A Generalised Quiescence Search Algorithm*

Don F. Beal
Department of Computer Science, Queen Mary College, London University, Mile End Road, London, England E1 4NS, UK

ABSTRACT

This paper describes how the concept of a null move may be used to define a generalised quiescence search applicable to any minimax problem. Experimental results in the domain of chess tactics show major gains in cost effectiveness over full-width searches, and it is suggested that null-move quiescence may be almost as widely useful as the alpha-beta mechanism. The essence of the mechanism is that null moves give rise to bounds *on position values which are more reliable than evaluations. When opposing bounds touch, they create a single value which is more reliable than ordinary evaluations, and the search is terminated at that point. These terminations are prior to any alpha-beta cutoffs, and can lead to self-terminating searches.*

1. The Idea of the Null Move

The null move means changing who is to move without making any other change to the game state. Some games, such as Go, allow the null move as a legal move; others such as chess, do not. In the overwhelming majority of positions in either game, the null move would be a poor move, because there would be moves that did something beneficial for the side to play. This is true even if the side to play is losing, because even then the losing side is expected to be trying to minimise the loss, or delay it. Situations where the best possible outcome is obtained by the null move are really very rare. In chess they occur sometimes in the late stages of the game and, although they comprise a negligible fraction of positions encountered in practice, they have a special name, *zugzwang*. In a zugzwang position, every piece is either blocked or "tied down" by some essential defensive function and every move loses hold of something. In Go, the null move is never useful during the main part of the game, but as the game reaches the very end, both players are approaching

*An earlier version of this paper appeared as "Experiments with the Null Move," in *Advances in Computer Chess* 5 (Elsevier Science Publishers, Amsterdam, 1989).

Artificial Intelligence **43** (1990) 85–98
0004-3702/90/$3.50

situations where they would lose rather than gain by playing another stone. When a player thinks that there is no further gain for him, that player will "pass" (= null move) and when both players pass consecutively, the game is over.

Since the null move is so rarely a good move (and not even legal in chess), why should it be included in minimax lookahead?

The answer lies in the structural properties of the computation that minimax lookahead is making, and has little to do with whether the null move is legal or not. It is also independent of the alpha-beta algorithm used to speed up minimax.

2. What Computation Is Minimax Making?

Since CHESS 4.5 (Slate and Atkin [18]), most chess programs, particularly the top performing ones, have used a search regime roughly characterized by: full-width search to a fixed depth, followed by a quiescence search. A typical quiescence search would be: captures and checks at ply 1, captures only after that.

Although the search is described as full-width, it is taken for granted that alpha-beta will be used. Essentially, the description above defines a tree to be searched, within which an alpha-beta search will look at as many moves as necessary. It is well-known that an alpha-beta search will give the same result as the vastly less efficient look-at-all-moves minimax.

The *computation* performed at each move is that of choosing a best move based on the tree to be minimaxed (a subset of the complete game tree), and the evaluation function values at all the tree's endpoints. The factor being singled out for attention here is the tree to be searched, i.e., its shape and size.

Clearly, CHESS 4.5-style programs perform a different computation if the depth changes, or if changes are made in the rules governing which moves comprise the main search or quiescence trees. The key questions would be "Is the computation any better?" and "How long does it take now?". In chess, except for certain relatively simple endgames, we have never been interested in any specific computation for move choice, but in getting "value for money," that is, quality of move choice for time spent.

The points of the last paragraph are stressed because, unlike alpha-beta, which gives dramatically reduced computation times for the *same* computation, the null-move algorithms to be described produce a dramatic reduction in computation time, but on a *different* computation. This makes it harder to judge the results.

In any case, we reached the most important question of all: "What shape and size should the search tree be or, in other words, which moves should be considered for searching?"

Note that alpha-beta should not be mentioned in the answer. Although alpha-beta indicates moves not worth searching, the context is different. Here the question is being asked of the tree *within which* alpha-beta is to operate.

The question is, of course, very old, perhaps the oldest in the 40-year history of computer chess. It is the question of selective search.

