On looking ahead in chess and other complex decision problems

John Geanakoplos Larry Gray

November 1991

Abstract

When we are faced with a decision, the ultimate consequences of our choices are often beyond our 'look-ahead horizon', even if we are aided by a fast computer, and we can usually only make educated guesses about the potential value of those intermediate consequences that are not beyond that horizon. One example of such a situation is the problem of choosing good moves in a game of chess. We develop a mathematical model that realistically describes chess and other decision problems, and use this model to investigate two questions: how to best use the information obtained by a given look-ahead procedure, and how to organize the look-ahead procedure so as to obtain the most useful information. We show that the usual algorithm used by chess-playing computers (the Shannon algorithm) is flawed because it does not take into account its own bounded rationality. We derive several meta-principles for decision making, and show that they explain some recent successful deviations from the Shannon algorithm. Several suggestions for further improvements are discussed.

1 Introduction

Deep Thought, a chess playing machine that can calculate more moves ahead than all other computer programs, recently took first place in the World Computer Chess Championships, has defeated a number of human grandmasters, and even challenged, albeit unsuccessfully, World Champion Gary Kasparov. Deep Thought's designers are working on an even more powerful micro chip with which Deep Thought will be able to calculate an additional 2 full moves ahead, giving it enough strength (so they claim) to beat Kasparov with ease, in spite of the fact that the machine's knowledge of the principles of chess is relatively primitive. Can we agree with Gary Kasparov, who still believes that sheer brute calculation cannot match human ingenuity?

Because chess is a complex setting in which we cannot accurately quantify our ignorance, and in which the calculation of all possible consequences of an

action is beyond the capabilities of any computer, even beyond those of any computer imaginable, we can use the game as a metaphor for studying decision making, learning, and expertise. A little thought about how computers play chess leads to the following puzzle: If computers could perfectly assess the value of a position in chess, they would not need to look ahead more than one more. On the other hand, if they are incapable of evaluating a position at all, it would do them no good to look ahead more than one move. Why is it then, that when their ability to evaluate positions lies somewhere between the two extremes, it seems to help them to look ahead as many moves as possible? One possibility is that positions become appreciably easier to evaluate as one goes deeper into the game. But can this be the whole story? After all, the typical foresight horizon of the strongest chess computer is only 4 or 5 moves, and who would argue that positions are on the average much simpler at move 15 in a game than they are at move 11? Of course some positions are clarified by looking ahead (this is essentially the basis of Judea Pearl's explanation in [JP], discussed below in Section 4), but at the same time, many simple positions have complicated successors further down the tree of moves.

The question that we have raised applies to many situations besides chess, and indeed our project is not really about chess, but about decision making in complex environments. Imagine, for example, a traveler who needs to drive from downtown Manhattan to the Upper West Side of New York City. She is only vaguely familiar with the layout of the city, but she does have a detailed map. Using the map is difficult in traffic, and it is time-consuming to park somewhere for a closer look. Furthermore, the map is not completely clear about one-way streets, and says nothing at all about traffic jams, road construction, or stop lights. Should she try to trace out as much of the route on the map as can be reliably remembered, including all sorts of alternate routes to allow for unforeseen complications (i.e., look ahead as far as possible, keeping in mind that no one could memorize a complete plan, allowing for all contingencies, in such a complex situation), or is it better to adopt, say, a one step look-ahead plan, such as always trying to head uptown and somewhat toward the west.

Consider also a father facing a difficult decision about how to discipline a child. Should he try to calculate all the possible ways in which his child, and his other children, might react, and then how they subsequently might react to his reaction to their reaction? Or should he simply do what seems fair on general principles? One can easily find similar examples in a variety of other settings, from the worlds of economics and politics, to the affairs of the heart.

The strength of computers lies in their ability to calculate the consequences of various decisions much more completely and farther into the future than any human being. Their weakness lies in their inability to properly apply general principles in the evaluation of any given situation.

Humans, on the other hand, can become quite sophisticated about applying general principles. For them, looking very far into the future is risky. Consider for example a chess master, commenting on some game, who declares that a

certain move is good (or bad), giving as proof a 15-move variation with "best" play by both sides. Although the evaluation of the final position in his variation may be unambiguous, and there may be no obvious blunder in the given line of play, the possibility that at each step there might have been a slightly better move leaves the conclusion still very much in doubt. One could say that the master's analysis is corrupted by 'roundoff error'.

Computers using the Shannon algorithm typically search exhaustively every possible move up to a certain depth, so they are not obviously subject to this kind of error. But they still must base their decisions ultimately on the evaluations of positions that lie on their 'foresight horizon'. Since these evaluations are quite error-prone, it still seems that the accumulation of errors could be a serious problem.

Are there rules, "meta-principles", prescribing when making choices based on general principles is superior to looking as far ahead as possible, and vice versa? If some calculating is to be done, are there meta-principles prescribing what to calculate? The Shannon algorithm for game play by computers, which we will describe in Section 2 is based on the belief that seeing further ahead is seeing better, and so comes firmly down on the side of calculation at the deepest possible level in a decision tree. But when sight is cloudy at best (as is the case when a computer looks at any given chess position), is this belief justified? Since the Shannon algorithm does not make any allowance for its own imperfect vision, we think not, at least not all of the time. We will, in fact, present examples in this paper to show that looking ahead can be very misleading to the decision maker.

Of course, we are not the first to have noticed that looking far ahead is not always the best policy. Both D. F. Beal [DB] and Dana Nau [DN] independently discovered that the Shannon algorithm becomes asymptotically useless with deeper look-ahead for certain very simple games. Their discoveries have led several researchers to try to explain the practical successes of the Shannon algorithm (see, for example, [JP] and [GS]) in the case of chess-playing computers. But relative to the tremendous amount of effort put into hardware and software intended to help computers look ever deeper, there has been surprisingly little work directed at understanding the *failures* of look-ahead in general, and the Shannon algorithm in particular.

Information and information processing considerations play an important role in economic theory, including capital theory, finance, macroeconomics, game theory, and labor theory. In all of these situations it is assumed that the agents are aware of their own limitations, and that they can quantify their ignorance. Across all of these decision problems, knowing more means doing better. Chess is a fascinating paradigm for decision making in complex environments precisely because, as in most real life situations, it is a setting in which we cannot accurately quantify our ignorance. We should therefore not be surprised if some natural decision making routines do not have the property that they perform better when they are given more information. As we shall see, the Shannon

algorithm is particularly vulnerable to this problem since it makes no attempt to compensate for its own fallibility.

In order to compensate for one's ignorance, it would seem necessary to quantify it, but how does one do that in a real life setting like chess? In our project we propose to solve this dilemma by following a two-step plan: First we build and analyze a precise model in which the ignorance is quantified. Our analysis yields several results concerning both optimal and effective decision making, and also leads to some heuristics for efficiently searching a decision tree. This first part of our program is carried out in Sections 2 through 5. Second, keeping in mind that our precise models cannot be faithful portraits of reality, we look for principles of decision making that are independent of the way in which we quantify ignorance. Development and discussion of these principles are found in Section 6. In Section 7 we show how they can be used to explain important modifications to the Shannon algorithm that are widely used in modern chess-playing computer programs. Suggestions for further applications of these principles to developing better decision making algorithms are found in Section 8.

2 Basic definitions and preliminary results

In this section we want to lay a foundation for the rest of the paper by making some definitions, introducing some notation, describing the Shannon algorithm, and giving an example that shows that the Shannon algorithm is in some cases far from optimal. Throughout the rest of the paper, we will find it convenient to speak of 'games', 'players', 'moves', etc., instead of 'decision problems', 'decision makers', 'decisions', and so on. We will also often use the game of chess to motivate our models. But we repeat the fact that we are really interested in a general class of decision problems rather than any specific kind of game.

We define a *game tree* to be a finite tree with labeled nodes. We will often use the terms node and *position* interchangeably. The root node on a game tree is sometimes called the *initial position*. The nodes that lie on the unique path from the root node to a given node x are called the *ancestors* of x. In this definition, we do not consider x to be an ancestor of itself. All of those nodes of which x is an ancestor are called the *descendants* of x. The immediate ancestor of x on the tree is called the *predecessor* of x. The immediate descendants of x are called the *successors* of x. Every node except the root node has a unique predecessor. Nodes that have no successors are called *terminal* nodes.

