Artificial Intelligence Based Board Game: Backgammon

Project Report submitted in partial fulfillment of the requirement for the degree of

Bachelor of Technology.

in

Information Technology

under the Supervision of

Ms. Ramanpreet Kaur

By

Tushar Gupta 111444

to

Jaypee University of Information and Technology

Waknaghat, Solan – 173234, Himachal Pradesh

Certificate

This is to certify that project report entitled "**Artificial Intelligence Based Board Game: Backgammon**", submitted by **Tushar Gupta** in partial fulfillment for the award of degree of Bachelor of Technology in Computer Science & Engineering to Jaypee University of Information Technology, Waknaghat, Solan has been carried out under my supervision.

This work has not been submitted partially or fully to any other University or Institute for the award of this or any other degree or diploma.

Date: 15.05.2014 **Supervisor's Name : Ms. Ramanpreet Kaur**

Designation: Assistant Professor (Grade-I)

Acknowledgement

I, Tushar Gupta am using this opportunity to express my gratitude to everyone who supported me throughout the course of this project. I am thankful for their aspiring guidance, invaluably constructive criticism and friendly advice during the project work. I am sincerely grateful to them for sharing their truthful and illuminating views on a number of issues related to the project.

I would also like to thank my project guide Ms. Ramanpreet Kaur who provided me with the knowledge being required and conductive conditions for my project.

Date: 15.05.2015 Tushar Gupta

ARTIFICIAL INTELLIGENCE BASED BOARD GAME

Backgammon

Table of Contents

1. Introduction

1 . 1 . Background

In 1979, the backgammon program BKG 9.8 won an exhibition match in Monte Carlo against the then-world-champion Luigi Villa by a score of 7-1. Although analysis of the match makes it clear that BKG had some good luck with the dice, and made several small errors, there is little doubt that the program is a very good backgammon player. The BKG program does not rely primarily upon tree searching to select a move, but upon a large base of backgammon knowledge coded in the form of an evaluation function. This function is applied to the positions resulting from each legal move in a given position to determine the best move to make, resulting in a fixed-depth, 1 ply search with terminal evaluation. During die course of the development of the evaluation function, several key ideas emerged as central to the success of the endeavour. Collectively, these ideas were referred to as the SNAC method of evaluation, standing for Smoothness, Nonlinearity, with Application Coefficients. The goal of the BKG project was to develop a fast program with a very high level of competence. To achieve speed, the evaluation function was written in a compiled language and in such a manner as to avoid expensive or unnecessary constructions wherever possible. The original goal of the PROGRAM2 project, which was begun in 1980, was to augment the BKG system with an explanation-generating component, to allow it to defend its choice of moves and comment on suggested moves.

1.2. Generating explanations

It has become apparent that in order for a computer program to be considered genuinely expert in some field, it is insufficient for it to only be able to make expert-quality decisions.

The expert program must in addition be able to justify its decisions in a manner that (at least) other experts in the field can understand. The reasons for this requirement are discussed in with respect to the MYCIN medical diagnosis system, and are quite generally applicable.

The MYCIN work focused primarily on two sorts of justification: explaining why any particular bit of information is needed in the process of a diagnosis, and explaining how any particular conclusion was reached. Since MYCIN'**S** control structure is a depth-first search through an AND/OR tree, answering the first question amounts to moving up the tree toward the root and explaining what the requisite information was a sub goal of, and answering the second question amounts to moving down the tree and explaining what sub goals were used in satisfying a particular rule. This intuitively-pleasing scheme was also used in [21] in the question-answering component of the SHRDLU blocks world program. Since PROGRAM does not employ a search procedure of this sort, it is not immediately clear how an analogous question-answering method might be devised for such a system. In order for PROGRAM to perform competently, however, the information which can be gained by tree searching must somehow be represented in PROGRAM's evaluation function. The move to make, then, is to replace analysis of the search tree with *analysis of the evaluation function.* In order to do this, the system needs to have access to the various components of the evaluation function that combine to form the overall judgement Whereas in BKG only the resulting *heuristic value* of a possible move was relevant, in PROGRAM the internal structure that computes the heuristic value is relevant as well. Given that internal structure, some means must be devised for expressing essentially continuous data in the discrete, symbolic fashion of natural language. Furthermore, in order for the generated explanations to be comprehensible, the internal structure of the evaluation function must correlate in certain direct senses with the structure of backgammon knowledge as humans comprehend it. These constraints led to the development of a set of guidelines for constructing evaluation functions, and, implicitly, a scheme for knowledge representation and use which is quite different in appearance and structure from most schemes that have been explored in AI research.

1.3. Non-discrete knowledge representation

Throughout this report we argue the dangers of discrete and extol the virtues of non-discrete knowledge representation, so it is important to make clear what we mean by the terms and to set the context for what we are trying to say. The basic distinction is between *few valued* and *many-valued* representations, with two valued Boolean logic at one extreme and uncountable infinite-valued differential equations at the other. By a non-discrete representation of a proposition, say, we mean allowing for a large set (twenty or fifty or thousands, depending on the situation) of possible degrees of confidence or belief in the proposition. Thus a

statement about chess positions such as " J C is an endgame" would be asserted with many differing degrees of confidence depending on how well each particular *x* fits the known constraints on being an endgame. When we speak of "continuous" values we mean precisely this sort of many-valued representation. We do not mean to suggest that there is something going on in mathematically continuous functions that cannot be adequately modelled in a finite-valued system. Nor do we mean to suggest that discrete symbol manipulation

in a two-valued system is by any means useless. When a distribution of continuous values is largely bimodal, as many are, making the assumption of two values affords a great simplification in the problem space and allows greater progress with a given amount of resources. In particular, solving the credit assignment problem to perform learning is greatly facilitated by discrediting the problem space and hypothesizing cause-and-effect

relationships between the resulting states, *provided* that the discretisation is performed in a manner appropriate to the domain and the goals of the problem-solving system.

What we will stress is that determining such appropriate discretizations is an inescapable, non-trivial task for intelligent systems operating in incompletely understood, complex domains, a task which must be performed in a domain-and-goal-dependent manner if discrete reasoning is to provide useful results. Furthermore, we will emphasize the serious limitation of few-valued reasoning: the occurrence of "boundary

cases", where the few-value assumption leads reasoning astray. We will show that this cannot be avoided when a system does not completely understand its domain, and must be handled with a many-valued representation.

1.4. Reasoning and Judgement

One of the outgrowths of this work has been the realization that the everyday notions of reasoning and judgement find apt expression within this structure. This is discussed at some length in Section 5, but it is worth taking a moment here to describe what we mean by the two terms. In very general terms, reasoning, as we see it, is the process of imagining the environment to be other than it is. The sine qua non of reasoning is the existence of an internal world model which can be perturbed to perform this imagining. By this definition, any sort of search in a problem space is reasoning, whether it be tightly constrained knowledge-intensive best-first search or a random walk. The process by which the PROGRAM

system tries out possible moves on an internal model of the backgammon board is a reasoning process. *Judgement,* in similarly general terms, we see as the process of *forming an interpretation of the environment with respect to a goal* A thermostat which classifies sensed temperature in terms of "too hot", "too cold" or "OK" is making simple judgements. The fundamental characteristic of judgement is that it is purposive that its results relate to some goal whereas reasoning, in *the* abstract, may not be. The possesses an evaluation function which interprets its environment, backgammon positions, with respect to the goal of winning the game. By this characterization, it should be clear that our position will be that all problem solving systems are to be viewed as performing both reasoning and judgement in varying degrees and combinations. Search strategies are methods of using judgement to control reasoning. In best-first searching, for example, large portions of the problem space may be ignored because it is judged that the best value obtainable in those portions is not as good as a value already at hand. It is not our primary purpose to defend these uses of die terms here, but to give a flavour for the distinction that will be made throughout this report. Table 1-1 summarizes some of the distinguishing aspects of reasoning and judgement as we see them.

Reasoning is	Judgement is
Qualitative	Quantitative
- few-valued, discrete	- many-valued, continuous
- response either unchanged	- graded response

or drastically changed by small differences	' to small differences
Causal	Correlative
- therefore, ultimately	- therefore, potentially
Sequential	Parallel

Table **1-1:** Comparison of reasoning and judgement

2. The PROGRAM implementation

For purposes of gaining perspective and bringing the discussion down to a more detailed level (at least temporarily), it is worthwhile to. look briefly at the "guts" of the PROGRAM program and see how it is organized. The breakdown is basically in terms of the general structure presented in the previous section, although the program was not formally designed that way when it was first constructed. The *Real World* consists of thirty-two integers of appropriate sizes representing the contents of the twenty four points of a backgammon board, the contents of each player's bar and home, the values of the two dice, the position of the doubling cube, and which side is to move next. There are constraints on the values of

these integers corresponding to the rules of backgammon. The *World Model* is essentially a duplication of the Real World, but it is transformed "egocentrically", soothe player who is to move always appears to be playing White and moving in the same direction around the

board. The World Model is such that it can always be derived as needed from the current Real World, so that the program has no sense of history; for example, it is not possible for the PROGRAM program to observe and exploit weaknesses on the part of an opponent.