3. Attempts at Selective Search

Shannon's legendary paper of 1950 [17], "Programming a Computer to Play Chess," described how minimax search should be of variable depth, only stopping at quiescent positions, and rejecting some moves at interior nodes of the tree. He called such strategies type-B, and suggested a function similar in spirit to CHESS 4.5-style quiescence search rules for stopping, but didn't suggest an actual function for rejecting moves at higher nodes.

Almost all the early chess programmers did indeed try to restrict the search to a subtree of moves that include "important" moves and exclude "irrelevant" moves. The Bernstein program of around 1957 [8], perhaps the most complete and effective of the very early chess programs, limited the search tree to the "best 7" at every node, using a sequence of plausible move generators to produce the "best 7." Ten years later, the Greenblatt program [10] became the best program of its day using carefully chosen quiescence rules aiming at tactical solidity.

Then, in 1973, CHESS 4.0 (later transmuted into CHESS 4.5 [18]) made the successful exchange of speed and efficiency for selection, and set the pattern for the next fifteen years. There was however, a vitally important qualification to the new pattern of do-it-all-to-fixed-depth-and-do-it-fast. Namely, the quiescence search. The endpoints of the fixed depth search were not places to apply the evaluation function, but places to start operating a simple and successful selective search, namely, the capture tree (augmented by one or two plies of checks if available). (There were also, of course, the vital mechanisms of iterative deepening, transposition tables, alpha-beta by then taken for granted, and many ingenious programming techniques.)

There have been much more radical attempts to find better selective search mechanisms. Harris (1974, [11]) proposed a bandwidth search centered on promising lines of play. The authors of the Russian program KAISSA (Adelson-Velskiy, Arlazarov and Donskoy, 1975, [1]), mention a variety of heuristics, including allowing the play of a null move when ahead in material. Berliner (1979, [7]) proposed the B* algorithm which uses separate optimistic and pessimistic estimates of position value at each node. Palay (1983, [15]) extended and developed B* to use probability distributions rather than optimistic–pessimistic ranges. Very recently, McAllester has described an elegant method, called conspiracy numbers, based on counting critical positions (1986, [14]).

In addition, there have been many attempts to use extensive domain-specific knowledge to create effective selective searches. Two major attempts that focussed on tactical play were Berliner's CAPS-II [6] and Wilkins' PARADISE [20].

However, none of these methods has proved effective enough to become widespread.

4. Quiescence Search

Quiescence searches are, of course, selective searches. They derive from the idea of expanding the search just enough, and only just enough, to avoid evaluating a position where tactical disruption is in progress.

Chess players can identify many types of "tactical disruption." The simplest notion, which leads to the idea of a capture tree, is to say that tactical disruption is present if an immediate capture is available. More sophisticated definitions would take account of checks, forks, trapped pieces, etc., and lead to more elaborate (and larger!) quiescence searches.

It is tempting, but not quite accurate, to think that a quiescence search could be defined by giving the rules for selecting the moves that make up the quiescence search tree. (If the rules produced no moves at a particular position the position would be a terminal node of the tree. Alpha-beta would be used within the tree.) This description sounds complete if rather condensed, but it is not quite right. To see what is missing, consider the position of Fig. 1.

The position in Fig. 1 is not terminal. White has a capture available. However, it is a disadvantageous capture and White ends up losing two pawns worth of material. This value (−2 pawns) is the value returned by a "quiescence search" according to the definition just given.

