Pattern Knowledge and Search: The SUPREM Architecture

Hans Berliner & Carl Ebeling [1]

January, 1988

CMU-CS-88-109

Abstract

We present a new problem solving architecture based upon extremely fast search and pattern recognition. This architecture, which we have named SUPREM, has been implemented in the chess machine/program Hitech, and has proven to be very successful. We describe the implementation in Hitech and the reasons for its success, and compare the SUPREM architecture to other well known problem solving architectures.

Certain interesting phenomena have become exposed as the result of our work with Hitech. The most important of these is that: "The further a process can look ahead, the less detailed knowledge it needs." We have also found that patterns can be formulated for quite complex problems in relatively small pattern recognizers, by eschewing generality and concentrating on likely patterns and their redundancies.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory Under Contract F33615-84-K-1520, and by an IBM graduate student Fellowship.

[1] Current Address:Department of Computer Science, FR-35, University of Washington, Seattle, WA 98195

1 Introduction

Problems in decision making are difficult because, in any problem worthy of being deemed a problem, there are either a large number of alternatives, or the implications of any number of alternatives can create trees of implications that are huge. Thus there could be many millions of leaf nodes in a search, all of which must be evaluated with respect to many factors.

An immediate quandary presents itself: The more knowledge one applies to each leaf node, the slower the process goes. The slower the process, the fewer the alternatives that can be investigated. In the past, this has led to a schism between animate and machine searchers: the former opting for the few nodes, deep understanding approach, and the latter for many nodes but shallow understanding.

Humans solve problems in a "knowledge-intensive" mode, applying small amounts of search when necessary. The human strategy is flexible and avoids the need to encode *all* knowledge. Many successful AI systems mimic the human style and rule-based systems offer the prime example. Competitive gaming systems typically employ the opposite scheme, relying primarily on search.

While each approach has its advantages, their strategic asymmetry has drawbacks, too: Relying primarily on either one eventually makes continued performance gains difficult. Figure 1 illustrates the tradeoff [2]. For a given task, the constant-performance curves exhibit an approximately hyperbolic shape. Consider a system positioned at point A: Making it "smarter" can increase overall performance to point B while slightly reducing search speed. Further increasing its knowledge could lead to point C, which would require enough extra processing to degrade overall performance. In a system far out along the search speed axis, at point D, the performance gradient nearly parallels the knowledge axis and still faster searching yields only modest performance gains. Increasing the knowledge used by the search, however, could be expected to yield large payoffs. The question is how to do this without slowing the search.

To date, most competent AI systems occupy knowledge/search regions near one axis or the other and further performance advances come slowly. For knowledge-intensive rule-based systems, complex interactions within the knowledge-base make adding still more knowledge increasingly difficult. The situation is a common one: Both humans and rule-based systems improve slowly when they know much and search little. For high-performance systems that rely on large searches, providing additional search capability usually means waiting for hardware advances.

The deep understanding approach has been adopted by rule-based systems where human expertise in the field is captured within a knowledge base. This knowledge is then applied to the problem in much the same way as an expert would apply it. While this approach has performed reasonably well in certain domains, in the arena of chess playing programs where the criteria for performance are quite precise and the domain extremely varied (10^{43} possible positions), deep understanding has not produced programs that play better than mediocre chess.

Using a serial machine for searching requires one to determine the best balance between deep and shallow searching. It is possible to use a large number of serial machines to split up the task of investigating the space. However, this runs afoul of just how many machines are available, how they can be tied together, and how they can share results. The latter consideration can have a dramatic effect on the efficiency of the search process. Typically, speed-ups of more than the square root of the number of processors are difficult to achieve in chess.

Experience with chess programs has shown that deep searching solves many of the problems that a shallow search with deep understanding has difficulty mastering. But deep search still requires a competent evaluation function to correctly evaluate the leaves of the search. While in theory the evaluation could be a very complex, open-ended procedure, involving changes of focus from global to local and back to global again, in most practical situations it is quite satisfactory merely to be able to apply what would pass as a first-cut approximation by a true human expert in the field. The problem is that such an expert would still understand a great amount, and this means that the evaluation of a single point in the search space can take so long that the effect of looking at millions of alternatives is to make the task intractable. The SUPREM

[2] Michie makes a similar point in [5]

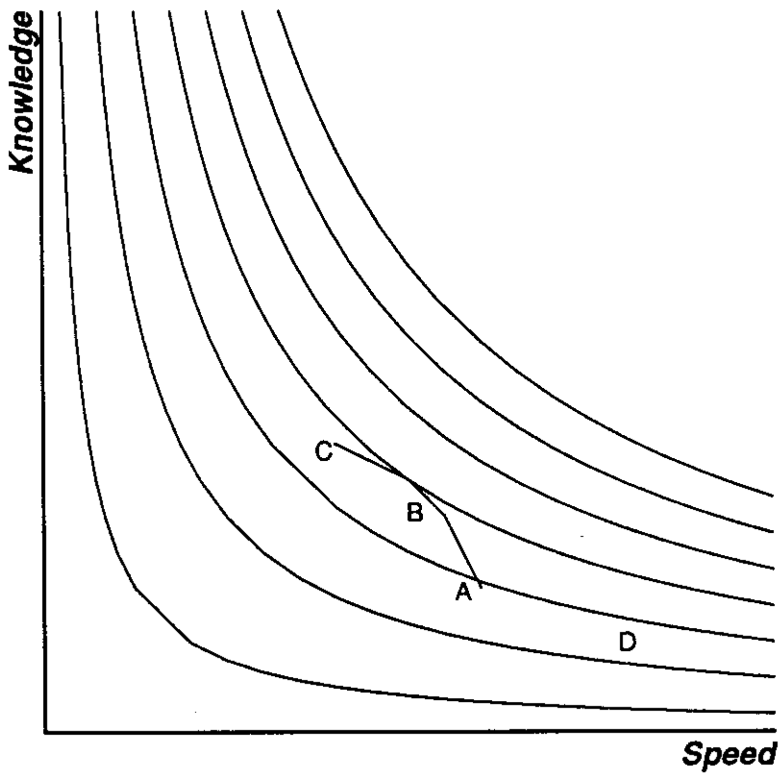

Figure 1: *The knowledge/search performance tradeoff*

(Search Using Pattern Recognition as the Evaluation Mechanism) architecture offers a way of applying high quality knowledge very cheaply.

2 The SUPREM Architecture

The system architecture has four major parts, as Figure 2 shows:

- The Oracle is SUPREM's primary knowledge repository and has all knowledge the system needs to operate. Since the knowledge is domain-dependent, each domain requires a unique Oracle. The Oracle has production-like rules which define interim goals for both sides and the patterns that are needed to recognize achievement of these goals during the search.
- The Searcher is given a search task by the Oracle. It then invokes the Move Generator and the Evaluator to do the search and evaluate the leaf nodes of the search.
- From a given domain state, the Move Generator generates subsequent states extremely rapidly by considering, in parallel, all feasible operators.
- The Evaluator assigns numerical values to each reached domain state by comparing selected parts of the state — as patterns of state components — to pre-tabulated patterns in its memory.

The four parts are organized into two separate subsystems. The Oracle is usually a general-purpose computer while the other three components comprise a special-purpose processor.

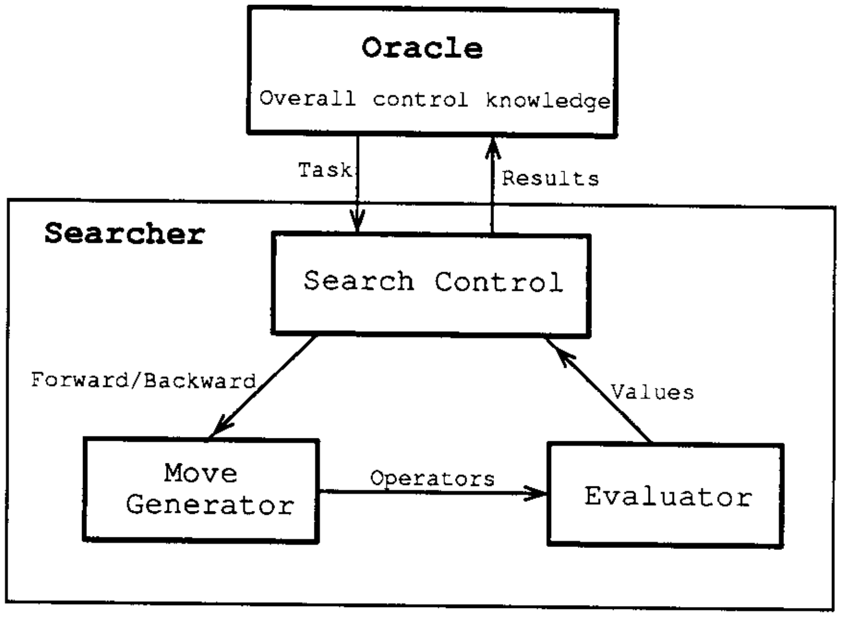

Figure 2: *The SUPREM architecture*

2.1 Operation of SUPREM

The Oracle has rules that define the state-classes in the domain, and the knowledge required to properly evaluate the states likely to be reached by a search starting from the current state. Knowledge takes the form of patterns plus conditions under which the pattern knowledge can be applied. This pattern-based strategy gives SUPREM its name.

The Oracle is smart but slow, and somewhat resembles a rule-based system, looking for those patterns that are most likely to be useful in the evaluation process. To do this well, the Oracle must be reasonably knowledgeable in the problem domain. At the start of each search iteration, the Oracle analyzes the problem and selects the appropriate knowledge. It then compiles the patterns into a form that can be evaluated by a simple table lookup and downloads the compiled patterns into units within the Evaluator that perform pattern recognition and evaluation. This strategy avoids the recomputation of pattern values during the evaluation process, and so significantly speeds the evaluation.

After downloading the patterns, the Oracle selects the depth of search and delegates control to the Searcher. Search then proceeds as follows:

1. At each new node, the Move Generator proposes the best operator to try next.
2. The Searcher broadcasts the candidate operator on the bus so that *Pattern Recognizing and Evaluation units: "recognizers"*, and any other state-maintaining units, can update their individual state descriptions.
3. The Evaluator scores relevant state features via independent recognizers that consult their internal pattern tables for values. One key advantage over other searching systems derives from SUPREM's use of numerous recognizers operating in parallel. The Evaluator combines recognizer outputs in an adder tree and returns the new state's value.
4. The Searcher decides whether to continue along the current branch or backtrack to some earlier point. Control recycles to step 1.

When the Searcher completes the specified search, control returns to the Oracle. The Oracle then decides whether to probe further or accept the solution returned.

2.2 SUPREM Applied to Chess

The SUPREM architecture is realized in Hitech, Carnegie-Mellon University's chess machine/program. A Sun-3 workstation is the host computer and contains the Oracle, which is written in C and consists of about 20,000 lines of code. About 40% of the host software is the Oracle program which contains pattern selection information and the pattern library. The pattern library encompasses approximately 40 downloadable patterns. The remainder of the host software controls the chess program, and maintains the top-level activity representation, an "opening book", and user interface.

The custom hardware in the Searcher can process approximately 175,000 chess positions/sec. This includes generating a move, and incrementally maintaining all representations and the values that these generate, and retracting all these when the time comes to back-track in the search.

There are 14 general recognizers in Hitech. These are units that are capable of detecting patterns that encompass 36 bits. Recognizers are used to detect the presence of partial-board patterns. Each recognizer has a set of patterns for which it is responsible. Included in the armory of a recognizer are tables for detecting the presence of the patterns, and tables or interpreting the meaning of detected patterns. The interpretation of a pattern may involve assigning a value to it, or mapping it into a class of patterns, so it can be combined with other patterns that have been detected to make larger patterns. There are three levels of interpretation in our recognizers, although there is nothing fixed about this. It would seem that the larger the recognizers and the more levels of mapping, the better. Our design was influenced primarily by cost.