The *Judgement Structure* comes in two basic parts. There are a set of support routines that extract useful information from the World Model. These routines compute a wide variety of functions that have been found useful for guiding actions, from simple matters such as the number of points a player has made in his home board to more complex and esoteric functions such as the second moment of inertia of a position about the centre of gravity, considering the men as point masses. The second part of the Judgement Structure is the hierarchically structured evaluation function that produces a measure of the goodness of a given position, using the World Model and the results of the support routines as input This component is at the heart of this research and is considered in Section 3. This division of the Judgement Structure is in some sense unnecessary, since the language in which the

structured evaluation function is expressed is powerful enough to compute the functions produced by the support routines directly from the World Model. However, to do so would lead to a significant slowdown in the speed of the system, due primarily to the cost of unrolling loops used in the support routines.9 The penalty for using support routines such as this is that the analysis system is unable to make any explanation for why the value of a support routine is what it is. A good way to view the support routines is that they provide an

"Enhanced World Model" in which determining behaviour is easier and works in more powerful terms. That the system is unable to explain the enhancements in lower level terms is no more upsetting than the fact that humans are typically unable, for example, explain how they recognize a familiar face. The *Action Set* for PROGRAM is simply all legal moves from a given position as determined by the constraints forming the rules of backgammon. In the implementation, this set is never directly collected; instead the move generator proceeds around the board looking for legal ways to play the dice, and whenever one is found it is played in the World Model and the Judgement Structure is applied to it. The incrementally best move is kept along with its heuristic value until all moves have been examined. This move generation and evaluation procedure forms the entirety of the *Reasoning System* in the PROGRAM implementation. The *Analysis System* is entered on the user's request after a move has been played. The task of interpreting die Judgement Structure for purposes of generating explanations is performed by comparing a suggested alternative move with the move just played. The Analysis System must distinguish significant from insignificant factors in a comparison, and for those issues judged worth mentioning, must produce some indication of the relative importance of each. The Analysis System is discussed in Section 4. Since PROGRAM is not a learning system, the Analysis System proceeds under the assumption that the Judgement Structure is perfect An Analysis System for a learning program would need to have access to statistical performance data on the program's behaviour, to be able to discover errors in its World Model and Judgement Structure.

3. Constructing evaluation functions

The approach to representing heuristic knowledge as an arithmetic Junction is presented stressing the necessity and the danger of non-linearity. The notion of an application coefficient is described An evaluation function for a simple task is developed as an example of the method and the pitfalls. The phenomenon of the blemish effect and the consequent unavoidability of "boundary cases" is discussed The language devised for the PROGRAM system is presented

3 . 1 . The task of an evaluation function

The basic task of the knowledge representation scheme for judgements is to *approximate the relationship between the sensory data and a utility metric.* The utility metric must specify how well a given set of sensory inputs is expected to satisfy the goals of the organism. The PROGRAM system defines a language in which its Judgement Structure is written. The language is described in some detail in Section 3.5; here, we will consider some of the issues that led to the language being defined as it was. An evaluation function is a more or less successful approximation depending on how well it is able to order the states of the Real World in terms of nearness to a goal state with respect to the actions the system can take. In general, the quality of the approximation can be traded off against the amount of searching done; at one extreme, the evaluation function only discriminates goal states and the search must blindly hunt for one, at the other extreme, the evaluation function totally orders the states of the Real World and a one-ply hill climber proceeds directly to the nearest goal state. Because of the combinatory nature of searching, it seems desirable to try as far as possible to push towards the high-quality approximation end of the spectrum.

3 . 1 . 1 . Linear and non-linear evaluation functions

One of the conceptually simplest methods of evaluating the quality of a given state of affairs in the Real World is just a linear combination of the inputs to the system (such as counting points to evaluate the quality of a bridge hand, or counting material on the chess board with some weighting scheme such as "A rook is worth five pawns" and so on). A linear evaluation function attempts to provide a uniform measure of utility for every state in the problem space, providing a total ordering with respect to nearness to goal states. For sufficiently simple spaces, a linear approximation can work well, unfortunately, they have to be *very* simple.

Any problem space where the utility of a given input is not constant or nearly so is too complex. Tic-tac-toe, for example, when the only inputs are the contents of the nine squares, is too complex for a linear combination of the inputs to succeed. By enhancing the original nine inputs with a few non-linear functions of them, such as the difference between the number of X's and O's on each possible winning line, a linear combination of this expanded set of inputs can be made to totally order the game states with respect to nearness of winning.

3.1.2. Heuristic knowledge

In more complex problems such as backgammon or chess, no one has yet found any computation, other than hypothetical complete game trees, that will totally order the states of the problem spaces. What have been found are large collections of heuristics that provide partial orderings for small subsets of the problem spaces. Heuristics are frequently expressed in the form "If conditions are then try to accomplish *y"* Interpreting such a heuristic in terms of an evaluation function, our first suggestion is that a heuristic specifies a region *x* of the problem space within which the relation between *y* and utility is approximately linear. A

collection of heuristics breaks a problem space up into a collection of (frequently overlapping) regions within which specified linear relationships between features and utility hold. Thus, the heart of the representation scheme is computations of the form:

Heuristic = (degree to which the current state is in region *x)**

(constant of proportionality between utility and *y)**(degree to which *y* is accomplished)

However, most heuristic knowledge does not relate directly to any abstract notion of utility or goodness. More commonly, a heuristic relates quantities which only indirectly bear on utility. Consider the following beginner's chess advice:

When there are 13 points worth of material on the board, or less, it is an endgame position.

This heuristic suggests a (rather discrete) method of estimating the "end gameness" of a chess position. It has nothing to do with utility directly, instead it helps determine to what degree you are in a region of the problem space where all the endgame heuristics are relevant to utility. Similarly, consider the backgammon blockading heuristic mentioned in Section 3.3:

The goodness of a blockading formation is proportional to the cube of (36 — the number of

rolls that allow a given man to escape across the blockade).

The addition of this heuristic caused a dramatic improvement in the quality of the BKG program's play. It does not relate directly to utility, since in some circumstances blockading enemy men is an undesirable goal, nor does it help determine a region in which other heuristics apply. What it does do is suggest a means of determining how well the goal of trapping enemy men is being accomplished. Other heuristics then specify how the goal of trapping enemy men relates to utility. In general, this sort of heuristic knowledge may relate quantities in a region, or relate quantities to degree of being in a region, or relate quantities to degree of accomplishment of a goal. Thus, the basic scheme for representing a bit of heuristic knowledge about some concept was:

(degree of w) = (degree to which the current state is in region *x)**

(constant of proportionality between wand >>)*

(degree to which *y* is accomplished)

The picture of the representation scheme that emerges from this is a layered, hierarchical structure. At the very top is heuristic value or utility. Immediately below that is a layer of heuristics all of which relate directly to utility. In each successive layer are sets of heuristics which provide methods of computing the values used by the layer above. At the bottom of the hierarchy are values which are produced directly by the World Model. As an example, a somewhat schematized version of the top few branches (and a couple of the leaves)

of the PROGRAM Judgement Structure is shown in Figure 3-1.

In PROGRAM, the Judgement Structure is treated in most places as if it were a tree, but in fact it is a directed acyclic graph. Since bits of heuristic knowledge are frequently useful in several different areas of the structure, there can be multiple parents for a concept as well as multiple children. MyPipcount, for example, is a component of several ascendancy chains leading to, among other things, considerations about how to bear

men off, how much to worry about maintaining mobility and a flexible position, and whether there is enough time to perform a return play or release play.1 3 Still, hierarchical structuring falls naturally out of the "patchwork" nature of heuristic knowledge, and is an important principle precisely because it supports the convenient expression of heuristic knowledge.

3.2. Representing collections of heuristics in an evaluation function

The basic method of representing heuristics presented in the previous section was designed so that a set of heuristics that all relate to utility can simply be added to derive a measure of the collective utility of a given state of affairs. The basic knowledge structuring mechanism is thus a sum of terms, where each term is composed of an *application coefficient A* specifying the region in which the heuristic applies, a *constant of proportionality* C, and a *feature* ^ which is to be related to utility. Figure 3-2 shows the general form.

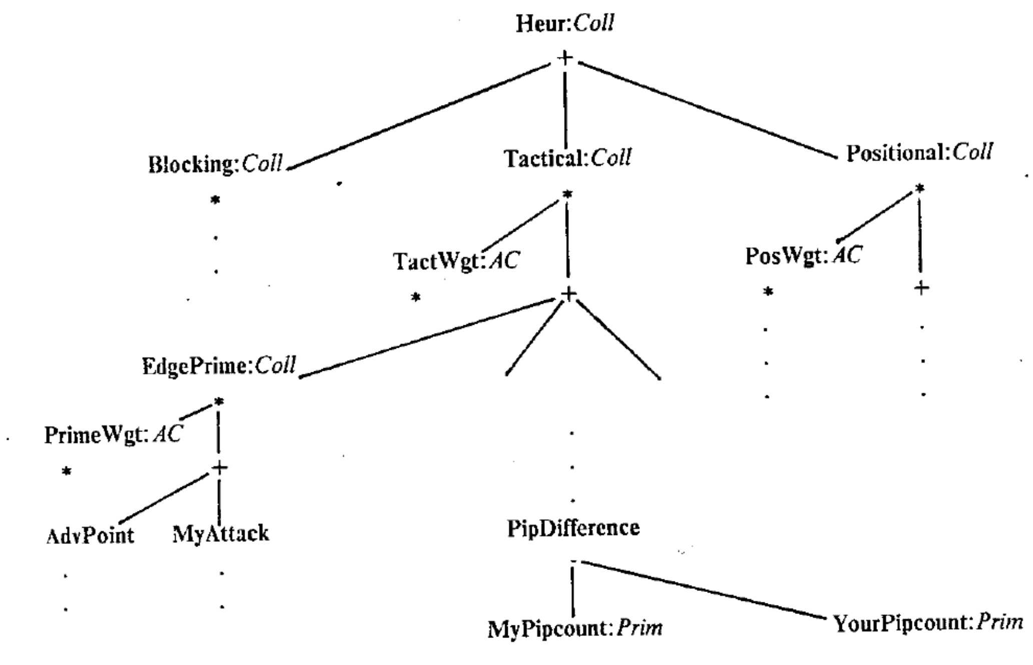

Figure 3-**1:** Schematic view of portions of die PROGRAM Judgement Structure.

