An Analysis of Move Ordering on the Efficiency of Alpha-Beta Search

Eric Thé

School of Computer Science
McGill University, Montreal

A Thesis submitted to the faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science

October, 1992

To my father and mother.

"Heuristics are bug-ridden by definition. If they didn't have bugs, then they'd be algorithms"

Anonymous

Acknowledgements

I would like to thank my supervisor, Prof. M. Newborn for his valuable guidance and support, and for his patient reading of the many manuscripts which culminated in this thesis.

My thanks are also to fellow students Philippe Locong, and Luc Boulianne for their much appreciated help with the many programming, software, hardware, and LaTeX-related bugs and probems I had to overcome to achieve this thesis.

Finally, I gratefully acknowledge my father who continually pushed me in the right direction and kept me going when the going got rough, as well as my cats Nikita and Ille Noire for keeping good company during those many long nights of labour.

Abstract

Move ordering is important to alpha-beta tree search efficiency since a well-ordered minimax game tree precipitates more cutoffs than a randomly ordered tree. The use of move ordering heuristics is shown to be a valuable addition to the alpha-beta algorithm. Eight different move ordering heuristics are described and implemented on a skeleton chess program. The relative gains in search performance are observed for each individual heuristic as well as for combinations of heuristics. It is shown that when searching minimax game trees to a depth of 6 to 9 plies with well-chosen "killer" heuristics, reductions in search time and tree size exceed 80% on average.

The effectiveness of move ordering heuristics is dependent on the depth of search and the scoring function used. With increased search depths, it is shown that the use of transposition tables to order moves becomes progressively more effective. Inversely, the use of Schaeffer's history heuristic is shown to be less effective with deeper searches.

The use of a simple material-based chess scoring function is seen to have a strong influence on the performance of the capturing moves heuristic. It is observed that in this context, the ordering of captures improves search performance for materially unstable positions only.

Résumé

La procédure alpha-bêta est d'autant plus efficace si l'on examine d'abord le meilleur coup. L'emploi de méthodes heuristiques pour ordonner les coups est illustré comme étant une importante extension à la procédure alpha-bêta. Huit méthodes heuristiques différentes pour ordonner les coups sont présentées et incorporées dans un programme d'échecs de base. Les réductions du degré effectif de l'arbre de jeu sont notées pour chaque méthode heuristique et pour certaines combinaisons de méthodes heuristiques. Avec un bon choix de méthodes heuristiques pour identifier les "coups meurtriers" dans des arbres de 6 à 9 niveaux de profondeur, nous obtenons des réductions de plus de 80% sur le temps de fouille et l'étendue de l'arborescence fouillée.

L'efficacité de méthodes heuristiques pour ordonner les coups dépend de la profondeur de fouille et de la fonction d'évaluation utilisée. Avec des fouilles plus profondes, l'emploi des tables de mémoire pour ordonner les coups aux positions déjà vues devient plus efficace tandis que l'emploi du "history heuristic" de Schaeffer devient moins efficace.

On illustre également l'influence importante d'une fonction d'évaluation strictement matérielle sur la performance de la méthode heuristique pour ordonner les captures. On démontre que dans ce contexte, l'utilisation de la méthode des captures ne réduit la fouille d'arborescence que pour des situations matériellement dynamiques.

Contents

1 Introduction 1

1.1 Game Tree Search 2

1.2 Move Ordering Heuristics 3

2 A Survey of Recent Computer Chess Search Techniques 6

2.1 Reasons for Selecting Chess 6

2.2 Minimax Tree Search and the Alpha-Beta Algorithm 7

2.3 Move Generation 8

2.4 Quiescence Search 10

2.5 Transposition Tables 11

2.6 Iteratively Deepening Search and Aspiration Search 13

2.7 Killer Heuristics 15

3 Implementation of Move Ordering Heuristics 17

3.1 Three Classes of Heuristics 17

3.2 The FIXAFAN Chess Program 18

3.3 Simple Level-Restricted Heuristics 21

3.4 Using the Transposition Table 22

3.5 Killer Move – Parent Move Association 22

3.6 Dynamic Move Ordering 23

3.7 The History Heuristic 24

3.8 Ordering of Capturing Moves 25

4 Performance Comparisons 26

4.1 Test positions and heuristics 26

4.2 Measurements 27
4.2.1 Nodecount 27
4.2.2 CPU Time 28
4.2.3 Average Branching Factor 28
4.2.4 Killer Average 28

5 Experiment Results **34**
5.1 How Results Were Compiled 34
5.2 Experiments Using Single Heuristics 35
5.3 Performance of Heuristics Coupled With CAPTURES 37
5.4 Combining a heuristic with CAPTURES & BEST_TT 39
5.5 Influence of Search Depth on Heuristic Performance 41
5.6 Correlation of Performance and Evaluation Function Used 43
5.7 Profiling the FIXAFAN Program 47
5.8 Other Combinations 49

6 Conclusion **51**

A Bratko-Kopec Test Positions **54**

B Experiment Statistics **59**

C Execution Examples **60**

D Source Code of FIXAFAN Program **85**

Chapter 1

Introduction

Tree-searching is a computational task which usually can not be performed without the help of great selectivity. Game-playing, parsing, the 8-puzzle and 15-puzzle, mathematical theorem proving, and other state-space problem solving programs must all undertake some form of tree search in order to find the best possible solution. The main objective of these programs is to achieve this high standard of selectivity. If the algorithm implemented in a tree-searching program is the most efficient algorithm known for such a problem and the results obtained are unsatisfactory, then one of the few remaining options left to improve the program's level of selectivity is the use of heuristics.

Heuristics are "rules of thumb" which help a program make time-saving decisions that are intended to produce efficient "shortcuts" in its calculations without jeopardizing the likelihood of returning a "correct" solution. For example, programs which play games such as chess can successfully find good moves to play with the help of heuristics. The increasing use of heuristics in game-playing programs was first noted in a survey paper by Newell, Shaw, and Simon;

> We have now completed our survey of attempts to program computers to play chess. There is clearly evident in this succession of efforts a steady development toward the use of more and more complex programs and more and more selective heuristics; and toward the use of principles of play similar to those used by humans [33].

Reliable heuristics can thus increase a game-playing program's ability to achieve it's intended purpose which is the selection of good moves for any particular game situation.

However, the validity and effectiveness of a heuristic is not always easy to determine or quantify. A heuristic may be very simple to implement and can produce large savings in computation time but it may not always return a satisfactory search result. In order to clearly understand the effects of heuristics in game-playing programs, one must first and foremost be aware of the implications of game tree search.

1.1 Game Tree Search

Two-person zero-sum games which require human intelligence such as Chess, Checkers, Go, and Othello, can all be solved computationally by programming a computer to search a state-space *tree* where *nodes* represent the problem's positions or states (i. e. game board), and where *branches* represent the operations which transform one state into another (i. e. a player's move). A search in a game tree of exponentially growing possibilities can be virtually endless. For example, the game of chess has an astronomically large number of states – over 10^{43} [40]! For trees of such size, finding the exact solution is totally impractical. An "acceptable" solution can be found by imposing a limit to the depth of search, or the time of search, and evaluating all horizon nodes. However, even depth-limited trees can become exceedingly large when searched only a few levels deep.

Expert human chess players seem to have the ability to concentrate on moves relevant to the current game situation and ignore other moves and their continuations which are believed not to have a significant influence on the game's denouement. Analogously to humans, chess programs attempt to "cut off" useless branches of the tree by selecting which moves are considered "bad" relative to the game situation and then using these to help minimize the size of the tree.

The basic algorithm used to "cut off" unnecessary moves in a game tree is the alpha-beta algorithm. This algorithm will cause the refutation of a node if one of its moves returns a score with a large enough variance – relative to the previous score — to render the searching of the remaining moves unnecessary. Without the alpha-beta algorithm, a minimax search of a hypothetical game tree with constant depth D and constant number of branches B at every node will examine

$$M(B, D) = B^D$$

terminal nodes. With an ideal ordering of moves, the alpha-beta algorithm will examine a

minimum of only,

$$AB(B, D) = B^{\lceil D/2 \rceil} + B^{\lfloor D/2 \rfloor} - 1$$

where $\lceil x \rceil$ is the smallest integer $\geq x$,
and $\lfloor x \rfloor$ is the largest integer $\leq x$

terminal nodes [17, 41]. This result was first given in a 1969 analysis of a Kalah-playing program by Slagle and Dixon [41], and shows the savings in computation the algorithm can achieve since as little as the square root of the maximum number of nodes need to be examined to obtain the same solution. Knuth and Moore also showed that the optimum ordering of a game tree is not as intuitively obvious as one might think [17]. Having the best possible move as the first successor of every node will not always result in a minimal alpha-beta tree traversal—unless the tree has uniform depth for *all* branches. Hence optimum tree ordering is really an idealist "goal" so one must instead concentrate on producing *strongly ordered* [23] game trees to help the algorithm achieve a good performance.

Further research has shown that when used with a random ordering of moves, the alpha-beta algorithm can still offer a significant improvement in performance. Knuth and Moore [17] showed that when random scores are assigned to the terminal nodes a tree of constant depth 2 and fanout B, the algorithm will examine on average;

$$\theta(B^2/\log_2 B) \quad \textit{of the } B^2 \textit{ terminal nodes.}$$

Shortly thereafter, Newborn [28] demonstrated that since the terminal nodes of game trees have scores which are branch-dependent, the average number of nodes examined when using a model which reflects this is;

$$\theta(B \log_2 B) \quad \textit{of the } B^2 \textit{ terminal nodes.}$$

It is therefore beneficial for a game-playing program to possess a reliable method of move ordering in order to take full advantage of the alpha-beta algorithm's potential of reducing the scope of a tree search.

1.2 Move Ordering Heuristics

How can we efficiently make use of heuristics in the domain of game tree search? Some heuristics in chess programs are used to reorder the moves generated at a node so that

the move(s) most likely to cause an alpha-beta cutoff are placed at or near the top of the move list. These heuristics therefore attempt to decrease the number of branches and nodes searched in order to find an "acceptable" solution. Some heuristics are more effective than others in actually accomplishing this task of tree-pruning [36, 22, 23, 20]. The ideal or "best" heuristic would therefore be that which would guide the search through the smallest subtree of a complete game tree and still guarantee a "correct" solution path.

There are two generalized methods of ordering a list of legal moves generated from a particular game position or tree node. With *static* move ordering heuristics, the ordering process occurs when a node is initially generated. However, with *dynamic ordering* the node's move list can also be intelligently reordered when the search returns to that node after examining part of its subtree.

An example of a static ordering heuristic is the use of *killer moves* [23, 25, 36, 1, 12, 32], a relatively inexpensive move reordering strategy. Killer moves are so named because these are moves most likely to cause the refutation of a node and its associated subtree. These moves are selected based on information gathered during the course of the tree search. The method used to choose which is the best killer move at a node can be very computationally inexpensive (eg. simply selecting the move which was responsible for the most recent cutoff at the same level in the tree), or slightly more involved and requiring some degree of calculation and tabulation (eg. Schaeffer's *history heuristic* [35] which keeps a killer "score" for each possible move by both players).

Dynamic ordering heuristics [41, 29] make use of the strong correlation between deep backed-up scores of a position and its value at shallower depths. The information gathered at deeper plies for a position can effectively be used to reorder moves at shallower plies for a similar position with many of the same moves and continuations. In this way, valuable information on a particular killer move becomes available to all parts of the game tree rather than being restricted to the ply at which the killer move was found.

Alternately, a different categorization of heuristics can be formalized if we consider the nature of the information used by a heuristic to order moves. All the above-mentioned heuristics are *game-independent* heuristics which use results obtained from the tree search itself. These heuristics take into account the direct consequences of searching a particular move in the game tree. This information is thus totally independent of the actual game used.

Game-dependent heuristics on the other hand use some aspect of the game's characteristics to order moves at a node without referring to previous search results for any particular move. The ordering of *capture moves* at the top of a node's move list is a good example of a *game-dependent* ordering strategy used during chess tree search. The use of capture moves has been the topic of discussion of a number of papers [5, 25, 6, 12] in the computer chess field and has proven to be quite effective while at the same time having negligible taxing effects on the rate of search of chess programs. It is a very cheap and valuable heuristic which should not be overlooked when developing a chess-playing program.

These heuristics, applied at each node, necessarily introduce some degree of computational overhead and thus increase the number of instructions needed to process each node. The most inexpensive heuristic is simply no heuristic at all; there is no overhead introduced but the program in this case will naively carry out a *brute force search* of the whole tree. While this guarantees the exactness of the solution obtained, it does not guarantee that the solution will be returned after a reasonable amount of real time. Even if the program can now search a tree at the fastest nodes-per-second rate possible, since there is no attempt to discard unnecessary branches the search might not terminate within a practical time limit (i. e. minutes or hours instead of days or weeks!). The heuristics must be implemented inexpensively so that the program retains the ability to process a node within an acceptable number of instructions while still maintaining a high standard of play.

In order to make a proper selection as to what type of move ordering heuristic should be used for chess one must take an investigative look at how each different heuristic performs in terms of effectiveness and simplicity of implementation. The effectiveness of a move ordering heuristic is readily verified by a comparison of the final node count for that game tree versus the same search carried out by a simple non-enhanced alpha-beta algorithm. The simplicity of the heuristic is reflected in the program's rate of nodes-per-second when compared against the rate of a normal alpha-beta implementation. A heuristic which would perform well on both criteria would thus be a logical choice since it would prove to be an excellent addition to a basic implementation of the alpha-beta algorithm. The work performed as part of this thesis thus attempts to evaluate the advantages of using different move ordering heuristics. Since the validity of some of these heuristics has not been fully certified—as of yet the benefits of using the killer heuristic are still vague this thesis shall hopefully enlighten other researchers on their true computational merit.

Chapter 2

A Survey of Recent Computer Chess Search Techniques

2.1 Reasons for Selecting Chess

Chess is a 2-person zero-sum, perfect information game. The fact that chess is a game without hidden information enables a program to accurately look ahead in a game tree, unlike games where a move depends on the random outcome of a dice. The game tree traversed by a chess program is thus an adversary tree where both players alternate in making a move. Chess is sufficiently complex and subtle in its implications so that neither player can hope to completely understand a typical game situation. These characteristics thus make the game appealing to programmers.

Chess is a game which provides a good barometer for measuring algorithm and heuristic performance since it is neither too simple (eg. Tic-Tac-Toe, Checkers, Othello), nor too hard to implement. The size of its game tree also makes it an ideal choice for research. In his innovative 1950 paper, Shannon [40] stated that the complete game tree for chess includes over 10^{43} positions. If searched at a rate of one million nodes per second, it would thus take 10^{60} years to obtain its mathematical solution! Therefore at the present moment, the only feasible method of finding—within a reasonable time limit— a solution to a chess tree is to search only part of the tree and use a scoring heuristic to evaluate the winning/losing value of each leaf node. The performance of a chess program depends on factors such as rate of search, the depth of search, the scoring function used, and the number of irrelevant

nodes pruned during the search. Since the first major computer chess tournament in 1970, chess ratings of the best programs have been increasing consistently. If the trend continues, a world champion rating of 2911 could be surpassed by the year 1991 [21].

2.2 Minimax Tree Search and the Alpha-Beta Algorithm

The goal of a game tree search is to calculate the *principal continuation* which represents the best line of play for both players given that a depth-limited search is carried out. The continuation is found by assuming that both sides will always make the best move at each turn. In this process a program will choose the highest valued move at all even levels of the tree and the lowest (i. e. the most negative) valued moves at all odd levels assuming that the first move is to be made by the computer and that the better the position, the more positive its score will be. This process of alternately selecting the highest valued move for each side to obtain a principal continuation is known as the *minimax tree search algorithm* [41, 32, 3].

One method of searching a minimax tree can be described in terms of five main procedural components. The **GENERATE(position,movelist)** procedure will accept as input a game position and generate a list of all the legal moves that the next player can make from the given position. Two procedures **UPDATE(move,position)** and **RESTORE(move,position)** are used to make a move on a position and once that new position has been fully evaluated, retract the move to restore the previous position in the tree. A scoring function **EVALUATE(position)** is used to determine the tactical value of a terminal position. These scoring functions usually consider material balance as the main factor in evaluating positions. However, other positional factors taken into account may include pawn structure, mobility of the pieces, king safety, and control of the center [15, 39, 12, 21]. Finally, **PC_UPDATE(move,depth)** is used to store the current move examined as a potential branch of the principal continuation if its score is an improvement over the best score so far at the current position.

The *minimax algorithm* can easily be extended to become the *alpha-beta algorithm* which performs a non-exhaustive depth-first search of a game tree. A complete minimax search of a game tree is usually too expensive to perform without the addition of this algorithm even if the tree is depth-limited—because all the leaf nodes must be examined, many of

which are not relevant to the outcome of the search.

The algorithm is shown in Figure 2.1 as a recursive pseudo-Pascal function similar to an example presented in Marsland's 1986 paper [25].

The only difference between this function and a basic minimax search are the few lines necessary for checking that scores are within the alpha and beta bounds. Scores outside these bounds indicate that a move need not be searched since it has no influence on the outcome of the search. This is a very significant and inexpensive improvement over an exhaustive minimax search since it can easily be shown that many nodes are unnecessarily searched without alpha-beta bounds.

2.3 Move Generation

The most expensive component of most chess programs as far as time consumption is concerned is the move generation procedure. Move generation has been the subject of a lot of research especially in the use of special purpose hardware [32, 5, 26, 11]. BELLE [8] used an 8X8 array of combinatorial circuits; it also used a priority circuit which selected the "next best move" to generate. This technique enabled the moves to be generated one at a time and in the desired order. Generating moves one by one is a clever time-saving method [5]; many moves are unnecessarily generated since alpha-beta cutoffs often occur after searching only the first few moves of a node's list. Having all moves ordered so that the potentially good moves are at the top of the list is beneficial in maximizing cutoffs and ensuring optimum performance [6, 22].

A similar hardware implementation is used in the HITECH [10] machine where an 8X8 array of VLSI chips is set up solely for move generation. Each of the 64 chips represent a square on the board and generates only moves which have that square as its destination. This method of generating moves "to a square" requires much more computational effort than the conventional "from a square" approach but it does produce a more desirable order of moves which can prove to be less expensive to search. The current world champion program DEEP THOUGHT has one of the most advanced VLSI move generators and can search over 700,000 nodes per second [21]. The use of hardware is not restricted to move generation; some machines make efficient use of a hardware evaluator for leaf nodes and there are also versions of circuitry for hash table management [24, 11].

```
FUNCTION  AB ( p:position;  a, b, depth:integer): integer;
                          /* p is a pointer to the current tree node */
                             /* a & b are current alpha-beta bounds */
                           /* depth is the current level in the tree */
                        /* function returns value of subtree at node p */
{
  score, j, value: integer;
  movelist: array[MAXSIZE] of movetype;
  if ( depth = MAXD )  then          /* MAXD is a global var. */
    return( EVALUATE(p) );                 /* leaf node score */
  GENERATE(p,movelist);              /* generate list of moves */
  if  EMPTY(movelist)  then          /* leaf node if no moves */
    return( EVALUATE(p) );
  score := -MAXSCORE;
  for  j := 0 to sizeof(movelist)  do
  {
    UPDATE(movelist[j],p);           /* make the current move */
    value := -AB (p, -b, -max(a,score), depth+1);
                                /* go to next level by calling
                                       negative inverse of AB */
    if ( value > score )  then
    { score := value;                 /* note new best score */
      PC_UPDATE(movelist[j],depth);    /* note new best move */
     }
    RESTORE(movelist[j],p);      /* restore previous position */
    if ( score >= b )  then
      return(score);                  /* an alpha-beta cutoff */
   }
  return(score);
}
```

Figure 2.1: Depth-First Alpha-Beta Function

Recent work has focused on methods of reducing the number of instructions required by the move generation procedure. One such method suggests using a list of all possible legal moves of all pieces from all possible board locations stored in one huge data structure. This large list is computed once and for all. When the program needs to produce a list of moves from a position it will use this data structure to appropriately fetch all the necessary legal moves instead of generating them. Using this method it was shown that an average of 200 machine instructions are sufficient to process a node during search [14]. Other innovative ideas try to focus on methods of differentially updating the list of moves after a position has been updated as opposed to regenerating all the moves from scratch. This notion of differentially updating chess information was first addressed by Scott [38] where he discussed the potential gains of chess programs which would use a differentially updated evaluation function and/or move generator.

2.4 Quiescence Search

When a depth-limited minimax search is carried out on a chess tree, the scores returned from the leaf nodes may not be accurate in part because of the *horizon effect* [25, 43, 4] of continuations which were not driven to quiescence. Non-quiescent continuations can therefore be examined deeper than the arbitrarily set depth limit until a stable or "quiet" position is reached, thus reducing the horizon effect. A variable-depth search which examines continuations until a tactically quiet leaf node is reached is known as a quiescent search. Examples of non-quiescent situations where the search is extended are capturing moves, moves which put a king in check, forks and moves where a piece becomes "pinned". Other programs include different reasons for deepening a search; HITECH [10] will go a level deeper for each neutralizing recapture of a piece, and BEBE [37] will look ahead for a pawn move into the opponent's first three rows.

The DEEP THOUGHT 1987 program uses an innovative approach of variable-depth search. Their "singular extension" heuristic will select the move which is most "likely to affect the outcome of the search if its value changes" for an extended search. This heuristic, unlike other common deepening heuristics, is not domain-specific to the game of chess. It can be applied to any two-person adversary tree since the selection of a "singular" move is based on the score returned to the node and not on the features of the move. Even if

the heuristic sometimes doubles the number of nodes searched for tactical positions, "the overhead usually pays for itself handsomely" [2].

When only capturing moves are searched and all other moves treated as terminal, the resulting narrow and non-uniform tree is defined as a capture tree. A study by Bettadapur [6, 7] dealt with the application of the alpha-beta algorithm to capture trees. Bettadapur demonstrated that ordering the captures from major to minor pieces and using minor pieces for capturing them first offered several orders of magnitude of performance gains over other orderings. In fact, any extra effort spent in ordering moves was shown to be beneficial.

2.5 Transposition Tables

Throughout the course of a game tree search some positions often reoccur at other branches as a result of a transposition of moves. These frequently encountered positions can be stored in a hash table along with pertinent information obtained after searching their subtrees. When these same positions reappear in the tree, the values stored in the table may be used to narrow the alpha-beta bounds, or suggest which is the best move to search first, or if the previous subtree search was complete, eliminate the need to search again. The latter purpose is the primary purpose of a transposition table since it enables large portions of the tree to be pruned without any fear of losing information. Hence, potentially valuable gains are reached at a very modest cost since hash table management introduces negligible computing overhead.

Information on every node searched throughout the tree is kept in the hash table. As illustrated in past literature [22, 25, 23], a minimal hash table entry typically consists of the following components:

KEY Contains an encoding of the position which also includes; whose turn to move, castling and en-passant possibilities.

MOVE The best move at that position, determined from the subtree search results.

SCORE Value of subtree returned at that position.

FLAG Indicates if the score returned is a true score or an upper/lower bound due to an alpha-beta cutoff.

L_PC Length of the principal continuation rooted at that position, in other words, depth of the subtree.

When a perfect match is established between the present position and the transposition table entry, one of the following scenarios may occur:

1. If L_PC is greater or equal to the depth of search remaining and if SCORE is an exact score, this value is returned.

2. If the above condition on L_PC holds, but SCORE is an upper/lower bound, it can be used to cause an immediate alpha-beta cutoff.

3. When L_PC is less than the remaining search depth the score is ignored and the search is carried out with MOVE searched first.

The potential advantage in the third case is that **MOVE** can once again prove to be the best move of the subtree. Choosing **MOVE** first will also guide the tree search towards other nodes contained in the transposition table thus taking full advantage of its available wealth of valuable information. Furthermore, if **MOVE** is the only move needed to be generated because of a subsequent cutoff, a complete move generation can be spared.

The preferred method of hashing in chess-playing programs was first presented by Zobrist [45]. His idea consisted of using a table of 12X64 random integers (plus a few extra entries to include castling and en-passant possibilities) which represented all the possible piece-square placements on the board. A hash code for a board position would then be obtained by performing an exclusive-or of all the integers associated with all the piece placements for that board. When a move was performed on the position P_i the new hash code P_j was obtained by simply exclusive-or-ing both the "from" piece/square I_f and the "to" piece/square I_t with P_i as shown:

$$P_j = P_i \; xor \; I_f \; xor \; I_t$$

This method is both fast and simple to implement, and offers a well-balanced distribution of hash table entries. The benefits of using transposition tables in this manner have proven to outweigh the management costs by a great margin.

While most programs start with an empty hash table after each move, a program developed by Slate [43] tested the effectiveness of keeping positions which were difficult to

evaluate in the transposition table for usage on other moves, games, or even playing sessions. This type of "rote-learning" algorithm permitted his program to "learn from experience" and not repeat previous tactical errors. This learning algorithm would prove to be an attractive addition to commercial chess programs since this would correct some of the rigidity and blind repetition characteristic of chess machines.

When memory size is at a premium refutation tables can be used instead of transposition tables. These tables hold refutation lines which are paths from the root to a leaf node which either produced a correct score or an upper bound. These tables are mainly used to guide progressively deepening searches and do so very efficiently. Refutation tables have also been used as a source of killer moves [1].

Aside from keeping track of move transpositions, hash tables have also been used to hash scores of pawn structures and also for keeping record of king safety [27].

2.6 Iteratively Deepening Search and Aspiration Search

When carrying out a depth-first search to a fixed depth, the chosen depth often proves to be a poor choice. If a time constraint is imposed on the search, a depth which is chosen too shallow forces the program to move unnecessarily quickly. Alternately, choosing a depth which is too great may result in an incomplete search since not all the moves at the root will have been investigated when time expires.

To solve this problem of time and depth control, Slate and Atkin introduced iteratively deepening search [42]. Rather than carrying out a single search to a fixed depth, a series of deeper and deeper searches were carried out until time expired. The information collected during the $D-1$ ply search was used to guide and accelerate the following D ply search. The cost of an iteratively deepening search (measured in terms of the number of terminal nodes scored) is given by the equation;

$$IDAB(B,D) = IDAB(B,D-1) + E(B,D)$$

where $E(B,D)$ is the expected cost of an alpha-beta search of a tree of depth D and branching factor B given a principal continuation of $D-1$ moves obtained from a $D-1$ ply search. The fact that this continuation will often prove to be the prefix of the D-ply principal continuation usually more than compensates for the previous $D-1$ iterations.

Since the first branch searched will most probably return the best continuation, the number of alternate branches examined is greatly reduced.

A major benefit of using iteratively deepening search lies in the associated use of the transposition table. As each new iteration is carried out, the table is used to retrieve pertinent information from previous searches and guide the search towards subtrees previously examined. The table thus becomes filled with valuable scores and moves which become reused and updated in future searches. For chess programs which use transposition tables, Marsland and Campbell [22] estimated that the value of $E(B, D)$ is approximately;

$$E(B, D) \cong AB(B, D) + (B - 1) * AB(B - 1, D - 2)$$

for $B > 20$ and $D > 4$. Large improvements in search time are shown in the midgame performance of the program BELLE. In fact, "it normally costs BELLE a factor of 5-6 to go one further ply, i. e. less then the expected cost of optimal alpha-beta" [44]. Therefore, $IDAB(B, D)$ is normally less than the cost of a regular D-ply search initiated from scratch.

Normally, each subsequent search is executed to a depth one ply deeper than the previous search but a program named L'EXCENTRIQUE was developed incorporating a 2-ply iteratively deepening search. This approach tried to eliminate any inconsistencies introduced by the alternation of attacking and defensive continuations returned by successive even-odd ply searches. However the effectiveness of this variation was not made clear [22], and is not used by the current leading programs.

Aspiration search reduces the size of the tree searched by using a minimal alpha-beta window [22, 34]. Instead of initially setting the alpha and beta bounds to $[-inf, +inf]$, a narrow window of $[V - w, V + w]$ is used to improve the algorithm's performance. V is the expected value of the tree and w is the window size (usually the value of a pawn). When used with iteratively deepening search, the score returned from an iteration is used as the center (V) of the window for the next iteration. There is a gamble involved since it is not guaranteed that the value of the tree will fall inside the window; the search may fail high (i. e. the true value of the tree may be higher than expected) or fail low (i. e. a lower value than anticipated). In the above cases the tree must be searched again with a half-infinite window of $[V + w, +inf]$ for the fail-high case and $[-inf, V - w]$ for the fail low case. When a re-search is done after a fail-high situation, a minimum of, $B^{\lfloor D/2 \rfloor}$ nodes must be looked at. After a fail-low situation, a minimum of, $B^{\lceil D/2 \rceil}$ nodes must be looked

at. An aspiration search usually fails only at positions where pieces are gained or lost. As a whole, aspiration searches have proven to be very effective and involve no implementation cost. The idea of aspiration search has also been successfully applied in parallel programs [23, 31].

2.7 Killer Heuristics

When moves are refuted during the course of a search, it is often due to the same one or two moves. The purpose of all killer heuristics is to identify these frequent "killer" moves and order them near the top of a node's move list in order to accelerate the occurrence of a cutoff. Another simple heuristic which is analogous to killer heuristics is the ordering of capturing moves at the top of the list. Gillogly was one of the first researchers to incorporate both the ordering of captures and killer moves in his program.

> Since the refutation of a bad move is often a capture, all captures are considered first in the tree, starting with the highest valued piece captured. This is an inexpensive process, since captures can be recognized and sorted during move generation. The killer heuristic is also used: if a move is a refutation for one line, it may also refute another line, so it should be considered first if it appears in the list of legal moves [12].

Gillogly's version of the killer heuristic is a simple version where information on killer moves is shared only among nodes at the same level in the tree. A typical implementation will keep track of the best one or two killer moves for each level of search [42].

Other more informed implementations of the killer heuristic attempt to gather information on killer moves from all relevant parts of the tree rather then just the same level. One such approach was discussed by Akl and Newborn [1] where the principal continuation array was used as source of killer moves. In their paper, both a level-dependent killer heuristic and a level-independent heuristic were tested with the level-independent strategy showing a better performance on average.

Johnathan Schaeffer's *history heuristic* [35, 36] is an even more generalized version of the killer heuristic. This scheme uses two 64X64 tables to keep count of every legal move's ability to either cause a refutation or return a good score. A "weight" is kept for each entry

in the table. When a move proves to be good (i.e. causes a cutoff or backs up a score) the weight associated with that move is increased by 2^d where d is the depth of the subtree rooted at the current node. When moves are generated they are ordered according to the information gathered in the history tables.

All the above killer heuristics are examples of static or *fixed ordering* techniques where moves are ordered for search when a position is generated and this ordering remains fixed for the duration of the search. Another class of move ordering heuristics deals with concept of *dynamic ordering* [41, 29]. When a move M at level D causes the refutation of a move at level $D-1$, at levels $D-2$, $D-4$,... the move lists should be dynamically reordered so that M becomes repositioned at or near the top. This strategy enables killer move information deep in the tree to be available for use at the shallow levels.

The subject of *dynamic ordering* was first introduced by Slagle and Dixon in 1969 [41] when they used a kalah-playing program to compare the results of *fixed ordering* versus *dynamic ordering*. Their version of the *dynamic ordering* procedure was slightly different where only nodes at levels 1 and 3 were dynamically reordered according to results obtained deeper in the tree. Their work lead to the conclusion that *dynamic ordering* offered a modest improvement over *fixed ordering*.

Although killer heuristics have been widely used, their effectiveness have never been very clearly defined. Gillogly's TECH program [12, 13] reported no improvement at all while others —CHESS 4.5 [42], OSTRICH [30], and BLITZ [16] —have claimed varying degrees of computational gain. No one has safely claimed that the potential gains of killer heuristics account for all of the overhead. As Schaeffer himself states;

> The effectiveness of the killer heuristic has been questioned. Gillogly observed no benefits, whereas Hyatt claims significant reductions in tree size—to as much as 20% the original size. This is a popular alpha-beta enhancement, yet no one really seems to know how effective it is [36].

This thesis will hopefully clarify this issue and provide some positive insights on the real advantages of implementing selected versions of killer heuristics.

Chapter 3

Implementation of Move Ordering Heuristics

3.1 Three Classes of Heuristics

Most simple move ordering heuristics can be grouped into three general classes. With these three categories we do not attempt to contain all existing move ordering heuristics. We do however, try to include most of the more popular heuristics used in chess to locate and reorder moves which are considered "killers".

The simplest class of ordering heuristics are those which select killer moves on the basis of results observed at other branches but at the same level only. These heuristics are usually based on very trivial concepts and present the lowest amounts of overhead during execution as well as during their implementation. Heuristics included in this category usually use very short lists (1 or 2 moves) and depend on simple counting methods in order to assign respective scores of pruning effectiveness to all legal moves.

The next category unites all heuristics which attempt to distribute information to other subtrees as well as to deeper and/or shallower levels of the search tree. So far, there have been different versions of such heuristics, each displaying varying degrees of success. The use of the transposition table has been a popular means of gathering information at a particular level in order to eventually guide the search at shallower (and possibly deeper) levels. Experiments by Akl and Newborn [1] have demonstrated the possible benefits of using the principal continuation array in order to find potential killer moves not only at the

same level D as the current move, but also at levels $D+2$, $D+4$, $D+6$, etc. This strategy was shown to be an improvement over a simpler strategy which restricted the searching of killer moves to only nodes which were at the same ply.

The *history heuristic*, conceived by Schaeffer in 1983 [35], uses a history table which maintains "scores" for every legal move from each side. These scores indicate the accumulated effectiveness of a move's "killer" ability throughout the whole search. A high score indicates a high number of cutoffs and/or a significant number of cutoffs at very shallow depths by the same move. So far, the advantages of the history heuristic have not been convincing. Schaeffer himself states that the table becomes flooded with information at deeper levels of the tree. In order to limit the performance losses of his heuristic, Schaeffer had to limit the depth of the information actually stored in the table.

> ...the results reported here were constrained to only save history information within the first 5-ply of the tree, but use the information throughout the entire tree. This constraint provided a small improvement in performance, confirming that history tables can become flooded with information, decreasing their usefulness [36].

The first two classes of heuristics described so far are two examples of game-independent heuristics. Such heuristics can be applied to all two-person games which produce minimax adversary trees since they depend solely on the values of scores returned as well as the effects of the alpha-beta algorithm. A third class of ordering heuristics can be defined as those which are specifically game-dependent. There are many different heuristics used which depend mainly on the properties of chess especially in the areas of quiescence and variable-depth search. In the area of killer move ordering, the most popular chess-dependent heuristic included in chess programs is the ordering of capturing moves at the top of all move lists. This heuristic is normally very simple to implement and can be very effective, especially when the capturing moves are ordered according to the weight of the captured pieces [7, 6].

3.2 The FIXAFAN Chess Program

In our work, we chose eight move ordering heuristics which were all implemented within the chess program FIXAFAN, designed especially for the purposes of this thesis. The

program was written in C on a Solbourne 5/600 dual processor composed of SPARC chips. This skeleton chess program is an implementation of the iteratively deepening alpha-beta algorithm with the addition of search windows to perform aspiration search.

The chosen width of our search window is 2 pawns on each side of the predicted root score (i. e. if score returned after iteration $N-1$ is S, then iteration N will be a search with a window of $(S-2, S+2)$). Thus, a search will be successful only if the score returned differs from the previous iteration score by at most $+/-1$. When developing the FIXAFAN program, wider window sizes were experimented with, but on the average the substantial number of extra cutoffs induced by a narrow window was much more desirable than a low frequency of re-search situations obtained using a wide window. The smallest difference between the values of two different positions is 1 since that is the value of one pawn and the scoring function used is exclusively based on material possession. This enables the values of positions to be incrementally updated after each move is done and undone. A material scoring function also allowed us to search the last level of each branch by expanding only one move, the best capture, for all nodes at the next-to-last ply. This scoring function does not explicitly recognize mates but a check-mate is found implicitly when the principal continuation for a position returns a score reflecting an unavoidable king capture. The king must therefore be physically captured before a check-mate situation can be recognized (see results in Table C.1 of appendix C). When a king is captured, the score given to the move is the score of the king capture minus the depth of the capture. This ensures that the "best" check-mate continuation is returned in such a situation. One should always keep in mind that the scoring function used was material-based when interpreting all results and comparisons presented in this thesis.

