An Exchange Evaluator for Computer Chess

Dan and Kathe Spracklen
10832 Macouba Pl
San Diego CA 92124

Three main tasks are basic to computer chess: generation of moves, evaluation of positions and selection between alternatives. Of these three, the central determining factor in the strength of the program relative to the capacity of the host machine is the *evaluation* segment. The reason for this is that any program must come to grips with the task of move generation, and various techniques of "pruning" decision trees are by now widely known. Furthermore, the smaller and slower the host machine, the more importance must be assigned to the evaluation facility. If a search can be carried to a great depth of *ply*, inaccuracies can generally be corrected long before the machine has been committed to a costly line of play. (A ply is a move by one player, ie: half of a complete move involving both players.) On the other hand, if processing limitations prevent a critical exchange from being examined to its conclusion, then not just accuracy but clairvoyance is demanded. Thus an attack evaluator assumes tremendous importance in a microcomputer chess program, much more so than in a large scale machine. But the limitations placed on the programmer of an 8 bit machine make it correspondingly more difficult to achieve this type of predictive power. The ability of Sargon (a chess playing program we wrote in Z-80 assembler language) to accurately forecast the outcome of an exchange has been the greatest single factor in its success.

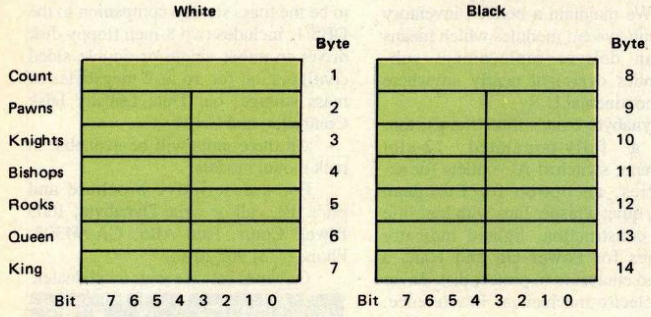


Table 1: Format of the attacker's array, a 14 byte array divided into two sections, seven bytes for White and seven bytes for Black. The first byte of each section contains the number of attackers (or defenders) in the array. The other six bytes contain the values of the pieces participating in the attack under analysis. Since no more than four bits are required per piece, two pieces are stored per byte and the array has a fixed format. The routine that fills the array assigns the first attacker of a given type to the low order four bits of the byte. A subsequent attacker of the same type is added by shifting up the low order four bits and inserting the new attacker.

Some Tactical Considerations

First, consider the capabilities desired of the routine. Assume that the computer is faced with evaluating the board position in figure 1. Black possesses a dangerous passed pawn that White has blockaded with a Knight. White is piling up attackers on the pawn and presently assaults it with King, Queen, and from behind the Queen, a Bishop: a total of three attackers. Black defends with Queen, Rook, and Knight; but the Black Knight is pinned against the Black King by White's Bishop, so Black really only has two usable defenders. Does this mean the pawn is lost? No, consider the order in which the exchange would occur. The King cannot legally capture first and the Bishop is behind the Queen, so the Queen must be the first taker. When Black responds with Rook takes Queen, Black has gained considerable material and is under no obligation to go any further with the exchange. To summarize the subtleties involved, the program must recognize transparent attacks through its own pieces which move in the same direction. It must recognize pins (and partial pins such as a Rook pinned along a rank or file). It must understand the relative values of attacking and defending

pieces, and, finally, it must realize that the exchange may be terminated at any point by either side. Pins are a whole topic in themselves, and Sargon's pinned piece routines will not be discussed in any detail. Instead, we shall concentrate on the exchange routine itself, which weighs the relative merits of the battles engaged on the board.