F = the feature being defined,
K = the set of features from which Fis derived,
Aj. = a real variable in the range (0,1), the application coefficient associated with this use of feature /
Cj = a signed, real constant associated widi this use of /
Figure 3-2: Method of combining heuristic knowledge

The summation operator is the most common mechanism whereby differing knowledge sources are combined: where apples and oranges are compared. We call *F* a *heuristic, feature, concept,* or *node,* corresponding to viewing the Judgement Structure as a collection of heuristic knowledge, combinations of input data, a symbolic hierarchy, or a parse tree of an expression. *F* is said to *depend* on *K.* This recursive

definition (in concert with similar definitions for the other operators, as in Table 3-2) terminates at the leaves of the parse tree with *primitive features* whose values are obtained directly from the the World Model.

3.2.1. Representing the limits of linearity

The *application coefficient, A,* is of critical importance in the construction of effective evaluation functions, as it is a main source of non-linearity in the function. The fundamental difficulty in using non-linear functions is the problem of *instability:* while a non-linear function may approximate utility quite closely over portions of the ranges of values of its inputs, the approximation can go drastically astray for some combinations of inputs if it is not carefully controlled. For example, IQ tests are supposed to approximate "intelligence" in some fashion. Idealistically, they are computed by the non-linear function *(raw score)/(age).*

This works adequately for ages in the range of about 6 to 16, but as the subject grows older, their IQ score by this metric begins to drop steadily. In reality, IQ scores are computed by the slightly more complicated function *(raw score)/min(ageJ6).* By clipping the denominator in this fashion the tests results become much more stable. The method *(Clip the inputs)* used to "fix up" the IQ computation is one of many ways to represent the limits of validity of an approximation. Some other common methods are:

• *Clip the result.* This technique is sometimes used in academia: Giving a test with, say, two hundred points possible, but grading on a one hundred point scale. It sacrifices resolution at the high end, and tends to be popular with the majority of students.

• *Scale the result.* This is frequently used when several results need to be linearly related, such as several exams that are supposed to have equal impact on die final evaluation. Scaling a result down sacrifices resolution over the-entire range, and scaling a result up sacrifices resolution in the other results.

• *Tighten applicability criteria.* This is the method most commonly used by symbol-manipulation programs, as in Winston's arch-learning program [22]. A crude definition of an arch (e.g., "three blocks") gives die wrong answer (i.e., is unstable) on many structures and is tightened up ("three blocks with two supporting the third and not touching each other", etc.) This method sacrifices resolution in the space between the old, looser criteria and the new ones. (E.g., if a system thought three stacked blocks formed an arch, but was told they didn't, then what should the system think they *do* form?) In the IQ example, it might say that unless you are between six and sixteen you do not possess an IQ.1 4

• *Break into multiple approximations.* This technique is frequently used in concert with tightening applicability criteria, to preserve the resolution that would otherwise be lost. It can be seen almost everywhere you look, from the aesthetics of Impressionism and Dada to quantum mechanics and Newtonian mechanics and relativistic mechanics. In theory, no resolution will be lost, however, the price is paid in the difficulty of evaluating states that are close to the "turnover points" between the approximations.

All of these methods and others have their place, circumstances in which they perform effectively. In many cases, however, statements can be made about how these techniques should be applied in order to avoid unexpected side-effects. Consider the issue of applicability criteria. As discussed below in Section 3.3, it is in general unsatisfactory to simply supply a numeric range or a Boolean predicate for a given relationship and only include it in the evaluation when the input is within the range, because this induces discontinuities at the extremes of the range. What is needed is a "degree of applicability" factor that decreases smoothly to zero as the input approaches the limits of linearity with respect to a relation, and this is precisely what application coefficients are designed to provide.

Local linearity: The fundamental condition for using non-linearity in this fashion is that the condition of *local linearity* be maintained: when evaluating positions *near* each other in the problem space, the evaluation function must be effectively linear in its inputs. Restricting non-linearity to the multiplication of features by application coefficients maintains that condition because the ACs change slowly and smoothly with respect to

the rate of actions. In a small region of the problem space, therefore, they are assumed to act as constants. A consequence of local linearity is that, while sufficiently similar sets of inputs may be evaluated and compared for relative utility, when the sets of inputs are different enough that the application coefficients are significantly changed, then comparisons of the computed heuristic values will not in general yield a valid absolute comparison of the two sets of inputs. Application coefficients provide the Judgement Structure with context sensitivity, allowing control over what information is deemed relevant in any given circumstances. It is widely recognized, for example, that the information important to the end game in chess is quite different from that needed in the middle game; it is also widely recognized that there is no well-defined boundary between the two phases of a game. An

application coefficient allows for the smooth "turning down" of the importance of middle game issues and a corresponding "turning up" of the importance of end game issues as the game progresses. Typical examples of application coefficients from the PROGRAM program include the pipcounts for the two players (which is an indicator of the expected length of the remainder of the game) and the degree of blockading of the opponent (which is a central strategic quantity indicating the degree to which one side is in charge of the situation.) In the evaluation function for the */ago* Othello program [17], the move number is

used as an application coefficient, to gradually sensitize the program to changes in context as the game progresses. This is effective because an Othello game cannot exceed sixty moves. Application coefficients are also being used in an evaluation function designed to judge plans for job-shop scheduling , measuring such features as the amount of free space remaining on the floor.

4. Generating explanations

The main related issues in explaining judgements are presented: determining what is relevant, and determining when a quantitative difference should be explained as a qualitative difference. The heuristics employed in PROGRAM for making such decisions are presented. Examples of PROGRAM's commentary are presented and discussed

The explanation mechanism of PROGRAM must handle two main issues. First, it must isolate the backgammon knowledge relevant to any particular query from the large amount of knowledge that does not bear on a given situation. The second issue is to provide some mechanism for deciding when quantitative changes should be viewed as qualitative change; in essence, to provide the ability to make distinctions that discrete systems enjoy

free by virtue of their discrete representation. (There was relatively little effort expended in the generation of natural language output; in the examples below, the output has been left "in the rough" as the system generated it. Most "language issues" have been ignored.) The explanations that PROGRAM generates cover only a portion of the possibly useful comments that could be made, depending on the particular situations and on the level of backgammon knowledge of the listener. Some of the explanation topics which PROGRAM does not address are discussed in Section 6.1.

4 . 1 . Finding relevant issues

PROGRAM is oriented around answering the question "Why did you make *that* move, as opposed to *this* move?" This reduces the explanation task to one of accounting for the *differences* between a pair of moves. The fundamental assumption of the explanation process is diet important differences between a pair of moves will be reflected by "large" changes in the values of die highest level concepts that are related to the differences. Letting *Movel* denote die move with the larger Heur and *Move!* the one with the smaller, define *Sconcept* = value of *concept* for *Movel* - value of *concept* for *Move2*. Referring to Figure 3-1, this assumption

implies that if *8* Blocking is "small", then backgammon knowledge related to blocking is not relevant to this comparison and should not be mentioned. If only one sub concept of Heur, say, Tactical, is not small, then all interesting differences are with respect to Tactical concepts, so the level of discourse for comparison is narrowed to just tactical knowledge, and the process repeats on the subconcepts of Tactical. That is the method by which the relevant backgammon knowledge is isolated. Beginning at Heur, the system searches down the tree until a level is reached at which more than one significant difference is found.2 3

As desired, if the two moves are radically different in their effects, the commentary will begin at a relatively abstract level (e.g., tactical and positional issues) and if the two moves are quite similar, the commentary will focus on the crucial differences at whatever level they are found. If the discussion is at a fairly narrow level, it is usually sufficient to just describe the differences and their magnitudes; at higher levels, this leads to unsatisfying, "hand waving" commentaries. The level at which an explanation feels satisfying varies from person to person and topic to topic2 4 , so we have adopted a simple heuristic. The broad concepts at the top of the tree are denoted *collections (Coll* in Figure 3-1), and the system is built to automatically "look inside" any collections that are mentioned; in effect, supplying for free the question "Why is there a difference in that collection?"

4 . 2 . Qualifying differences in magnitude

The above discussion is predicated upon having the ability to recognize "large" or "significant" differences in the values of concepts. In a two-valued system, any difference is a large one (on the order of *True* versus *False)*, and the process of recognizing significant differences is done *outside* of the system, during the generation of discrete primitive observations of a continuous world. Unfortunately, it is impossible to determine what should be considered significant without considering the *context* within which the judgement is to be made. A difference of ten feet, for example, is much larger in the context of "Distance I am from the ground" than in the context of "Distance I am from the moon." A context provides a means of classifying differences into fuzzy classes or "buckets" such as "about the same", "somewhat larger", and so on. The number of classes will vary depending on personal taste as well as the degree of refinement of the knowledge base. The heuristics in PROGRAM employ different numbers of buckets, from three ("insignificant", "significant", "major") to six ("not significantly", "slightly", "somewhat", "much", "very much", and "vastly"). In terms of the PROGRAM structure, the context of a concept is the set of more general concepts of which it is a part. Some of the more primitive concepts, such as

MyPipcount, appear in several places in the Judgement Structure and can therefore be judged in several different contexts. To judge a difference in context, it is necessary to determine how that difference affects the value at the top of the Judgement Structure. Given the sign and magnitude of a difference that is to be considered a significant improvement for Heur, this *goodness metric* can be propagated down through the tree to determine how much better or worse one move is than another with respect to a given concept in a given position. In Figure 3-1, for example, if the goodness metric for Heur is assumed to be +10, and in a given position TactWgt equaled 3, then the goodness metric for EdgePrime would be +10/3, and a 5 EdgePrime of less than 10/3 would be judged "about the same", a SEdgePrime between 10/3 and 20/3 would be "somewhat better", and so on. Using this procedure would require an *a priori* goodness metric for Heur. In PROGRAM, probably the most satisfying overall context would be "What is the expected value of die game?" with a goodness metric of perhaps a hundredth of a point. Such an evaluation function could in theory be built, and in fact the system has an independent computation used to approximate the expected value, which is used in making doubling decisions. The original BKG evaluation function only needed to order the possible moves with respect to a given initial position, and this "relative" nature remained through the translation to the PROGRAM-style evaluation function, so Heur values resulting from different initial positions are not directly comparable.

With respect to a given position, however, various heuristics have been devised that empirically give satisfactory results in the determination of significant differences.