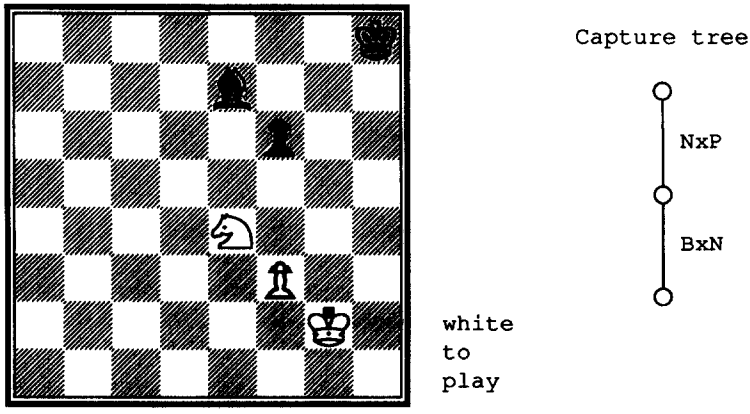


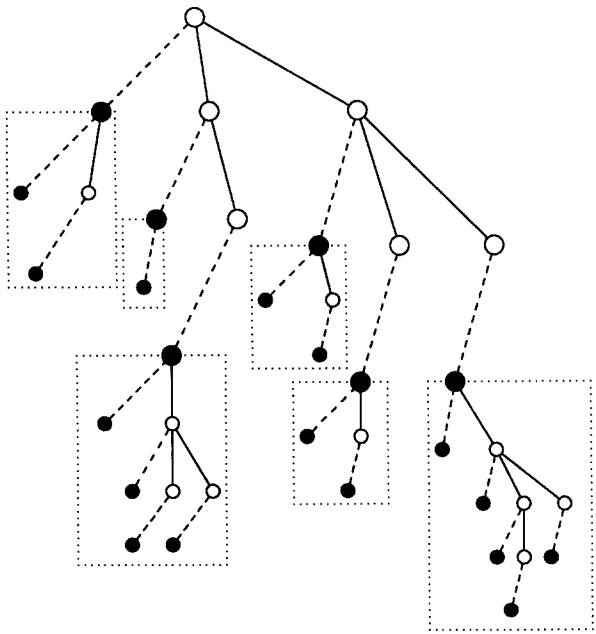
Fig. 1. Position illustrating capture quiescence.

Clearly, this is wrong. White need not enter into the capture. The correct definition of a quiescence capture tree includes: "at each node, the side to play is given the option of choosing the best capture *or* taking the static evaluation." White has the option to "stand pat."

This option to "stand pat" can equally well be viewed as an option to choose the evaluation after a null move, rather than the static evaluation now, since the material doesn't change with a null move. Although involving the null move seems an unnecessary complication at first sight, regarding the value as arising from playing the null move enables capture search to be seen as merely one special case of a general algorithm.

5. Game Independence

An interesting property of the concept of a capture tree is that definitions can be given in a game-independent way. By game-independent is meant that all game-specific knowledge is packaged in an evaluation function (in this case statically counting up the value of pieces on the board), and then game-independent rules define the search regime. An earlier paper [2] reported that the game-independent rules for "consistency search" produced a capture tree from the "material balance" evaluation in chess.



Dark circles are terminal nodes of quiescence searches.
The second-order quiescence search tree is highlighted by larger nodes.

Fig. 2. Illustrative shape of search tree for second-order null-move quiescence.

However, there is another game-independent search regime, null-move quiescence search, that also produces a capture tree when given the material balance evaluation. Null-move quiescence search is the subject of this paper.

Null-move quiescence becomes particularly interesting when it is "bootstrapped." Because it can be applied to *any* evaluation function, it can be applied as a second-order search to the values obtained by the first-order quiescence searches. The idea here is that just as "counting material" can be packaged in a black box and called an evaluation function, so a result obtained by a null-move search can be packaged in a black box and called an evaluation function. Thus, the rules can be applied a second time at a higher level. Figure 2 illustrates the effect diagrammatically.

Applying this to material balance in chess, the second-level quiescence search, although it still contains no chess knowledge beyond counting up material, selects moves such as checks, moves out of check, attacks on pieces, defences, blocks, and other types of sharp tactical moves that chess players concentrate on. Perhaps the surprising thing is that the branching factor of the second-level tree is only an average of about four (compared to an average of about 35 legal moves), and the success rate in finding combinations is very high indeed. (It does, however, have some distinctive weaknesses.)

