

ON THE CONSTRUCTION OF EVALUATION FUNCTIONS FOR LARGE DOMAINS *

Hans Berliner
Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pa. 15213

ABSTRACT

We present the SNAC method of encoding knowledge in a polynomial function. The most common use for such a function is to evaluate competing alternatives in a problem solving situation. The SNAC method was discovered during research on a program that plays backgammon. It has resulted in highly consistent, skilled performance, and has made adding new knowledge very easy. This has resulted in large performance increments in the backgammon program.

We show how to create sensitive evaluation functions, and how to avoid stability problems in non-linear functions. We also demonstrate two effects, not previously found in the literature: the *suicide construction*, and the *blemish effect*.

I. Introduction

In problem solving there exists a set of states of the domain, a sub-set of these that are terminal states (corresponding to the achievement of some goal of the domain), and a set of operators that transform one state into another. In a given domain, it may be possible to apply any one of dozens of operators to a given state. Further, the current state may require hundreds of consecutive operator applications in order to arrive at a terminal state. Since such a search grows exponentially with depth, generally no path to a terminal state from a given starting state will be found within a reasonable effort. This is especially true in large domains ($>10^{14}$ states).

When it is not reasonable to expect the solving process to search toward a terminal state, we must have recourse to knowledge to lead the way to a goal. The knowledge is in the form of "properties" or "features" that can be used to describe any state of the domain. Each such feature can take on a range of values and thus defines a dimension in a hyper-space of features. A polynomial that is the sum of various functions on these features is used to assign values to nodes and thus locate them in the hyper-space. These values are then used to order the nodes with respect to their goodness (closeness to goals of the domain). The AI literature does not contain much information about how to construct evaluation functions; only the work of Samuel[4, 5] attempts to shed light on how the construction characteristics of the function (rather than the content) bear on the performance of the program using the function.

We have been developing a backgammon program since 1974 [2]. During this time, we ran into many

construction problems dealing with the lack of context provided by a linear polynomial (as Samuel had indicated), and problems of occasional erratic behavior when certain types of non-linearity are introduced. Recently, we have developed a method which we call SNAC (for Smoothness, Non-linearity, and Application Coefficients) which appears to have remedied all previous problems. We present details of the method below.

II. Non-linearity

Linear functions have difficulty in accurately approximating complex behavior over a large range. Consider a simple price relationship between oranges and apples. If we assert that an orange is twice as valuable as an apple, we may be stating something that is correct *on average*. However, this type of advice would not be satisfactory when there is a great orange glut and a shortage of apples. Thus, a linear function seldom is *sensitive* enough over a large range; e.g. it fails to take into account the relative supply of oranges and apples, and will thus prove to be too constricting at times.

However, a linear function is very well behaved. Arithmetically combining two variables, each of which could have a range of 0:50, produces a resulting range of 0:2500 when multiplication is used. The contribution of such a term to the evaluation could vary widely, causing *stability* problems for the value of the polynomial.

Another type of problem with non-linear functions is the following. Say, we have some advice to the system in the form of $S=I \cdot D$, where S is suffering, I is the intensity of pain, and D is the duration of pain. The object is to minimize the value of S (as there will be a minus sign in front of this term in the polynomial). This seems to be a well formed piece of advice. People, in general understand that the idea is to reduce both I and D . However, a program that is allowed to manipulate D , when faced with excruciating pain that is difficult to remove, may well recommend suicide. This usually does not qualify as a solution. However, such advice can be forthcoming even when some other term of the polynomial places a large premium on staying alive. We therefore term a relation, where there exists the potential to manipulate one of the variables in a generally undesirable direction, a *suicide construction*.

Non-linearity is desirable because of the increased sensitivity it provides, while care must be taken to control volatility and avoid the suicide construction.

*This work was sponsored by the Defense Advanced Research Projects Agency (DOD), Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551.