There are also 8 Global-State recognizers. Global-State recognizers monitor certain variables that may be crucial to deciding whether a certain pattern is meaningful, or determining the degree of meaningfulness. These recognizers feed their values into the final interpretation of any general recognizer, and are intended to provide global context. Details of this can be found in Section 6.3.

The Oracle analyses the root position to be processed, and determines the type of knowledge that is most appropriate. Since recognizers are a finite resource, only a fraction of the available patterns can be downloaded, and it is the task of the Oracle to do this effectively. The Oracle determines the stage of the game and other salient features of the root position in order to make its decisions. Once the best set of patterns have been identified, they are downloaded into the appropriate recognizers, and the search can begin.

3 Hitech Functional Description

Hitech, like most successful chess machine/programs executes an α-β *iterative-deepening* search [7]. This *full-width* search involves searching all alternatives at a node except those that can be mathematically eliminated as having nothing to do with the solution. The search must know what the complete set of alternative moves is at any stage in the search, and keep track of those already tried as well as those remaining to be tried.

Iterative deepening involves doing a complete search to depth N, followed by a complete search to depth N+1, followed by another to depth N+2, and so on as time allows. It has been shown that this method does not waste time, as may at first appear. Its efficiency results from the information that is computed during each iteration and then saved to be available during succeeding iterations. One item of information saved is the best move in each position as remembered from the last time the position was visited. Since the effectiveness of alpha-beta is very much dependent on the order in which moves are tried, it is an important advantage to try the likely-best moves first. Another item of information is the value of the sub-tree below each node and the depth to which it was searched. If the identical position should be found again in the tree, it may be possible to avoid searching it altogether.

We now describe in detail how SUPREM is realized in Hitech, starting with how the Searcher is organized.

8	BQ				RQ			
7		BQ			RQ			BQ
6			BQ	N	P RQ	N	BQ	
5			N	KP BQ	KP RQ	KP BQ	N	
4	RQ	RQ	RQ	K RQ	*	K RQ	RQ	RQ
3			N	KP BQ	KP RQ	KP BQ	N	
2			BQ	N	P RQ	N	BQ	
1		BQ			RQ			BQ
	a	b	c	d	e	f	g	h

Figure 3: *The set of ever-possible moves to the single square* e4.

4 The Searcher

4.1 The Move Generator

The speed of the operator that provides the means to move from one point in the search space to the next places an upper bound on the size of the search space that can be examined. In chess this is often the factor that limits the search size since the rules that govern how pieces may move are rather complicated.

The move generator also plays a crucial role in the speedup obtained by the α-β algorithm. It can be shown that optimal efficiency is achieved if the best move is examined first at each node in the search tree[4]. Moreover, at many nodes of the tree only one move needs to be generated if it provides a refutation. Thus the order in which moves are generated has a great effect on the speed of the search. Unfortunately, typical software move generators are forced to generate all moves since it is difficult to determine *a priori* which move to generate first.

4.1.1 A Fast Parallel Circuit for Move Generation

All previous software and hardware move generators have been concerned with only the moves of the pieces that are actually on the board in a given position. Software move generators generally compute the entire set of moves in the most convenient order and then sort them according to some set of heuristics. In contrast, Hitech's move generator examines in parallel every move that could *ever* be possible, and computes the subset of those moves that are actually legal in the current position. It then selects what it projects to be the best move from this set of legal moves one at a time as required.

We call the set of moves that could ever be possible the *ever-possible* moves, which can be described as the set of triples {piece, origin, destination} in the cross product (Piece × Square × Square) that are allowed by the rules of chess. This set can be enumerated by examining all the moves that could ever be made by each piece in turn. For example, one can place the queen on each of the 64 squares in turn, listing those destination squares to which the queen could move on an empty board. Another way to enumerate this set is to examine each square in turn and list all the moves that can be made to that square by the different pieces. We have chosen the latter method. Figure 3 shows the ever-possible moves to the square e4.

An ever-possible move is actually legal in a position if the following three independent conditions are satisfied:

- The *origin* condition: The appropriate piece must be present on the origin square.
- The *destination* condition: The destination square must either be empty or be occupied by an opponent's piece.

- The *sliding* condition: For sliding moves (queen, rook, bishop and two square pawn moves), the squares between the origin and destination square must be empty.

As an example, consider the move bQ/b4–e4. For this move to be legal, the black queen must first be on the square b4, the square e4 must be empty or contain a white piece, and the squares c4 and d4 must be empty.

There is a fourth condition that the move generator ignores: The player's king must not be in check after the move is made. This condition is computed after making the move by checking whether the king is attacked. The moves generated by the first three conditions are often called *pseudo-legal* moves; we will continue to call them legal with the understanding that the fourth condition is checked elsewhere.

The computation required to decide whether each of the ever-possible moves is legal is straightforward, requiring only a handful of gates operating on information about the state of the squares affecting the move. A parallel move generator comprised of one of these simple circuits for each ever-possible move can compute the entire set of legal moves very quickly in parallel. A key observation made years ago by Allen Newell is that there are only about 4000 ever-possible moves for each side. Although this still represents a large circuit, VLSI technology can make this kind of parallel solution attractive.

4.1.2 Computing the Ever-Possible Moves

Each ever-possible move can be thought of as a pattern to be recognized. This pattern involves some small subset of the board state and recognizing the pattern instance that corresponds to a legal move is straightforward. Thus the move generator really comprises a large number of pattern recognizers, one for each ever-possible move. Moreover, these pattern recognizers can all fire in parallel to determine the set of legal moves.

Each move recognizing circuit operates on a subset of the state variables that represent the board position, producing as output a boolean value indicating the legality of a single move. The straightforward implementation of the state variables is an 8 by 8 array representing the board, where each array location is a 4-bit value encoding the piece occupying that square of the board. The state variables are easily maintained incrementally as the game state changes. With the exception of the relatively rare cases of castling and *en passant*, moves affect only two squares and thus two writes into the array are sufficient to update the state variables when a move is made or backed up[3]. Each of these writes is called a *halfmove* and either places or removes a piece on a square.

The problem with this implementation is the communication of the state variables to the move recognition circuits. Since each of the 4000 move recognition units requires about 5 inputs, on the order of 20,000 wires are required for communicating the state variables to the recognition circuits. One of the characteristics of VLSI circuits is the high cost of communication. This comes about because the size of the active devices has been reduced to the point that the wires between adjacent devices require as much, or more, area than the devices themselves. If signals are required to go any distance at all, the space they consume can overwhelm that used by active circuitry. Moreover, the delay attributed to long wires can dominate the gate delay for VLSI circuits. This disparity is even more pronounced if signals are required to cross chip boundaries.

The key idea of the move generator architecture that permits a reasonable VLSI implementation is that of duplicating the state variables throughout the move computation circuits so that the inputs to those circuits are available where they are used instead of being communicated through many different wires. Of course the state variables themselves must be maintained as moves are made, but this requires far less communication. Only 10 wires are required to communicate the address and data when writing the state variable array. While these 10 wires must be routed throughout the circuit to all state variables, this can be done with a regular layout to minimize wiring space. This transformation in the circuit is represented in Figure 4.

There are some important points to be made about this circuit transformation. First, it makes a single chip implementation more feasible by drastically reducing the communication between circuit elements on

[3]Moves made when extending a branch of the tree are reversed when the search back-tracks. An alternative solution would be to maintain the state variables in a stack.

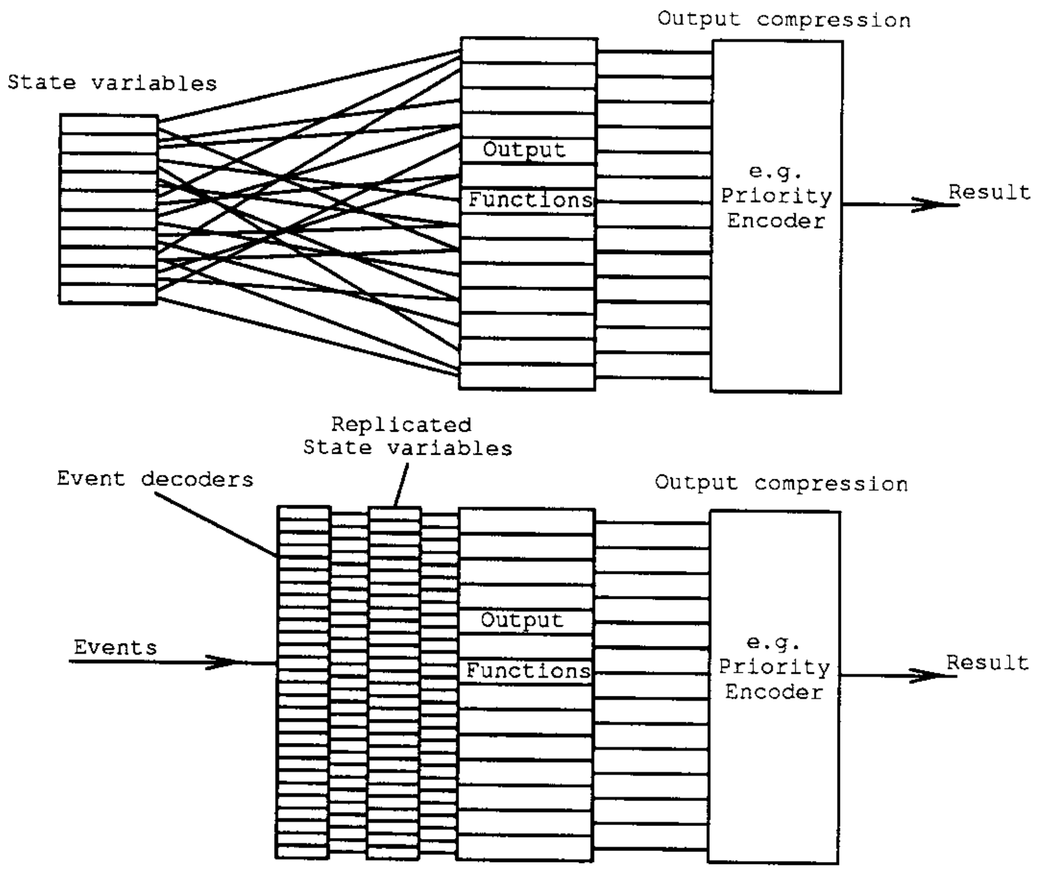

Figure 4: *The tangled wiring network that carries the state variables to the function circuits in the first circuit is replaced by a single event bus that communicates the change in the state variables. Copies of the state variables used by each function are kept within the function circuit itself.*

the chip. Second, if the entire circuit is too large to fit on one chip, the circuit can be partitioned almost arbitrarily onto several chips. This is in contrast to many parallel circuits that are constrained by the communication pattern to an all-or-nothing approach where the circuit is not feasible if it does not fit on a single chip. This also allows the implementation to track advances in technology. While the initial implementation was done using 64 chips, it could be done with just 4 chips using current technology.

One can view the move generator as a write-only memory where the memory output is a set of functions of the memory state. The memory is not the usual one, however. A single write instruction modifies many different memory locations, and since a single memory value is used by many different functions, each location is replicated in each function circuit to minimize the amount of wiring required. One can also view move generation as a large pattern recognition problem for which there are 8000 possible features (4000 moves for each side) to be recognized in a board position. Each of the 8000 circuits acts as a feature recognizer that looks for one particular pattern corresponding to one ever-possible move. As the board changes, the information about the change is broadcast to all the recognizers, each of which determines whether a feature is present or absent.

It is the ability of this architecture to perform a large amount of pattern recognition in parallel that gives SUPREM much of its power. Note, however, that in the case of move generation the patterns that are recognized do not change and thus the patterns can be compiled into hardware. In the more general case such as position evaluation, the hardware must be flexible enough to allow the Oracle to program the set of patterns that are relevant.

