CMU-CS-80-113

Some Observations on Problem Solving[1]

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

24 April 1980

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

[1]Also presented at the CSCSI Conference, Victoria, B.C., 1980

Abstract

In this paper, we make the case that problem solving based on immutable goals, selection of operators to bring these goals closer, and discrete logic to both select operators and evaluate outcomes is effective only in very small domains: Instead, methods using search and continuous evaluation functions do well in any sized domain as long as the evaluation functions have a certain structure. Discrete reasoning systems manipulate discrete valued entities. However, serious errors can occur when the value of a continuous variable is discretized, especially if this is done before the value is needed for final output. Because of the need to prevent this source of large errors during the evolution of problem solvers that must survive while they master their domain, we infer that the generality-specificity dimension of problem solving runs from ends-oriented to means-oriented, and from continuous to discrete. Finally, we conjecture about the structure of computing machinery for problem solvers that must evolve from general to specific.

1 Introduction

Means-oriented problem solving requires a method of selecting a sequence of operators that may lead to a goal. This involves knowing the potential of available operators, and possibly the closeness of non-goal states to goal states. It is widely held that this type of activity is a major part of human problem solving, and that the selection of suitable operators is achieved using rule-based or pattern-based knowledge.

It is also possible to have simply an ends-oriented problem solving approach. This involves generating a set of alternatives (by generate and test procedures such as searches) and then evaluating the leaves of the test set to find the best path to pursue. Usually one attempts to generate the largest set of alternatives that can be processed with the resources available. This gives a brute-force aspect to the method; it attempts to discover the best path by investigating the maximum number of alternative paths, rather than by attempting to apply knowledge to guide the investigation into those areas that appear most promising.

The two methods can best be distinguished in that the means-oriented method must have knowledge of the potential of operators so that it can choose wisely among the available ones. This has lead (in GPS [12]) to the "table of differences" that gives a clue as to which operator is most likely to produce maximum progress. To date, the generation of data to guide the selection of operators has been done almost exclusively by humans (programmers). Thus, it appears unlikely that data of this type can be generated mechanically for domains of (say) 10^{12} states, yet humans are able to make good decisions in such large domains. Means-oriented methods and ends-oriented methods both will require knowledge of how good a current state is; in the first instance to decide which branch to pursue (as being closest to the goal) and in both instances in order to identify the goodness of leaf nodes that are reached.

Evaluation can be thought of as being done by a function that assigns a scalar value to a state, thus making it possible to compare its goodness to that of another state. In small domains this process may be little more than the identification of goal-states, or the identification of states that have some salient feature that must elevate it above any state not having such a feature. This dominance type of reasoning is usually quite adequate in small domains, thus giving evaluation a discrete character; yes/no or a sorting into a small number of equivalence classes. However, in larger domains the full power of a polynomial function, with its ability to trade-off the value of one term of the polynomial against the value of another, may be required. At its full potency, the polynomial can take on a (more or less) continuous set of values, and should (if totally effective) be able to correctly order all states in the domain with respect to nearness to goal-states. In practice, such effectiveness is not achievable in interesting domains, so it is desirable that the ordering, if not totally effective, at least not produce large decision errors (such as sending the solver off in the opposite direction, or leaving it stranded on a hill-top). We shall show that the structure of the evaluation polynomial has a great deal to do with its effectiveness.

The selecting of effective operators at a node, apart from being governed by decision rules, could also be done by evaluating the state that each available operator produces and selecting the best. It should be noted that, while there is a sequential flavor to a reasoning process that moves from one premise to the next to achieve its aims, the evaluation polynomial is essentially a parallel construction, with each term independent of all others. Thus reasoning, discrete, and sequential appear to go together, while judgement (evaluation), continuous, and parallel go together also.

2 Two Examples

Consider the problem of mating with a King and a Rook versus King (KRK) at chess which is a medium size problem with a state-space of about 10^5. All instruction books for humans will indicate that the correct *procedure* (thus *means-oriented*) is to use the rook to build a fence around the black king (see lower left of Figure 1), and gradually constrict the fence until the mate is there. It is rather interesting that this advice suffices for humans. Clearly, they have enough structure to interpret these instructions and produce the correct effect. I have never heard any beginner complain about the adequacy of these instructions, although I remember being temporarily at a loss the first time I tried the exercise because fence constriction, taken to the ultimate, results in stalemate. Thus, a last-minute change of strategy is required.

However, when one attempts to implement the same instructions for a computer program, some very vexing problems occur. They deal with exactly how to go about constricting the fence, since at times no constricting move is possible without letting the opposing king out (lower right of Figure 1). Also, there is the problem of not intersecting the fence by placing one's own king on the fence line and thus allowing the opposing king to cross (for instance if K-B3 in lower left, then K-R6 and black's king has escaped). These problems are so prolific that one author [21] has complained that if such a simple problem be that difficult to program, then chess itself must be impossible. If one wishes to program chess using only the means-oriented (rule based) approach, then Zuidema is probably right in that no set of humans will be able to write all the required rules.