6. The Null-Move Quiescence Search

It is easiest to describe as a program. Figure 3 shows a simplified version, called QUIESCE.

QUIESCE is a particularly simple version to illustrate the essential mechanism. *bestv* is initialised to a null-move value, rather than to the lower end of the alpha-beta range. Notice that *all* evaluations take place after the null move. QUIESCE does not have a depth limit. It can however still terminate. Whenever the null-move value is greater than or equal to the upper end of the alpha-beta range, the search will stop at that position. Only the null move will have been explored, and that is evaluated directly. QUIESCE can be, and often is, a self-limiting, self-terminating search.

Also note that the function *evaluate* could itself be a search. Thus to obtain a second-order null-move quiescence search, QUIESCE1 would be defined with

```
QUIESCE(lower, upper)  integer lower, upper;
{  integer bestv;
   makenull;  bestv ← – evaluate(–upper,–lower);  unmakenull;
   foreach move m do
   {  if( bestv >= upper )  return(bestv);
      make(m);  v ← – QUIESCE(–upper, –bestv);  unmake(m);
      if( v > bestv )  bestv ← v;
   }
   return(bestv);
}
```

Fig. 3. Simplest version of null-move quiescence search.

evaluate = material balance, and QUIESCE2 would be defined with *evaluate* = QUIESCE1.

The reader may have already noticed that QUIESCE, when using material balance as the evaluation function, will search all (or potentially all) moves at nonterminal nodes—not merely capture moves—despite the earlier claim that this quiescence mechanism would produce a capture tree. What will happen is that noncaptures will indeed be searched by QUIESCE, but will always terminate with a cutoff after the null move at the next ply. This single ply of search-and-reject behaviour for all noncaptures creates a 1-ply "fringe" around a capture tree skeleton. The fringe could be optimised away in an implementation specialised to a capture search. The essential requirement for optimising away the "fringe" for QUIESCE is that an incremental change to the evaluation function can be computed without actually making the move. Any evaluation function which meets this requirement could be similarly optimised.

A more practical version of null-move quiescence would use iterative deepening, transposition tables, ordering heuristics, etc., as used with full-width minimax searches.

The question of a depth limit raises more subtle issues that it might seem. If the search is truncated at a depth limit, the result should be not a single value, but a triple. This is explained later.

7. Performance of the Generalised Null-Move Quiescence Search

Null-move quiescence is a general mechanism, capable of operating with any evaluation function. However, it is particularly effective at solving tactical problems in chess. First-order quiescence on material scores (capture trees) is well-known to be cost-effective. The performance of second-order quiescence on material was tested on the 300 tactical positions in the Reinfeld book, *Win at Chess*, [16]. This test set has been used on other chess programs and algorithms.

One modification was made to the definition of second-order quiescence. That was to enhance the value returned from the first-order quiescence (i.e., the result from a capture tree) with the ability to see checkmates. This is an ad hoc modification, but is a cost-effective compromise between putting the knowledge about checkmate in with the material balance evaluation (where it might be thought to belong, but where it is very costly in program time) and leaving it out altogether.

With this modification, material double-quiescence solved 276 of the 300 positions. Only positions where the right move was found for the right reason, or the program demonstrated a flaw in Reinfeld's published solution, were counted as correct. This result can be compared with BELLE (Thompson [19]), which was reported by Palay [15] to only solve 273 within a 3-minute per position limit.

The significance of this result is not so much that the score happens to be higher than 1983 BELLE, but the "value for money" that it represents. The score itself *is* good considering the paucity of knowledge used, but the time taken is the real result. The results below show that BELLE's score can be obtained for approximately one-fifteenth of the effort.

The actual execution times (in z8000 microcomputer seconds) are given in Table 1. "F" entries indicate that no solution was found because the material-only evaluation function has insufficient knowledge to solve the problem. These may be compared with BELLE times by dividing by 100, as BELLE looks at approximately 100 times as many positions per second (about 160,000 versus about 1500 for the z8000 program). Of the solved positions, the longest took 16 seconds of BELLE-equivalent time. 273 positions (BELLE's score) can be obtained with a time limit of 12 BELLE-equivalent seconds per position, thus consuming only a fifteenth of BELLE's effort.