A *labeling* is a function ℓ from the set of nodes of $\mathcal{T}$ into a finite set L, the set of *position types*. Thus, for each node x, $\ell(x)$ is the label attached to x. From a mathematical point of view, it will be seen that the label on a node x contains the information that determines the way in which the labels on the successors of x are generated. In applications, we take the point of view that the label on a position x contains the information that a sophisticated player could glean by examining the position x.

We will want to model both 1-person and 2-person games. In the 2-person games, the players are called A and B. We will not necessarily assume that A is the person that moves in the initial position. We will also not necessarily assume that the players alternate moves. This last statement means that 2-person games include 1-person games as a special case.

We will assume that all terminal positions x have been assigned a value $v(x)$. This represents the payoff that is received by Player A if the game ends at x. The payoff to Player B is $-v(x)$, so we are restricting our attention to zero-sum games. We will soon see how to use the values of the terminal positions to assign values to the remaining positions.

In general, we will require that the label on a position x contain at least the following information:

1. Whether or not x is a terminal position.
2. If x is a terminal position, the value $v(x)$.
3. Which player is to move at position x.

These requirements are quite formal, and in practice, when we speak about the label of a position, we will often suppress one or more of the items listed. Thus, for example, in most cases it is not necessary to explicitly write out the part of the label that says which player is to move, since that information is often understood from the context.

In addition to the above items, the label of a position may also include other information, about such things as the complexity of the position, the tendency of the position to lead to other positions of a certain type, and so on. In some cases, a labeling assigns a value to each position, not just the terminal positions. These aspects of labels are up to the model maker. Various possibilities will be illustrated by examples to be given later.

In either the 1- or the 2-person case, a *move* in a given position x is a choice of one of the successors of x. When we are discussing the situation in which a move is to be made in position x, then we often call x the *current position*, and the successors of x the *alternatives*. Once a decision is made, then one of the alternatives becomes the new current position.

Chess, like many other games, can be represented by a finite tree, since the rules include provisions for preventing a game from going on forever. The two players are traditionally called 'White' and 'Black' instead of A and B. The terminal positions have the values 1, 0, or -1, corresponding to a Win, Draw, or Loss for White. In 1912, the great logician Zermelo showed how all of the non-terminal positions in the chess game tree can be given values by a simple inductive procedure. Suppose that all of the successors of a given position x have been assigned values. If White is to move at x, then the value of x is defined to be the maximum of the values of the successors of x. If Black is to move, the value of x is defined to be the minimum of the values of the successors

of x. It is easy to give an inductive argument to show that in this way, all of the positions x can be assigned a value $v(x)$. We call this value the *Zermelo value* of x.

The same procedure can be applied in general, for any choice of the values of the terminal positions (not just 1, 0, -1) in any finite 1- or 2-person zero-sum game. From now on, we will assume that this procedure has been carried out in our game trees, so that $v(x)$ is defined for all positions x. Of course, we do not assume that knowledge of $v(x)$ is available to the players of the game.

The meaning of the Zermelo value $v(x)$ is as follows: if the game has reached the position x, then from that point on, there exists a strategy for player A that ensures that the payoff is at least $v(x)$, and a strategy for player B that ensures that the payoff (to A) is no more than $v(x)$. So if both players play optimally (in the usual game-theoretic sense), starting from position x, they will both follow such strategies, and the game will end at a terminal node with value $v(x)$. This statement is easy to prove by induction, starting at the terminal nodes. Zermelo used this argument to show that the initial position in chess must have a well-defined value (1, 0, or -1), which indicates the result of a game between two perfect chess players. (This argument only implies existence; the actual value of the opening position in chess is not known.)

The procedure described above by which values are assigned to positions is called *min-max back propagation*. The idea of assigning values to positions by back propagation was already familiar to leading chess players of the 19th century. It had also been introduced into economics by the great Austrian economist Karl Menger in the 1870s. (He imputed value to productive machinery from the value of the consumption goods produced.) However, back propagation alone is useless unless the entire game tree is known. In games like chess, and in many practical problems, the tree is simply too large. Another idea was needed before back propagation could be applied to complex decision problems.

In the context of chess, this missing idea was provided by Wilhelm Steinitz, the first universally acclaimed world chess champion and the acknowledged father of positional chess. (Incidentally, Steinitz lived in Vienna at the same time as Menger.) He reasoned that the value of a position could sometimes be obtained from general principles, without any further calculations. To the extent that positional considerations lead to correct values, they must agree with the results of back propagation. Emanual Lasker, the mathematician who defeated Steinitz for the world title, claims that Steinitz's insight led to the death of the so-called 'romantic school' in chess. If it can be seen on purely positional grounds that White has at least a draw, there is no point in Black making long calculations to find a brilliant winning move. Until Black can see that White has made a mistake, no such move will be available.

The Shannon algorithm, first suggested by the information theory pioneer Claude Shannon in the 1950s, combines the ideas of Zermelo and Steinitz. We now describe the basic algorithm. Let us call the entire game tree $\mathcal{T}$. Assume

that the game has reached a given position, which as usual we call the current position. For definiteness, assume that player A is the one who must make the move in the current position. In order to choose among the alternatives, player A follows these steps:

1. Starting at the current position, player A calculates as many of the descendants of the current position as time allows. This procedure gives a finite tree $\mathcal{S}$ whose root node is the current position. We call $\mathcal{S}$ the *search tree*. If x is a position in $\mathcal{S}$ whose successors are not all contained in $\mathcal{S}$, we say that x is on the *horizon* of $\mathcal{S}$. We write $\mathcal{H}$ for the set of horizon positions. Typically, many of the positions in $\mathcal{H}$ will not be terminal positions (of the original game tree $\mathcal{T}$), so player A will not know their values.

2. Player A nevertheless assigns a value to each position in $\mathcal{H}$. Such an assignment is called a *positional value*. We write $p(x)$ for the positional value assigned to a horizon position x. In our mathematical model, we will want to be more explicit about the source of $p(x)$, by saying for example how it depends on the information contained in $\ell(x)$, but for now, we merely assume that $p(x)$ is given.

3. Player A now assigns values to the ancestors of the horizon positions in $\mathcal{S}$ by min-max back propagation. In particular, this leads to an assignment of values to each of the alternative nodes leading from the current position.

4. Player A chooses the alternative which has been assigned the highest value. (In the case where it is B's move, B will of course choose the alternative with the lowest value.)

We have described one step of the basic algorithm, which is repeated until a terminal position is reached. We have not been very precise about how the search tree $\mathcal{S}$ is chosen, and in fact, Shannon was also not completely definite about this point. However, he did suggest two possibilities. The simplest is called *constant depth search*, in which $\mathcal{S}$ equals the current position x, together with all descendants of x that lie within k steps of x on $\mathcal{T}$, where k is some fixed positive integer, known as the *search depth*. The second method for choosing $\mathcal{S}$ is called *variable depth search*, which simply means that some parts of the original tree $\mathcal{T}$ are searched deeper than other parts, depending on some search rule. Modern computer programs basically use a constant depth search (although they don't usually carry out the search in the order described), with the depth k being determined somewhat by the complexity of the position and by the amount of time available for computation. Some limited variable depth procedures have been more or less grafted on to avoid certain kinds of mistakes that often result from a constant depth search, but attempts to implement a sophisticated variable depth search rule have been largely abandoned. We will have more to say on this point later.

To summarize: a finite search tree $\mathcal{S}$ is calculated, either by constant depth or variable depth search, starting from the current position x; the horizon positions (terminal nodes of the search tree) are assigned values using positional evaluation; the ancestors of x in $\mathcal{S}$ are assigned values by min-max back propagation; a choice is made among alternatives based on these values; the process is repeated each time a move needs to be made. Note that this method would lead to optimal play if the positional evaluations of the horizon positions always agreed with their Zermelo values.

