goes down if things go badly in the subprocess. At each step of the subprocess, we therefore send back the expectation of T' to T as a new and better expectation of its focus. If this value goes down, then some other bud takes over as focus and the subprocess is frozen. It may be resumed if we later revert to the old focus.

However, that method is not altogether satisfactory. The expectation of the focus which T provides is based on a long-range look-ahead, where the ply from p to $q(p)$ is only one small step. The expectation of T', on the other hand, is based on a close analysis of possible paths from p to $q(p)$. These two estimates complement rather than exclude each other. We would like to combine them.

For that purpose we notice at first that because of look-ahead in the top-level tree,

we may prefer one member of $d(q)$ to another as a target in the subprocess. We therefore generalize the lottery game to the following

Lottery Game with Appreciation Function. Given I, L, i^0, v, and g as before, but also a function c, which assigns real-number values to the members of L. The game runs as before, with the following modifications: (a) whenever B has put a member i of L on the table, A has an option of continuing or terminating; and (b) if A terminates when each player has made k moves, and an image i is on the table, then A earns $c(i)g(2k)$ points of credit.

In this game, if an i which maximizes $c(i)$ is on the table, it always pays for A to stop. Otherwise, he has to weigh the chances of finding a better i against the decline in the value of $g(2k)$ with continued play.

Clearly, look-ahead can be performed as before, and the lemmas of the preceding sections still hold. We have to define the expectation $e(t)$ of a node t as the first applicable case in the following table:

0	if t displays i^0
$c(t)g(k)$	if t of depth k is terminal and displays a member of L
$s(t)g(k)$	if t of depth k is terminal
$\max\,(c(t)g(k), \sup_{q\in Q} f_q(t))$	if t is an A-node
$\sum_{i\in I} v(f_i(t))e(f_i(t))$	if t is a B-node

Let us return now to the problem with subprocesses in the PPS. Suppose in the top-level lottery game, we have selected a best bud $f_q(t)$ where t corresponds to s which displays p and p is not a member of $d(q)$. We then have to run a subprocess transformation problem from p to $d(q)$ and therefore need a lottery tree from $h(p)$ to $H(d(q))$. Let

$$H(d(q)) = \{i_1', i_2', \cdots, i_k'\}.$$

Suppose we solve the subprocess transformation problem and get to i_j'. Our expectation for the remaining main process is clearly

$$c(i_j') = \sum_{i\in I} v(i_j', q, i)\, e(f_i(f_q(t)))$$

where e is computed in the top-level tree. This is therefore the appreciation function with which we have to run the subprocess. Naturally, the subprocess may initialize lower, second-order subprocesses, which must be run the same way. Therefore, the second-level appreciation function is dependent on the first-level expectation function, and therefore on the first-level appreciation function. This process may proceed recursively and to arbitrary depth.

For consistency, we also consider the top-level lottery game as a game with appreciation, with $c(i) = 1$ for all i in the top-level L set. With this choice of c, the introduction of an appreciation function is a purely formal matter.

We are now ready to define the recursive PPS in strict terms. We do this work rather meticulously, as the definition will be the basis of proofs about special properties in the recursive PPS method.

An *attempt from p to M* is a quintuple $\langle p, M, S, T, c\rangle$ where p is a member of P, M is a subset of P, S is a solution tree with $\langle p\rangle$ as root, T is a plan for S, and c is an appreciation function for $H(M)$.

If $\langle p, M, S, T, c\rangle$ is an attempt, $b = f_q(t)$ is a bud in T, $s = \langle \cdots, p'\rangle$ in S corresponds to t, and $\langle p', M', S', T', c'\rangle$ is an attempt from p' to $d(q)$, then the latter attempt *serves* the former *for* b. An *attempt structure* from p to M is a tree (in the obvious sense of the word) whose root is an attempt from p to M; where every other node serves its predecessor, and where no node is served by more than one successor for the same bud.

Let $a = \langle p, M, S, T, c\rangle$ be an attempt in an attempt structure A. If some successor a' in A serves a for b, then we write $a' = f_b(a)$ and say that b is *attempted*. A bud which is not attempted is *fresh*.