FIXAFAN does not perform extended search along non-quiescent lines of play. Each iteration is carried out to a fixed depth of search. This not only made the program easier to develop, but more importantly made the interpretation of all statistical results easier to understand and also simplified the calculation of all averages and performance comparisons. A hash table of over half a million entries was implemented in order to detect and handle transpositions of moves and the occurrence of identical positions. With this hash table, a search will sometimes be cutoff at a node deep in the tree using the information gathered from an earlier identical node at a shallower depth. The effect of this is thus an artificial means of extending a search past the search depth for the deeper of the two identical nodes.

-m# where '#' > 0, is the maxdepth at which the program will stop its ID search if timelimit is not reached first. Default = 10.

-t# where '#' > 0, indicates the timelimit in real seconds after which the search stops if stopdepth is not reached first. Default = 120.

-w# where '#' > 0, defines the width of the search window in pawns. Default = 2 (2 pawns).

-l# where '#' > 0, creates an 'active' board for iteration number # showing all moves being updated and restored as long as the RETURN key is pressed.

-p# prints out the search tree for iteration number #.

-s# steps through iteration number # by stopping after each branch from the root has been searched.

-g plays a complete game interactively with the user.

Figure 3.1: Execution options used by FIXAFAN

Our move generation routine is very simple and linear. It uses as many register variables as possible in order to minimize the number of operations needed to generate all moves from a piece. When implementing our program, the main goal was not an optimal speed of search because of obvious restrictions imposed by the machine used. We were more concerned with the development of a reliable skeleton program which would help us in observing the effects of killer heuristics.

When executing the FIXAFAN program, a number of useful execution options are available to the user. These options are listed in Figure 3.1. Note that the options **-l**, **-p**, and **-s** were implemented for debugging purposes.

In addition to these options there are the 8 killer heuristics. Among the eight heuristics chosen, three are simple level-restricted heuristics, four are heuristics which use information throughout the entire tree, and the final heuristic is the chess-dependent ordering of capturing moves. The remainder of this chapter is devoted to the full description of these eight heuristics.

3.3 Simple Level-Restricted Heuristics

A simple level-restricted heuristic has the following two characteristics: First, all information collected during the course of a search at ply D will be used to reorder moves generated at other nodes at ply D only. Second, for each level of search, a small list of the best one or two candidates for a killer move is kept, lists of more than two moves in most cases do not seem to increase the heuristic's performance. In our research, three heuristics were implemented which correspond to the above criteria. These three strategies were named **LAST_1**, **LAST_2**, and **BEST_KM**.

The **LAST_1** heuristic simply keeps track of the most recent move which caused a cutoff for each level of the tree. When a certain move is searched and the returned subtree score causes a refutation at ply $D-1$, that move is stored in an array **HIGHEST[maxdepth]** at position D. When the GENMOVES( ) routine is invoked at another branch and at ply D, each move generated is compared with **HIGHEST[**D**]**, if a match is found the move is placed at the top of the move list. This heuristic is the simplest to implement and requires the least amount of overhead.

LAST_2 works in a similar fashion as LAST_1 with the added complexities of maintaining a list of the last two killer moves for each ply instead of just strictly the last one. We now use a double list **KMLIST[maxdepth][2]** to keep track of the two most recent killer moves. The most recent move M which caused a refutation at a node of ply D is kept in **KMLIST[**D**][0]** and the most recent move not equal to M which caused a cutoff at the same ply is stored in **KMLIST[**D**][1]**. Hence, whenever GENMOVES( ) is called, if the move stored at **KMLIST[**D**][0]** is generated, it will be placed at the first position of the move list. If the second last move **KMLIST[**D**][1]** is produced, it is placed at position two. By taking this idea one step further and maintaining a list of the last 3 or 4 moves at each level, we hoped to improve on the performances of the LAST_1 and LAST_2 heuristics. However we noted no improvement and in fact a slight increase in execution time was confirmed when compared with the simpler one and two move versions. We therefore restricted this general idea to the LAST_1 and LAST_2 strategies.

The **BEST_KM** strategy maintains a count of which move caused the greatest number of alpha-beta cutoffs for each level of search. In order to record the frequency of "kills" for each legal move of both sides, a table **KMTABLE[8][8][8][8][2]** is used. This table

is addressed by using the two coordinates of the FROM square and the two coordinates of the TO square, each of which range between 0 and 7. The extra array dimension is used to distinguish between black and white player moves. Each entry in the table represents a legal move and contains a count of the total number of refutations caused so far by that move. Whenever a move's frequency of "kills" surpasses the current leader for ply D, the old move is replaced by the new "best killer" at position D of array **HIGHEST[]**. Once again, when GENMOVES( ) produces a move which matches with the entry at **HIGHEST[D]**, that move is immediately placed at the top of the list.

3.4 Using the Transposition Table

Information accumulated by transposition tables can provide a potentially effective means of reordering moves at other nodes of the tree. The use of a transposition table can hence help in the distribution of knowledge throughout an entire search tree. When a refutation occurs, or a move returns a "good" score, the move is stored in the transposition table as that position's "bestmove". When a position is retrieved from the table, the "bestmove" is tried first to see if it will result in a cutoff. If a cutoff does not result from searching this move, all remaining moves for the position are generated and searched.

This strategy is named **BEST_TT** for our program. Hopefully, the desired consequence of using this strategy will be a reduced number of calls to GENMOVES( ) which generate all the moves at a node. Often, the amount of computation required to generate a complete set of moves from a position does not justify itself since a refutation often occurs after searching only a few moves. As with all other ordering strategies in our program, the BEST_TT is an option that can be invoked or suppressed. When the BEST_TT strategy is bypassed, the transposition table will still function normally but it shall not be used to help reorder any generated list of moves.

3.5 Killer Move – Parent Move Association

By establishing an association between every move which causes a cutoff and its parent move, we are able to reorder a move list using the outcomes of searches done at other positions which were not exactly identical but tactically similar in nature. This is the

ASSOC strategy and provides an interesting alternative to the use of the transposition table for identification of repeated scenarios in a game tree. This strategy first appeared in Marsland's AWIT program [22]. Whereas a transposition table will associate a potentially good move with a distinct position, the **ASSOC** heuristic associates a good move with an opponent's previous move. For any chess player, this would seem to be a very logical strategy. Positions may differ slightly from one line of play to another but often a move will be answered with the same reply in almost all of these lines of play. A capturing move almost always will be countered by the same recapture of the capturing piece. Obvious moves like these shall not be suggested by a transposition table if the position differs from an earlier position by a pawn movement or some other tactically irrelevant piece displacement.

The implementation of the **ASSOC** feature requires a table of *killer successors* or children–for each possible parent move. A table **KMCHILD[8][8][8][8][2]** is hence maintained so as to hold a potentially good move for each of all legal moves by both sides. This table is structured and addressed just like the **KMTABLE** structure used for the BEST_KM heuristic. When a position is updated from ply D to ply $D+1$ using a move M, the best "child move" K of M is retrieved from the KMCHILD table and placed in the array **HIGHEST[maxdepth]** at position $D+1$. When the new position is expanded with the GENMOVES( ) routine, each move generated is verified for a match with the entry K in HIGHEST[ ]. When the correct killer move is found, it is immediately promoted to the top of the list. When a different move K_2 later becomes a alternate "killer child" for the same parent move M, it simply replaces the old entry K in KMCHILD. There is no accounting made of the frequency of occurrence of different "killer successors" for a specific parent move.

3.6 Dynamic Move Ordering

The concept of dynamically reordering moves refers to the process of returning to an interior node after exploring some of its moves and then reordering its remaining unsearched move list using killer moves found two (or 4, 6,...) levels deeper [21, 29]. The **DYNAMIC** option uses the same data structures used for the LAST_2 option but with a different approach. The **KMLIST[maxdepth][2]** array stores the two last moves for each level of search, but these moves shall be used two levels higher up in the tree. When the search

returns to a previously generated node at depth D after the RESTORE( ) routine, the DYNAMIC heuristic will first verify to see if there are any moves left to search. If so, then the remaining list is scanned in hopes of locating either move **KMLIST**$[D+2]$**[0]** or move **KMLIST**$[D+2]$**[1]**. If the best move **KMLIST**$[D+2]$**[0]** is found, it will be repositioned as the next move to be examined. If the second best move is located and the best move was not located, then the second best move will be placed as the next branch.

The DYNAMIC heuristic thus offers a *true reordering* of moves at a node. So far, we have been using the term "reorder" to describe the notion of moves being generated in an alternate order other than what would be the normal output of the GENMOVES() routine. In the case of this heuristic, a node's move list will not be reordered during its generation, but rather at a later stage of the tree search.

3.7 The History Heuristic

When Schaeffer's *history heuristic* was introduced [35], it offered an innovative way to distribute information collected during a search to all branches and levels of the tree. Schaeffer's heuristic was included in our program under the name **HISTORY**. Its implementation requires a table of scores **KMTABLE[8][8][8][8][2]** just like the table implemented for the BEST_KM heuristic. The difference lies in the manner in which the scores are gradually updated. In order to reflect an estimate of the "value" of a move, Schaeffer chose to consider the depth of the subtree cutoff by a move as a measure of its "killer power". Therefore, when a move M at depth D causes an alpha-beta cutoff, the entry corresponding to M in KMTABLE will be incremented by a factor of 2^{MAXD-D}, where $MAXD$ is the maximum depth of search. This ensures that cutoffs which occur at shallow depths of the tree are given greater importance than cutoffs deep in the tree.

When moves are generated by GENMOVES( ), each new move is verified in KMTABLE to see what is its current value. Whenever a move has a higher HISTORY value than the previous high, it is immediately placed at the top of the list. A move which has a second highest score so far is placed at the list's second echelon. When the move generation routine is complete, the two highest valued moves according to KMTABLE shall be the two top seeds of the output list.

3.8 Ordering of Capturing Moves

To complete the set of chosen heuristics, the **CAPTURES** option was implemented to analyze the effects of using a chess-dependent move ordering strategy. All other heuristics previously mentioned are based on the occurrences of alpha-beta cutoffs and also on the scores returned by subtree searches, these heuristics are in no way restricted to chess game tree applications. The ordering of capturing moves at the top of move lists is a popular chess-dependent heuristic and has proven to be very efficient according to past research [6, 7, 13].

This heuristic takes place entirely during the GENMOVES( ) routine. Each time a move which captures an opponent's piece is generated, it is inserted at its proper location according to the value of the piece captured. When all moves have been generated, all the capturing moves are at the front of the list loosely ordered from biggest piece captured to all pawn captures. The ordering of captures from biggest piece to smallest piece was shown to be very effective according to work on capture search by Bettadapur in 1986 [6].

The CAPTURES heuristic was implemented in such a manner that it could be coupled with any of the seven other heuristics in order to observe the performance of a chess-dependent heuristic combined with chess-independent heuristics. When the use of another heuristic forces the top capturing move to be bumped out of the first (or second) position, the capturing move is shifted to the second (or third) place and the following capture move is relegated to the end of the CAPTURES list. The CAPTURES heuristic thus becomes a second-order heuristic while the other game-independent heuristic takes first precedence.

Chapter 4

Performance Comparisons

With the eight heuristics--BEST_KM, LAST_1, LAST_2, ASSOC, BEST_TT, DYNAMIC, HISTORY, and CAPTURES--implemented in the FIXAFAN program, experiments were set up to evaluate the relative performances of each one in chess game applications. Some of these heuristics were combined in order to verify the possibilities of improved performance using 2 or more strategies. The methods employed and measurements made to record these experiments are presented here.

4.1 Test positions and heuristics

The experiments were made using the Kopec positions, a set of 24 test positions that have been widely used to evaluate chess program performance since their appearance in 1982 [18]. Other experiments were also carried out using some of the more recent 25 test positions of a 1985 compilation by Kopec, Newborn, and Yu [19]. For each test position, the 22 executions listed in figure 4.1 were performed to test all ordering heuristics and selected combinations of heuristics. Note that a *no heuristic* option is included here so that each single heuristic can be evaluated according to its improvement factor relative to the non-enhanced alpha-beta algorithm.

All these tests were repeated many times in order to obtain satisfactory averages for each position which would give an honest and true account of the performances achieved in each case.

Single Heuristics		Combinations	
CODE	DESCRIPTION	CODE	DESCRIPTION
0	no heuristic	ck	CAPTURES + BEST_KM
k	BEST_KM	c1	CAPTURES + LAST_1
1	LAST_1	c2	CAPTURES + LAST_2
2	LAST_2	ca	CAPTURES + ASSOC
a	ASSOC	cb	CAPTURES + BEST_TT
b	BEST_TT	cd	CAPTURES + DYNAMIC
c	CAPTURES	ch	CAPTURES + HISTORY
d	DYNAMIC	cbk	CAPTURES + BEST_TT + BEST_KM
h	HISTORY	cb1	CAPTURES + BEST_TT + LAST_1
		cb2	CAPTURES + BEST_TT + LAST_2
		cba	CAPTURES + BEST_TT + ASSOC
		cbd	CAPTURES + BEST_TT + DYNAMIC
		cbh	CAPTURES + BEST_TT + HISTORY

Figure 4.1: 22 tests for each chess position

4.2 Measurements

4.2.1 Nodecount

A measure of search performance can be obtained by counting the number of bottom positions or *leaf nodes* of the game tree [41, 22, 23]. For constant-depth trees where all leaf nodes are at the same level, it can be shown that this number dominates the total number of nodes in a tree. However, trees generated by the FIXAFAN program have a non-negligible percentage of leaf nodes which are not at the limited depth of search because of branches amputated by the results of the transposition table. Since the evaluation of leaf nodes in our program is a simple count of the material value for both sides, the amount of computation required at leaf nodes is in reality almost negligible. By counting only the number of bottom positions, one thus treats all leaf nodes equally and also assumes that the interior nodes are negligible. Hence the number of leaf nodes is not strongly correlated with the actual running time of our program.

Counting the number of interior nodes in the search tree is the best way to evaluate the performance of the FIXAFAN program in a manner which is strongly correlated to the program's running time. Therefore, in our experiments all results which are expressed with, or use the term **nodecount**, refer the total number of interior nodes over all completed iterations. These are the nodes where the bulk of the computation takes place since that is where GENMOVES( ) is performed and where the moves are ordered by the selected heuristics.

4.2.2 CPU Time

Another important measure for comparing heuristic and algorithm performance is the actual CPU time consumed. This measurement is not a true reflection of actual search efficiency because timing results are machine dependent and usually vary depending on system load conditions. In the FIXAFAN program two UNIX[1] functions were used to assist us with the timing aspects. The GETTIMEOFDAY( ) routine was used to record the amount of real time elapsed and GETRUSAGE( ) was used to record the total time of CPU usage. All tests were conducted during nighttime hours to take advantage of minimal system load. With the number of interior nodes searched and the total CPU time we can obtain the rate of interior nodes searched per CPU second. This nodes-per-second value shows the relative speed (or loss of speed) of a heuristic compared to other heuristics.

4.2.3 Average Branching Factor

The Average Branching Factor (ABF) of a node in a tree is a reflection of the tree's "bushiness". Search trees produced by chess-playing programs are generally "bushy" with B, the number of branches, usually much higher than the depth D. Normally, D ranges from 5 to 10 levels, while B is much higher. The ABF of a typical midgame position was evaluated to 38.0 by DeGroot in 1965 [9]. This figure was later revised to include the occurrence of in-check positions, it was then shown that the ABF over midgame positions was 37.36 and the ABF over complete games was 32.83. In our program the ABF value is used to show the average of moves actually searched from a node rather than the average of moves generated. The ABF value shall therefore illustrate a heuristic's ability in narrowing tree width.

4.2.4 Killer Average

Theoretical analysis Theoretically, for a fixed depth game tree of depth D, there is a certain maximum number of interior nodes at which an alpha-beta cutoff is possible. Due to the essence of the algorithm it is not possible to obtain alpha-beta cutoffs at all interior nodes. By determining the maximum number of cutoff points in a tree of depth D we can also set an upper bound on the theoretical maximum value of its killer average—the ratio

[1] UNIX is a trade mark of AT&T Bell Laboratories.

of alpha-beta cutoff points versus the total number of interior nodes at which moves were ordered.

Let us assume a theoretical model of a game tree of uniform fanout B. Analyzing a subtree of depth 2 we see that $(B-1)$ of the B interior nodes at ply 1 could produce cutoffs (Fig. 4.2). For a tree of depth 3 we have the same 2-ply subtree as the first branch of the root. A backed up score at the root will permit cutoffs at the $(B-1)$ other ply 1 nodes. Thus a total of $2(B-1)$ cutoff points are possible for 3-ply trees (Fig. 4.3). For 4-ply trees we simply determine the additional number of cutoff points possible on the $(B-1)$ other branches from the root since the first branch represents the 3-ply tree. We see that for each of these branches there is a possible cutoff at ply 1 and B cutoffs at ply 3 (Fig. 4.4). This adds $(B+1)(B-1)$ nodes to our total for 3-ply trees thus an accumulated sum of B^2+2B-3 nodes. As we can see in figure 4.5 another $(B+1)(B-1)$ cutoff points are added to this total for 5-ply trees when examining the (B-1) other root branches once a score has been returned from the first move. This brings the total to $2B^2+2B-4$ cutoff points.

It is now easy to notice a pattern in the number of extra cutoff points added to a tree when searching an additional ply. When the depth of search D is even, we have;

$$B^{(D/2)-1}(B-1)+B^{(D/2)-2}(B-1)+\cdots+(B-1)$$

extra possible cutoff points added to the total for ply $D-1$. When D is odd the same number of extra cutoffs as for $D-1$ is added to the previous total. Following these guidelines, we obtain the following totals for the optimum number of cutoffs points in a tree of fanout B:

DEPTH	Number of Cutoff Points
1	0
2	$B-1$
3	$2B-2$
4	B^2+2B-3
5	$2B^2+2B-4$
6	B^3+2B^2+2B-5
7	$2B^3+2B^2+2B-6$
EVEN	$B^{(D/2)}+2B^{(D/2)-1}+2B^{(D/2)-2}+\cdots-(D-1)$
ODD	$2B^{\lfloor D/2\rfloor}+2B^{\lfloor D/2\rfloor-1}+\cdots-(D-1)$

If the above maximum number of cutoffs are successful, and if these cutoffs all occur after searching the first branch of the cutoff node, then the resulting tree will have the minimum number of interior nodes theoretically possible. A re-analysis of the same trees

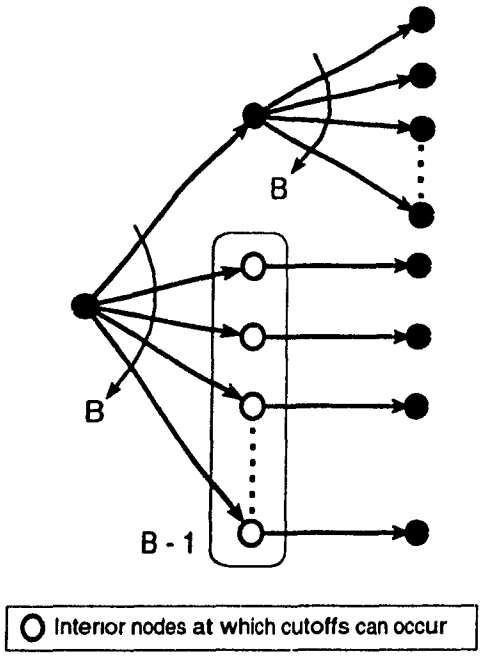


Figure 4.2: Minimal alpha-beta tree of depth 2

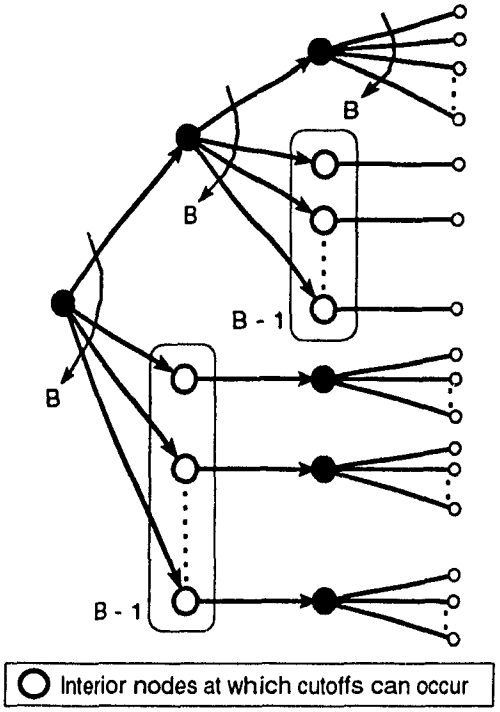


Figure 4.3: Minimal alpha-beta tree of depth 3

shown in figures 4.2 - 4.5 shows that for trees of uniform fanout B we again note a repeating pattern for the number of additional nodes added to this minimum total for each extra ply searched:

For EVEN plies, the minimum number of extra interior nodes is;

$$B^{(D/2)-1}(B-1) + 2\,B^{(D/2)-2}(B-1) + 2\,B^{(D/2)-3}(B-1) + \cdots + 1$$

For ODD plies, this number becomes;

$$2\,B^{\lfloor D/2 \rfloor - 1}(B-1) + 2\,B^{\lfloor D/2 \rfloor - 2}(B-1) + \cdots + 1$$

The addition of these number gives us the minimum possible number of interior nodes for trees of uniform fanout B:

DEPTH	Number of Interior Nodes
1	1
2	$B+1$
3	$3B$
4	B^2+4B-1
5	$3B^2+4B-2$
6	B^3+4B^2+4B-3
7	$3B^3+4B^2+4B-4$
EVEN	$B^{(D/2)} + 4\,B^{(D/2)-1} + 4\,B^{(D/2)-2} + \cdots - (D-3)$
ODD	$3\,B^{\lfloor D/2 \rfloor} + 4\,B^{\lfloor D/2 \rfloor - 1} + \cdots - (D-3)$

If we now take the ratio of maximum cutoff points over minimum interior nodes, we obtain the highest possible percentage of killer move ordering effectiveness. We can see that for trees of EVEN depth as the fanout increases, this ratio approaches 1/1. But as DEPTH increases, the ratio gradually fades away from this 1/1 bound. For trees of ODD depth, the maximum ratio is close to 2/3 for large fanouts. But as the trees start to deepen, this optimum ratio gradually decreases as well. Therefore, we theoretically expect killer moves to cause a cutoff at over 90% of all interior nodes in the best case for trees of EVEN depth and at almost 66.7% of all interior nodes of ODD-depth trees. However, in practice these idealist performance results are never attained and in contrary to theoretic expectations, "good" heuristics seem to perform better with deeper trees than with shallow trees due to increasing usage of search information. Therefore, need more realistic means of measuring the true effectiveness of a move ordering heuristic.

Practical analysis The killer average (KA) is an arbitrary method of interpreting a heuristic's "batting average". It uses a count of the total number of "kills". A *kill* is when

a cutoff occurs within the subset of moves reordered by the killer heuristic at a node. If a certain heuristic or combination of heuristics reordered N moves at the front of a node's movelist, and a cutoff occurs after searching one of these N opening moves, then a proper *kill* is registered. The KA is thus defined as:

$$KA(D) = \frac{\#\ of\ kills\ at\ level\ D}{\#\ of\ nodes\ at\ level\ D}$$

In our experiments, ABF and KA values were calculated for each level of search and averaged over all completed iterations. This offers the possibility to see where a heuristic seems to be most effective; at even levels or odd levels, deep in the tree or at shallow depths near the root. However an important consideration must be taken into account when interpreting these figures. Due to the fact that FIXAFAN evaluates only material balance at all leaf nodes, the last level of an iteration is not expanded and searched thoroughly as with all previous levels. Instead, the highest capture is immediately taken as the best move and all other moves are ignored. The consequence of this shortcut is that the ABF for the last level of every iteration can never exceed 1.00. It shall be slightly below 1.00 because of transposition table pruning and also occasional terminal checkmate positions. The KA value for the last level of an iteration shall also be very high—above 0.85—because of this modification in the basic algorithm.

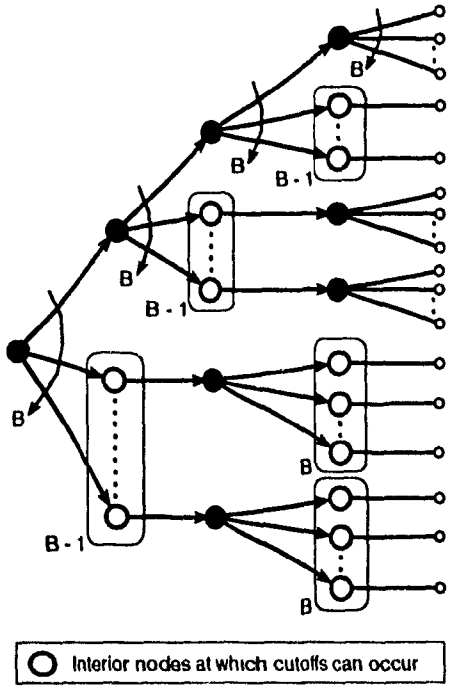


Figure 4.4: Minimal alpha-beta tree of depth 4

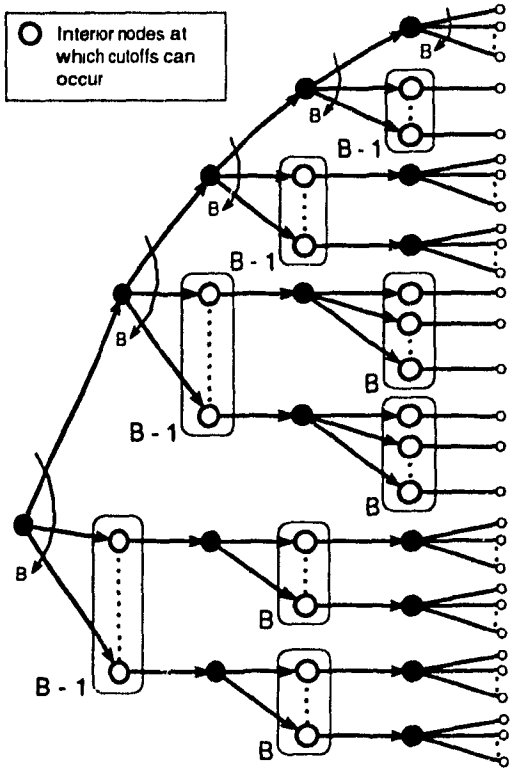


Figure 4.5: Minimal alpha-beta tree of depth 5

Chapter 5

Experiment Results

In this chapter we shall discuss the results of our experiments with the eight move ordering heuristics described in chapter 3. These results are summarized in three types of comparisons:

1. experiments involving just one of the eight heuristics,
2. experiments using CAPTURES combined with another heuristic, to see the effects of a game-dependent heuristic combined with game-independent heuristics,
3. BEST_TT & CAPTURES, the two most effective heuristics, combined with a third heuristic.

This chapter shall illustrate the efficiency of the alpha-beta algorithm when used with these heuristics in terms of execution time, rate of execution, and the sizes and fanouts of the game trees searched.

5.1 How Results Were Compiled

The 22 heuristic tests described in section 4.1 were performed a minimum of four times for each of the 24 Kopec test positions in order to obtain satisfactory averages for the execution times as well as for the rates of interior nodes searched per second. With each of the 24 test positions, a maximum search depth was selected so as to ensure that an iterative deepening alpha-beta search up to that depth would most likely terminate within 1 to 3 minutes of CPU time on average. For mid-game positions, a maximum of 6 or 7 plies was

found to be adequate and for end-game positions, a search up to 9 plies was performed. Thus, the 24X22 test executions were not all completed to the same maximum search depth but instead were set up to terminate within the same CPU time range. Finally, for each of the 22 heuristic tests, the results for the 24 Kopec positions were grouped and averaged in order to globally summarize the performance of each test. The results for CPU time and interior nodes reflect the heuristics' efficiency in reducing the amount of game tree states searched, the rate of interior nodes per second shows the computational speed lost due to heuristic management overhead, and the width or "bushiness" of the game tree is reflected by the average branching factor as well as the killer average

5.2 Experiments Using Single Heuristics

The performance of each heuristic is compared to the null heuristic {0} as a reference for the evaluation of relative gain/loss in search effectiveness.

Among the eight individual heuristics tested in solo, the most impressive results were obtained by the BEST_TT {B} heuristic which offered the best results in average time and size of tree searched. However one should note that the average time figure is also dependent on the rate of interior nodes per second searched. The rate of search for BEST_TT proved to be considerably faster than all the other heuristics because of the "TRYMOVE" feature. When a position is retrieved from the transposition table and the information associated with it is insufficient to warrant a refutation, the "bestmove" stored in the table will be generated along with all other moves from that same piece. The remaining bulk of legal moves from that node shall not be generated unless the search of the "bestmove" subtree does not succeed in producing a cutoff. The considerable savings in computation time is thus confirmed by this very high search rate.

The ordering of capturing moves {C} produced the narrowest trees since its resulting Average Branching Factor proved to be the lowest overall value. The CAPTURES heuristic also registered the highest Killer Average with successful move orderings at over 80 percent of all interior nodes. This strategy proved to be second best to the BEST_TT heuristic in both average time and interior nodes expanded. This demonstrates that capturing moves are often good moves even though they are selected without the usage of "previous" search results. In other words, the CAPTURES heuristic can be applied "blindly" without in-

	Int.nodes	Avg.time	Int.nds/sec	A.B.F.	Kill.Avg.
{0}	442306.7	189.10	2297.39	5.01	44.06 %
ASSOC {A}	222323.1	103.53	2132.67	4.55	56.93 %
DYNAMIC {D}	181647.0	92.99	1939.14	4.45	65.96 %
HISTORY {H}	397259.5	180.54	2122.21	5.18	67.89 %
BEST_KM {K}	304770.4	138.46	2185.80	4.87	51.17 %
LAST_1 {1}	199351.7	86.83	2238.60	4.58	65.43 %
LAST_2 {2}	181518.1	85.74	2067.03	4.56	65.65 %
CAPTURES {C}	144444.3	63.15	2278.76	4.07	80.41 %
BEST_TT {B}	137905.7	48.13	2531.66	4.19	69.74 %

Table 5.1: Results for 8 solo heuristics

	Int.Nodes	Ratio	Avg.time	Ratio	A.B.F.	Ratio
0	442306.7	1.0	189.10	1.0	5.01	1.0
H	397259.5	0.898	180.54	0.955	5.18	1.034
B	137905.7	0.312	48.13	0.255	4.19	0.836

Table 5.2: HISTORY & BEST_TT versus null heuristic

quiring about the performance of a move at the same ply or at deeper and/or shallower plies with a great deal of success. It is therefore a very efficient *uninformed* move ordering strategy (Tab. 5.1).

The most inconsistent and often least effective of the eight heuristics was the HISTORY {H} heuristic. Schaeffer's approach to move ordering placed last in the criteria of interior nodes searched with a reduction of just 19.0% of the nodecount for {0} (Tab. 5.2). When compared with the seven other solo heuristics, HISTORY {H} had the third slowest rate of interior nodes per second and the biggest fanout by a considerable margin, even bigger than {0}. However, it is interesting to note that its Killer Average was third highest after both CAPTURES {C} and BEST_TT {B} respectively. The unusual combination of high Killer Average coupled with high Average Branching Factor can be explained by the fact that the HISTORY heuristic works often (i. e. the reordering of moves often produces a cutoff) but it will rarely cause a cutoff after the first one or two moves at any node. Thus, Scheaffer's heuristic most often produces its desired effect much too late in a move list to be in noticeably beneficial to a search.

Comparing the three simple level-restricted heuristics BEST_KM, LAST_1, and LAST_2, we see that LAST_1 {1} and LAST_2 {2} proved to be far more useful and worthwhile to implement than BEST_KM {K}. The LAST_2 {2} heuristic was slightly better than LAST_1

{1} in all aspects except for rate of search which is understandable due to a slight increase in overhead required to maintain a 2-element list. In fact, LAST_2 {2} proved to be second best of the chess-independent heuristics—since CAPTURES is the only chess-dependent heuristic—with third best solo results in both average time and interior nodes.

Among the heuristics which are level-independent, the DYNAMIC {D} and ASSOC {A} methods showed relatively good improvements in average time, interior nodes, and A.B.F. when measured against {0}. The DYNAMIC {D} heuristic placed fourth best in interior nodes searched amongst the eight solo heuristics, as well as scoring the third lowest A.B.F. value with ASSOC {A}, LAST_2, and LAST_1 close behind. When analyzing the results for DYNAMIC {D} we see that this heuristic was quite efficient in reducing the tree size but unfortunately required too much overhead to be optimally time-efficient. In fact, it's rate of interior nodes per second was the lowest of the eight heuristics. A more time-efficient implementation of this heuristic would have elevated its performance up to the level of the LAST_2 heuristic since DYNAMIC {D} is simply a level-independent version of LAST_2.

5.3 Performance of Heuristics Coupled With CAPTURES

In the following comparisons we shall use as reference the performance of the CAPTURES {C} heuristic. In this way we shall be able to determine if the coupling of a game-independent heuristic with CAPTURES {C} offers any supplemental gains in search efficiency. In these experiments, the game-independent heuristics have highest priority when deciding where a move will be ordered at a node. This means that the CAPTURES heuristic is set up as the "second-choice" condition for move ordering.

As illustrated in table 5.3, improvements on CAPTURES' results were confirmed by the combinations {CB}, {CA}, {C1}, and {C2}. Combining BEST_TT {B} with CAPTURES {C} produced the fastest average time amongst all these combinations as well as the lowest number of interior nodes. The second best average times were achieved using the LAST_1 {1} and CAPTURES {C} combination, and the second lowest total of interior nodes was obtained with the LAST_2-CAPTURES combo. Thus, {C2} generated smaller trees than {C1} but was slower than {C1} because of loss of speed due to overhead. Results for {CA} were also encouraging, average time was slightly higher than for {C2} and its A.B.F. was

	Int.nodes	Avg.time	Int.nds/sec	A.B.F.	Kill.Avg.
C	144444.3	63.15	2278.76	4.07	80.41%
CA	124555.1	59.03	2080.37	4.08	81.49%
CB	103926.5	35.52	2521.35	4.14	84.50%
CD	130499.4	68.66	1907.71	4.23	82.35%
CH	150772.8	73.16	2085.42	4.43	83.02%
CK	139482.7	64.63	2161.81	4.11	81.25%
C1	124182.3	56.45	2201.42	4.13	82.24%
C2	123167.2	60.87	2029.09	4.18	82.45%

Table 5.3: Heuristics coupled with CAPTURES

	Int.Nodes	T.T.Hits	Int.Nds : TThits
0	442306.7	122008.9	3.625
C	144444.3	40217.9	3.592
D	181647.0	50208.1	3.618
CD	130499.4	33597.1	3.884

Table 5.4: Transposition table hits for DYNAMIC

second best overall after CAPTURES {C}.

Results inferior to CAPTURES {C} were produced by the heuristic tests for {CH} and {CK}. The relative complexity of the DYNAMIC heuristic was again confirmed by the ambiguous results of {CD} which expanded 9.7% less nodes than CAPTURES but was 16.3% slower in rate of search– the slowest of all 22 tests. This suggests that a complete optimization of DYNAMIC's implementation would seem a worthwhile investment. Another interesting observation concerning {CD} is the fact that using these heuristics together guided the tree searches to the lowest overall ratio of transposition table hits to interior nodes searched. It should also be noted that as a solo heuristic, DYNAMIC {D} produced the smallest number of move transpositions. This suggests that dynamically reordering moves when backtracking to a node steers the tree search away from previously visited board positions hence restraining the transposition table's potential usefulness (Tab. 5.4).