4.1.3 Move Selection

The move generator as described thus far consists of a large parallel circuit that computes the legality of the approximately 4000 ever-possible moves for each side. There remains the task of isolating the few moves that are actually legal. This process of move selection is extremely important since the order in which moves are tried makes a large difference to the efficiency of the α-β search.

One solution would be to use a suitably modified priority encoder to identify the legal moves in a serial manner. The drawback of this method is that it imposes a static ordering on the set of ever-possible moves. This static ordering cannot take into account dynamic factors such as square safety that can drastically change the value of a particular move.

Static ordering can be done, however, on a square by square basis. That is, the dynamic factors that affect the value of moves affect all moves to one square in the same way. For example, if the opponent is guarding the square, the value of all moves to the square is reduced because of the likelihood that the moved piece will be captured. Thus the move generator is partitioned into 64 parts, one for each square, with a priority encoder used to select the moves to each square without sacrificing search efficiency. In the process of generating moves to a square, the value of each move is estimated by noting the value of the moving piece, the captured piece, if any, and the safety of the destination square. This safety calculation is possible because the parallel move computation circuit actually computes the entire set of legal moves. The moves are then dynamically ordered among squares based on the value associated with each move.

4.1.4 Maintaining the Context of Active Positions

The α-β search traverses the search tree in depth-first order so that at any position the first move is made and its subtree examined before the subtrees of the remaining moves are examined. At about half the nodes in the search tree only one move is examined and thus the move generator produces moves one at a time when required. Thus the context of the move generator must be saved from the time one move is generated until the next move is requested, during which time the move generator is processing the positions in the subtree. The move generator context includes the state variables that represent the board position, which are restored automatically by performing inverse moves when the search backs up the tree, and information about which moves have already been generated in each position. This latter information is kept by each chip in a stack that remembers the most recent move tried from a position. The priority encoder then uses this to generate the next untried move.

4.1.5 Special Operations

The move generator supports several operations that serve to speed up the search algorithm. The first is used during the part of the search that attempts to play out the active components of a position. This *quiescence* search is done after the pre-determined search depth is reached and examines only capture moves and responses to check. A control signal from the chess machine is asserted during the quiescence search which informs the move generator chips to generate only capture moves. This signal is not asserted if the side to move is in check so that all escaping moves are allowed. However, the move generator also is able to avoid most of the illegal moves that arise when escaping check.

Another special operation allows the chess machine to query the move generator about the legality of a particular move. This is used to verify whether a move suggested by another module such as the hash table or killer table is actually legal. This operation can be done very quickly since the move generator has a list of all legal moves.

4.1.6 The Move Generator Chip

The current implementation of the move generator consists of 64 chips, one for each board square. The block diagram for this chip is shown in Figure 5. Besides the move computations, each chip contains a maskable priority encoder, a stack of move indices for representing the current search context, a PLA for

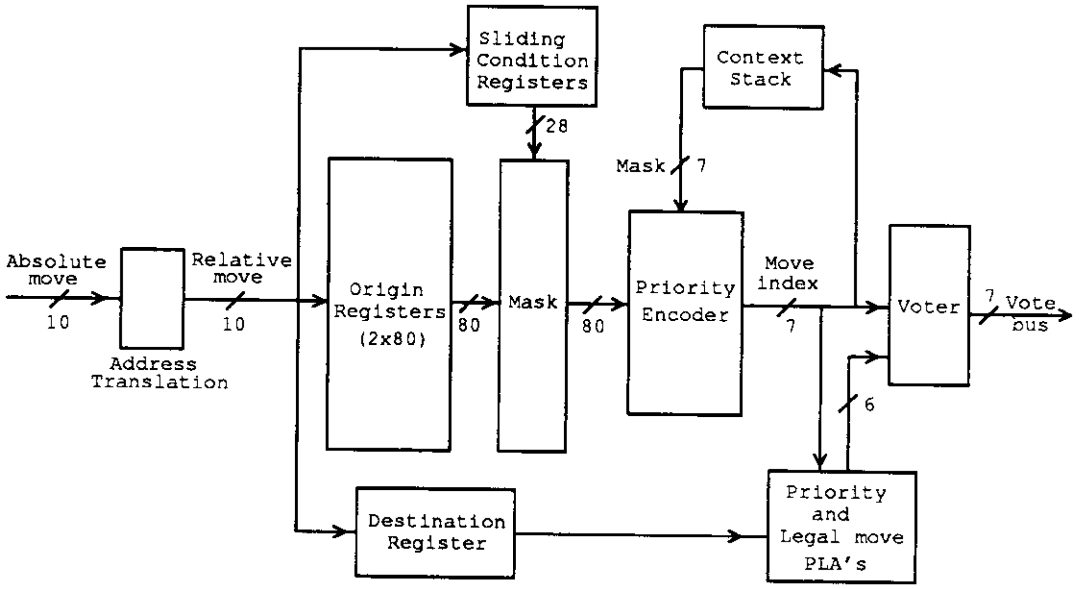

Figure 5: *Move generator chip block diagram.*

performing the dynamic move priority calculation, and a distributed voting circuit that is used in conjunction with the other chips for selecting the move with the highest value. The voting is done by having each chip place the priority of its move on a bus that all chips can examine. When the chip with the highest valued move recognizes that no other chips have a better move, it presents its move. More details of the move generator chip can be found in [3]. The move generator chip was designed so that it can generate the moves to any of the 64 different squares. Each of the 64 chips is assigned to a different square of the board during initialization. Designing just one chip and using 64 copies takes advantage of the relatively inexpensive replication costs of VLSI chips. In so doing, there are some inefficiencies, especially at the edge of the board where there are fewer legal moves than at the center. While this particular implementation was convenient for prototyping the design, the move generator architecture allows a range of implementations using fewer chips or even wafer-scale integration to decrease board area, power consumption and delay. This architecture is thus suitable for a variety of technologies with different circuit densities.

5 Position Evaluation

The third component of the classic chess program that goes along with the search control and move generation is position evaluation. In theory, the search terminates only at nodes whose values are win, loss or draw, but such a termination condition cannot be used in practice since complete game trees for most interesting positions consist of at least 10^{100} nodes. The evaluation function is used to approximate the result of a complete search by computing values by which different positions can be compared. Each leaf node, as determined by some *a priori* depth limitation, is assigned a value by the evaluation function to produce an ordering on the set of all leaf nodes with respect to the probable outcome of a complete search from that node. While material is the most important consideration, many more subtle factors arise when comparing positions, since a winning position can be created without any win of material. An evaluation function that does not recognize these *positional* factors is sure to lose, even to an average player, given the search depths presently feasible.

If one examines a typical winning line of play in a game between good players, there is a gradual progression in the value of the evaluation function from an even position to the final win. At first the advantage is subtle with one player having a positional advantage because his pieces are deployed more effectively than his opponent's. As the game progresses, this positional advantage is increased until at some point a player actually establishes a material advantage. Eventually the winning player is able to use his superior forces to engineer a mate. Positions in which the material balance is at stake are called *tactical* while those in which the players are jockeying for a positional advantage are called *positional.*

Computers tend to be very good at tactical play since the material computation is very simple, and to maintain complete accuracy in deep calculation may strain even a human World Champion. It is in computing the subtle positional factors that computers have difficulty. Computers often use their superior tactical ability to save weak positions that were reached because of an inadequate understanding of the positional defects of earlier decisions. The problem is that these positional defects manifest themselves only after many moves so that no reasonable search can hope to discover their eventual effect. As more than one programmer has discovered, the only advantage that a deep search has in these cases is that the positional problem becomes apparent somewhat sooner, with a slightly better chance of survival.

Evaluation, then, comprises many different factors, all of which must be taken into consideration when comparing two positions. While it is possible to carry the result of this evaluation as a vector of values, each representing some component of the evaluation, the values are typically combined into a simple scalar that represents the weighted sum of the components. These weights are assigned according to the relative importance of each component[6]. Choosing the correct weights is difficult because components may be more or less important depending on the exact position. In these cases, a correct evaluation requires that the weights be themselves the result of some "higher-level" evaluation. These dynamically computed weights are called *application coefficients*[1]. The problem of combining the evaluation components into one scalar is difficult even for human players, involving decisions such as what constitutes sufficient positional compensation for some loss of material. How the application coefficients are computed is a matter of judgement and experience.

There is a complex relationship between the evaluation function and the depth of search. As the search is able to see deeper, the evaluation function needs to know less about tactical factors other than simple material. For example, a capture is discovered by a one ply search which knows only about material, the concept of a double attack is discovered by a three ply search, and the threat of a double attack by a five ply search. Thus the evaluation function used by an eight ply search has less need to know explicitly about a double attack than a six ply search. Other positional factors, such as board control and piece mobility, appear to be less important when searching deeply. The key to building an effective evaluation function is deciding which factors are indeed important relative to the power of the search.

There are three related considerations involved with designing an evaluation function. The first is the identification of the knowledge required to understand positions sufficiently well. The level of play of the program ultimately depends on how well the evaluation function can distinguish good and bad positions. Identifying the components that the evaluation must understand is the job of an expert who knows the problem intimately and can identify and correct deficiencies discovered through experience with the program. The second consideration is whether these components can be computed efficiently, and this depends on the complexity of the evaluation and the power of the computational method. Finally, a decision must be made about which evaluation components to include in the final evaluation function. This decision must consider the tradeoff between search speed and the extent to which the evaluation should understand each position. This again is the product of the experience and judgement of an expert but also depends on how efficiently the evaluation can be implemented.

The tradeoff between search speed and knowledge is a classic one: including more knowledge in the search necessarily slows down the search. In computer chess, the emphasis has shifted over the past decade in favor of speed. Since 1980, the fast searchers, Belle and Cray Blitz, have dominated computer chess as against the more knowledgeable programs such as NUCHESS. This reflects the difficulty of encoding the relevant chess knowledge and bringing it to bear efficiently. In many cases, the time required to analyze some complex facet of a position slows the search to the point where the overall play of the program is diminished instead of enhanced. The problem is how to increase the knowledge without decreasing the speed. It is just this ability of Hitech to perform complicated analyses extremely quickly that allows it play at such a high standard.

5.1 Evaluation Complexity

The different components of the evaluation function have widely varying computational requirements. We classify an evaluation function as first-order, second-order or higher-order based on its computational complexity. An evaluation function, $f(S)$, is defined as first-order if, and only if,

$$f(S) = \sum_i g(i, s_i) \tag{1}$$

where i ranges over all the squares of the board in position S and s_i is the piece on square i. In other words, a first-order evaluation can be computed by examining the state of each square of the board independent of the other pieces and squares. Moreover, the overall value is computed as the linear sum of the values of the separate squares. This means that the function f can be computed incrementally during the search. That is, if the difference between two positions S^i and S^f involves only one square, δ, then

$$f(S^f) = f(S^i) - g(\delta, s_\delta^i) + g(\delta, s_\delta^f) \tag{2}$$

since

$$f(S^f) = \sum_{j \neq \delta} g(j, s_j^f) + g(\delta, s_\delta^f)$$

and

$$f(S^i) = \sum_{j \neq \delta} g(j, s_j^i) + g(\delta, s_\delta^i).$$

Once the initial value of the evaluation function is established, computing its value at new positions is accomplished by computing the function g on the s_δ's that describe the difference between neighboring positions in the search. There is a close relationship between the s_δ's used here and the halfmove operators that we have used previously to describe the incremental change from one position in the search to another. A *remove piece* halfmove corresponds to the negative term in equation 2 and a *place piece* halfmove to the positive term. Since most moves affect two squares, a total of four halfmoves are required to move the search from one position to another. If $g(i, \text{NULL}) = 0$, where NULL represents the state in which no piece occupies the square, then only two halfmoves are needed for non-capturing moves, and three for captures.

