Look-Ahead and One-Person Games

Richard S. Rosenberg
Department of Computer Science
University of British Columbia
Vancouver 8, B.C., Canada

September 1972

Abstract

A preliminary investigation of the role of look-ahead in one-person games is presented. The use of look-ahead in comparing the effectiveness of different heuristic functions is discussed. There is a survey of recent work in tree-searching including that of Michie and Doran (1966), Hart, Nilsson, and Raphael (1968), and Pohl (1969, 1970). Based on some of Pohl's results, two theorems are proven in section 2 which suggest a possible use for look-ahead. Some experimental results are presented in section 3 which satisfy to a limited degree the aforementioned theorems and some additional observations are made. In conclusion, an attempt is presented to relate look-ahead to the notion of 'informedness' introduced by Hart, Nilsson, and Raphael. Finally, some further directions for research are suggested.

Look-Ahead and One-Person Games

1. Introduction

During the period that programs for game-playing have been written, a paradigm for a heuristic approach to games has evolved. For most non-trivial games, i.e. those for which an algorithm is unknown or the tree defined by the game is too large to be explored exhaustively in a feasible manner, the standard method is to construct a procedure for evaluating board configurations. We will assume that we are dealing, for the moment, with two-person games of complete information (which excludes such games as poker with bluffing) and which can be associated with a board of some sort, i.e. games such as checkers (Samuel 1959, 1967), chess (Newell, Shaw, Simon 1958; Greenblatt et al 1967), kalah (Slagle, Dixon 1969) GO, Hexapawn, qubic.

The procedure for evaluating configurations (i.e. the current board together with the player to move) defines implicitly a metric for the games. In the simplest cases this procedure is a function which maps from boards to the real numbers. Thus any two boards can be compared by evaluating the function. Under usual conditions the greater the value the better the board. A simple notion of better is closer to a winning board. That is, a player who follows a path through the game tree choosing highly valued boards should improve his chances of winning. He cannot guarantee a win, of course, because the evaluation function (as we shall henceforth call it) only represents an estimation of how good a board is.

There is another way of viewing an evaluation function which is closer to the purpose of this paper. A perfect evaluation function would be equivalent to an algorithm in the following sense. If there exists an algorithm for a game it means that at every turn the player chooses a move which results in a win (or at worst a draw) based on the algorithm. A perfect evaluation function would operate in a similar way. The player using it would apply the function to all possible successor boards of the current one and choose the one which evaluates to a win condition. Note that a perfect evaluation function need only produce such values as 1 for a win, 0 for a draw and -1 for a loss. Of course, it might produce additional values to represent a quick or slow win or loss or even an elegant win.

By choosing a series of boards evaluating to 1, a player would be guaranteed a win. But for most non-trivial games no such perfect evaluation function has been found and so the alternative is to use a heuristic evaluation function together with a look-ahead and back-up procedure. That is, instead of basing a decision upon the values of the immediate successor boards, the player (either computer or human, in principle) will produce a tree of moves down to some arbitrary depth, usually not very deep, evaluate terminal boards and then back up these values to the initial board. The usual form of back-up is mini-maxing which will now be described with reference to Figure 1.

It is player A's turn to move and he is called the MAX player while player B is called the MIN player. In Figure 1, squares denote MAX nodes and circles MIN nodes. The tree of moves is produced down to some depth in this case 2. It is not necessary that the tree be constructed uniformly to a fixed depth or that the branching factor be a constant. See Figure 2 for another possible game tree.