The unimpressive results of using Schaeffer's HISTORY heuristic were once again evident when combined with CAPTURES {C}. The {CH} pairing had the worst average time of all pairings as well as worst A.B.F. and worst interior nodes total. Thus, even when coupled with CAPTURES {C}, HISTORY {H} does not seem to be a reliable move ordering heuristic. However, it should be noted that {CH} obtained the second highest

	Int.nodes	Avg.time	Int.nds/sec	A.B.F.	Kill.Avg.
CB	103926.5	35.52	2521.35	4.14	84.50%
CBA	82933.2	30.34	2329.23	4.29	84.75%
CBD	91407.6	36.91	2115.92	4.26	85.17%
CBH	106129.5	41.75	2173.34	4.41	82.62%
CBK	102945.0	36.79	2421.51	4.10	84.71%
CB1	93626.8	32.84	2449.05	4.22	85.09%
CB2	94090.1	34.29	2296.19	4.23	85.25%

Table 5.5: {CB} plus a third heuristic

Killer Average. This simply re-confirms the previously mentioned hypothesis which stated that HISTORY creates a high percentage of alpha-beta cutoffs but these cutoffs usually occur much later than at the first one or two moves of a movelist.

5.4 Combining a heuristic with CAPTURES & BEST_TT

For this third set of experiments, we observed the possible advantages of using the two best heuristics, BEST_TT and CAPTURES, combined with a third heuristic. The superiority of these two heuristics coupled together was confirmed by their dominance over all the other pairings involving CAPTURES. We therefore use the results for CAPTURES with BEST_TT as a basis for our comparisons. It is important to specify the order of priority of the heuristics used. The BEST_TT heuristic had highest priority. Therefore, a move stored in the transposition table was searched immediately before any other move at all nodes. The second highest priority was given to the other game-independent heuristic while CAPTURES had third level priority. These new results were consistent with the trends observed in section 5.3 (Tab. 5.5).

Improvements were obtained with the combinations {CBA}, {CB1}, and {CB2}. The best time performance and the lowest number of interior nodes for any heuristic or combination of heuristics was obtained by {CBA}. The second best average time result obtained by {CB1} was in large part due to its fast rate of computation {CB1} had the fastest nodes per second rate amongst the 6 heuristic trios. The third best was {CB2} which had a slower rate of computation speed and searched slightly more nodes than {CB1}. However this combination had the highest Killer Average among all heuristic tests.

The consistency with the scenario of section 5.3 was maintained when we observed

	Int.Nodes	Ratio	Avg.Time	Ratio	A.B.F.	Ratio
CB	103926.5	1.0	35.52	1.0	4.14	1.0
CBA	82933.2	0.798	30.34	0.854	4.29	1.036
CBH	106129.5	1.021	41.75	1.175	4.41	1.065

Table 5.6: {CBA} & {CBH} versus CAPTURES + BEST_TT heuristics

	Int.nodes	Avg.time	A.B.F.	int.nds/sec.
0	442306.7	189.10	5.01	2297.39
Best ratio	CBA: 82933.2 (0.188)	CBA: 30.34 (0.160)	C: 4.07 (0.812)	B: 2531.66 (1.102)
Worst ratio	0: 442306.7 (1.0)	H: 180.54 (0.955)	H: 5.18 (1.034)	CD: 1907.71 (0.830)

Table 5.7: Best and worst performances

the inferior performances of groupings {CBK}, {CBD}, and {CBH}. The below standard results of {CBK} were mainly due to the poor Killer Average registered by BEST_KM. Throughout all three sets of experiments, BEST_KM proved to be a weak level-independent heuristic because of its low Killer Average. The {CBD} trio had a low rate of interior nodes per second which remains true to the trend set by {CD} and {D}. A reduction of the overhead produced by our implementation of DYNAMIC would probably raise the level of {CBD} near the top of its class since this heuristic combination tallied the second lowest overall total of interior nodes searched. We noticed a clear increase in effectiveness for HISTORY when combined with BEST_TT {B} and CAPTURES {C} in contrast with the {CH} duo However the {CBH} trio was the least impressive of the 6 three-strategy combinations. It's Killer Average was the lowest and the A.B.F. was the largest of the 6 trios (Tab. 5.6)

A summary of the best and the worst performing heuristic tests in terms of interior nodes, average time, A.B.F., and rate of nodes per second are shown in table 5.7. Accompanying ratios show the relative increase or decrease when compared with the null heuristic test. We see that on the average over all 24 test positions, the {CBA} combination was able to cut down the size of the game trees searched by a factor of 81% and also achieved an 84% reduction in execution time. One can see the enormous potential of using killer heuristics such as these with a more time-efficient implementation on a faster processor.

	Depth 6 (~ 7)			Depth 7 (~ 8)			Depth 8 (~ 9)		
	K.H.	Nodes	Time	K.H.	Nodes	Time	K.H.	Nodes	Time
avg		34257.2	12.22		158650.4	57.39		900134.0	334.35
1	B	24099.4	7.92	B	88467.1	27.10	B	386149.3	130.90
2	A	30834.6	11.26	C	129661.0	45.86	C	752063.9	274.76
3	C	32772.5	11.38	D	133377.1	52.36	D	771838.5	308.99
4	2	33513.0	12.22	2	156959.7	57.60	2	778042.5	302.01
5	D	33711.5	12.92	1	157092.5	54.05	A	824811.7	300.34
6	1	34769.6	11.87	A	158567.5	57.67	1	845065.7	298.80
7	H	40923.8	15.49	K	212108.1	73.21	K	1193154.5	419.34
8	K	45433.1	14.74	H	232969.9	91.30	H	1649945.7	639.59

Table 5.8: Results of Depth Comparisons

5.5 Influence of Search Depth on Heuristic Performance

In order to observe if a deeper search would enhance the performances of certain heuristics we conducted another set of experiments involving eight of the 24 test positions. The positions chosen were the eight positions which produced relatively "fast" rates of search, and which therefore could be searched to a depth of at least 8 plies in 8 iterations within a reasonable amount of computational time. These were the Kopec positions 2, 3, 6, 8, 11, 14, 17, and 24. Note that positions 6 and 8 are end-game positions and thus were searched to a depth of 9 plies so as to terminate within the same time range as the other six positions.

The results of these tests involving the eight solo heuristics are shown in table 5.8. When we compare the results of each individual heuristic against the overall averages for each of the three search depths shown—numbers in parentheses are depths for positions 6 and 8—we see that some of the heuristics changed positions in the standings as search depth increased. We did not tabulate results for search depths shallower than 6 plies since these results did not illustrate much differences between the individual heuristics.

To better illustrate the characteristic trends displayed by some of the heuristics, we present a table of ratios which compares each heuristic's nodes total against the overall average for each depth (Tab. 5.9).

This table shows that most of the heuristics seemed to improve slightly in their effectiveness as the search depth increased. The exceptions to this trend were the HISTORY, ASSOC, and BEST_KM heuristics. The most obvious improvements were shown by the

	depth 6 (~ 7)	depth 7 (~ 8)	depth 8 (~ 9)
B	0.782	0.558	0.429
A	0.900	0.999	0.916
C	0.957	0.817	0.836
2	0.978	0.989	0.864
D	0.984	0.841	0.858
1	1.015	0.990	0.939
H	1.195	1.468	1.833
K	1.268	1.337	1.326

Table 5.9: Depth Comparisons: Ratios of Nodes to Overall Averages

BEST_TT heuristic which steadily improved its first place ranking in total nodes searched with deeper searches. This illustrates the increasing importance of using transposition tables with deep tree searches.

Notable improvements were also displayed by the CAPTURES and DYNAMIC heuristics. At shallow search depths, these two heuristics show very average results, but with deeper searches they show their superiority over the other heuristics except for BEST_TT. This trend can easily be understood with the DYNAMIC move ordering strategy since killer moves deep in the tree become potentially good "killers" at the shallower levels. The characteristics of the DYNAMIC strategy are not as apparent when this strategy is used on shallow trees.

There were two heuristics, ASSOC and BEST_KM, which seemed to show no improvements in performance with deeper searches while the HISTORY heuristic showed a steady decline in pruning efficiency. At search depths of 5-6 plies, the HISTORY heuristic compiled statistics which were close to the overall average. But for deeper searches, this heuristic gradually worsened its nodes totals to move past the BEST_KM heuristic and into last place. These findings are consistent with Scheaffer's own observations on his heuristic:

> From 5 to 6-ply, the percentage improvement attributable to the history heuristic decreases. There are two plausible explanations. The first is that the trees are getting so large that, although the history heuristic is just as effective as before, its performance relative to the larger tree seems poorer. ...A second reason is that the history heuristic tables are becoming over-loaded [36].

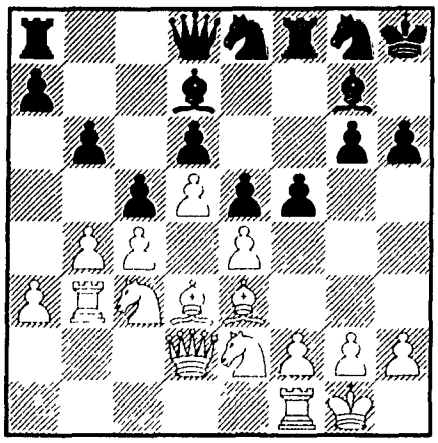
Since all our tests were carried out on trees of 6 to 9-plies, the HISTORY heuristic was rendered ineffective due to these large tree sizes. Schaeffer's heuristic seems better suited for applications with shallow trees of 5-plies or less.

We also examined the performances of heuristic combinations with deepening search depths. All the heuristic combinations seemed to perform better with increasing search depths. However, the individual characteristics previously discussed for the solo heuristics were not as evident when they were combined with CAPTURES or BEST_TT & CAPTURES. There were no major changes in the relative standings of these combinations as the search depth increased. These results were therefore considered inconclusive and were not included as part of this thesis.

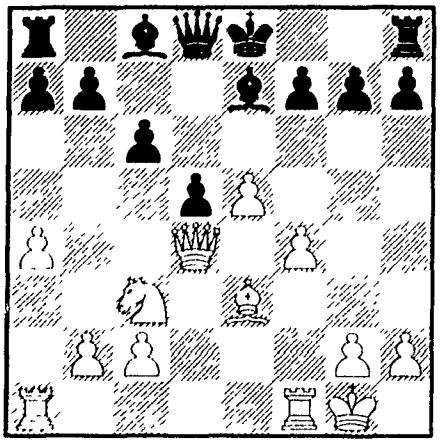
5.6 Correlation of Performance and Evaluation Function Used

So far the results that we have been reporting are simply a cumulative summary of all results for the 24 Kopec test positions. If we take a closer look at the performance of killer heuristics on a specific instance of a chess game, we see that the improvements precipitated by move ordering depend very much on the game situation. The effectiveness of a killer heuristic is dependent on several factors, however in our case the most significant influence on our heuristics came from our trivial scoring function. Since our program used a simple scoring function which evaluated just the material balance on both sides, our killer heuristics performed well on positions which offered good possibilities of captures, exchanges, or sacrifices. Quiescent or quiet positions did not produce satisfactory move ordering performances. The mobility of the pieces in a chess position also improved the power of killer heuristics but to a lesser degree. Among the 24 test positions used, some were very much quiescent while others were "volatile", these positions clearly displayed these performance characteristics. Their results are discussed here briefly in order in illustrate our relationship between heuristic effectiveness and non-quiescence of a chess position.

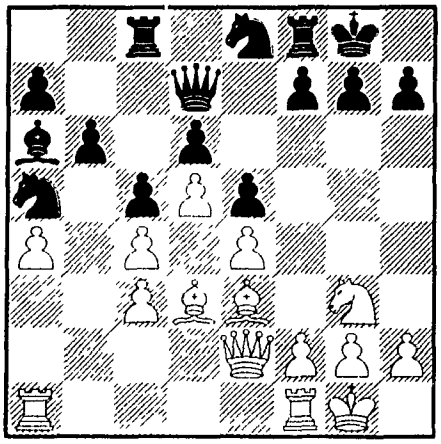
The least impressive performances were registered with Kopec board 24 (Fig. 5.1). One look at this position confirms that this is not much of a tactical position. Since there are few capturing possibilities here, our scoring function does not make much of a difference between the possible legal moves. It seems that any move would be as good as any other. Thus, the consequences of using killer heuristics with this position were not of much positive use, if fact they were often negative. Specifically, just 10 of the 21 heuristic combinations were better than the null heuristic in terms of time. The best of the tests, {B}, showed



Board 24: (WHITE to move)



Board 23:
(BLACK)

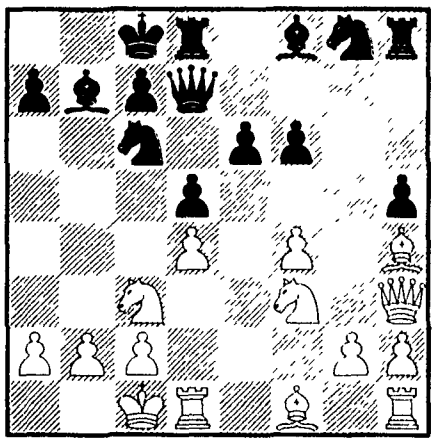


Board 11:
(WHITE)

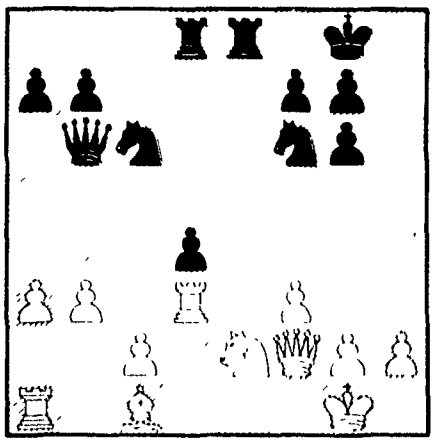
Figure 5.1: Relatively Quiescent Test Positions

a reduction in time of just 46.4%. These surprising results were confirmed when we used board 24 for our tests with varying depths. When this position was searched 1 ply deeper to a depth of 8 plies, 9 of the 21 heuristic tests took less time than the null heuristic while the effectiveness of BEST_TT was reduced to 36.6%.

Partially negative results of using killer heuristics were also attributed to positions 23 and 11 (Fig. 5.1). With position 23, 9 of the 21 heuristic tests were worse than the null test, and with position 11, {C11} used up 1.60 times more time to search the same tree as {0}. Once again both these positions show little opportunity for attack and appear to be a relatively quiescent chess positions. It should be noted that for all the test positions which



Board 9:
(WHITE)



Board 10:
(BLACK)

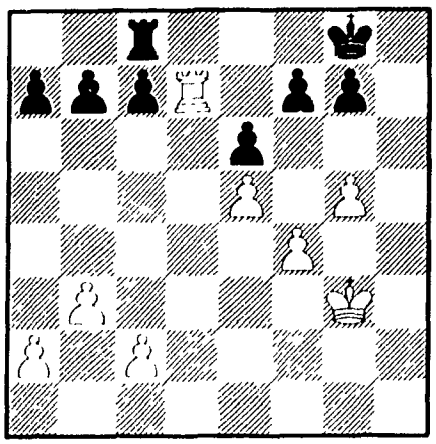
Figure 5.2: Examples of Tactically Stable Test Positions

precipitated adverse results when searched using killer heuristics, the CAPTURES heuristic was not able to reduce the time taken by {0}. This fact confirms our earlier statements concerning the importance of potential captures in tactical positions which encourage the use of move ordering heuristics.

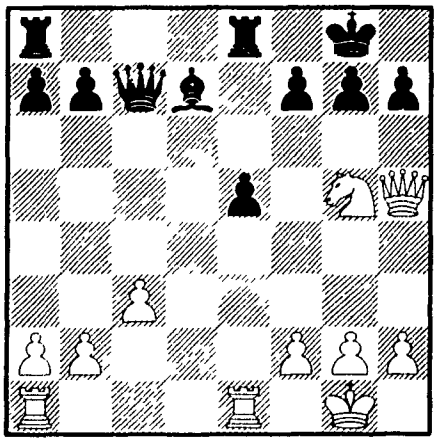
Although most of the test positions did not produce negative results when tested with killer heuristics, some positions showed relatively little gains in performance. Such was the case for positions 9 and 10 (Fig. 5.2). The best test result for position 9 gave a 45.5% reduction in time while position 10 at best showed a 47.5% reduction. These positions offered one or two potential exchanges in pieces but apparently not enough to warrant a substantial increase in search performance.

Some of the positions were neither too quiescent nor too tactically "volatile" and seemed to belong in the "middle of the pack". For example, the best-case time reduction for position 20 was 67.5% when compared with the null test. This and other positions with search time reductions of approximately 70% could be considered as an arbitrary average of performance for our heuristic tests if we eliminate all positions with extremist results — in other words all quiescent positions and all potentially game-breaking tactical positions.

The subset of non-quiescent tactical positions offered optimum conditions for the testing of our killer heuristics (Fig. 5.3). These positions usually present an obvious point of attack where the next move could make or break a game. Positions 6 and 12 would fit in



Board 6:
(WHITE)



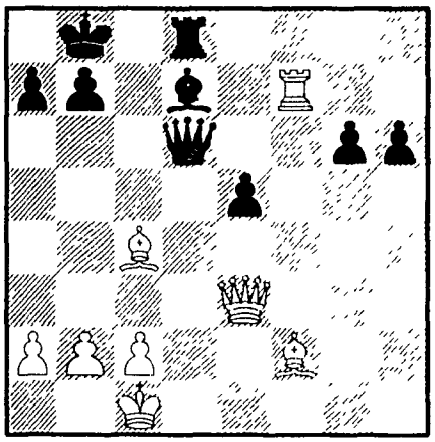
Board 12:
(BLACK)

Figure 5.3: Non-Quiescent Test Positions

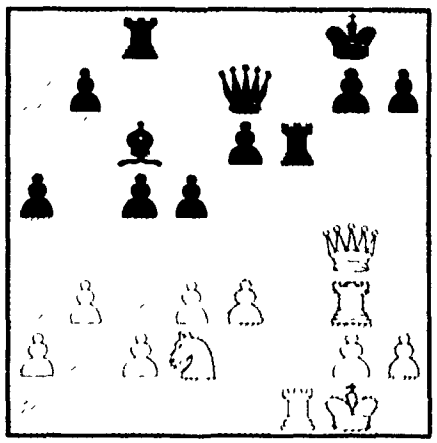
this category of positions and the results of their heuristic tests show very high gains in performance; a best-case 93.0% time reduction for 12, and an 91.7% reduction for 6. With our heuristics, non-quiescent chess positions would produce time reductions of roughly 90% and higher.

The largest improvements in search time where obtained with positions 1 and 15 (Fig. 5.4). Both were able to reduce the search time of the null test by over 95% with their best test result -- 96.1% for position 1 and 95.6% for position 15. These two positions are both volatile in appearance and imply a large number of captures and re-captures. Therefore, they are well-suited for the application of killer heuristics using our trivial material-based evaluation function. This example as well as all of the above-mentioned cases illustrate the strong correlation between killer heuristic effectiveness and the level of complexity of the scoring function used. If one decides to implement a more complex and sophisticated evaluation function then the importance of potential captures is reduced and killer heuristics thus become more applicable to a wider range of test positions.

One heuristic which performed generally well regardless of the tactical characteristics of the position was the BEST_TT heuristic based on the transposition table's 'bestmove' entries. This heuristic showed consistently positive results when used in solo or in combination with other strategies for all 24 test positions. For instance, if we look at two of the most quiescent positions—positions 23 & 24—which gave inferior results for about half of



Board 1:
(BLACK)



Board 15:
(WHITE)

Figure 5.4: Test Positions with Highest Killer Heuristic Results

the 21 tests, we see that the BEST_TT heuristic used alone managed to give reductions in time of 43.7% for position 23 and 46.4% for position 24. These were the two lowest results registered by this heuristic. This high level of consistency confirms the reliability of the BEST_TT heuristic since it guarantees significant savings in search time when used with any game position.

5.7 Profiling the FIXAFAN Program

Measuring the time spent in each part of the FIXAFAN program is an accurate way to locate any obvious defects in its implementation. A UNIX profiler can be used to obtain the execution profile of a program. This profile statistically shows the amount of time spent in each part of the executed program and also displays this time as a percentage of the total execution time. The UNIX profiler was hence used in selected executions to monitor the behavior of the FIXAFAN program.

The profiler was invoked several times on 4 different test executions 2 were complex mid-game positions and 2 were simple end-game positions. These executions are shown in appendix C and the averages for these profiles are shown in table 5.10. Board 1 is a late mid-game position. Although there are not that many pieces remaining, both queens are "open" in the sense that they both can produce a large number of moves at each interior node of the tree. In this case move generation is at its most expensive. A profile of the

	End-game positions		Mid-game positions	
	board 6{C}	board 8{CB2}	board 1{C1}	board 24{CH}
GENMOVES()	52.9%	51.4%	72.3%	71.1%
ALPHA_BETA()	12.0%	12.0%	7.8%	7.1%
UPDATE()	8.3%	6.8%	4.0%	3.3%
RESTORE()	4.1%	3.4%	1.3%	1.9%
Hash tab. func.	7.6%	8.7%	6.7%	4.2%
Killer mv func.	0.5%	1.6%	0.5%	0.4%
UPDATE_PC()	1.4%	0.8%	0.4%	0.4%

Table 5.10: Profiler results for 4 tests

{C1} test was chosen because it was this heuristic combination which produced the smallest cumulative total of calls to GENMOVES( ) among all the executions for board 1.

Position 24 reflects an early mid-game situation where both players still have full possession of all their pieces. The move generation procedure is quite expensive here due to the fact that almost all pieces from both sides have some mobility even though few immediate captures are possible. For this position the {CH} test expanded the largest number of interior nodes by a considerable margin and so the profile shows how little time is spent in the alpha-beta procedure when compared with other mid-game profiles.

Positions 6 and 8 are both end-game positions where there are an array of pawns left on both sides and only 1 major piece left for both players other than the king. The CAPTURES {C} heuristic test was profiled for board 6 to see the profile of an inexpensive but effective heuristic. We see that the amount of time spent in the killer move ordering department is much less than for the other end-game profile of a good heuristic test {CB2} on position 8. These two profiles show that end-game move generation consumed about 52% of the execution time on average compared to the 72% figure attained by typical mid-game positions. With end-game positions consisting solely of king and pawns, the profiler showed that move generation required less than 50% of the execution time.

The profiler also dissolved any lingering doubts we might have had concerning the implementation costs of using killer heuristics. All the functions related to killer move selection required between 0.5% and 1.5% of the execution time for most of the heuristic combinations. The one exception was the DYNAMIC heuristic, the DYNAMIC_ORDER( ) procedure usually demanded a little more than 6% of the execution time. This is because the ordering of moves is done outside the confines of the GENMOVES( ) procedure when

	Int.nodes	Avg.time	Int.nds/sec	A.B.F.	Kill.Avg.
1D	181809.0	81.98	2189.97	4.41	66.01 %
2D	181647.0	78.42	2291.89	4.45	65.96 %
1CD	129965.5	58.41	2174.36	4.22	82.35 %
2CD	123400.4	57.35	2270.51	4.23	82.35 %
0	442306.7	189.10	2297.39	5.01	41.06%
CBA	82933.2	30.34	2329.23	4.29	81.75%

Table 5.11: Tests using DYNAMIC, CAPTURES, LAST_1 & LAST_2

the search RETURNS to a node. With all the other heuristics, the ordering of moves is done as an integral part of the GENMOVES( ) routine.

5.8 Other Combinations

While assembling together the necessary data for the evaluation of our 22 different heuristic tests, we realized that the next logical step forward in our research was to enquire about the possible benefits of implementing other combinations than the ones discussed so far. After attempting some new "recipies" of heuristic combinations such as using DYNAMIC and CAPTURES with a third heuristic and investigating into the possibilities of combining four (or more) heuristics together, we soon realized that no more improvements in tree search could be attained.

The combination of DYNAMIC with any other heuristic was too slow in speed of search due to DYNAMIC's low rate of nodes per second. We show here the results of 4 tests involving DYNAMIC, CAPTURES, and the LAST_1 and LAST_2 heuristics. We see from table 5.11 that combinations {1CD} and {2CD} performed better than the {1D}, {2D} combinations. In fact, results for {1CD} and {2CD} were practically identical. In many cases, these two heuristic tests expanded exactly the same number of interior nodes. However, their performances were well below the standards set by the strongest heuristics such as {CBA}. The {1D} and {2D} pairings displayed some improvement when compared with the DYNAMIC heuristic in solo but showed only minimal gains over the LAST_1 and LAST_2 solo results. This exact same pattern is repeated when we compare the results for {1CD} and {2CD} with {CD}, {1C}, and {2C}. In fact, the {1CD} trio had slightly inferior results than the {1C} duo.

As with the set of tests involving the DYNAMIC heuristic, combining four heuristics

together introduced too much extra overhead which more than neutralized the modest gains obtained from any further reductions in tree size. The results of these supplementary experiments were not considered of any significant importance to our work so we preferred not to discuss them in this thesis.

Chapter 6

Conclusion

The work in this thesis provides a general overview of game tree search and a closer, more introspective look at move ordering heuristics in the context of chess programs. Eight different move ordering heuristics were implemented on a skeleton chess program. A series of tree search tests were performed to clarify and quantify the true effectiveness of these *killer heuristics.*

Among the eight heuristics chosen, three are level-dependent, four are level-independent, and one heuristic—the ordering of CAPTURES at the front of a node's move list is dependent on the properties of chess. CAPTURES on average cuts down the number of interior nodes to 33% of the total accumulated for the null tests and reduces the execution time to 33% the time taken by the null tests. When combined with any of the seven other heuristics, CAPTURES raises the efficiency of all these heuristics substantially.

The most reliable game-independent heuristic is the BEST_TT heuristic which also proved to be the most effective solo heuristic. BEST_TT reduces tree size by 70% and search time by 75% on average. This ordering heuristic is also very consistent since it showed strong results over the complete set of all 24 test positions.

Among the level-dependent heuristics, good performances were noted for the LAST_1 and LAST_2 strategies. Both these heuristics offer reductions of 40% to 50% in both interior nodes expanded and execution time when compared with the null tests. Increasing the number of killer moves listed per level to three or more moves does not seem to give any additional gains in tree search. Hence, for simple level-dependent heuristics it is preferable to implement small lists of 1 or 2 moves. The third level-dependent heuristic BEST_KM

does not offer significant gains in search performance.

Both the DYNAMIC move ordering and the parent-child move ASSOC heuristics are effective level-independent heuristics. They both reduce total nodes expanded and search time by about 50% on average. DYNAMIC ordering is more effective than ASSOC at reducing the tree size---its total for interior nodes expanded is as low as for LAST_2—however its overhead is non-negligible due to repeated reordering at each node and thus responsible for its slightly disappointing time results It remains to show that an optimized implementation of DYNAMIC move ordering would be as time-efficient as LAST_1 and LAST_2.

The HISTORY heuristic requires a little less overhead than DYNAMIC but does not reduce tree size noticeably. Consequently, HISTORY consumed just 4.5% less execution time on average than the null tests. Although this heuristic has a high cutoff rate, most of these cutoffs occur too late in a move list to be effective. A more sophisticated ordering of moves using the HISTORY tables would have been more beneficial but the extra overhead introduced would have nullified some, if not most, of these supplemental gains.

A third set of experiments were conducted to verify the combined effects of CAPTURES, BEST_TT, plus a third heuristic. Results for these tests were superior to the results obtained when combining CAPTURES with another heuristic—except with BEST_TT. Small improvements in the execution time of the CAPTURES + BEST_TT pairing were noticed when complemented by the ASSOC heuristic, as well as by the level-dependent LAST_1 and LAST_2 heuristics.

Results of a profiler show that most of the killer heuristics used take up a small percentage of the overall execution time--between 0.5% and 1.5%. The DYNAMIC and HISTORY heuristics are the only time-consuming exceptions to this. The best results in both execution time and tree size were registered by CAPTURES + BEST_TT + ASSOC which required only 16% of the time and expanded 19% of the nodes expanded during the null tests. These figures confirm that killer heuristics are inexpensive to implement and that a good choice of heuristics can improve tree search performances by at least 80% on average for trees of 6 to 9 plies.

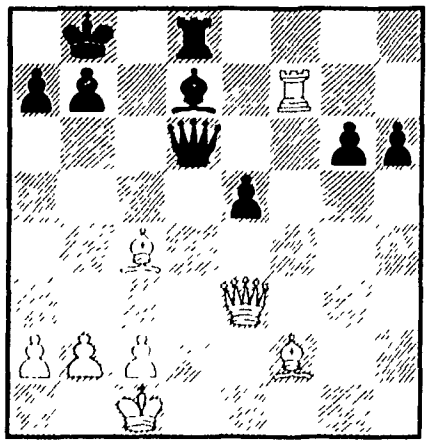
Some heuristics when used alone perform differently as the depth of search is gradually incremented. The BEST_TT heuristic became more and more effective as the search depth was increased from 6 plies to 8 plies on a selected subset of the test positions. This charac-

teristic makes BEST_TT and transposition tables an essential tool for searching deep trees generated by end-game positions. The CAPTURES, DYNAMIC, LAST_1, and LAST_2 heuristics also showed improvements with deeper search depths. The HISTORY heuristic's performance deteriorated with deeper trees due to an overloading of the history tables. This explains the disappointing ranking of this heuristic when compared to other heuristics which are positively depth-influenced.

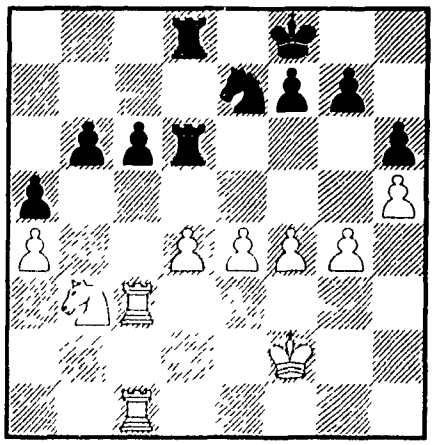
The deterministic abilities of the scoring function used has a strong influence on the performances of killer heuristics. The scoring function used in our skeleton program is strictly based on material possession. This accounts for the highly successful results of using the CAPTURES heuristic on positions showing unstable game situations. When used with quiet or quiescent positions, the ordering of CAPTURES is in fact a handicap to the search and precipitates adverse results. It therefore remains to model the effects of a more complex scoring function and thus show that killer heuristics can become applicable to a wider scope of positions when used in this context.