The *expectation* of a node t in T in a in A is defined as above, with the following modifications:

(a) The expectation of a passé B-node is defined to be the expectation of its factual successor.

(b) The expectation $e(b)$ of an attempted bud b in a is defined as $E(T')$ where T' is the planning tree in $f_b(a)$.

(c) The depth of a node in T is counted from the base of T, as described in Section 5.

In the sequel, we often talk about "a bud b in a" when we actually mean "bud b in the planning tree of a"; "$E(a)$" when we actually mean "$E(T)$, where T is the planning tree of a"; etc.

If b is a bud in a, then the two-tuple $\langle a, b\rangle$ is a *bud in* A. If a fresh bud b in a is a best bud in a, then $\langle a, b\rangle$ is a *best fresh bud from* a. Recursively, if an attempted bud b in a is a best bud in a, and $\langle a'', b''\rangle$ is a best fresh bud from $f_b(a)$, then $\langle a'', b''\rangle$ is a best fresh bud from a. A best fresh bud from the root of A is a *best fresh bud in* A.

We now define two operations for extending and reducing an attempt structure. Intuitively, these are the operations of initializing a subprocess transformation problem, viz. of terminating a subprocess successfully and returning the result.

Let $b = f_q(t)$ be a fresh bud in an attempt a in an attempt structure A. The operation of *extending* A *from* $\langle a, b\rangle$ is performed as follows:

Notation. Suppose $a = \langle p, M, S, T, c\rangle$; suppose t in T corresponds to s in S; and suppose the face of s is p'.

1. Define $M' = d(q)$;
   $S' = \{\langle p'\rangle\}$;
   $T' =$ a finite stump of the lottery tree from $h(p')$ to $H(M')$, large enough to be a plan for S';
   c' through

$$c'(i') = \sum_{i \in I} v(i', q, i)\, e(f_i(f_q(t)))$$

   where $i' \in M'$, and e is computed on T with c.

2. Add $\langle p', M', S', T', c'\rangle$ as a new node $f_b(a)$ in A.

It is easily seen that A' is still an attempt structure after this operation.

We now get to the reverse operation. Let $a' = f_b(a)$ be an attempt in an attempt structure A; let

$$a = \langle p, M, S, T, c\rangle,$$
$$a' = \langle p', M', S', T', c'\rangle,$$

and let $b = f_q(t)$, where t in T corresponds to s in S. The operation of *reducing* A *in* a' is possible and defined iff some node in S' displays a member $\bar{p}$ of M', and is then performed as follows.

1. Remove a' and its possible successors from A.
2. Add one more node to S. The new node is obtained from s by adding q and $\bar{p}$ at the end of the sequence s.
3. If desired, add more nodes to T, at least so many that it is still a plan for S in spite of its extension.
4. Insert the new S and T for the old S and T in a and in A.

The *full PPS* (or recursive PPS) is the method of solving a transformation problem from p to M as follows:

1. Define $S = \{\langle p \rangle\}$;
   T = a finite stump of the lottery tree from $h(p)$ to $H(M)$, large enough to be a plan for S;
   c through $c(i) = 1$ for all i in $H(M)$;
   $A = \{\langle\langle p, M, S, T, c\rangle\rangle\}$.
2. Perform the following routine iteratively until some member of S (in the root of A) displays a member of M: (2a) select a best fresh bud $\langle a, b\rangle$ in A; (2b) extend A from $\langle a, b\rangle$; (2c) reduce A in as many nodes as possible.

To understand this specification, notice (1) if $p' \in d(q)$ in the definition of the extension, then the extension to A from step (2b) will immediately be reduced in step (2c); and (2) each time a node $a' = f_b(a)$ vanishes in reduction, one more node is added to the solution tree in a, which may make it possible to reduce a in its turn. Therefore, several reductions may occur in sequence in step (2c).

This terminates our description of the PPS method. In Section 7 we discuss its optimality.