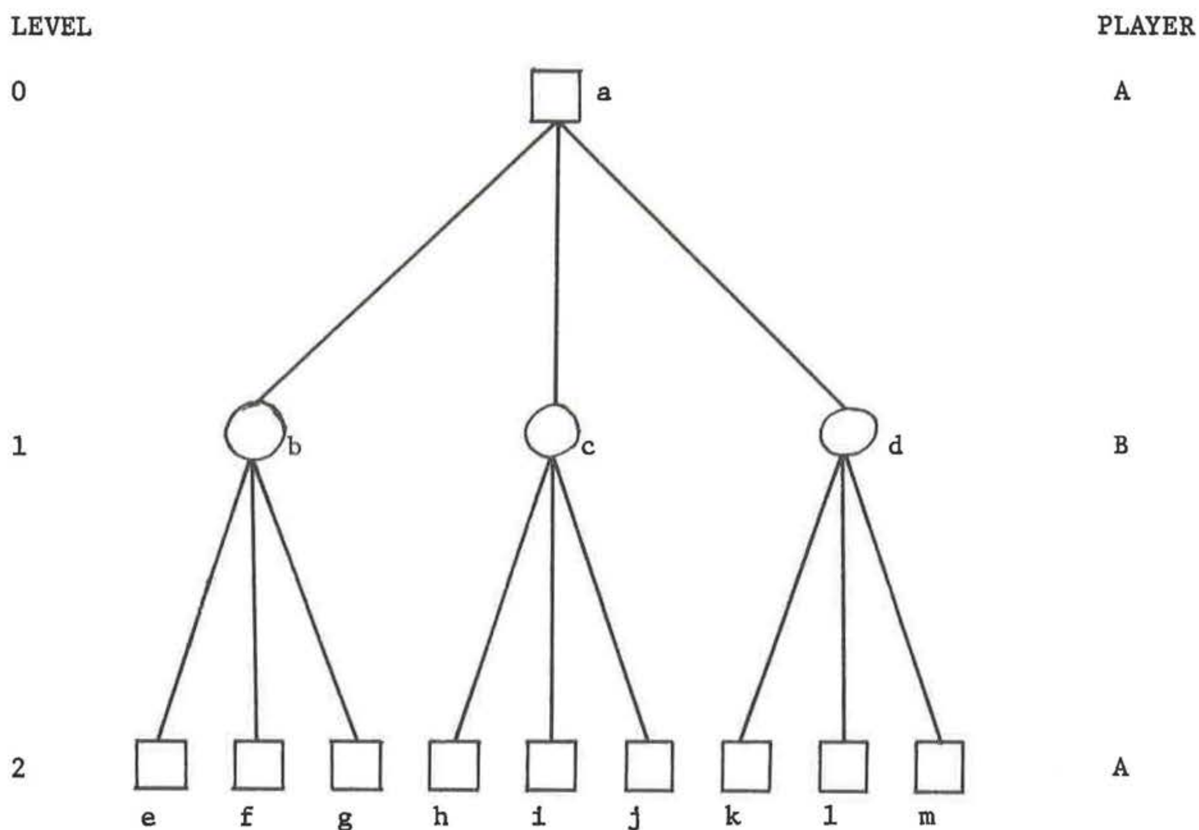

Figure 1 A Simple Example of Mini-Maxing

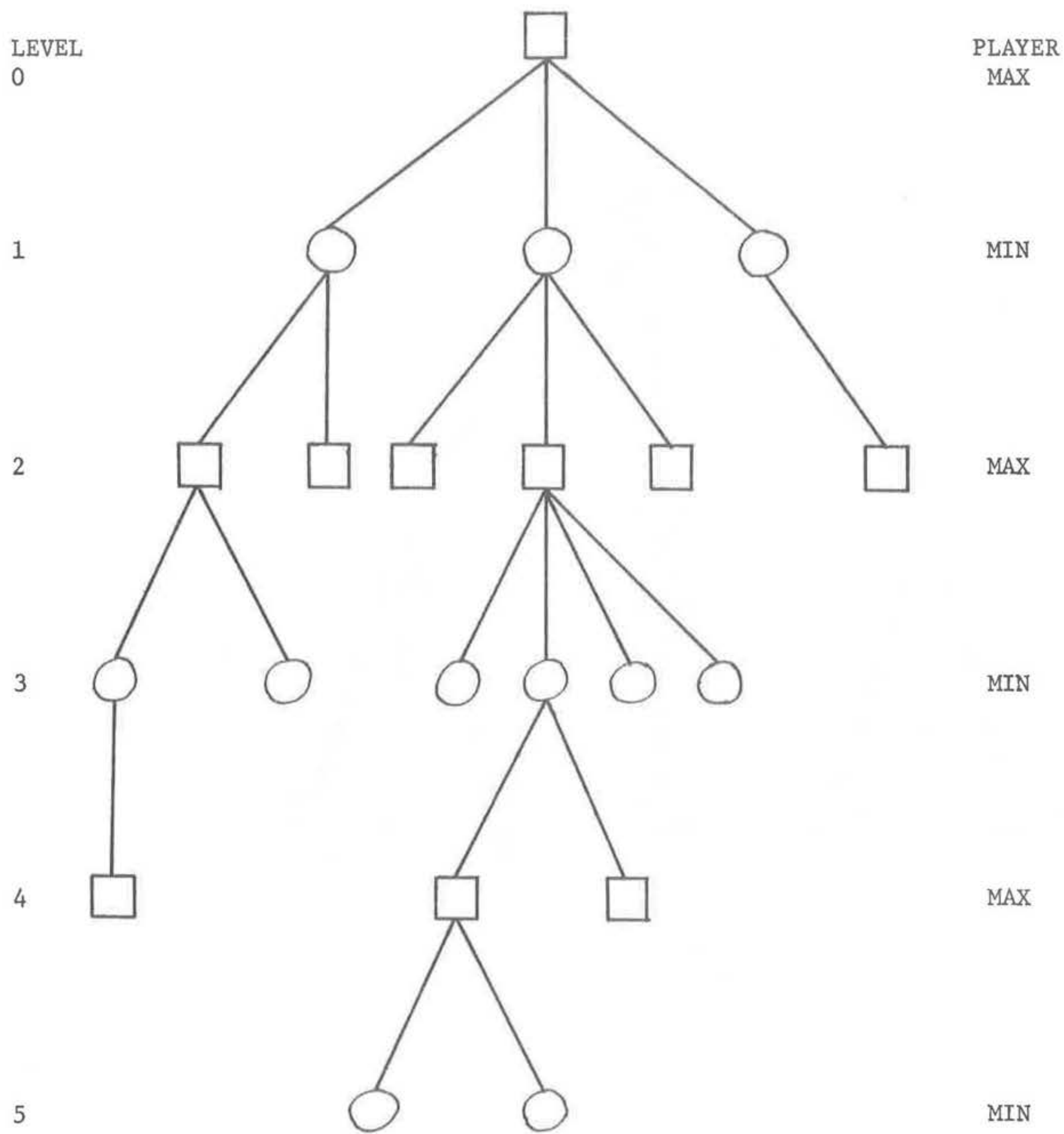

Figure 2 An Example of Mini-Maxing

In either case the terminal nodes are evaluated and the mini-maxing procedure is carried out as follows:

1. For a MAX node:

   If the successor MIN nodes all have values, the MAX node is assigned the maximum value of these.

2. For a MIN node:

   If the successor MAX nodes all have values, the MIN node is assigned the minimum value of these.

There are alternate back-up procedures to mini-maxing, Slagle and Dixon's M and N programming (1970) being one of these.

The basic idea behind mini-maxing is that if it is my turn to move, I will try to optimize my position and for player A who is MAX this means trying to achieve the highest valued board while for player B this means choosing the lowest valued board. All of this is based on the assumptions that the evaluation function is fairly good, and that player B is using a very similar function to make his own moves so that by preventing A from achieving high valued boards B is improving his chances of winning.

At the termination of this process, board a will be assigned a value equal to one of the values of boards e through m. That is, the value of board a represents the highest valued board player A can achieve, two moves hence. Thus A will choose one of boards b, c, and d for his move, depending upon which was assigned the value backed up to board a. If more than one board has this value some, usually arbitrary, decision is made (see Slagle, Dixon, 1970)

We define look-ahead to be the process whereby an explicit or implicit tree or subtree of moves is produced from some initial board. It is not necessary of course that the entire tree be produced before terminal nodes are evaluated. In fact considerable savings in time without loss of information can be achieved by a controlled depth first search called the alpha-

beta procedure (Samuel, 1967; Slagle, Dixon 1969). This achievement depends on the fact that information gained in the early part of the search can be utilized to save searching large parts of the tree later on.

We can now return to the notion of a perfect evaluation function and consider it with respect to look-ahead. In using the perfect evaluation function, the player need not construct any part of the tree except for the immediate successors. (It is even possible for such a function applied to the current board to produce the single required move). Thus the perfect evaluation contains implicitly sufficient look-ahead to guarantee a win and it should be possible to compare the power of heuristic evaluation functions in terms of the degree to which they differ in effective look-ahead. We will examine this notion more thoroughly with respect to one-person games in a later section.

It is the underlying assumption in all game-playing programs which employ a heuristic evaluation function together with look-ahead and back-up that an increase in the depth of look-ahead will result in an increase in the accuracy of the evaluator. Clearly in the limit this is the case for if look-ahead could be carried out to the actual end of the game, the evaluator would thereby be perfect. Slagle and Dixon (1969) have investigated this relation and indeed have found a positive correlation between an improvement in the heuristic function and an increase in the depth of look-ahead. We wish to report some results based on one-person games which also bear on this relation.

2. One-Person Games

2.0 Why One-Person Games?

The term one-person game is a rather cumbersome name for puzzle; that is, a game in which a single player attempts to achieve some desired state from an initial state via a sequence of legal moves. The first question to be discussed is what role does look-ahead have in playing a puzzle? In two person games, one uses look ahead to improve the heuristic function by trying to investigate a broad class of moves and responses prior to choosing a move. However, in puzzles there is no opponent and therefore all moves are made by the player himself.

As a first approach, one might consider look ahead as a means of avoiding cul-de-sacs or at least trying to anticipate the intermediate effects of certain moves. This will obviously add to the computational effort of playing and does not have the justification of avoiding blunders that look ahead provides in two-person games. We will argue in section 2.2 that under certain conditions look-ahead can result in computational savings.

Another question we wish to explore is the use of depth of look-ahead as a means of comparing two heuristic functions. Given two heuristic functions h_1 and h_2 such that h_1 is more effective than h_2 (leaving aside for now any formal notion of effective) we might be interested in the following question:

What is the minimum uniform look-ahead depth, ℓ, such that h_2 together with this look-ahead is as effective as h_1?

This question will be discussed in section 4.