Appendix A

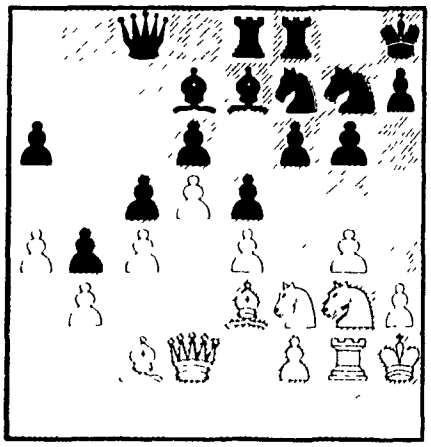
Bratko-Kopec Test Positions



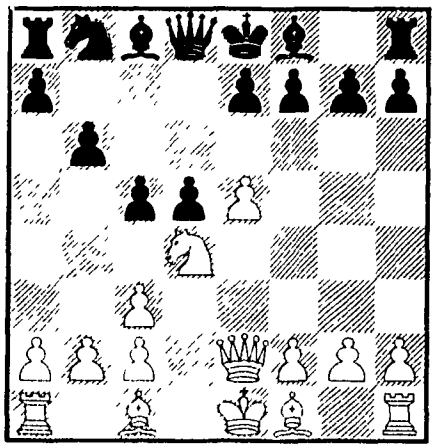
Board 1: BLACK d6–d1



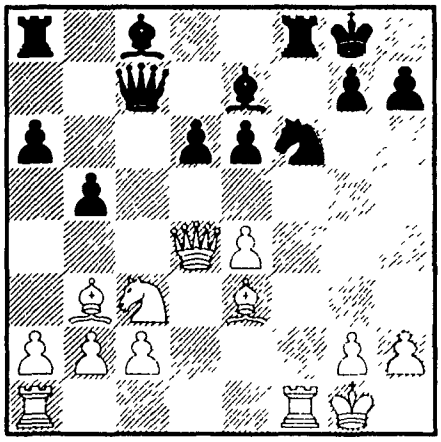
Board 2: WHITE d4–d5



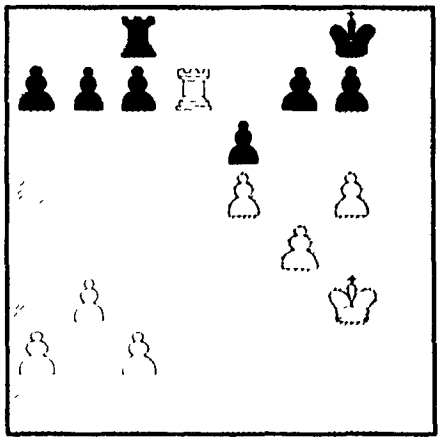
Board 3: BLACK f6 f5



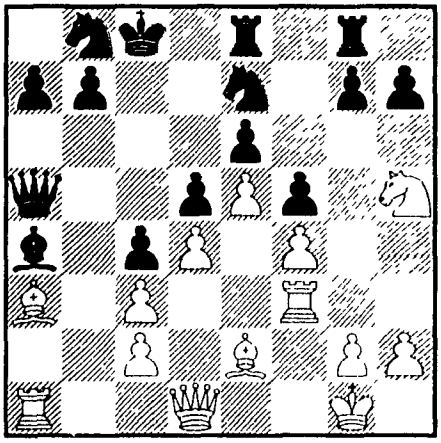
Board 4: WHITE e5–e6



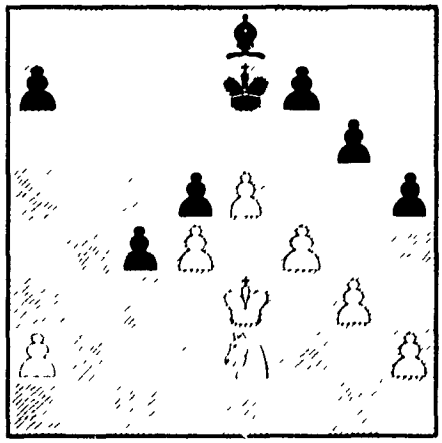
Board 5: WHITE c3-d5



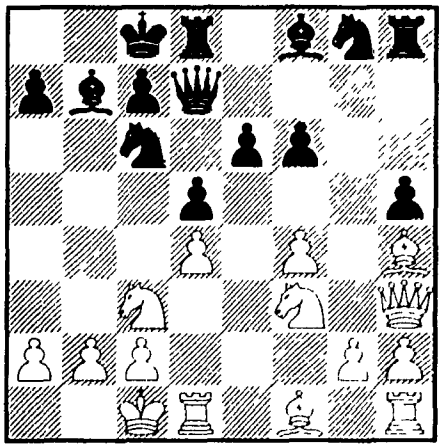
Board 6: WHITE g5 g6



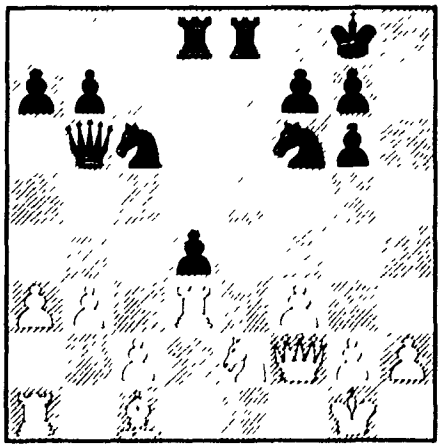
Board 7: WHITE h5-f6



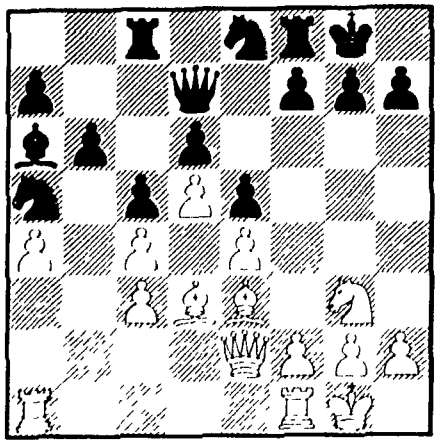
Board 8: WHITE f4 f5



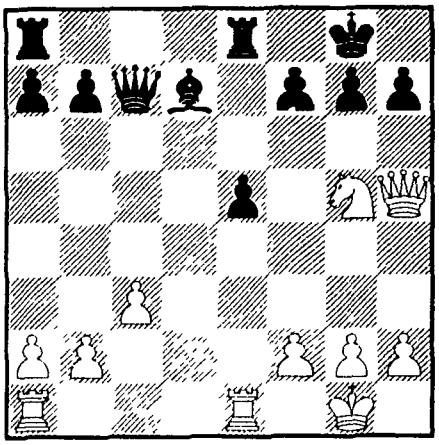
Board 9: WHITE f4-f5



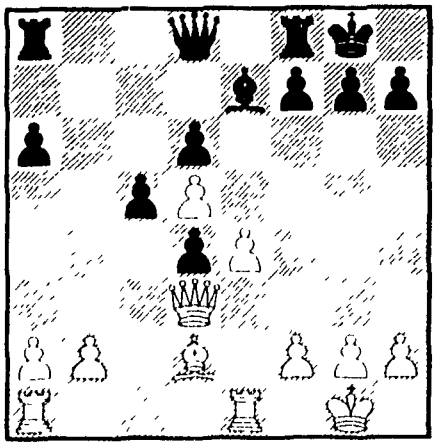
Board 10: BLACK c6 e5



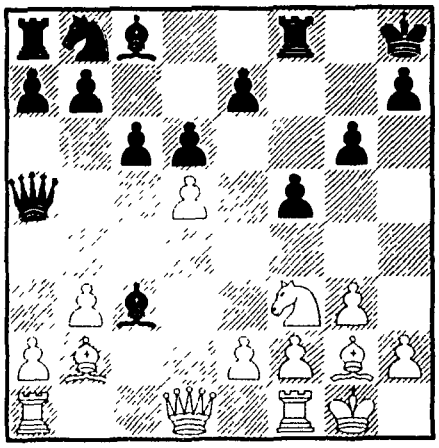
Board 11: WHITE f2-f4



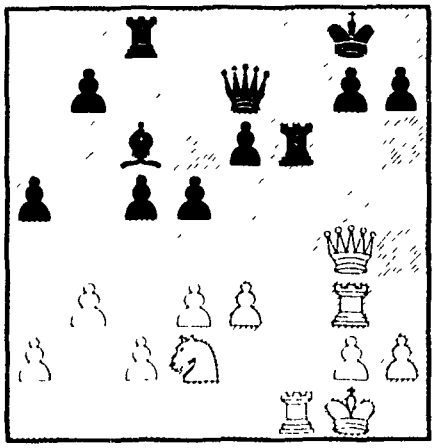
Board 12: BLACK d7-f5



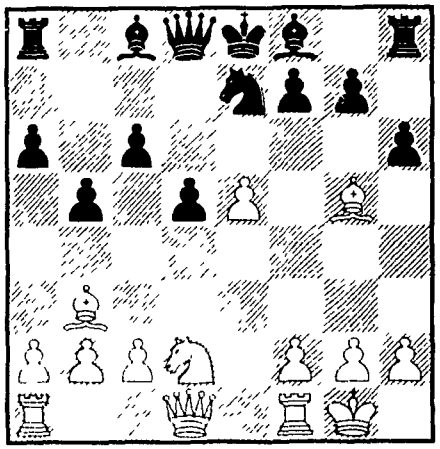
Board 13: WHITE b2-b4



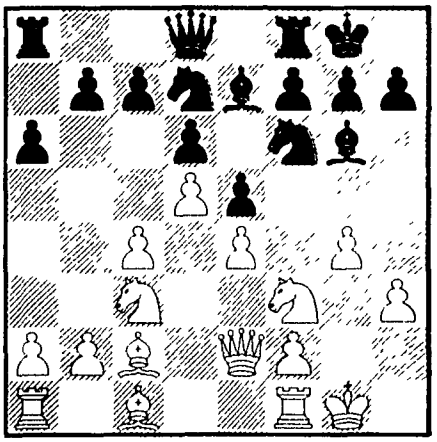
Board 14: WHITE d1–d2



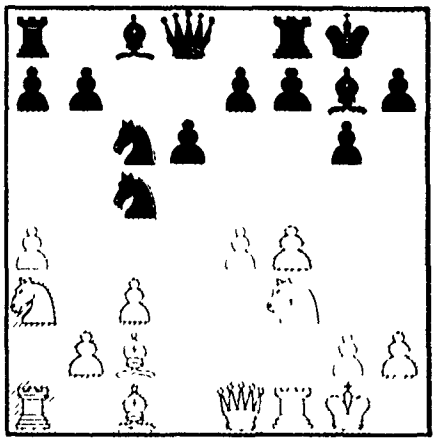
Board 15: WHITE g1-g7 ♟



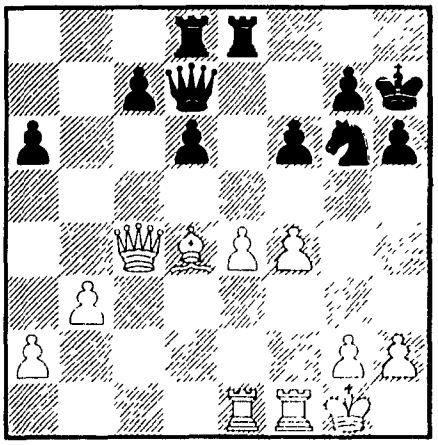
Board 16: WHITE d2–e4



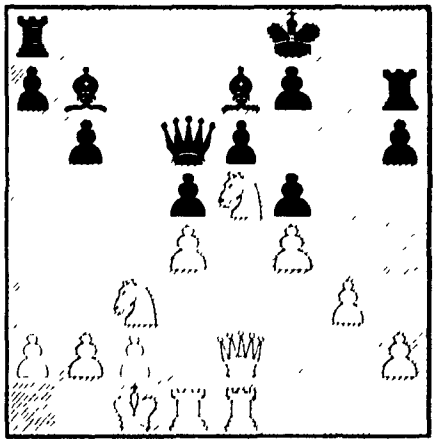
Board 17: BLACK h7-h5



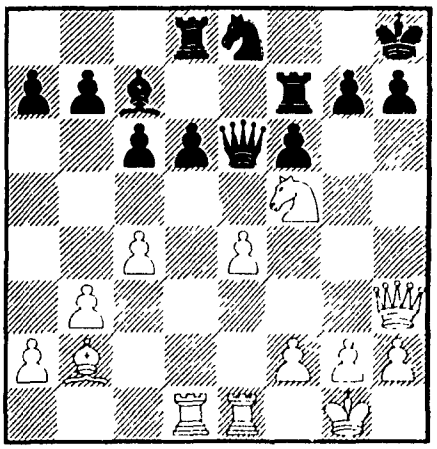
Board 18: BLACK c5 b3



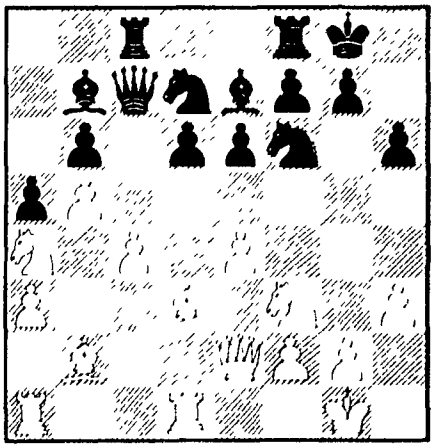
Board 19: BLACK e8–e4 ♗



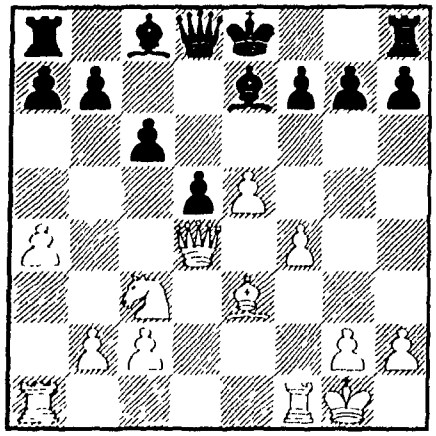
Board 20: WHITE g3 g4



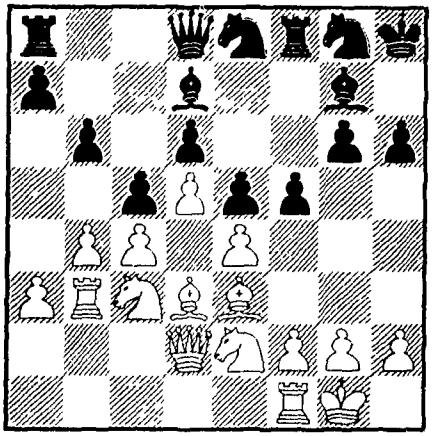
Board 21: WHITE f5–h6



Board 22: BLACK b7 e4 ♗



Board 23: BLACK f7-f6



Board 24: WHITE f2–f4

Appendix B

Experiment Statistics

	Int.nodes	Avg.time	Int.nds/sec	A.B.F.	Kill.Avg.
{0}	442306.7	189.10	2297.39	5.01	44.06 %
HISTORY {H}	397259.5	180.54	2122.21	5.18	67.89 %
BEST_KM {K}	304770.4	138.46	2185.80	4.87	51.17 %
ASSOC {A}	222323.1	103.53	2132.67	4.55	56.93 %
LAST_1 {1}	199351.7	86.83	2238.60	4.58	65.43 %
DYNAMIC {D}	181647.0	92.99	1939.14	4.45	65.96 %
LAST_2 {2}	181518.1	85.74	2067.03	4.56	65.65 %
CAPTURES {C}	144444.3	63.15	2278.76	4.07	80.41 %
BEST_TT {B}	137905.7	48.13	2531.66	4.19	69.74 %
1D	181809.0	81.98	2189.97	4.44	66.01%
2D	181647.0	78.42	2291.89	4.45	65.96%
1CD	129965.5	58.41	2174.36	4.22	82.35%
2CD	123400.4	57.35	2270.51	4.23	82.35%
CH	150772.8	73.16	2085.42	4.43	83.02%
CK	139482.7	64.63	2161.81	4.11	81.25%
CD	130499.4	68.66	1907.71	4.23	82.35%
CA	124555.1	59.03	2080.37	4.08	81.49%
C1	124182.3	56.45	2201.42	4.13	82.24%
C2	123167.2	60.87	2029.09	4.18	82.45%
CB	103926.5	35.52	2521.35	4.11	84.50%
CBH	106129.5	41.75	2173.34	4.41	82.62%
CBK	102945.0	36.79	2421.51	4.10	84.71%
CB2	94090.1	34.29	2296.19	4.23	85.25%
CB1	93626.8	32.84	2449.05	4.22	85.09%
CBD	91407.6	36.91	2115.92	4.26	85.17%
CBA	82933.2	30.34	2329.23	4.29	84.75%

Appendix C

Execution Examples

The output files of 24 test executions follow. These files are structured as follows:

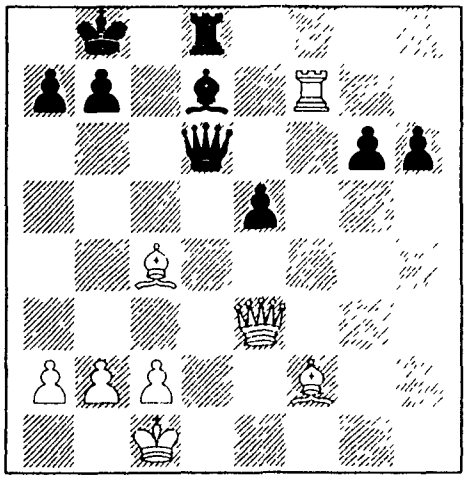
The top portion of the file indicates the Bratko-Kopec test position used, and the killer move heuristics used. When the program is run without any arguments, all the heuristics are "off" by default.

The middle section shows the test position on the left and a table of results on the right. This table has *DMAX* rows, where *DMAX* is the total number of search iterations performed. The first column shows the principal continuation found with the root move at depth 0. The leaf node score is shown at depth *DMAX*. Note that this score is normalised by one pawn value (+/ 1) in order to reduce the value of horizon captures. The second column lists the total number of interior nodes expanded after each completed iteration. The third column shows the average branching factor at each ply for the whole search. Note that the A.B.F. for depth *DMAX* - *1* is always 1 since only the best move is searched at the next to last level. The last column shows the killer average for each ply for the whole search. Note that the killer average for the next to last ply is not an honest result when the CAPTURE heuristic is used since the best move at that level is almost always a capturing move. A non-zero killer average at ply 0 indicates a registered "kill" for the CAPTURES heuristic at the root node on the first iteration. Below the test position and table, are the cummulative totals and averages for the whole search including total time and rate of interior nodes per second.

The bottom section shows the *DMAX* - *1* previous principal continuations displayed left to right as well as the time of completion for each iteration.

Board: BD.1 {C1} Killer Moves: Captures ON Assoc. off BestTT off
Dynamic off Last 1 ON Last 2 off
History off BestKM off

To Move: BLACK Depth: 7



Dp	Princ.Cont.	Nodes	A.B.F.	%kills
0	d6 d1		33.67	0.00
1	c1-d1 ♛	1	1.31	86.89
2	d7-g4	42	9.23	60.35
3	d1-c1	316	4.51	81.84
4	d8-d1	1781	6.81	65.46
5	c1-d1 ♜	8132	1.24	99.19
6	g4 d1 ♔	19195	1.00	84.16
7	score=84	321722		

Best move: d6-d1
CPU Time: 133.31 Int. Nodes: 321722 ABF Avg: 2.17
T.T. Hits: 33305 Nodes/sec.: 2413.31 % Kills: 93.69

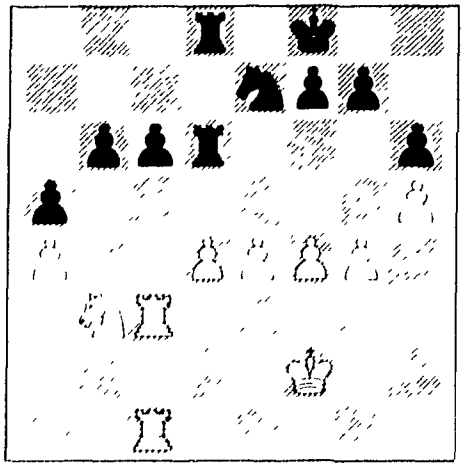
Previous Continuations:

It.	Time	0	1	2	3	4	5
1	0.00	e5-e4 .					
2	0.00	d7-c6 .	e3-e5 ♟				
3	0.12	d6-b4 .	c4-d3 .	b4 -b2 ♗			
4	0.76	d7-c6 .	e3-a7 ♟	b8-c8 .	f7-b7 ♟		
5	3.30	d7-c6 .	e3-a7 ♟	b8-c8 .	a7 e3 .	e5 e4 .	
6	21.12	d6-d4 .	e3-d4 ♛	e5 d4 ♕	f2 g3 .	b8 a8 .	f7 d7 ♝

Table C.1: Sample execution for board 1 with {C1} heuristics

Board: BD.2 {B} Killer Moves: Captures off Assoc. off BestTT ON
Dynamic off Last 1 off Last 2 off
History off BestKM off

To Move: WHITE Depth: 7



Dp	Princ.Cont.	Nodes	A.B.F.	%kills
0	e4-e5		28.43	14.29
1	d6-d5	1	1.87	72.22
2	f4-f5	35	15.50	50.30
3	d5-d7	150	1.63	80.16
4	c3-c2	1581	13.48	50.92
5	b6-b5	1023	1.37	85.43
6	a4-b5 ♟	24083	1.00	92.45
7	score= 0	60803		

Best move: e4 e5
CPU Time: 18.03 Int. Nodes: 60803 ABF Avg: 2.85
T.T. Hits: 29703 Nodes/sec.: 3372.32 % Kills: 80.78

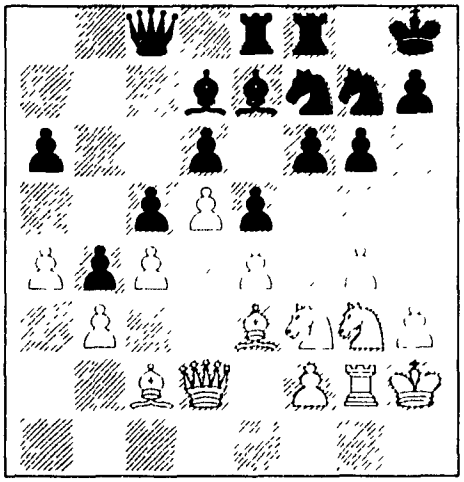
Previous Continuations:

It.	Time	0	1	2	3	4	5
1	0.00	b3 a5 ♟					
2	0.00	e4 e5 .	d6 d1 ♙				
3	0.06	e4 e5 .	d6 d5	b3-a5 ♟			
4	0.53	e4 e5 .	d6 d5 .	f4 f5 .	d5 d1 ♕		
5	1.26	e4 e5 .	d6 d5 .	f4 f5	d5 d7 .	b3 a5 ♟	
6	7.76	e4 e5 .	d6 d5 .	f4 f5 .	d5-d7 .	f5-f6 .	d7-d4 ♙

Table C.2: Sample execution for board 2 with {B} heuristic

Board: BD.3 {CB} Killer Moves: Captures ON Assoc. off BestTT ON
Dynamic off Last 1 off Last 2 off
History off BestKM off

To Move: BLACK Depth: 7



Dp	Princ.Cont.	Nodes	A.B.F	%kills
0	g6 g5		23.29	14.29
1	f3 e1	1	2.09	96.30
2	a6 a5	29	10.29	64.31
3	g3 f1	128	2.20	96.31
4	e7 d8	960	7.86	72.31
5	f1 g3	3170	2.33	95.76
6	d7 g4 ♗	17977	1.00	98.12
7	score= 5	69191		

Best move: g6-g5
CPU Time: 22.66 Int. Nodes: 69191 ABF Avg. 3.23
T.T. Hits: 22254 Nodes/sec.: 3053.44 % Kills: 91.95

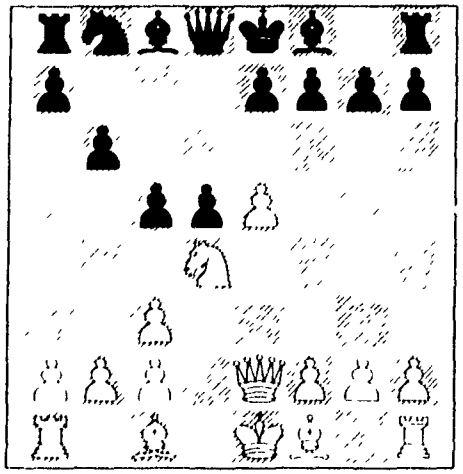
Previous Continuations:

It.	Time	0	1	2	3	4	5
1	0.00	d7-g4 ♗					
2	0.00	g6-g5 .	d2-b4 ♟				
3	0.05	g6-g5 .	f3 e1 .	d7 g4 ♗			
4	0.31	g6 g5 .	f3 e1 .	a6 a5 .	d2 b4 ♟		
5	1.08	g6 g5 .	f3 e1 .	a6 a5 .	g3 f1 .	d7 g4 ♗	
6	6.25	g6-g5 .	f3-e1 .	a6-a5 .	g3 f1 .	f6 f5 .	d2 b4 ♟

Table C.3: Sample execution for board 3 with {CB} heuristics

Board: BD 4 {CB1} Killer Moves: Captures ON Assoc. off BestTT ON
Dynamic off Last 1 ON Last 2 off
History off BestKM off

To Move: WHITE Depth: 7



Dp	Princ.Cont.	Nodes	A.B.F.	%kills
0	d4-b3		33.89	0.00
1	a7-a6	1	2.11	93.17
2	h1-g1	40	14.34	53.11
3	c5-c4	173	1.79	92.68
4	b3-d2	1715	11.44	62.61
5	d5-d4	5601	1.74	86.87
6	e2-c4 ♟	32376	1.00	95.49
7	score= 0	125977		

Best move: d4 b3
CPU Time: 42.03 Int. Nodes: 125977 ABF Avg: 2.86
T.T. Hits: 33267 Nodes/sec.: 2997.31 % Kills: 84.51

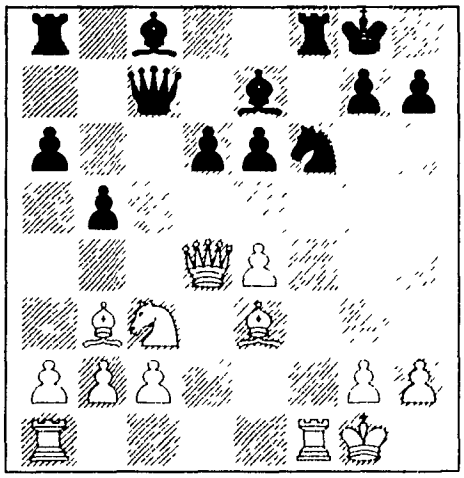
Previous Continuations:

It.	Time	0	1	2	3	4	5
1	0.00	e5 e6 .					
2	0.00	d4 b3 .	c5 c4 .				
3	0.07	e2 b5 .	b8 c6 .	d4 c6 ♞			
4	0.55	d4 b3 .	c5 c4 .	b3-d2 .	d5-d4 .		
5	1.89	d4 b3 .	c8 a6 .	c3 c4 .	d5-c4 ♙	e2-c4 ♟	
6	10.72	d4 b3 .	c8 a6 .	e2 d4 .	a6-f1 ♗	h1-f1 ♝	c5-c4 .

Table C.4: Sample execution for board 4 with {CB1} heuristics

Board: BD.5 {A} Killer Moves: Captures off Assoc. ON BestTT off
Dynamic off Last 1 off Last 2 off
History off BestKM off

To Move: WHITE Depth: 6



Dp	Princ.Cont.	Nodes	A.B.F.	%kills
0	d4-b6		12.67	16.67
1	c7 b6 ♕	2	1.80	28.11
2	e3 b6 ♕	59	16.62	37.66
3	b5 b1	201	1.36	36.11
4	c3-b1	3797	6.96	80.57
5	f6 e4 ♘	31121	1.00	99.94
6	score= 0	113292		

Best move: d4-b6
CPU Time: 60.77 Int. Nodes: 113292 ABF Avg. 1.70
T.T. Hits: 46888 Nodes/sec.: 1864.28 % Kills: 58.69

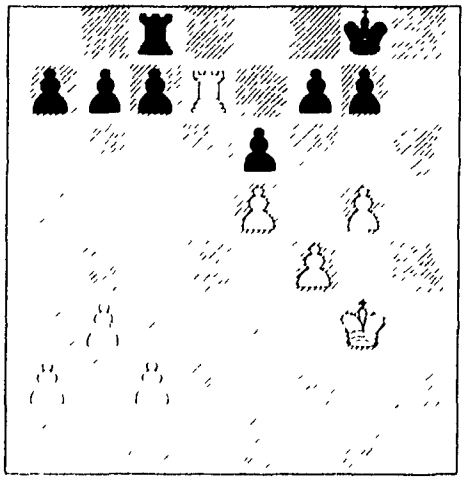
Previous Continuations:

It.	Time	0	1	2	3	4
1	0.00	d4-f6 ♞				
2	0.00	c3-b1 .	f6-e4 ♘			
3	0.09	c3-b1 .	a6 a5 .	d4 f6 ♞		
4	1.72	c3-b1 .	a6-a5 .	d4-d3 .	f6 e4 ♘	
5	16.38	d4-b6 .	c7 b6 ♕	e3 b6 ♕	b5 b4 .	f1 f6 ♞

Table C.5: Sample execution for board 5 with {A} heuristic

Board: BD.6 {C} Killer Moves: Captures ON Assoc. off BestTT off
Dynamic off Last 1 off Last 2 off
History off BestKM off

To Move: WHITE Depth: 9



Dp	Princ.Cont.	Nodes	A.B.F.	%kills
0	d7-d4		21.44	11.11
1	a7-a6	1	1.58	29.17
2	d4-d3	26	11.80	13.21
3	a6-a5	91	1.84	36.95
4	d3-d2	571	9.63	32.06
5	b7-b6	1812	1.89	52.14
6	g3-f2	8077	8.22	39.02
7	a5-a4	22457	1.57	55.41
8	b3-a4 ♟	84074	1.00	63.99
9	score= 0	221014		

Best move: d7 d4
CPU Time: 73.63 Int. Nodes: 224014 ABF Avg: 3.01
T.T. Hits: 153077 Nodes/sec.: 3042.43 % Kills: 51.07

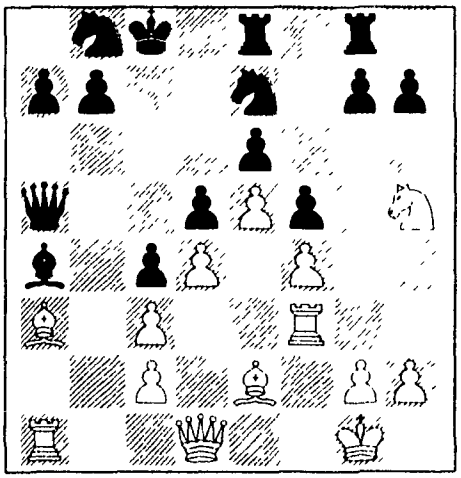
Previous Continuations:

It.	Time	0	1	2	3	4	5	6	7
1	0.00	d7 f7 ♟							
2	0.00	d7 d4 .	a7 a6 .						
3	0.03	d7 d4 .	a7 a6 .	g5 g6 .					
4	0.52	d7 d4 .	a7 a6 .	g5 g6 .	f7 g6 ♙				
5	1.18	d7 d4 .	a7 a6 .	d4 d3 .	a6 a5 .	g5 g6 .			
6	2.18	d7 d4 .	a7 a6 .	d4 d3 .	a6-a5 .	g5 g6 .	f7-g6 ♙		
7	7.13	d7 d4 .	a7 a6 .	d4 d3 .	a6-a5 .	g5-g6 .	a5-a4 .	b3-a4 ♟	
8	27.29	d7 d4 .	a7 a6 .	d4-d3 .	a6-a5 .	g5-g6 .	b7-b6 .	g5-g6 .	f7-g6 ♙

Table C.6: Sample execution for board 6 with {C} heuristic

Board: BD.7 {C1} Killer Moves: Captures ON Assoc. off BestTT off
Dynamic off Last 1 ON Last 2 off
History off BestKM off

To Move: WHITE Depth: 6



Dp	Princ.Cont.	Nodes	A.B.F.	%kills
0	a3-e7 ♞		27.40	20.00
1	e8 e7 ♗	2	3.71	80.35
2	f3 e3	37	9.55	62.67
3	a1 c2 ♘	171	2.01	95.19
4	a1-a5 ♕	1258	7.52	79.93
5	c2-d1 ♕	7283	1.00	98.77
6	score=-1	55034		

Best move: a3–e7 ♞
CPU Time: 27.67 Int. Nodes: 55034 ABF Avg: 5.65
T.T. Hits: 17003 Nodes/sec.: 1988.94 % Kills: 84.50

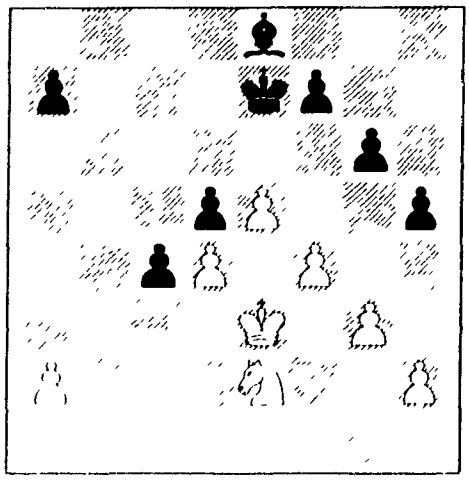
Previous Continuations:

It.	Time	0	1	2	3	4
1	0.00	a3–e7 ♞				
2	0.00	a3–e7 ♞	e8–e7 ♗			
3	0.09	a3–e7 ♞	e8–e7 ♗	a1-a4 ♝		
4	0.58	a3–e7 ♞	e8–e7 ♗	f3-f2 .	a5-c3 ♘	
5	3.53	a3–e7 ♞	e8-e7 ♗	f3–e3 .	a5-b6 .	a1 a4 ♝

Table C.7: Sample execution for board 7 with {C1} heuristics

Board: BD.8 {CB2} Killer Moves: Captures ON Assoc. off BestTT ON
Dynamic off Last 1 off Last 2 ON
History off BestKM off

To Move: WHITE Depth: 9



Dp	Princ.Cont.	Nodes	A.B.F.	%kills
0	e3-d2		13.44	0.00
1	a7-a6	1	2.19	82.50
2	a2-a3	17	6.19	52.99
3	e8-a4	79	2.20	79.43
4	d2 c1	429	5.45	56.06
5	a4-b3	1250	2.12	78.19
6	e2-g1	4867	4.88	59.44
7	b3 a2	12130	1.75	76.38
8	e5-e6	38287	1.00	68.98
9	score= 0	93642		

Best move: e3 d2
CPU Time: 22.16 Int. Nodes: 93642 ABF Avg: 2.58
T.T. Hits: 31126 Nodes/sec.: 4225.72 % Kills: 72.58

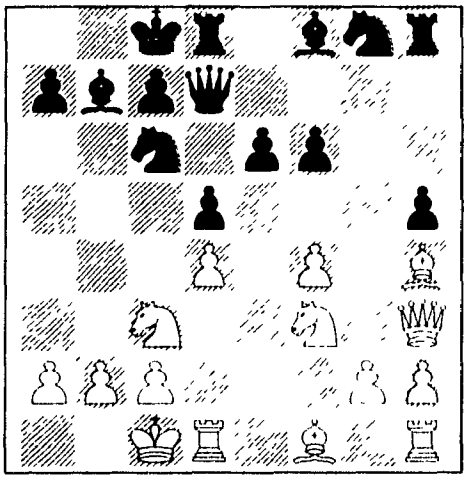
Previous Continuations:

It.	Time	0	1	2	3	4	5	6	7
1	0.00	e5 e6 .							
2	0.00	e5 e6 .	f7-e6 ♙						
3	0.02	e3-d2 .	c4 c3 .	e2 c3 ♟					
4	0.11	e3 d2 .	a7 a6 .	e5-e6 .	f7-e6 ♙				
5	0.30	e3 d2 .	a7 a6 .	a2-a3 .	c4-c3 .	e2-c3 ♟			
6	1.17	e3 d2 .	a7 a6 .	a2-a3 .	e8-a4 .	e5-e6 .	f7-e6 ♙		
7	2.91	e3 d2 .	a7 a6 .	a2-a3 .	e8-a4 .	d2-c1 .	a4-b3 .	e5-e6 .	
8	9.37	e3 d2 .	a7 a6 .	a2-a3 .	e8-a4 .	d2-c1 .	a4-b3 .	e5-e6 .	f7-e6 ♙

Table C.8: Sample execution for board 8 with {CB2} heuristics

Board: BD.9 {CH} Killer Moves: Captures ON Assoc. off BestTT off
Dynamic off Last 1 off Last 2 off
History ON BestKM off

To Move: WHITE Depth: 6



Dp	Princ.Cont.	Nodes	A.B.F.	%kills
0	f1 b5		32.71	14.29
1	d7 d6	1	2.71	57.81
2	h4 g3	40	16.21	55.38
3	a7 a6	234	2.28	92.18
4	b5 c6 ♞	2301	8.24	80.85
5	b7 c6 ♗	10725	1.00	84.23
6	score= 2	82884		

Best move: f1–b5
CPU Time: 48.99 Int. Nodes: 82884 ABF Avg: 6.39
T.T. Hits: 27544 Nodes/sec.: 1691.86 % Kills: 83.77

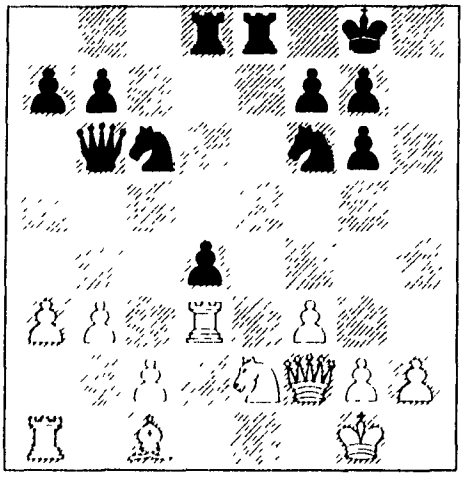
Previous Continuations:

It.	Time	0	1	2	3	4
1	0.00	h3-e6 ♟				
2	0.00	h4-e1 .	c6-d4 ♙			
3	0.13	f1-b5 .	a7-a6 .	b5-c6 ♞		
4	1.19	f1-b5 .	a7-a6 .	b5-c6 ♞	d7 c6 ♗	
5	6.31	f1-b5 .	a7-a6 .	h3-g3 .	c6-d4 ♘	d1 d4 ♞

Table C.9: Sample execution for board 9 with {CH} heuristics

Board: BD.10 {CBA} Killer Moves: Captures ON Assoc. ON BestTT ON
Dynamic off Last 1 off Last 2 off
History off BestKM off

To Move: BLACK Depth: 6



Dp	Princ.Cont.	Nodes	A.B.F.	%kills
0	b6-b5		29.80	20.00
1	c2-c4	2	2.47	92.63
2	b5-e5	43	12.09	68.67
3	e2-d4 ♟	191	2.43	94.57
4	c6-d4 ♘	1645	7.42	82.13
5	f2-d4 ♞	8877	1.00	91.70
6	score= 0	50442		

Best move: b6-b5
CPU Time: 19.43 Int. Nodes: 50442 ABF Avg: 5.92
T.T. Hits: 18148 Nodes/sec.: 2596.09 % Kills: 85.76

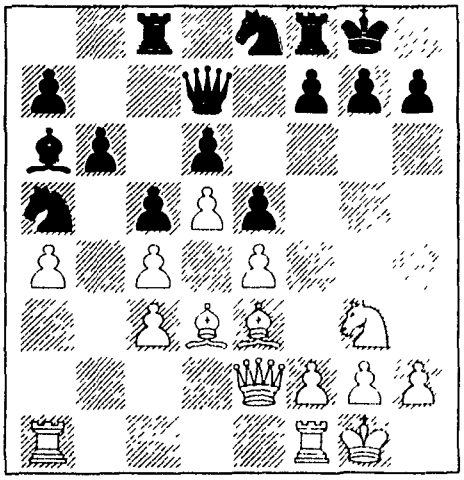
Previous Continuations:

It.	Time	0	1	2	3	4
1	0.00	e8 e2 ♘				
2	0.00	b6 c7 .	f2-d1 ♟			
3	0.07	b6-b5 .	d3-d1 .	e8-e2 ♘		
4	0.56	b6-b5 .	d3-d1 .	d4-d3 .	d1-d3 ♟	
5	3.54	b6-b5 .	c2 c4 .	b5-e5 .	e2-d4 ♟	d8-d4 ♘

Table C.10: Sample execution for board 10 with {CBA} heuristics

Board: BD.11 {CA} Killer Moves: Captures ON Assoc. ON BestTT off
Dynamic off Last 1 off Last 2 off
History off BestKM off