Over the years, refinements have been made in this algorithm, but it is still the basic method used by virtually all computer programs for choosing moves in chess. The amazing thing is that this approach has been so successful, in spite of the fact that the positional evaluation functions of computer programs are relatively crude. Given a typical chess position to evaluate, the computer is oblivious to many important features of the position that would be apparent to any skillful chess player. Shannon himself realized that faulty values assigned to the horizon nodes may lead to poor choices when the algorithm is applied, and he suggested two ways around this problem: either program more sophisticated positional principles into the computer, to improve its evaluation function, or have the computer increase the depth of search. In practice, almost all of the improvement in the play of chess computers seems to have come from the second approach.

Our question is why this should help at all, at least when the later positions are as hard to evaluate as the earlier ones. To illustrate this question, we give a simple example, taken from [GG1]. The behavior of the Shannon algorithm for a 2-person version of this example was first analyzed in [S]. The 'optimal choice' formula for our example was first given in [FP]. This formula is a very special case of the general formula that we will derive in Section 3. The elementary probability computations given in the current section can also be found in [FP].

Our example involves a 1-person game. Suppose that the game tree $\mathcal{T}$ is a finite binary tree with some fixed depth n. Thus there are 2^n terminal positions. Assume that exactly one of these terminal position, which we call the 'winning' position, has Zermelo value 1, and that the remaining terminal positions ('losing') have the value -1. Let the winning position be chosen at random from all of the terminal positions, each one being equally likely. As described earlier, the non-terminal positions y in $\mathcal{T}$ can now be assigned values $v(y)$ by back propagation. (Keep in mind that there is only one player involved here, so back propagation always proceeds by taking maxima.) The result will be that all of the ancestors of the winning position will be assigned the value 1, and the rest of the positions will be assigned -1.

Let $e(y), y \in \mathcal{T}$ be independent, identically distributed random variables that take the values 1 and -1, with

$$P(e(y) = -1) = \varepsilon,$$

where ε is a number in the interval $[0, 1/2]$, called the *error probability*. This

error probability is assumed to be known to the game player. Assume that the random variables $e(y)$ are also independent of the choice of the winning position. Let

$$p(y) = e(y)v(y), y \in \mathcal{T}.$$

We will say that an error occurs at node y if $v(y) \neq p(y)$. Clearly the events "an error occurs at y" are independent, and have probability ε.

Note the role of probability in the description just given. It enters into both the assignment of Zermelo values, and and into the relationship between the Zermelo and positional values. In the next section, we will be more precise about the ways in which we want randomness to affect our models.

We imagine that the player can 'see' the positional values, but not the Zermelo values. One may think of $p(y)$ as a 'noisy' signal that gives information about the Zermelo value $v(y)$. Or one can think of the positional values as 'informed guesses' about the Zermelo values. In any case, the player must make choices based on the positional values only.

Suppose the player decides to use the basic Shannon algorithm to make choices. Since we are dealing with a binary tree, the choice to be made at each step is between two alternatives. Given that the current position has Zermelo value 1, exactly one of the two alternatives will be correct. Let π_k be the conditional probability that the correct alternative is chosen as the result of applying the Shannon algorithm with a constant depth search to depth k, given that the current position has Zermelo value 1.

Assuming that the current position was reached by repeated application of the Shannon algorithm with depth k, the value of π_k does not depend on the location of the current position, nor on the number of previous steps used to get to the current position, as long as the current position is far enough from the terminal nodes to allow for a depth k search. The reasons for this are the homogeneous manner in which the game tree and Zermelo values were defined, and the independence assumption concerning the occurence of errors. It follows that the probability of making m correct moves in a row by repeatedly applying the Shannon algorithm at depth k is $(\pi_k)^m$.

It is easy to compute the quantity π_k. The search tree $\mathcal{S}$ is, of course, a binary tree. The horizon set $\mathcal{H}$ contains 2^k positions. Assuming that the current position has Zermelo value 1, half of the positions on the horizon are descendants of the correct alternative. Call the correct alternative y_c, and the incorrect alternative y_w. Let Y_c be the positions in $\mathcal{H}$ that are descendants of y_c, and let Y_w be the positions in $\mathcal{H}$. The Shannon algorithm assigns y_c the value 1 if and only if at least one of the positions in Y_c has positional value 1. Conditioned on y_c having Zermelo value 1, this event occurs with probability

$$\begin{aligned} p_c &= (1-\varepsilon) + \varepsilon(1 - (1-\varepsilon)^{2^{k-1}-1}) \\ &= 1 - \varepsilon(1-\varepsilon)^{2^{k-1}-1}). \end{aligned}$$

The position y_w is assigned the value -1 by the Shannon algorithm if and only if all of the positions in Y_w have positional value -1. Conditioned on y_c having Zermelo value 1, this event occurs with probability

$$p_w = (1-\varepsilon)^{2^{k-1}}.$$

We conclude that

$$\pi_k = p_c p_w + \frac{1}{2}(p_c(1-p_w)) = \frac{p_c(1+p_w)}{2}.$$

It is easy to see that $\pi_k \to 1/2$ as $k \to \infty$. In fact, π_k is decreasing in k, so the ability of the Shannon algorithm to pick out the correct alternative degrades to total randomness as the depth of the search increases! This phenomenon, is called the *search-depth pathology*. It was first discovered (independently) by Beal [B] and Nau [N] for models that are different than the one we give, and studied by [S] for a 2-person version of our example.

Since our example is a 1-person game, it is not hard to see what goes wrong with the Shannon algorithm when the search level is deep. The probability of mistaking a -1 for a 1 in at least one of the horizon nodes in the 'bad' set Y_w becomes quite high. Such a mistake gets propagated back to the alternative y_w, and the game player is usually forced into a coin toss to decide between y_c and y_w. The back propagation of errors leads to a disaster for the player. All of the other examples in the literature behave in this manner for much the same reasons (although things are not so transparent in the 2-person examples).

In [GG1], we proposed that better moves could be made for examples like the one just given if *all* of the information were used that is available to the game player that looks ahead k levels in the game tree. The Shannon algorithm only uses the information contained in the positional values of the positions on the horizon of the search tree. Thus, in the Shannon algorithm, searching deeper does not mean using more information; it only means using different information.

It turns out that for the example described above, there is a simple formula that can be used to make optimal use of the information contained in the positional values of the positions in the search. This formula was discovered by M. Fischer and S. Paleologou [FP]. This formula is a special case of a much more general formula to be derived in Section 3. The general formula is quite easy to prove, so we omit the derivation of the special case.

We assume that the decision maker has calculated the search tree $\mathcal{S}$ by performing a constant depth search to depth k, and that the positional values of the positions in $\mathcal{S}$ are known. Consider the quantity

$$R(x) = \sum_{S \in \mathcal{S}(\S)} \prod_{y \in S} \left(\frac{1-\varepsilon}{\varepsilon}\right)^{p(y)},$$

where for a given position x, $\mathcal{S}(\S)$ is the collection of all paths from the current position (i.e., the root node of $\mathcal{S}$) through the position x to some position on the horizon of the search tree. It will be shown in Section 3 that $R(x)$ is proportional to the conditional probability that x has Zermelo value 1, given that the current position has Zermelo value 1 and given the positional values of all of the positions in the search tree $\mathcal{S}$.

We can use the formula as follows. Let the two alternatives be x_1 and x_2. Calculate $R(x_1)$ and $R(x_2)$, and choose the alternative that gives the larger value (breaking ties as before). This procedure maximizes the conditional probability that the correct move will be chosen, given the available information. We will prove in Section 4 that this probability converges to 1 as the depth increases, provide the error probability is not too large. We will also discuss the relevance of this procedure to the problem of repeatedly making correct decisions.

It turns out that there are many interesting examples in which the Shannon algorithm performs suboptimally (and often quite poorly) for which we can provide a more suitable algorithm. Analysis of these examples is contained in Sections 3 and 4.

3 General model and results

In the preceding section, we defined what we mean by a game tree, and we gave an example. An important feature of that example was that it involved a game tree whose labels (which in that case were simply the Zermelo values of the positions) were determined randomly. In general, we imagine that our players are presented with a random game tree, and that their choices are based on (i) a priori probabilistic information that they have about the game tree, as well as (ii) the information that is gained by looking ahead in the tree and evaluating positions. We now wish to formalize this point of view.