The prime example of incremental evaluation is that of material evaluation which totals each player's material. In this case the function g is defined simply by

$$g(\delta, s_\delta) = \text{Value}(s_\delta)$$

where values are assigned to each piece based on their relative strengths. By contrast, determining whether a pawn is isolated cannot be done incrementally, since it requires information about the presence of pawns on three adjacent files.

A second-order evaluation function is one that depends on the relationship between two or more squares and thus cannot be computed incrementally. A second-order evaluation function cannot be described by equation 1 and must be described by the more general equation

$$f(S) = g(s_1, s_2, \ldots, s_k), \text{where } k > 1. \tag{3}$$

In the worst case, g may be a function over the entire board, but generally second-order evaluations depend on a subset of the squares. For example, each legal move computation performed by the move generator is a second-order evaluation. The legality of a move depends on the state of the origin, destination and possibly intervening squares. This second-order evaluation operates over a relatively small subset of the board state, which led to the simple parallel architecture based on distributed state described in Section 4.1. We will now show that this architecture can be generalized to perform position evaluation.

5.2 The Role of the Oracle

The Oracle is located in the host computer and cooperates with the Searcher. It is invoked at the start of a search to perform a detailed analysis of the root position to determine:

1. The present state-class of the root, and
2. The state-classes that are likely to be encountered during the search, and which would therefore require evaluation.

The Oracle allows the evaluation to be tuned to the region that the search is likely to cover. This was not required in the case of the move generator since the set of ever-possible moves does not change. The ability to determine the patterns relevant to the search locale greatly increases the efficiency of the pattern recognition architecture.

The Oracle must be both knowledgeable in its domain (chess), and be an excellent resource allocator. Typically, there is more knowledge that could be put into recognizer units than there are units. However, certain simplifying assumptions help here. For instance, once the amount of material for one side drops below a certain threshold, it is unlikely that considerations of opponent's king safety will come into play. This allows the focus to be gradually shifted to the endgame, and recognizers would be assigned such tasks as they are freed from performing other tasks. This type of tracking of what is important allows the same recognizer units to gradually change the patterns they are monitoring, providing a great deal of flexibility with a small number of units.

This flexibility is achieved by generality in the hardware which can receive down-loaded tables at the start of the search to deal with a large variety of situations. There is some risk in using the Oracle to decide what knowledge is to be used. For instance, in one branch of the search many pieces could be swapped off, reaching a very simplified position, far away from the root, for which the designated knowledge would not be suitable. Also, it is possible for material on the board to **increase** due to pawn promotion, and thus king safety could again become a consideration, after having been dismissed as inconsequential in the Oracle analysis. However, such occurrences are very rare.

This effect can be lessened by making the evaluation less dependent on assumptions made at the root. This trades off increased evaluation hardware for more precise evaluation. Given a fixed amount of evaluation, the Oracle is always faced with the decision about which components to include for any one search. This decision is much easier if the hardware is not too cramped.

6 Implementing the Evaluation Function

First-order evaluation is interesting since it is extremely easy to compute, requiring only the specification of the function g in Equation 1. The domain of g is the cross product of the set of pieces and the set of squares. Since there are 12 different pieces and 64 squares, g can be completely specified by a table with 768 entries Moreover, many different first-order evaluations can be combined into one since

$$f_1(S^f) + f_2(S^f) = f_1(S^i) - g_1(\delta, s^i_\delta) + g_1(\delta, s^f_\delta) + f_2(S^i) - g_2(\delta, s^i_\delta) + g_2(\delta, s^f_\delta)$$

and thus

$$g(i,s) = g_1(i,s) + g_2(i,s), \forall i, s.$$

Since it is so inexpensive to compute a first-order evaluation, it is advantageous to cast as much of the evaluation function as possible as first-order evaluation. Although at first glance there appears to be little besides simple material computation that is first-order, second-order evaluations can often be approximated by a first-order evaluation. While the result is less precise than that derived through second-order evaluation, in cases where the second-order evaluation is difficult to compute or not important enough to warrant the extra computation, the first-order evaluation can provide a performance gain if it adds any additional understanding at all since it does not slow down the search.

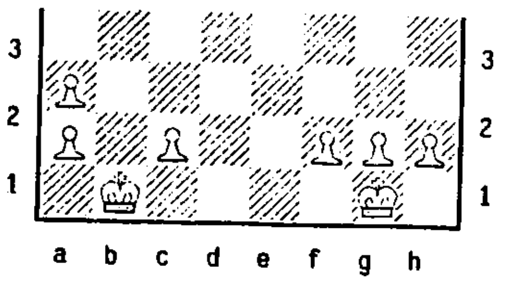

Figure 6: The king Not Safe.....Safe

As an example, consider the problem of king safety. In Figure 6 the king on the right side of the figure is safely hidden from attack behind the three pawns. Removing one of these pawns would put the king in some jeopardy, removing two would expose the king to serious threats, but removing all three would be a disaster. Advancing the pawns one square would weaken the shelter somewhat. This simple king shelter evaluation can be approximated by defining the function g as shown in Figure 7.

16	16	16
25	25	25

Figure 7: *The function g for pawn shelter as defined for the pawns on the six squares in front of the king, where a pawn is worth about 100 points.*

This definition of g gives the pawn shelter at the right side of Figure 6 a bonus of 75 points. Advancing one pawn loses 9 points, losing one pawn costs 25 points, and losing all three costs 75 points. Since the loss of the second and third pawn is much more serious than the loss of the first, a more precise pawn shelter evaluation would adjust the amount each pawn is worth *based on how many of the others are still present.* Moreover, capturing from b2 to a3 (as shown on the left side) is much worse than advancing from h2 to h3, but this cannot be reflected by this first-order evaluation. However, the first-order approximation does give some idea about the value of these pawns, and while it is not as good as second-order evaluation, it is better than no evaluation at all.

Using an Oracle and first-order evaluation to approximate second-order evaluation yields surprisingly good play when used in combination with deep search. In fact, Hitech initially used only incremental evaluation in conjunction with an Oracle, attaining an estimated rating of about 2100. But it would sometimes make serious mistakes because it did not understand some things about chess that involve second-order evaluation that cannot be reasonably approximated by a first-order evaluation controlled by an Oracle. The primary deficiency involved some basic ideas about pawn structure such as doubled and isolated pawns. These concepts, which are crucial to playing high calibre chess, simply cannot be computed without second-order evaluation.

6.1 Computing Second-Order Evaluations

As defined by Equation 3, a second-order evaluation must be expressed as an arbitrary function over the state of more than one square. Such an evaluation can be computed serially, as is usually done, but only at a cost to the search speed. The move generator has the same problem since the legal move computation is a second-order evaluation. This section describes an architecture for the evaluation function that is similar to

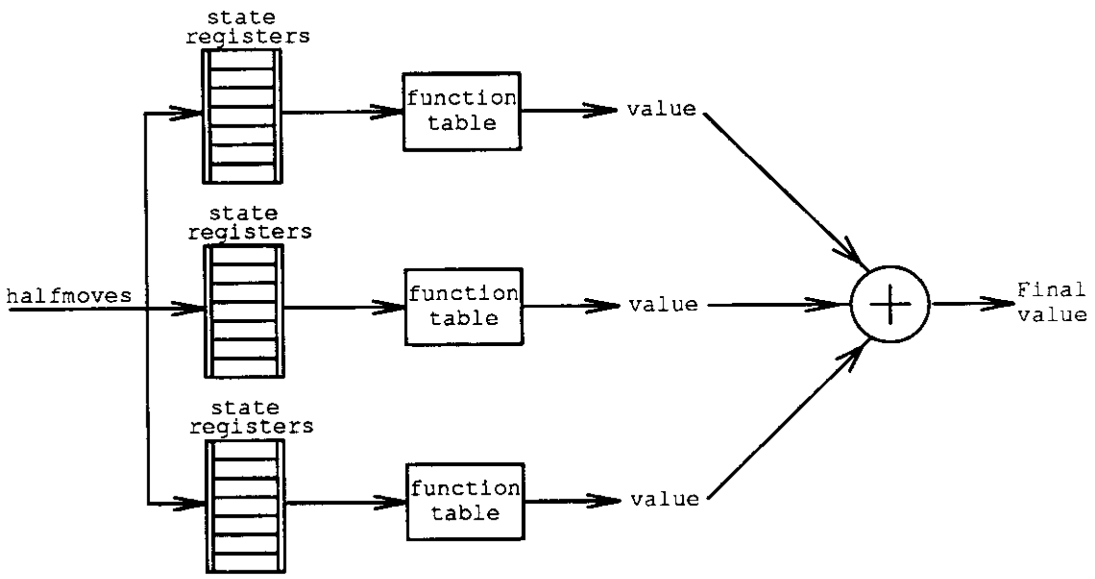

Figure 8: *The overall evaluation function is the combination of the parallel evaluation of independent components.*

that used for move generation. In the case of the move generator, each legal move computation operates on the set of state variables consisting of the origin square, the destination square, plus intervening squares for sliding moves, resulting in a boolean value indicating whether the move is legal. The state information for each move is maintained by a set of registers that are updated incrementally by halfmoves broadcast over the move bus.

In the case of an evaluation function, the computation operates over the state relevant to the particular evaluation component, and the result is a number representing the value of a particular state assignment. If the state over which the function operates is relatively small, then the function can be computed by table lookup. For the king safety example of Figure 6, the state can be represented as six bits, one for each pawn shelter location, and thus a 64 entry table is sufficient to compute the second-order pawn shelter evaluation precisely. The overall evaluation function is divided into a number of different evaluation components such as pawn structure and king safety, each of which is analyzed separately and the results added together to reach a final value. This overall evaluation architecture is summarized in the simple block diagram in Figure 8.

For most second-order evaluation components encountered in chess, the amount of relevant state is too large for simple table lookup to be used as the evaluation method. In these cases, the evaluation is divided into a number of subcomponents that can be analyzed separately and then combined. For example, the evaluation of king safety can be divided into the evaluation of the pawn shelter, the location of the king with respect to the castling privilege, and the attacking opportunities of the opponent. The subcomponents are defined such that the state under analysis is small enough to allow a table lookup evaluation. Moreover, each subcomponent evaluation reduces the total state under consideration by discarding redundant and irrelevant facts. For example, we reduce the pawn shelter state from about 8000 possible categories to only 16 final categories of shelter. A similar reduction takes place with respect to king location and opponent's attacking chances. Finally, all these factors are combined to produce 256 categories of king safety. This allows table lookup to be used to combine the results of the individual subcomponents into a final value. This mapping-down process is illustrated in Figure 9.