We have said nothing about the associated back-up procedure but for puzzles mini-max is not appropriate. Perhaps the backed up value

should just be the optimum value of the terminal boards (maxi-max). In the following section some current results on tree searching in one-person games will be reviewed.

2.1 Some Results on Tree Searching

2.1.1 The Graph Traverser

This is a program written by Doran and Michie (1966) to find a path between two specified nodes in a graph. We have previously discussed only trees but the generalization to graphs does not alter the problem essentially other than to introduce the difficulty of returning to a node already visited. Doran and Michie define the notion of an economic search as follows:

> The task is to find a path across the graph as economically as possible, i.e. with as little labour as possible expended in the search, avoiding as far as possible false trials, blind alleys and meanderings far from the final path. If the path is short we say that the solution is 'elegant'. If the search is short, we say that the solution is 'economical'.*

We will now describe the procedure very briefly. The program applies to any problem which can be formulated as a graph. That is, the nodes of the graph correspond to states of the problem and an arc from node a to node b indicates that state b can be achieved from state a by the application of one member of the set of legal operators. It is not necessary that the graph be symmetric so that in the above node a may not be reachable from node b by a single operator.

---

* Doran and Michie (1966) p. 237.

The problem statement can be given by the following definitions:

The problem is specified by a graph $G \equiv \{X,\Gamma\}$ where X is the set of nodes (problem states) and where Γ is a function which maps X into itself. For any specific node $x \varepsilon X$, $\Gamma(x)$ is the set of nodes resulting from the application of all legal operations, defined in the particular one-person game, to x. Γ in a sense contains the rules of the game. Thus given a starting node $s \varepsilon X$ and a goal node $g \varepsilon X$, it is required to find a path from s to g, i.e. a sequence of nodes $x_1, x_2, \ldots x_n$ for some n, such that

1) $x_1 = s$ and $x_n = g$
2) for all m such that $1 \leq m \leq n-1$, $x_{m+1} \varepsilon \Gamma(x_m)$.

Economy, then corresponds to a minimal number of applications of Γ in order to find a solution and elegance corresponds to a minimal n.

To use the procedure the following essential information must be supplied:

(1) A procedure develop, specifying Γ defined above.

(2) A procedure evaluate, specifying the heuristic evaluation function E which is a function from X to the non-negative integers. E is intended to be a function which for any node x estimates the minimum number of arcs from x to g.

(3) Starting and goal nodes.

There are other control parameters required and these will be mentioned where necessary.

The procedure begins by applying Γ to s to produce the set of immediate successors $\Gamma(s)$. Each node in this set is evaluated by applying the function E and the node with the smallest value is expanded next (ties being broken arbitrarily). Since the problem space is a graph, among the successors of any node x may be a node which has already been produced. No connection is then made from x to this node. Thus the

process produces a tree which is a subgraph of the problem graph. This tree contains two kinds of nodes, those which are developed and those which are undeveloped.

The process will terminate when the node g occurs among the successors of the currently expanded node. But for non-trivial problems the condition may arise that the growing solution tree exceeds the preset storage limitations of the computer. In the earliest version of the program, the best undeveloped node is selected and the path from s to it is printed. Then the search is continued using this node as a start node. The number of repetitions of this process is limited by a program parameter. There is no guarantee at all that the search will succeed under these conditions.

In later work on the Graph Traverser, Doran (1967), instead of discarding the entire tree up to the current best node, discards only that part of the tree a fixed number of steps along the path to the best node.

Most of the early expermental results are based on a class of sliding block puzzles called the eight and fifteen puzzles. The eight puzzle consists of eight numbered square blocks in a three by three array. A typical arrangment is shown below

1	2	3
4	5	6
7	8	0

Figure 3.

A typical configuration of the eight puzzle

The square labelled 0 is the empty square. Any configuration can be altered by moving into the empty square one of the numbered squares adjacent to it. It should be noted that the set of configurations of this puzzle can be divided into two subsets. For any two elements of the set there exists a sequence of moves which transforms one into the other if and only if they are in the same subset.

This puzzle has been completely solved by Schofield (1967) in an exhaustive fashion and it is known, for example, that the longest minimum solution path is 30 moves. Figure 5 shows the initial and goal configurations for a 30 move problem

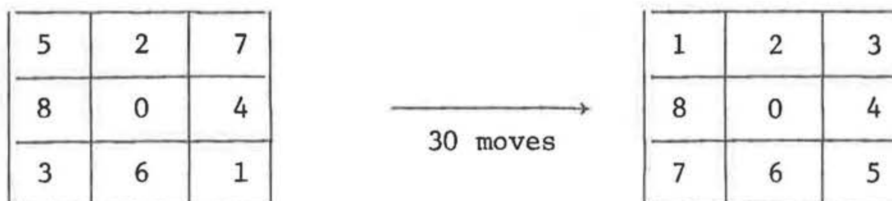

Figure 4 An eight puzzle requiring 30 moves

The successor function Γ for this puzzle is well-defined. Several different evaluation functions E have been used among which are the following:

(1) The value assigned to a configuration is the total number of pieces out of place. Thus the goal will have zero value as it will with the other functions.

(2) Assign a score p_i to the ith piece according to the number of moves it is distant from its goal position, disregarding the barrier posed by intervening pieces (i.e. the city-block distance). Then an evaluation function based on the <u>position count</u> P of a given configuration is given by

$$P = \sum_{i=1}^{8} p_i$$

For example, the left side configuration of Figure 4 has a position count P equal to

$$(4+0+4+0+4+0+4+0) = 16$$

(3) Assign a score s to each piece by checking around the then non-central squares in turn, allotting 2 for every piece not followed by its proper successor and 0 for every other piece. Further, a piece in the centre scores one. An empty non-central square is ignored in applying the succession criterion. (i.e. there is no great difficulty in obtaining the proper sequence if two pieces are separated by a blank square). A <u>sequence count</u> S is given by

$$S = \sum_{i=1}^{8} s_i + c \text{ where } c=1 \text{ if the central square is non-empty and } =0 \text{ otherwise}$$

For the left side of Figure 5, the sequence count S is

$$(2+2+2+2+2+2+2+2) + 0 = 16$$

(4) A weighted combination of (2) and (3) such as

$$E = P + w\,S.$$

Note that a small change in the value of w may not cause a change in the choices determined by the evaluation.

Doran and Michie use the fourth evaluation function which we shall call E_4 in most of their experiments. A battery of tests were run for a fixed value of w=9 over a number of problems with minimum solution paths varying from 8 to 30. Another set of tests was run for w taking values of 0, 1/9, 2/3, 3/2, and 9.

Best results in terms of such performance indicators as the length of the path produced, L and the number of nodes developed, D occur for w in the range 3/2 to 9. Both L and D seem to be correlated, i.e. there is no evidence that by changing w either L or D can be improved at the expense of the other.

We might also mention Doran and Michie's definition of a quantity they call penetrance. If L^* is the minimal path length for a given problem then L^*/L is the path efficiency and L^*/D is the development efficiency (since L^* is the minimum number of nodes to be developed.) Note that $L^*/D = L^*/L \times L/D$

They define the penetrance, L/D, as the fraction of the total number of nodes developed which are incorporated into the actual path found.

For any non-trivial problem L^* is not likely to be known but L and D are determined if the Graph Traverser succeeds in finding any solution. Penetrance in fact represents the degree to which the search tree is elongated and narrow rather than bushy.