III. Smoothness

Any function on the set of features in our evaluation hyper-space will define a surface. Let us consider what can result from a lack of smoothness in the surface. If there is a ridge, a discontinuity, or a sudden step in the surface, then this is a place where the values on one side of such a blemish may be quantitatively very different from those on the other side. Thus, a very small change in the value of some feature could produce a substantial change in the value of the function. When the program has the ability to manipulate such a feature, it will frequently do so to its own detriment. Simply put, the program will act to hurry across such a blemish if it leads to a favorable evaluation, and wait as long as possible to cross it, if it leads to unfavorable consequences. This behavior resembles the horizon effect [1], although caused by a very different set of circumstances. We now name this effect the *blemish effect*. Because of the blemish effect, smoothness is absolutely essential for reliable performance.

The above may seem so obvious that the reader would feel that no one could possibly overlook such a thing. However, this is far from the case. There is very little published on the structure of evaluation polynomials that are used in game playing programs. One exception is the work of Samuel [4, 5]. Samuel investigates how to achieve non-linearity without creating an unstable function. His solution, in both cited works is to map a variable with a large range onto a small range. In the earlier work, he creates "binary connective terms" which are a reduction to a binary value of a range that was large to start. Clearly, this will cause the blemish effect in the vicinity where the value of the variable changes from 0 to 1. In the later work, large ranges are reduced to ranges of from 3 to 7 in order to fit more easily into a "signature table" of limited size. Again, the blemish effect will occur near the locations where the value changes occur. We conjecture that the reason that Samuel's program did not perform better after learning non-linear functions is that the blemish effect caused it to commit serious errors occasionally. We now are able to observe the blemish effect in the performance of older versions of our backgammon program.

Whenever the coefficient of a non-smooth term is under the control of the program, the blemish effect can occur. Consider a chess evaluation term that says "in an endgame the king should be centrally located, and at other times it should be located near the corners". Let us assume the king is presently in a corner, and the "endgame" is defined as being those positions where there is less than a certain amount of material on the board. The program may then avoid swaps in material in order to consider its king's position as "good" (non-endgame) even though the endgame is imminent. Here a step is created by the coefficient that defines

endgame, and this acts to the program's detriment. A correct definition of endgame would be a smooth function from early middle game to late endgame. In this way, the degree of endgameness increases with each swap of material.

IV. Application Coefficients

We have indicated how sensitivity can be achieved, and how the blemish effect can be avoided by the use of smoothness. However, there is a major problem in avoiding the creation of terms that are very volatile. Otherwise, the program may be trying to produce some extreme value in the volatile term, because it will outweigh the combined values of all the other terms of the polynomial. This problem stems from the fact that it is very difficult to anticipate the range of values that a term may take on over a large domain. This is especially true when functions continue to be changed during program development. Volatility can be avoided by constraining the values that one of the variables can take on. If this were done in a construction such as $S=I \cdot D$, then the volatility of the term would disappear. Yet the constrained variable would provide more sensitivity to context than a constant coefficient.

There are two ways of achieving this: 1) By fixing the value of the variable to be that in the original problem situation (frozen variable), i.e. not recomputing it for each new node, and 2) By choosing variables that vary very slowly (application coefficients). We have used both methods successfully.

Using the value of a frozen variable gives the problem solving process a global outlook, where all terms using this variable are viewed as they would have been in the original situation. This has the advantage of not letting small variations create too much of an effect, and suppressing volatility for large variations. It has the disadvantage of making the process insensitive to certain kinds of changes. This method is good for functions that require some discrimination to determine the degree to which they apply, but are not required to discriminate minimal changes. This method has been used previously for efficiency reasons, when the cost of recomputing the variable at each new node is high.

The other method is to use *application coefficients*. An application coefficient is a variable that tends to vary slowly with operator applications (moves) due to the nature of the domain. Thus for a set of nodes that are a few operator applications apart, the value of the variable will tend to be relatively constrained. This results in a coefficient that will provide sensitivity without volatility. We give examples of typical application coefficients below.

Our typical evaluation polynomial is of the form $V=A_1F_1+A_2F_2+\dots+A_nF_n$, where the A_i 's are application coefficients or frozen variables, and the

F_i 's are functions of features of the domain. We have found that while there are usually many dozens of useful F_i 's in a domain, on the order of six or fewer application coefficients appear to emerge. These are strongly related to ideas such as game phase, ease of winning, readiness for attack, etc.