The above method amounts to a recursive factoring of the evaluation function into subfunctions that have limited input. Our implementation uses 8K x 8 memories for each mapping unit which limits the subfunctions to 13 bits of state input. The current hardware uses three levels of mapping to reduce a total of

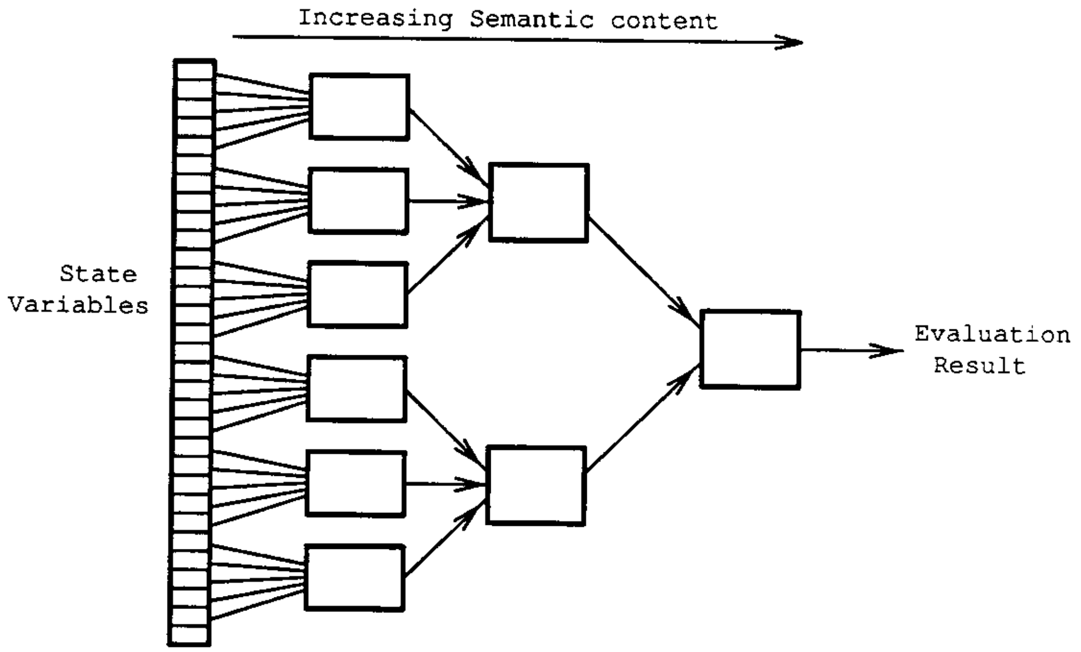

Figure 9: *Complex evaluations are performed in a series of steps that map the overall state down to a final value.*

36 bits of state and context information to an 8-bit value. These tables can be specified before each search which allows the Oracle to decide which evaluation components are most important within the scope of the search.

6.2 The State Variables

Each evaluation component operates over a subset of the entire set of state variables. The representation to be used for this subset must be specified as part of the function specification. As in the move generator, the values of these variables are modified incrementally as the search makes and unmakes moves while traversing the search tree. The state variables are kept in a set of state *accumulators* whose contents are defined and updated using a table of values called the state description table. This table contains an entry for each possible halfmove, indicating the amount to be added to the accumulator when a piece is placed and the amount to be subtracted when the piece is removed.

The entries in the table determine the meaning of each bit in the accumulator. In practice, the accumulator is divided into a number of fields, each representing some relevant set of facts about a position. A field may represent a pattern or a sum. For example, one field may count the number of pawns, another may keep track of the location of the king and another may indicate whether a bishop is on a particular square. The carry chain of the accumulator can be broken at predetermined points in order to keep signed data from overflowing from one field to another. The data that is stored in the state accumulator is not raw, but already encoded via the state description table. By knowing that certain states cannot exist or are not of interest, the number of field entries can be reduced. If we are interested in a pattern of 3 elements, but only whether or not they are all present, it would be wasteful to assign a bit to each element. Instead, one uses the arithmetic power of the accumulators to assign a value of "1" to each element. Then when all are present, the accumulator will indicate "3", and no other value will be of interest. In this way only two bits are used to compute this function instead of three, as would be required by the naive method.

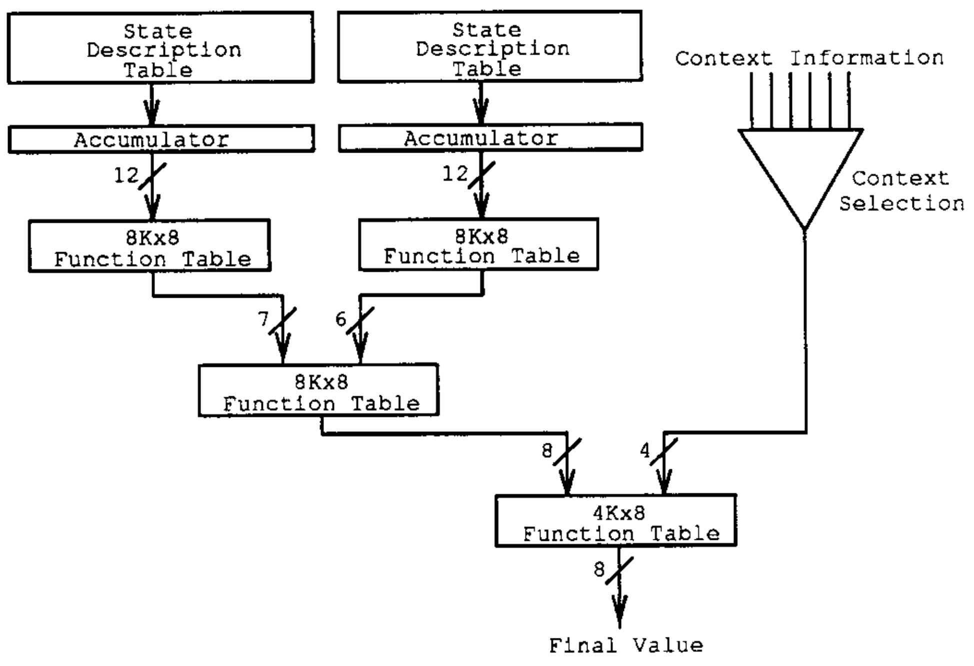

Figure 10: *The design of a single evaluation unit.*

6.3 Application Coefficients and the Role of Context

We have assumed up to this point that each evaluation component can be analyzed independently and the results summed into an overall value. In many cases, the results cannot be combined linearly. That is, there are nonlinear relationships between the components that affect how the results are combined. For example, the relative importance of the king safety component depends on the amount of material remaining. Or if the king is under severe attack, it may not matter much that the pawn structure is strong. This contextual information can be applied via *application coefficients* [1] which specify how much weight a component carries in light of other factors.

We have built into our evaluation hardware the ability to include the results of other evaluation components as part of an evaluation. Up to three results from other evaluation units can be selected with which to perform a final mapping on the component evaluation value. This obviously has the opportunity for feedback paths, but we do not allow this since the time for the circuit to settle would become unbounded. One might note the resemblance of these context connections to the lateral inhibition connections in neural networks.

Figure 10 gives the overall design of one of our second-generation evaluation units. Two levels of mapping are used to compute an intermediate value from the current state, and a final mapping applies selected context information to produce a final value. Hitech's Evaluator contains 14 of these general recognizer units and 6 additional recognizers that are dedicated to pawn structure. Another 8 smaller recognizers are used for computing global state such as total material, material balance, and the number of remaining pawns. This global state provides context whereby the interpretation of the other recognizers can be modified.

This second-generation hardware differs from our first-generation hardware, in size but not basic design. The amount of state has been extended from 20 to 24 bits, the ability to use context information has been added, the number of units has been increased from 8 to 14, and 8 global state recognizers have been added. We should point out that the global state recognizers can be used either to provide context information to the evaluation or to extend the amount of board state used by a recognizer. This gives the new evaluation hardware a great deal more flexibility than the original design.

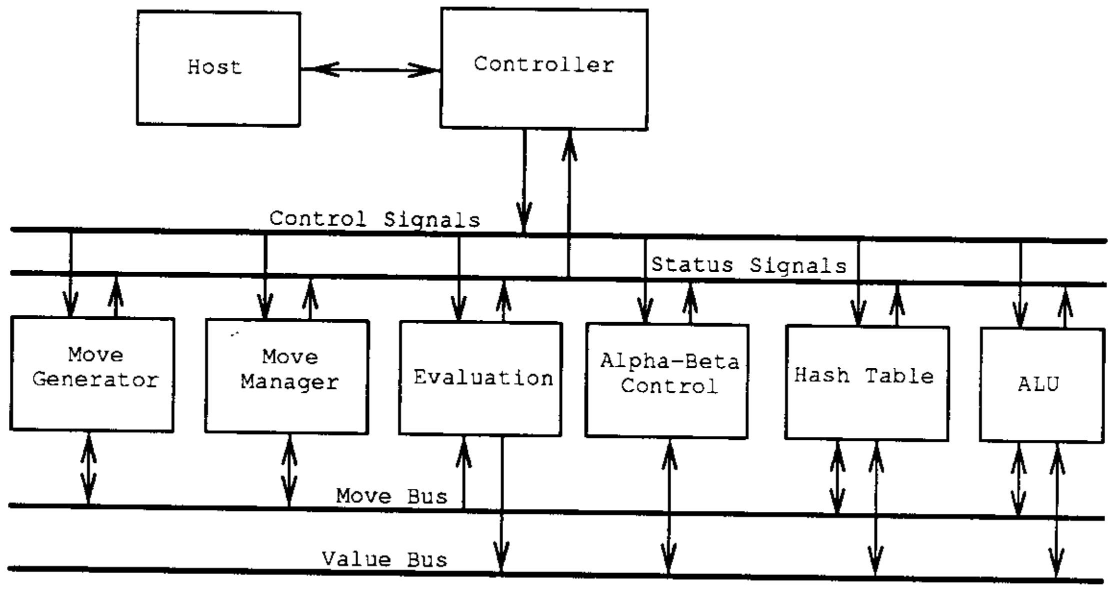

Figure 11: *The chess machine comprises a set of modules that operate under control of a microprogrammed controller and communicate over two shared busses.*

7 The Hitech Chess Machine

The previous sections have described a parallel architecture for move generation and position evaluation, and presented the details of their implementation. These functions must be augmented by other functions such as α-β search control and repetition detection for the program to be fully operational. This section describes the chess hardware and software that makes up the balance of the Hitech chess machine.

The structure of the chess machine, shown in Figure 11, is similar to that used by other special purpose machines such as Belle[2]. A controller runs the α-β algorithm using specialized hardware units to perform the computationally complex tasks of move generation and evaluation. Two data busses connect the chess machine modules. The first is the move bus, which communicates the halfmove operators used to move from one position in the search tree to the next. The second is the value bus, used to communicate values between the evaluation function and those modules that use values to perform the α-β decisions. Each module may generate status signals indicating the presence of specific conditions that are used by the controller to modify the flow of control.

The structure of the controller is shown in Figure 12. Microprogram control flow is specified by a next address field in the microinstruction. Branches are performed by OR'ing selected status signals into four bits of the next address field. By assigning branch targets appropriately, 2-way, 4-way, 8-way and 16-way branches can be performed based on any subset of the four status signals. Microsubroutines are supported by a subroutine return address stack.

The controller interfaces to a host, currently a Sun-3 workstation, via an interface to the host bus. This interface contains several registers that are written and read by the host program to communicate data to and from the chess machine and to initiate execution of microprograms. Data is passed between the host and chess machine via registers connected to the move and value busses. Commands executed by the chess machine are regarded as subroutine calls by the host, where the arguments are written into the move and value bus registers before writing the command register and the results are read back when the operation is complete.