The fact that the product of path efficiency and penetrance is equal to development efficiency raises a question about the possible predictive power of the penetrance measure. Some tests indicate that there is some correlation between the percentage of puzzles solved during a second partial search and the value of the penetrance over the preceding first partial search. Recall that a solution may require several partial searches after each of which the path to the current best node is printed and the search begun anew from this node.

As a final point, we should consider the possibility of operator selection at a given node. Instead of fully developing some node it may be possible to select the operator with the best chance of producing a good successor. Thus there will be three categories of nodes - undeveloped, partially developed, and fully developed. For partially developed nodes information must be kept as to which operators have so far been applied.

The mechanism of operator selection opens up many experimental possibilities such as various orderings in the individual operators and the possibility of learning to improve this ordering. Of course learning may also be involved in determining optimum values for w in E_4 or the parameters in other evaluation functions. It is clear that the selection of operators also involves the use of information, usually heuristic, about the problem. Compare this with the work of Newell and Simon on the General Problem Solver (1961) and Ernst and Newell (1969).

More recent results on the Graph Traverser are reported by Michie (1967), Doran (1968), Michie and Ross (1970) and Marsh (1970).

Many other experiments have been run using the fifteen puzzle with more complex evaluation functions, as well as the travelling salesman problem, but we will now turn our attention to some theoretical results on tree searching, which will be used subsequently.

2.1.2 Results of Hart, Nilsson and Raphael (1968)

The title of their paper "A Formal Basis for the Heuristic Determination of Minimum Cost Paths", accurately describes their approach. We wish to briefly describe their results which are among the first to attempt to formalize heuristic tree searching. Much more detail is given in Nilsson's book (1971).

In their notation a graph G is defined by two sets $\{n_i\}$ the set of elements called nodes and $\{e_{ij}\}$ the set of directed line segments called arcs. The cost of traversing or executing arc e_{ij} is c_{ij}. In general the existence of arc e_{ij} does not imply the existence of arc e_{ji}. To prove their results it is necessary that there be a real number $\delta > 0$ such that for all i,j, $c_{ij} \geq \delta$. Graphs satisfying this criterion are called δ-graphs.

A path from n_1 to n_k is defined to be an ordered set of nodes $(n_1, n_2, \ldots n_k)$ with each n_{i+1} a successor of n_i. Every path has a cost which is obtained by adding the individual costs, $c_{i,i+1}$, of each arc in the path. We define $h(n_i, n_j)$ to be the cost of an optimal path from n_i to n_j, i.e. a path having the smallest cost over the set of all paths from n_i to n_j. A few more definitions and we will be able to state their main results. If s is the single start node and G_s is the subgraph from s defined by the problem then we define a nonempty subset T of nodes in G_s as the goal nodes. For any node $n \varepsilon G_s$, an element of $t \varepsilon T$ is a preferred goal node if and only if the cost of an optimal path from n to t does not exceed the cost of any other path from n to any member of T. This cost can be represented by $h(n) = \min_{t \varepsilon T} h(n,t)$.

An algorithm is said to be admissible if it is guaranteed to find an optimal path from s to a preferred goal node of s for any δ-graph. It is clear that admissible algorithms may differ both in the order in which they expand nodes and in the number of nodes expanded (the economy of the procedure). A search algorithm A* is given and shown to be admissible. This algorithm is much like the Graph Traverser but differs in the nature of the evaluation function which is defined in the following way:

For any graph G_s and any goal set T let f(n) be the actual cost of an optimal path constrained to go through n, from s to a preferred goal node of n. Then f(s) = h(s) and in fact f(s) = f(n) for any node n on an optimal path. It is possible to write f(n) as

$$f(n) = g(n) + h(n)$$

where g(n) is the actual cost of an optimal path from s to n and h(n) is the actual cost of an optimal path from n to a preferred goal node of n. Recall that in the Graph Traverser, the evaluation function E is an estimate of the minimal number of arcs from any node x to to the goal node g.

Estimates, $\hat{g}(n)$ and $\hat{h}(n)$, are proposed for $g(n)$ and $h(n)$, respectively, where $\hat{g}(n)$ is taken to be the cost of the path from s to n having the smallest cost so far found by the algotithm, a quantity which is easy to calculate. The condition for the algorithm A* to be admissible is that the estimate $\hat{h}(n)$ be such that for all n, $\hat{h}(n) \leq h(n)$, i.e. $\hat{h}(n)$ must be any lower bound of h. A natural question which now arises is how to compare admissible algorithms. One obvious criterion equivalent to Doran and Michie's (1966) concept of economy is the number of nodes developed in finding the optimal solution. Two additional concepts are defined before a result about the optimality of algorithms can be proved. An admissible algorithm is said to be optimal if no other admissible algorithm expands fewer nodes.

Optimality depends first on the heuristic function $\hat{h}$ satisfying what is called, the consistency assumption, a kind of triangle inequality rule which states that for any two nodes m and n

$$\hat{h}(m) - \hat{h}(n) \leq h(m,n)$$

i.e. the difference between the estimated costs to the goal from any pair of nodes m and n must be a lower bound or the actual cost of an optimal path from m to n. This assumption implies a uniformity of application of the heuristic function over the problem space.

The second requirement depends on the following definition Nilsson (1971):

> We shall say the algorithm A is <u>more informed</u> than algorithm B if the heuristic information used by A permits computing a lower bound on h(n) that is everywhere strictly larger (for all nongoal nodes n) than that permitted by the heuristic information used by B.

Finally, the following <u>Theorem</u> is proven

<u>Theorem</u>: let A and A* be admissible algorithms such that A* is more informed than A, and let the consistency assumption be satisfied by the $\hat{h}$ used in A*. Then for any graph if node n was expanded by A*, it was also expanded by A.

Proof: See Nilsson (1971) pp 61-65.

In other words, one procedure will be at least as economical as another if its heuristic function is a strictly better lower bound of the perfect function. And a heuristic function which satisfies this condition with respect to another is said to be more informed.

We will refer to this concept of being more informed when we discuss the relation between look-ahead and heuristic evaluation functions.

2.1.3 Pohl's Results (1969,1970a, 1970b)

Pohl describes an algorithm which he calls the Heuristic Path Algorithm (HPA) and which is similar both to the Graph Traverser (1966) and the Hart, Nilsson, and Raphael (1968) algorithm. Perhaps, it is worthwhile to present HPA in some detail since this is the algorithm that is used in the experiments to be reported in section 3.

The problem will be defined by a start node s, a terminal node t and a successor function Γ, such that for any node x, $\Gamma(x)$ yields the set of successors of x. Let $g(x)$ be the number of edges from s to x as found by the search and let $h(x)$, the heuristic function, be an estimate of the number of edges from x to t. Then we define an evaluation function $f(x) = (1-w_t)g(x)+w_t h(x)$ where $0 \leq w_t < 1$. Finally, let S be the set of nodes already visited, sometimes called the expanded or developed nodes and let $\overline{S}$ be the set of nodes directly reachable from S in one edge but not already in S.