The Data Structures

The basic data structure used by the exchange evaluator is the attackers array. It is a 14 byte area divided into two sections, seven bytes for White and seven for Black. The first byte of each section contains the number of attackers (or defenders) contained in the array. The other six bytes in each section store the piece values of the pieces participating in the attack. Since no more than four bits are required, two pieces are stored per byte, and the array has a fixed format. Table 1 illustrates the arrangement within a section. The routine which fills the array assigns the first attacker of a given type to the low order four bits of the byte. A subsequent attacker of the same type is added by shifting up the low order four bits and inserting the new attacker in its place. The instruction used to implement this is the rotate left digit (RLD) (see figure 2). If a piece attacks from behind the Queen, such as the Bishop in figure 1, it is placed in the high order four bits of the Queen byte. From that position it will not come into play in the attack until after the Queen has captured. It is possible for two Rooks to attack through the Queen. In this situation one Rook is stored behind the Queen and the other in the King byte, pushing him up behind the Rook if he is involved in the attack. (By the rules of chess, the King cannot capture unless all defenders are exhausted, so he is properly placed behind the Rook.)

A note about overflows: the table is necessarily limited in size and is adequate for all the pieces originally on the board. If pawn promotions result in multiple pieces and a table overflow occurs, the excess pieces are ignored in evaluating the exchange.

An Overview of the Exchange Evaluator

The exchange evaluator (XCHNG) operates on a prefilled attacker's array. The array itself is filled by the *attack save* (ATKSAV) routine as attackers are discovered by the *attacker's* routine (ATTACK). The latter two routines are important, and recent changes to them have resulted in a significant improvement in the performance of Sargon, but they are not discussed in this article. The attacker's array describes a specific battle over a given occupied square. The player who occupies the square is the defender and the player with the opposite color is the attacker. The attacker's section is examined for the lowest valued attacker. That piece is compared in value to the piece on the occupied square. If the attacker is lower in value than the defended piece, we know at once that we can win material by capturing that piece. We don't yet know how much, because the piece may have been totally undefended, or it may be that our lower value piece will be captured in return. For example if our Bishop attacks an enemy Rook, we can be sure at least of "winning the exchange" (a phrase chess buffs use to describe trading a Rook for a minor piece, ie: for a Bishop or Knight). But to find out whether the whole Rook is ours for free or if we must give up our Bishop in return, we must toggle the attacker/defender roles, since our Bishop now occupies the square, and run through the analysis again. Of course back when the Bishop was retrieved from the attacker's array, it was also removed, the attack count decremented, and its position filled with zeroes.

The evaluation is not so obvious when the attacker is of higher value than the piece on the occupied square. In this case there are only two situations in which you would want to capture. One occurs when the attacked piece is totally undefended, and the

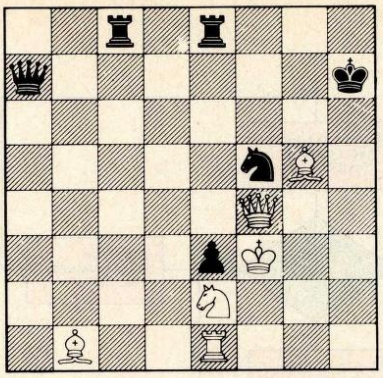


Figure 1: Sample board position. White's Bishop is indirectly attacking the pawn, so the value of the Bishop is stored in the high order four bits of the Queen byte (which is directly attacking the pawn) in the attacker's array. See table 1.

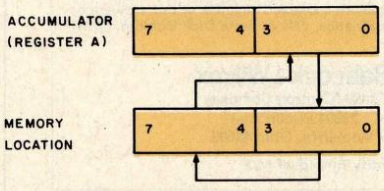


Figure 2: The Rotating Left Digit (RLD) instruction, used to add attackers to the attacker's array.