To Move: WHITE Depth: 7



Dp	Princ.Cont.	Nodes	A.B.F.	%kills
0	e3-g5		31.00	14.29
1	a5-b7	1	4.12	62.50
2	g5-d2	38	8.35	72.80
3	b7-a5	271	3.70	74.43
4	d3 c2	1589	5.93	76.61
5	a5- b7	12065	3.01	90.01
6	a4 a5	58003	1.00	96.00
7	score= 0	271966		

Best move: e3-g5
CPU Time: 148.12 Int. Nodes: 271866 ABF Avg: 3.66
T.T. Hits: 56948 Nodes/sec.: 1835.44 % Kills: 86.33

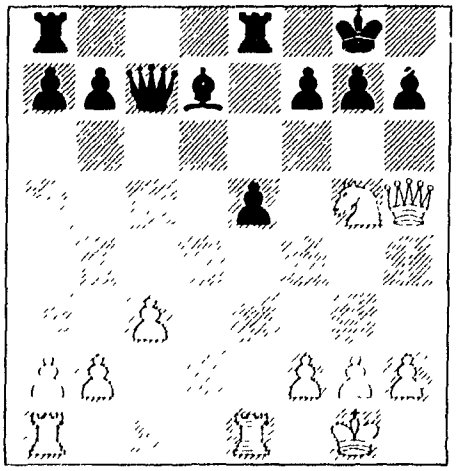
Previous Continuations:

It.	Time	0	1	2	3	4	5
1	0.00	e3-c5 ♟					
2	0.00	d3-b1 .	d7-a4 ♙				
3	0.14	d3-b1 .	a5-b7 .	e3-c5 ♟			
4	0.89	e3-d2 .	a6-c4 ♙	d3-c4 ♟	a5-c4 ♙		
5	6.56	e3-g5 .	a5-b7 .	g5 d2 .	b6 b5 .	c1 b5 ♟	
6	32.16	e3-g5 .	a5-b7 .	g5-d2 .	b7 a5 .	d3 c2 .	d7 a4 ♙

Table C.11: Sample execution for board 11 with {CA} heuristics

Board: BD.12 {CB1} Killer Moves: Captures ON Assoc. off BestTT ON
Dynamic off Last 1 ON Last 2 off
History off BestKM off

To Move: BLACK Depth: 7



Dp	Princ.Cont.	Nodes	A.B.F.	%kills
0	d7–f5		32.71	14.29
1	g1–h1	1	2.71	95.61
2	f5–g6	40	13.84	57.07
3	h5–g4	214	2.08	96.92
4	g8–h8	2948	4.18	87.99
5	g5–h3	7818	3.27	92.08
6	c7–c3 ♙	25694	1.00	96.23
7	score= 0	84525		

Best move: d7–f5
CPU Time: 32.26 Int. Nodes: 84525 ABF Avg: 3.42
T.T. Hits: 19434 Nodes/sec.: 2620.12 % Kills: 91.53

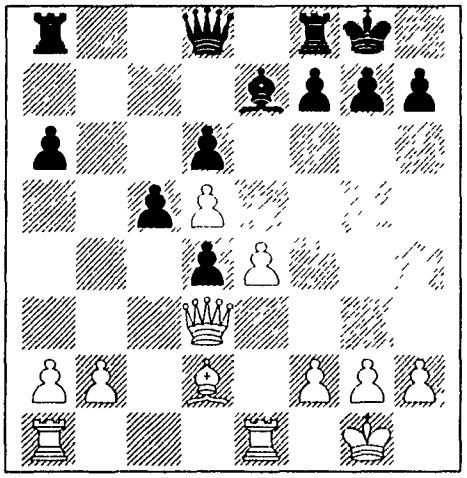
Previous Continuations:

It.	Time	0	1	2	3	4	5
1	0.00	c7 c3 ♙					
2	0.00	a7-a6 .	e1–e5 ♟				
3	0.10	a7-a6 .	h5–f3 .	c7–c3 ♙			
4	1.23	c7 c4 .	h5–h7 ♟	g8–f8 .	e1–e5 ♟		
5	3.06	d7 f5 .	g1–h1 .	g8–f8 .	h1–g1 .	c7–c3 ♙	
6	9.87	d7 f5 .	g1- h1 .	f5–g6 .	h5–g4 .	g8–h8 .	e1–e5 ♟

Table C.12: Sample execution for board 12 with {CB1} heuristics

Board: BD.13 {CB} Killer Moves: Captures ON Assoc. off BestTT ON
Dynamic off Last 1 off Last 2 off
History off BestKM off

To Move: WHITE Depth: 7



Dp	Princ.Cont.	Nodes	A.B.F.	%kills
0	d3–b1		38.71	14.29
1	c5–c4	1	1.58	75.93
2	a2–a3	47	22.98	32.46
3	d4 d3	161	1.49	76.28
4	g1-h1	2798	17.72	11.23
5	c4–c3	7361	1.34	80.33
6	b1–d3 ♟	58447	1.00	86.25
7	score= 0	155419		

Best move: d3–b1
CPU Time: 44.52 Int. Nodes: 155419 ABF Avg: 2.67
T.T. Hits: 57764 Nodes/sec.: 3490.99 % Kills: 76.90

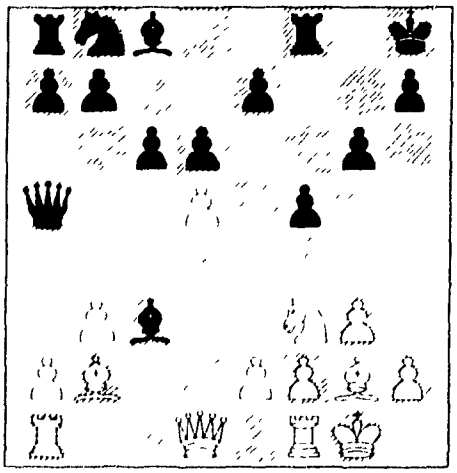
Previous Continuations:

It.	Time	0	1	2	3	4	5
1	0.00	d3–d4 ♟					
2	0.00	d3–b1 .	d4-d3 .				
3	0.05	d3–b1 .	d4–d3 .	b1–d3 ♟			
4	0.80	d3–b1 .	c5–c4 .	e4–e5 .	d6–e5 ♗		
5	2.03	d3–b1 .	c5–c4 .	a2–a3 .	c4–c3 .	d2 c3 ♟	
6	18.43	d3–b1 .	c5–c4 .	a2–a3 .	d4–d3 .	g1-h1 .	c4 c3 .

Table C.13: Sample execution for board 13 with {CB} heuristics

Board: BD.14 {C} Killer Moves: Captures ON Assoc. off BestTT off
Dynamic off Last 1 off Last 2 off
History off BestKM off

To Move: WHITE Depth: 7



Dp	Princ.Cont.	Nodes	A.B.F.	%kills
0	d1-d2		24.33	22.22
1	h8-g8	2	3.16	93.62
2	d2-c3 ♝	34	8.15	75.76
3	a5-c3 ♕	177	3.48	92.04
4	b2 c3 ♕	1085	6.91	78.65
5	c6-d5 ♘	5985	1.69	96.38
6	b3-b4	24360	1.00	99.18
7	score= 1	78641		

Best move: d1 d2
CPU Time: 37.51 Int. Nodes: 78641 ABF Avg: 2.81
T.T. Hits: 25048 Nodes/sec.: 2094.86 % Kills: 92.72

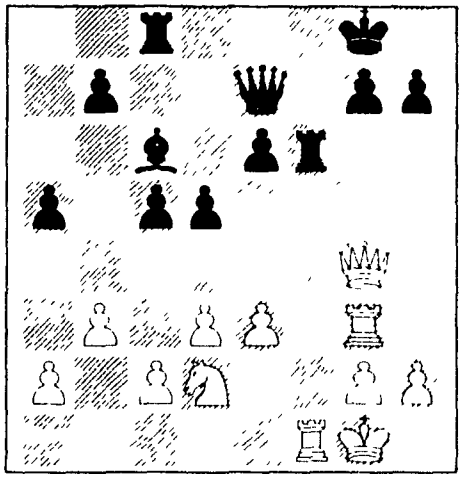
Previous Continuations:

It.	Time	0	1	2	3	4	5
1	0.00	b2 c3 ♝					
2	0.00	b2 c3 ♝	a5 c3 ♗				
3	0.08	b3-b4 .	a5 b4 ♙	b2-c3 ♝			
4	0.50	b3 b4 .	a5 b4 ♙	b2-c3 ♝	b4-c3 ♗		
5	2.91	d1 d2 .	c3-b2 ♗	d2-a5 ♕	b2-a1 ♖	f1-a1 ♝	
6	11.81	d1 d2 .	c3 b2 ♗	d2-a5 ♕	b7-b6 .	a5-e1 .	b2-a1 ♖

Table C.14: Sample execution for board 14 with {C} heuristics

Board: BD.15 {CB2} Killer Moves: Captures ON Assoc. off BestTT ON
Dynamic off Last 1 off Last 2 ON
History off BestKM off

To Move: WHITE Depth: 7



Dp	Princ.Cont.	Nodes	A.B.F.	%kills
0	g4 g7 ♟		36.18	18.18
1	e7 g7 ♕	1	2.30	91.19
2	f1 f6 ♜	19	12.89	62.68
3	g7 g3 ♖	213	1.69	93.69
4	h2 g3 ♕	2107	8.88	71.07
5	a5 a1	6793	1.21	95.18
6	b3 a4 ♟	29090	1.00	97.11
7	score= 1	71130		

Best move: g4–g7 ♟
CPU Time: 24.34 Int. Nodes: 74130 ABF Avg: 2.31
T.T. Hits: 12751 Nodes/sec.: 3045.60 % Kills: 91.85

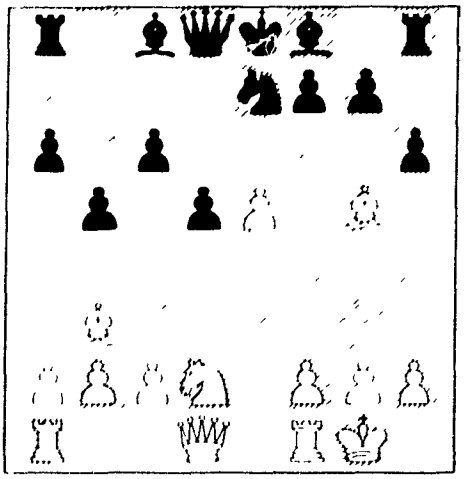
Previous Continuations:

It.	Time	0	1	2	3	4	5
1	0.00	f1–f6 ♜					
2	0.00	f1–f6 ♜	g4–f6 ♖				
3	0.09	g4–g5 .	f6–f1 ♖	g5 e7 ♕			
4	0.87	g4–h3 .	f6–f1 ♖	g1 f1 ♜	a5 a4 .		
5	2.46	g4–h3 .	f6–f1 ♖	g1–f1 ♜	a5 a4 .	h3 h7 ♟	
6	11.27	g4–g7 ♟	e7–g7 ♕	f1–f6 ♜	g7–g3 ♖	h2 g3 ♕	a5 a4 .

Table C.15: Sample execution for board 15 with {CB2} heuristics

Board: BD.16 {CBD} Killer Moves: Captures ON Assoc. off BestTT ON
Dynamic ON Last 1 off Last 2 off
History off BestKM off

To Move: WHITE Depth: 7



Dp	Princ.Cont.	Nodes	A.B.F.	%kills
0	d2 e4		27.44	22.22
1	h6-g5 ♗	2	2.42	91.98
2	e4-d6	38	11.44	65.34
3	e8 d7	171	2.91	87.07
4	d6-f7 ♟	1263	10.84	59.17
5	d8-e7	3903	1.41	91.40
6	f7-h8 ♜	33023	1.00	98.54
7	score= 2	206729		

Best move: d2 e1
CPU Time: 93.56 Int. Nodes: 206729 ABF Avg: 2.44
T.T. Hits: 28729 Nodes/sec.: 2209.59 % Kills: 87.97

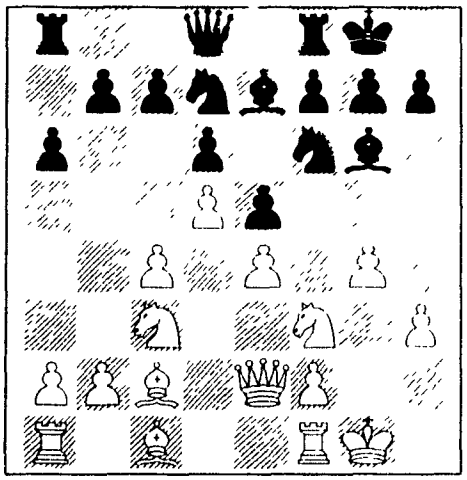
Previous Continuations:

It.	Time	0	1	2	3	4	5
1	0.00	g5-e7 ♞					
2	0.00	g5 e7 ♞	f8 e7 ♗				
3	0.08	d1 h5 .	h6-g5 ♗	h5-h8 ♜			
4	0.18	g5 h4 .	g7-g5 .	h4-g3 .	b5-b4 .		
5	1.36	g5-h4 .	g7-g5 .	h4-g3 .	b5-b4 .	b3-d5 ♟	
6	12.81	d1-h5 .	a6-a5 .	g5-e7 ♞	f8-e7 ♗	h5-e2 .	a5-a4 .

Table C.16: Sample execution for board 16 with {CBD} heuristics

Board: BD.17 {CBH} Killer Moves: Captures ON Assoc. off BestTT ON
Dynamic off Last 1 off Last 2 off
History ON BestKM off

To Move: BLACK Depth: 7



Dp	Princ.Cont.	Nodes	A.B.F.	%kills
0	a6 a5		22.13	11.29
1	f3-h1	1	2.90	91.87
2	f6-e8	28	7.80	73.07
3	h4 g6 ♝	141	3.55	92.69
4	h7 g6 ♘	1111	5.61	80.21
5	g1-h2	5103	2.90	91.22
6	a5 a4	29870	1.00	92.63
7	score= 0	1035 15		

Best move: a6–a5
CPU Time: 41.18 Int. Nodes: 103545 ABF Avg: 3.48
T.T. Hits: 34510 Nodes/sec.: 2514.45 % Kills: 89.17

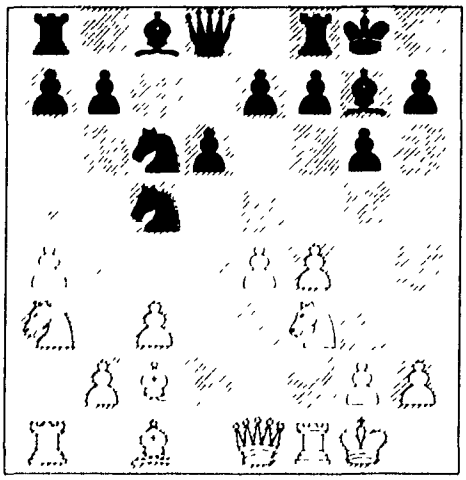
Previous Continuations:

It.	Time	0	1	2	3	4	5
1	0.00	g6–e4 ♙					
2	0.00	a6–a5 .	f3–e5 ♟				
3	0.06	a6–a5 .	g4–g5 .	g6–e4 ♙			
4	0.45	f6–e8 .	g1–h2 .	g8–h8 .	f3-e5 ♟		
5	2.06	a6–a5 .	f3–h4 .	f6–e4 ♙	h4–g6 ♝	h7-g6 ♘	
6	12.41	a6–a5 .	f3–h4 .	f6–e8 .	h4–g6 ♝	h7-g6 ♘	g1 h2 .

Table C.17: Sample execution for board 17 with {CBH} heuristics

Board: BD.18 {H} Killer Moves: Captures off Assoc. off BestTT off
Dynamic off Last 1 off Last 2 off
History ON BestKM off

To Move: BLACK Depth: 6



Dp	Princ.Cont.	Nodes	A.B.F.	%kills
0	c5-d7		36.00	16.67
1	a3-b1	1	4.04	18.14
2	c6-b8	45	12.55	53.27
3	a4-a5	283	4.51	46.62
4	b7-b6	3483	6.29	69.05
5	a5-b6 ♟	23545	1.00	91.71
6	score= 0	168537		

Best move: c5-d7
CPU Time: 102.50 Int. Nodes: 168537 ABF Avg: 6.04
T.T. Hits: 58792 Nodes/sec.: 1644.26 % Kills: 63.84

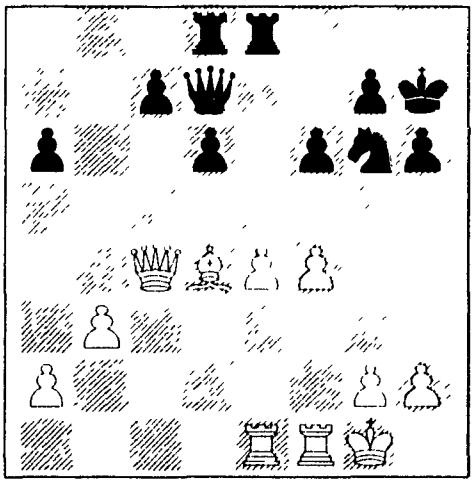
Previous Continuations:

It.	Time	0	1	2	3	4
1	0.00	c5 a4 ♙				
2	0.00	c5-d7 .	a4-a5 .			
3	0.16	c5-d7 .	a4-a5 .	c6-a5 ♙		
4	1.89	c5-d7 .	a3-b1 .	c6-b8 .	a4-a5 .	
5	15.73	c5 d7 .	a3-b1 .	c6-b8 .	a4-a5 .	g7-c3 ♙

Table C.18: Sample execution for board 18 with {H} heuristics

Board: BD.19 {B} Killer Moves: Captures off Assoc. off BestTT ON
Dynamic off Last 1 off Last 2 off
History off BestKM off

To Move: BLACK Depth: 6



Dp	Princ.Cont.	Nodes	A.B.F.	%kills
0	a6 a5		28.50	16.67
1	c1 d3	1	2.39	91.12
2	d6 d5	36	9.31	57.72
3	d1 b2	209	3.11	80.86
4	a5 a1	1290	4.58	76.59
5	c4 d5 ♟	6974	1.00	99.28
6	score= 0	30685		

Best move: a6-a5
CPU Time: 11.89 Int. Nodes: 30685 ABF Avg: 4.32
T.T. Hits: 10172 Nodes/sec.: 2580.74 % Kills: 77.34

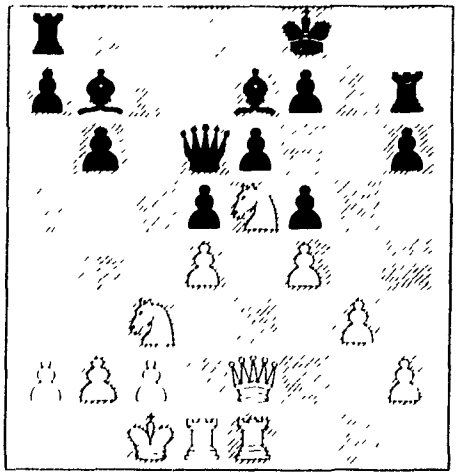
Previous Continuations:

It.	Time	0	1	2	3	4
1	0.00	g6-f4 ♙				
2	0.00	d6-d5 .	c4-d5 ♟			
3	0.08	a6-a5 .	c4-d3 .	g6-f4 ♙		
4	0.52	a6-a5 .	c4-d3 .	d6-d5 .	d4 f6 ♟	
5	2.58	a6-a5 .	c4-d3 .	d6-d5 .	d4-b2 .	d5-c4 ♙

Table C.19: Sample execution for board 19 with {B} heuristics

Board: BD.20 {CBA} Killer Moves: Captures ON Assoc. ON BestTT ON
Dynamic off Last 1 off Last 2 off
History off BestKM off

To Move: WHITE Depth: 6



Dp	Princ.Cont.	Nodes	A.B.F.	%kills
0	c3-b1		33.50	16.67
1	f8-g8	1	2.01	96.00
2	e5-f3	42	13.88	59.94
3	d6-b4	184	1.84	96.27
4	c2-c3	1762	8.78	76.07
5	b4-d4 ♙	6762	1.00	97.53
6	score= 0	39391		

Best move: c3-b1
CPU Time: 15.59 Int. Nodes: 39391 ABF Avg: 6.11
T.T. Hits: 19935 Nodes/sec.: 2526.68 % Kills: 83.81

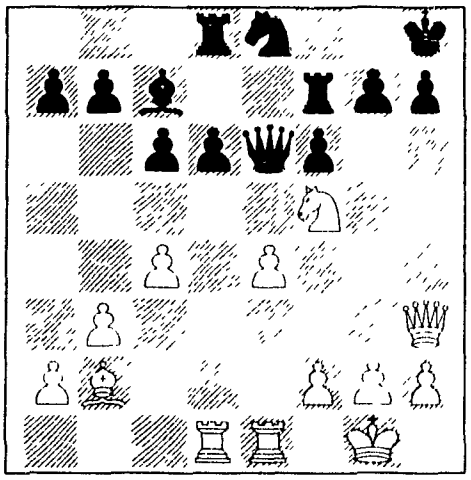
Previous Continuations:

It.	Time	0	1	2	3	4
1	0.00	c3-d5 ♟				
2	0.00	e5-d3 .	d6-f4 ♙			
3	0.06	e5-d3 .	b6-b5 .	e2-e6 ♟		
4	0.64	c3-b1 .	f8-g8 .	e5-f3 .	d6-f4 ♙	
5	2.56	c3-b1 .	f8-g8 .	e5-f3 .	d6-b4 .	e2-e6 ♟

Table C.20: Sample execution for board 20 with {CBA} heuristics

Board: BD.21 {CBD} Killer Moves: Captures ON Assoc. off BestTT ON
Dynamic ON Last 1 off Last 2 off
History off BestKM off

To Move: WHITE Depth: 6



Dp	Princ.Cont.	Nodes	A.B.F.	%kills
0	f5-h4		39.33	16.67
1	e6 h3 ♕	1	2.32	96.17
2	g2-h3 ♕	49	13.64	69.74
3	c6-c5	263	1.72	96.61
4	h4 f3	2082	9.77	76.34
5	d6 d5	9432	1.00	98.17
6	score= 0	57741		

Best move: f5-h4
CPU Time: 26.32 Int. Nodes: 57741 ABF Avg: 6.57
T.T. Hits: 26020 Nodes/sec.: 2193.81 % Kills: 84.52

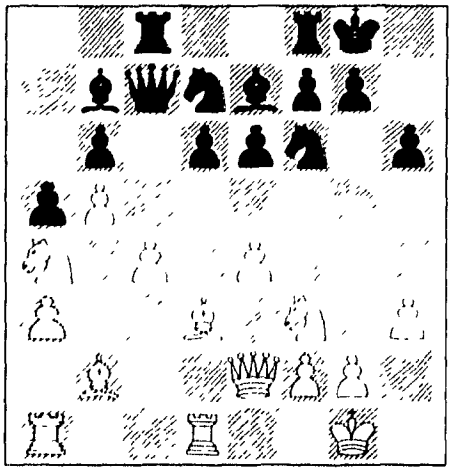
Previous Continuations:

It.	Time	0	1	2	3	4
1	0.00	d1-d6 ♟				
2	0.00	b2-f6 ♟	e8-f6 ♗			
3	0.17	f5-g7 ♟	e6-h3 ♕	g2-h3 ♕		
4	0.98	f5-h4 .	e6-h3 ♕	g2-h3 ♕	c6-c5 .	
5	4.87	f5-h4 .	e6-h3 ♕	g2-h3 ♕	c6 c5 .	d1 d6 ♟

Table C.21: Sample execution for board 21 with {CBD} heuristics

Board: BD.22 {II} Killer Moves: Captures off Assoc. off BestTT off
Dynamic off Last 1 off Last 2 off
History ON BestKM off

To Move: BLACK Depth: 6



Dp	Princ.Cont.	Nodes	A.B.F.	%kills
0	e6-e5		27.67	16.67
1	a4-c3	1	2.58	80.00
2	f6-e8	35	12.62	37.91
3	d3-c2	163	4.12	57.66
4	d6-d5	1448	6.78	68.95
5	c4-d5 ♟	8923	1.00	97.80
6	score= 0	77996		

Best move: e6-e5
CPU Time: 43.85 Int. Nodes: 77996 ABF Avg: 6.29
T.T. Hits: 27435 Nodes/sec.: 1778.70 % Kills: 65.86

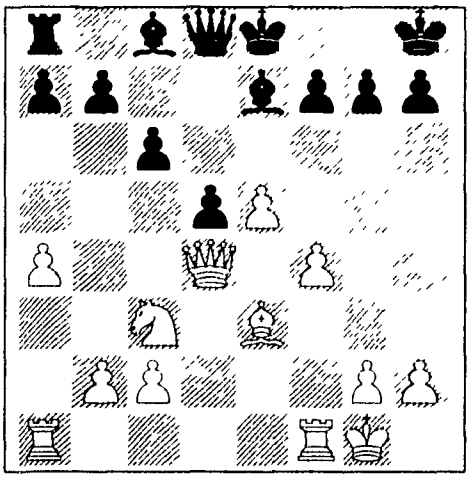
Previous Continuations:

It.	Time	0	1	2	3	4
1	0.00	f6-e4 ♙				
2	0.00	e6-e5 .	a4-b6 ♟			
3	0.08	e6-e5 .	a4-c3 .	f6-e4 ♙		
4	0.76	e6-e5 .	a4-c3 .	d6-d5 .	c4-d5 ♟	
5	4.73	e6 e5 .	a4-c3 .	f6-e4 ♙	d3-e4 ♞	b7-e4 ♗

Table C.22: Sample execution for board 22 with {II} heuristics

Board: BD.23 {CD} Killer Moves: Captures ON Assoc. off Best/TT off
Dynamic ON Last 1 off Last 2 off
History off BestKM off

To Move: BLACK Depth: 6



Dp	Princ.Cont.	Nodes	A.B.F	%kills
0	d8-a5		29.33	0.00
1	d4 d3	1	1.37	71.43
2	a5-c7	37	9.10	64.38
3	c3-d1	222	3.91	85.92
4	c6-c5	2193	4.57	81.56
5	e3 c5 ♟	17651	1.00	99.48
6	score= 0	94533		

Best move: d8-a5
CPU Time: 58.00 Int. Nodes: 94533 ABF Avg: 4.55
T.T. Hits: 16506 Nodes/sec.: 1629.88 % Kills: 81.99

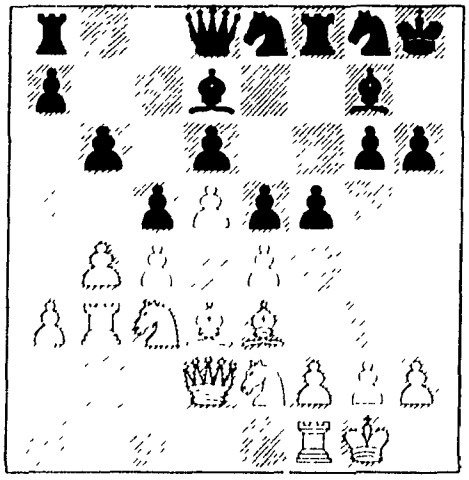
Previous Continuations:

It.	Time	0	1	2	3	4
1	0.00	c6-c5 .				
2	0.00	c6-c5 .	c3-d5 ♟			
3	0.17	c6-c5 .	d4-d3 .	c5-c4 .		
4	1.36	c6-c5 .	d4-d3 .	c5-c4 .	e3-a7 ♟	
5	12.94	d8-a5 .	d4-d2 .	a5-b4 .	e3-a7 ♟	a8-a7 ♙

Table C.23: Sample execution for board 23 with {CD} heuristics

Board: BD.24 {CBH} Killer Moves: Captures ON Assoc. off BestTT ON
Dynamic off Last 1 off Last 2 off
History ON BestKM off

To Move: WHITE Depth: 7



Dp	Princ.Cont.	Nodes	A.B.F.	%kills
0	e4-f5 ♟		34.43	14.29
1	c5-b4 ♙	1	1.67	97.50
2	a3-b4 ♟	42	18.51	54.70
3	g6-f5 ♙	160	1.72	97.61
4	e3 h6 ♟	1660	14.91	60.57
5	g7-h6 ♗	5721	2.02	95.76
6	d2-h6 ♝	43170	1.00	95.31
7	score= 0	180867		

Best move: e4 f5 ♟
CPU Time: 78.83 Int. Nodes: 180867 ABF Avg: 3.20
T.T. Hits: 58019 Nodes/sec.: 2294.39 % Kills: 92.62

Previous Continuations:

It.	Time	0	1	2	3	4	5
1	0.00	e3-c5 ♟					
2	0.00	b4-c5 ♟	d6-c5 ♙				
3	0.06	b4-c5 ♟	d6-c5 ♙	e3-c5 ♟			
4	0.70	b4-c5 ♟	d6-c5 ♙	e4-f5 ♟	f8-f5 ♙		
5	2.39	b4-c5 ♟	d6-c5 ♙	e4-f5 ♟	g6-f5 ♙	e3-c5 ♟	
6	19.50	b4-c5 ♟	d6-c5 ♙	e4-f5 ♟	g6-f5 ♙	e3-c5 ♟	f5-f4 .