The comparison is on the basis of giving each program the full 180 (or 12) seconds on every position. Both programs need far less on most positions, as

Table 1
Times (seconds on z8000) to solution for the 300 Reinfeld positions. Positions 92 (= number 12 in test set 5), 157, 210, 249, 264 and 296 showed Reinfeld's solution to be incorrect and were scored as "correct by demonstrating flaw." Positions 116, 129, 130, 149, 204, and 223 produced Reinfeld's solution move but with lesser gain shown. These were also scored as "correct by demonstrating flaw"

	Test sets (called quizzes in Reinfeld)														
Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	3	42	F	1	7	F	2	57	31	1	1	29	213	124	35
2	F	F	2	29	20	1	5	4	19	7	1	53	8	1629	4
3	70	7	1	6	130	11	4	6	F	21	6	223	F	54	28
4	3	1	1	3	1	15	66	14	8	1	1205	17	32	70	315
5	1	1	1	23	1	30	18	22	15	153	91	2	789	1151	890
6	3	1	F	48	F	21	1	F	21	15	5	47	1	313	2
7	1	1	1	2	F	1	12	25	42	18	637	3	F	5	46
8	1	3	32	2	61	29	8	28	9	3	35	32	F	9	24
9	17	2	62	29	3	5	18	10	16	5	18	F	270	482	14
10	40	7	3	3	17	1	66	8	3	F	73	F	51	151	23
11	17	301	9	615	90	63	22	15	212	1	18	136	F	71	500
12	1	8	4	2	45	3	3	1	2	5	24	78	177	24	28
13	4	4	3	7	F	12	52	17	3	203	F	46	7	85	53
14	8	2	1	1	2	23	8	1	4	62	43	173	62	9	20
15	1	7	10	21	1	11	6	F	4	29	47	91	37	40	3
16	4	1	5	258	598	33	5	1	37	318	26	16	142	30	162
17	17	1	2	10	2	1	60	244	5	1	172	1241	45	32	F
18	34	1	27	43	7	32	11	3	192	20	5	35	20	47	5
19	4	1	6	17	2	10	40	5	4	10	2	F	F	29	F
20	1	1	1	107	F	37	5	1	730	32	776	649	114	886	23

Table 1 shows for the z8000 program (remember that 1200 z8000 seconds are needed for 12 BELLE-equivalent seconds). 235 positions were solved in less than 1 second of BELLE-equivalent time although, as times to solution are not given in [15], this cannot be compared with BELLE. The deepest search needed was for position 96 (= number 16 in test set 5), which went to 16 ply plus captures.

Apart from the see-checkmates enhancement, the experimental implementation differed from QUIESCE only in that special-purpose capture tree code was used for the first-order search, and that the second-order QUIESCE used iterative deepening with move ordering and already-calculated values obtained from the previous iteration. The program was written in assembly language, derived from a tournament chess program.

8. Comparisons with Other Programs

As a comparison with knowledge-based programs, PARADISE (Wilkins [20]) scored 89 out of the first 100 (compared to 92 for QUIESCE), and took about ten times as long. This is not really a meaningful comparison though, and Wilkins' work was actually a tour de force in the difficult research area of creating programs that use knowledge instead of search.

From private discussions, it seems that a few other programs have used the null move as an additional move in ordinary minimax. The idea is that in positions where the side to play is significantly ahead, the null move, although not the best, will nevertheless be enough to produce a cutoff, and, being a quieter move, may have a smaller subtree than the best move. This was the use mentioned in the KAISSA paper [1]. This technique may have some benefit in full-width searches, but it involves doing a normal depth search below the null move, rather than direct evaluation as in QUIESCE. Closer to QUIESCE was the program MERLIN (Kaindl [12]), which used the null move to find threats.