Actually, the ends-oriented approach for doing KRK had already succeeded several years earlier, although with a great deal of structured knowledge together with very small searches [7]. But the real power of evaluation functions when combined with search was demonstrated with great simplicity and elegance as follows [1]: Consider the upper right of Figure 1. Here a gradient exists from the center to the corner. Let the major term in the evaluation function be "how decentralized the black king is". Since the same evaluation function is used by both sides, Black will resist being decentralized. Thus,

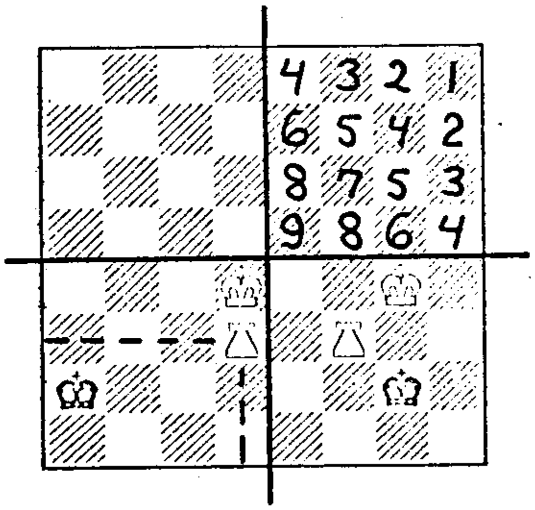

Figure 1: King and Rook vs. Rook

it is sufficient for White to choose the sequence of moves that decentralizes Black the most until the task is completed. This measure would be sufficient for terminal nodes of a 9-ply search. For shallower searches, a subsidary term which values keeping White's king near the Black one, and a still less significant term that values keeping White's rook near the White king, allow the mate to be performed by a 3-ply search. This search need only use the evaluation function, know the value of material (so as not to lose the rook) and the rules of chess, so as to mate and not stalemate. The resulting program could be written and debugged by a second year undergraduate in about 5 or 6 hours.

Finally, the same problem is capable of ultimate solution. A data base can be created for all possible positions of KRK. Then working backwards from those positions that are mates, one can assign a number to all other positions. That number represents the minimum number of moves that are required to produce a mate. Positions that are draws (stalemate or lost rook), will be assigned a value of infinity. Using this data base it will always be possible to produce the shortest mate in any situation, by merely generating all legal moves and selecting the one that moves to the position of lowest value. This task has actually been performed by a number of researchers, first by M. R. B. Clarke [6], and has produced the first (though trivial) computer-produced chess knowledge. Clarke showed that it is possible to mate in at most 16 moves from the most difficult position, whereas it always had been thought to require 17.

That there is a trade-off between the amount of knowledge and the amount of search is very clearly shown above. A data base of 10^5 suffices to produce optimal play as does a search to 31 ply (intractable unless a dynamic programming approach is used to identify identical nodes and turn the tree into a graph). The most desirable solution though, for problems of this level of complexity is a heuristic one, in which only a satisficing, rather than an optimal solution is obtained. A shallow, ends-oriented search serves well here. A simple construction (the gradient) puts a key measure onto the evaluation process. At certain depths of search this suffices for successful performance of the task. At shallower depths of search some small amounts of additional knowledge are required. To select an adequate move, without search or a complete data base, requires large amounts of carefully structured knowledge. The optimum trade off between search and knowledge in the above example appears to be in the area of a 5-ply search (1 second duration) with a 2-term polynomial. It is interesting to note that one can characterize this problem (as is possible of all mates of a lone king) as simply a decentralization problem. This characterization is simple, precise and very useful. Yet humans characterize it differently, possibly because of the effect of culture on the primitives available for perceiving the problem.

The above is a typical medium sized problem as judged by the size of its state space. Let us now examine a small problem: i.e. the Monkeys & Bananas (M&B) problem [10] with at most a couple of hundred states. As usually stated, the M&B problem has a few operators: move monkey (X), move box (X), climb box, and reach (X), each of which may have some applicable pre-condition and effect some tranformation on the state. Then depending on what your favorite paradigm is, you can solve the problem using GPS, predicate calculus, etc. However, these formulations all require a number of restrictions on the real problem (to make it tractable), together with machinery specifically designed to make the solution proceed in the necessary direction. Even then, solving the problem produces a formidable challenge to the solving system.

Let us now pose M&B as a search problem. We can complicate the problem to the point where many techniques would find it next to impossible to solve by increasing the number and scope of the operators. The operator for moving the box produces movements of exactly one foot in one of the 8 compass directions. The same is true for moving the monkey. Also, allow the monkey to climb down from the box as well as climbing up. In addition, allow the monkey to throw the box against the cage (making it unclimbable), to tear a lath off it (making it unclimbable), and to reach in each of the eight compass directions. A goal state is one in which the monkey touches the bananas. In this problem statement, quite a few operators may be applicable at any one time. The state description specifies the 3-dimensional location of monkey, box, and bananas, and the condition of the box.