While an absolute goodness metric would cleanly solve the problem of determining the overall significance of a difference in a concept, there is reason to suspect that such a metric might be so difficult to come by for complex domains that systems in general must make do without. Clearly, an error-free, absolute goodness metric for a problem space, of sufficient resolution to distinguish between individual actions, would constitute an algorithmic solution to the problem. If such a metric were available for a domain and a goal, few would consider a system using it to be performing "problem-solving" in any satisfying way. On the contrary, the very spirit of heuristic knowledge is tiiat it is not truly global or absolute, but instead specifies some set of circumstances in which this much more of something is that much better, *relative* to those circumstances. When actions are determined by using heuristic knowledge to compare results, the justifications for those actions, ultimately, can have no higher appeal than "It seemed like the best thing to do under the circumstances."

There are several different heuristics employed for qualifying differences in the commentary generated by PROGRAM. Three of them are used to generate preliminary remarks about the general game situation and the overall relative merit of the two moves, the others are used to judge differences in concepts within the Judgement Structure, depending on the particular operation involved in the concept. In all cases, the task is basically the same: to produce a set of "buckets" labelled with appropriate words and to fit a difference into a bucket that would seem appropriate to the competent backgammon player reading the commentary. The tasks and methods as of the writing of this paper are as follows.

4.2.1. Judging the absolute status of the game

PROGRAM's commentaries always begin with a statement of its opinion as to who's ahead in the game and by how much. This sets the stage for the explanation derived from the comparison of the moves, since which high level goals are relevant typically depend on who has the advantage and by how much. Two statements are made, one about how far ahead*or behind the player is in the race, and one about whether the player is at an advantage or disadvantage.2 6 The first statement is obtained by taking the difference in the pipcounts and bucketing the value according to the common backgammon dogma. The second statement derives from PROGRAM's computation of the expected value of the game. The expectation computation is an admittedly crude piecewise approximation producing a number from -300 to + 300 with 100 representing an almost certainty of winning the game, 200 for winning a gammon, 300 for winning a backgammon, and symmetrically for losing and the negative values. There are some seven or eight functions combined in the approximation, which does not use application coefficients to combine the heuristics. The expectation is not used in the Judgement Structure, so the blemishes in the computation do not impact the quality of the move selection. Outside of the explanation system, the expectation is used only for the Double/Don't double and Accept double/Reject double decisions, which do not involve the hypothesization machinery of the World Model and thus escape the clutches of the blemish effect2 7 Within the explanation system, while a human expert might occasionally quibble about the precise wording used (thinking "strong advantage", say, would be more appropriate than "very strong advantage" in a given position), the computation is accurate enough that the error is very rarely seen to be more than "off by a bucket."

4.2.2. Judging the relative overall quality of the two moves

After setting the stage, the commentary makes a statement about the relative worth of the two moves. PROGRAM, of course, played the best move it could find, so naturally it finds the suggested move to be, at best, not significantly worse than the move it played. This computation (Figure 4-1) is based on the heuristic values of the two moves in the context of the range of heuristic values possible for all legal moves. The difficulty which the Goodness(c)2 8 computation is addressing is that, while a given difference in heuristic

value may represent a vast difference in quality in one set of circumstances, in another set of circumstances the given difference may represent a relatively serial difference in quality. In some situations, when there are only a few relevant heuristics, the best move may be distinguished from a significantly inferior one by only a few points, while in a more involved situation a difference of dozens of points may be relatively unimportant. Let *Hj* ... *U* be the heuristic values of all *n* legal moves sorted into decreasing order. Then *H,* is the heuristic value of the */* move that was actually made. Let *H c* be the heuristic value of the move *c* suggested for comparison. For move /, define:

WRank(z) = i/ **VT**

GoodnessO) = WRank(0|//; - / / . | / V | / / ; - *Hn/2*

Then for move *c:* Goodness(c) < 0.2 —• "Actual move about as good as suggested move"

< 0.8 —> "Actual move better than suggested move"

> 0.8 —• "Actual move much better than suggested move"

Figure 4-1: Determining relative goodness of a comparison move

The computation uses the heuristic value of the median move as an estimate of the volatility of the position to obtain a notion of how valuable a point of heuristic value is in the current position. The WRank (weighted rank) function adds a measure of sensitivity to the number of moves that are better than the move suggested for comparison. The statistical rank of a move is modulated by a function of the number of possible moves to produce this measure; the intuition behind this is basically that it is better to be the second best of one hundred moves than second best of ten. In many cases, however, die number of possible moves is deceptive.

Frequently, there will be several moves all of which address the major issues in the position and only differ in subtle ways, and a whole string of other moves of much lower heuristic value which do not Taking the square root of the number of moves is a rough attempt to compensate for this long tail of terrible moves, so, for example, being second best of ten is considerably better than being 20t h best of 100.

4 . 2 . 3 . Judging the magnitude of a difference in a component of a SUM node

Wide the exception of the high level summations that are denoted collections and are discussed below, differences in concepts that are part of SUM nodes are handled as shown in Figure 4-2. The computation uses the Local Range of a concept to estimate how much gain is possible in this concept in this situation, and

Let: *s* be an element of SUM node that is not denoted a collection,
Ss = value of *s* for the actual move — value of *s* for the suggested move,
Local Range**(5)** = the difference between the highest and lowest values obtainable for *s* in the current position,

Global Range(s) = the difference between die highest and lowest values of *s* observed in the history of the Judgement Structure, and

SumDiff(s) = 255/(LocalRange**(5**)+max(GlobalRange**(5**),50)).
Then any SumDiff(,s) < 0.15 is not worth mentioning, and
> 0.5 —• "very much better",
> 0.25 -+ "much better",
> 0,15 —*• "somewhat better".

Figure 4-2: Determining significant differences in components of SUMs
the GlobalRange as an estimation of the general "value of points" for this concept.
Let *c* be a collection. Define:

SigColl(c) = if |5c| < 10 then 0.0 else |5c/SHeur|
Then: SigColl(c) < 0.25 —• insignificant; not worth mentioning
< 0.50 —• "a significant factor"
> 0.50 -» "a major factor"

Figure 4-3: Determining significant differences in collections

4.2.4. Judging the magnitude of the difference in a collection

The high level SUMs that are denoted collections are bucketed into just three groups: those which make an insignificant contribution to the overall difference in the value of the SUM and should not be mentioned, those which make a significant contribution, and those which make a major contribution. The collections in PROGRAM's Judgement Structure are such that they all contribute equally to the overall heuristic value, so the heuristic for qualifying differences in collections is based on computing the percentage of the SHeur which

collection accounts for. The details of the computation are shown in Figure 4-3. Insisting that a Scollection be at least ten points of heuristic value in order to be mentioned is a generally conservative guideline, so PROGRAM tends to mention topics which a human backgammon expert would consider relatively unimportant much more frequently than it skips one an expert would consider critical (see for example Figures 4-6, 4-7,

and the accompanying text.) Furthermore, in many positions some of the Scollections will have opposite signs, representing compensating aspects of the two moves (e.g., Figure 4-7), so diet SHeur is significandy smaller than the sum of the |6collections|, making it easier for a Scollection to make the significance cutoff. The last two heuristics above, for handling SUMs and collections, perform most of the commentary generated by the explanation system, but occasionally differences in nodes using other operators crop up. (Note that due to the nature of the procedure for finding relevant issues, which terminates when *more than*

one significant difference is found at a given node (Section 4.1), unary operators such as square and cube are never candidates for discussion.31) Each non-unary operator has a routine attached to it to perform explanation should multiple differences occur in its arguments; these are straightforward given the semantics of the operator. For example, the boolean IF operator has a routine capable of explaining that in one move a certain condition was true while in the other move it was false, and the differences in die values of the

then-clause and else-clause account for the overall difference in the value of the concept.

5. Limitations and extensions, summary and conclusion

The major weaknesses of PROGRAM both as a program and as an instance of a representation scheme are discussed with respect to the issues of proper behavior, making explanations, and questions of implementation. Directions in which the work could be extended are considered. The main points of the report are recapitulated

5 . 1 . Weaknesses and limitations

It can often be quite difficult to separate implementation-dependent weaknesses from failings in a general method. For the PROGRAM system it is particularly troublesome since we have at present only the outlines of a general method — principles, guidelines, and intuitions — rather than a full-blown knowledge representation language and judgement/reasoning system. Accurate evaluation of the general method will have to await more research into making judgemental systems, but it is possible now to make some rough assessments of

where some of the trouble spots are likely to be. This section first discusses problems with the PROGRAM system viewed mostly as a special-purpose expert system, and then considers difficulties for the more general model.

5 . 1 . 1 . The system

As a backgammon player, the system is more than adequate. The weakest aspect of its play is in making doubling decisions, which is a direct consequence of the lack of a sufficiently high resolution and accurate computation of the expected value of the game. In addition, there are incompletenesses in the Judgement Structure. As the performance of the system improved, the knowledge needed to improve it further became more and more specialized, so the incremental improvement began to fall off. From a point of view of effective use of research time, improving the evaluation function further eventually reached the point of diminishing returns. As a consequence of this, PROGRAM will occasionally make the wrong move in a position, or make the right move for the wrong reason, and the commentary in those situations reflects that. (The explanation system, in fact, turned out to be a useful debugging aid, for just that reason.) As a backgammon commentator, the PROGRAM system works reasonably well in many cases, displays a tendency towards verbosity a significant amount of the time, and very occasionally omits mention of an aspect of a position that most competent backgammon players, and ourselves, would have expected to be at least mentioned, if not prominently featured. Overall, the main failings of the PROGRAM system as an expert system can be summarized as follows:

1. *Failures in qualification heuristics.* A major limitation which prevents the PROGRAM organization from being extended to a more general-purpose judgement making and explaining system is the lack of an adequate general-purpose statistics gathering mechanism. The current qualification heuristics are specifically designed for PROGRAM's particular Judgement Structure, and they succeed as often as they do, based on the very limited sort of minimum and maximum value information that is collected for each concept, only because of that. As they stand, there are occasionally circumstances in which the heuristics become quite unstable and disregard an important feature *2. Failures in the Judgement Structure.* There are tilings about backgammon that people know that

\PROGRAM does not know. For example, the program has no explicit notion of contact among one's own forces. Having men and points close together allows for safer movement by increasing the chance of being able to move from point to point radier than leaving blots. A computation of the average distance from any man to the next point is an important measure of the contact of the men which is not in the current system. In PROGRAM, the only sensitivity to this issue is provided somewhat indirectly by the second moment of inertia computation.