7. *Strategies and Expected Length*

We saw in Section 5 that in each step the iterative PPS tries to minimize the expected length of the remaining solution path. For the recursive PPS, we would be interested in minimizing the expected length of the remaining *total* solution path, up and down the various levels of subprocesses. To accomplish this, it is necessary that the same decreasing function g be used on all levels, i.e. that work on all levels is equally rated.

Another, and less trivial, requirement is that effort on various levels be *additive*. The PPS must be able to select the best choice when offered one alternative with much work in the subprocesses and little work in the main process, and another alternative with a different distribution of work load.

A third requirement is that the planning mechanism account correctly for subprocess calls that may occur a couple of steps ahead in the plan. Essentially, this is the requirement that $v(i, q, i')$, where i is not a member of $H(d(q))$, correctly predicts the outcome of future subprocesses.

Due to space limitations we cannot give the full argument and the full proofs here. We give the main results, and the interested reader is referred to a mimeographed addendum to the paper, available from the author.

Let us make the following assumptions.

Assumption on I: For each operator q and each object p, the image $h(p)$ must

"contain" the truth value of "$p \in d(q)$." In other words, there must exist some function $\bar{d}$ from Q to 2^I such that

$$p \in d(q) \equiv h(p) \in \bar{d}(q).$$

With this assumption we need never be in doubt during the planning process about whether an application problem has been solved or not.

Assumption on g: The fatigue function g must be selected such that $g(x) = e^{-hx}$ for some constant $h > 0$.

Redefinition of v. When i' is not a member of $d(q)$, the value of $v(i, q, i')$ must be slightly modified. The exact definition relies on material that must be omitted here and therefore it cannot be stated. The basic idea with the modification is that $v(i, q, i')$ should express not only the probability but also the "probable number of steps" for getting from i to i' by applying q. The "probable number of steps" K enters as a factor $g(K) = e^{-hK}$.

A consequence of the modification in v is that we obtain

$$\sum_{i' \in I} v(i, q, i') \leq 1,$$

without necessarily having equality. However, this does not disturb the theory.

Finally, we need the following.

General assumption: In any attempt $a = \langle p, M, S, T, c \rangle$, where p is arbitrary, M is either $d(\check{q})$ for an arbitrary $\check{q}$ or the M of the root of an attempt structure,[2] $S = \{\langle p \rangle\}$, T is the complete tree and its root is t, and c is $c^1_{H(M)}$, then, for any given q (which may or may not coincide with $\check{q}$), the quantity

$$\sum_{i \in I} v(l_j, q, i)\, e(f_i(f_q(t)))$$

must be the same for all l_j in $\bar{d}(q)$.

Basically, this assumption says: (1) all members of the top-level M must be equally appreciated; (2) if all members of M on one level are equally appreciated, then all members of M on an immediate sublevel must also be equally appreciated.

Under these assumptions, the following theorem holds:

THEOREM 8.11. *The best fresh bud selected by the PPS in each step of its operation minimizes the expected length of the remaining solution.*

The proof is given in the full (mimeographed) version of this paper.

One might argue against the practical usefulness of this theorem that it is based on the "general assumption" above and that assumption will rarely be satisfied in practice. However, the "general assumption" should be considered as an ideal, not a normal situation. In practice, we always have a certain "skewness" in the appreciation functions. This skewness causes an error in an equality used for the proof of Theorem 8.11, and it should be possible to give bounds for this error. We would thus obtain a further pruning criterion. Such error bounds and pruning criteria must be determined with the techniques of numerical analysis.

8. *Remarks on the Practical Use and the Possible Modifications of the PPS*

For what problems can the PPS be used?—For the same problems as have previously been attacked with the GPS or other heuristic methods. This includes auto-

[2] I.e. it is the M of a transformation problem given to PPS from outside.

matic theorem-proving (cf. Gelernter 1959 [1], Newell 1961 [5], Slagle 1968 [10]) and algebraic simplification (cf. Slagle 1963 [8]).