Now, as a search problem, we would be willing to allow (say) a 4-ply search and evaluation of terminal nodes. Our evaluation function will value primarily the closeness of the monkey to the box, secondarily the closeness of the box to the floor and to the bananas, and thirdly the closeness of the monkey's hands to the bananas. It is rather clear that such a paradigm will succeed with extreme efficiency and ease in discovering a very good solution. The monkey will approach the box to satisfy the primary term, push it under the bananas to satisfy the secondary term, and climb the box and reach for the bananas to satisfy the tertiary term. I realize, of course, that M&B is only used as a pedagogical tool to demonstrate problem solving paradigms. In fact, that is exactly what I am using it for.

3 Problem Solving Performance in Large Domains

Let us now examine the experience of various problem solving methods in large domains. There are a few efforts to apply means-oriented methods to checkers [15, 16]; and chess [2, 3, 13, 20]. Samuel's program was a marvel for its time, but has more recently been soundly trounced by a full-width searching (ends-oriented) program with much less knowledge [17]. The Baylor-Simon MATER program worked only in very restricted situations. Thus this was more a case of exposing the power of a useful move selection heuristic (the move that allows the fewest replies) than an attempt to cover the domain of mating combinations, not to speak of the realm of combinations in chess. The Berliner program did reasonably well at doing chess combinations, but was inept when no combinations were to be found or when its knowledge was not quite up to finding them. Pitrat introduced the notion of plans to select moves that, deeper in the search, were compatible with an initial goal. He also introduced methods for patching a plan when it ran into difficulties, but his approach relied heavily on brute force searching and very simple plans. The Wilkins program considerably improved on the last two efforts above with knowledge comparable to the Berliner program and plans of great sophistication that effectively controlled the plausible actions deeper in the search. This program was able to attain very high solution rates on chess combination positions, once its knowledge base had been built up to an appropriate level.

In all these efforts one paramount fact has intruded itself upon us: a very small change in the problem environment can make a large difference in what is the correct action, and what, therefore, the problem solver may or may not be able to do. Thus, the way means-oriented programs are improved is by the writing of ever more exception rules. In the end, the search is supposed to catch those exceptions that were not explicitly programmed in.

However, there is a great deal more to chess than executing combinations. This has been shown dramatically by the Northwestern University chess group, whose program CHESS 3.0 (and up) has been the perennial winner of almost all important computer chess events. While means-oriented programs wallow in trying to solve relatively small sub-domains of chess, CHESS 4.6 (and up) [19] has in the last 3 years moved up to challenge good human players, some of whom it has defeated. This program relies heavily on a full-width search with iterative deepening[2] which is made more efficient by the installation of a hash table that:

1. Guides the search into the promising sub-trees discovered by the previous iteration, and

2. Terminates the search at positions that are identical to those already searched in the current iteration.

Ken Thompson of Bell Telephone Laboratories has shown that organizing the above method for parallel computation and using special purpose hardware produces further significant speed-ups. Thus, in chess and checkers the hand-writing is clearly on the wall. Brute force searching with relatively little knowledge will soon be able to beat almost all the players in the world. Whether knowledge oriented programs will be required for the World Champion level in chess is a moot point; however, in this writer's opinion the programs will play with so much greater consistency, that with just small amounts of additional knowledge, they will perservere to the World title.

The situation is quite different in GO, however, where the magnitude of the task would appear to make the use of ends-oriented methods quite difficult because of the large number of alternatives. In fact, no one has tried such methods, and the best program to date [14] makes heavy use of specially designed GO constructs to guide its play. However, its play in this most difficult of games is far from being able to give even intermediate players a decent game.

At backgammon, a program that uses no search but relies solely on evaluation of all possible moves emanating from the current position [5], has recently defeated the reigning World Champion by the lop-sided score of 7-1; a result that must be somewhat discounted due to the stochastic nature of the game. Again, this was the result of an ends-oriented approach, that we describe to some degree in the next section.

Apart from games, experience with speech understanding systems has shown that discrete reasoning systems do not do as well as a brute force searching system using a fraction of the knowledge [8].

It should be clear from the above that, if at all possible, ends-oriented methods should be employed. They are easier to implement, succeed better, and may be the only realistic way in certain domains. Further, the methods have been shown to be applicable to many domains that were thought to be too complex to ever be subjugated by brute-force searching.

[2]A full-width search at each node looks at all alternatives in the tree that have not been logically eliminated by alpha-beta pruning. Iterative deepening involves doing first an N ply search, then an N + 1 ply, etc., until the allocated time resources have been expended.

We hope the above has made clear our first thesis: that means-oriented problem solving has proven robust only in small domains. Some shallow searching plus some simple terminal evaluation has in an overwhelming number of cases been shown to be superior to the business of solving problems by operator selection and reasoning. This is definitely true for machine oriented problem solving, and the evidence is so strong that one wonders how living organisms get along without using this, if, in fact, they do.