For simplicity, we shall assume from now on that all of our game trees have the following property:

Each non-terminal position has exactly b successors.

Here b is an integer parameter greater than or equal to 2. The reader will have no difficulty generalizing our results to trees with variable numbers of successors. If x is a non-terminal position, we will designate the successors of x by $x1, x2, \ldots, xb$, in accordance with a common notational convention used for trees.

We mentioned earlier that the label on a position contains the information needed to generate the descendants of that position. Of course, this is only relevant for labels used on non-terminal positions. Thus, we assume that for each element c of the set of labels L such that c is used to label non-terminal positions, there exists a probability distribution β_c on the set L^b. We call β_c a

successor distribution. In the model we are about to define, if a position x has label c, then β_c tells us probabilistically what to expect about the successors of x and their labels. In particular, for any ordered b-tuple $C = (c_1, \ldots, c_b)$ of labels, $\beta_c(C)$ equals the conditional probability that the successors of a position x are labeled by C, given that x has label c. We assume that this conditional probability does not depend on the location of the node x in the tree, nor is it changed if we also condition on the labels of any positions which are not descendants of x. This last statement is akin to the 'Markov property', so we will call our random game trees *Markovian.* More formally, we have

Definition. A *Markovian game tree* with label set L and successor distributions $\{\beta_c, c \in L\}$, is a random game tree determined by the following condition: Let x be any non-terminal position in the tree, and let $Y(x)$ be the set of positions which are not successors to x. Let $C = c_1, \ldots, c_b$ be a b-tuple of labels in L, and let $A(C)$ be the event that the successors of x are labeled by $c_1, \ldots, c_b$. Then

$$P(A(C)|\ell(y), y \in Y(x)) = P(A(C)|\ell(x)) = \beta_{\ell(x)}(C).$$

This definition is very similar to the definition of a type of stochastic process known as a 'multi-type branching process' (see Harris [H]). However, the tree structure is more or less suppressed in multi-type branching processes, unlike in the above definition.

Implicit in the definition is that the successor distributions must be such that the corresponding random game tree is finite with probability 1, since we insisted earlier that game trees be finite. We will not have much to say about what kinds of restrictions this places on the successor distributions. In the examples that we give, it will always be clear that the random game trees are finite with probability 1. Perhaps in a future paper, we will address the technicalities that arise from either (i) allowing infinite game trees, or (ii) formulating appropriate general conditions on the successor distributions to ensure finite game trees with probability 1.

To illustrate the definition, let us see what it has to do with the example discussed in Section 2. In that example, we had $b = 2$. The labels on the positions are of the form $(\pm 1, k)$, with the first component equaling the Zermelo value, and the second component equaling the depth of the position in the tree. Thus, the initial position has label $(1, 0)$, the successors of the initial position are labeled either $(1, 1), (-1, 1)$ or $(-1, 1), (1, 1)$, and so on. For terminal positions, $k = n$, since we assumed in the example that the tree had depth n. The successor distributions are defined by

$$\beta_{(1,k)}[(1, k+1), (-1, k+1)] = \beta_{(1,k)}[(-1, k+1), (1, k+1)] = 1/2$$

$$\beta_{(-1,k)}[(-1, k+1), (-1, k+1)] = 1,$$

for $0 \leq k < n$. Note that the main function of the second component of the label is to mark terminal positions. In practice, one may suppress this component without confusion.

We also want to formalize the notion of positional evaluations. For this purpose we assume that given the random game tree $\mathcal{T}$, there is a collection $\{e(y) : y \in \mathcal{T}\}$ of independent, identically distributed random variables, called the *error variables*, each taking values in some finite set E.

Definition. An *evaluation function* is a function $\varphi : L \times E \to (-\infty, \infty)$. The positional evaluation of a position $y \in \mathcal{T}$ is given by $p(y) = f(\ell(y), e(y))$.

There are many equivalent ways in which we could have handled positional evaluations. For example, we could have chosen to include the noise variables as part of the random label assigned to each node. The approach that we have taken seems to be adequate for our purposes, and to cause less notational problems than the alternatives.

It is easy to see how the definition of evaluation function relates to our simple example. Let $E = \{-1, 1\}$. Given an error probability ε, let the error variables $e(y)$ have probability distribution on E defined by $P(e(y) = -1) = 1 - P(e(y) = 1) = \varepsilon$. For $v, e \in \{-1, 1\}$, we let $\varphi(v, e) = ve$. (Technically speaking, we have not quite conformed to our definition in defining φ here, since the labels in our example take the form (v, k). But evaluations do not depend on the depth k in the tree in that example, so as indicated earlier, we find it convenient to suppress the k.)

We claim that using Markovian decision trees together with evaluation functions as we have defined them provides us with realistic models for decision problems. We take the point of view that the label of a node x contains all of the information needed to generate (randomly) the successors to that node. We imagine that a very sophisticated decision maker can "see" $\ell(x)$ when looking at x. To decide or estimate the value of x, the sophisticated decision maker does not need to look at nodes that are not successors of x. The future unfolds in a Markovian way for the sophisticated observer. For example, a grandmaster in chess can typically evaluate a position fairly accurately without knowing how the position was arrived at, or what other positions could have been arrived at by making different moves earlier in the game.

On the other hand, an unsophisticated decision maker may not be able to use all of the information contained in the label. Chess computers, for example, see very little of the relevant information contained in any given position. Our model allows for this. For example, in the simple model analyzed in the previous section, we assumed that the decision maker could only "see" $p(x)$, which was a noisy version of $v(x)$. Even though the decision tree itself was Markovian, the part seen by the decision maker was not.

In our general model, we allow the future to unfold randomly from each node x, even when all of the information contained in $\ell(x)$ is known. This accords well

with many real-life situations, even deterministic ones. For example, in chess, it is often the case that even after long analysis, a top grandmaster can only make probabilistic statements about the value of a position or about the kinds of positions that are likely to arise later on in the game, such as "This position offers good chances for a queenside attack for White, with a difficult defense in store for Black". These kinds of statements can be viewed as statements about the successor distributions β_ℓ.

By assuming that the decision maker can only look a bounded distance into the future, we are assuming a certain type of ignorance. And when we restrict the ability of the decision maker to use all of the information contained in the label of a position, we are assuming another type of ignorance. The Shannon algorithm essentially behaves as if neither of these types existed. Some of the work done by other researchers on the search-depth pathology have taken into account the first type of ignorance. As far as we know, no one considered the second type before it was discussed in [GG1].

It is a mathematical fact that *any* rule for generating decision trees satisfies the Markov property, provided the set of labels is suitably enlarged. But this fact is quite useless to us, because it is unreasonable to assume that even sophisticated decision makers have unlimited capacity for distinguishing situations from one another. Even grandmaster chess players claim to discriminate among surprisingly few different types of positions. We will usually want to make the set of labels as *small* as possible. Many interesting examples are possible with only four labels, as in the example in Section 2.

Most of the models considered in the literature fit easily into the Markovian mold. In the most common model, trees are generated by assigned Zermelo values independently to the terminal nodes according to some fixed distribution, and then by assigning positional values to the nodes by randomly perturbing the Zermelo values, similar to what was done in our earlier example. Such models satisfy our definitions when the natural labeling $v(x)$ is used. The models considered in [S] and [Ba] are explicitly Markovian. One model that doesn't fit nicely into our general setup is one in which exactly k of the terminal nodes are assigned Zermelo value 1, with the remaining terminal nodes having value -1. For $k > 1$, there is no natural labeling that makes such a model Markovian. (Of course, if $k = 1$, we have our earlier example.)

We conclude this section with some general mathematical results concerning our model. These results are quite easy to derive. Unfortunately, they are not very useful as far as direct practical applications are concerned. We will, however, find them to be valuable starting points for the analysis of more specialized situations in Section 4, and we will also refer to them when we derive general principles in Section 6.