The work that is closest in spirit to QUIESCE is the B* algorithm of Berliner [7], and the experiments performed by Palay [15] on probabilistic modifications of B*. In B*, it is necessary to obtain upper and lower bounds for all positions in the tree. Berliner's paper on B* suggested that the bounds would come from evaluation knowledge. Palay experimented with a method which obtained bounds from a shallow (2-ply) search from the current position. The 2-ply search made two moves in succession for one side or the other, and thus provided biased results which were taken as bounds for each side. The main search would then proceed deeper from the current position, within the bounds found from the 2-ply search.

Conceptually, this has something in common with QUIESCE although Palay's algorithm is considerably more complicated. With an effort limit of 4 hours of VAX 11/780 time per position Palay's program solved 245 of the Reinfeld problems. (4 hours of VAX 11/780 time is very roughly equivalent to 3 minutes of BELLE time.)

9. Reasons Why Null-Move Quiescence Is Effective

Null-move quiescence takes a given evaluation function, applies a search regime to it, and returns a value from the search which is more reliable than the given evaluation function. As can be seen from an inspection of QUIESCE, the value returned always originates as a null-move value.

Why should null-move values be more reliable than ordinary values?

The answer is, "They aren't." It is their role as *bounds* that is more reliable. That is, the statement "the value at position P is *at least* X," where $X = evaluate(nullmove(P))$, is much more reliable than the statement "the value at P is X," or "the value at P is $evaluate(P)$." In general, as the search proceeds, tighter and tighter bounds can be acquired, until a lower bound meets an upper bound. At this point, the search closes off with a single *reliable* value.

Thus the value that QUIESCE returns is not "just" a null-move value, but is the result of two "opposing" null-move values touching. Figure 4 illustrates the smallest possible tree.

More typically, the top-level null-move value will not establish a closing bound. In this case, the depth-1 searches will not all consist of a single null-move evaluation, but some will expand to another ply of search.

A simple model of minimax [3] showed that the benefit of minimax lookahead in random trees conforming to an overall clustering tendency increased with the clustering, and that the degree of clustering in a chess endgame (KPK) was amply sufficient for beneficial (rather than pathological) lookahead behavior.

The concepts of that model can be used to consider what the benefits of null-move quiescence lookahead might be. The result, not presented in detail here, is that if the null-move value was assumed to take values unrelated to other local values, then the error propagation properties were very similar to

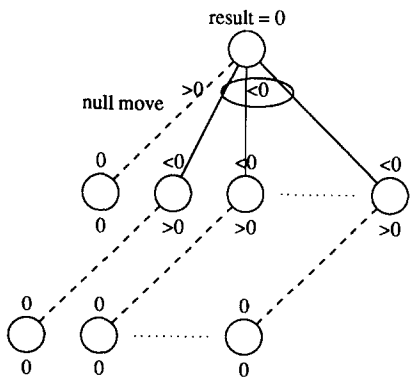


Fig. 4. Smallest possible tree for QUIESCE. *Negamax convention*: numbers below node are from the point of view of player making move below; numbers above are from the point of view of players making move above.

full-width lookahead. This would mean that the expected reliability of QUIESCE would be low, since the null-move values are raw evaluations, not deep searches, and can occur at any depth.

Perhaps it isn't surprising though, that null-move quiescence would be useless if the null-move value was unrelated to other local values.

If instead a positive assumption is made about the properties of the null move, namely, that it is highly unlikely for a null-move value to be higher than the best of the other moves, then the error-reduction factor changes from approximately

$$\frac{b}{1-k}\sqrt{k(k-f+kf)}$$

to approximately zero. In other words that, with that kind of model, a null-move quiescence search removes all first-order error terms.

Of course, this all depends on the assumption that the null-move value is not higher than the best of the real moves. This has to be assumed separately both for true game values and the heuristic values being searched. The assumption about true values is an assumption that this position is not zugzwang. The assumption about the heuristic values amounts to an additional assumption that the evaluation function is really measuring something related to the game.

Both these assumptions seem to be reasonable in practice, and their truthfulness could perhaps be measured for evaluation functions in chess, although this has not been attempted.