4 The Structure of Evaluation Functions

The principal usefulness of evaluation functions is for guiding a problem solving process that is unlikely to reach a domain defined goal (i.e. a complete solution) during its present probe, and must thus settle for a step in what is considered to be the right direction toward a solution. A number of reasons now appear to favor using evaluation functions where possible over the reasoning methods that have been considered fundamental in the past:

1. It is possible to simultaneously pursue several goals with this method. Each term (or a small set of terms) in the polynomial could be considered a possible sub-goal to be pursued. Thus the degree to which each has been achieved may be ascertained. This is extremely difficult to do under reasoning paradigms, as one goal will be paramount in such procedures. Such a goal, in turn, determines the valid sub-goals, and all others are ignored. Such methods will prefer success at a primary goal to success at a number of secondary goals that may, in fact, be superior. Interactions between sub-goals may be taken care of in the evaluation function by the use of non-linear terms.

2. Two major problems with evaluation functions have been that they were thought to be lacking in context sensitivity, and it was possible for a hill-climbing process using such evaluation functions to get stuck on a sub-optimal hill and not be able to get off. However, in the pursuit of goals and sub-goals, proper construction of the evaluation function will produce smooth transitions from one state to another, even if the first state represents a major goal that has just been achieved, and the focus must now shift to a new goal. Below, we demonstrate how to construct evaluation functions properly.

The essential DO's of constructing evaluation functions are embodied in my SNAC method that was used in the backgammon program that beat the World Champion last year [5]. SNAC stands for Smoothness, Non-linearity, and Application Coefficients.

Non-linearity is extremely important for expert performance. A constant coefficient can at best portray the *average* usefulness of the term associated with it. There will be times when this average value will be at considerable variance with what expert judgement will consider correct, and this is where systems using linear functions will fail. Non-linear functions can produce the necessary context by combining the action of several variables into one term. However, the key to using non-linear functions is smoothness. This is where Samuel made a serious methodological error when he found that his non-linear functions did not perform better than his earlier linear ones [16].