The algorithm HPA is given as follows, Pohl(1970a):

1. Place s in S and calculate $\Gamma(s)$ placing them in $\overline{S}$.

   if $x \varepsilon \Gamma(s)$ then $g(x)=1$ and $f(x) = (1-w_t)+ w_t\ h(x)$

2. Select $n \varepsilon \overline{S}$ such that $f(n)$ is a minimum.

3. Place n in S and $\Gamma(n)$ in $\overline{S}$(if not already in $\overline{S}$) and calculate f for the successors of n.

   If $x \varepsilon \Gamma(n)$ and $x \notin S$ then $g(x)=1+g(n)$

and $f(x) = (1-w_t)\ g(x) + w_t\ h(x)$

4. If n is equal to t then halt, otherwise go to step 2.

In this formulation of the evaluation function, a value of $w_t=0$ corresponds to an exhaustive breadth first search, a value of $w_t=1$ corresponds to a pure heuristic search (like that performed by the Graph Traverser), and $w_t=\frac{1}{2}$ is very much like the type of search produced by the Hart, Nilsson, Raphael procedure (1968) where both components of the evaluation function are equally weighted. We wish to state some of the theorems proved by Pohl (1970a) both to indicate the nature of the results obtained and because some of these results will be needed in the next section.

Theorem 1

If h is perfect then for $1 \geq w_t \geq 1/2$, the search by HPA is optimal, i.e. includes the fewest nodes in set S in finding a solution path.

To discuss error in heuristic functions and its effect on search, Pohl (1970a) gives the following definitions:

Let h^* = perfect estimator
ε = a bound on the error 0,1,2,...
h = given heuristic function,
$(h^*-\varepsilon) \leq h \leq (h^*+\varepsilon)$ in the problem domain.

The investigation to follow will be similar to a worst case analysis in numerical analysis. It is a particular failing of the theory of heuristic tree searching that there are no results based on a statistical or probabalistic formulation.

An h is to be constructed satisfying the constraints given above but in such a manner as to mislead HPA to the greatest extent. In case of ties, HPA will always choose the worst nodes, i.e. those off the

solution path. For example to make h as bad as possible

$$h = h^{*}+\varepsilon \text{ for each node on the solution path}$$

and

$$h = h^{*}-\varepsilon \text{ for each node off the solution path.}$$

The following theorems are established, Pohl (1970a):

Theorem 7

Let k be the distance from the root node to the goal node and $f=\frac{1}{2}(g+h)$ be the function used by HPA, then the maximum number of nodes expanded in a binary tree space is

$$2^{\varepsilon} \cdot k + 1$$

where ε is the error bound on h.

Theorem 8

Let k be the distance from the root node to the goal node and f=h be the function used by HPA, then the maximum number of nodes expanded in a binary tree search space is

$$k+1 \quad \varepsilon=0$$

$$\frac{4^{\varepsilon}}{2} \cdot k+1 \quad \varepsilon\geq 1$$

Theorem 9

If HPA is searching any tree structured space for some goal node then

(a) f=h will visit at least as many nodes as $f=\frac{1}{2}(g+h)$ in the sense of the above worst case analysis.

(b) If h_1 is bounded in error by ε_1 and h_2 by ε_2 with $\varepsilon_2>\varepsilon_1$, then in the sense of the worst case analysis h_2 can only search more nodes than h_1 for the same w_t value.

The important point to note is that the inclusion of g in an evaluation function cannot harm and may improve the function. Intuitively this is because g lends a breadth first component to the search and if the

heuristic component tends to cause the search to go off on some tangent, g will exert a conservative, stabilizing force. There are of course two limitations to the above remarks. First most problem domains are graphs and not trees and second, the results are based on a worst case analysis.

2.2 The Role of Look-Ahead

From the discussion of section 2.1, mainly the experimental work of Doran and Michie (1966), attempts to improve the economy of the heuristic search depend on changes in one or more parameters in the evaluation function. For example most of the experimental work on the eight puzzle involves variation in the coefficient w in the evaluation function P+wS (see section 2.1.1). This presupposes a heuristic function potentially good enough to produce acceptable results and requiring only some careful tuning. Another way of improving an evaluation function may be through the use of look ahead.

Given an evaluation function look-ahead can be used in the following way. We refer to Pohl's algorithm HPA described in 2.1.3. If f is an arbitrary evaluation function we define f_ℓ to be a new evaluation function to be applied to an arbitrary node x by producing the subtree from node x to a depth ℓ, evaluating the terminal nodes of this tree using f and assigning to node x the minimum value. Thus whenever a node is developed or expanded the evaluation function f_ℓ, which should be thought of as a procedure, is applied to every successor node. In step 2 of the algorithm the node $n \varepsilon \overline{S}$ is selected such that $f_\ell(n)$ is a minimum. Two questions arise:

(1) Is $f_\ell(x)$ a better estimate of the "goodness" of x than f?

(2) Even assuming that the answer to (1) is in the affirmative doesn't the extra computation involved in evaluating $f_\ell(x)$ outweigh any advantage it may have?

In partial answer to these we note that if look ahead does not improve the evaluation function, it has been a waste of effort in most game playing programs. We will return to this point in section 3, but first we wish to examine question (2) more carefully. In most of the work reviewed, no attention has been paid to the cost of computing the evaluation function. When one function is said to be better than another, it is implicitly assumed that while it develops fewer nodes, its computational costs are no greater than the other function. It is to this point that we now direct our attention. In the spirit of Pohl's (1970a) work the investigation will be on the basis of a worst case analysis.

Theorem 2.2.1

Let (1) h_1 be a heuristic function with bounded error ε_1 an integer,

(2) h_2 be a heuristic function with bounded error ε_2 (an integer) such that $\varepsilon_2 > \varepsilon_1$

(3) k be the distance from the root node to the goal

(4) the problem domain be represented by an m-ary tree;

then the algorithm HPA using $f_2(x) = \frac{1}{2}\{g(x) + h_2(x)\}$ will in the worst case develop c_1 k more nodes than it would using $f_1(x) = \frac{1}{2}\{g(x) + h_1(x)\}$, where $c_1 = m^{\epsilon_2} - m^{\epsilon_1}$.

In an obvious generalization of Pohl's (1970a) Theorem 7 from binary trees to m-ary trees, HPA using f_1 will expand $m^{\varepsilon_1 k+1}$ nodes, whereas using f_2 it will expand $m^{\varepsilon_2 k+1}$ nodes. Therefore f_1 will expand $m^{\varepsilon_2 k}-m^{\varepsilon_1 k}$ fewer nodes than f_2. This can be rewritten as

$$
\begin{aligned}
& m^{\varepsilon_2 k}-m^{\varepsilon_1 k} \\
&= (m^{\varepsilon_2}-m^{\varepsilon_1})^k \\
&= c_1^k
\end{aligned}
$$

This result is really an expansion of part (b) of Theorem 9, Pohl (1970a). An explicit quantity can be derived because the tree structured space is here assumed to be regular (i.e. m-ary). The following result will show under what conditions it may be adv isable to use a better but computationally more expensive heuristic function, again based on a worst case analysis.