Let us assume that we have chosen a search tree $\mathcal{S}$ and a position $x \in \mathcal{S}$. We will write ℓ for the labeling assigned to the nodes in $\mathcal{S}$, even though strictly speaking, ℓ actually refers to the labeling of the entire tree $\mathcal{T}$. Let V be a real number. We are interested in deriving a formula for the conditional probability

that $v(x) > V$, given the information contained in the positional values $p(y)$ for $y \in \mathcal{S}$. Let $\tilde{p}(y), y \in \mathcal{S}$ be real numbers. A simple application of Bayes' rule gives

$$\begin{aligned} & P(v(x) > V | p(y) = \tilde{p}(y), y \in \mathcal{S}) \\ &= \sum_{\tilde{\ell}} P(v(x) > V, \ell = \tilde{\ell} | p(y) = \tilde{p}(y), y \in \mathcal{S}) \\ &= \sum_{\tilde{\ell}} \frac{P(v(x) > V, p(y) = \tilde{p}(y), y \in \mathcal{S} | \ell = \tilde{\ell}) P(\ell = \tilde{\ell})}{P(p(y) = \tilde{p}(y), y \in \mathcal{S})}, \end{aligned}$$

where the sum is taken over all labelings $\tilde{\ell}$ of $\mathcal{S}$. The Markovian assumption and the definition of positional values imply that the events $v(x) > V$ and $p(y) = \tilde{p}(y), y \in \mathcal{S}$ are conditionally independent, given the event that $\ell = \tilde{\ell}$, so we have

$$\begin{aligned} & P(v(x) > V | p(y) = \tilde{p}(y), y \in \mathcal{S}) \\ &= \frac{\sum_{\tilde{\ell}} \left[P(v(x) > V | \ell = \tilde{\ell}) P(p(y) = \tilde{p}(y), y \in \mathcal{S} | \ell = \tilde{\ell}) P(\ell = \tilde{\ell}) \right]}{P(p(y) = \tilde{p}(y), y \in \mathcal{S})} \\ &= \frac{\sum_{\tilde{\ell}} \left[P(v(x) > V | \ell = \tilde{\ell}) P(p(y) = \tilde{p}(y), y \in \mathcal{S} | \ell = \tilde{\ell}) P(\ell = \tilde{\ell}) \right]}{\sum_{\tilde{\ell}} \left[P(p(y) = \tilde{p}(y), y \in \mathcal{S} | \ell = \tilde{\ell}) P(\ell = \tilde{\ell}) \right]}. \end{aligned}$$

The summands in the preceding expression are made up of three different quantities that depend on $\tilde{\ell}$:

$$\begin{aligned} P_1 &= P(\ell = \tilde{\ell}) \\ P_2 &= P(p(y) = \tilde{p}(y), y \in \mathcal{S} | \ell = \tilde{\ell}) \\ P_3 &= P(v(x) > V | \ell = \tilde{\ell}). \end{aligned}$$

Each of these can be made more explicit.

The first quantity, P_1, can be expressed in terms of the successor distributions. For each non-horizon position $y \in \mathcal{S}$, let $y1, \ldots, yb$ be the successors of y, as usual. Let r be the root node of $\mathcal{S}$. Then

$$P_1 = P(\ell(r) = \tilde{\ell}(r)) \prod_{y \in (\mathcal{S} \setminus \mathcal{H})} \beta_{\tilde{\ell}(y)}(\tilde{\ell}(y1), \ldots, \tilde{\ell}(yb)).$$

In practice, the quantity $P(\ell(r) = \tilde{\ell}(r))$ could be approximated by simulating the process by which the game tree is generated.

The second quantity, P_2, can also be written as a product. For each real number q and each label c, let

$$d(q, c) = P(p(z) = q | \ell(z) = c),$$

where z is any position in the tree (our assumptions on the error variables imply that $d(q,c)$ does not depend on the choice of z). This quantity can be computed from the distribution of the error variables and the function φ used to define the position values. Since we assumed that the error variables are independent, we have

$$P_2 = \prod_{y \in \mathcal{S}} d(\tilde{p}(y), \tilde{\ell}(y)).$$

We now come to the quantity P_3. We will describe a back-propagation procedure for finding $P(v(y) > V | \ell = \tilde{\ell})$ for all $y \in \mathcal{S}$. Call this quantity $\pi(y)$. Note that $P_3 = \pi(x)$. First consider $y \in \mathit{horizon}$. The Markovian assumption implies that

$$\pi(y) = P(v(y) > V | \ell(y) = \tilde{\ell}(y)).$$

We will assume that the quantity on the right side of this expression is known. In practice, it could be approximated using simulation. Now suppose that y is a position whose successors $y1, \ldots, yb$ are in $\mathcal{H}$. There are two cases: (i) y is a position in which Player A is to move; (ii) y is a position in which Player B is to move. In the first case

$$\begin{aligned}
\pi(y) &= P(v(y1) > V \text{or} v(y2) > V \text{or} \cdots \text{or} v(yb) > V | \ell = \tilde{\ell}) \\
&= 1 - P(v(y1) \leq V \text{and} v(y2) \leq V \text{and} \cdots \text{and} v(yb) \leq V | \ell = \tilde{\ell}) \\
&= 1 - \prod_{m=1}^{b} (1 - P(v(ym) > V | \ell = \tilde{\ell}) \\
&= 1 - \prod_{m=1}^{b} (1 - \pi(ym)).
\end{aligned}$$

We have used the fact that, because of the Markovian assumption, the events $v(ym) > V, m = 1, \ldots, b$ are conditionally independent, given $\ell = \tilde{\ell}$. In case (ii),

$$\begin{aligned}
\pi(y) &= P(v(y1) > V \text{and} v(y2) > V \text{and} \cdots \text{and} v(yb) > V | \ell = \tilde{\ell}) \\
&= \prod_{m=1}^{b} \pi(ym).
\end{aligned}$$

We have described here how to back-propagate the quantity $\pi(y)$ from horizon positions to predecessors of horizon positions. It is easy to see that this procedure can be continued all the way back to the root of $\mathcal{S}$, using

$$\pi(y) = 1 - \prod_{m=1}^{b} (1 - \pi(ym))$$

if Player A is to move in position y, and

$$\pi(y) = \prod_{m=1}^{b} \pi(ym)$$

otherwise. This form of back-propagation is called *product rule* back-propagation. Note that if $\pi(y) = 0 \text{or} 1$ for all $y \in \mathcal{H}$, then product rule back-propagation reduces to min-max back-propagation. This will occur if the labels on the horizon positions in $\mathcal{S}$ give the Zermelo values of those positions.

To summarize, we have shown that

$$P(v(x) > V | p(y), y \in \mathcal{S})$$
$$= \frac{\sum_{\tilde{\ell}} P(\ell(r) = \tilde{\ell}(r)) \pi(x, \tilde{\ell}) \prod_{y \in \mathcal{S} \setminus \mathcal{H}} \beta_{\tilde{\ell}(y)}(\tilde{\ell}(y1), \ldots, \tilde{\ell}(yb)) \prod_{y \in \mathcal{S}} d(p(y), \tilde{\ell}(y))}{\sum_{\tilde{\ell}} P(\ell(r) = \tilde{\ell}(r)) \prod_{y \in \mathcal{S} \setminus \mathcal{H}} \beta_{\tilde{\ell}(y)}(\tilde{\ell}(y1), \ldots, \tilde{\ell}(yb)) \prod_{y \in \mathcal{S}} d(p(y), \tilde{\ell}(y))},$$

where we have written $\pi(x, \tilde{\ell})$ for $\pi(x)$ to emphasize the dependence on $\tilde{\ell}$. As mentioned earlier the sums are taken over all labelings $\tilde{\ell}$ of $\mathcal{S}$.

In principle, at least, all of the terms in this expression can be computed directly from the parameters of the model (i.e., from the successor distributions and the distribution of the error variables), or by approximation through simulation. This last method could also be used to approximate the parameters themselves. Of course, the model itself is only an approximation of real-world situations, and so in applications of the formula, questions of robustness should be considered.

Assuming that all of the individual terms in the formula can be computed accurately, there still remains one major difficulty: the number of summands in the formula grows exponentially in the number of positions in the search tree $\mathcal{S}$. If the search tree has depth k, there are b^k positions in the horizon alone, so computation time grows super-exponentially in the depth of the tree. The good news is that it will be possible to draw some general conclusions from the formula that don't rely on being able to carry out the actual computation. Also, in certain special cases (to be discussed in the next section) computation time is cut down to a reasonable size because most of the summands equal 0.