Table C.24. Sample execution for board 24 with {CBH} heuristics

Appendix D

Source Code of FIXAFAN Program

```
/*************************************************************************
  chess.h:
          Includes all global constant and variable declarations.
*************************************************************************/

#define TRUE 1
#define FALSE 0
#define DEAD -1
#define UNDEAD 0
#define EXACT 0
#define BOUND 2097152
#define TRYMOVE 666
#define HASHSIZE  524287    /* size of the hash table */

#define MAXSCORE 50         /* used to set wide search window */
#define MAXMAX  500         /* used to set infinite window */
#define MAXMVS 120          /* size of move list */
#define MAXD 20             /* size of stack */

#define X 0                 /* empty square */
#define K 1                 /* king */
#define Q 2                 /* queen */
#define R 3                 /* rook */
#define N 4                 /* knight */
#define B 5                 /* bishop */
#define P 6                 /* pawn */
#define E 7                 /* en-passant pawn capture */

#define WH 8                /* white bit */
#define BL 16               /* black bit */
#define BOTH 24             /* both' (= no color) */
#define QUEEN 32            /* new queen */
#define CAST  128           /* flag used for castleing */

typedef int T_piece,        /* contents of a square */
            T_cost,         /* cost funtion */
            index;          /* coordinates inside the board */

typedef struct { int m,k; } T_PS;    /* used for hash code tuple */
```

```
typedef struct { index r,c; } T_sq;  /* coordinates of a square */

/* board */
typedef struct {
          T_piece  bd[12][12];  /* board */
          T_cost   h;           /* cummulative score of board */
          int tomove,idle,      /* player to move and opponent (BL,WH) */
              castlestat,       /* castleing status register */
              castlefl[2],      /* indicates if castleing is allowed */
              EP;               /* EP shows if en-passant is allowed */
          T_PS  hashcode;       /* hash code for the position */
        } T_board;

/* move */
typedef struct {
          T_sq from,to;         /* to and from squares */
          T_piece prey;         /* captured piece */
        } T_move;
T_move inmove;                  /* used to read in a move */

/* node */
typedef struct {
          int mv,               /* index of move currently examined */
              bestmove,         /* index of best move found */
              score,            /* score of the node */
              num,              /* number of moves in mvl */
              killmvs,          /* num of killer moves at top of list */
              oldEP;            /* EP code for that position */
          T_PS oldhash;         /* hash code for that position */
          T_move mvl[MAXMVS];   /* list of moves */
        } T_node;

/* principle continuation array */
struct T_pc { T_move move;  struct T_pc *next; }
    *Gpcarray[MAXD],            /* array of pointers to rows of
                                   the pc array used to copy the
                                   best continuation so far */
    *pc,                        /* temporary variable */
    *oldpc;                     /* holds the pc from last iteration */

/* transposition table */
typedef struct {
        int   entry,            /* information stored: score,
                                   depth, bestmove, exact/bd flag. */
              hcode;            /* 32-bit hashing code */
        } T_hentry;

T_hentry  htable[HASHSIZE];

T_PS  PS_table[15][8][8];       /* piece-square table of random
                                   nos. for updating hash codes */
/* killer moves */
typedef struct {
          T_move   km;          /* killer move */
          int    freq;          /* # of cutoffs caused by move */
        } T_killer;
```

```
T_move kmchild[10][10][10][10][2]; /*  killer mv. assoc. w/ parent */
int    kmtable[10][10][10][10][2]; /*  hash table of killer moves */
T_move kmlist[MAXD][2];            /*  last 2 killer mvs. at a ply */
T_killer  highest[MAXD];           /*  most effective killer moves */

/* statistics table */
typedef struct {
          int   count,          /* current counter of branches */
                nodes,          /* # nodes at a level */
                kills,          /* # effective killer moves */
                total;          /* total branches for the level */
        } T_stats;

T_stats  stats[MAXD][MAXD];     /* stats table of all fanouts for
                                   each level in each iteration */
float   fanouts[MAXD],          /* avg. fanout for each level */
        killpower[MAXD],        /* effectiveness of killer moves */
        ABF, KILLS;             /* overall fanout & %kills */

T_board Gboard,board;           /* Game board & search board */

int val[32],              /* values of the pieces */
    it_dep,               /* depth of ID search */
    pcscore,              /* last principal continuation score */
    window,               /* initial extent of window on both
                             sides of 'pcscore' */
    stopdepth,            /* absolute maximum depth of search */
    stopcheck,            /* flag to check for castling */
    nodecount,            /* number of nodes expanded so far */
    hits,                 /* number of Transposition table hits */
    TTtotal,              /* TT hits after completed iteration */

    /* flags for the eight move ordering heuristics */
    assoc,
    bestTT,
    captures,
    dynamic,
    history,
    bestkm,
    last1,
    last2;

T_node   S1,S2,lstack[MAXD];    /* private stack with two extra nodes
                                   at the top to initilize scores
                                   (S1,S2,lstak) MUST STAY TOGETHER */

int   live,               /* true if -l parameter is given */
      game,               /* true if -g parameter is given */
      step,               /* true if -s parameter is given */
      tree;               /* true if -t parameter is given */

double  t0,t1,timelimit,dt,ct; /* time counters */
double  daytime(),cputime();   /* time functions */

void chess_game(), update(), restore(), updatepc(), cpmov();
void readboard(), readmove(), outmove(), outboard(), outnode();

void clr(),go(),pause(),setterm();   /* for screen output */
```

```
/*************************************************************************
 *************************************************************************
  chess.c :
                                FIXAFAN
                          a chess playing program

by          : Eric The    8413720
supervisor : M. Newborn

  The following program plays chess starting from any position. The
board is read from the keyboard if no "file" parameter is given. Options
can also be supplied:
     -m# : where '#' > 0, is the maxdepth at which the program will
           stop its ID search if timelimit is not reached first.
           Default = 10.
     -t# : where '#' > 0, indicates the timelimit in real seconds
           after which the search stops if stopdepth is not reached
           first. Default = 120.
     -l# : where '#' > 0, creates an 'active' board for iteration
           number # showing all moves being updated and restored
           as long as the RETURN key is pressed.
     -p# : prints out the search tree for iteration number #.
     -s# : steps through iteration number # by stopping after each
           branch from the root has been searched.
     -g  : plays a complete game interactively with the user
  Any argument not starting with a "-" will be taken to be the name of
the input file to use for reading the board (see 'readboard()' in
IO_lib.c)

  The source files for the program are:
    chess.c       : main program, reads in the arguments
    Schess.c      : does the game search
    AB_lib.c      : does the A-B search on a subtree
    Ttable.c      : transposition table management routines
    IO_lib.c      : all I/O operations
    chessmv.c     : move generation and board updating

  They have to be compiled with the -c option and then the object files
have to be linked together with the additional libraries -ltermcap,
and -lm.

*************************************************************************
*************************************************************************/

#include "chess.h"

#include <stdio.h>
#include <math.h>

int i;

main(argc,argv)
int argc; char *argv[];
{ FILE *infile;
  char *inname;

  infile = stdin;
```

```
    live = DEAD,   tree = FALSE;   step = FALSE;   game = FALSE;

    /*-----------------------------------*/
    /* initialize all eight heuristics OFF */
    /*-----------------------------------*/
    assoc=FALSE; bestTT=FALSE; captures=FALSE; dynamic=FALSE;
    history=FALSE; bestkm=FALSE; last1=FALSE; last2=FALSE;

    stopdepth = 10;              /* default maximum number of iterations */
    timelimit = 600.0;           /* default maximum CPU time limit */
    window = 2;                  /* default size of search window */

    for (i=1;i<argc;i++)
      if (*argv[i] == '-')
        switch (*(argv[i]+1)) {

          /*-----------------------------------*/
          /* initialize requested execution options */
          /*-----------------------------------*/
          case 'l': live = atoi(argv[i]+2); break;
          case 'm': stopdepth = atoi(argv[i]+2); break;
          case 't': timelimit = (double) atof (argv[i]+2); break;
          case 'w': window = atoi(argv[i]+2); break;
          case 's': step = atoi(argv[i]+2); break;
          case 'p': tree = atoi(argv[i]+2); break;
          case 'g': game = TRUE; break;

          /*-----------------------------------*/
          /* turn ON requested move ordering heuristics */
          /*-----------------------------------*/
          case 'a': assoc = TRUE; break;
          case 'b': bestTT = TRUE; break;
          case 'c': captures = TRUE; break;
          case 'd': dynamic = TRUE; break;
          case 'h': history = TRUE; break;
          case 'k': bestkm = TRUE; break;
          case '1': last1 = TRUE; break;
          case '2': last2 = TRUE; break;
        }
      else
        infile = fopen(argv[i],"r+");

    setterm();

    if (infile == NULL) {
      printf ("CANNOT OPEN FILE\n");
      return;
    }

    readboard(infile,&Gboard);        /* read in initial game board */
    chess_game();

    go(23,0);  if (game)  printf(" GAME OVER !!\n");
    exit(0);
}
```

```
/*************************************************************************
 *************************************************************************
  Schess.c :
            Implements a chess playing program using the
            iteratively deepening alpha-beta algorithm.

*************************************************************************
*************************************************************************/

#include  "chess.h"
#include  <stdio.h>
#include  <math.h>
#include  <sys/types.h>
#include  <sys/time.h>
#include  <sys/resource.h>
#define   maxply 0
#define   minply 1

int row,ro,co;        /* used for screen output */

/*************************************************************************/
/*  chess_game:                                                          */
/*        Finds the best move it can by calling 'IDAB_search()',         */
/*      outputs the move and the new position as well as a few           */
/*      statistics. It then inputs the opponent's move and               */
/*      repeats until the end of the game.                               */
/*************************************************************************/

void  chess_game()
{ int i,end,windo;

  windo=window;                  /* size of alpha-beta search window */
  nodecount=0;               /* counter for number of expanded nodes */
  hits=0;                   /* counter for number of hash table hits */

  /*-----------------------------------------------------------------*/
  /* if computer's color is black, let opponent play first if this */
  /* is an actual game (i.e. "game"= TRUE)                          */
  /*-----------------------------------------------------------------*/
  if ((game) && (Gboard.tomove==BL))
  { clr();
    outboard(&Gboard,1,32);
    go (9,31);  printf(" WHITE TO START.");  go (20,5);
    Gboard.tomove=WH;  Gboard.idle=BL;
    readmove(stdin,&inmove);
    update(inmove,&Gboard,0);
   }

  while (abs(Gboard.h)<50)
  {
    /*---------------------------------------------------------------*/
    /* erase contents of Killer move tables, Hash table, and stats */
    /*---------------------------------------------------------------*/
    clearkmt();   inithash();   initstats();

    clr();
    outboard(&Gboard,2,2);
```

```
go (10,1); if (Gboard.tomove==WH) printf(" WHITE TO MOVE..");
                        else     printf(" BLACK TO MOVE..");
go (1,1); printf("*TT,KM:");
if (assoc) printf(" A");
if (bestTT) printf(" BT"),
if (captures) printf(" C");
if (dynamic) printf(" D");
if (history) printf(" H");
if (bestkm) printf(" BK");
if (last1) printf(" L1"),
if (last2) printf(" L2");   printf(".");
fflush(stdout);
board=Gboard,       /* make copy of Game Board for A-B search */

/*-----------------------------------------*/
/* call Iterative-deepening Alpha-Beta search */
/*-----------------------------------------*/
end = IDAB_search(stopdepth,timelimit,windo);

/*---------------------------*/
/* Output move and statistics */
/*---------------------------*/
dt=(daytime() - t0),        /* obtain difference in real time */
pc=Gpcarray[0];              /* obtain principle continuation */

/*-----------------------------------------------------------*/
/* check for game-ending check-mate situation, if OK update */
/* board with first move of principle continuation          */
/*-----------------------------------------------------------*/
if ((Gpcarray[0]->next==NULL)||((Gpcarray[0]->next->move.prey&7)!=K))
  update(pc->move,&Gboard,0);

/*-----------------------------------------------------------*/
/* disable castling flags for either player if move caused  */
/* a change in the castling conditions                      */
/*-----------------------------------------------------------*/
if ((Gboard.castlefl[0]==0)
    && (((Gboard.castlestat & 8)==8) || ((Gboard.bd[2][2]&7)!=R))
    && (((Gboard.castlestat & 4)==4) || ((Gboard.bd[2][9]&7)!=R)))
  Gboard.castlefl[0]=1;
if ((Gboard.castlefl[1]==0)
    && (((Gboard.castlestat & 128)==128) || ((Gboard.bd[9][2]&7)!=R))
    && (((Gboard.castlestat & 64)==64) || ((Gboard.bd[9][9]&7)!=R)))
  Gboard.castlefl[1]=1;

if (tree==FALSE)             /* if NOT in the tree printout mode */
{ outboard(&Gboard,2,2);

    outstats();                          /* calculate statistics */

  go (1,60);  printf(" A.B.F.   Killer %%");
  row = 2;
  while ((row-2)<end)
  { go (row++,26);
    if (pc!=NULL) { outmove(pc->move);  pc=pc->next; }
    else  printf("         ");
    if (fanouts[row-3])
    { go (row-1,60);
```

```
    printf("%6.2f    %5.2f%%",fanouts[row-3],killpower[row-3]);
   }
 }
go (row,24); printf("  score=%2d  ",pcscore);

if (step)                /* if we are in the trace search mode */
{ go (11,10);
  printf(" stat=%d  ",board.castlestat);
  outboard(&board,2,2); getc(stdin);
 }

/*--------------------------*/
/* print remaining statistics */
/*--------------------------*/
go (row+1,43); printf("Day time: %4.2f          ",dt);
go (row+2,43); printf("CPU time: %4.2f          ",ct);
go (row+3,45); printf("A.B.F.: %4.2f ",ABF);
go (row+1,62); printf("%7d gennodes",nodecount);
go (row+2,62); printf("%5.2f nodes/sec.",(double)nodecount/ct);
go (row+3,62); printf("  %5.2f %% kills  ",KILLS);
go (row+4,62); printf("%7d T.T.hits",TTtotal);
go (11,0);     printf("PC move:    "); outmove(Gpcarray[0]->move);
go (21,5);

if (Gpcarray[0]->next!=NULL)
{
  /*-------------------------------------------------*/
  /* check for game-ending check-mate by the opponent */
  /*-------------------------------------------------*/
  if ((Gpcarray[0]->next->move.prey&7)==K)
  { printf(" YOU WIN!!! Thank you for playing");
    go (11,8); printf("  Resignation  ");
    return; }

  /*-------------------------------------------------*/
  /* check for game-ending check-mate by the computer */
  /*-------------------------------------------------*/
  if ((Gpcarray[0]->next->next!=NULL) &&
      ((Gpcarray[0]->next->next->move.prey & 7)==K))
  { printf(" CHECK MATE!!! Thank you for playing");
    return; }
 }

if (game==FALSE)  return;            /* if NOT real game mode */

/* otherwise, read opponent's next move */
go (10,1);
if (Gboard.tomove==WH) printf(" WHITE TO MOVE..");
               else    printf(" BLACK TO MOVE..");

readmove(stdin,&inmove);
update(inmove,&Gboard,0);

/*---------------------------------------*/
/* check for change in castling conditions */
/*---------------------------------------*/
if ((Gboard.castlefl[0]==0) && ((Gboard.castlestat&12)==12))
  Gboard.castlefl[0]=1;
```

```
      if ((Gboard.castlefl[1]==0) && ((Gboard.castlestat&192)==192))
        Gboard.castlefl[1]=1;

      /*-----------------------------------------------------*/
      /* reset window size, node counter, and hash table counter */
      /*-----------------------------------------------------*/
      windo=window; nodec unt=0; hits=0;
     }
    if (tree) exit(0);              /* if we are in print tree mode */
   }
 }   /* chess_game() */

/*************************************************************************/

int left,right;        /* low and high bound of search window */

/*************************************************************************/
/*  IDAB_search.                                                         */
/*        Performs an iterative deepening alpha-beta search using a      */
/*      window set at the beginning of each iteration and if it fails,   */
/*      the iteration is done again with a "half-infinite" window.       */
/*************************************************************************/

IDAB_search(maxd,maxt,windoe)
int maxd;              /* maximum depth */
double maxt;           /* maximum wall clock time */
int windoe;            /* extent of window on both sides of pcscore */
{ int  ply,OK,TILT;
  char fail;

  /*-------------------------------------------------------------*/
  /* establish pcscore and left/right boundaries of search window */
  /*-------------------------------------------------------------*/
  pcscore=Gboard.h;
  left=pcscore-windoe;
  right=pcscore+windoe;

  TILT=FALSE;
  it_dep=1;  ro=13;
  go (1,20); printf(" D"); go (1,37); printf("Nodes    time    redo");
  go (2,20); printf(" 0");
  fflush (stdout);

  /*-----------------------------------------------------*/
  /* obtain start times for CPU time and real time counters */
  /*-----------------------------------------------------*/
  t0=daytime();
  t1=cputime();
  ct=0 0;

  while ((it_dep<=maxd) && (ct<maxt))
  {
    /*-------------------------------------------------------*/
    /* if we are in live mode, set flag to make board "come alive" */
    /*-------------------------------------------------------*/
    if (live==it_dep) { live=UNDEAD; }
    go (2+it_dep,20);  printf("%2d",it_dep);
```

```
/*-------------------------------------------------------------*/
/* if this is not the first iteration then reorder the sequence */
/* of moves to follow the continuation of previous iteration    */
/*-------------------------------------------------------------*/
if (it_dep>1)
{ nodecount++;
  ply=reorder(lstack,it_dep,Gpcarray[0],left,right);
 }
/*-------------------------------------------------------------*/
/* otherwise generate the root node and start first iteration */
/*-------------------------------------------------------------*/
else { lstack[2].oldEP = board.EP;
       OK=genmoves(&lstack[2],0,FALSE,0,0);
       lstack[2].score=left;
       if (OK==FALSE) { printf(" no moves to play!\n"), return(0); }
       if (tree==1) printf("\n\n\n\n\n\n\n");
       ply=0;
      }

/*--------------------------*/
/* perform alpha-beta search */
/*--------------------------*/
alpha_beta(ply,it_dep,left,right,lstack,Gpcarray,maxt);

if (ct>=maxt)  break;                /* check for time expiration */
if (OK)  go (2+it_dep,45);  else  go (2+it_dep,53);
  printf("%5.2f",ct);
if (live == UNDEAD)  live=DEAD;        /* turn off live mode flag */

if (ct<maxt)
{ OK=TRUE;

  /*-----------------------------------------------*/
  /* check if score returned is a "fail-high" score */
  /*-----------------------------------------------*/
  if  (lstack[2].score>=right)
  { left=pcscore + windoe - 1;  OK=FALSE;  fail='H';
    if (TILT==FALSE) { right=MAXSCORE; TILT=TRUE; }
    else { right=MAXMAX; }
   }
  else {
         /*-----------------------------------------------*/
         /* check if score returned is a "fail-low" score */
         /*-----------------------------------------------*/
         if (lstack[2].score<=left)
         { right=pcscore-windoe + 1;  OK=FALSE;  fail='L';
           if (TILT==FALSE) { left= -MAXSCORE; TILT=TRUE; }
           else { left= -MAXMAX; }
          }
        }

  if (OK==FALSE)
  {
    /*-----------------------------------------------------*/
    /* if score returned was not a true score, do not output */
    /* all of the iteration statistics                      */
    /*-----------------------------------------------------*/
    pc = Gpcarray[0];
```

```
    if (it_dep!=tree)           /* if NOT in the print tree mode */
    { row = 2;
      while (pc!=NULL)
      { if (row >= (it_dep+1))  break;
        go (row++,26); outmove(pc->move);
        pc=pc->next;
       }
      go (row,24);  printf("  score=%2d  ",lstack[2].score);
      go (it_dep+2,36);  printf("%6d",nodecount);
      go (2+it_dep,50);  printf("; %c",fail);
      fflush(stdout);

      if (step)            /* if we are in the search trace mode */
      { go (11,10);
        printf(" stat=%d  ",board.castlestat);
        outboard(&board,2,2); getc(stdin); }
     }
  }
else
{
  /*-------------------------------------------------*/
  /* output complete iteration statistics and set up the */
  /* next iteration if we have not reached "maxd"        */
  /*-------------------------------------------------*/
  if (it_dep<maxd)
  { ro = 14 + ((it_dep-1)%9);
    it_dep++;                       /* increment iteration depth */
    TILT=FALSE;   TTtotal=hits;

    /*---------------------------------*/
    /* set new pcscore and window bounds */
    /*---------------------------------*/
    pcscore=lstack[2].score;
    left=pcscore-windoe; right=pcscore+windoe;

    if ((it_dep-1)!=tree)          /* if NOT in print tree mode */
    {
      /*-------------------*/
      /* output statistics */
      /*-------------------*/
      pc = Gpcarray[0];
      go (1,24); row = 2;  co= -8;
      printf("Princ.Cont.");
      go (13,0);  printf("Prev.continuations:");
      while (pc!=NULL)
      { /* output the principle continuation found so far */
        if (row >= (it_dep+1))  break;
        go (row++,26); outmove(pc->move);
        if (it_dep<=10)
        { co=co+8;  go (ro,co);  outmove(pc->move);
          fflush(stdout); }
        pc=pc->next;
       }
      go (row,24),  printf("  score=%2d  ",lstack[2].score);
      go (it_dep+1,36);  printf("%6d",nodecount);
      if (it_dep<=10)
      { go (ro,co+7);  printf("=%2d  ",lstack[2].score); }
      fflush(stdout);
```

```
            if (step)              /* if we are in search trace mode */
            { outboard(&board,2,2);  go (11,10);
              printf(" stat=%d  ",board.castlestat);
              getc(stdin); }
            }

           /*-------------------------------------------------------*/
           /* end the search if the continuation found leads to a mate */
           /*-------------------------------------------------------*/
           if (abs(pcscore)>MAXSCORE)  return(it_dep);
          }
         else {
               /*-----------------------------------------------*/
               /* we have reached "maxd" so set up exit from IDAB */
               /*-----------------------------------------------*/
               it_dep++;
               pcscore=lstack[2].score;   TTtotal=hits;
               go (it_dep+1,36);  printf("%6d",nodecount); }
        }
      }
     if (ct>=maxt)  break;               /* check for time expiration */
    }  /* outer while */

   if ((ct>=maxt) && (lstack[2].score!=left))
   { pcscore=lstack[2].score;
     return(it_dep);
    }
   return(--it_dep);
  }  /* IDAB_search() */

/*************************************************************************/
/*  initstats:                                                           */
/*        Initializes stats table to zero.                               */
/*************************************************************************/

initstats()
{ int i,j;
  for (i=0; i<MAXD; i++)
    for (j=0; j<MAXD; j++)
    {stats[i][j].count=0; stats[i][j].nodes=0,
     stats[i][j].kills=0, stats[i][j].total=0,}
 }

/*************************************************************************/
/*  outstats:                                                            */
/*        Calculates results of stats table (eg. fanouts).               */
/*************************************************************************/

outstats()
{ int  it,d,tot,kkk,nods,Ttot,Tkkk,Tnods;

  Ttot=0;  Tnods=0;  Tkkk=0;
  tot=0;   nods=0;   kkk=0;
  for (it=1; it<=it_dep; it++)
  { tot += stats[it][0].total;
    kkk += stats[it][0] kills;
    nods += stats[it][0].nodes;
```

```
    }
   fanouts[0]=(float)tot/nods;
   Ttot+=tot;  Tnods+=nods;
   killpower[0]=(float)kkk*100/nods;   Tkkk+=kkk;

   for (d=1; d<it_dep; d++)
   { tot=0;  nods=0;  kkk=0,
     for (it=d+1; it<=it_dep; it++)
     { tot += stats[it][d].total;
       kkk += stats[it][d].kills;
       nods += stats[it][d] nodes;
      }
     fanouts[d]=(float)tot/nods,
     killpower[d]=(float)kkk*100/nods;
     if (d < (it_dep-1))
     { Ttot+=tot;  Tnods+=nods;  Tkkk+=kkk; }
    }
  ABF=(float)Ttot/Tnods;
  KILLS=(float)Tkkk*100/Tnods;
 }

/*************************************************************************/
/*  daytime:                                                             */
/*           Returns the time of day in seconds.                         */
/*************************************************************************/

double daytime()
{ struct timeval bb;
  gettimeofday(&bb,NULL),
  return((double)(bb.tv_sec) + (bb.tv_usec/1000000.0));
 }

/*************************************************************************/
/*  cputime·                                                             */
/*           Returns the CPU time in seconds.                            */
/*************************************************************************/

double cputime()
{ struct rusage bb;
  getrusage(NULL,&bb);
  return((double)(bb ru_utime.tv_sec+(bb.ru_utime.tv_usec/1000000.0)));
 }
```

```
/*************************************************************************
 *************************************************************************
  AB_lib.c :
           Includes the procedures used for performing an alpha-beta
           search.
           alpha_beta: performs alpha_beta search.
           updatepc  : updates the principal continuation array
           reorder   : takes the stack and board along the principal
                       continuation.

*************************************************************************
*************************************************************************/

#include "chess.h"
#include <math.h>
#include <stdio.h>
#define maxnode 0
#define minnode 1
#define m_bd  board

/*************************************************************************/
/*  insertkm:                                                            */
/*        Increments the entry in 'kmtable' which corresponds to         */
/*      the move 'kmv'.  This table keeps count of how many              */
/*      times each legal move caused a refutation.  When an              */
/*      entry exeeds the current 'highest' killer for the                */
/*      current depth 'dp', then 'highest[dp]' is set to 'kmv'           */
/*************************************************************************/

insertkm(kmv,dp)
T_move kmv;  int dp;
{ if (++kmtable[kmv.from.r][kmv.from.c][kmv.to.r][kmv.to.c][dp&1]
       > highest[dp].freq)
  { highest[dp].km=kmv;  highest[dp].freq++; }
 }

/*************************************************************************/
/*  equal:                                                               */
/*        Routine used to search a list for a particular move            */
/*        Checks if 'listmv' is the same move as 'mv'                    */
/*************************************************************************/

equal(listmv,mv)
T_move  listmv,mv;
{ if ((listmv.from.r==mv.from.r) && (listmv.from.c==mv.from.c)
      && (listmv.to.r==mv.to.r) && (listmv.to.c==mv.to.c))
    return(1);
  else  return(0);
 }

/*************************************************************************/
/*  clearkmt:                                                            */
/*        Clears the contents of killer move table by setting            */
/*      all the frequencies to 0.                                        */
/*************************************************************************/
```

```
clearkmt()
{ int  i,j,k,l,m;
  T_move  empty;                    /* 'empty' is a default null move */

  empty.from.r=0;  empty.from.c=0;  empty.to.r=0;  empty.to.c=0;

  for (i=0; i<10; i++)
   for (j=0, j<10; j++)
    for (k=0; k<10; k++)
     for (l=0; l<10, l++)
      for (m=0, m<2; m++)
      { kmtable[i][j][k][l][m]=0,  kmchild[i][j][k][l][m]=empty; }

  for (i=0, i<MAXD, i++)
  { highest[i].freq=0,   highest[i].km=empty;
    kmlist[i][0]=empty,   kmlist[i][1]=empty, }
 }

/*************************************************************************/
/*  alpha_beta.                                                          */
/*        This procedure will perform an alpha-beta search using         */
/*      'stak' and 'Gboard'  'dpth' indicates the current depth          */
/*      of the stack, 'maxd' indicates the maximum depth of              */
/*      search, 'left' and 'right' are the low and hi bounds of          */
/*      the search window (alpha and beta)  'pcarr' is used to           */
/*      store the principal continuation and 'mt' is the time at         */
/*      which the function stops "in its tracks" and returns.            */
/*      Check-mate is not checked, the high value of the king            */
/*      should garantee a correct behavior of alpha-beta                 */
/*************************************************************************/

alpha_beta(depth,maxd,left,right,stak,pcarr,mt)
int depth,maxd,left,right,
T_node stak[],
struct T_pc *pcarr[];
double mt;
{ register int  leafsc;       /* used for returning score of the best leaf
                                 when at 'maxd-1' without updating the
                                 board or advancing the stack. */
  register T_node *pnode;     /* points to 'stak[depth]' */
  register T_move *pmv;       /* points to a move in pnode */
  int     sc,height,flag,redo;
  T_move  bestmv,tempmv;

  pnode = stak + depth + 2;   /* an offset of 2 is always added to
                                 the depth in order to make room
                                 for "score(-1)" and "score(-2)". */
  stats[maxd][0].nodes++;
  redo=FALSE,
  stak[1].score = right,                    /* represents "score(-1)" */
  if ((right== MAXSCORE)||(right== MAXMAX)) redo=TRUE;
  stak[0].score = left;                     /* represents "score(-2)" */
  if ((left== -MAXSCORE)||(left== -MAXMAX)) redo=TRUE;

  for (;;)
  { while ((depth<maxd) && (abs(m_bd.h)<50))     /* depth-first search */
    { if (pnode->num==0)
      {
```

```
  /*-----------------------------------------*/
  /* set up a new tree node and generate moves */
  /*-----------------------------------------*/
  pnode->mv=0;  pnode->bestmove=0;
  pnode->score= (pnode-2)->score;
  pnode->oldEP= m_bd.EP;
  if (genmoves(pnode,depth,FALSE,0,0) == 0)
  { pnode->score = 0; break; }              /* position is a tie */
 }

if (depth==maxd-1)                    /* use of leafsc at 'maxd-1' */
{   register int i,maxi,lsc,numb;

    /*-----------------------------------------------------*/
    /* if 'captures' is ON then look only at capturing moves */
    /*-----------------------------------------------------*/
    if (captures)  numb = pnode->killmvs;
             else  numb = pnode->num;
    i = 0;  maxi=1;
    leafsc = val[pnode->mvl[i++].prey&31];
    switch (depth&1)
    { case maxnode:
        while (i<numb)
          if ((lsc=val[pnode->mvl[i++].prey&31]) < leafsc)
            { leafsc = lsc,  maxi = i; }
          else  if ((captures) && (lsc!=0))  break;

        /*-----------------------------------------------------*/
        /* reduce value of horizon capture. increment stats. */
        /*-----------------------------------------------------*/
        if (leafsc!=0)
        { leafsc++;  stats[maxd][depth].kills++; }
        break;
      case minnode:
        while (i<numb)
          if ((lsc=val[pnode->mvl[i++].prey&31]) > leafsc)
            { leafsc = lsc;  maxi = i; }
          else  if ((captures) && (lsc!=0))  break;

        /*-----------------------------------------------------*/
        /* reduce value of horizon capture. increment stats. */
        /*-----------------------------------------------------*/
        if (leafsc!=0)
        { leafsc--;  stats[maxd][depth].kills++; }
     }
    leafsc = m_bd.h - leafsc;
    if (depth!=0) pnode->num = maxi;
    pnode->mv = maxi-1;
    depth++; pnode++;          /* increment depth to reach 'maxd' */
 }

else                                          /* interior node */
{
  pnode->oldhash=m_bd.hashcode;             /* save old hash code */
  update(pnode->mvl[pnode->mv],&m_bd,maxd);

  if (assoc)
  {
```

```
  /*-----------------------------------------------------*/
  /* obtain best 'kmchild' of move 'pmv' which was used to */
  /* update the board, store that move in 'highest[]'      */
  /*-----------------------------------------------------*/
  pmv=&(pnode->mvl[pnode->mv]);
  highest[depth+1].km = kmchild[pmv->from.r][pmv->from.c]
                              [pmv->to.r][pmv->to.c][depth&1];
 }

if (maxd==tree) outmove(pnode->mvl[pnode->mv]);
depth++; pnode++;                              /* increment depth */
pnode->num=0;

/*-------------------------------------------------*/
/* retrieve info on new position in the hash table */
/*-------------------------------------------------*/
Hretrieve(m_bd.hashcode,&height,&sc,&flag,&bestmv,pnode-1);

if ((height>(maxd-depth))||(height==(maxd-depth)))
{ if (flag==EXACT)  { pnode->score=sc;      /* hash table hit' */
                      pcarr[depth]=NULL;
                      if (maxd==tree)
                        printf("[%d,EXACT]",sc);
                      hits++;  stats[maxd][depth].nodes--;
                      break, }
  else
  { pnode->score=sc;
    if ((depth&1)==maxnode)
    {
      if (sc>=(pnode-1)->score)             /* hash table hit! */
        { hits++;  stats[maxd][depth].nodes--;
          if (maxd==tree)
            printf("[%d,BOUND]",sc); break; }
      else pnode->score=(pnode-2)->score;
     }
    else        /* minnode */
    {
      if (sc<=(pnode-1)->score)             /* hash table hit! */
        { hits++;  stats[maxd][depth].nodes--;
          if (maxd==tree)
            printf("[%d,BOUND]",sc); break; }
      else pnode->score=(pnode-2)->score;
     }
  }
 } else  pnode->score=(pnode-2)->score;

if ((bestTT) && (height>=0))   /* try the best move from table */
{ pnode->oldEP= m_bd.EP;

  /*-----------------------------------------------------*/
  /* generate moves for one piece only! (last 2 parameters) */
  /*-----------------------------------------------------*/
  genmoves(pnode, depth, FALSE, bestmv.from.r, bestmv.from.c);

  pnode->mv=findmove(pnode->mvl,bestmv);
  if (pnode->mv > MAXMVS)            /* 'bestmv' was NOT found */
  { go(12,0); outboard(&board,2,2);
    printf(" %d\n",pnode->mv);
```

```
        outmove(bestmv);
        outmove((pnode-1)->mvl[(pnode-1)->mv]);
        outnode(pnode);
        printf(" ** TT error");  exit(0);
       }

      pnode->bestmove=pnode->mv;
      pnode->killmvs=pnode->mv+1;
      if (depth!=maxd-1)
        pnode->num=TRYMOVE;    /* flag for partial move generation */
     }
   } /* outer else */
 } /* while */                          /* gone as deep as possible */

if (depth==maxd)
{ pnode->score=leafsc;
  if (maxd==tree) printf("=>%2d    ",leafsc);
 }
else
if (abs(m_bd.h)>50)                     /* if move leads to a mate */
  { stats[maxd][depth].nodes--;
    switch((depth-1)&1)         /* score will favor shallowest mate */
    { case maxnode: pnode->score=m_bd.h + (depth-1); break;
      case minnode: pnode->score=m_bd.h - (depth-1); }
   }

/*-------------------------*/
/* backtrack up to the root */
/*-------------------------*/
for (;;)
{
  if ((depth!=maxd) && (height<(maxd-depth)) && (abs(m_bd.h)<50))
  { flag=BOUND;        /* set the validity of score for hash table */
    if ((depth&1)==maxnode)
    { if ((pnode->score < (pnode-1)->score)
          && (pnode->score!=(pnode-2)->score))
          flag=EXACT;
     }
    else     /* if minnode */
         { if ((pnode->score > (pnode-1)->score)
               && (pnode->score!=(pnode-2)->score))
           flag=EXACT;
          }

    /*-----------------------------------------------------------*/
    /* store info on completed node in hash table BEFORE restoring */
    /*-----------------------------------------------------------*/
    Hstore(m_bd.hashcode,maxd,depth,pnode->score,
          flag,pnode->mvl[pnode->bestmove],pnode-1);
  }
  depth--;  pnode--;  height= -1;              /* decrement depth */

  /*------------------*/
  /* minimax algorithm */
  /*------------------*/
  switch (depth&1)
  { case maxnode:if ((pnode+1)->score > pnode->score)
                 { if ((pnode->score=(pnode+1)->score) < right)
```

```
                updatepc(pnode->mvl[pnode->mv],depth,pcarr,maxd);
                pnode->bestmove=pnode->mv;
               }
              break;
  case minnode:if ((pnode+1)->score < pnode->score)
              { if ((pnode->score=(pnode+1)->score) > left)
                 updatepc(pnode->mvl[pnode->mv],depth,pcarr,maxd);
                pnode->bestmove=pnode->mv;
               }
 }

/*-----------------------------*/
/* restore to previous position */
/*-----------------------------*/
if (depth!=(maxd-1))
{ restore(pnode->mvl[pnode->mv],&m_bd,maxd);
  stats[maxd][depth+1].nodes++;
  m_bd.hashcode=pnode->oldhash;  m_bd.EP=pnode->oldEP;
 }
pnode->mv++;  stats[maxd][depth].total++;

if ((depth>0) && (pnode->mv < pnode->num))
{
  /*----------------------*/
  /* alpha-beta algorithm */
  /*----------------------*/
  switch (depth & 1)
  { case maxnode: if (pnode->score >= (pnode-1)->score)
                  {
                    /*-----------------------------*/
                    /* update killer move counters */
                    /*-----------------------------*/
                    prepare_kill(pnode,depth,maxd);
                    if (pnode->mv <= pnode->killmvs)
                      stats[maxd][depth].kills++;
                    pnode->bestmove=pnode->mv-1;
                    if (maxd==tree) printf(" AB");

                    /*--------------------------*/
                    /* store info in hash table */
                    /*--------------------------*/
                    Hstore(m_bd.hashcode,maxd,depth,pnode->score,
                           BOUND,pnode->mvl[pnode->mv-1],pnode-1);

                    /*-----------------------------*/
                    /* restore to previous position */
                    /*-----------------------------*/
                    depth--; pnode--;
                    restore(pnode->mvl[pnode->mv],&m_bd,maxd);
                    stats[maxd][depth+1].nodes++;
                    stats[maxd][depth].total++;
                    pnode->mv++;  m_bd.EP=pnode->oldEP;
                    m_bd.hashcode=pnode->oldhash;
                   }
                  break;

    case minnode: if (pnode->score <= (pnode-1)->score)
                  {
```

```
                        /*-------------------------------*/
                        /* update killer move counters */
                        /*-------------------------------*/
                        prepare_kill(pnode,depth,maxd);
                        if (pnode->mv <= pnode->killmvs)
                          stats[maxd][depth].kills++;
                        pnode->bestmove=pnode->mv-1;
                        if (maxd==tree) printf(" AB");

                        /*---------------------------*/
                        /* store info in hash table */
                        /*---------------------------*/
                        Hstore(m_bd.hashcode,maxd,depth,pnode->score,
                               BOUND,pnode->mvl[pnode->mv-1],pnode-1);

                        /*-----------------------------*/
                        /* restore to previous position */
                        /*-----------------------------*/
                        depth--; pnode--;
                        restore(pnode->mvl[pnode->mv],&m_bd,maxd);
                        stats[maxd][depth+1].nodes++;
                        stats[maxd][depth].total++;
                        pnode->mv++;  m_bd.EP=pnode->oldEP;
                        m_bd.hashcode=pnode->oldhash;
                       }
                      /*-------------------------------*/
                      /* check time elapsed here only */
                      /*-----------------------------*/
                      ct=cputime() - ti;
    }
  }

/*-------------------------------------------*/
/* if all root moves have been searched, return */
/*-------------------------------------------*/
if ((depth==0) && (pnode->mv >= pnode->num))
  return;
else  { if ((step==maxd) && (depth==0))
        {
          /*-----------------------------------------*/
          /* if trace search mode, print out root move */
          /*-----------------------------------------*/
          outboard(&m_bd,2,2);  go (11,9);
          printf(" score=%d  ",stak[2].score);
          fflush(stdout);
          getc(stdin);  go (11,0);
          outmove(pnode->mvl[pnode->mv]);
          getc(stdin);
         }
        if ((ct>=mt) || (maxd==1))       /* check for time limit */
          return;
        if (pnode->mv < pnode->num)
        { if (pnode->num==TRYMOVE)
          {
            /*-------------------------------------------*/
            /* if partial move generation was NOT sufficient */
            /* then generate all moves and try again         */
            /*-------------------------------------------*/
```

```
                tempmv=pnode->mvl[(pnode->mv)-1];
                m_bd.EP=pnode->oldEP;
                nodecount--;    /* don't count move generation twice */
                genmoves(pnode,depth,FALSE,0,0);

                /*-----------------------------------------------*/
                /* find previously searched move and place it last */
                /*-----------------------------------------------*/
                pnode->mv=findmove(pnode->mvl,tempmv);
                tempmv.prev=pnode->mvl[pnode->mv].prev;
                pnode->mvl[pnode->mv]=pnode->mvl[pnode->num-1];
                pnode->mvl[pnode->num-1]=tempmv;
                pnode->bestmove=0;  pnode->mv=0;
                pnode->num--;             /* one less move to search */
               }
              if (dynamic)        /* DYNAMIC move ordering done here */
                dynamic_order(pnode,maxd,depth);
              break;
             }
           }
     } /* inner for-loop (backtrack) */

    if (maxd==tree) { printf("\n"); leafsc=1;       /* print tree mode */
                      while (leafsc<=depth)
                      { printf("       "); leafsc++; }
                     }
   } /* outer for-loop */
 } /* alpha_beta() */

/*************************************************************************/
/*  updatepc:                                                            */
/*       updates the principal continuation array 'pca' with move 'mov' */
/*     at depth 'd' assuming 'pca' goes down to depth 'md'.              */
/*************************************************************************/

void updatepc(mov,d,pca,md)
T_move  mov;  int  d;  struct T_pc *pca[];  int md;
{ struct T_pc *t,*q;  int i;

  if (md==tree) { if (d!=(md-1))                    /* print tree mode */
                  { printf("\n");  i=0;
                    while (i++ < md)
                      printf("       "); }
                  printf("*%d:%d>",d,lstack[d+2].score);  outmove(mov);
                 }
  if (pca[d]==NULL)
  { pca[d]=(struct T_pc *) malloc (sizeof(struct T_pc));
    pca[d]->next = NULL, }
  pca[d]->move = mov;

  if ((d<(md-1)) && ((mov.prev & 7)!=K))
  { t=pca[d+1];   q=pca[d];
    while (t!=NULL)
    { if (q->next == NULL)
      { q->next=(struct T_pc *) malloc (sizeof(struct T_pc));
        q->next->next=NULL; }
      q = q->next;  q->move = t->move;
      t = t->next;
```

```
    }
   q->next = NULL;
  }
 else pca[d]->next = NULL;
}


/*************************************************************************/
/*  reorder:                                                             */
/*        Takes the stack 'stk' and 'lboard' along the principal         */
/*      continuation in 'pc'. 'l' and 'r' are used to initialize the     */
/*      scores of the nodes. 'stk' is supposed to be empty (depth=0)     */
/*************************************************************************/

reorder(stk,dep,pc,l,r)
T_node  stk[];  int  dep;  struct T_pc *pc;  int l,r;
{ struct T_pc *pcm;
  int  d,j;

  if (dep==tree)  printf("\n A-B for d=%d\n",dep);
  pcm=pc;   stk[2].score= l;

  for (d=0; d<(dep-1);)
  { if (pcm==NULL)  break;
    j=0;
    for(;j<stk[d+2].num;)        /* find princ. cont. move in the list */
    { if ( (stk[d+2].mvl[j].to.r==pcm->move.to.r)
        && (stk[d+2].mvl[j].to.c==pcm->move.to.c)
        && (stk[d+2].mvl[j].from.r==pcm->move.from.r)
        && (stk[d+2].mvl[j].from.c==pcm->move.from.c))  break;
      j++;
     }
    if (j>0 && j<stk[d+2].num)
    { stk[d+2].mvl[j]=stk[d+2].mvl[0];
      stk[d+2].mvl[0]=pcm->move; }
    stk[d+2].mv=0;
    stk[d+2].oldhash=m_bd.hashcode;

    update(stk[d+2].mvl[0],&m_bd,dep);

    if (assoc)                /* obtain best 'kmchild' of updated move */
      highest[d+1].km = kmchild[pcm->move.from.r][pcm->move.from.c]
                               [pcm->move.to.r][pcm->move.to.c][d&1];

    if (dep==tree)  outmove(stk[d+2].mvl[0]);
    d++;
    stk[d+2].num=0;  stk[d+2].bestmove=0;
    stk[d+2].oldEP=m_bd.EP;

    genmoves(&stk[d+2],d,FALSE,1,0);

    if (d==1) stk[d+2].score= r;
        else  stk[d+2].score= stk[d].score;
    pcm=pcm->next;
   }
  stk[d+2].mv=0;
  return(d);
 }  /* reorder() */
```

```
/*************************************************************************/
/*  prepare_kill:                                                        */
/*        Will take the current move in 'pnode' and identifies it        */
/*      as a killer move depending on which move ordering                */
/*      strategies are currently in effect.                              */
/*************************************************************************/

prepare_kill(pnode,depth,mxd)
register T_node  *pnode;  int depth,mxd;
{ register T_move  *pmv,*kmv;
  T_killer  tmv;

  kmv=&(pnode->mvl[(pnode->mv)-1]);        /* 'kmv' is the killer move */

  if ((last2) || (dynamic))
  {
    /*-----------------------------------------------------------*/
    /* move last move to position 2 and place 'kmv' in first pos. */
    /*-----------------------------------------------------------*/
    if (equal(kmlist[depth][0],*kmv)==0)
    { kmlist[depth][1]=kmlist[depth][0];
      kmlist[depth][0]=*kmv; }
   }

  if (assoc)
  {
    /*-----------------------------------------------------------*/
    /* obtain parent 'pmv' of 'kmv' and enter 'kmv' in 'kmchild'  */
    /*-----------------------------------------------------------*/
    pmv=&((pnode-1)->mvl[(pnode-1)->mv]);
    kmchild[pmv->from.r][pmv->from.c]
           [pmv->to.r][pmv->to.c][(depth-1)&1]=*kmv;
    return; }

  if (bestkm)
  {
    /*--------------------------------------*/
    /* update table of killer move counters */
    /*--------------------------------------*/
    insertkm(*kmv,depth);
    return; }

  if (history)
  {
    /*------------------------------------------------------*/
    /* update table using Scheaffer's depth-dependent formula */
    /*------------------------------------------------------*/
    kmtable[kmv->from.r][kmv->from.c]
           [kmv->to.r][kmv->to.c][depth&1] += 1 << (mxd-depth);
    return; }

  if ((last1) && (!last2))  highest[depth].km = *kmv;
}
```

```
/*************************************************************************/
/*  dynamic_order:                                                       */
/*        Dynamically reoderes moves at an interior node using           */
/*      info gathered 2 levels deeper using the last_2 strategy          */
/*************************************************************************/

dynamic_order(pnode,maxd,depth)
register T_node  *pnode;  int  maxd,depth;
{ register int  diff,i,keep;
  T_move   tmv,highmv,nextmv,swapmv;

  keep=pnode->mv;
  if ((diff=(maxd-depth)) > 2)
  { highmv=kmlist[depth+2][0];
    nextmv=kmlist[depth+2][1];
    for (i=pnode->mv; i<pnode->num; i++)
    { if (equal(pnode->mvl[i],highmv) && (i>pnode->mv))
      { tmv=pnode->mvl[pnode->mv];
        pnode->mvl[pnode->mv]=pnode->mvl[i];
        pnode->mvl[i]=tmv;
        if (pnode->mv >= pnode->killmvs)
          pnode->killmvs = pnode->mv + 1;
        return;
       }
      if ((keep==pnode->mv) && equal(pnode->mvl[i],nextmv))
        keep=i;
     }

    if (keep > pnode->mv)
    { tmv=pnode->mvl[pnode->mv];
      pnode->mvl[pnode->mv]=pnode->mvl[keep];
      pnode->mvl[keep]=tmv;
      if (pnode->mv >= pnode->killmvs)
        pnode->killmvs = pnode->mv + 1;
     }
   }
 }
```

```
/*************************************************************************
 *************************************************************************
  Ttable.c :
      Contains all procedures related to management of the transposition
    table of all chess positions encountered during an iterative search.

*************************************************************************
*************************************************************************/

#include "chess.h"
#include <stdio.h>
#include <math.h>

/*********************************************************************/
/*                                                                   */
/* initPStab(): initializes all entries of the 15 X 8 X 8            */
/*              piece-square table of random numbers.                */
/*                                                                   */
/* inithash():  sets all entries of the hash table as empty by       */
/*              initializing the 32-bit "entry" field to 0.          */
/*                                                                   */
/*********************************************************************/

initPStab()
{ int  i,j,k;

  for (i=0; i<15; i++)
    for (j=0; j<8; j++)
      for (k=0; k<8; k++) {
        PS_table[i][j][k].m=random();
        PS_table[i][j][k].k=random();
      }
 }


inithash()
{ int i;  register T_hentry  *hi;

  hi = htable;
  for (i=0; i<HASHSIZE; i++)
    (hi++)->entry=0;
 }

/*********************************************************************/
/*   Hinsert():                                                      */
/*         encodes info such as; PCd (lenght of princ. cont.), sco   */
/*       (score), fl (EXACT or BOUND), and mv (bestmove), into       */
/*       location "key" of the hash table.                           */
/*********************************************************************/

Hinsert(key,PCd,sco,fl,mv)
int  key,PCd,sco,fl;  T_move mv;
{ int  mvcode;
  mvcode=((mv.from.r)<<12) + ((mv.from.c)<<8) + ((mv.to.r)<<4) + mv.to.c;
  htable[key].entry=PCd + (mvcode<<5) + (sco<<22) + fl;
}
```

```
/*************************************************************************/
/*   Hretrieve():                                                        */
/*         Procedure to retrieve an entry from the hash table. If        */
/*       the requested entry is not found in the table a height          */
/*       of (-1) is returned. "height" is the length of the              */
/*       effective subtree searched from this chess position.            */
/*************************************************************************/

Hretrieve(Hcode,height,score,flag,mov)
T_PS Hcode;  int  *height,*score,*flag; T_move *mov;
{ int key,next,mcode;
  register T_hentry  *hockey;
  register int  hobit;

  /*---------------------------------------*/
  /* obtain address of hash entry to retrieve */
  /*---------------------------------------*/
  key=Hcode.k & HASHSIZE;   hockey=htable + key;

  /*-------------------------------------------------------*/
  /* first see if hashcode is identical to entry's hashcode */
  /*-------------------------------------------------------*/
  if (hockey->hcode!=Hcode.m)
  {
    /*-------------------------------------------------------*/
    /* check next 5 locations in hash table to find same entry */
    /*-------------------------------------------------------*/
    next=0;
    while (next++ < 6)
    { if ((++key)==HASHSIZE)    /* wrap-around back to entry 0 */
         hockey=htable;  else  hockey++;
      if (hockey->hcode==Hcode.m) break; }
   }

  /*-------------------------------------------------------*/
  /* if an identical entry was found, retrieve its information */
  /*-------------------------------------------------------*/
  if (hockey->hcode==Hcode.m)
  {
    hobit=hockey->entry;
    *height=hobit & 31;
    *score=(hobit) >> 22;
    *flag =(hobit & BOUND);
    mcode =(hobit & ((1<<21)-1)) >> 5;

    /*-----------------------------------*/
    /* extract the bestmove from "mcode" */
    /*-----------------------------------*/
    mov->from.r =( mcode >>12);
    mov->from.c =((mcode & ((1<<12)-1)) >> 8);
    mov->to.r   =((mcode & ((1<<8)-1)) >> 4);
    mov->to.c   =( mcode & 15);
   }
  /*-------------------------------------------------------*/
  /* otherwise return height= -1 to indicate entry was not found */
  /*-------------------------------------------------------*/
  else   *height = -1;
}
```

```
/*************************************************************************/
/*  Hstore():                                                            */
/*        Used to store an entry into the hash table , if the entry      */