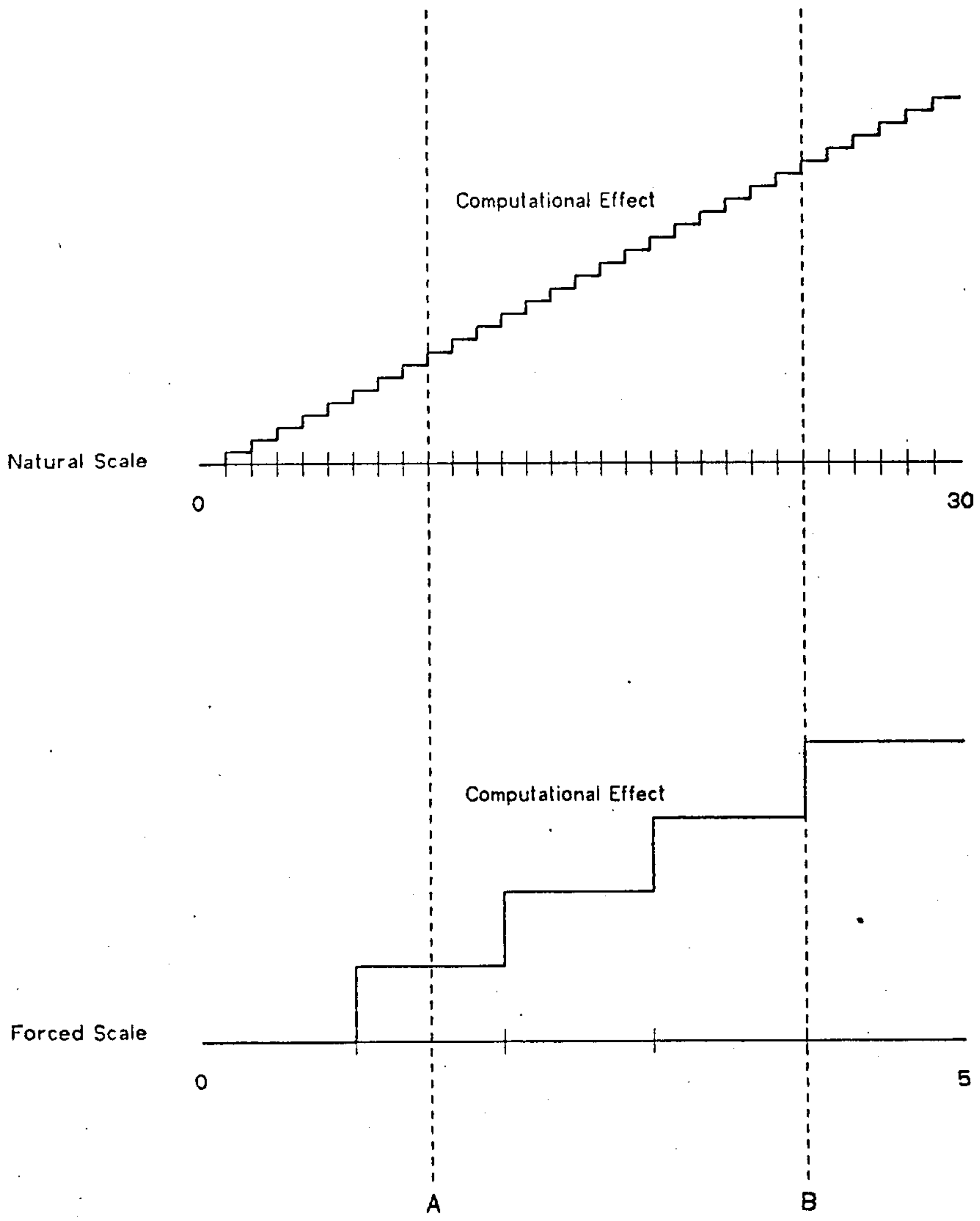

Figure 2: Effect of Smoothness

Smoothness relates to the rate of change of a function for adjacent parts of the domain. Samuel, for several of his variables, subdivided their natural range into a compressed range, so that the variable could only take on a few values and thus the Signature Table would be smaller. However, this was a fundamental error as can be seen in Figure 2. If a variable has a value near vertical line A, then in both the lower (large grain) and upper (smooth) situations, a small change in the value of the abscissa will produce only a small change in the value of the ordinate. However, near vertical line B the situation is quite different. Here, for the lower situation, a small change in the value of the abscissa can produce a very large change in the value of the ordinate. Such a construction will provide opportunities for a program to manipulate the value of the ordinate to an extent unwarranted by its actual utility, and this may cause the program to make serious errors.

This type of behavior can occur whenever there are sharp boundaries in the evaluation space. Assume a chess program has a different method for evaluating middle-game situations than it does for evaluating end-game situations. Experts agree that such disparate types of positions should be evaluated differently. Further, assume such a program has a middle-game position that it likes, but this position would receive a poor evaluation if seen as an end-game. If swapping material would cause the position to be evaluated as an end-game, then the program would go to great lengths to avoid swaps. This could well cause it to encounter severe and unnecessary problems in the play. The converse of this problem also occurs: the program hurries into the end-game because the center control situation is unfavorable in the middle game.

Smoothness in functions is the answer to this problem. There is a slow metamorphosis of middle-game to end-game, and during this phase the values of both phases must be recognized, although the middle-game is waning and the end-game waxing. Any attempt to draw a sharp line between these is doomed to failure because it will result in occasional unwarranted attempts to stay on one side of the boundary or cross it too quickly.

The key to accomplishing smooth transitions is Application Coefficients. An application coefficient is a variable that measures something global, yet varies very slowly in the current context. It multiplies a term in a polynomial, thus providing context about the importance of the term under current conditions. We have investigated a number of domains and found good application coefficients in all of them. Their character is that they measure some trend or change of phase. Because they vary slowly and smoothly, the program will not be trying to manipulate them over a significant range (as by deliberately staying in the middle-game because it has good control of the center, and this is not valuable in the end-game). For chess, material on the board is a good application coefficient, and this will produce a smooth metamorphosis between phases and there will be no boundaries near which catastrophes can occur.

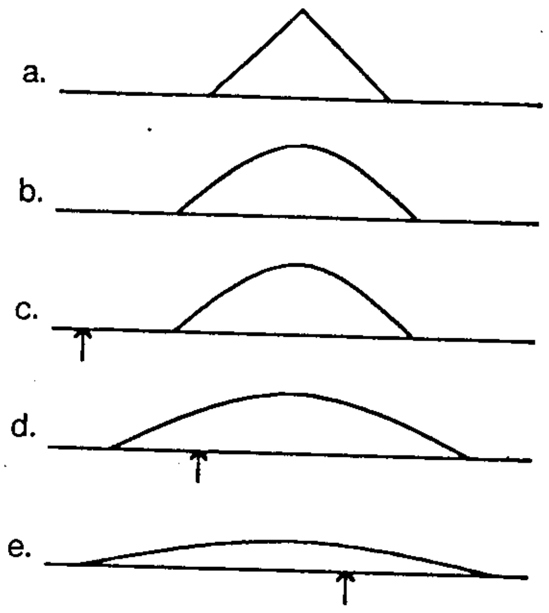

Figure 3: Effect of SNAC on Hill Shape

Application coefficients can also prevent the previously mentioned problem of a hill-climbing program getting stuck on a sub-optimal hill. Figure 3 shows the problem. With linear polynomial evaluation functions, the hills in the evaluation surface will have pointed peaks and this will make it quite likely to get stuck on such a hill (3a). With non-linear functions, the peak is less pronounced so that it may be easier to descend a once-climbed hill, if some other high ground is in view of the searching process (3b). However, with application coefficients it is possible to change the contour of the hill even as it is being climbed. This is shown in 3c through 3e; the arrow showing the proximity of the current state to the hill. At a distance, the hill looks as in 3c. This allows the height of the hill to be compared to that of other landscape features that may be achievable. However, as the hill is climbed, it begins to flatten (3d), making the achievement of the summit less desirable (since we are almost there anyway), and resulting in the program looking for the next set of goals before even fully achieving the current set. As it sets out for the next goal, the hill flattens still further (3e). This flattening is controlled by application coefficients that detect the degree of progress in achieving the goal, and reduce its importance as it comes closer to being achieved. This paradigm recalls the situation in which a football player begins to run with the ball before he has caught it. The point is: if the controlling human processes solved problems sequentially rather than in parallel, such behavior would be unlikely to occur.