3. *Failures in the Analysis System.* The commentary that PROGRAM generates covers only a small portion of the possible things that a human might find worthy of mention. Some of the sorts of things that PROGRAM cannot explain include:

• Counterfactuals. When comparing moves, people will frequently make statements such as

"Well, that move *would have been* better if I hadn't already gotten a man off, so you could

try to gammon me. But since I have, . . . " We considered some aspects of generating this sort of comment, and we implemented a function that would try to compute what values a set of application coefficients would have to have in order to make the suggested move better than the actual one. The results were inconclusive, but suggested that an absolute goodness metric would make such statements much easier to make.

• Upward branching dependencies. In rare circumstances, the critical difference between two moves will be due to a change in a relatively low-levfcl feature which is part of several ascendancy chains. In such a case, the method of finding relevant issues, which starts from the top of the tree, finds many genuine differences to report but never reaches the single difference which was responsible for them all. Given the organization of the Judgement Structure, the general task of finding such a "man behind the scenes" presents enough difficulties that, since the situation seemed unusual, we did not attempt an implementation.

o Cancelling differences. The fundamental assumption of the explanation generating process, as discussed in Section 4, is that important differences between moves will be reflected by significant changes in the highest level concepts related to the difference. In one sense, this is obvious: how could a concept which has not changed significantly be relevant to a comparison? There are cases, however, where two moves will have cancelling differences so that a higher level concept shows no significant change. It could be argued that a comparison of moves, especially one meant for a non-expert listener, should mention such cases, because it may not be obvious how some perhaps obvious differences between the positions actually cancel out. This circumstance seemed rare, and so dependent on what seems obvious to the listener and what does not, that we decided not to worry about it.

4. *Failures in the reasoning system.* There are situations where the PROGRAM system could significantly improve its play if it could perform a little more searching. In situations where both sides have many ways to hit each other's blots, for example, exhaustively searching through one or perhaps two subsequent dice rolls and moves would provide valuable information that is difficult to capture in a static evaluation. The.determination of whether any particular time is the right time to double could also benefit from additional searching

capability. Another situation in which PROGRAM could benefit from a more powerful reasoning system involves the computation of die application coefficients. Most of the application coefficients are computed once per move, based on the current position, and then used as constants during the evaluations of the hypothetical subsequent positions. They are computed under the assumption that at least one of the possible subsequent positions will be near the current position, in terms of the values of the application coefficients. This is the local linearity assumption discussed in Section 3.2.1. There are circumstances, however, in which the assumption fails. For example, consider a situation in which the application coefficients based on the current position indicate that die game is basically a flat-out race, and a major goal is to avoid being hit. For a given roll, however, it might be impossible to avoid leaving one or even two blots, resulting in a situation where the system is almost certain to be hit when the opposing player moves. Given that additional knowledge gained from an initial round of evaluations, a more powerful reasoning system could realize that running game considerations were becoming less relevant, recompute the application coefficients under the assumption that a blot hitting contest may ensue, and re-evaluate its options.

5.1.2. The general model

The difficulty of producing a computation for an absolute goodness metric for a complex domain stems from die conflicting needs of resolution and accuracy. In order to gain resolution, there must be significant heuristic knowledge available which is relevant in every possible position, and in order to gain accuracy, the strength of each heuristic (i.e., the values of its scaling constant) must be precisely tuned globally with respect to all the other heuristics, even those which are very far removed from the region of applicability of a particular heuristic. In complex problem spaces, this can be extremely difficult. An absolute goodness metric must have sufficient bandwidth to represent minute differences when the true utility surface is nearly flat and still have room to accurately represent the height of the highest mountains and deepest trenches. The route taken for the evaluation function in BKG and PROGRAM was to sacrifice overall comparability to gain sufficient resolution to usually pick the right move. The expectation computation used to make doubling decisions sacrifices resolution to gain overall comparability. There are hopes that one may have one's cake and eat it too in this situation. For example, recent work [8] has shown that it is possible to derive an overall measure of the degree to which a set of constraints are being violated which retains comparability over the entire problem space. If a method can be demonstrated for causing such a set of constraints to converge on an expected utility computation for a complex domain, in a reasonable amount of dme, then the problem will be solved. Such a learning algorithm is not yet in hand, but work is progressing towards that end. Another route around this difficulty, a familiar one in fact, from the proper vantage point, is to make do with less resolution in the evaluation function, and compensate for it with a more powerful reasoning system. Eurisko's extremal value heuristic discussed in Section 3.4 is an example of this. When the organism has complete control over which part of the problem space should next be explored, as Eurisko does when it designs spaceships, it can avoid a lot of relatively flat regions of the problem space entirely. In domains such as backgammon, when such flexibility is not available because the accessible regions of the problem space depend on whf re the system currently is and how the Real World decides to react, such knowledge can still supply likely landmarks to work towards. The various search and planning techniques that have been studied, such as means-ends analysis, can then be brought to bear on the task of getting there. Reasoning methods can compensate for defects in judgement, but only to a point, and only at a price. Unless the judgemental abilities are fairly sophisticated, and knowledge gained from reasoning can be converted into judgement, we feel the price of reasoning is so high as to be intolerable.

5 . 2 . For further research

As is usual, the amount of work remaining to be done is much greater than that which has already been accomplished. Settling the question of whether or not absolute goodness metrics are the right things to look for, as discussed above, is one major issue bearing on the construction of any sort of "emptyPROGRAM" knowledge representation scheme. Some other issues which we have thought about in varying amounts are discussed in this section.

5 . 2 . 1 . The quality of a judgement

The PROGRAM system has no notion of whether or not a position which it is evaluating is under the aegis of its known heuristics; for all possible positions in the problem space, the judgement structure returns a single value to the reasoning system: how good the position looks. For a system such as PROGRAM, with such severely limited reasoning capabilities, this is adequate, but a more sophisticated system would also want to know how much confidence the judgement structure has in the judgement that it has produced. If the position is an instance that has been seen many times before, the judgement structure should be able to evaluate it with confidence and say so, and if the position is particularly strange or unusual in some way, diet should be reported as well.One way to begin to provide such as confidence measure would be to examine the values of the application coefficients in the position. If they are mostly small or zero, the system has moved into a region of the problem space where its judgement will be of limited help. When some of the application coefficients are

near their maximum values, the system is on familiar ground and the confidence should be good. There are difficulties with this good-sounding idea. For example, PROGRAM's judgement structure is complete enough that some application coefficients will always be significantly non-zero, yet there are situations when the quality of the judgements are suspect. This may, of course, be due to inadequate heuristic knowledge relative to human standards, rather than because of poor-quality judgement in spite of adequate knowledge. More likely, this derives from the fact that some application coefficients are weaker than others, in the sense that regions of applicability can be widely varying sizes, and that the largest, most general

application coefficients, by themselves, do not suffice to make consistently good judgements.

Another factor which should be considered when determining the quality of a judgement is the shape of the local evaluation surface. When there is a strong slope in the landscape the judgements should be better than when the system is in, say, a very shallow bowl. Assuming an absolute goodness metric, the range of possible heuristic values in the states adjacent to the current state would provide a measure of the slope of the land, and thus provide information about the current quality of judgements.

5 . 2 . 2 . Learning to make judgements

The process of learning to make judgements involves changing the evaluation surface to more accurately model the domain. There are, basically, two kinds of learning that must take place. The more immediate kind of learning is a "tuning" process, whereby an existing heuristic is improved by adjusting any of its components: adjusting the shape of the region of applicability by altering the computation of the application coefficient, adjusting the constant of proportionality, or (recursively) adjusting the computation of die lowerlevel

heuristic from which the given heuristic is derived. Although that is simple to state, it is not so easy to see through to an algorithm for performing these adjustments, since it requires solving the credit assignment problem for a complex, non-linear system. Furthermore, even given an instance where a heuristic is somehow determined to be faulty, it is possible to fix

it up, for that particular instance, by tinkering with any of the components. However, unless the right changes are made, the surface as a whole may move away from what it is attempting to model rather üian towards it. There are rules of thumb to help determine which component should be adjusted; for example, changing the constant of proportionality is likely to be the safest change to make, since that is the only quantity which is not potentially shared by other heuristics. Similarly, changing the shape of an application coefficient is likely to cause the most widespread effects. The more open-ended sort of learning that must go on is the creation of new concepts — adding new heuristics, creating new goals to be accomplished, and defining new application coefficients. One strategy for accomplishing this could be a bottom-up approach relying on a generalization facility. In the most extreme case of a completely empty Judgement Structure, a move which leads directly to a won or lost position can be

stored as a concept with a high positive or negative utility. Such concepts would proliferate and generalizations would be attempted to produce a more compact representation. The PROGRAM system could be made to perform an analogous, but more powerful and risky, sort of learning. Rather than waiting until a won or lost position is reached, the system could watch for large changes in the value of the expectation computation, and use that as a weaker indicator of good or bad behavior. Over the course of many learning trials, one could hope that the vagaries of good and bad dice would cancel out, leaving a set of positions from which both tuning adjustments and generalizations could be computed. If the expectation computation is systematically biased, of course, dlis could lead the system away from proper behavior rather than converging. The final court of appeal is always and only won and lost positions. Such a learning scheme involves relatively little reasoning, and is roughly analogous to the judgementimproving scheme discussed at the end of Section 5.1. A system with a powerful reasoning system could learn much faster than a system without, because the adjustments to be made when the system misjudges a situation can be computed more specifically, even given only a small number of instances of the problem, using cause-and-effect hypotheses to "diagnose" the failure.40 A reasoning-weak system must exploit the statistical properties of large numbers of instances to separate the culprits from the innocent bystanders, using correlative, radier than causative, mediods.