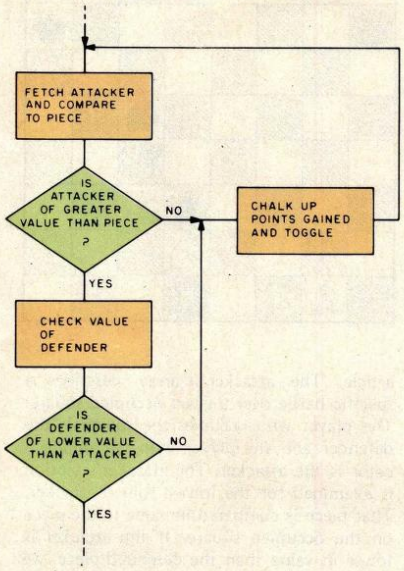


Figure 3: Summary of the flow of exchange evaluation. If the attacker is of the same value as the piece under potential attack, material cannot be lost by swapping, and the piece may in fact be taken for free. To determine the potential for winning material, assume the capture takes place, switch (or "toggle") the roles of defender and attacker and run through the analysis again.

other occurs when the attacked piece is defended by a piece of the same or higher value, and we can back up the attack with yet another attacker. Suppose, for example, our Queen attacks an enemy pawn. If the pawn is completely unguarded, we can, of course, take it for free. We might also want to take it if it is defended by the opponent's Queen and we can recapture with, say, a Bishop which attacks from behind our Queen. But any time the attacked piece is defended by a piece of lower value than the attacker, we can terminate the exchange right there, since it would not be to the advantage of the attacker to continue. For example, if our Queen attacked a pawn that was defended by an enemy pawn, we wouldn't consider making the capture.

If the attacker is of the same value as the piece on the occupied square, we know we can't lose material by swapping, and the piece might be ours for free. So to find out what we stand to gain, we assume the capture takes place, switch (or "toggle") the attacker/defender roles, and run through the analysis again (see the summary in figure 3).

Quantizing the Evaluation

We now have a general plan for the flow of the evaluator. What is needed is a means of quantizing the results and coming up with a points total, the exchange residue, which accurately describes that particular battle. The exchange residue is zero at the onset of the analysis and will be adjusted up or down as the evaluation proceeds. At each iteration the number of points at stake is the value of the piece which currently occupies the square in question. If the analysis calls for a capture on the first iteration, the points at stake are added to the exchange residue. Thus the exchange residue will contain the number of points lost by the initial defender (or, conversely, won by the initial attacker). We will maintain this frame of reference throughout the evaluation. If the analysis requires that attacker/defender roles be toggled, and a capture occurs on the second iteration, the points at stake would be subtracted from the exchange residue. Suppose we again have a situation where our Bishop attacks an enemy Rook. The points at stake are the assumed value of the Rook, and let's suppose we value the Rook at five points. We know the analysis will call for Bishop takes Rook, so at that time the five points for the Rook will be added to the initially zero exchange residue. Then