Thus application coefficients can change the program's view of what it should be doing, even as it is doing it. For instance, in my backgammon program one of the major goals is to blockade some of the opponent's men. However, if such a maneuver succeeds, the blockade must eventually be lifted in order to bring one's own men into the homeboard to proceed with the win. In order to have the program understand the desirability of the blockading goal, there are application coefficients that gauge the overall situation and raise or lower the desirability of blockading based upon global considerations. Such constructions can be tuned to give truly amazing performance: perceiving (as it appears) when blockading is appropriate and when it is not.

That is not to say that a domain should never be partitioned into sub-domains for evaluation purposes. Sometimes, that is the only sensible thing to do, but it must be done judiciously. For instance, in backgammon there will come a time in the game where the two sides are no longer in contact and both are racing for home with no further impeding of each other. In such a phase it is senseless to consider such features as blockading potential, board control, etc. Since the coefficients of such terms would be zero during the running game, evaluations generated for such running game situations will differ considerably in magnitude from those generated for competing non-running game positions. Yet it may be necessary to choose between such positions. What is needed in such a situation is a common measure along which both types of positions may be evaluated. This is attained by computing the win probability for each type of position. The best of each position type can first be chosen using the evaluation function appropriate to each sub-domain. Then the best position in each sub-domain can be compared to select the best over-all course of action. New sub-domains should only be created when there is a clear basis for doing this, as the number of potential comparisons grows with the square of the number of sub-domains, and each comparison is a potential source of decision error.

The use of SNAC functions in my backgammon program turned it from a mediocre competitor to an expert level program, with only a small influx in additional backgammon information, as is documented in [5].

5 Why Discrete Systems Fail in Large Domains

To here, I have tried to make two points:

1. That the combination of search and good evaluation functions produces a very fine problem solver for many domains, and

2. That the evaluation functions must be carefully constructed and are more powerful when non-linear.

Now it is time to take cognizance of the rather apparent degradation that takes place when problem solvers relying on boolean decision making are applied to large domains. The degradation takes place in the process of goal selection, the process of operator selection, and the process of evaluating closeness to goals in non-terminal nodes of the domain. From the variety of evidence presented, we consider it apparent that this is not the fault of researchers or lack of effort, but rather of the nature of the problem and the method. It appears that the idea of applying boolean decision rules to a large domain just will not work unless the domain is quite regular (as is mechanics, where a few principles have ultimately been shown to account for all macro behavior). Thus, as the exponential explosion prevents any attempt to produce satisfactory decision functions based on predicates (too many predicates required), any attempt to subdivide the domain without true basis in fact succumbs to the problems we have described in Section IV.

Let us try to determine the reasons why it is so difficult to improve a discrete problem solver. The effectiveness of a problem solver is measured by the nearness of the system proposed solution to the best or an adequate solution. If an algorithm for a particular domain is not known, then it is likely that effectiveness will be achieved gradually through increases in sensitivity to what a correct solution is. By sensitivity, we mean the number of states in the domain that are now ordered correctly, ignoring how far off misordered states are.

Given that the effectiveness of a problem solver is to be improved, sensitivity can be increased by having two states, that formerly had the same value, no longer have the same value. If two such states have similar state descriptions, it is possible to think of them as being somewhat adjacent in some mapping of the domain onto a multi-dimensional surface. To produce the new sensitivity, it is possible to place a partition between the two states so that states on each side of the partition will now be treated differently. However, this will result in many other adjacent states now being on one side or the other of the new partition, thus possibly altering their treatment too.

As earlier sections of this paper have shown, there is a definite risk associated with increasing the sensitivity of a discrete problem solver. The risk stems from the fact that introducing a partition, while it may improve the sensitivity of the problem solver, may also result in radically misordering certain states. Partitioning will only work properly if:

1. There really is a discrete difference between identifiable sets of states in this part of the domain, and

2. The partition is drawn absolutely correctly so as to not have any state on the wrong side of the partition.

Apart from partitioning domains, increases in sensitivity can also be achieved by creating a smooth gradient between the two states that are now to be treated differently. This will affect the values of other adjacent states, but not in such a severe manner as to cause misclassifications.

A critical observation here is that drawing a partition, irrevocably fixes the value of some variable at some discrete interval (as in Figure 2, lower scale). If such a variable is quasi-continuous, and if its value is to be used later for other computations, then it is certainly preferable to postpone the discretizing process as long as possible and retain its value in quasi-continuous form.

Among large systems, MYCIN-like systems are considered to exercise their expertise very well. These systems apparently avoid the partitioning problem in large domains by the use of probabilistic indicators [18]. Because of this, they can hardly be considered to reason in the boolean manner, but rather one gets the flavor of evaluation with summation of likelihoods.