5.2.3. Network organization

There are circumstances in which it would be useful to have downward and lateral flow through the judgement structure as well as the upward flow of evaluation. For example, one of the bugs that had to be ironed out of earlier versions of the evaluation function involved simultaneously using an input in two inconsistent ways, such as counting a given man as part of an attacking group and as part of a blockade formation, even though the man couldn't possibly be both. By careful construction of the evaluation function, such anomalies have been eliminated, but sometimes only by paying the price of fixing a given man's function using local heuristic rules. A better solution might incorporate relaxation methods, allowing alternate interpretations of inputs and middle level heuristics to compete with each other and settle down to a harmonious global interpretation of the inputs. Such an organization would be more in the spirit of our "No unnecessary discretization" motto because it would allow relevant knowledge at all levels to be brought to bear on the apparently "low-level" task of interpreting the function of a given man.

5.2.4. Architectures for hybrid systems

We have suggested that systems which are competent at *both* judgement and reasoning, rather than just one or the other, are likely to be the most successful problem solvers, but we have said little about how such a fruitful union might be created. Since we are committed to recognizing computation time as a primary design constraint, we need an architecture of some sort which will support both reasoning and judgement; one diet provides flexibility for the reasoning system, speed for the judgemental system, and a mechanism for moving

knowledge gained in the reasoning system into the judgemental system. The question of how to implement a fairly general-purpose World Model mechanism, which was side-stepped in the PROGRAM system, must be faced directly. In this section we speculate on what such a hybrid architecture might look like. One possible implementation for a World Model would be a constraint-satisfaction parallel network such as the kind proposed in [8]. Nodes in the network represent hypotheses about the Real World, and links between die nodes represent knowledge about the likelihood of the two hypotheses being true simultaneously. A subset of those nodes would correspond to the sensors of the organism, and thus would be true, initially, if and only if the measurements of the Real World by the sensors indicates that they should be.4 1 The rest of the network, corresponding to the interpretation of the portion of die Real World which is not directly observable, would settle into states which are (hopefully) maximally consistent with the observed data.

An action of the system could be defined by a set of hypotheses and truth-values which those hypotheses are to take as a result of the action. The determination of whether an action is possible at any given time, and thus the computation of the Action Set, could be performed by checking whether any of the hypotheses associated with the action are inconsistent with the current state of the hypotheses in the World Model, in the sense diet they would cause a "practically impossible" link to be asserted or a "practically mandatory" link to • be denied. In such cases, the corresponding action would be not be considered as part of the Action Set

(This is not totally satisfactory, since the inconsistency might arise as a consequence of a number of "fairly unlikely" links rather than a single "impossible" one. Actually setting the values of the action-hypotheses and measuring the change in the overall consistency of the network would be more reliable, but there are difficulties there as well.) The Judgement Structure, then, uses the states of the hypotheses in the World Model as input and

produces a notion of utility for the situation. For purposes of exposition, it suffices to imagine the Judgement Structure as being implemented in the obvious manner, with a processor for each node computing the value of the node and collecting statistics on the computed values. A single judgement would be performed, assuming no pipelining, in time proportional to the height of the structure: approximately the log of the number of heuristics in the knowledge base. A reasoning system at the level of the one in PROGRAM could then be implemented. It would hypothesize possible actions, allow the World Model to settle, apply the Judgement Structure, and perform the most favorably judged action. The Real World would respond to the action, leading to a new set of observable data, and the process would repeat If the Judgement Structure produced utility on an absolute scale, the architecture would also support searching more than just one ply away from the current position.

To move knowledge into the judgement system, it is necessary to extract it somehow from a tree search or other reasoning process. Some of the techniques explored in belief revision research (for a selected survey, see [6]) could be of use to hypothesize cause-and-effect relationships, leading to heuristic rules for incorporation into the Judgement Structure.

Open issues with respect to such an architecture include the following: The issue of meta-knowledge: in order to be effective, the system must have the "reflective" ability suggested in our model by the Analysis System. For best learning performance, a system needs to be able to judge not only the quality of a state in the problem space, but also the quality of its own reasoning processes. Some mechanism for representing die system's reasoning processes in the World Model might allow existing judgemental machinery to be applied to this task.

The^issue of plausible move generation: how to use judgement to *propose* avenues to explore rather than only *selecting* among alternatives produced by a knowledge-poor mechanism like exhaustive search. This impacts the problem of how to produce the Action Set which was mentioned above. In a sense, this suggests the notion of a stimulus-response organization as a component of an entire system — using judgementa apparatus to produce codings for actions ratiier than codings for utility.

5.3. Conclusion

• Few-valued knowledge representation is useful because it provides a great reduction in the size of a problem space by treating many states of the space as though they were equivalent, and it allows for the creation and testing of discrete cause-and-effect hypotheses. Few-valued knowledge representation is susceptible to serious errors because the binary or few-valued assumption is usually only an approximation, and with boundary cases the "round off error can accumulate and lead to absurd conclusions. One way to gain the benefit of few-valued reasoning without the risk of catastrophic error is to add judgemental knowledge. (Sections 1.3,1.4, 3.4, 5.2, 5.3)

• Many-valued, non-discrete representation is useful because it provides sensitivity to subde differences which would be lost in a few-valued representation. In many cases the subtleties are unimportant, but in certain inevitable circumstances they are critical. By maintaining a finegrained representation the smaller factors are considered in proportion to their importance.

(Sections 1.3,1.4, 3)

• Reasoning is considered as the process of perturbing an internal world model. A world model is composed of abstract symbols, probably in a fairly discrete representation, which approximates effects from causes. The major job of the world model is maintaining the consistency of the mapping between the internal representation and the observed behavior of the environment. Perturbing the world model allows for all sorts of reasoning, mathematics, and the simulation which demonstrates Turing equivalence. (Sections 1.4, 2.1, 5.1)

• Judgement is considered as the process of interpreting a state of the world model with respect to a goal. A judgement structure is a hierarchy of heuristic rules which approximate the goodness or utility of a problem state. Good judgement is fundamentally non-discrete in that it is sensitive to the small differences between problem states which a few-valued representation would ignore.

(Sections 1.4, 2.1, 3, 5.1)

• Reasoning alone is powerless since judgement is required to differentiate the utilities of possible alternate paths of action. Reasoning with too little judgement is fragile in the face of error oruncertainty. With good judgement, however, reasoning is capable of great flexibilityof behavior, by virtue of the assumption of cause and effect. (Sections 3.4, 5.2, 6.4)

• Judgement alone has limited generality since it is tailored to a particular environment and goal hierarchy. Without a world model and a reasoning system, judgement degenerates into a (possibly highly sophisticated) stimulus-response organization. With a very basic reasoning system, and in a domain that allows for a simple world model, the judgement structure of the PROGRAM system displays behaviour rivalling human performance. With a powerful reasoning system capable of retailoring a judgement structure as needs change and knowledge is refined and extended, a system may be able to show high levels of performance on diverse problems without intolerable computation delays. (Sections 5.2, 6.4)

• Heuristic knowledge can be viewed as a three-way relationship between a context and two values which can be approximated as having a linear relationship within that context. It is critical in judgement structures to avoid hard boundaries between regions of applicability of heuristic knowledge. Application coefficients provide a method of combining heuristic knowledge so as to effect smooth transitions between regions in which differing heuristics are relevant. (Section 3)

• The process of explaining a judgement involves comparing the values diet the nodes of the Judgement Structure have in one problem state with the values diet they have in another, and isolating a small set of nodes whose differences largely account for the difference in the overall utilities. By searching the structure from die heuristic value node at the root, continuing down until more than one significant difference is found in the descendants of a node, the explanation begins at the lowest level of discourse that is high enough to encompass the key differences between the two problem states. The differences deemed relevant are quantized into a small number of magnitude classes. Since the importance of a given size difference depends upon the circumstances in which it appears, the bucketing process is performed by classification-specific heuristic rules. An absolute goodness metric for an entire problem space would simplify such classifications greatly, but may be too much to hope for. (Section 4)