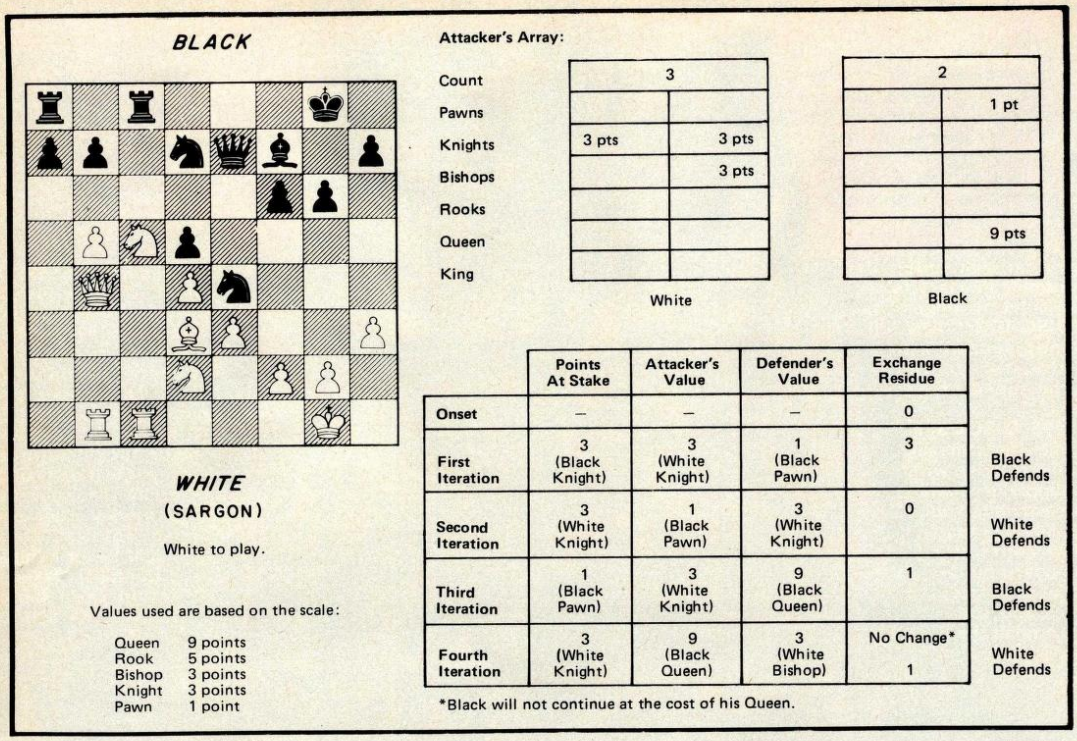


Figure 4: Analysis of a typical chess battle, in this case at the K4 square, taken from the game of Keres versus Najdorf, International Tournament at Margate, 1939. The associated chart shows how the points at stake, attacker's value, defender's value and exchange residue are altered at each successive iteration.

the attacker/defender roles are toggled, and if our Bishop, worth say three points, is recaptured, those three points would be subtracted from the exchange residue leaving a current residue of two points. If the battle continues, on the third iteration the points are again added, and on the fourth subtracted, etc. Figure 4 gives a typical battle and the associated chart shows how the points at stake, attacker's value, defender's value and exchange residue are altered at each successive iteration.

A note on the bounds of the exchange residue is pertinent here. The exchange residue will always be a positive number. This is clearly so, since for it to go negative the attacker would have to engage in an unsound exchange, such as the Queen capturing a pawn defended by another pawn as in a previous example. Such an exchange would be a blunder. We will assume that this won't occur on the part of our opponent, and we will eliminate it from our moves. The exchange residue will also have as a maximum the number of points at stake initially, since the defender will not make a move that will cost more than has already been lost. Thus, $0 \leqslant$ exchange residue $\leqslant$ value of attacked piece.

Programming the Evaluator

Great care is necessary in coding the routine, since it must be executed once for every attacked piece on the board. If we assume that an average of five pieces will be under attack at a time, this means the routine will be executed five times for every board evaluated. Since typically 5,000 to 12,000 board positions will be evaluated by the most recent version of Sargon using a 4 ply search, this means the exchange evaluator may be executed up to 60,000 times in determining a single move. So an inefficiency in execution time as slight as needlessly pushing and popping four registers would be magnified to a total cost of three seconds (assuming a 2 MHz clock) in the time required to process a single move. For this reason chess programmers must quickly become familiar with the relative execution times of their machine's instructions. If the exchange evaluator seems obscure, the

AF	(Various Uses)	Program Status Flags
BC	Defender Count	Attacker Count
DE	Attacker Section Address	
HL	Defender Section Address	
IX	Index to Value Array	
IY		

Table 2: Map of the registers used by the exchange evaluator.

AF'	Defender	Program Status Flags
BC'	Attacked Piece Value	Flag to Defender Side
DE'		Exchange Residue
HL'		Attacker Value

Table 3: Map of the BC, DE, and HL registers used by the attacker/defender routines.

BC	Attacker Count	Defender Count
DE	Defender Section Address	
HL	Attacker Section Address	

blame lies in just such considerations.

Since nearly every register in the Z-80 processor is utilized in the routine, a map is provided for reference in the discussion (see table 2). Although Sargon is coded in Z-80 assembler language using TDL mnemonics, no prior knowledge of the specific instructions is assumed. However, it is assumed that the reader is familiar with some microprocessor assembly language. Two routines are described: XCHNG, which performs the actual evaluation, and NEXTAD, which searches the attacker's array for the next attacker or defender.