4 Results for interesting special cases

To give some indication of what is possible, we will give some examples of some useful formulas that can be derived for models like the ones just described. To make things somewhat concrete, we will assume that the type of a given position x contains two pieces of information: the true value $z(x) = 1$ or -1 of the position, and the probability $p(x)$ that we can correctly guess $z(x)$. Thus, $p(x)$ is an indicator of how difficult it is to evaluate the value of x. We assume that when we look at the position x, we can correctly determine the probability $p(x)$ (experts generally can tell how "clear" or "unclear" a position is, although perhaps not with the precision that we assume here). We will write $q(x)$ as a shorthand for $1 - p(x)$. This is the error probability. All guesses about the values $z(x)$ for various positions x are made independently. Our guess of the value of $z(x)$ is called $g(x)$. Thus, if we make such a guess, $g(x)$ is either 1 or

-1. However, there may be some positions x for which we make no guess (i.e., for lack of computation time), in which case we let $g(x) = 0$.

Let S be any set of nodes in the tree, and let $\rho(S)$ be the a priori probability that S is precisely the set of nodes y in the tree whose true values are equal to 1. In the Markovian case, it is easy to calculate $\rho(S)$ as a product of transition probabilities. After making the guesses g, the posterior probability that S is precisely the set of actual nodes labeled with 1's can be written as $P(S|g) = c(g)R(S|g)$, where $c(g)$ is a constant depending on the guesses g and

$$R(S|g) = \rho(S) \prod_{y \in S} \left(\frac{p(y)}{q(y)} \right)^{g(y)} .$$

This expression can be used to obtain formulas for interesting conditional probabilities. For example, to find the conditional probability $P(x)$ that a given node x has the value $z(x) = 1$, given the guesses made about other nodes y in the tree, we evaluate

$$P(x) = \frac{\sum_{S \ni x} R(S|g)}{\sum_S R(S|g)}.$$

The sum in the numerator is taken over sets S that contain the node x, and the sum in the denominator is taken over all sets S. This formula tells us which of several nodes we should move to, if we want to maximize our probability of moving to a good node, given our observations: we simply move to the node x with the highest $P(x)$. However, as we shall see, this way of using the formula is not very practical, and we have other more important uses in mind.

There are several ways in which the computation of $P(x)$ can be simplified. First of all, many sets S represent ways of labeling the tree with 1's which are inconsistent either with backward induction or with some extra assumptions we may have made about the model, and those sets can be left out of the formula (since for them, $\rho(S) = 0$). Secondly, since the denominator does not depend on x, and since we are often only interested in comparing the sizes of $P(x)$ for different choices of x, we frequently only need to compute the numerator, or any other quantity that is proportional to the numerator. In the Markovian case, if we are only interested in the relatives sizes of the numerators, and if we assume that there is no a priori reason to distinguish between nodes, then it is possible to restrict the sum to those sets S that contain no nodes other than the root node, the node x, and the successors of x. In the case of a 1-person decision tree in which it is known a priori that there is only one correct terminal node, each of which is equally likely, with fixed guessing probability p and a fixed number of branches stemming from each non-terminal node, we can use the following expression in the place of $R(S)$:

$$\tilde{R}(S) = \begin{cases} \left(\frac{p}{q}\right)^{2N(S)} & \text{if } S \text{ is a path to a terminal node} \\ 0 & \text{otherwise} \end{cases} ,$$

where $N(S)$ is the number of nodes in S for which we guess the value 1. Since there is only one correct terminal node, the only possible correct labelings of the tree are those in which the set of 1's is a path to a terminal node, explaining why $\tilde{R}(S)$ is 0 for other sets S. (This is essentially the formula found by Fischer).

5 Heuristics for searching the decision tree

We can also obtain a formula which tells us, for any nodes x and y, how informative the true value of y is about the true value of x. First we define the quantity

$$\Delta(x,y) = P(z(x) = 1 | z(y) = 1) - P(z(x) = 1 | z(y) = -1),$$

which describes how much $P(z(x) = 1)$ changes as $z(y)$ changes. To obtain a useful measure of how informative the value of y is about the value of x, we need to take into account how much $z(y)$ changes, so we define

$$I(x,y) = \Delta(x,y) P(z(y) = 1) P(z(y) = -1).$$

The quantity $P(z(y) = 1)P(z(y) = -1)$ is proportional to the variance of $z(y)$, and $I(x,y)$ is proportional to the expected change in $P(z(x) = 1)$ that results from observing $z(y)$. An easy calculation shows that

$$\begin{aligned} I(x,x) &= P(z(x) = 1)P(z(x) = -1) \\ &\geq \Delta(x,y)P(z(y) = 1)P(z(y) = -1) = I(x,y). \end{aligned}$$

To compute $\Delta(x,y)$ in case y is an immediate successor of x in a 2-person game between White and Black,

$$\Delta(x,y) = \begin{cases} \frac{P(\text{all immediate successors of } x \text{ have the value -1})}{P(z(y)=-1)} \\ \qquad \text{if it is White's move in position } x \\ \frac{P(\text{all immediate successors of } x \text{ have the value 1})}{P(z(y)=1)} \\ \qquad \text{if it is Black's move in position } x \end{cases}.$$

It is clear that if White is on the move at x, then the immediate successor y which maximizes $\Delta(x,y)$ is the y with the highest $P(z(y) = 1)$. If Black is on the move at x, the immediate successor y which maximizes $\Delta(x,y)$ is the y with the lowest $P(z(y) = 1)$. The same is true for $I(x,y)$ in both cases, as is evident from the relationship between $I(x,y)$ and $\Delta(x,y)$.

There is a nice formula for $\Delta(x,y)$ (and hence for $I(x,y)$) in the Markov case when y is a successor of x, but not necessarily an immediate successor.

If $x, y_1, \ldots, y_k, y$ is a chain of immediate successors leading from x to y, then assuming the Markov property, we can compute $\Delta(x, y)$ by taking the following product of terms from the previous formula:

$$\Delta(x, y) = \Delta(x, y_1)\Delta(y_1, y_2) \cdots \Delta(y_k, y) .$$

Furthermore, in this case, we can calculate the informative value about $z(x)$ of a *guess* about y:

$$I(x, g(y)) = \Delta(x, y)P(z(y) = 1)P(z(y) = -1)(p(y) - q(y)).$$

When deciding which successor of x one should first examine to learn about the value of x, simply choose the y for which $I(x, g(y))$ is largest. As with the formulas related to $P(x)$, the most important applications of these expressions are not for the precise evaluation of any particular quantities, but instead for what they can tell us in general.

6 Some general principles

We will give here five principles that can be deduced from the expressions for $R(S)$ and $I(x, g(y))$. In each case, we will first state the principle in everyday language in the context of decision making, and then we will say how the principle is derived from the formulas. Afterwards, we will talk about how the Shannon algorithm fares with respect to these principles, and we will mention several guidelines used by human decision makers that seem to be in agreement with our principles.

1. **The proximity principle.** Think about the immediate consequences of your actions first! More precisely, if y and y' are both successors of x, and if y' is also a successor of y, then y is more informative about x than is y'. We can prove this mathematically, by looking at the formula for $I(x, g(y))$, assuming the Markov property. Suppose y and y' are both successors of x, and that y' is also a successor of y. Then the expressions for $I(x, y)$ and $I(x, y')$ both have the factor $\Delta(x, y_1) \cdots \Delta(y_k, y)$ in common, where y_k is the immediate predecessor of y. What remains in the two expressions is the factor $P(z(y) = 1)P(z(y) = -1) = I(y, y)$ in $I(x, y)$, and $\Delta(y, y')P(z(y') = 1)P(z(y') = -1) = I(y, y')$ in $I(x, y')$. We have already seen that this second factor is never bigger than the first, so $I(x, y) \geq I(x, y')$, in accordance with our principle. The same relationship holds between $I(x, g(y))$ and $I(x, g(y'))$, provided $p(y')$ is not larger than $p(y)$. The relationship can sometimes be reversed if we are sufficiently certain of our guesses about y' and/or sufficiently uncertain of our guesses about y. In such a case, the "clear sight principle" comes into play, as explained below.