Appendices

```
#include <iostream>
#include <stdio.h>
#include <string.h>

using namespace std;

float start1;
float start2;
float end1;
float end2;
//system('clear');
void print_state(){
      float temp;

      if(start2 == 0){
            temp=start2;
            start2= start1;
            start1 =temp;

            temp =end2;
            end2= end1;
            end1= temp;
      }

      if(start1 == -1)
            cout << "pass";
      else if(start1 == 0)
            cout <<   "Z ->  "  <<end1 << "\n";
      else if(end1 ==0)
            cout  <<start1 << " "<< "O"<< "\n";
      else
            cout  <<start1 << " "<< end1 << "\n";
      cout << "\n";
      if(start2 == -1){
            cout << "pass";
      }
      else if(start2 == 0)
            cout <<   "Z -> "  << end2  <<"\n";
      else if(end2 ==0)
            cout  <<start2 << " "<< "O"<< "\n";
      else
            cout <<start2<< " " << end2 <<"\n";
      cout << "\n";

      /*
      cout << "last config";
      for(int i=1;i< 25 ;i++)
            cout << " "<< inter_board[i];
      cout << "\n";
      */

}

void entry(int inter_board[],int temp_start1,int temp_end1,int
temp_start2,int temp_end2){

      start1=temp_start1;
      start2=temp_start2;
      end1=temp_end1;
      end2=temp_end2;

      /*
      cout <<"\n"<<"start1 -> " <<start1 <<"  end1 -> " <<end1 <<"
start2 -> " <<start2 <<"end2 -> " <<end2 ;



      cout << " \n final board   \n" ;

      for(int j=1;j<25;j++)
            cout << "  " << inter_board[j];
      cout << "\n";
      */
}

float evaluation(int board[] , int bar_opp, float kill_section[]){
// bar_opp is difference
      float
state[5],pip_count[24],total_pip_count=0,total_pair,total_pegs,final
=0;

      float pip_section[5]={0,0,0,0,0};
      float pair_section[5] ={0,0,0,0,0};
      float pegs_section[5] ={0,0,0,0,0};

      for(int i=1 ;i<=24 ;i++){
            if(board[i] >=1 ){
                  pip_count[i] = (25 -i) *board[i];

                  if(i >= 1 && i <= 6 )
                        pip_section[1] += pip_count[i];
                  if(i >= 7 && i <= 12 )
                        pip_section[2] += pip_count[i];
                  if(i >= 13 && i <= 18 )
                        pip_section[3] += pip_count[i];
                  if(i >= 19 && i <= 24 )
                        pip_section[4] += pip_count[i];

                  total_pip_count = total_pip_count + pip_count[i];
            }
      }

      //cout << total_pip_count;

      for(int i=1 ;i<=24 ;i++){
            if(board[i] >=2 ){

                  if(i >= 1 && i <= 6 )
                        pair_section[1] += 1;
```

```
                if(i >= 7 && i <= 12 )
                     pair_section[2] += 1;
                if(i >= 13 && i <= 18 )
                     pair_section[3] += 1;
                if(i >= 19 && i <= 24 )
                     pair_section[4] += 1;
           }
      }
      total_pair = pair_section[1] +  pair_section[2] +
pair_section[3] +  pair_section[4] ;
      //cout << total_pair;

      for(int i=1 ;i<=24 ;i++){
           if(board[i] >=1 ){

                if(i >= 1 && i <= 6 )
                     pegs_section[1] += 1;
                if(i >= 7 && i <= 12 )
                     pegs_section[2] += 1;
                if(i >= 13 && i <= 18 )
                     pegs_section[3] += 1;
                if(i >= 19 && i <= 24 )
                     pegs_section[4] += 1;
           }
      }
      total_pegs = pegs_section[1] +  pegs_section[2] +
pegs_section[3] +  pegs_section[4] ;

      //cout << total_pegs;



      state[1] = 4*((total_pip_count -
pip_section[1])/total_pip_count) + 3*(pegs_section[1]/total_pegs) +
2*(pair_section[1]/total_pair);
      state[2] = 3*((total_pip_count -
pip_section[2])/total_pip_count) + 2*(pegs_section[2]/total_pegs) +
4*(pair_section[2]/total_pair);
      state[3] = 2*((total_pip_count -
pip_section[3])/total_pip_count) + 3*(pegs_section[3]/total_pegs) +
4*(pair_section[3]/total_pair);
```

```
      state[4] = 3*((total_pip_count - 
pip_section[4])/total_pip_count) + 2*(pegs_section[4]/total_pegs) + 
4*(pair_section[4]/total_pair);


      //cout<< "\nstates   " << state[1] << "  " << state[2] << "  " 
<<state[3] << "  "<< state[4];
      for(int i=0;i<2;i++){
            switch((int)kill_section[i]){
                  case 0:    break;


                  case 1:    state[1] += 1.0;
                             break;
                  case 2: state[2] +=0.6;
                             break;
                  case 3: state[3] += 0.3;
                             break;
                  case 4: state[4] += 0.1;
                             break;
                  default:   break;
            }
      }


      final = state[1] + state[2] + state[3] + state[4];
      //printf("\n    final :: %f\n", final );


      return final;
}



void move(int board[] , int dice1,int dice2, int bar_own , int 
bar_opp){


      float evaluation_value =0 , temp_evaluation_value =0;


      int 
inter_board[25]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
      int inter_start1= -1,inter_start2= -1,inter_end1 = -
1,inter_end2 = -1,inter_bar_opp = 0;
```

```
      int temp_board[25] , temp_temp_board[25],temp_start1 = -
1,temp_start2 =-1 ,temp_end1 = -1,temp_end2 = -1,temp_dice
=0,temp_bar_opp,temp_bar_own;
      float temp_kill_section[2]= {0.0,0.0};

      int dice1_done=0,dice2_done=0;
      int BearedOff =0;
      for(int k=1;k<25;k++)
            temp_board[k] = board[k];

      if(dice2 > dice1){
            temp_dice=dice2;
            dice2=dice1;
            dice1=temp_dice;
      }

      temp_bar_opp=bar_opp;
      temp_bar_own=bar_own;

      int bear_off=0;

      for(int i= 1;i< 18;i++){
            if(board[i]>0)
                  bear_off++;
      }

      if(bear_off == 0 && bar_own ==0){
            for(int i=19;i<=24;i++)
            {

                  if(temp_board[i]>=1 && dice1_done != 1)
                  {
                        if(dice1>(24-i)){
                              temp_start1=i;
                              temp_end1=0;
                              temp_board[i]--;
                              BearedOff++;
                              dice1_done=1;
```

```
                    }
                    else if(temp_board[i+dice1]>=-1)
                    {
                          if(temp_board[dice1+i]==-1)
                          {
                                temp_board[i+dice1]=1;
                                bar_opp += 1;
                          }
                          else
                                temp_board[i+dice1]++;

                          temp_start1=i;
                          temp_end1 = i+dice1;
                          temp_board[i]--;
                          dice1_done=1;
                    }
              }
              if(temp_board[i]>=1 && dice2_done != 1)
              {
                    if(dice2 > (24-i))
                    {
                          temp_board[i]--;
                          BearedOff++;
                          dice2_done=1;
                          temp_start2 =i;
                          temp_end2=0;
                    }
                    else if(temp_board[i+dice2] >= -1 &&
dice1_done !=1)
                    {
                          if(temp_board[i+dice2] == -1 )
                          {
                                temp_board[i+dice2]=1;
                                bar_opp +=1;
                          }
                          else
                                temp_board[i+dice2]++;
```

```
        temp_start2= i;
        temp_end2 = i+dice2;
        temp_board[i]--;
        dice2_done = 1 ;

        if(temp_board[i+dice2+dice1]>=-1)
        {
            if(temp_board[i+dice2+dice1] == -1)
            {
                temp_board[i+dice2+dice1] = 1;
                bar_opp +=1;
            }
            else
                temp_board[i+dice2+dice1] +=1;

            temp_start1 = i+dice2;
            temp_end1 = i+dice2+dice1;

            int fg = temp_start1;
            temp_start1 = temp_start2;
            temp_start2 =fg;

            fg = temp_end1;
            temp_end1 = temp_end2;
            temp_end2 =fg;


            temp_board[i+dice2]--;
            BearedOff++;
            dice1_done = 1;
            break;
        }
    }
    else if(temp_board[i+dice2] >= -1)
    {
        if(temp_board[i+dice2] == -1 )
        {
            temp_board[i+dice2]=1;
```

```
                            bar_opp +=1;
                        }
                        else
                            temp_board[i+dice2]++;
                        temp_board[i]--;
                        dice2_done = 1 ;

                        temp_start2 = i;
                        temp_end2 = i+dice2;


                    }
                }
                if(dice1_done == 1 && dice2_done == 1)
                {
                    cout << "break";
                    break;
                }
            }
            inter_start1 = temp_start1;
            inter_end1 = temp_end1;
            inter_start2= temp_start2;
            inter_end2 = temp_end2;

            cout << "start1->" ;
            cout <<inter_start1 << '\n';
            cout << "start2->" ;
            cout <<inter_start2 << '\n';

        }
        else if(bar_own >0){

            while(bar_own !=0   && (dice1_done !=1 || dice2_done
    !=1)){
                    // printf("yes\n");
                if(temp_board[dice1] >= -1 && dice1_done != 1) {

                    // printf("yes1\n");
            }

            if(temp_board[dice1] == -1){
                  temp_bar_opp++;
                  temp_board[dice1]= 1;
                  temp_kill_section[0] = 1 ;
            }
            else{

                  temp_board[dice1]++;
            }

            temp_start1 = 0;
            temp_end1 = dice1;
            bar_own--;
            dice1_done = 1;
      }

      else if(temp_board[dice2] >= -1 && dice2_done != 1){

            // printf("yes2\n");
            if(temp_board[dice2] == -1){

                  temp_bar_opp++;
                  temp_board[dice2]=1;
                  temp_kill_section[1] = 1 ;
            }
            else{
                  temp_board[dice2]++;
            }

            temp_start2 = 0;
            temp_end2 = dice2;
            bar_own--;
            dice2_done = 1;
      }

}
```

```
          if(dice1_done != 1){
                cout<<"Dice 1 Done\n";

                for(int i=1;i<25;i++){

                      temp_bar_opp=bar_opp;
                      temp_start1= -1;
                      temp_end1 = -1;
                      temp_kill_section[0] = 0;

                      for(int k=1;k<25;k++)
                            temp_board[k] = board[k];

                      if(temp_board[i] >= 1 && i+dice1<25 &&
temp_board[i+dice1] >= -1){
                            if(temp_board[i+dice1] ==-1){
                                  temp_bar_opp++;
                                  temp_board[i+dice1]=1;
                                  temp_kill_section[0] = 1 + ((i-
1)/6);
                            }
                            else

                                  temp_board[i+dice1]++;

                            temp_board[i] --;