Are the categories of heuristic information that the PPS presumes (i.e. the functions h, s, and v) usually available?—The use of functions like h and s is a standard method; see, e.g. Slagle 1968 [10]. The introduction of a new function v, on the other hand, is a difficulty. It also makes the selection of h and the set of images less trivial, as these functions have to cooperate. There is no reason to believe that this problem cannot be solved, but the question cannot be answered definitely until experimental results become available.

We regret that we do not have any single example of a problem-solving environment where the v function has been evaluated and has proved useful. The reason we have not yet tried to use our method is that the PPS can be expected to be efficient only in environments with complicated objects and complicated operators. (For simple objects and operators, no essential simplification from objects to images is possible; so look-ahead in the planning tree T is no quicker than actual search of the solution tree S.) Complicated objects and operators are of course the practically useful ones, but implementing such an environment on a computer will require much effort. Therefore, before we computed the v function for some complicated environment, we wanted to know what it could be used for. The results presented here form part of the answer.

Are any modifications of the PPS necessary before it can be applied to these problems?—For the theorem-proving application, yes. There, we could consider p as the given formula to be proved, M as a (given) set of axioms and theorems, and Q as a set of rules of inference, turned backward. The PPS then has to handle operators which yield several subproblems that all have to be solved, i.e. several branches that all have to be extended to M. (As a trivial example, one operator may tell us to prove $A \wedge B$ by first proving A and then proving B.) The extension to the PPS for handling such tasks, is formally trivial. It is intuitively obvious from the results in this paper that the expectation of a set of AND-connected buds is to be computed as the product of the expectations of the individual buds. (This formula is also given in Slagle 1968 [10], although he does not perform any look-ahead and does not explicitly use any fatigue function. In other words, he has $g(x) \equiv 1$.)

However, one problem with this extension is not absolutely trivial: which component in a preferred "bundle" (i.e. a set of AND-connected buds) should the PPS tackle first? One might guess that the easiest problem should be tackled first, because if we are actually working with the wrong bundle, we would like to know as soon as possible, in order to switch focus. The bundle component with the highest expectation would then be the component that is the most likely to deteriorate. However, a closer analysis on this point would be interesting and important.

Can the look-ahead method in the PPS be improved?—Yes, definitely. The function e used here does not "know" that the problem solver will be able to back up during the future problem-solving process. An improved expectation which accounts for this would be defined, for example, through a kind of pseudoexecution of the PPS routine throughout the finite planning tree. Such computations would require time, but they might be worthwhile on the top level.

Is the PPS a good model of human planning in problem-solving situations?—

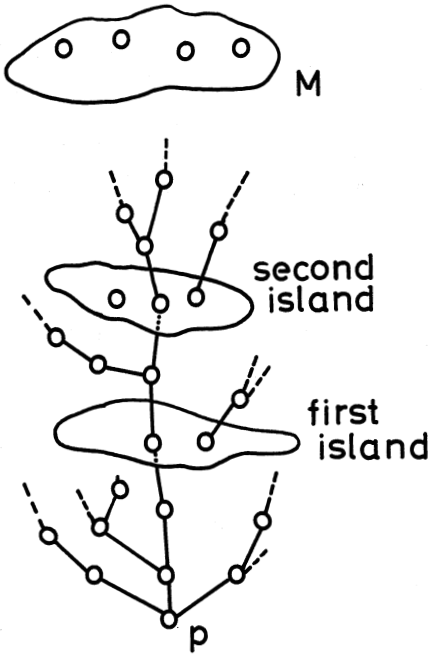

FIG. 1

Although we do not have any ambition to create a psychological model, this question is interesting in view of the present superiority of human beings in the problem-solving field. To the extent that the PPS says: "first I will apply this operator, then I will apply that one . . .," it does not reproduce human behavior, except in very special cases. It seems to us that human planning usually has the character of establishing a few "islands" (a term attributed to Minsky) half-way and quarter-way to the solution, as illustrated by Figure 1. These islands serve to focus the diverging solution tree from time to time.