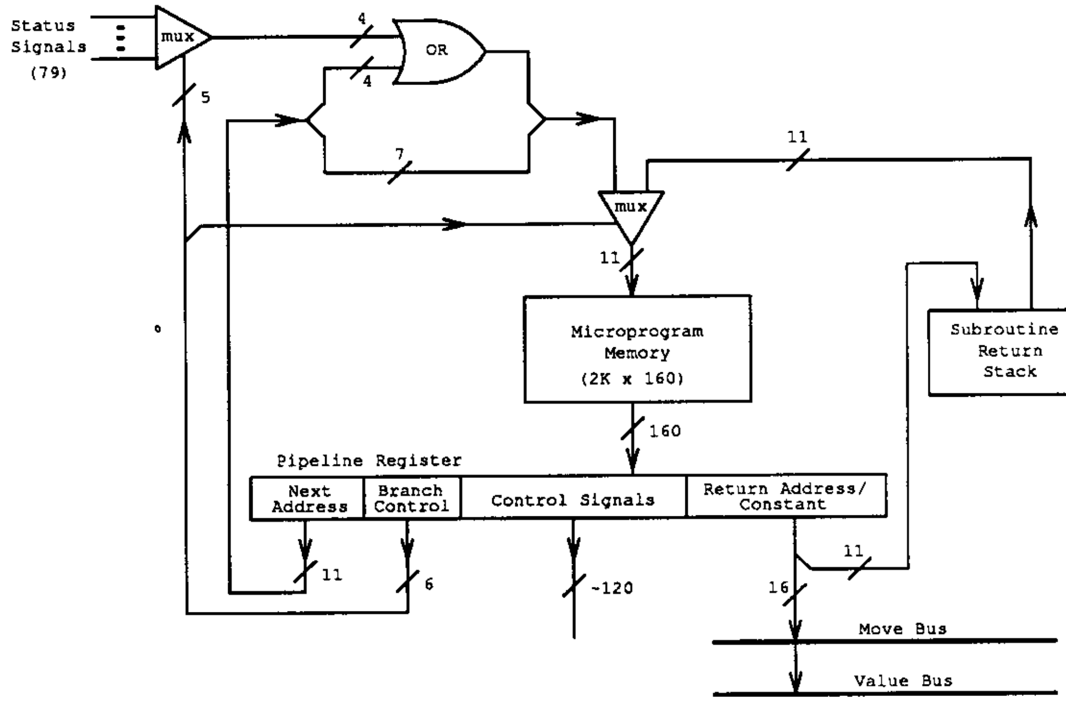

Figure 12: *The chess machine controller.*

7.1 Move Generator

The move generator used by Hitech is built from 64 of the VLSI chips described in Section 4.1, with additional special hardware to generate castling and *en passant* moves, which are not handled by the chips. These special moves are only rarely legal and thus are handled as exceptional conditions by the controller. If one of the special moves is legal, a status flag is asserted which causes control to be diverted to a special routine that generates the corresponding halfmoves. In the case of castling, for example, the hardware asserts a status flag if the player still has the castling privilege and there are no pieces between the king and rook. The controller then queries the move generator to determine whether the opponent controls any of the squares that the king must cross.

Another special case is that of pawn promotion. Although the move generator produces pawn advances to the last rank, it does not perform the actual promotion. This is done by special hardware that recognizes any pawn move to the last rank and raises a status signal to force the controller to intervene and promote the pawn to each of the four possible major pieces in turn.

7.2 Evaluation

Hitech's evaluation is divided into two parts: Incremental, first-order evaluation using a simple table defined by the oracle, and general second-order evaluation performed by the hardware described in Section 6.1. The actual programming of the second-order evaluation hardware is specified using a compiler that allows one to define the functionality of each unit in a straightforward manner. The Oracle determines before each search which evaluation components to include. This decision is based on the number of available units, the phase of the game, and the board position itself. Compiling and downloading the evaluation tables can take 15-20 seconds, but the host software recognizes cases where tables have already been set up and the startup time averages less than 5 seconds per search.

7.3 The Transposition/Refutation Hash Table

This hash table is an optimization that allows results that have already been computed earlier in the search to be used to reduce the search time[8]. One function of the hash table is to detect transpositions, i.e. positions that are reached via two different paths in the search tree. By saving the result of each subsearch in the tree, the effort to search a position when it is reached a second time can be saved. The use of the hash table is complicated by α-β cutoffs that result in values that are bounds and not exact values. We maintain only one value for each position along with flags indicating whether the value is an exact value, a lower bound or an upper bound. In some cases it is necessary to re-search a subtree if the α-β values have changed since the previous search.

The second use of the hash table is to extend the usefulness of iterative deepening by using the information gained during previous search iterations to improve the move ordering. The hash table saves not only the result of previous searches, but also the move that achieved the result or forced an α-β cutoff. Searching this move first when searching the position to a greater depth yields near-optimal move ordering. Moves produced by the hash table are checked for legality by the move generator, since ignoring the possibility of collisions in the hash table could be fatal.

Hitech employs a depth-based replacement algorithm for positions in the hash table. Nodes that root a deep search subtree have precedence over nodes that root shallower ones since one wants to keep the information that is likely to be the most expensive to replace. Our measurements showed that depth-based replacement outperforms simple replacement by a factor of two.

Our measurements have shown that the combination of very good move ordering by the move generator combined with the ordering information provided by transposition table yields an α-β search efficiency that is only about 40% worse than the optimal possible. That is, on average, Hitech searches only about 1.4 times the number of nodes that an α-β search with perfect move ordering would search.

7.4 The ALU

A general-purpose 16-bit ALU (AMD 29116) allows the microprogrammer to perform arbitrary computations on moves and values within the machine. The ALU is connected to both the move and value data busses, and condition codes generated by ALU operations can be tested by the controller. The ALU is typically used to experiment with modifications to the search algorithm. When the experiments verify the utility of some refinement, then explicit hardware support can be provided. Currently the ALU is used to discover recapture sequences and to implement a hybrid hash table replacement algorithm.

7.5 Microprogramming Support

Microprograms are written as C programs using predefined macros that are then compiled, linked with a microassembler, and executed to produce the object microprogram. This allows the programmer to use C constructs in the microprogram as well as the C preprocessor for macros and conditional assemble. The microassembler module assigns addresses to microinstructions based on branch conditions so that the minimum amount of memory is used. It also uses information about the timing requirements of the various modules to automatically generate the correct clock timing based on the operations performed by each microinstruction.

7.6 Executing the Search

Figure 13 shows the operation of the hardware during the inner loop of the α-β search. This is an elaboration of the standard depth-first search, showing those tasks that are performed in parallel.

The move generation phase consists of computing the dynamic priority of the next move and voting to determine the chip with the best move. This move is made by executing three halfmoves, which are saved

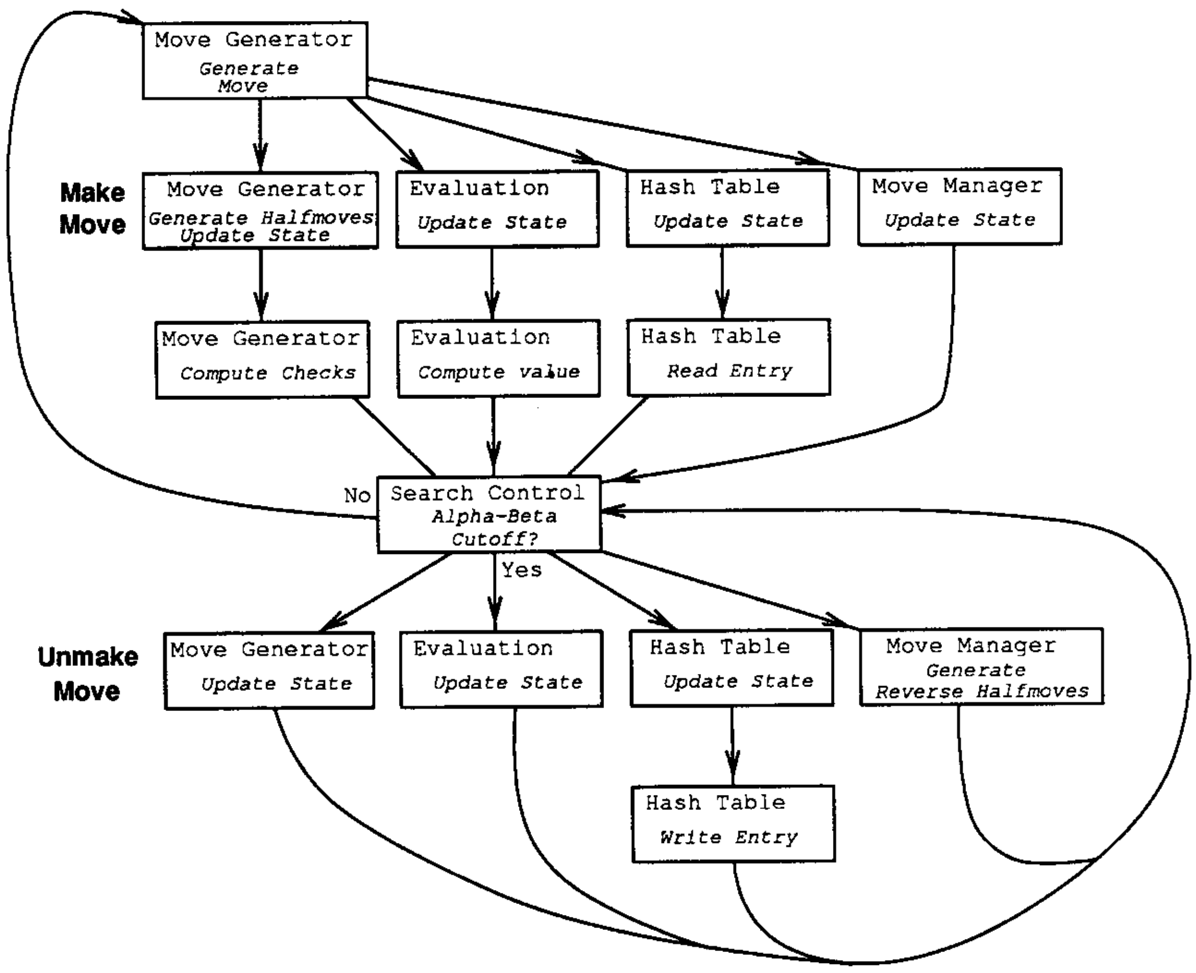

Figure 13: *An outline of the execution of the α-β search on the Hitech hardware.*

on a stack and used to unmake the move when the search backtracks. This making and unmaking of moves forms the backbone of the search. All modules that depend on the state of the search change that state when halfmoves are executed. After a move has been made, each of the modules computes new results based on the new position. The move generator calculates whether the new position is legal and whether the side to move is in check and the escape check mechanism should be invoked. The evaluation hardware computes a new value based on the new state, and the hash and repetition tables read the entries corresponding to the new position.

At this point, the search control decides whether a leaf position has been reached or whether the branch should be extended another ply. This decision is based primarily on depth and quiescence considerations, although the hash and repetition tables may provide an immediate value for the position. If indeed a leaf has been reached, the most recent move is reversed using the halfmoves saved on the stack and an entry is written into the hash table depending on the replacement algorithm. Otherwise the move generator or possibly the hash table produces the move with which to extend the tree.

7.7 Hitech Software

The Hitech software comprises a combination of a microprogram that performs the inner loop of the standard α-β search; a host program that performs the Oracle analysis, time control and user interface; and interface routines that pass information between the host and the chess machine. The microprogram includes a standard quiescence search that examines all captures and responses to check. Moves that escape from check are not counted as a ply since they are considered to be forced moves. Recapture moves are also detected and not counted as a ply under the assumption that the recapture was forced. Our definition of a recapture is a capture that re-establishes the material to the level that is considered to be the value of the root position. This definition means that recaptures in lines of play that maintain the material status of the root position are not counted. For instance, if the tree search starts out with NxN, and the answer PxN comes either immediately or after some intermediate non-captures, it will not be counted, as it re-establishes the equilibrium that existed at the root.

The Oracle in the host program performs an analysis before every search and downloads information to the evaluation hardware. The host program then builds the first level of the search tree and orders the root moves using shallow searches to establish exact values for each position at the first ply. From there, the program does iterative deepening using the chess hardware until the time allocation algorithm decides that there is not enough time for another iteration. This algorithm decides how much time to spend on any one move based on the amount of time and number of moves left in the game and an estimate of the difficulty of the position. After a move is made, Hitech assumes that the opponent will make the expected response and begins a new search. If the opponent makes a different move, this search is discarded. The user interface allows the operator to set the game parameters, and entertains the operator with a variety of interesting information about the progress of the search including the current prime variation.

8 Hitech Performance

Our experience with the Hitech chess machine indicates that the SUPREM architecture successfully combines high expertise levels with extremely fast search. Hitech examines approximately 175,000 positions per second, 10^6 times more positions than a human player. This ability has resulted in excellent moves that even Masters at the scene could not predict. Hitech has risen rapidly into the top 1% of ranked US chess players, and it is still climbing as Figure 14 shows.