10. Depth-Limited Versions of QUIESCE, Fat Values and Nested Minimax

As the search proceeds, null-move values will narrow the range of possible values. These values will, of course, be passed around the tree for use by the alpha-beta mechanism. However, it should be noted that the bounds arising from null moves are *not* alpha-beta "bounds." Null-move values produce a bound on the actual value of the current position. Alpha-beta "bounds" are a limitation on the range of interest we currently have in the actual value.

This distinction becomes important if the search is terminated prematurely (that is, by a depth limit). In this case null-move bounds can be returned as part of the top-level result (whereas it would not make sense to return alpha-beta "bounds" as results). In general, the result will be a range (more picturesquely called a "fat value") which is reliable plus an unreliable value within it. At the terminal depth, only an unreliable point value is available, but as values are backed up, they can be combined with null-move values. The result of this combination is a reliable fat value with an unreliable point value located within it. The "⟨reliable range⟩ + unreliable value inside" pattern gives rise to the concept of "nested evaluations" or "nested minimax" (Beal [4]).

```
QUIESCE-D(lower, upper, d) integer lower, upper, d;
{  integer bestv;
   if(d = 0)  { v ← evaluate(lower,upper);  return(–INF,v,INF);  }
   LOWER ← – INF;  UPPER ← – INF;
   makenull;  bestv ← – evaluate(–upper,–lower);  unmakenull;
   if(bestv > lower)  LOWER ← bestv;
   foreach move m do
   {  if( bestv >= upper )  return(LOWER,bestv,INF);
      make(m);  Lm,vm,Um ← QUIESCE-D(–upper, –bestv, d-1);  unmake(m);
      L ← – Um;  v ← – vm;  U ← – Lm;
      if( L > LOWER )  LOWER ← L;
      if( v > bestv )  bestv ← v;
      if( U > UPPER )  UPPER ← U;
   }
   return(LOWER,bestv,UPPER);
}
```

Fig. 5. Depth-limited version of null-move quiescence search.

Figure 5 shows QUIESCE-D, a depth-limited version of QUIESCE. It handles two distinct reliability levels and therefore produces evaluations with one level of nesting. Two applications of QUIESCE-D, in the manner of Fig. 2, would produce three reliability levels, and hence an evaluation with two levels of nesting. Nested evaluations would also arise from a minimax search that had recourse to a special-purpose evaluation that only became applicable occasionally.

Multiple applications of QUIESCE, or a variety of special-purpose evaluation mechanisms, could in principle produce evaluations with any number of levels of nesting. An example of the procedure for minimaxing with doubly nested evaluations is given in "Selective Search without Tears" (Beal [5]).

The QUIESCE of Fig. 3 could, of course, be depth-limited without attempting to distinguish reliable values which arose from null-move termination from unreliable values obtained at the depth limit. This amounts to throwing away the fat value part of the result, and although the resulting algorithm is simple and more familiar, it loses valuable information. For example, an iterative search should be stopped when a definite result is obtained. Also, incomplete searches can yield valuable fat value information. Figure 6 shows an example where the final value is not known, but the move choice is already definite. Another interesting situation arises when the known values offer a choice between a safe move, $[0, 0]$, and an uncertain move that may bring gain or loss, $[-2, 3]$, as in Fig. 7.

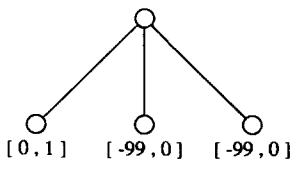


Fig. 6.

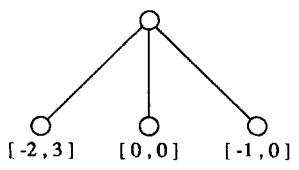


Fig. 7.