The formula for $P(x)$ also bears out the proximity principle, since nodes y that are closer to x typically figure in more of the terms $R(S|g)$ for sets S that contain x.

2. **The relevance principle.** Examine the most promising alternatives first! Or, in the context of games, continue your investigations first along the apparent intended lines of play. In particular, if you want to learn about x by looking at one of the immediate successors of x, look at the node y that is a priori most likely to be chosen by the player on the move. Again, this principle is based on the formula for $I(x, y)$ (assuming the Markov property). It is easily checked that each term $\Delta(y_i, y_{i+1})$ in the product increases if y_{i+1} is a better choice for the player on the move in position y_i, and that the last term $\Delta(y_k, y)P(z(y) = 1)P(z(y) = -1)$ also increases if y is a better move for the player on the move in y_k.

   If y succeeds y', then the relevance principle implies that y should be examined before y', since the chances are necessarily higher that y will be played than that y' will. The relevance principle thus implies the proximity principle.

   If y and y' both succeed x, and y' is deeper in the tree than y, but *does not* succeed y, it might seem that we should first examine y. But this will not be true if along the path from x to y', the Δ terms are unusually high. This will occur, as the formula shows, when the moves along the path to y' are more or less "forced" (i.e., the alternatives are obviously bad for the player on the move). Moreover, $I(x, y')$ is also increased if y' is a good position for the player who is currently trying to make the decision.

3. **The family principle.** Make decisions that lead to many good options, rather than relying on one excellent possibility. Having many successors nodes with reasonably good prospects is better than having a few successors with great prospects. Formulating this principle in precise mathematical terms leads to some interesting questions of statistics that, to our knowledge, have never been addressed. Namely, suppose you observe two sets (or "families") of numbers, and want to choose the set that contains the largest number (the "best child"). If your observations are subject to error, how should you choose? The family principle says that you need to consider the apparent combined strengths of all the children in a family, rather than just that of the apparently best child. There are two distinct reasons for this. The first is that for the family that has a greater number of "promising" children, it is more likely that one of them will actually be much better than your observations led you to believe. The second is that for a family with one very promising child and many mediocre children, the fact that you observed many mediocre children makes it more likely that your estimation of the apparently outstanding child is too high.

The formula for $P(x)$ tells us how to do this for a given model, but more research is needed to find more clear cut guidelines.

4. **The stability principle.** Consistency increases confidence. If our guesses were accurate, we would guess values along the intended path that changed very little. Thus, if there is a terminal node whose predecessor nodes look consistently good from beginning to end, we should prefer that over a terminal node whose predecessor nodes appear to fluctuate in value, which in turn is preferable to a terminal node with consistently bad predecessors. If there is a lot of fluctuation in the values along the path to a terminal node, we may want to increase our confidence in its evaluation by looking at its successors. One mathematical manifestation of this principle comes out of Fischer's $P(x)$ formula for the special case of a 1-person decision tree with only one correct path. When trying to decide which of the terminal nodes is most likely to be the correct one, the formula implies that one should choose the one that lies on the end of the path that contains the most nodes for which we have guessed the value 1, provided we have looked at the same number of nodes for each path.

5. **The clear sight principle.** If possible, base your decisions on clear-cut results. In the formula for $P(x)$ we see that observations with the highest $p(y)/q(y)$ ratio receive the most weight. In the extreme case, when all of the $q(y)$ are very small, the Shannon algorithm might work well. We must emphasize the "very small" here, because this principle is essentially in conflict with the previous principles, and it doesn't take much in the way of observation errors to make them dominate. Judea Pearl shows in his book, under certain special assumptions on the decision tree, the most important of which is indepence of the values of the terminal nodes, that if the proportion of absolutely clear positions ($q(y) = 0$) in the tree goes to 1 as the tree gets deeper, then the search-depth pathology goes away, and basing your decision on the values of the terminal nodes is a good idea. This principle is the reason he gives for the successes of the Shannon algorithm. The Shannon algorithm should be modified to include some assessment of the size of $q(y)$ for each node that it looks at.

Not surprisingly, there are several folk maxims that sound like the principles just given. For example, the proximity and stability principles both are related to the phrase "the end never justifies the means", while the family principle sounds like "don't put all your eggs in one basket". This last phrase is also a warning about reliance on the clear sight principle in cases where we think our guessing errors are too unlikely to worry about ("I've got a sure thing in the fifth race ..."). "Follow your nose" could be interpreted as the proximity principle, and "go with a winner" is clearly a form of the relevance principle. "A bird in the hand is worth two in the bush" is a version of the clear sight principle. It is amusing to see these proverbs appear in the mathematical analysis.

Great chess players also seem to be aware of the principles we have given. A grandmaster typically does not choose a move based on looking far ahead. Instead, he usually chooses one or two plausible moves based on looking only one move ahead (the proximity principle), and then investigates those moves further by following what look like the most likely continuations (the relevance principle). In certain circumstances, they do look ahead to try to find positions whose value is obvious (the clear sight principle), so that they can avoid making blunders or falling into traps, or so that they can possibly find forcing moves that lead to a clearly won position. They have a keen sense of what types of positions will make available many good continuations for them (the family principle), and they are suspicious of lines that seem to end well, but are reached by way of positions that appear unsafe (the stability principle).

7 Implications for the chess world

Since the Shannon algorithm is unaware of its own "bounded rationality", it ignores the principles just enunciated. As far as we know, in the 40 years since Shannon first introduced his method, there have been only three modifications that attempt to compensate for these limitations. The first one, introduced early on, allows the positional evaluator to assign a whole range of values to positions, even though there are only three possible true values for any position (win, lose or draw). The effect of this procedure is that positions which are clear wins or losses are given extra weight (because they are given very high or low values respectively) in the Shannon algorithm. This is in keeping with the clear sight principle.

The second innovation is a limited kind of stability analysis. It is typically used in chess when a piece capture, say White queen takes Black knight, occurs at the end of a path in the tree of moves being searched. Immediately after this capture, the value of the position appears to suddenly have changed, in favor of White, since Black is missing a knight. But it is often the case that in such circumstances, White's queen can be taken by Black on the very next play. Since queens are usually worth more than knights, the position is in fact good for Black. This phenomenon is known as the "horizon effect", and is avoided by searching deeper in those positions where pieces (such as the White queen) are "hanging", or vulnerable to being captured. Note how this is a special case of applying the stability principle.

The third modification is known as "singular extension", and was first introduced by the Deep Thought team in 1987. It noticeably improved the play of Deep Thought, and is now used by all of the top chess playing programs. Briefly, the idea behind singular extension is that whenever the program finds a move that is vastly better than all of its alternatives, it looks deeper to check itself. Singular extension can be understood directly from the relevance principle. If a move to y following x is dramatically superior to its alternatives, then $\Delta(x, y)$

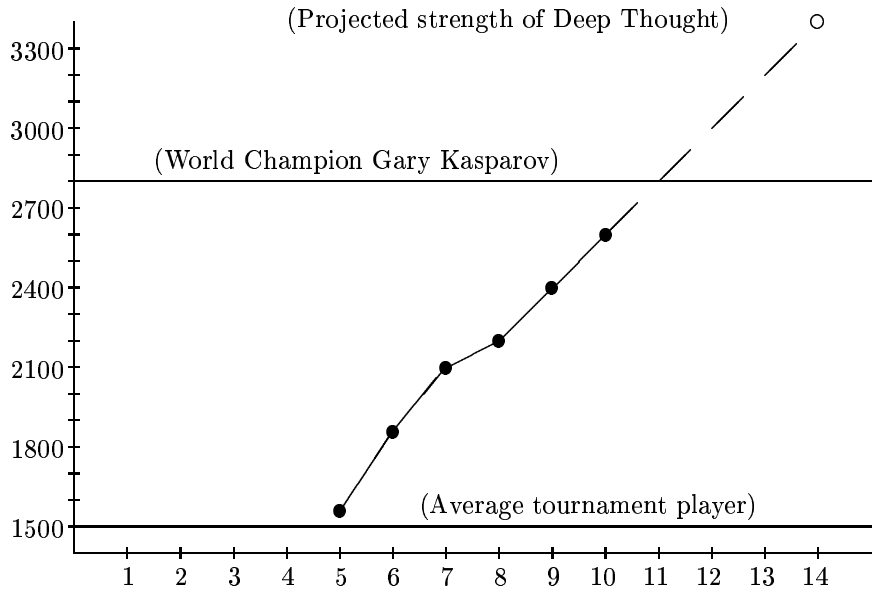

Figure 1: How depth of look-ahead is supposed to increase chess strength

will be very high, and the relevance of successors of y will consequently also be high, and thus worthy of further investigation.