Using the Exchange Residue

The exchange evaluator has completed its work once it has returned the outcome of the battle. But the evaluation segment is by no means complete. Information gleaned by analyzing attacks must be blended with data concerning piece mobility, development, total material and any other heuristics included in the program. The total picture is the responsibility of a routine called POINTS, which is not discussed here. But it is useful to see how POINTS makes use of the information returned by XCHNG.

The exchange evaluator must be called to examine every potential battle on the

Listing 1: The Sargon exchange evaluator, written in Z-80 assembler language with TDL mnemonics.

```
Label    Op Code    Operand        Commentary

XCHNG:   LDA        P1             Fetch the attacked piece into register A.
                                   The piece includes a color flag in bit 7
                                   (0 for White, 1 for Black) and the piece
                                   type in bits 2-0.
         LXI        H,WACT         Load into the HL and DE register pairs.
         LXI        D,BACT         The beginning addresses of the White and
                                   Black sections of the attackers array.
         BIT        7,A            Test the color flag bit of the piece and
         JRZ        XC5            skip the XCHG if the piece is White.
         XCHG                      Otherwise swap the contents of the HL and
                                   DE registers. The result is to produce a
                                   pointer to the defender's section of the
                                   attackers array in the HL register pair and
                                   a pointer to the attacker's section in the
                                   DE pair.
XC5:     MOV        B,M            Fetches the byte pointed to by the HL pair
         LDAX       D              into the B register. Fetches the byte pointed
         MOV        C,A            to by the DE pair into the A register, then
                                   moves it into the C register. Since the first
                                   byte if each section of the attacker's array is
                                   the count (see table 1), we now have the
                                   total number of defenders in register B and
                                   attackers in C.
         EXX                       Swap registers BC, DE, and HL for registers
                                   BC', DE', and HL', ready to initialize the
                                   rest of the data used by the exchange
                                   evaluator.
         MVI        C,0            Register C contains a flag which tells when
                                   the attacker/defender roles have been
                                   toggled. Each time the roles are reversed,
                                   register C is incremented. Then by examining
                                   bit 0 of C, we can tell which side is being
                                   examined. A value of 0 indicates the at-
                                   tacker's side is under consideration, and a
                                   value of 1 the defender's side.
         MVI        E,0            Initialize the exchange residue.
         LIXD       T3             T3 is an index by piece type into an array,
         MOV        B,PVALUE(X)    called PVALUE, which contains the point
                                   value (the worth) of each type of piece.
```