As of December, 1987, Hitech had played 100 rated tournament games, principally against human competitors and achieved a record of 71-29. It has won the 1987 Pennsylvania State Championship tournament with 15 Masters in it, and had previously won another tournament with 4 Masters in it. Hitech has obtained draws in tournament play against three players ranked among the top 40 US players; one of whom is among the ten top US players. These accomplishments are unique for any machine. Hitech also won the 1985 ACM North American Computer Championships with a perfect score of 4-0, defeating defending champion Cray Blitz (running on a Cray X-MP 48).

Table 1: Hitech Rated Tournament Record

Epoch	All Games	Experts	Masters	Super-Masters	Perf. Rtng.
1	13 - 4	1 - 0	1.5 - 3.5	0 - 0	2158
2	34.5 - 11.5	12 - 2	13.5 - 3.5	1 - 6	2383
3	13 - 8	7 - 3	2 - 4	0 - 1	2216
4	12.5 - 5.5	7 - 0	2.5 - 0.5	3 - 5	2475

Hitech's current US Chess Federation rating is 2372, making it a high-rated Master, and surpassing all previous chess programs by about 170 points. If one averages Hitech's rating increase from the initial 2076 it earned in its first tournament in May, 1985, to its present rating, over the set of games it has played,

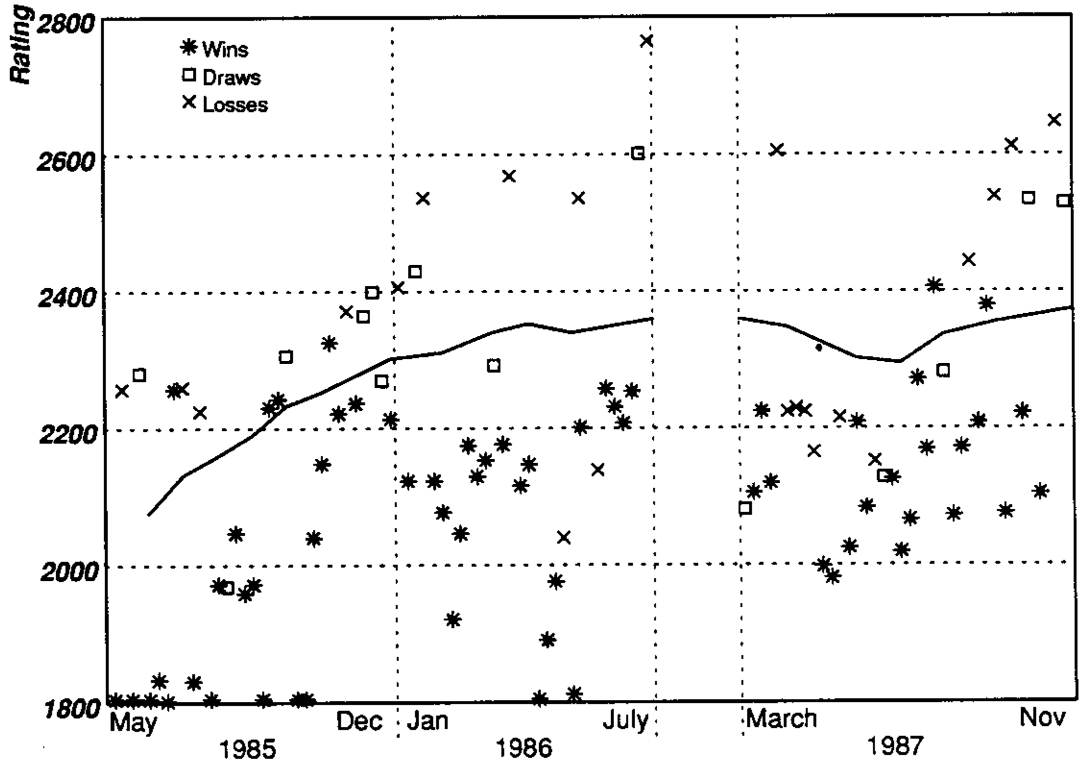

Figure 14: *Hitech performance history*

it averages out to a gain of about 3 rating points per game. However, Hitech has not always progressed steadily. We consider Hitech's career to be divided into 4 epochs:

1. May - September, 1985 without any pattern recognizing capability.
2. October, 1985 - July, 1986 with the 1st generation pattern recognizers.
3. February, 1987 - July 1987 with the 2nd generation pattern recognizers, marred by many software problems.
4. Since August, 1987 when the software problems associated with changing hardware seem to have been overcome.

Table I shows that during epoch 1, Hitech won 30% of its games against Masters. In epoch 2, when we added pattern-based evaluation, Hitech's success rate against Masters rose to 79%. However, its success rate against super-masters (those rated above 2400 in the USCF rating scale) was not noteworthy. In epoch 3, results were very uneven. We feel these should be ignored, and this view is substantiated by comparing the results against Experts, Masters and Super-Masters across epochs 2, 3, and 4. In epoch 4 Hitech's potential can again be seen, as the second-generation pattern recognizers discussed in this paper begin to take effect. Here it improves its performance against every class of player, including beating some super-masters. Hitech's percentage of wins does not increase from epoch to epoch, as the quality of competition keeps getting better, as can be seen in the individual columns.

The most significant column in Table I is the Performance Rating column. This is the statistical estimate of the most likely strength of player that would have achieved what Hitech did, considering the strength of opponents and result. Hitech plays chess well enough to be among the top 0.5 of 1% of all registered chess players in the World. It generally has an Expert (the rank below Master) level understanding of chess concepts and strategy. However, its tactical ability — the ability to calculate the direct consequences of a move — is close to that of a Grandmaster (the highest level of human play). This potential comes from its powerful search, which is much more thorough though not always as deep as that of the best humans. Hitech's search causes sparkling things to happen from time to time:

- Hitech has during its life made 10 moves, in tournament settings with a number of very good players watching, that were both excellent and **not anticipated** by **any** of the expert spectators.
- Hitech has found dozens of refutations of textbook examples that have been in the literature for many decades.

9 Why does SUPREM work so well?

9.1 Serial versus parallel computing

One factor that has historically militated against search is the necessity, in really big searches, to employ very simple evaluation functions. The fundamental difficulty lies in trading off the costs of doing evaluations against those of additional searching on a serial machine. In the past, this limitation has typically produced minimal evaluation functions, since smart ones do not perform sufficiently better to justify their computational expense. However, in a suitable parallel implementation, evaluation can become quite complex. And when evaluation is done in parallel, the expense of many evaluations is not much greater than that of one. SUPREM embodies an appropriate form of parallelism and Hitech has shown that the paradigm works well.

9.2 Problem Solving Alternatives

Let us consider the major known ways in which problem solving can be accomplished:

9.2.1 Pure Reasoning

Pure reasoning has not been very successful. It has been shown time and again that extraneous facts confound the solving process. Further, a pure reasoning process cannot cope with the Real World in that data must be converted to factual clauses to reason with. Is "Aristotle is a man" given or must this be extracted from the environment. If the latter, many additional issues arise. For instance, if Aristotle is 17 years old is he a man or a boy. Questions of grain also come up in actions. For instance in the "Monkey and Bananas" problem, the box could be moved to an infinite number of locations on the floor of the cage. Should one settle for 1000 locations? To only allow moving it to "under the bananas" or leaving it where it is, is so restrictive as to remove a great deal of difficulty from the problem. Pure reasoning systems seem to have reached their present high in doing simple robot planning problems in very friendly environments, and no adversary to create problems.

9.2.2 Hill Climbing

In hill climbing, the solving process will follow the path of steepest ascent until it either reaches the goal or gets stuck on a local hill. This situation is portrayed on the left of Figure 15, where a gradient is shown in the form of concentric contour lines. Presumably the hill climbing process would inch along, one step at a time, reaching higher ground with each step. One important point to note is that even though progress is made at each step, there is no guarantee that the optimal point at distance "d" will be reached. This is because there may be a higher point at distance "d", which is reached by following a lesser gradient initially and then reaching a point where steeper ascent is possible. However, a hill climber will reach a goal, unless it finds a non-goal local hill (see right of Figure 15) from which it cannot escape. The latter situation can

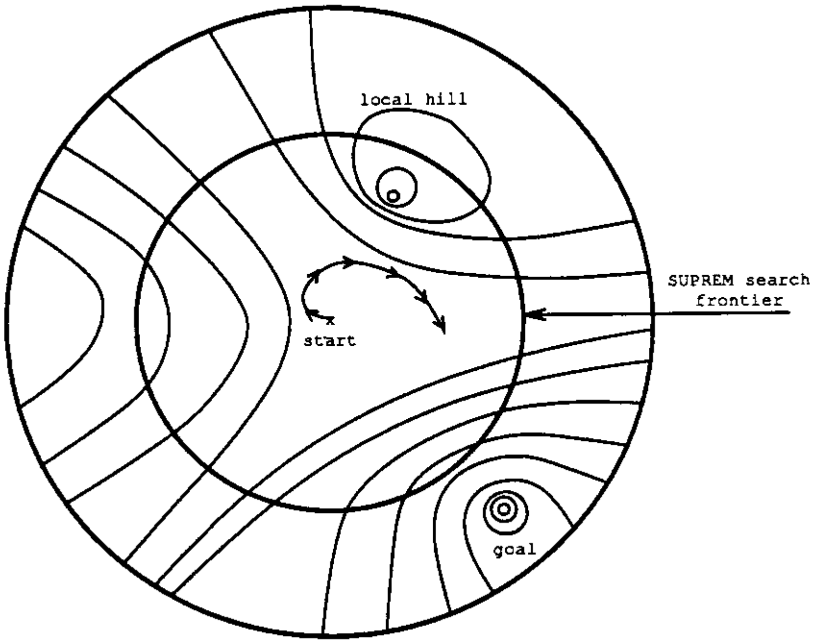

Figure 15: *Comprehensive search in SUPREM.*

arise since a hill-climber does not backtrack, and will thus never return to a lower point in the evaluation space. This method of coping can be seen in many creatures low on the evolutionary scale. A worm, which may only need to find an appropriately wet place, does very well just following the gradient of increasing wetness.

9.2.3 Rule-Based Systems

A rule-based system attempts to improve upon hill climbing by applying a great deal of knowledge to the local situation before taking a step, and allowing backtracking when a path does not live up to its earlier promise. Rule based sytems can have evaluation functions that take account of certain known properties of the current problem state, and chart a course that may appear to be non-optimal to a simple evaluation function. Such systems can thus show a great deal of flexibility in dealing with a problem. However, they still solve problems one step at a time, and must nilly-willy accept the snail-like paths that could come (see center of Figure 15) from not knowing exactly how best to make progress.

Typically, rule-based systems have been used to solve problems by finding an actual solution, rather than by attempting solution through a series of successive approximations as is done in chess and other environments where it is not possible to search to the end of the game (or final solution). This has meant that either a domain must be highly constrained (so as to make few alternatives viable), or the depth of the solution must be rather shallow in order for a rule-based system to be effective. Almost all rule-based systems are applied to domains that are thus friendly. It will be interesting to see if anyone can construct a rule-based system to deal with a domain as complex (10^{43} states) and difficult to search (35 alternative operators on average) as chess. One early example of a rule-based system [9] required some 40 rules to produce a program that could successfully mate with a king and rook against a king. This result does not augur well for rule-based systems in domains of high complexity and few constraints.