In chess, typical application coefficients are: 1) What stage the game is in, as denoted by the amount of material on the board, and the degree to which the pieces are still in their original positions; 2) The degree to which the winning side is winning, indicated by $(W-V)/(W+V)$, where W is the score of the winning side and V that of the losing side; and 3) the ease of being able to win, which is a function of the number of pawns left for the winning side and the ease with which they may be exchanged, and the degree to which the position is blockaded. Similar application coefficients exist in backgammon.

Such application coefficients provide a program with a great deal of context for making decisions. Thus a program that understands the stage of the game in chess, as a function of amount of material on the board, will allow its king to gradually achieve a more central position as the amount of material diminishes. Further, the *suicide construction* can be avoided by using a frozen variable in place of one that can be varied adversely. In the example quoted earlier, the duration of life of the subject becomes frozen, and that value must be used in all functions that could otherwise be subject to the suicide construction.

V. Results

We have three methods of measuring the performance of our backgammon program: 1) Performance on a problem set in an intermediate level instruction book [3], 2) Games against other backgammon programs or earlier versions of our own program, 3) Performance against human opponents.

Our previous best version before using the SNAC method was called BKG 8.0. This program was the result of about 30 man-months of effort. The present version is BKG 9.7, the result of about 8 additional man-months of work. In tests on the problem book, BKG 8.0 achieved a score of 45% based on 74 problems that it could attempt. Without any pre-testing, BKG 9.7 achieved 66% on the full set of 77 problems.

Against the best commercially available backgammon micro-processor, BKG 8.0 achieved 56% of the points, while BKG 9.5 (considerably inferior to BKG 9.7) scored 78% of the points in a set of 100 games. BKG 9.7 is now much too good to test against the micro-processor. Our current tests pit BKG 8.0 vs. BKG 9.7, with BKG 9.7 scoring 64% of the points.

BKG 8.0 played in the Carnegie-Mellon University backgammon championships in spring of 1978 and lost its first two matches thus being eliminated. In May

1979, BKG 9.7 played in a tournament of intermediate players in Portola Valley, California and won its first two matches before losing to the ultimate winner of the tourney in the 3rd round. The competition in the California tournament was somewhat better than that in the earlier tourney.

BKG 8.0 has been available on PDP-10 machines for some time and has been regarded as a good game playing program, and by far the best backgammon program around. Yet, as the above results indicate, the SNAC method has resulted in a rather significant improvement in the program. In evaluating the above, the reader should bear in mind that in backgammon, chance plays a significant role. Professional backgammon players will tell you that a few percentage points difference in skill is all they need to have an opponent that they can win from consistently. A 60% edge is quite extreme.

The recent improvement of the program was made possible by the SNAC method that made it possible to: 1) Organize existing knowledge so that the functions are sensitive to local conditions (non-linearity) without being subject to significant volatility, 2) Avoid the blemish effect (which used to cause occasional serious errors), and 3) Add new smooth knowledge functions that contribute their part without creating opportunities for new blemish effect situations.

ACKNOWLEDGEMENTS

The author wishes to acknowledge the influence of discussions with David Slate on his understanding of the role of smoothness in evaluation. Suggestions by Allen Newell have greatly benefitted the organization of this paper.

REFERENCES

- [1] Berliner, H. J., "Some Necessary Conditions for a Master Chess Program", *Proceedings 3rd International Joint Conference on Artificial Intelligence*, pp. 77-85, August 1973.
- [2] Berliner, H. J., "Experiences in Evaluation with BKG - A Program that Plays Backgammon", *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, IJCAI Press, Computer Science Dept., Carnegie-Mellon Univ., Pittsburgh, Pa. 15213.
- [3] Holland, T., "Better Backgammon", Reiss Games, Inc., New York, 1974.
- [4] Samuel, A. L., "Some Studies in Machine Learning Using the Game of Checkers", *IBM Journal of Research and Development*, Vol. 3, No. 3, 1959, pp. 210-229.
- [5] Samuel, A. L., "Some Studies in Machine Learning Using the Game of Checkers, II - Recent Progress", *IBM Journal of Research and Development*, Nov. 1967, pp. 601-617.