Note: A documented source listing of the entire Sargon program is available for $15 from Dan and Kathe Spracklen, 10832 Macouba Pl, San Diego CA 92124.

```
                                    The index is loaded into the IX index
                                    register and then the value of the piece un-
                                    der attack is loaded into the B register. So
                                    register B contains the number of points at
                                    stake in this attack.
        CALL    NEXTAD              Getting the value of the next attacker in
                                    register A. NEXTAD also sets the zero flag
                                    if there are no more attackers.
        RZ                          Return if no more attackers.
XC10:   MOV     L,A                 Save the attackers value in the L register.
        CALL    NEXTAD              Getting the value of the next defender in
                                    register A, and setting the zero flag if no
                                    more defenders.
        JRZ     XC18                If no defender, the piece is lost. Go chalk up
                                    points gained.
        EXAF                        Save the defender by swapping AF and AF'
                                    registers.
        MOV     A,B                 Move the value of the attacked piece into
        CMP     L                   the A register to then compare its value to
        JRNC    XC19                that of the attacker. Branch to XC19 if the
                                    value of the attacker is not greater than the
                                    value of the piece, to chalk up points gained
                                    and toggle.
        EXAF                        To reach this point, the attacker must be
                                    worth more than the piece under attack. So
                                    it is necessary to consider the value of the
                                    defender. This instruction swaps A and A'
                                    again to restore the value.
XC15:   CMP     L                   Compare the value of the defender to the
        RC                          attacker. If the defender is worth less, return.
                                    It will not be to the attacker's advantage to
                                    continue the exchange.
        CALL    NEXTAD              Otherwise get the value of the next attacker.
        RZ                          Return if none. If the defender is worth the
        MOV     L,A                 same or more than the attacker, the ex-
                                    change should continue, provided there is
                                    another attacker available to recapture. Save
                                    the new attacker's value in the L register.
        CALL    NEXTAD              Then find out if there are any more de-
        JRNZ    XC15                fenders to contend with. If so, jump back to
                                    XC15 and repeat the process.
XC18:   EXAF                        The exchange is terminated. There are no
                                    more defenders. The zero flag is set, so save
                                    it by swapping AF and AF'.
        MOV     A,B                 Get the value of the attacked piece.
XC19:   BIT     0,C                 Test for attacker's or defender's side.
        JRZ     XC20                Skip if on the attacker's side. Otherwise
        NEG                         negate the value of the attacked piece. (On
                                    successive iterations the value is alternately
                                    added and subtracted.)
XC20:   ADD     E                   Add the previous exchange residue to the
        MOV     E,A                 new points won or lost and store the result
                                    as the new exchange residue.
        EXAF                        Restore the last defender and the zero flag.
        RZ                          Return if there are no more defenders.
        MOV     B,L                 The last attacker becomes the new defender.
        JMP     XC10                Move his value into the B register and return
                                    to XC10 for another iteration.
NEXTAD: INR     C                   Increment side flag.
        EXX                         Swap registers BC, DE, HL for BC', DE',
                                    HL', getting the set that contains the at-
                                    tacker and defender counts.
        MOV     A,B                 Swap attacker and defender counts.
        MOV     B,C
        MOV     C,A
        XCHG                        Swap attacker's array pointers. The register
                                    map is now as in table 3.
        XRA                         Zero the A register and compare it to the
        CMP     B                   attacker count. Go return if there are
        JRZ     NX6                 none.
        DCR     B                   Otherwise decrement the count, since one
                                    will be removed from the array.
NX5:    INX     H                   Check the next byte of the attacker's array,
        CMP     M                   looking for an attacker.
        JRZ     NX5                 If not in this byte, go check the next.
        RRD                         Otherwise rotate the attacker into the A
                                    register. The rotate right digit (RRD) is the
                                    reverse of the rotate left digit illustrated
                                    in figure 2.
        DCX     H                   Decrement HL to back up the pointer. With
                                    two attackers stored per byte, the routine
                                    will return to the same byte to look for the
                                    next one.
NX6:    EXX                         Swap registers BC, DE, HL and BC', DE'
                                    and HL' back again.
        RET                         Return.
```

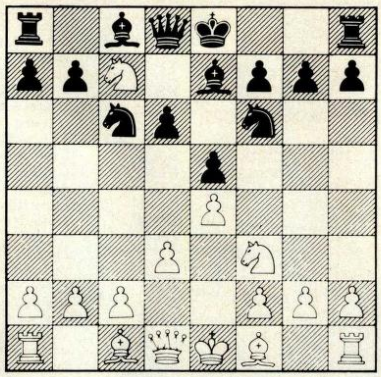
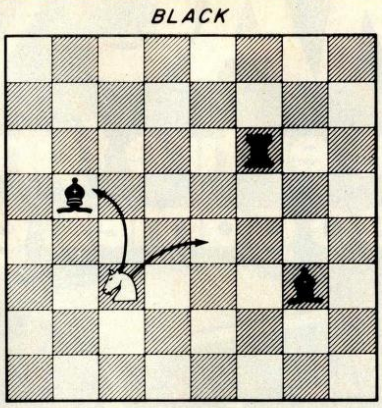


Figure 5: Potential problem arising from the author's evaluation scheme: White's Knight is attacking the Black King and Rook, for which White gains 3/4 of the Rook's value. The Knight is doomed to be captured by Black's Queen, but subtracting the Knight's value from this number still gives White an illusion of material gain. The authors avoided this problem by having the program check to see if the piece that has just moved is subject to capture.