/*      already exists in the table it is only inserted if the           */
/*      height of the previously searched subtree is less or equal       */
/*      to the height of the presently completed subtree from the        */
/*      same position.                                                   */
/*************************************************************************/

Hstore(hc,MXD,dep,score,flag,mov)
T_PS  hc;  int  MXD,dep,score,flag;  T_move  mov;
{ int  key,next,h;
  register T_hentry  *hockey;

  /*-----------------------------------------------------*/
  /* obtain address of hash position to insert new entry */
  /*-----------------------------------------------------*/
  key=hc.k & HASHSIZE;   hockey= htable + key;

  /*-------------------------------------------------------------*/
  /* if height (h) is non-zero, compare height of old entry with */
  /* height of new entry to see which is more useful             */
  /*-------------------------------------------------------------*/
  next=0;
  while (h=(hockey->entry & 31))
  { if ((h<=(MXD-dep)) && (hockey->hcode==hc.m))
      break;
    if (hockey->hcode==hc.m)
      return;    /* entry was found but old entry is more valuable */

    next++;      /* look at next location in the table */
    if (next==6)
      return;    /* no room available in table for a new entry, program
                    searched for empty slot in 5 adjacent locations */

    if ((++key)==HASHSIZE)          /* wrap-around back to entry 0 */
    { hockey=htable; key=0; }
    else  hockey++;
   }

  /*-----------------------------------------------*/
  /* insert new hashcode and new entry information */
  /*-----------------------------------------------*/
  if (hockey->hcode!=hc.m)
    hockey->hcode=hc.m,
  Hinsert(key,MXD-dep,score,flag,mov);
 }

/*************************************************************************/
/*  findmove() :                                                         */
/*        used to return location of a specific move in a movelist       */
/*************************************************************************/

findmove(list,Bmv)
T_move  list[],Bmv;
{ int  i;
```

```
i=0;
for (;;)
{ while ((list[i].from.r!=Bmv.from.r)||(list[i].from.c!=Bmv.from.c))
    i++;
  if ((list[i].to.r==Bmv.to.r) && (list[i].to.c==Bmv.to.c)) return(i);
  i++;
 }
}
```

```
/*************************************************************************
 *************************************************************************
  IO_Lib.c :
         Includes all input/output functions. Must be linked with
         "termcap" which is used by the first 3 functions.

         setterm: gets the control strings for clearing the screen and
                  going to a specific position on the screen.

         go     : puts the curser at a specific location on the screen.
         clr    : clears the screen.
         pause  : waits for the user to hit a <return>.

         outpc    : outputs a piece.
         outsq    : outputs a square in european notation (ex: a1).
         outmove  : outputs a move.
         outnode  : outputs an entire node (debugging...).
         outboard : outputs a board (position).
         readmove : inputs a move.
         readboard: inputs a board (position).

*************************************************************************
*************************************************************************/

#include <stdio.h>
#include "chess.h"

char *term,          /* current terminal type */
     *getenv(),      /* gets an environment variable value */
     *tgetstr(),     /* gets a string from termcap */
     *tgoto(),       /* transforms "cm" string for some row/col values */
     cap[1024],      /* holds the termcap value for '*term' */
     *cm,            /* points to coded control sequence for going
                        directly to a position on the screen */
     *cl,            /* points to control string for clearing the
                        screen */
     area[100],      /* holds the values read from termcap
                        ('cm' & 'cl') */
     *parea;         /* points to the next free space in 'area' */

/*************************************************************************/
/*  setterm:                                                             */
/*        Reads the "cl" (clear screen) and "cm" ('cursor move')         */
/*      strings form the termcap of the current terminal type            */
/*      ('TERM' environment variable). 'cl' and 'cm' point to            */
/*      these strings. Must be called before using 'go()' or             */
/*      'clr()'.                                                         */
/*************************************************************************/

void setterm () {
  term = getenv ("TERM");
  tgetent (cap,term);
  parea = area;
  cm = tgetstr ("cm",&parea);
  cl = tgetstr ("cl",&parea);
}
```

```
/*************************************************************************/
/*  go:                                                                  */
/*     Puts the cursor at row 'r' and column 'c' .Must call 'setterm()' */
/*    before using 'go()' for the first time.                           */
/*************************************************************************/

void go (r,c)
int r,c; {
  printf ("%s",tgoto(cm,c,r));
}

/*************************************************************************/
/*  clr:                                                                 */
/*      Clears the screen. Must call 'setterm()' before using 'clr()'    */
/*    for the first time.                                                */
/*************************************************************************/

void clr() {
  printf ("%s",cl);
}

/*************************************************************************/
/*  pause:                                                               */
/*        Prints a message at bottom of screen and waits for user to     */
/*      press a <return>.                                                */
/*************************************************************************/

void pause() {
  go (22,20); printf ("PRESS <RETURN>");
  getc (stdin);
}

/*************************************************************************/
/*  outpc:                                                               */
/*      Outputs the piece corresponding to 'p', followed by a space.     */
/*       K is a king.                                                    */
/*       Q is a queen.                                                   */
/*       B is a bishop.                                                  */
/*       N is a knight                                                   */
/*       R is a rook.                                                    */
/*       P is a pawn.                                                    */
/*       E refers to an en-passant pawn capture.                         */
/*                                                                       */
/*      White pieces are in upper case and black ones in lower case      */
/*      An empty square (no color) is '.'                                */
/*************************************************************************/

void outpc (p)
int p; {
  char pc;
  switch (p & 7) {
    case K : pc = 'K'; break;
    case Q : pc = 'Q'; break;
    case B : pc = 'B'; break;
    case N : pc = 'N'; break;
    case R : pc = 'R'; break;
    case P : pc = 'P'; break;
    case E : pc = 'E'; break;
```

```
  }
  switch (p & BOTH) {
    case 0 : printf ("."); break;
    case BL : pc += 'a' - 'A';
    case WH : printf ("%c",pc); break;
    case BOTH : printf ("X");
  }
  printf (" ");
}

/*************************************************************************/
/*  outsq:                                                               */
/*        Outputs the square 'sq' in european (?) notation               */
/*      (a1, a2, ... , h8).                                              */
/*************************************************************************/

void outsq (sq)
T_sq sq; {
  char c;
  c = 'a' + sq.c - 2;
  printf ("%c%1d",c,sq.r-1);
}

/*************************************************************************/
/*  outmove:                                                             */
/*        Output the move 'm' ('m.from', 'm.to', 'm.prey').              */
/*      Example: "d1 d8 q" means from d1 to d8 taking a black queen.     */
/*************************************************************************/

void outmove (m)
T_move m; {
  outsq (m.from);
  outsq (m.to);
  printf (" ");
  outpc (m.prey);
  printf (" ");
}

/*************************************************************************/
/*  outnode:                                                             */
/*        Outputs the contents of the node '*n' (used for debugging).    */
/*************************************************************************/

void outnode (n)
T_node *n; {
  int i;
  printf ("  moves= %4d    score= %4d \n",n->num,n->score);
  for (i=0;i<n->num;i++)
    outmove (n->mvl[i]);
}

/*************************************************************************/
/*  outboard:                                                            */
/*        Outputs the board '*mb' at row 'r' and column 'c' on the       */
/*      screen.                                                          */
/*      Some extra information could be added (score, tomove...).        */
/*************************************************************************/
```

```
void outboard (mb,row,col)
T_board *mb; int row,col;
{
  int r,c;
  for (r=9;r>=2;r--) {
    go (row++,col);
    for (c=2;c<10;c++)
      outpc (mb->bd[r][c]);
  }
  fflush (stdout);
}

/*************************************************************************/
/*  readmove:                                                            */
/*        Reads a move from '*infile' into '*mm'. The move is entered    */
/*      like in "e2e4<return>".                                          */
/*      Absolutely no checking is made for the move. To be added by      */
/*      calling 'genmoves()' and testing if the move is in the list.     */
/*************************************************************************/

void readmove(infile,mm)
T_move *mm;
FILE *infile;
{ char c[50];

  go(21,20);
  printf(" Please enter move ( ex. a4b5 ):");
  fscanf(infile,"%s",c);
  mm->from.c=(c[0] - 'a') + 2;
  mm->from.r=(c[1] - '1') + 2;
  mm->to.c=(c[2] - 'a') + 2;
  mm->to.r=(c[3] - '1') + 2;
  mm->prey=Gboard.bd[mm->to.r][mm->to.c];

  /*---------------------------------*/
  /* check for an En-Passant capture */
  /*---------------------------------*/
  if (Gboard.EP)
  { if ((((Gboard.bd[mm->from.r][mm->from.c] & 7) == P) &&
        (mm->to.r==(9-(Gboard.tomove/3))) && (mm->to.c==Gboard.EP))
      mm->prey=(E|Gboard.idle); }

  /*--------------------------------*/
  /* check for promotion to a Queen */
  /*--------------------------------*/
  if ((Gboard.tomove == WH) && (mm->to.r == 9) &&
     ((Gboard.bd[mm->from.r][mm->from.c] & 7) == P))
    mm->prey = mm->prey|QUEEN;
  else
  { if ((Gboard.tomove == BL) && (mm->to.r == 2) &&
        ((Gboard.bd[mm->from.r][mm->from.c] & 7) == P))
     mm->prey = mm->prey|QUEEN; }

  /*-----------------------------------------------------------*/
  /* if King move, set flag to indicate first movement of King */
  /*-----------------------------------------------------------*/
  if ((((Gboard.bd[mm->from.r][mm->from c] & 7) == K) &&
     (mm->from.c==6) && (Gboard.castlefl[Gboard.tomove/16]==0))
```

```
    mm->prey = mm->prey|CAST;
  }

/*************************************************************************/
/*  readboard:                                                           */
/*        Reads a position into '*mb' from '*infile'. The first line     */
/*      is the player to move ("W" or "B") and the eight following       */
/*      lines are eight characters representing the contents of          */
/*      the board (same convention as 'outpc' above). Row a is at        */
/*      the bottom and h at the top, column 1 is the first character     */
/*      of each line and 8 is the last. If 'infile == stdin', the        */
/*      instructions for entering the board are output on the            */
/*      screen.                                                          */
/*        Also sets up the material value of the presence of a piece     */
/*      (opponent's pieces have negative value) and sets up the          */
/*      two-colored border squares.                                      */
/*        Also sets up the initial castling status flags used            */
/*      to monitor castling possibilities for both players.              */
/*************************************************************************/

void  readboard(infile,mb)
FILE *infile; T_board *mb;
{ char c[8];  int  i,j,p,cl;

  for (i=0;i<32;i++)
    val[i]=0;

  /*-----------------------------------------------------------------*/
  /* initialize array of piece values, only elements 9-23 are used */
  /*-----------------------------------------------------------------*/
  val[9]=100;val[10]=9;val[11]=5;val[12]=3;val[13]=3;val[14]=1;val[15]=1;
  for (i=9;i<16;i++)
    val[i+8] = -val[i];

  if (infile==stdin) {
    clr();
    printf("\n  Welcome to the FIXAFAN chess playing program : \n\n");
    printf("  Please enter your choice of color (W or B):");
  }

  fscanf(infile,"%s",c);              /* opponent's choice of color */
  if ((c[0]=='B') || (c[0]=='b'))
       mb->tomove=WH;                 /*  'tomove" field represents
                                          computer's color */
  else { mb->tomove=BL;
         /*-------------------------------------------------*/
         /* if opponent chooses White, negate values of pieces */
         /*-------------------------------------------------*/
         for (i=8; i<=23; i++)
           val[i]= -val[i];
        }

  mb->idle=(mb->tomove ^ BOTH);
  mb->h=0;    mb->hashcode.m = random();  mb->hashcode.k = random();

  /*------------------------------------------- -*/
  /* initialize Piece-Square table of random numbers */
  /*-----------------------------------------------*/
```

```
initPStab();

if (infile==stdin) {
  printf("  Please enter board one row at a time:\n");
  printf("   black pieces=> 'k','q','b','n','r','p' \n");
  printf("   white pieces=> 'K','Q','B','N','R','P' \n");
  printf("   blank square=> '.' \n");
  printf("   black pieces at top of board, white pieces at bottom\n\n");
}

for (i=9; i>=2; i--)
{ fscanf (infile,"%s",c);
  for (j=2; j<=9; j++) {
    if (c[j-2]>='a' && c[j-2]<='z') {
      c[j-2] += 'A'-'a';
      cl = BL;
    }
    else
      cl = WH;
    switch (c[j-2])
    { case 'K': p = K; break;
      case 'Q': p = Q; break;
      case 'B': p = B; break;
      case 'N': p = N; break;
      case 'R': p = R; break;
      case 'P': p = P; break;
      case '.': p = X;
     }
    if (p==X)
      mb->bd[i][j]=0;
    else {
    /*-----------------------------------------------------------------*/
    /* initialize value of board squares and cummulative board score */
    /*-----------------------------------------------------------------*/
      mb->bd[i][j]=p|cl;
      mb->h += val[p|cl];

    /*-------------------------------------------------------*/
    /* initialize both parts of cummulative board hashcode */
    /*-------------------------------------------------------*/
      if (cl==BL) p+=7;
      mb->hashcode.m^=PS_table[p-1][i-2][j-2].m;
      mb->hashcode.k^=PS_table[p-1][i-2][j-2].k;
    }
  }

  /*-------------------------------------------------------*/
  /* initialize boundaries of the board to BOTH = "no color" */
  /*-------------------------------------------------------*/
  mb->bd[i][0]= BOTH;  mb->bd[i][1]= BOTH;
  mb->bd[i][10]=BOTH;  mb->bd[i][11]=BOTH;
}
for (j=0;j<12;j++) {
  mb->bd[0][j]=BOTH;
  mb->bd[1][j]=BOTH;
  mb->bd[10][j]=BOTH;
  mb->bd[11][j]=BOTH;
}
```

```
/*-------------------------------------------------------------*/
/* if King is in its initial position, initialize castling flags */
/* start with flags for Black player                           */
/*-------------------------------------------------------------*/
if (mb->bd[9][6]!=(K|BL))  mb->castlefl[1]=1;
else {
       /*-----------------------------------------------*/
       /* check if Black left Rook is at initial position */
       /*-----------------------------------------------*/
       if (mb->bd[9][2]!=(R|BL))  mb->castlestat=mb->castlestat|128;
       else {
              /*-----------------------------------------------*/
              /* check if squares between Rook & King are empty */
              /*-----------------------------------------------*/
              if ((mb->bd[9][3])||(mb->bd[9][4])||(mb->bd[9][5]))
                  mb->castlestat=mb->castlestat|32;
              else {
                     /*-----------------------------------------*/
                     /* include left castling option in hashcodes */
                     /*-----------------------------------------*/
                     mb->hashcode.m^=PS_table[14][0][6].m;
                     mb->hashcode.k^=PS_table[14][0][6].k; }
              /*-----------------------------------------------*/
              /* hashcode to indicate left Rook is still unmoved */
              /*-----------------------------------------------*/
              mb->hashcode.m^=PS_table[14][0][4].m;
              mb->hashcode.k^=PS_table[14][0][4].k; }

       /*-----------------------------------------------*/
       /* check if Black right Rook is at initial position */
       /*-----------------------------------------------*/
       if (mb->bd[9][9]!=(R|BL))  mb->castlestat=mb->castlestat|64;
       else {
              /*-----------------------------------------------*/
              /* check if squares between Rook & King are empty */
              /*-----------------------------------------------*/
              if ((mb->bd[9][7])||(mb->bd[9][8]))
                  mb->castlestat=mb->castlestat|16;
              else {
                     /*-----------------------------------------*/
                     /* include left castling option in hashcodes */
                     /*-----------------------------------------*/
                     mb->hashcode.m^=PS_table[14][0][7].m;
                     mb->hashcode.k^=PS_table[14][0][7].k; }
              /*-----------------------------------------------*/
              /* hashcode to indicate left Rook is still unmoved */
              /*-----------------------------------------------*/
              mb->hashcode.m^=PS_table[14][0][5].m;
              mb->hashcode.k^=PS_table[14][0][5].k; }
     }

/*---------------------------------------*/
/* repeat above tests for the White player */
/*---------------------------------------*/

if (mb->bd[2][6]!=(K|WH))  mb->castlefl[0]=1;
else {
```

```
            if (mb->bd[2][2]!=(R|WH))  mb->castlestat=mb->castlestat|8;
            else {
                   if ((mb->bd[2][3])||(mb->bd[2][4])||(mb->bd[2][5]))
                       mb->castlestat=mb->castlestat|2;
                   else {
                          mb->hashcode.m^=PS_table[14][0][2].m;
                          mb->hashcode.k^=PS_table[14][0][2].k; }
                   mb->hashcode.m^=PS_table[14][0][0].m;
                   mb->hashcode.k^=PS_table[14][0][0].k; }

            if (mb->bd[2][9]!=(R|WH))  mb->castlestat=mb->castlestat|4;
            else {
                   if ((mb->bd[2][7])||(mb->bd[2][8]))
                       mb->castlestat=mb->castlestat|1;
                   else {
                          mb->hashcode.m^=PS_table[14][0][3].m;
                          mb->hashcode.k^=PS_table[14][0][3].k; }
                   mb->hashcode.m^=PS_table[14][0][1].m;
                   mb->hashcode.k^=PS_table[14][0][1].k; }
           }
    }
```

```
/*****************************************************************************
 *****************************************************************************
 chessmv.c :
          Includes functions dependent on board representation.

          movegen : generates a moves list from the game board
          update  : updates the board by executing a move
          restore : restores the board by undoing a move

*****************************************************************************
*****************************************************************************/

#include <stdio.h>
#include "chess.h"
#define  m_board  board

/*****************************************************************************/
/*   addmv:                                                                  */
/*         Adds a move to the node pointed by 'n'. The other                 */
/*       parameters "r,c,tr,tc,prey" describe the move. The                  */
/*       remaining parameters "check,k,l,row" are used                       */
/*       only when checking for the legality for a potential                 */
/*       castling move.                                                      */
/*         The variable 'nextcap' must be set to the first move              */
/*       of the moves list so that the captures can be placed                */
/*       at the top of the list.                                             */
/*         The use of a macro instead of a function increases                */
/*       speed of genmoves by 25%.                                           */
/*****************************************************************************/

#define  addmv(n,r,c,ti,tc,pr,check,x,k,l,row) \
   { register  T_move *ad;   \
     register  int  ii,frr,fcc,trr,tcc; \
     if (check)  \
     { frr=r;  fcc=c;  trr=tr;  tcc=tc;\
       if (/*((frr==row) && (fcc>=k) && (fcc<=l)) \
          ||*/ ((trr==row) && (tcc>=k) && (tcc<=l))) \
         stopcheck=TRUE; } \
     else  \
     { if ((captures) && (pr & BOTH))  \
       { n->mvl[n->num] = n->mvl[n->killmvs]; \
         ii=0; \
           /* order captures as they are being generated */ \
         while ((abs(val[pr&31]) < abs(val[n->mvl[ii].prey&31]))\
                && (ii < n->num))  ii++; \
         if ((ii+1)<n->killmvs) \
         { n->mvl[n->killmvs] = n->mvl[ii+1]; \
           n->mvl[ii+1] = n->mvl[ii]; }  \
         else \
           n->mvl[n->killmvs] = n->mvl[ii]; \
         /* 'ad' is the position where the move will be inserted */\
         ad = n->mvl + ii;  n->killmvs++; \
        } \
       else  \
          /* insert the move at the end of the list */ \
         ad = n->mvl + (n->num); \
       if ((x != -1) && (x < n->num) && (x != (n->killmvs-1))) \
```

```
      {   /* if 'x' is positive then move will be placed at the top */\
        if ((x+1)<n->num) \
        { *ad = n->mvl[x+1];  n->mvl[x+1] = n->mvl[x]; } \
        else \
          *ad = n->mvl[x];  \
        ad = n->mvl + x;  \
        if (n->killmvs==x)  n->killmvs++; \
       }  \
        /* insert the move in the list */ \
      ad->from.r = r; \
      ad->from.c = c; \
      ad->to.r = tr; \
      ad->to.c = tc; \
      ad->prey = pr; \
      n->num++; \
     } \
    }

/*************************************************************************/
/*  kmorder:                                                             */
/*       Determines exact location of a killer move in the move          */
/*     list. i.e. returns location in variable 'x' as either 0,          */
/*     1, or 2 (depending on value of offset 'k', which may              */
/*     reserve location 0 for an old principle continuation move).       */
/*************************************************************************/

#define kmorder(n,d,r,c,tr,tc,tm,prey,k,x) \
   x= -1;  /* default value if the move is NOT a killer move */ \
   if ((last2) || (dynamic)) \
   { if ((tc== kmlist[d][0].to.c) && (tr== kmlist[d][0].to.r) && \
        (c==kmlist[d][0].from.c) && (r==kmlist[d][0].from.r) && \
        ((prey&31)==( kmlist[d][0].prey&31)))  \
      x=k;  /* return value 'k' for the last killer move */ \
    else  \
    { if ((tc== kmlist[d][1].to.c) && (tr== kmlist[d][1].to.r) && \
          (c==kmlist[d][1].from.c) && (r==kmlist[d][1].from.r) && \
          ((prey&31)==( kmlist[d][1].prey&31))) \
        x=k+1;  /* return value 'k+1' for the 2nd last killer move */ \
     } \
    } \
   if (history) \
   { pts=kmtable[r][c][tr][tc][tm]; \
     if (pts > minpts)  /* keep track of 2 highest point totals */ \
       { if (pts > high)  \
         { minpts=high;  high=pts;  x=k; }  \
         else  \
         { minpts=pts;  x=k+1; } \
        }  \
    }  \
   else \
     if (((last1)&&('last2)) || (bestkm) || (assoc)) \
     { if ((tc== highest[d].km.to.c) && (tr== highest[d].km.to.r) && \
          (c==highest[d].km.from.c) && (r==highest[d].km.from.r) && \
          ((prey&31)==(highest[d].km.prey&31)))  \
        x=k;   /* only one possible killer move per ply here */ \
      }
```

```
/*************************************************************************/
/*    genmoves:                                                          */
/*          Generates all moves possible from the scratch                */
/*        board into the node pointed to by 'n'. Returns the             */
/*        number of moves generated. The king is considered              */
/*        as a regular piece, so the moves are pseudo-legal.             */
/*        Generates en-passant & castleing moves whenever                */
/*        the castleing status flags permits it.                         */
/*        The game board is not a parameter, the use                     */
/*        registers , '*a' & '*aa', and very repetitive code             */
/*        attempt to increase speed of execution.                        */
/*************************************************************************/

genmoves (n,d,check,k,l)
T_node *n;
int   d,check,k,l;          /* used for checking if the free squares
                               between king & rook are not in check */
{ register index r,c;       /* 'from' square for which moves are
                               currently being generated */
  index rr,cc;              /* current 'to' square for rays */

  register T_piece   *a,    /* points to square [r,c] */
                    *aa,    /* points to square [rr,cc] */
                  prey,     /* saves contents of 'to' square */
                  idle,     /* saves 'lboard.idle' in register */
                tomove,     /* saves 'lboard.tomove' in register */
           castlestat,      /* saves 'lboard.castlestat' */
                   EP;      /* saves 'lboard.EP' */

  register int  row, col, shift, x, y, z, tm, next,
                pts, minpts, high;

  if (check==0)             /* if this is a genuine move generation */
  { nodecount++;  n->num=0;  n->killmvs=0; }
  stopcheck=0;    minpts=0;   high=0;
  castlestat = m_board.castlestat;   EP = m_board.EP;

  if (check)   /* if this is just a check for castling interference */
  { tomove = m_board.idle;  idle = m_board.tomove;    tm=0;
    if (tomove==WH)
    { row=9; a=&m_board.bd[10][9]; next=-20; r=10; z=-1; c=2; }
    else
    { row=2; a=&m_board.bd[1][9]; next=4; r=1; z=1; c=2; }
   }
  else                      /* if this is a genuine move generation */
  { tomove = m_board.tomove;   idle = m_board.idle;    tm=d&1;
    if (tomove== WH)
    { row=2;  shift=1;  next=-20;  z=-1;
      /*-------------------------------------------------------------*/
      /* non-zero 'l' parameter sets up generation for 1 piece only */
      /*-------------------------------------------------------------*/
      if (l)
      { a=(&m_board.bd[k][l])+19;  r=k+1;  c=l;  k=0; }
      else
      { a=&m_board.bd[10][9];  r=10;  c=2; }
     }

    else      /* BL */
```

```
  { row=9;  shift=16;  next=4;  z=1;
    if (1)
    { a=(&m_board.bd[k][1])-5;  r=k-1;  c=1;  k=0; }
    else
    { a=&m_board.bd[1][9];  r=1;  c=2; }
  }
 }

for (y=0; y<8; y++)
{ a += next;          /* address of next board location to look at */
  r += z;             /* row */
  for (; c<10; c++)  /* column */
  { a += 1;
    if (stopcheck==0)        /* variable used to abort generation */
    {if (*a & tomove)                /* if there is a BL/WH piece */

     {switch (*a & 7)                        /* what type of piece? */
      {
        case K :                                              /* KING */

          if ((m_board.castlefl[tomove/16]==0) && (check==0))
          { if (((castlestat & (10*shift))==0)
                && ((m_board.bd[row][2] & 7)==R))
            {
              /*-------------------------------------------*/
              /* check for castling possibility (left side) */
              /*-------------------------------------------*/
              genmoves(n,d,TRUE,2,6);      /* call with check=TRUE */

              if (stopcheck==0)
              {
                /*---------------------------------*/
                /* if NO reason to forbid castling */
                /*---------------------------------*/
                prey=CAST;  col=4;
                if (check==0)   kmorder(n,d,r,c,r,col,tm,prey,k,x);
                addmv(n,r,c,r,col,prey,check,x,check,check,check);
               }
              else  stopcheck=0;
             }
            if (((castlestat & (5*shift))==0)
                && ((m_board.bd[row][9] & 7)==R))
            {
              /*-------------------------------------------*/
              /* check for castling possibility (right side) */
              /*-------------------------------------------*/
              genmoves(n,d,TRUE,6,9);      /* call with check=TRUE */
              if (stopcheck==0)
              {
                /*---------------------------------*/
                /* if NO reason to forbid castling */
                /*---------------------------------*/
                prey=CAST;  col=8;
                if (check==0)   kmorder(n,d,r,c,r,col,tm,prey,k,x);
                addmv(n,r,c,r,col,prey,check,x,check,check,check);
               }
              else  stopcheck=0;     /* reset 'stopcheck' to null */
             }
```

```
    /*-----------------------------------------*/
    /* if king has not moved yet set 'stopcheck' */
    /*-----------------------------------------*/
    if ((r==row) && (c==6))  stopcheck=1;
      else  stopcheck=0;
   }

  /*-----------------------------------------*/
  /* check regular movement in all 8 directions */
  /*-----------------------------------------*/
  if (((prey= *(a-13)) & tomove) == 0)
  { if (stopcheck)  prey=prey|CAST;
    if (check==0)   kmorder(n,d,r,c,r-1,c-1,tm,prey,k,x);
    addmv(n,r,c,r-1,c-1,prey,check,x,k,l,row); }
  if (((prey= *(a-12)) & tomove) == 0)
  { if (stopcheck)  prey=prey|CAST;
    if (check==0)   kmorder(n,d,r,c,r-1,c,tm,prey,k,x);
    addmv(n,r,c,r-1,c,prey,check,x,k,l,row); }
  if (((prey= *(a-11)) & tomove) == 0)
  { if (stopcheck)  prey=prey|CAST;
    if (check==0)   kmorder(n,d,r,c,r-1,c+1,tm,prey,k,x);
    addmv(n,r,c,r-1,c+1,prey,check,x,k,l,row); }

  if (((prey= *(a-1)) & tomove) == 0)
  { if (stopcheck)  prey=prey|CAST;
    if (check==0)   kmorder(n,d,r,c,r,c-1,tm,prey,k,x);
    addmv(n,r,c,r,c-1,prey,check,x,k,l,row); }
  if (((prey= *(a+1)) & tomove) == 0)
  { if (stopcheck)  prey=prey|CAST;
    if (check==0)   kmorder(n,d,r,c,r,c+1,tm,prey,k,x);
    addmv(n,r,c,r,c+1,prey,check,x,k,l,row); }

  if (((prey= *(a+11)) & tomove) == 0)
  { if (stopcheck)  prey=prey|CAST;
    if (check==0)   kmorder(n,d,r,c,r+1,c-1,tm,prey,k,x);
    addmv(n,r,c,r+1,c-1,prey,check,x,k,l,row); }
  if (((prey= *(a+12)) & tomove) == 0)
  { if (stopcheck)  prey=prey|CAST;
    if (check==0)   kmorder(n,d,r,c,r+1,c,tm,prey,k,x);
    addmv(n,r,c,r+1,c,prey,check,x,k,l,row); }
  if (((prey= *(a+13)) & tomove) == 0)
  { if (stopcheck)  prey=prey|CAST;
    if (check==0)   kmorder(n,d,r,c,r+1,c+1,tm,prey,k,x);
    addmv(n,r,c,r+1,c+1,prey,check,x,k,l,row); }
  if (check==0)  stopcheck=0;
  break;

case Q : ;                                          /* QUEEN */

case B :                                           /* BISHOP */

  /*-----------------------------------------*/
  /* check BISHOP movement in all 4 directions */
  /*-----------------------------------------*/
  aa = a; rr = r; cc = c;
  while ((prey= *(aa -= 13)) < 8)
  { if (check==0)   kmorder(n,d,r,c,rr-1,cc-1,tm,prey,k,x);
    addmv(n,r,c,--rr,--cc,prey,check,x,k,l,row); }
```

```
  if ((prey & tomove) == 0)
  { if (check==0)   kmorder(n,d,r,c,rr-1,cc-1,tm,prey,k,x),
    addmv(n,r,c,--rr,--cc,prey,check,x,k,l,row); }

  aa = a; rr = r; cc = c;
  while ((prey= *(aa -= 11)) < 8)
  { if (check==0)   kmorder(n,d,r,c,rr-1,cc+1,tm,prey,k,x);
    addmv(n,r,c,--rr,++cc,prey,check,x,k,l,row); }
  if ((prey & tomove) == 0)
  { if (check==0)   kmorder(n,d,r,c,rr-1,cc+1,tm,prey,k,x);
    addmv(n,r,c,--rr,++cc,prey,check,x,k,l,row); }

  aa = a; rr = r; cc = c;
  while ((prey= *(aa += 13)) < 8)
  { if (check==0)   kmorder(n,d,r,c,rr+1,cc+1,tm,prey,k,x);
    addmv(n,r,c,++rr,++cc,prey,check,x,k,l,row); }
  if ((prey & tomove) == 0)
  { if (check==0)   kmorder(n,d,r,c,rr+1,cc+1,tm,prey,k,x);
    addmv(n,r,c,++rr,++cc,prey,check,x,k,l,row); }

  aa = a; rr = r; cc = c;
  while ((prey= *(aa += 11)) < 8)
  { if (check==0)   kmorder(n,d,r,c,rr+1,cc-1,tm,prey,k,x);
    addmv(n,r,c,++rr,--cc,prey,check,x,k,l,row); }
  if ((prey & tomove) == 0)
  { if (check==0)   kmorder(n,d,r,c,rr+1,cc-1,tm,prey,k,x);
    addmv(n,r,c,++rr,--cc,prey,check,x,k,l,row); }

  if ((*a & 7) == B)  break;     /* if QUEEN do NOT break */

case R :                                              /* ROOK */

  if ((check==0) && (r==row) && ((*a & 7)!=Q) &&
      ((castlestat & (12*shift))!=(12*shift)))
  {
    /*-----------------------------------------*/
    /* if ROOK has not moved yet set 'stopcheck' */
    /*-----------------------------------------*/
    if (((c==2) &&
        ((castlestat & (8*shift))==0)) ||
        ((c==9) &&
        ((castlestat & (4*shift))==0)))
      stopcheck=1; }

  /*---------------------------------------*/
  /* check ROOK movement in all 4 directions */
  /*---------------------------------------*/
  aa = a;  rr = r;
  while ((prey = *(aa -= 12)) < 8)
  { if (stopcheck) prey=prey|CAST;
    if (check==0)   kmorder(n,d,r,c,rr-1,c,tm,prey,k,x);
    addmv(n,r,c,--rr,c,prey,check,x,k,l,row); }
  if ((prey & tomove) == 0)
  { if (stopcheck) prey=prey|CAST;
    if (check==0)   kmorder(n,d,r,c,rr-1,c,tm,prey,k,x);
    addmv(n,r,c,--rr,c,prey,check,x,k,l,row); }

  aa = a;  rr = r;
```

```
    while ((prey = *(aa += 12)) < 8)
    { if (stopcheck) prey=prey|CAST;
      if (check==0)   kmorder(n,d,r,c,rr+1,c,tm,prey,k,x);
      addmv(n,r,c,++rr,c,prey,check,x,k,l,row); }
    if ((prey & tomove) == 0)
    { if (stopcheck) prey=prey|CAST;
      if (check==0)   kmorder(n,d,r,c,rr+1,c,tm,prey,k,x);
      addmv(n,r,c,++rr,c,prey,check,x,k,l,row); }

    aa = a;  cc = c;
    while ((prey = *(aa -= 1)) < 8)
    { if (stopcheck) prey=prey|CAST;
      if (check==0)   kmorder(n,d,r,c,r,cc-1,tm,prey,k,x);
      addmv(n,r,c,r,--cc,prey,check,x,k,l,row); }
    if ((prey & tomove) == 0)
    { if (stopcheck) prey=prey|CAST;
      if (check==0)   kmorder(n,d,r,c,r,cc-1,tm,prey,k,x);
      addmv(n,r,c,r,--cc,prey,check,x,k,l,row); }

    aa = a;  cc = c;
    while ((prey = *(aa += 1)) < 8)
    { if (stopcheck) prey=prey|CAST;
      if (check==0)   kmorder(n,d,r,c,r,cc+1,tm,prey,k,x);
      addmv(n,r,c,r,++cc,prey,check,x,k,l,row); }
    if ((prey & tomove) == 0)
    { if (stopcheck) prey=prey|CAST;
      if (check==0)   kmorder(n,d,r,c,r,cc+1,tm,prey,k,x);
      addmv(n,r,c,r,++cc,prey,check,x,k,l,row); }
    if (check==0)  stopcheck=0;
    break;

  case N :                                          /* KNIGHT */

    /*-------------------------------------------------*/
    /* check KNIGHT movement to all 8 possible squares */
    /*-------------------------------------------------*/
    if (((prey = *(a-25)) & tomove) == 0)
    { if (check==0)   kmorder(n,d,r,c,r-2,c-1,tm,prey,k,x);
      addmv(n,r,c,r-2,c-1,prey,check,x,k,l,row); }
    if (((prey = *(a-23)) & tomove) == 0)
    { if (check==0)   kmorder(n,d,r,c,r-2,c+1,tm,prey,k,x);
      addmv(n,r,c,r-2,c+1,prey,check,x,k,l,row); }
    if (((prey = *(a+23)) & tomove) == 0)
    { if (check==0)   kmorder(n,d,r,c,r+2,c-1,tm,prey,k,x);
      addmv(n,r,c,r+2,c-1,prey,check,x,k,l,row); }
    if (((prey = *(a+25)) & tomove) == 0)
    { if (check==0)   kmorder(n,d,r,c,r+2,c+1,tm,prey,k,x);
      addmv(n,r,c,r+2,c+1,prey,check,x,k,l,row); }
    if (((prey = *(a-14)) & tomove) == 0)
    { if (check==0)   kmorder(n,d,r,c,r-1,c-2,tm,prey,k,x);
      addmv(n,r,c,r-1,c-2,prey,check,x,k,l,row); }
    if (((prey = *(a-10)) & tomove) == 0)
    { if (check==0)   kmorder(n,d,r,c,r-1,c+2,tm,prey,k,x);
      addmv(n,r,c,r-1,c+2,prey,check,x,k,l,row); }
    if (((prey = *(a+10)) & tomove) == 0)
    { if (check==0)   kmorder(n,d,r,c,r+1,c-2,tm,prey,k,x);
      addmv(n,r,c,r+1,c-2,prey,check,x,k,l,row); }
    if (((prey = *(a+14)) & tomove) == 0)
```

```
  { if (check==0)   kmorder(n,d,r,c,r+1,c+2,tm,prey,k,x);
    addmv(n,r,c,r+1,c+2,prey,check,x,k,l,row); }
  break;

case P :                                              /* PAWN */

  if (tomove == WH) {
    if (EP)                    /* if en-passant possibility */
    { if ((r==6) && ((c==EP+1) || (c==EP-1)))
      { prey=(E|idle);
        if (check==0)
          kmorder(n,d,r,c,7,EP,tm,prey,k,x);
        addmv(n,r,c,7,EP,prey,check,x,k,l,row);
       }
     }

    /*-------------------------------------*/
    /* check the two possible pawn captures */
    /*-------------------------------------*/
    if (((prey = *(a+11)) & BOTH) == idle)
    { if (r==8) prey=prey|QUEEN;
      if (check==0)   kmorder(n,d,r,c,r+1,c-1,tm,prey,k,x);
      addmv(n,r,c,r+1,c-1,prey,check,x,k,l,row); }
    if (((prey = *(a+13)) & BOTH) == idle)
    { if (r==8) prey=prey|QUEEN;
      if (check==0)   kmorder(n,d,r,c,r+1,c+1,tm,prey,k,x);
      addmv(n,r,c,r+1,c+1,prey,check,x,k,l,row); }

    /*-------------------------------------*/
    /* check for ordinary straight ahead move */
    /*-------------------------------------*/
    if (((prey = *(a+12)) & BOTH) == 0) {
    { if (r==8) prey=prey|QUEEN;
      if (check==0)   kmorder(n,d,r,c,r+1,c,tm,prey,k,x);
      addmv(n,r,c,r+1,c,prey,check,x,k,l,row); }

      /*-----------------------------*/
      /* check for opening double move */
      /*-----------------------------*/
      if ((r==3) && (((prey = *(a+24)) & BOTH) == 0))
      { if (check==0)   kmorder(n,d,r,c,r+2,c,tm,prey,k,x);
        addmv(n,r,c,r+2,c,prey,check,x,k,l,row); }
    }
  }

  else {     /* BL */
    if (EP)                   /* if en-passant possibility */
    { if ((r==5) && ((c==EP+1) || (c==EP-1)))
      { prey=(E|idle);
        if (check==0)
          kmorder(n,d,r,c,4,EP,tm,prey,k,x);
        addmv(n,r,c,4,EP,prey,check,x,k,l,row);
       }
     }

    /*-------------------------------------*/
    /* check the two possible pawn captures */
    /*-------------------------------------*/
```

```
                if (((prey = *(a-13)) & BOTH) == idle)
                { if (r==3) prey=prey|QUEEN;
                  if (check==0)  kmorder(n,d,r,c,r-1,c-1,tm,prey,k,x);
                  addmv(n,r,c,r-1,c-1,prey,check,x,k,l,row); }
                if (((prey = *(a-11)) & BOTH) == idle)
                { if (r==3) prey=prey|QUEEN;
                  if (check==0)  kmorder(n,d,r,c,r-1,c+1,tm,prey,k,x);
                  addmv(n,r,c,r-1,c+1,prey,check,x,k,l,row); }

                /*-------------------------------------*/
                /* check for ordinary straight ahead move */
                /*-------------------------------------*/
                if (((prey = *(a-12)) & BOTH) == 0) {
                { if (r==3) prey=prey|QUEEN;
                  if (check==0)  kmorder(n,d,r,c,r-1,c,tm,prey,k,x);
                  addmv(n,r,c,r-1,c,prey,check,x,k,l,row); }

                  /*-------------------------------*/
                  /* check for opening double move */
                  /*-------------------------------*/
                  if ((r==8) && (((prey = *(a-24)) & BOTH) == 0))
                  { if (check==0)  kmorder(n,d,r,c,r-2,c,tm,prey,k,x);
                    addmv(n,r,c,r-2,c,prey,check,x,k,l,row); }
                }
              }
              break;
            }
           }
          }
         else return;                    /* 'stopcheck' was non-zero */

         if ((1) && (check==0))       /* if 1-piece partial generation */
           return(n->num);
       } c=2;            /* reset column to 2 before incrementing row */
     }
     return(n->num);                    /* number of moves generated */
    }

/*****************************************************************/
/*   bad:                                                        */
/*       Indicates if square is outside the board.               */
/*****************************************************************/

bad (sq)
T_sq  sq;
{ if (sq.c>9 || sq.c<2)  return(1);
  if (sq.r>9 || sq.r<2)  return(1);
  return(0);
 }

/*****************************************************************/
/*   update:                                                     */
/*         Makes the move 'mv' on '*brd', all the relevant info  */
/*       in '*brd' is updated.  Special attention is given to    */
/*       the management of the castling status flags.            */
/*****************************************************************/
```

```
void  update (mv,brd,show)
T_move mv;  T_board *brd;  int show;
{ int    shift,clr,row,p;
  register int   hashm,hashk;

  hashm= brd->hashcode.m;
  hashk= brd->hashcode.k;
  if (bad(mv.to) || bad(mv.from))
  { outmove(mv);
    printf(" BAD MOVE'\n");
    exit(0); }

  p=brd->bd[mv.from.r][mv.from.c] & 7;          /*  value of piece */
  if (brd->tomove==BL) p+=7;

  /*-------------------------------------------------------*/
  /* update two hashcodes using TO & FROM piece-square values */
  /*-------------------------------------------------------*/
  hashm^=PS_table[p-1][mv.from.r-2][mv.from.c-2].m;
  hashk^=PS_table[p-1][mv.from.r-2][mv.from.c-2].k;
  hashm^=PS_table[p-1][mv.to.r-2][mv.to.c-2].m;
  hashk^=PS_table[p-1][mv.to.r-2][mv.to.c-2].k;

  /*-------------------------*/
  /* update game board itself */
  /*-------------------------*/
  brd->bd[mv.to.r][mv.to.c] = brd->bd[mv.from.r][mv.from.c];
  brd->bd[mv.from.r][mv.from.c] = 0;
  brd->h -= val[mv.prey&31];        /* adjust board score with prey */

  /*----------------------*/
  /* check for en-passant */
  /*----------------------*/
  if ((mv.prey & 7)==E)
  { brd->bd[row=(7 - (brd->tomove/8))][mv.to.c]=0;
    p=P;  if (brd->idle==BL)  p+=7;
    hashm^=PS_table[p-1][row-2][mv.to.c-2].m;
    hashk^=PS_table[p-1][row-2][mv.to.c-2].k; }
  else
  if (p=(mv.prey&7))
  { if (brd->idle==BL)  p+=7;
    hashm^=PS_table[p-1][mv.to.r-2][mv.to.c-2].m;
    hashk^=PS_table[p-1][mv.to.r-2][mv.to.c-2].k; }
  if (brd->EP!=0)
  { hashm^=PS_table[14][1][brd->EP-2].m;
    hashk^=PS_table[14][1][brd->EP-2].k; }
  if (((brd->bd[mv.to.r][mv.to.c] & 7)==P) &&
      (abs(mv.to.r - mv.from.r)==2))
  {
    /*-------------------------------------------------------*/
    /* set up en-passant possibility for opponent's next move */
    /*-------------------------------------------------------*/
    brd->EP=mv.to.c;
    hashm^=PS_table[14][1][brd->EP-2].m;
    hashk^=PS_table[14][1][brd->EP-2].k;
   }
  else  brd->EP=0;          /* NO possibility for en-passant capture */
```

```
/*--------------------*/
/* check for Queening */
/*--------------------*/
if (mv.prey & QUEEN)
{ brd->bd[mv.to.r][mv.to.c] = Q|brd->tomove;
  p=Q;  if (brd->tomove==BL)  p+=7;
        hashm^=PS_table[p-1][mv.to.r-2][mv.to.c-2].m;
        hashk^=PS_table[p-1][mv.to.r-2][mv.to.c-2].k;
  p+=4; hashm^=PS_table[p-1][mv.to.r-2][mv.to.c-2].m;
        hashk^=PS_table[p-1][mv.to.r-2][mv.to.c-2].k;
  brd->h -= (val[Q|brd->idle] + val[P|brd->tomove]);
 }

/*--------------------*/
/* check for castleing */
/*--------------------*/
if ((mv.prey & CAST) &&
    ((brd->bd[mv.to.r][mv.to.c] & 7)==K))
{ brd->castlefl[clr=brd->tomove/16]=1;
  if (mv.to.c==8)
  { brd->bd[row=(brd->tomove - clr - 6)][7]=R|brd->tomove;
    brd->bd[row][9]=0;
    p=R;  if (brd->tomove==BL)  p+=7;
    hashm^=PS_table[p-1][row-2][7].m;
    hashk^=PS_table[p-1][row-2][7].k;
    hashm^=PS_table[p-1][row-2][5].m;
    hashk^=PS_table[p-1][row-2][5].k; }
  else if (mv.to.c==4)
       { brd->bd[row=(brd->tomove-clr-6)][5]=R|brd->tomove;
         brd->bd[row][2]=0;
         p=R;  if (brd->tomove==BL)  p+=7;
         hashm^=PS_table[p-1][row-2][0].m;
         hashk^=PS_table[p-1][row-2][0].k;
         hashm^=PS_table[p-1][row-2][3].m;
         hashk^=PS_table[p-1][row-2][3].k; }
 }

else

  /*------------------------------*/
  /* update castleing status flags */
  /*------------------------------*/
{ if (brd->tomove==BL)  shift=16;  else shift=1;

  if ((brd->castlefl[clr=brd->tomove/16]==0) &&
      ((brd->castlestat & (12*shift))!=(12*shift)))
  { if (mv.from.r==(row=(brd->tomove - clr - 6)))
    switch (mv.from.c)
    { case 2: /* flag for left rook movement */

              if (((brd->bd[mv.to.r][mv.to.c] & 7)==R) &&
                  ((brd->castlestat & (8*shift))==0))
              { brd->castlestat=brd->castlestat|(8*shift);
                hashm^=PS_table[14][0][(shift/4)].m;
                hashk^=PS_table[14][0][(shift/4)].k;
                if ((brd->castlestat & (2*shift))==0)
                { hashm^=PS_table[14][0][(shift/4)+2].m;
                  hashk^=PS_table[14][0][(shift/4)+2].k; }
```

```
          }
          break;
  case 3: ;

  case 4: ;

  case 5: /* flag for empty squares left of king */

          if ((brd->castlestat & (8*shift))==0)
          { if ((brd->bd[row][3]==0) &&
                (brd->bd[row][4]==0) &&
                (brd->bd[row][5]==0))
            { brd->castlestat = brd->castlestat
                              & (255-(2*shift));
              hashm^=PS_table[14][0][(shift/4)+2].m;
              hashk^=PS_table[14][0][(shift/4)+2].k; }
          }
          break;
  case 7: ;

  case 8: /* flag for empty squares right of king */

          if ((brd->castlestat & (4*shift))==0)
          { if ((brd->bd[row][7]==0) &&
                (brd->bd[row][8]==0))
            { brd->castlestat = brd->castlestat
                              & (255-shift);
              hashm^=PS_table[14][0][(shift/4)+3].m;
              hashk^=PS_table[14][0][(shift/4)+3].k; }
          }
          break;
  case 9: /* flag for right rook movement */

          if (((brd->bd[mv.to.r][mv.to.c] & 7)==R) &&
              ((brd->castlestat & (4*shift))==0))
          { brd->castlestat=brd->castlestat|(4*shift);
            hashm^=PS_table[14][0][(shift/4)+1].m;
            hashk^=PS_table[14][0][(shift/4)+1].k;
            if ((brd->castlestat & shift)==0)
            { hashm^=PS_table[14][0][(shift/4)+3].m;
              hashk^=PS_table[14][0][(shift/4)+3].k; }
          }
 }

/*-----------------------------------------*/
/* if piece moved back to player's first row */
/*-----------------------------------------*/
if (mv.to.r==row)
{ if ((mv.to.c>=3) && (mv.to.c<=5))
  {
    /*-----------------------------------------*/
    /* cancel flag for free squares left of king */
    /*-----------------------------------------*/
    if ((brd->castlestat & (8*shift))==0)
    { brd->castlestat=brd->castlestat|(2*shift);
      hashm^=PS_table[14][0][(shift/4)+2].m;
      hashk^=PS_table[14][0][(shift/4)+2].k; }
  }
```

```
        else if ((mv.to.c==7) || (mv.to.c==8))
             {
               /*-----------------------------------------*/
               /* cancel flag for free squares right of king */
               /*-----------------------------------------*/
               if ((brd->castlestat & (4*shift))==0)
               { brd->castlestat=brd->castlestat|shift;
                 hashm^=PS_table[14][0][(shift/4)+3].m;
                 hashk^=PS_table[14][0][(shift/4)+3].k; }
              }
      }
     }

    /*---------------------------------------*/
    /* if piece moved into opponent's first row */
    /*---------------------------------------*/
    if ((brd->castlefl[clr=brd->idle/16]==0) &&
        (mv.to.r==(brd->idle - clr - 6)))
    { if (brd->idle==BL)  shift=16;  else shift=1;
      switch (mv.to.c)
      { case 3: ;
        case 4: ;
        case 5: /* cancel opponent's left side flag */
                if ((brd->castlestat & (8*shift))==0)
                { brd->castlestat=brd->castlestat|(2*shift);
                  hashm^=PS_table[14][0][(shift/4)+2].m;
                  hashk^=PS_table[14][0][(shift/4)+2].k; }
                break;
        case 7: ;
        case 8: /* cancel opponent's right side flag */
                if ((brd->castlestat & (4*shift))==0)
                { brd->castlestat=brd->castlestat|shift;
                  hashm^=PS_table[14][0][(shift/4)+3].m;
                  hashk^=PS_table[14][0][(shift/4)+3].k; }
      }
     }
    }

    /*-----------------------------------------------------*/
    /* change of player's turn, also done in the hash codes */
    /*-----------------------------------------------------*/
    brd->tomove ^= BOTH;
    brd->idle ^= BOTH;
    hashm^=PS_table[14][7][7].m;
    hashk^=PS_table[14][7][7].k;
    brd->hashcode.m=hashm;  brd->hashcode.k=hashk;

    if (live==UNDEAD)         /* if 'live' mode, show board movement */
    { go(2 + 9 - mv.from.r,2 + (mv.from.c * 2 -4));
      outpc(brd->bd[mv.from.r][mv.from.c]);
      go(2 + 9 - mv.to.r,2 + (mv.to.c * 2 -4));
      outpc(brd->bd[mv.to.r][mv.to.c]);
      getc(stdin);
     }
}
```

```
/*****************************************************************/
/*   restore:                                                    */
/*         Undoes the move 'mv' on '*brd'. 'mv' must be the      */
/*       last move made on '*brd'. All info is restored to       */
/*       the previous state of the board with special            */
/*       attention given to the castling status flags.           */
/*       Hashcodes are not restored but are retrieved using      */
/*       the 'oldhash' field of 'T_node' structure.              */
/*****************************************************************/

void restore (mv,brd,show)
T_move mv;  T_board *brd;  int show;
{ int  shift,clr,row;

  /*-----------------------------------*/
  /* restore game board and board score */
  /*-----------------------------------*/
  brd->bd[mv.from.r][mv.from.c] = m_board.bd[mv.to.r][mv.to.c];
  brd->bd[mv.to.r][mv.to.c] = mv.prey & 31;
  brd->h += val[mv.prey & 31];

  /*--------------------*/
  /* check for Queening */
  /*--------------------*/
  if (mv.prey & QUEEN)
  { brd->bd[mv.from.r][mv.from.c] = P|brd->idle;
    brd->h -= (val[Q|brd->idle] + val[P|brd->tomove]); }
  else
  /*----------------------*/
  /* check for en-passant */
  /*----------------------*/
  { if ((mv.prey & 7)==E)
    { brd->bd[7 - (brd->idle/8)][mv.to.c] = P|brd->tomove;
      brd->bd[mv.to.r][mv.to.c] = 0; }
   }

   /*------------------------------*/
   /* restore castling status flags */
   /*------------------------------*/

   clr=brd->idle/16;

   /*-------------------------------*/
   /* if move made was castling move */
   /*-------------------------------*/
   if ((mv.prey & CAST) && ((brd->bd[mv.from.r][mv.from.c]&7)==K))
   { brd->castleflg[clr]=0;
     if (mv.to.c==8)                /* restore right side castling */
     { brd->bd[row=(brd->idle - clr -6)][9]=R|brd->idle;
       brd->bd[row][7]=0; }
     else if (mv.to.c==4)            /* restore left side castling */
          { brd->bd[row=(brd->idle - clr -6)][2]=R|brd->idle;
            brd->bd[row][5]=0; }
    }

   else
   { if (brd->idle==BL)  shift=16;
                  else shift=1;
```

```
if (mv.from.r==(row=(brd->idle - clr -6)))
switch (mv.from.c)
{ case 2: /* restore flag for left rook movement */

          if (((brd->bd[mv.from.r][mv.from.c] & 7)==R) &&
              (mv.prey & CAST))
            brd->castlestat=brd->castlestat&(255-(8*shift));
          break;
  case 3: ;

  case 4: ;

  case 5: /* restore flag for squares left of king */

          if ((brd->castlestat & (8*shift))==0)
            brd->castlestat=brd->castlestat|(2*shift);
          break;
  case 7: ;

  case 8: /* restore flags for squares right of king */

          if ((brd->castlestat & (4*shift))==0)
            brd->castlestat=brd->castlestat|shift;
          break;
  case 9: /* restore flag for right rook movement */

          if (((brd->bd[mv.from.r][mv.from.c] & 7)==R) &&
              (mv.prey & CAST))
            brd->castlestat=brd->castlestat&(255-(4*shift));
}

/*-----------------------------------------*/
/* if move was made to the player's first row */
/*-----------------------------------------*/
if (mv.to.r==row)
{ if ((mv.to.c>=3) && (mv.to.c<=5))
  {
    /*-----------------------------------*/
    /* reset flag for squares left of king */
    /*-----------------------------------*/
    if ((brd->castlestat & (8*shift))==0)
      if ((brd->bd[row][3]==0) &&
          (brd->bd[row][4]==0) &&
          (brd->bd[row][5]==0))
        brd->castlestat=brd->castlestat & (255-(2*shift));
  }
 else
    if ((mv.to.c==7) || (mv.to.c==8))
    {
      /*-----------------------------------*/
      /* reset flag for squares right of king */
      /*-----------------------------------*/
      if ((brd->castlestat & (4*shift))==0)
        if ((brd->bd[row][7]==0) && (brd->bd[row][8]==0))
          brd->castlestat=brd->castlestat & (255-shift);
    }
}
```

```
    /*-----------------------------------------------*/
    /* if piece was moved to the opponent's first row */
    /*-----------------------------------------------*/
    if ((brd->castlefl[clr=brd->tomove/16]==0) &&
        (mv.to.r==(row=(brd->tomove - clr -6))))
    { if (brd->tomove==BL)  shift=16;  else shift=1;
      switch (mv.to.c)
      { case 3: ;
        case 4: ;
        case 5: /* reset flag for opponent's left side */
                if ((brd->castlestat & (8*shift))==0)
                  if ((brd->bd[row][3]==0) &&
                      (brd->bd[row][4]==0) &&
                      (brd->bd[row][5]==0))
                    brd->castlestat=brd->castlestat&(255-(2*shift));
                break;
        case 7: ;
        case 8: /* reset flag for opponent's right side */
                if ((brd->castlestat & (4*shift))==0)
                  if ((brd->bd[row][7]==0) &&
                      (brd->bd[row][8]==0))
                    brd->castlestat=brd->castlestat&(255-shift);
      }
     }
    }

    /*----------------------------*/
    /* reset player's turn to move */
    /*----------------------------*/
    brd->tomove ^= BOTH;
    brd->idle ^= BOTH;

    if (live==UNDEAD)      /* if 'live' mode, restore board movement */
    { go(2 + 9 - mv.from.r,2 + (mv.from.c * 2 -4));
      outpc(brd->bd[mv.from.r][mv.from.c]);
      go(2 + 9 - mv.to.r,2 + (mv.to.c * 2 -4));
      outpc(brd->bd[mv.to.r][mv.to.c]);
     }
}
```