                            temp_start1 = i;
                            temp_end1 = i+dice1;
                            break;
                      }
                }
          }


          else if(dice2_done != 1){
                cout<<"Dice 2 done\n";
                for(int i=1;i<25;i++){
                      temp_bar_opp=bar_opp;
```

```
                    temp_start2= -1;
                    temp_end2 = -1;
                    temp_kill_section[1] = 0;

                    for(int k=1;k<25;k++)
                         temp_board[k] = board[k];

                    if(temp_board[i] >= 1 && i+dice2<25 &&
temp_board[i+dice2] >= -1){
                         if(temp_board[i+dice2] ==-1){
                              temp_bar_opp++;
                              temp_board[i+dice2]=1;
                              temp_kill_section[1] = 1 + ((i-
1)/6);
                         }
                         else
                              temp_board[i+dice2]++;

                         temp_board[i] --;
                         temp_start2 = i;
                         temp_end2 = i+dice2;
                         break;
                    }
               }
          }
          inter_start1 = temp_start1;
          inter_end1 = temp_end1;
          inter_start2= temp_start2;
          inter_end2 = temp_end2;
     }

     else{
          for(int i=1;i<25;i++){

               temp_bar_opp=bar_opp;
               temp_start1= -1;
               temp_end1 = -1;
               temp_kill_section[0] = 0;
```

```
                for(int k=1;k<25;k++)
                      temp_board[k] = board[k];
                if(temp_board[i] >= 1 && i+dice1<25 &&
temp_board[i+dice1] >= -1){
                      if(temp_board[i+dice1] ==-1){
                            temp_bar_opp++;
                            temp_board[i+dice1]=1;
                            temp_kill_section[0] = 1 + ((i-1)/6);
                      }
                      else
                            temp_board[i+dice1]++;

                      temp_board[i] --;
                      temp_start1 = i;
                      temp_end1 = i+dice1;
                }



                for(int j=1; j< 25;j++){



                      temp_start2= -1;
                      temp_end2 = -1;
                      temp_kill_section[1] = 0;



                      for(int h=1;h<25;h++)
                            temp_temp_board[h] = temp_board[h];


                      if(temp_temp_board[j] >= 1  && ( (j+dice2) <
25)&& temp_temp_board[j+dice2] >= -1){


                            if(temp_temp_board[j+dice2] ==-1){
                                  temp_bar_opp++;
                                  temp_temp_board[j+dice2]=1;
                                  temp_kill_section[1] = 1 + ((j-
1)/6);
```

```
                        }
                        else
                              temp_temp_board[j+dice2]++;

                        temp_temp_board[j] --;
                        temp_start2 = j;
                        temp_end2 = j+dice2;
                  }

                  if(temp_start1 != -1 && temp_start2 != -1){
                        int x  =temp_bar_opp - bar_opp;
                        temp_evaluation_value = 
evaluation(temp_temp_board,x,temp_kill_section);
                        //temp_evaluation_value =0;


                        if(temp_evaluation_value >= 
evaluation_value){
                              evaluation_value 
=temp_evaluation_value;
                              for(int k=1; k<25;k++)
                                    inter_board[k] = 
temp_temp_board[k];
                              inter_start1= temp_start1;
                              inter_start2= temp_start2;
                              inter_end1 = temp_end1;
                              inter_end2= temp_end2;
                              inter_bar_opp = temp_bar_opp;
                              //cout << "\n start1-> " << 
inter_start1 << " end1-> "<<inter_end1<<" start2-> "<< 
inter_start2<<" end2->  " << inter_end2 << "  value ->"<< 
temp_evaluation_value;
                        }
                  }
            }
      }


      //reverse
      temp_dice=dice2;
      dice2=dice1;
      dice1=temp_dice;
```

```
          for(int i=1;i<25;i++){

               temp_bar_opp=bar_opp;
               temp_start1= -1;
               temp_end1 = -1;
               temp_kill_section[0] = -1;

               for(int k=1;k<25;k++)
                    temp_board[k] = board[k];

               if(temp_board[i] >= 1 && i+dice1<25 &&
temp_board[i+dice1] >= -1){
                    if(temp_board[i+dice1] ==-1){
                         temp_bar_opp++;
                         temp_board[i+dice1]=1;
                         temp_kill_section[0] = 1 + ((i-1)/6);
                    }
                    else
                         temp_board[i+dice1]++;

                    temp_board[i] --;
                    temp_start1 = i;
                    temp_end1 = i+dice1;
               }


               for(int j=1; j< 25;j++){


                    temp_start2= -1;
                    temp_end2 = -1;
                    temp_kill_section[1] = -1;


                    for(int h=1;h<25;h++)
                         temp_temp_board[h] = temp_board[h];
```

```
                      if(temp_temp_board[j] >= 1  && ( (j+dice2) <
25)&& temp_temp_board[j+dice2] >= -1){

                           if(temp_temp_board[j+dice2] ==-1){
                                temp_bar_opp++;
                                temp_temp_board[j+dice2]=1;
                                temp_kill_section[1] = 1 + ((j-
1)/6);
                           }
                           else
                                temp_temp_board[j+dice2]++;

                           temp_temp_board[j] --;
                           temp_start2 = j;
                           temp_end2 = j+dice2;
                      }

                      if(temp_start1 != -1 && temp_start2 != -1){

                           int x  =temp_bar_opp - bar_opp;
                           temp_evaluation_value =
evaluation(temp_temp_board,x,temp_kill_section);
                           //temp_evaluation_value =0;

                           if(temp_evaluation_value >=
evaluation_value){
                                evaluation_value
=temp_evaluation_value;
                                for(int k=1; k<25;k++)
                                     inter_board[k] =
temp_temp_board[k];
                                inter_start1= temp_start1;
                                inter_start2= temp_start2;
                                inter_end1 = temp_end1;
                                inter_end2= temp_end2;
                                inter_bar_opp = temp_bar_opp;
                                //cout << "\n start1-> " <<
inter_start1 << " end1-> "<<inter_end1<<" start2-> "<<
inter_start2<<" end2->  " << inter_end2 << "  value ->"<<
temp_evaluation_value;
```

```
                        }
                    }
                }
            }
        }
        if( temp_start1 == -1 && temp_start2 == -1){
            // printf("yes\n");
            for(int i=1;i<25;i++){
                temp_bar_opp=bar_opp;
                temp_start1= -1;
                temp_end1 = -1;
                temp_kill_section[0] = -1;

                for(int k=1;k<25;k++)
                    temp_board[k] = board[k];

                if(temp_board[i] >= 1 && i+dice1<25 &&
    temp_board[i+dice1] >= -1){
                    if(temp_board[i+dice1] ==-1){
                        temp_bar_opp++;
                        temp_board[i+dice1]=1;
                        temp_kill_section[0] = 1 + ((i-1)/6);
                    }
                    else
                        temp_board[i+dice1]++;

                    temp_board[i] --;
                    temp_start1 = i;
                    temp_end1 = i+dice1;
                    break;
                }
            }

            for(int i=1;i<25;i++){
                temp_bar_opp=bar_opp;
                temp_start2= -1;
                temp_end2 = -1;
                temp_kill_section[0] = -1;
                for(int k=1;k<25;k++)
                     temp_temp_board[k] = temp_board[k];

                if(temp_temp_board[i] >= 1 && i+dice2<25 &&
temp_temp_board[i+dice2] >= -1){
                     if(temp_temp_board[i+dice2] ==-1){
                          temp_bar_opp++;
                          temp_temp_board[i+dice2]=1;
                          temp_kill_section[0] = 1 + ((i-1)/6);
                     }
                     else
                          temp_board[i+dice2]++;

                     temp_temp_board[i] --;
                     temp_start2 = i;
                     temp_end2 = i+dice2;
                     break;
                }
           }

           inter_start1 = temp_start1;
           inter_end1 = temp_end1;
           inter_start2= temp_start2;
           inter_end2 = temp_end2;
      }
      entry(inter_board,inter_start1,inter_end1,inter_start2,inter_e
nd2);
}

int main(int argc, char **argv){
      int board[25];
      int dice1,dice2;
      int bar_own=0,bar_opp;
      char x[30];

      //   int kill_section[2]={0,0};
```

```
	for(int i=1;i<25;i++)
		cin >> board[i];

	scanf("%s",x);
	if( strcmp(x,"e") == 0)
		bar_own =0;

	else{
		for(int  i=0;i<30;i++){
			if(x[i] == 'a'){
				bar_own ++;
			}
		}
	}
	cin >> dice1 >> dice2 ;
	bar_opp =0;
	/*
	cout << "x ==  " << x ;
	cout << "  initial board   \n" ;
	for(int j=1;j<25;j++)
		cout << "  " << board[j];
	cout << "\n";
	cout << "  dice1   " <<dice1<< "  dice2   "  << dice2<< "
bar_own   "  << bar_own<< "  bar_opp   "  << bar_opp;
	cout << "\n";
	*/
	//float temp_evaluation_value =
evaluation(board,bar_opp,kill_section);
	move(board,dice1,dice2,bar_own,bar_opp);


	print_state();
	/*
	cout << " \n initial final board   \n" ;
	for(int j=1;j<25;j++)
		cout << "  " << board[j];
	cout << "\n";
	cout << "  dice1   " <<dice1<< "  dice2   "  << dice2<< "
bar_own   "  << bar_own<< "  bar_opp   "  << bar_opp;
```

```
    cout << "\n";
    */

    return 0;
}
```

References

[1] Berliner, H. J.

Some Necessary Conditions for a Master Chess Program.

In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence,* pages 77-85. 1973.

[2] Berliner, H. J.

On the Construction of Evaluation Functions for Large Domains.

In *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, pages 53-55. 1979.

[3] Berliner, H. J.

Backgammon Computer Program Beats World Champion.

Artificial Intelligence 14(2):205-220, September, 1980.

[4] Berliner, H. J., & Ackley, D. H.

The PROGRAM System: Generating Explanations from a Non-Discrete Knowledge Representation.

In *Proceedings of the National Conference on Artificial Intelligence.* AAAI, 1982.

[5] Davis, R., Buchanan, B., & Shortliffe, E.

Production Rules as a Representation for a Knowledge-Based Consultation Program.

Artificial Intelligence 8,1977.

[6] Doyle, J., & London, P.

A Selected Descriptor-Indexed Bibliography to the Literature on Belief Revision.

SIGART Newsletter (71):7-23, April, 1980.

[7] Haley, P., Kowalski, J., McDermott, J., & McWhorter, R.

PTRANS: A rule-based management assistant.

Technical Report, Carnegie-Mellon University Department of Computer Science, 1983.

In preparation.