9.2.4 SUPREM

Since SUPREM is not designed to search to the end of the problem, it resembles a hill climber more than any other of the above. However, it is a hill climber with *seven-league boots.* Since the SUPREM architecture is based on powerful searching, it will be able to conceptually take many steps forward before having to make a decision about the goodness of the state it is currently visiting. If one takes the worm analogy for hill climbing, then SUPREM is a giant worm that can look 9 steps ahead, including backtracking on inferior steps, before taking one. This has many obvious advantages:

- It will not be fooled by small local variations in gradient that could cause a hill climber to go off in a wrong direction.
- It does not need the kind of knowledge that rule-based systems require in order to select the next action to try, because it will try **all** actions out to the search frontier (see Figure 15). This results in simplification of the knowledge needed.
- SUPREM seems to have the best of two worlds. It has a rule-based system as an Oracle to lay out the knowledge it needs to take its next step. The actual step is taken as the result of an extensive examination of the alternatives using high levels of understanding.
- Since it also does not see to the end, SUPREM could also, willy nilly, set an irregular course to its target. However, this course should be much more direct than that of a standard hill climber, or even an extremely well-informed expert system. Further, since SUPREM takes only one step at a time while seeing forward many, it is not as necessary to have as detailed knowledge of the frontier as a process that looks less far ahead. This is a *very important point that makes it much easier to provide knowledge for the SUPREM paradigm.*

The latter is an observed fact from studies with Hitech. The explanation seems to lie in that in order to take the first step correctly it is only necessary to head for the correct **region** of the state-space. Low level details are not too important as after SUPREM has taken several steps forward, the frontier will also have moved forward. At this new juncture, details that were not previously visible will now dominate the decision concerning what is the best direction in which to head. Thus the fine detail that is used to estimate the value of single (or very small number of) steps, appears not to be needed.

We consider this observation to be closely akin to the situation in planning, where macro-plans can be compiled without paying attention to small details. It is taken for granted that some method will exist for dealing with the solving of the low-level details, once the major ones have been worked out. This makes it extremely useful to have knowledge of important features that have a permanent, or at least semi-permanent role, in the solution process. A system that can see forward only one step at a time cannot afford this, as it must utilize every possible clue in an effort to decide the immediate next course of action. We consider the rule "**The further a process can look ahead, the less detailed knowledge it needs**" to be generally true. A process that can see actual goals need only be able to identify these. A process that can see within a few steps of a goal will on average not need much understanding of intermediate goals. A process that can only see one step ahead and is far removed from any goal needs tremendous amounts of information. Our rule subsumes all these cases and appears to be a generalization. It also appears to also be implied by the curves of Figure 1.

Our evidence for the rule is two-fold. Firstly, as knowledge for Hitech was being crafted, we used appropriate knowledge from an earlier chess program. Hitech outsearched the old program by 4 ply. It was found that the knowledge that Hitech needed was principally the high-level knowledge, and it never required most of the low level knowledge that the earlier program had. The second piece of evidence is that in order to change the present behavior of the program, it is necessary to add pattern information that represents substantial sub-goals. This will affect its behavior, whereas applying heuristics with small values will not have any meaningful effect.

The usefulness of "important feature" knowledge thus leads directly into SUPREM's use of pattern recognition. As was shown earlier, this second-order knowledge is able to determine important relations among features; relations that form patterns which have a permanent role in the solving process. We treat this subject in the next section.

10 The Usefulness of the Recognizers

In order for SUPREM to operate effectively it is necessary to evaluate leaf nodes in the search very, very quickly. In Hitech this is accomplished in approximately 1 usec. Further, the structure as described in Section 6 does evaluations in essentially constant time, regardless of the number of recognizers and the patterns they are seeking to detect.

We believe the structure of the recognizers to be very general. This structure evolved from direct experience with Hitech and is still evolving. The important issues are:

- They are able to compute second-order knowledge which is essential.
- They are fast.
- They are implementable at reasonable cost and space.
- Evaluations can be factored into pieces small enough to fit into a recognizer and then recombined at will. This allows dealing with very complex situations.
- Application Coefficients, which are needed to provide context information can be readily computed.

There are some disadvantages to our scheme, that should be noted. However, presently none of these appear to be important.

- The recognizers are loaded once for each root search. This means that if the situation during the search were to change drastically, so that the down-loaded knowledge were no longer applicable, the evaluation of such leaf nodes would be inaccurate since new knowledge could not be downloaded. This problem rarely appears, and there is no immediate remedy, nor do we see an urgent need for one.
- Each recognizer has a limited size; i. e. 13 bits in each branch. This allows a great deal of information to be dealt with. For instance, in King-Safety, one recognizer deals with the pawn shelter, location of king, open lines available to the opponent, and whether his own rook is locked in by the king. The Global-State recognizer then provides information on the amount of material on the board, which modulates the final value. However, it is clear that expertise comes through better understanding, and there will always be more features that could be incorporated. Thus, there is pressure for ever-larger recognizers.
- The interconnection scheme among recognizing units is not completely general. This would be too expensive. The Oracle software knows exactly which recognizers can be connected to which others, and while, in principle, it would be very desirable for results to be shared among the recognizers, this lack has not yet become an important hindrance for progress.

Our theory of "recognizing" as evinced by the recognizers is as follows:

1. Top level bits are extracted from the current state according to the needs of the pattern that is to be detected. As stated above the number of bits is governed by cost considerations. The adequacy of any particular width must surely be dependent upon the domain. However, we have found that the present 12 bits plus one external signal in each main branch of the recognizer is capable of a great deal. Typical tables that are down-loaded into a particular recognizer are capable of detecting three or four distinct but related patterns.

2. Mapping-down is necessary to get an eventual interpretation of the top level bits. It should be noted that as classes or interpretations are found, it is possible to aggregate values, classes, or interpretations. Thus a class at the second level could represent a compound of inputs from several patterns in the same branch of the recognizer. In effect, each map-down step abstracts some meaning from the level above it, until a final meaning in the form of an evaluation is produced. The three-fold mapping with approximately a two-one reduction of bits in each mapping step appears to be adequate in most cases.

3. The restrictiveness introduced by going to fewer bits in each map-down step is quite similar to what one would expect in a learning paradigm. The need to represent what is known in fewer bits forces generalization and thus learning. In our application there is no machine learning, and the representations we use are very likely much too terse to be discovered by today's learning algorithms. However, in principle, a structure of this type could be taught, either by experience or by a tutor.

4. The ability to gradually extract meaning from a set of bits provides great flexibility in knowledge encoding. For instance, in the typical pattern loaded into a recognizer, there are 2^{24} states times 2^{12} global states which are mapped down to an interpretation which is at most 2^8. To extract meaning from so much data, the input of which has already been filtered, in 1 usec puts competitive systems at a tremendous disadvantage.

Although it is not always possible to encode every facet of every potential pattern, we have found that in general we are able to encode any item of knowledge that we have striven hard to encode. Since the inception of pattern recognizers, we have continued to add successful patterns to the knowledge base at the rate of one or two a month. We believe that Hitech's knowledge can be increased to produce further growth in performance.

11 Applying SUPREM to other Domains

Certain large and well-formed problems that have a clearly defined domain with definite rules, could benefit from the application of SUPREM. One such domain is analytic chemistry. Here a molecule is known to contain a precise number of each of possibly dozens of kinds of atoms. However, the overall structure of the molecule may not be known. Further, the actual molecule is known to have certain chemical properties that reveal themselves in the structure.

The analysis method using SUPREM would build the molecule incrementally, making decisions about where each as yet unused atom would best be located. There would be a move generator for each atom type, and this would know all possible bonding sites for the atom in the current molecular structure. The sites would be known in order of general preferrence. The search would involve placing atoms into the molecular structure. Pattern recognizers would identify the presence of desirable and undesirable radical groups that define the chemical properties of the molecule under consideration.

Such radical groups are *chunks* or patterns, and the Oracle would have the knowledge as to which patterns are associated with the known chemical properties of the molecule being searched for. The evaluation function would be loaded at run time with the correct pattern information. The presence of such patterns in a candidate configuration would increase the score for that configuration. With a machine to apply the SUPREM paradigm, it would be possible to examine hundreds of millions of alternative configurations in reasonable amounts of time. At the end, the hundred configurations with the best scores could be printed out for examination by a qualified chemist. Experience with chess shows that such searches will regularly turn up things that the whole population of chess masters has not discovered.

12 Summary and Conclusions

We have described an architecture for problem solving that differs from any previously known. The SUPREM paradigm combines large searches with the power of pattern recognition. The former allows visiting any node within a limited, but large, search horizon that could contain a solution. The latter allows detecting complex conditions that could indicate the value of a node. Pattern recognition provides the ability to ascertain relations among elements in a domain state, and is thus considerably more powerful than the application of heuristics that merely indicate that the presence of some element of a domain state is good to a prespecified degree. Without pattern recognition large amounts of additional search would be required to find the properties of a domain state that patterns can encode.

We have built a chess machine/program, Hitech, that embodies this architecture. Hitech has achieved spectacular results in its domain and exceeded previous performance highs for programs by almost a complete class on the human scale. We have learned a number of interesting things that appear to be general.

1. Patterns are capable of creating a rough evaluation surface because the discerning of an important relation allows placing more emphasis on its presence than would be prudent in a purely heuristic scheme, where each item of knowledge contributes only a small amount toward a favorable appraisal.
2. Rough evaluation surfaces, while usually undesirable, are actually desirable when the search projects far enough ahead. This is because real mountains on the surface are worth achieving. False mountains, [4] if far enough away when first detected, can almost always be avoided, if enough time is available to change course.
3. Thus the big search has a second advantage: it allows detecting false goals and avoiding them. A search of very limited horizon will not be able to do this, and thus requires much more accurate information.
4. Finally, we have found that it is possible to build pattern detection hardware that:
   - Is inexpensive to build and contains a great deal of power for detecting patterns that the Oracle considers to be near the root, and pertinent to the evaluation.
   - Makes possible the parallel detection of patterns and their synthesis into an eventual evaluation at time costs that are essentially constant even as the number of recognizing units grows.

We feel this style of computing has a very potent future, and look forward to having it tried in other domains.

13 Acknowledgements

A machine of this complexity could not have been built by two people. Hitech is the result of a team effort. The hardware has been designed and built by Carl Ebeling; the Oracle is based largely on the chess program Patsoc, by Hans Berliner. The above concepts and implementation are the result of much interaction between the two principals. Other students and department employees have contributed significantly in the design of the searching software, the programming of the control system, testing of the whole structure, and other facets. Persons who deserve mention are Gordon Goetsch, Murray Campbell, Larry Slomer, and Andy Palay. Since this research involves the two areas of AI and VLSI, not every Computer Science Department could have supported this project, and we are grateful for the opportunity provided by CMU and its Research Sponsors.

References

[1] H. J. Berliner. On the construction of evaluation functions for large domains. In *Sixth International Joint Conference on Artificial Intelligence*, pages 53–55, IJCAI, August 1979.

[2] J. H. Condon and K. Thompson. Belle chess hardware. In *Advances in Computer Chess III*, Pergamon Press, 1982.

[4] One can think of a false mountain as one that has an excellent facade as one approaches, but the terrain behind it falls off rapidly.

[3] C. Ebeling. *All the Right Moves: A VLSI Architecture for Chess.* PhD thesis, Carnegie-Mellon University, April 1986.

[4] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6:293–326, 1975.

[5] D. Michie. A theory of advice. In E. W. Elcock and D. Michie, editors, *Machine Intelligence 8*, Ellis Horwood Limited, Chichester, Sussex, England, 1977.

[6] A. L. Samuel. Some studies in machine learning using the game of checkers, recent progress. *IBM Journal of Research and Development*, 601–617, November 1967.

[7] D. J. Slate and L. R. Atkin. Chess 4.5 - The Northwestern University chess program. In P. W. Frey, editor, *Chess Skill in Man and Machine*, Springer-Verlag, Berlin, 1977.

[8] A. L. Zobrist. *A Hashing Method with Applications for Game Playing.* Tech Report 88, Computer Science Department, University of Wisconsin, 1970.

[9] C. Zuidema. Chess, how to program the exceptions. In *Afdeling Informatica*, Amsterdam, 1975.