But such planning can be interpreted as a special case of what the PPS does. A person's "understanding" of the islands would be realized in the PPS as a set of images for each island. Also, for each such island L_k, the PPS would use one operator q_k, which is the identity operator, except that its *domain is restricted to* L_k. A person's knowledge that objects in a certain class i can usually be transformed into this island would be expressed in PPS by

$$\sum_{i' \in L_k} v(i, q_k, i') \approx 1.$$

In this way, the path from the initial object to the first island is pushed down into a subprocess, and the PPS can perform look-ahead beyond the island. By pushing each step from one island to the next into a subprocess, the PPS therefore generates on the top level a gross plan for the entire solution.

As a by-product of the proof of Theorem 8.11, we know that each strategy which runs through subprocesses has an equivalent inside the top-level planning tree. Therefore, it is probably a good idea to subvention the use of subprocesses that take us to islands. This can be done either by using a pessimistic static evaluation

function (like s_0) or by artificially increasing the expectations E_j above what the definition says (Section 6). At a minimum, the E_j should be subventioned with a factor e^{2h} to compensate for the extra "cost" of applying one identity operator, which our scheme (above) makes necessary.[3]

With these conventions, it appears that the PPS is able to imitate reasonably well what people do when they plan an approach to a manipulative problem.

REFERENCES

1. GELERNTER, H. Realization of a geometry theorem-proving machine. In Proc. Int. Conf. Inform. Process., UNESCO, Paris, 1959, pp. 273–281; and in *Computers and Thought*, F. A. Feigenbaum and J. Feldman (Eds.), McGraw-Hill, New York, 1963.
2. MICHIE, D., AND CHAMBERS, R. A. Boxes: an experiment in adaptive control. In *Machine Intelligence 2*, E. Dale and D. Michie (Eds.), Oliver and Boyd, London, 1968.
3. NEWELL, A., SHAW, J. C., AND SIMON, H. A. Report on a general problem solving program for a computer information processing system. Proc. Int. Conf. Inform. Process., UNESCO, Paris, pp. 256–264.
4. NEWELL, A., SHAW, J. C., AND SIMON, H. A. A variety of intelligent learning in a general problem solver. In *Self-Organizing Systems*, M. Yovitts and S. Cameron (Eds.), Pergamon Press, New York, 1960.
5. NEWELL, A., AND SIMON, H. A. GPS, a program that simulates human thought. In *Computers and Thought*, E. A. Feigenbaum and J. Feldman (Eds.), McGraw-Hill, New York, 1963; also Lernende Automaten, Proc. Conf. Learning Automata, Technische Hochschule, Karlsruhe, W. Germany.
6. SAMUEL, A. L. Some studies in machine learning using the game of checkers. *IBM J. Res. Develop. 3* (July 1959), 211–229; In *Computers and Thought*, E. A. Feigenbaum and J. Feldman (Eds.), McGraw-Hill, New York, 1963.
7. SAMUEL, A. L. Some studies in machine learning using the game of checkers, II—Recent progress. Memo No. 52, Stanford A. I. Project, Stanford, Calif.
8. SLAGLE, J. R. A heuristic program that solves symbolic integration problems in Freshman calculus. In *Computers and Thought*, E. A. Feigenbaum and J. Feldman (Eds.), McGraw-Hill, New York, 1963.
9. SLAGLE, J. R., AND DIXON, J. K. Experiments with some programs that search game trees. Lawrence Radiation Lab., Preprint UCRL-70552, Lawrence, Calif.
10. SLAGLE, J. R., AND BURSKY, P. Experiments with a multi-purpose, theorem-proving heuristic program. *J. ACM 15*, 1 (Jan. 1968), 85–109.

RECEIVED JUNE, 1968; REVISED SEPTEMBER, 1968

[3] Actually, subventioning may be worthwhile in other nontrivial application problems as well. Otherwise, the theoretically best strategy will usually not perform any subprocess calls at all, and splitting a problem into subproblems is known to be good heuristics. Compare Michie 1968 2], which is also relevant to the third question above.