At this point it should be noted that implicit in HPA is the requirement that every newly generated node must be compared with the nodes previously assigned to the sets S and $\overline{S}$ in order to weed out redundant nodes. This is an obvious requirement for graph structured problem domains where the same node can be reached by different routes. But it can also arise in tree-structured domains which have this form because identical problems states are not identified. Now Hart, Nilsson, Raphael (1968) prove a lemma which shows that for admissible algorithms if h satisfies the consistency assumption, a node which has already been developed will never be generated again, i.e. if the algorithm expands a node then an optimal path to that node has already been found.

In the theorem to follow since h may exceed the perfect estimator, the algorithm HPA using it is not admissible. Thus the necessity for comparing nodes will still exist and it is upon this necessity that the result depends.

Theorem 2.2.2

Given the conditions (1) - (4) of Theorem 2.2.1 and the fact that for all x the cost of computing $h_1(x)$ is greater than the cost of computing $h_2(x)$; then there exists a problem represented by an m-ary tree with a sufficiently long solution path of length k such that algorithm HPA using $f_1(x)=\frac{1}{2}\{g(x) + h_1(x)\}$ will require less computation time than if it were to use $f_2(x)=\frac{1}{2}\{g(x) + h_2(x)\}$.

The proof depends on showing that the extra cost of computation using h_1 increases linearly with k while the extra cost of computation of h_2 due to an increased number of nodes depends on the square of k. From Theorem 2.2.1, the number of extra nodes h_2 develops is $c_1 \cdot k$ where $c_1 = m^{\varepsilon_2} - m^{\varepsilon_1}$. For the number of nodes h_1 develops the extra computation is $c_2 \cdot k \cdot m^{\varepsilon_1}$, i.e. the extra cost of h_1 increases as $O(k)$. All other computations for both heuristics are the same.

Because h_2 must deal with an additional $c_1 \cdot k$ nodes (in the worst case of course) there will be the associated cost of comparisons to avoid redundant nodes. We can compute this cost in the following way.

Let $n_1 = k \cdot m^{\varepsilon_1}$ and

$n_2 = c_1 \cdot k$.

Then for a simple comparison search the number of comparisons to be made is

$$\sum_{i=n_1}^{n_1+n_2-1} i = \sum_{i=1}^{n_1+n_2-1} i - \sum_{i=1}^{n_1-1} i$$

$$= \frac{(n_1+n_2-1)(n_1+n_2)}{2} - \frac{(n_1-1)n_1}{2}$$

$$= \frac{n_2^2}{2} + \frac{2n_1n_2}{2} - \frac{n_2}{2}$$

Since $n_2 = c_1 k$, the number of comparisons for a simple search is $O(k^2)$. Therefore the computational cost of the additional comparisons h_2 must make increases as $O(k^2)$. Thus the conclusion of the theorem.

This is a weak theorem and its general applications may not be very widespread but it does suggest that there may be some problems where look-ahead may improve the heuristic function. Further it is clear that there are more efficient comparison schemes such as binary search and hash coding and the $O(k^2)$ number of comparisons is too high. Binary search will lower this to $O(k \log k)$ for which the theorem still holds although a larger k will be necessary. Using hash coding will further decrease this with a small linear increase in costs due to the hash function computation. It will therefore be necessary to carry out some experimentation in order to see the applicability of this result.

3. Experimental Results

3.1 Program

The program is written in ALGOLW and is essentially the same as the program appearing in Pohl's thesis (1969). The latter is actually an implementation of Pohl's bi-directional search, while the former an implementation of HPA (see section 2.1.3) is uni-directional with some additional features. Obviously the most important feature is a look-ahead procedure which has been incorporated into the evaluation function. Under the control of an input look-ahead parameter, LK, the value of a node n is determined by producing the complete tree from n to a depth LK, evaluating the terminal nodes, and assigning the lowest value to n. Thus for zero look-ahead, LK will be zero and no tree will be produced.

To be more precise, we should note that the evaluation of the terminal nodes is in fact their heuristic evaluation, i.e. the h(x) part

of $f(x) = (1-w_t)g(x) + w_t h(x)$. To avoid overshooting the goal each node produced in the look-ahead tree is compared to the goal node and the search is terminated if the comparison is successful. This adds to the computational effort involved in carrying out look ahead.

With reference to the final remarks in the previous section, hash coding is used to compare nodes in order to weed out redundant ones. This immediately suggests that the results we are looking for may not be easy to find. In fact the point of these experiments is really to exhibit a problem for which the existence statement of Theorem 2.2.2 holds. In no way, do we intend to present an exhaustive set of experiments over a wide range of problems. We hope the results presented will be suggestive for again it should be noted that the above theorem is proved under very restrictive conditions.

The domain of experimentation is the eight-puzzle discussed earlier and specific problems are taken from Doran and Michie (1966) and Schofield (1967). We are unable to compare our results directly with Michie and Doran's (1966) for two reasons. First they do not describe the individual problems for which they have obtained results and second their evaluation function is purely heuristic, i.e. there is no g term. Finally, to measure the computational effort, we take the amount of central processor time used.

3.2 Results

All the results are derived from five specific problems. The goal configuration is the same for all of these, namely.

```
1 2 3
8 0 4
7 6 5
```

and the initial configurations are shown in Table I together with the lengths of optimal solution paths.

Problem Number	Initial Configuration	Optimal Path Length
1	216 408 753	18
2	825 106 734	20
3	824 107 563	22
4	824 107 356	24
5	765 108 324	30

TABLE I The list of problems

The evaluation function used is

$$f(x) = (1-w_t)g(x) + w_t\, h(x)$$

where $h(x) = P+w\,S$, the E_4 defined in section 2.1.1.

The basic parameters in each run are

the problem

the maximum number of iterations - MAXITER

w_t - almost always equal to .5

w - the parameter in the heuristic function

LK - the look ahead depth

First some very general observations. On the basis of Theorem 2.2.2, we would expect that problems with fairly long solution paths may be necessary in order to achieve a large number of results for which look-ahead reduces the overall computation. The experiments reported here range over problems with solution paths varying between 18 and 30 and these are not long enough to ensure a high proportion of successful runs. Thus, in almost all cases, for a fixed value of w, increasing the look-ahead depth LK results in a decrease in the number of nodes developed but an increase in the computation time. However, a complete set of 25 runs for problem 3 with w taking the values .1, .5, 1.5, 3.0 and 9.0 and LK taking values 0,1,2,3, and 4 produced two results which satisfy Theorem 2.2.2. Consider Table II, which for each value of w and LK, gives the number of nodes developed (D), the path length (L) and the computation time in seconds (T).

This problem has an optimum path length of 22. For w = 3.0 and LK = 0, after 200 iterations which took 5.5 seconds, a solution was not found. However when LK was increased to 4, after 27 iterations an optimal solution was found in 4.2 seconds. A similar situation occurred for w = 1.5. In this instance, the process without look-ahead discovered an optimal solution after 142 iterations requiring 3.7 seconds while with look-ahead to depth 4 an optimal solution was found after only 23 iterations in 3.5 seconds.