board for a given position. In some of the attacks, the side which has completed a move will have lost points. In others the side about to move will be in danger of losing material. As battles are evaluated one by one, the highest points lost for the side having moved is maintained. This value represents the amount of material this side stands to lose, and it is subtracted directly from the material score. Two scores are maintained for the side about to move: the highest points lost, and the second highest points lost. Both values are saved, because it is assumed that this side will always use its move to rescue its highest value piece. Then only 3/4 of the value is deducted for the loss of the second highest piece, since deducting the entire value would make attacks look as good as captures (see the text box for an example of this procedure). Bonus points are given to each side for additional battles won, but this is still experimental and may not be needed.

One problem that arose with this evaluation scheme was the Knight's tendency to engage in useless forks. In figure 5 we see White's Knight attacking the enemy King and Rook, for which White gains 3/4 of the Rook's value. The Knight is of course doomed to be captured by the Queen and



Making an Immediate Capture More Attractive Than an Attack

In the diagram, the program (White) has two attractive moves: capture the Bishop, or move the Knight so that it simultaneously attacks the Black Rook and the second Bishop. The program assumes that in the latter case Black will protect the Rook (the more valuable of the two pieces) by moving it away. The decision then reduces to one of capturing the first Bishop or simply attacking the second Bishop. In order to insure that the capture takes place, the program assigns 3/4 the normal value to the second Bishop so that the capture looks more attractive. The drawback to this technique is that it precludes the possibility of intentionally avoiding an immediate capture for strategic purposes; this would require a much more complicated program, of course.

can never carry out its threat, but subtracting the Knight's value still gives White an illusion of material gain. Sargon avoids this problem by checking to see if the piece that has just moved is subject to capture. If so, we assume that the side about to move can escape both attacks. The attack with the highest points lost is ignored completely and the attack with the second highest points lost is moved up in its place.

Current Limitations and Future Developments

The problem of the Knight fork as just discussed is only one of a whole set of difficulties. Pinned pieces, the overworked piece, discovered attacks and other motifs can all occur dynamically during play of a board position, but are difficult to evaluate statically. Attacks of this nature are second order attacks and are not considered in the exchange evaluator we described. There are eight possible second order attacks (see table 4). The first group are the discovered attacks and the second group are the transparent attacks. If all of the second order attacks could be taken into account, the evaluation would be much improved. Currently work is being done to accomplish this. Ultimately, of course, the entire board should be considered as a single complex battle. How close to this ideal can static evaluation progress? At what point does static evaluation begin to take more time than the look-ahead itself? Where will compromises in the evaluation be least harmful? Currently in the field of computer chess there is a tendency to downplay the importance of look-ahead in future developments. Has look-ahead reached a dead end? Will it be replaced by a Sargon-like exchange analysis? These are open questions.■

Table 4: Second order attacks. This type of attack, including pinned pieces, overworked pieces, discovered attacks, and so on, is not considered in the exchange evaluator described in this article.

Group	Type	Description
Discovered Attacks	W1 → B1 ⌒ W2	W1 attacks B1. If B1 moves, W1 defends W2.
	W1 → W2 ⌒ W3	W1 defends W2. If W2 moves, W1 defends W3.
	W1 → B1 ⌒ B2	W1 attacks B1. If B1 moves, W1 attacks B2. (PIN)
	W1 → W2 ⌒ B1	W1 defends W2. If W2 moves, W1 attacks B1.
Transparencies	W1 → B1 → W2	W1 attacks B1. B1 attacks W2. W1 defends W2 through B1.
	W1 → W2 → W3	W1 defends W2. W2 defends W3. W1 defends W3 through W2.
	W1 → B1 → B2	W1 attacks B1. B1 defends B2. W1 attacks B2 through B1. (PIN)
	W1 → W2 → B1	W1 defends W2. W2 attacks B1. W1 attacks B1 through W2.