All of the modifications described here were motivated more by ad hoc practical considerations than by any deep understanding of general principles. There has been no attempt at all to be systematic, yet the three most important improvements in the Shannonn algorithm can all be understood on the basis of our general principles. In the following section, we will make some practical suggestions for further improvements and for further research.

But first, let's take a look at some anecdotal evidence that seems to show that there is indeed room for such applications. Consider, for example, the graph in Figure 3, a version of which appeared in a recent Scientific American article about Deep Thought. The apparent straight line relationship between depth of look-ahead and chess-playing strength (measured according to the chess rating system of the U.S. Chess Federation) is quite impressive. It is the basis for the prediction that an addition 2 full moves of look-ahead will enable Deep Thought to beat the human World Champion. (In chess and other 2-person games, 1 full move is completed when each player moves once. To avoid confusion, the plays made by each player during a single full move are called half-moves, or "plies". Depth of look-ahead is measured in plies.)

However, a closer look at the data used in the graph throws a different light on the matter. The first four points on the graph represent the play of a

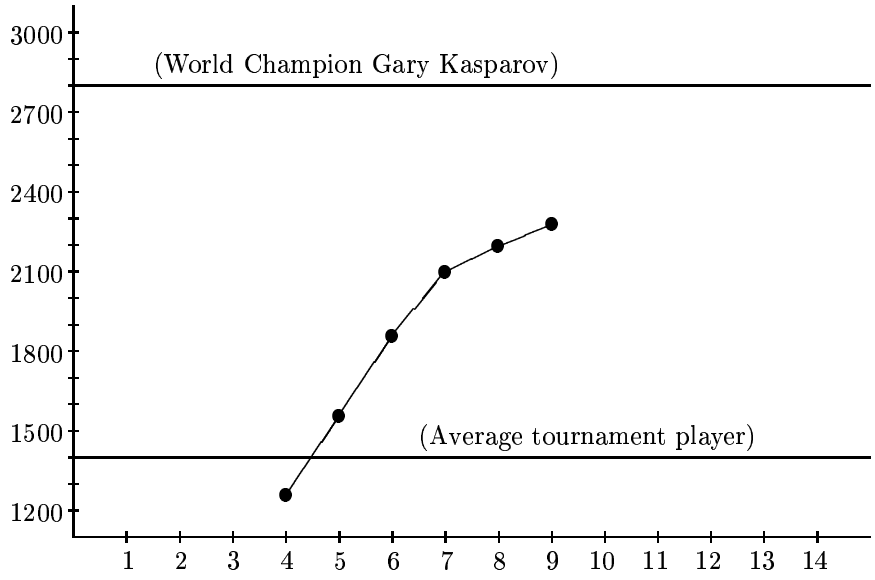

Figure 2: Belle plays against itself at different depths

single program, called Belle, from the early 1980s. The creators of Belle ran an experiment in which it played against itself, using different depths of look-ahead, from 4 plies to 9 plies. A large number of games were played, and based on the outcomes, the strengths of the program at different depths were estimated, resulting in the graph shown in Figure 4. Note the leveling off occurring at 8 and 9 plies. This leveling off was hidden in the graph in Figure 3, because the fifth and sixth points in that graph represent the play of two other chess programs, namely Hitech at 9 plies and Deep Thought at 10 plies. The graph in Figure 3 is essentially comparing apples and oranges. Both Hitech and Deep Thought use the singular extension modification of the Shannon algorithm, while Belle did not. Advances have also been made since the early 1980s in the development of algorithms for guessing the values of chess positions. Can it be that the Deep Thought team, who wrote the Scientific American article, is being a little too optimistic about the value of deeper look-ahead? If, as we suspect, Deep Thought's improvement curve levels off when they have reached the ultimate in computing speed, where will they go for improvements?

Here are some further tidbits:

1. The opening position in a chess game is completely symmetric. As a result, the positions after the starting position become less symmetric, and therefore generally more difficult to assess. Here is exactly the situation where the hypothesis of non-decreasing error probabilities in the evaluations is

most justified. And experience shows that look-ahead is practically useless for computers in the opening. All of the strong programs use "opening books", which is to say that as far into the game as possible (typically the first 10 moves or so), they follow standard lines that have been worked out for them by human players.

2. Although Deep Thought was the clear winner of the last World Computer Chess Championship, its supremacy was not so clear in the recent North American Chess Computer Championship, where it tied for first with Mephisto, a commercially available chess computer that is based on the same 68030 computer processor that is used by many of today's micro-computers (Deep Thought was awarded first prize on the basis of tie-breaker points). Mephisto certainly does not have anywhere near the look-ahead computing power of Deep Thought. The most recent news is that Mephisto has been tied in a tournament by a program called Chess Machine, which runs on a card that plugs into a standard slot in an IBM PC, and costs about $750.

3. Deep Thought recently played in a tournament sponsored by IBM. The other participants were 6 German grandmasters and 1 master. Deep Though placed 7th, worse than is to be expected. Some people think that Deep Thought's strength lies partially in its style of play, which is somewhat different than that of humans. Now that humans have become more accustomed to the strengths and weaknesses of the computer, they are learning to adapt, and the relative strength of the computer is going down.

4. One of us (Larry) has a chess program, called Chessmaster 2100, which definitely plays *worse* at 8 plies than it does at 6 or 7 plies.

5. Computers are bad at following consistent plans in their play, especially when they search quite deeply. On the other hand, there is a well-known game played between two computers during a tournament in 1975 in which one of them seemed to following a very consistent plan through approximately 20 moves. The interesting thing is that this program never looked more than one ply ahead.

6. When a computer is given more time to look ahead, say two or three days, its play, relative to humans given the same amount of time, gets considerably weaker. For example, with an average of three days per move, Deep Thought plays weaker than grandmaster level. Conversely, in speed chess, with an average of 5 or 10 seconds per move, Mephisto is at least equal to the World Speed Chess Champion, Mikhail Tal. There may be interesting implications here for the ways in which look-ahead helps the computer and those in which it does not.

It seems to us that the emphasis on calculating speed in computer chess has made many computer scientists blind to the some of the clues provided by incidents like those just given.

8 Where do we go from here?

We have derived a set of general principles for optimal decision-making in complex environments. As a concrete application we investigated the game of chess, and we have tried to show that our principles explain all of the major improvements in computer chess programs (aside from those connected with computing speed). We further suggest that they roughly correspond to principles used by good human players.

We also believe that the practical application of our principles to chess play has not yet been exhausted. The clear sight principle and the relevance principle have indeed been partly embodied in the strongest programs. But the stability principle is used only in a very attenuated form, as a reason for further look-ahead when the material is unstable. The application could be enormously enhanced by looking ahead after any position whose positional evaluation is unstable along the path leading up to it. The principle could be applied even without further look-ahead, by using the guessed values of the predecessors of a position to alter our guessed value for the position itself. The proximity principle guaratees that unless the positions are getting easier to evaluate, early evaluations are more reliable anyway. The family principle suggests that we should let our evaluation of a position depend somewhat on the guesses we make for its neighbors. The precise mathematical form of this will depend on the kind of ignorance that is present. Furthermore, implementation of a procedure based on the family principle is probably less compatible with the Shannon algorithm than any of the other suggestions we have made. But we believe that after much research, there may be something of practical value here as well.

As far as the world outside of chess goes, it seems that people have been at least vaguely aware of our principles all along. But by giving them a somewhat more precise formulation, we have illustrated how mathematics can refine and clarify, and even improve, the way in which we make day to day decisions.