LK	w														
	.1			.5			1.5			3.0			9.0		
	D	L	T	D	L	T	D	L	T	D	L	T	D	L	T
0	200	-	5.6	200	-	5.7	142	22	3.7	200	-	5.5	65	26	1.5
1	200	-	9.9	200	-	9.9	92	22	4.3	63	28	3.0	63	28	3.0
2	200	-	13.3	200	-	13.2	58	26	3.8	59	26	3.8	32	26	2.1
3	200	-	19.5	184	22	17.9	59	22	5.6	32	28	3.1	26	22	2.6
4	200	-	30.2	90	24	13.5	23	22	3.5	27	22	4.2	30	26	4.6

TABLE II Results obtained for problem 3

D - number of developed nodes

L - path length (- means no solution was obtained)

T - computation time in seconds

LK - look ahead depth

w - parameter in the heuristic function

The evaluation function f with zero look-ahead is $f(x) = 1/2[g(x) + P + wS]$ and with depth four we have f_4 where w = 1.5 or 3. Clearly f_4 requires considerably more computation than f, but when applied to problem 3 it actually results in a saving in computation time.

The five problems were picked randomly and for only one problem and only two pairs of values for w and LK were successful results obtained. And as can be seen from Table II, it was only when the use of f_4 resulted in a very rapid solution that sucess was achieved. In one instance, only 23 developed nodes were produced which is almost the minimum number possible. As a general observation one could say that the cases where look-ahead might prove helpful are very narrowly circumscribed. Where the heuristic function is itself not very effective, limited look-ahead will probably not help; similarly, for a very good heuristic function, it is not likely that limited look-ahead will improve matters. Somewhere in between is an area which may yield worthwhile results.

3.3 Anomalies and Interesting Points

In the course of the experiments there arose a few results which deserve brief comment. Problem 1 was chosen from Michie and Doran (1966) because the complete solution tree is given. (It is also given as an example by Nilsson, 1971) By setting w_t to 1, the program acts as the Graph Traverser, relying only upon the heuristic function. Two complete sets of runs were made for this problem, one with w_t set to 1 and the other with w_t set to .5. Table III shows some of these results.

First a note of explanation. Wherever $D<L$ for a set of parameter values, this implies that the goal node was found during the look-ahead process. The first point to note is that for w_t = 1.0 and w = 1.5 and 3.0,

LK	w_t	w					
		.5		1.5		3.0	
		D	L	D	L	D	L
0	1.0	100	-	25	18	24	18
	0.5	39	18	33	18	23	18
1	1.0	53	38	65	46	65	46
	0.5	27	18	21	18	21	18
2	1.0	39	27	33	27	17	18
	0.5	22	18	19	18	17	18
3	1.0	39	31	22	24	22	24
	0.5	18	18	22	24	22	24
4	1.0	15	18	21	24	21	24
	0.5	15	18	21	24	21	24

TABLE III Some results obtained for problem 1

D - number of nodes developed

L - path length

w_t = 0.5 - inclusion of g(x) in the evaluation function
= 1.0 - exclusion of g(x); program behaves like the Graph Traverser

LK - look ahead depth

w - parameter in the heuristic function

the program performs very badly when LK = 1 compared to LK = 0. That is, a look-ahead of depth 1 causes a serious deterioration in performance. This situation results from the parity conditions involved in the length of any solution path to this problem. It can be shown (Schofield, 1967) that any solution to problem 1 must be of even length. Look-ahead to an odd depth, sets the solution off on a poor direction by badly estimating the number of steps to the goal node, resulting in the worst case in a solution path 2½ times as long as the optimal one. This suggests that the _a priori_ notion that look-ahead applied to an arbitrary evaluation function in an arbitrary problem domain will inevitably improve the situation is certainly not well taken.

Now the effect of g(x) can be noted. Referring to the same set of parameter values mentioned above, we note that a solution characterized by 65 developed nodes and path length of 46 has been substantially reduced to 21 developed nodes and an optimal path length of 18. For this reason w_t = .5 was used for the remainder of the experiments. This striking result is suggested by Pohl's _Theorem 9_, stated in section 2.1.3 which, however, was proved only by a worst case analysis. At least for the conditions of this problem, the addition of g(x) results in a major improvement in some cases, no improvement in a few others and only a slight worsening in one case.

Finally one minor point remains. In most versions of tree searching algorithms, if the search for the next node to expand on the basis of the evaluation function value results in a tie, this tie is broken arbitrarily. However, it is possible that different implementations of the same algorithm may produce different solutions depending on the actual procedure used to break ties. Our program begins its search for the node with minimal value from the most recently generated node back to the first

node generated. The first occurrence of the minimal value, if there are more than one, is automatically selected as the tie breaker. This is certainly an arbitrary method for tie-breaking. Recall the notion of admissibility defined by Hart, Nilsson, Raphael (1968). If the conditions for admissibility are satisfied an optimal solution will be produced; however, if they are not then solutions may depend on the tie-breaking procedure. Consider Table IV, where the method described above for tie-breaking is called TIE-1 and where TIE-2 is that method modified by beginning the search at the earliest node generated and proceeding to the most recent. At least for the cases LK = 0 and 1 there is a significant difference in the number of nodes developed. Although the path lengths are the same in this case, other results showed differences in this factor as well.

Because most algorithms are not likely to be admissible, tie-breaking procedures should be carefully described so that results can be correctly evaluated.

4. Conclusions

4.1 Look-Ahead and "Informedness"

Recall the definition of 'more informed' by Hart, Nilsson and Raphael (1968) quoted in section 2.1.2. This describes a means for comparing the effectiveness of two heuristic functions but not a very practical one. The usual notion of effectiveness, as we have seen, is expressed in terms of the number of nodes expanded in finding a solution. It is further assumed that the heuristic functions compared are equivalent in terms of their computation times. In some cases it may be better to measure the effectiveness of two heuristic procedures in terms of their overall computational efforts. This is, in effect, the import of Theorem 2.2.2.

In terms of our discussion on look-ahead we might interpret the notion

LK	w = 3.0			
	TIE - 1		TIE - 2	
	D	L	D	L
0	40	32	71	32
1	36	32	54	32
2	46	32	48	32
3	39	34	46	32
4	44	32	43	32

TABLE IV Some results obtained for problem 5

D - number of nodes developed

L - path length

LK - look ahead depth

w - parameter in the heuristic function

TIE-1, TIE-2 - see text

of 'more informed' in the following way. Given heuristic functions h_1 and h_2 which are equivalent in computation times, which generally implies that they differ only in parameter values, such that h_1 develops fewer nodes than h_2 we could say in a loose manner that h_1 is more informed than h_2. The formal definition is considerably more restrictive. How much more informed, might be determined by a series of experiments aimed at discovering a minimum uniform look-ahead depth ℓ such that h_2 becomes as effective as h_1. That is, h_1 'contains' an implicit look-ahead of depth ℓ and thereby saves a certain amount of computational effort compared to h_2.

By relating effectiveness to look-ahead, we make these ideas intuitively clearer than by speaking of heuristic functions as being better lower bounds of perfect estimators. For example, with zero look-ahead a value of $w = 1.5$ results in a more economical solution for problems than does a value of $w = .5$. Similarly, for problems 2 and 3 the value of $w = 1.5$ produces a solution with fewer than 200 developed nodes where none is produced for $w = .5$. See Table V for the actual numbers. Now if the look-ahead depth is increased to 3 while keeping $w = .5$, results are achieved which are comparable to those for $w = 1.5$ and zero look-ahead. Of course computation times are also increased by a factor varying between 2.5 and 5.