6 Models and Sensitivity

The effectiveness of a problem solver depends on how well its domain is being modelled. Most domains can be modelled at many levels of detail. Consider that the morning weather forecast predicts a 40% chance of showers, when it could conceivably produce a cumulative precipitation curve over time for that day for each acre in the metropolitan area. If the domain is discrete and the model also, then a correctly formulated discrete model can be very effective. This is the case in most small domains and in domains that are "regularized", e.g. the game of NIM for which a simple rule can find the winning move even though the size of the domain is infinite. If the nature of the domain is not completely understood, opting for a quasi-continuous model appears preferable. This applies both to operator selection in a problem solver that applies knowledge to this process (because misordering operators can also have a very deleterious effect), and to evaluation.

Based on the action requirements and the accuracy and availability of input data, a model is chosen. It is desired to have that model function near the top of its effectiveness. When a given model does not perform near that level, then either the input data are insufficient, or the model is insensitive to certain things. The selection and improvement of a model appears to governed by the following principles:

1. For each model of each domain there is an optimum sensitivity. If the model utilizes a greater degree of sensitivity, it wastes computational resourses; if it uses lesser sensitivity, it will fail to "understand" or react to certain things. However,

2. Each increase in sensitivity in the problem solver is accompanied by an increase in risk of incorrect interpretation or action.

Consider the chess middle-game, end-game situation mentioned in Section IV. In a particular implementation, a program may consider that trying to control the center in the end-game is important, even though it is really not. This would produce occasional ordering errors because the program would value center control in the end-game more than is warranted by reality. Thus, it would occasionally fail to achieve a more worthwhile goal.

Now, assume the space is partitioned so that middle-game and end-game are no longer on the same side of the partition, and control of the center is valued only in the middle-game. This will result in better ordering of most end-game situations, but will occasionally cause serious problems akin to myopia when transitions from middle-game to end-game are involved. This is the risk involved, and as we have shown earlier, it can produce serious problems that would render the value of the increased sensitivity questionable.

Another way of looking at the problem is the following: Assume a system is capable of only two responses and a partition in the domain determines which response is given. The naive probability of response error is 0.5. However, assume an ordering of the states of the domain exists such that all states above a certain state in the ordering are on one side of the partition and all the remaining states are on the other side. Under such conditions, errors are much more likely to be made in the vicinity of the partition than elsewhere.

If a variable is to be used to produce a *final* boolean decision, then there is no difference between using a partition and using some distance function of the place in the ordering to produce the answer. However, if this is an *intermediate* result that may later be combined with other data, then there is a great deal to be gained by retaining some fuzzy representation of the result; i.e. the distance from the partition. Thus, it is frequently more useful to know that an event occurred at sunset, than that it occurred during daytime. Consider the importance of distance from high noon when evaluating the ability of an observer to see an event accurately. When it may be important to carry forward some of the properties of the original measurement, a quasi-continuous measure serves better. In such cases, where a gradient measures the property in question, the likelihood of error would be equally distributed throughout the domain. Since under such conditions, general remedies exist for reducing the error in any state, such a paradigm would seem preferable, when the value may be interim or when partitioning cannot be justified by the intrinsic properites of the domain.

7 The Evolution of Problem Solving Systems

There is no doubt that a highly discretized problem solving structure is the most effective one possible when such a model is applicable and the data it requires are available. After all, that is what science is all about. However, if a model produces some errorful responses then care must be taken in achieving discretization. In a sequential problem solver, a number of small errors is preferable to one large error. Research in game playing programs has shown again and again that such a system is no better than its weakest link. Further, the very ability to discriminate the condition of small error as against no error at all, is the hallmark of the expert. Each presently surviving organism considers itself to have adapted adequately. However, an expert organism of a particular species may be able to distinguish errorful acts in a somewhat inferior (but currently surviving and self-confident) specimen of its species. This, again, supports the view that small errors are tolerable, and are done away with gradually over time.

Thus, for large domains (and in real life almost everything is large) problem solvers must first and foremost be able to produce reasonable decisions (ones that are not too far off the mark). To do this, fuzzy methods are much more satisfactory than those that reason. Because highly discretized problem solving is so difficult to achieve, it is almost certainly preceded by other less exact decision methods in the ontogeny of any evolving problem solver.

Thus, it would seem that starting with smooth, continuous functions and gradually discretizing them would be a good strategy for achieving increased sensitivity. As increased sensitivity is achieved over time, most of the previously effective problem solver must still be in place. Thus, there will be a mixed bag of problem solving techniques, ranging from the use of continuous functions to discrete logic. In such an environment, it appears extremely likely that many intermediate variables will retain their original fuzzy character because higher level constructs are presently made from them. These notions would apply equally well to animate and inanimate problem solving systems.

Assuming the above ideas are valid, there must be a way for the problem solving systems of living organisms to evolve in this direction, both during the life of the organism and the life of the species. One possible solution to this problem is the variable coefficient. Such coefficients, as they vary between 0.0 and 1.0, have several known uses:

1. As a characteristic function in fuzzy set theory, indicating to what extent the element to which it applies is a member of the set.

2. As an application coefficient in SNAC that controls applicability of a concept.

3. For controlling truth value in certain belief systems.

When such a value has gravitated as close to an extremal value as can be detected by the system, then we no longer have smooth variation between the limits, but a boolean entity. We conjecture that this paradigm accounts for the behavior of Piaget's pre-conservation children, where the physical extent of a set of objects is taken to be the best criterion of the *amount* of the set, until it is learned that conservation (when applicable) dominates "extent".