Bibliography

[1] S.G. Akl and M. Newborn, *The Principle Continuation and the Killer Heuristic*, Proc. Annual Conf. Assoc. Comput. Mach., 1977, pp 466-473.

[2] T. Anantharaman, M. Campbell, and F. Hsu, *Singular Extensions: Adding Selectivity to Brute-Force Searching*, ICCA Journal, December 1988, pp 135-143.

[3] D.F. Beal, *Mixing Heuristic and Perfect Evaluations: Nested Minimax*, ICCA Journal, March 1984, pp 10-15.

[4] H.J. Berliner, *Some Necessary Conditions for a Master Chess Program*, Proceedings of the Third International Conference on Artificial Intelligence, 1973, pp 77-85.

[5] P. Bettadapur, *Design Assists for Capture Search in Chess*, M.Sc. thesis, Dept. of Computing Science, University of Alberta, Edmonton, 1986.

[6] P. Bettadapur, *Influence of Ordering on Capture Search*, ICCA Journal, December 1986, pp 180-188.

[7] P. Bettadapur and T.A. Marsland, *Accuracy and Savings in Depth-Limited Capture Search*, Int. J. Man-Machine Studies 29, 1988, pp 497-502.

[8] J. Condon and K. Thompson, *Belle Chess Hardware*, Advances in Computer Chess 3, M.R.B. Clarke (ed.), Pergamon Press, Oxford, 1982, pp 45-54.

[9] A.D. DeGroot, *Thought and Choice in Chess*, Mouton Publishers, The Hague, 1965 (2nd Ed. 1978).

[10] C. Ebeling and A. Palay, *The Design and Implementation of a VLSI Chess Move Generator*, Proceedings 11th Annual Symposium on Computer Architecture, 1984.

[11] E.W. Felten and S.W. Otto, *A Highly Parallel Chess Program*, unpublished manuscript, 1988.

[12] J. Gillogly, *The Technology Chess Program*, Artificial Intelligence 3, 1972, pp145-163.

[13] J. Gillogly, *Performance Analysis of the Technology Chess Program*, Ph.D. thesis, Computer Science Dept., Carnegie-Mellon University, Pittsburgh, 1978.

[14] J. Goulet, *Data Structures for Chess Programs*, M.Sc. thesis, School of Computer Science, McGill University, Montreal, 1986.

[15] H. Horacek, *Knowledge-Based Move Selection and Evaluation to Guide the Search in Chess Pawn Endings*, ICCA Journal, August 1983, pp 20-37.

[16] R.M. Hyatt, *Cray Blitz - A Computer Chess Playing Program*, M.Sc. thesis, University of Southern Mississippi, 1983.

[17] D. Knuth and R. Moore, *An Analysis of Alpha-Beta Pruning*, Artificial Intelligence 6, 1975, pp 293-326.

[18] D. Kopec and I. Bratko, *A Test for Comparison of Human and Computer Performance in Chess*, Advances in Computer Chess 3, (Clarke, M.R.B.,ed.), Pergamon Press, 1982, pp 31-56.

[19] D. Kopec, M. Newborn, and W. Yu, *Experiments in Chess Cognition*, Advances in Computer Chess 4, D. Beal (ed.), Pergamon Press, Oxford, 1982, pp 45-54.

[20] R.E. Korf, *Real-Time Heuristic Search: New Results*, Automated Reasoning.

[21] D. Levy and M. Newborn, *How Computers Play Chess*, W.H. Freeman & co. (ed.), Computer Science Press, New York, 1991.

[22] T.A. Marsland and M. Campbell, *A Survey of Enhancements to the Alpha-Beta Algorithm*, ACM81 National Conf. Proc. (Los Angeles, Calif., Nov.1981), ACM, New York, pp 109-114.

[23] T.A. Marsland and M. Campbell, *Parallel Search of Strongly Ordered Game Trees*, Computing Surveys, Vol. 14, No. 4, 1982, pp 533-551.

[24] T.A. Marsland and F. Popowich, *Parallel Game-Tree Search*, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. PAMI-7, No. 4, 1985, pp 442-452.

[25] T.A. Marsland, *A Review of Game-Tree Pruning*, ICCA Journal, March 1986, pp 3-19.

[26] T.A. Marsland, *Workshop Report on Theory and Practice in Computer Chess*, Reprinted from ICCA Journal, Vol. 10, 1987, pp 205.

[27] H.L. Nelson, *Hash Tables in CRAY BLITZ*, ICCA Journal, Vol. 8, No. 1, 1985, pp 3-13.

[28] M. Newborn, *The Efficiency of the Alpha-Beta Search on Trees with Branch-Dependent Terminal Node Scores*, Artificial Intelligence 8, 1977, pp 137-153.

[29] M. Newborn, *Recent Progress in Computer Chess*, Advances in Computers, vol. 18, M.C. Yovits (ed.), Academic Press, New York, 1979, pp 59-114.

[30] M. Newborn, *A Parallel Search Chess Program*, ACM Annual Conference, 1985, pp 272-277.

[31] M. Newborn, *Unsynchronized Iteratively Deepening Parallel Alpha-Beta Search*, IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol. 10, No. 5, 1988, pp 687-694.

[32] M. Newborn, *Computer Chess: Ten Years of Significant Progress*, Advances in Computers, Vol. 29, 1989, pp 197-250.

[33] A. Newell, J.C. Shaw, and H.A. Simon, *Chess Playing Programs and the Problem of Complexity*, IBM J. Research & Development 4 (2), 1958, pp 320-335.

[34] A. Reinefeld, J. Schaeffer, and T. Marsland, *Information Acquisition in Minimal Window Search*, Proc. 9th Int. Conf. on A.I., 1985, pp 1040-1043.

[35] J. Schaeffer, *The History Heuristic*, ICCA Journal, August 1983, pp 16-19.

[36] J. Schaeffer, *The History Heuristic and Alpha-Beta Enhancements in Practice*, unpublished manuscript, March 1988, University of Alberta.

[37] T. Scherzer, Private Communication, 1985.

[38] J.J. Scott, *A Chess-Playing Program*, Machine Intelligence 4, B. Meltzer and D. Michie (ed.), Univ. of Edinburgh Press, Edinburgh, 1969, pp 255-266.

[39] R. Seidel, *Chess, How to Understand the Exceptions!*, ICCA Journal, March 1985, pp 14-16.

[40] C.E. Shannon, *Programming a Computer for Playing Chess*, Philosophical Magazine 41, 1950, pp 256-275.

[41] J.R. Slagle and J.K. Dixon, *Experiments with some Programs that Search Game Trees*, J. Assoc. Comput. Mach. 16, 1969, pp 189-207.

[42] D.J. Slate and L.R. Atkin, *Chess 4.5 - The Northwestern University Chess Program*, Chess Skill in Man and Machine, P.W. Frey (ed.), Springer-Verlag, New York, 1977, pp 82-118.

[43] D.J. Slate, *A Chess Program that Uses its Transposition Table to Learn from Experience*, ICCA Journal, June 1987, pp 59-70.

[44] K. Thompson, Private Communications, Oct.-Nov. 1981.

[45] A.L. Zobrist, *A Hashing Method with Applications for Game Playing*, Tech. Rep. 88, Computer Sciences Dept., University of Wisconsin, Madison, 1970.