We should note that increasing the look-ahead depth to 1 improves performance for problem 1 but not for problems 2 and 3. Thus as a preliminary hypothesis we might say that an evaluation function f_1 with $w = .5$ and a look-ahead of 3 is as effective as f_2 with $w = 1.5$ and zero look-ahead. It is of course considerably more time consuming and this is why we can say further that f_2 is more informed than f_1. That is, it contains information that f_1 will have to expend more computation effort to acquire.

LK	Problem 1				Problem 2				Problem 3			
	w				w				w			
	.5		1.5		.5		1.5		.5		1.5	
	D	L	D	L	D	L	D	L	D	L	D	L
0	39	18	33	18	200	-	172	20	200	-	142	22
1	27	18			189	20			200	-		
2	22	18			153	20			200	22		
3	18	18			138	20			184	22		

TABLE V Some results obtained from problems 1, 2, and 3

D - number of nodes developed

L - path length

LK - look ahead depth

w - parameter in the heuristic function

PROBLEMS 1, 2, 3 - see TABLE I

4.2. Further Discussion

We have reported a number of experiments based on the 8-puzzle which begin to indicate the usefulness of look-ahead. These experiments will be extended to the fifteen-puzzle where solution paths are much larger. Under these circumstances we can expect look-ahead to result in significant savings in computation time.

As a first step in exploring the usefulness of look-ahead, we have kept the look-ahead depth a constant for the entire run. It may be the case that the depth of look-ahead should vary depending on the expected length of the remainder of the solution path. That is, initially we may be willing to invest more effort into look-ahead but as we approach the goal a much smaller look-ahead depth may be sufficient. This suggests the possibility of a variable depth correlated somehow with the heuristic function's estimate.

Another avenue for investigation is the possibility of saving the look-ahead tree for some of the more promising nodes. This will result in the saving of computation required for re-growing some of these trees. There will be extra bookkeeping involved in order to erase subtrees no longer thought to be useful and to keep track of those still in existence.

4.3 Final Remarks

We wish to mention the work in look-ahead done in a more restricted context by David S. Johnson (1968) in his M.S. thesis. His work was carried out for a very special class of games called "Tree Solitaire" and various look-ahead strategies were explored. A more practical application of look-ahead is described by A.L. Cherniavsky (1972). He uses a look-ahead method to resolve conflicts in a timetable compilation problem for a single-track railway.

A discussion of the most recent work on the Graph Traverser and its relation to the problem of the formation of plans is given by Michie (1971).

Acknowledgements

The financial support of the National Research Council of Canada through their grant 67-5552 is gratefully acknowledged. I would especially like to recognize the important contribution of James Kestner. He originally proved Theorems 2.2.1 and 2.2.2 in a graduate course I conducted in Artificial Intelligence.

Bibliography

Cherniavsky, A.L., (1972), "A Program for Timetable Compilation by a Look-Ahead Method, "Artificial Intelligence 3,1 pp. 61-76.

Doran, J.E. (1967), "An Approach to Automatic Problem Solving," in N.L. Collins, and D. Michie (eds.), Machine Intelligence 1, American Elsevier Publishing Company, Inc. pp. 105-123.

———(1968), "New Developments of the Graph Traverser", in E. Dale, and D. Michie (eds.), Machine Intelligence 2, American Elsevier Publishing Company, Inc. pp 119-135.

——— and D. Michie (1966), "Experiments with the Graph Traverser Program," Proc. R. Soc. A,294 pp 235-59.

Ernst, G.W. and A. Newell (1969), GPS: A Case Study in Generality and Problem Solving, ACM Monograph, Academic Press, Inc.

Greenblatt, R. et al (1967), "The Greenblatt Chess Program," Proc. AFIPS Fall Joint Computer Conference pp 801-810

Hart, P.E., N.J. Nilsson, and B. Raphael (1968), "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," IEEE Trans. of Syst. Sci and Cyb. 4,2 pp 100-107.

Johnson, D.S. (1968), "Look-Ahead Strategies in One Person Games with Randomly Generated Game Trees," M.S. thesis, M.I.T. and Artificial Intelligence Memo No. 205, Massachusetts Institute of Technology Project MAC, July 1970.

Marsh, D. (1970), "Memo Functions, the Graph Traverser, and a Simple Control Situation," in B. Meltzer, and D. Michie (eds.), Machine Intelligence 5, American Elsevier Publishing Company, Inc. pp 281-300.

Michie, D. (1967), "Strategy-Building with the Graph Traverser," in N.L. Collins, and D. Michie (eds.), Machine Intelligence 1, American Elsevier Publishing Company, Inc., pp 135-152.

——— (1971), "Formation and Execution of Plans by Machine," in N.V. Findler and Bernard Meltzer (eds.), Artificial Intelligence and Heuristic Programming, American Elsevier Publishing Co., N.Y.

——— and R. Ross (1970), "Experiments with the Adaptive Graph Traverser", in B. Meltzer, and D. Michie (eds.), Machine Intelligence 5, American Elsevier Publishing Company, Inc., pp 301-318.

Newell, A., J.C. Shaw, H. Simon (1958),"Chess Playing Programs and the Problems of Complexity," IBM J. Res. Develop. 2 pp 320-335. Reprinted in E. Feigenbaum and J. Feldman (eds.), Computers and Thought, McGraw-Hill Book Co., 1963.

Newell, A., and H. Simon (1961), "GPS, A Program that Simulates Human Thought," in Lernende Automaten, Munich: R. Oldenbourg K.G. Reprinted in E. Feigenbaum and J. Feldman (eds.), Computers and Thought, McGraw-Hill Book Co., 1963.

Nilsson, N.J. (1971), Problem-Solving Methods in Artificial Intelligence, McGraw-Hill Co.

Pohl, I. (1969) "Bi-Directional and Heuristic Search in Path Problems", Ph.D. Thesis, Stanford University and Report 104, Stanford Linear Accelerator. Stanford University, Stanford, California.

__________(1970a) "First Results on the Effect of Error in Heuristic Search," in B. Meltzer, and D. Michie (eds.), Machine Intelligence 5, American Elsevier Publishing Company, Inc. pp219-236.

__________ (1970b) "Heuristic Search Viewed as Path Finding in a Graph" Artificial Intelligence 1,3 pp 193-204.

Samuel, A. (1959), "Some Studies in Machine Learning Using the Game of Checkers", IBM J. Res. Develop 3,3 pp 210-229, Reprinted in E. Feigenbaum, and J. Feldman (eds.), Computers and Thought, McGraw-Hill Book Co., 1963.

__________ (1967), "Some Studies in Machine Learning Using the Game of Checkers, II, Recent Progress," IBM J. Res. Develop. 11,6

Schofield, P.D.A. (1967) "Complete Solution of the 'Eight-Puzzle'," in N.L. Collins, and D. Michie (eds.), Machine Intelligence 1, American Elsevier Publishing Company, Inc. pp 125-133.

Slagle, J., and J. Dixon (1969) "Experiments with Some Programs That Search Game Trees", J.ACM 16,2 pp 189-207.

__________, __________ (1970), Experiments with the M & N Tree Searching Program," Comm. ACM 13,3 pp. 147-154.