Thus, an essential element of any evolving problem solver would appear to be computing elements capable of graded response.[3] Beyond that, we do not want to propose here that human problem

[3]We are well aware of previous work in the field of perceptrons and the demonstration of the limitations of *linear* perceptrons in [11]. However, we are here proposing a special class of *non linear* perceptron.

solvers use full-width shallow searches and evaluation procedures as those that have been so successful in computer programs. However, we do consider it likely that processes based on constraint satisfaction (as first implemented in Waltz's vision system [9]) or tightly controlled knowledge directed searches (as in the B* tree search algorithm, [4]) are developed to play the role that brute-force searching does in the previously references programs. Both methods could be used to screen out obvious misfits in the solution process, in the first case through a low level combinatorial analysis and in the second case through estimation of the limits of usefulness of each alternative. Evaluation would be done using SNAC-like methods at each level of the solution process.

Finally, let me briefly address an issue that may be brought up by some. The theory of computation decrees that any continuous system can be simulated to any desired degree of fidelity by a Von Neumann machine. That is not the issue here. The issue is one of complexity. Certain computing elements perform certain tasks more efficiently than others, and in this case the required elements are such that they can provide graded acceptance of signals, and graded response. To use boolean circuits to provide the response required by the complex domains that are encountered every day would appear to be so difficult that (we hold) even evolution would not have been able to build a satisfactory system out of such components. The real question is how did a system that has graded response come to evolve into a system that can manipulate symbolic entities. It may be that, in our desire to simulate the highest levels of human behavior, we have been overlooking the fundamental information processing that is required to produce the variables that support such performance.

References

[1] Atkin, L. R., Gorlen, K., and Slate, D.
Chess 3.0 - An Experiment in Heuristic Programming.
1971.

[2] Baylor, G. W., and Simon, H. A.
A Chess Mating Combinations Program.
In *Proceedings of AFIPS*, pages 431-447. AFIPS, 1966.

[3] Berliner, H. J.
Chess as Problem Solving: The Development of a Tactics Analyzer.
PhD thesis, Carnegie-Mellon University, 1974.

[4] Berliner, H.
The B* Tree Search Algorithm: A Best-First Proof Procedure.
Artificial Intelligence 12(1), 1979.

[5] Berliner, H. J.
Backgammon Computer Program beats World Champion.
Artificial Intelligence 14(1), 1980.

[6] Clarke, M. R. B.
Appendix. King and Rook against King.
In M. R. B. Clarke (editor), *Advances in Computer Chess 1*, pages 116-118. Edinburgh University Press, 1977.

[7] Huberman, B. J.
A Program to Play Chess Endgames.
PhD thesis, Stanford University, 1968.

[8] Lowerre, B., & Reddy, R.
The Harpy Speech Understanding System.
In W. A. Lea (editor), *Trends in Speech Recognition*, . Prentice-Hall, 1980.

[9] Mackworth, A. K.
Consistency in Networks of Relations.
Artificial Intelligence 8(1), 1977.

[10] McCarthy, J.
A Tough Nut for Proof Procedures.
AI Project Memo 16, Stanford University, 1964.

[11] Minsky, M., & Papert, S.
Perceptrons.
The MIT Press, 1969.

[12] Newell, A., Shaw, J. C., 'and Simon, H. A.
Report on a General Problem Solving Program for a Computer.
In *Information Processing: Proceedings of International Conference on Information Processing*, pages 256-264. UNESCO, Paris, 1960.

[13] Pitrat, J.

A Chess Combinations Program Which Uses Plans.
Artificial Intelligence 8(3), 1977.

[14] Reitman, W., and Wilcox, B.
The Structure and Performance of the INTERIM.2 Go Program.
In *Sixth International Joint Conference on Artificial Intelligence*, pages 711-719. IJCAI, 1979.

[15] Samuel, A. L.
Some Studies in Machine Learning Using the Game of Checkers.
IBM Journal of Research and Development 3(3):210-229, 1959.

[16] Samuel, A. L.
Some Studies in Machine Learning Using the Game of Checkers, II - Recent Progress.
IBM Journal of Research and Development , November, 1967.

[17] Samuel, A.
The Duke vs. Stanford Computer to Computer Checker Match.
SIGART Newsletter (63), June, 1977.

[18] Shortliffe, E. H., & Buchanan, B. G.
A Model of Inexact Reasoning in Medicine.
Mathematical Biosciences 23, 1975.

[19] Slate, D. J., & Atkin, L. R.
CHESS 4.5 -- The Northwestern University Chess Program.
In P. Frey (editor), *Chess Skill in Man and Machine*, . Springer Verlag, 1977.

[20] Wilkins, D.
Using Plans in Chess.
In *Sixth International Joint Conference on Artificial Intelligence*, pages 960-967. IJCAI, 1979.

[21] Zuidema, C.
Chess, How to Program the Exceptions.
Technical Report, Afdeling Informatica, 1975.