11. Conclusions and Comparison with Other Search Techniques

A good result on tactical problems in chess is relatively easy to achieve. There may well be positions and domains in which QUIESCE is not very successful. However, its simplicity and generality, and cost effectiveness in chess tactics suggest that it may be a better algorithm to start from than full-width searching (which has its own disadvantages). Its generality means that it could be applied to any minimax problem, thus making it a technique as widely applicable as alpha-beta. If it should turn out that material balance in chess was typical, rather than exceptional, then null-move quiescence would be as important as alpha-beta to cost-effective searches.

Improvements and extensions to QUIESCE can include not only iterative deepening, transposition tables, and move-ordering heuristics, which are currently standard technology in full-width searches, but also additional interior evaluations to test for zugzwang, additional interior evaluations known to be of higher reliability in special situations, and mechanisms controlling the effort spent at different levels of the quiescence hierarchy.

The logic of the null move can be related to two other search algorithms. If the null move has a particular value, it is probably the case that several moves have at least that value. If one cares to assume any particular average number, say 6, then a closed-off null-move search is equivalent to a locked value of degree 6. Locked-value searches [3] are searches which continue until they strike small local configurations where more than N (the lock number) have to change to change the value of the configuration. More flexibly, each null-move bound can be regarded as preparing one side of an eventual double-sided lock on a deeper position.

McAllester's conspiracy numbers [14] have some affinity with the earlier concept of locked values, but are more general. Conspiracy numbers allow incremental accumulation of "locks" over the whole tree, and use global information over the whole tree to decide tree growth. Nevertheless, just as with the locked values, the null move can be regarded as providing (cheaply) the equivalent of a conspiracy number of 6 (or whatever). This idea may improve the economics of conspiracy number algorithms.

ACKNOWLEDGEMENT

I am indebted to Hans Berliner for many helpful comments on an earlier version of this paper.

REFERENCES

1. G.M. Adelson-Velskiy, V.L. Arlazarov and M.V. Donskoy, Some methods of controlling the tree search in chess programs, *Artificial Intelligence* **6** (1975) 361–371.
2. D.F. Beal, An analysis of minimax, in M.R.B. Clarke, ed., *Advances in Computer Chess* **2** (Pergamon, Oxford, 1980) 103–109.

3. D.F. Beal, Benefits of minimax, in: M.R.B. Clarke, ed., *Advances in Computer Chess* **3** (Pergamon, Oxford, 1982) 17–24.
4. D.F. Beal, Mixing heuristic and perfect evaluations: Nested minimax, *ICCA J.* **7** (1984) 10–15.
5. D.F. Beal, Selective search without tears, *ICCA J.* **9** (1986) 76–80.
6. H.J. Berliner, Chess as problem solving: The development of a tactics analyzer, Ph.D. Thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA (1974).
7. H.J. Berliner, The B* tree search algorithm: A best-first proof procedure, *Artificial Intelligence* **12** (1979) 23–40.
8. A. Bernstein, et al., A chess-playing program for the IBM 704 computer, in: *Proceedings Western Joint Computer Conference* (1958) 157–159.
9. J.J. Gillogly, The technology chess program, *Artificial Intelligence* **3** (1972) 145–163.
10. R.D. Greenblatt, D.E. Eastlake and S.D. Crocker, The Greenblatt chess program, *Fall Joint Computer Conference* **31** (1967) 801–810.
11. L.R. Harris, The heuristic search under conditions of error, *Artificial Intelligence* **5** (1974) 217–234.
12. H. Kaindl, Searching to variable depth in computer chess, in: *Proceedings IJCAI-83*, Karlsruhe, FRG (1983) 760–762.
13. D.E. Knuth and R.W. Moore, An analysis of alpha-beta pruning, *Artificial Intelligence* **6** (1975) 293–326.
14. D.A. McAllester, A new procedure for growing mini-max trees, Tech. Rept., Artificial Intelligence Laboratory, MIT, Cambridge, MA (1985).
15. A.J. Palay, Searching with probabilities, Rept. CMU-CS-83-145, Carnegie-Mellon University, Pittsburgh, PA (1983).
16. F. Reinfeld, *Win at Chess* (Dover, New York, 1945).
17. C.E. Shannon, Programming a computer to play chess, *Philos. Mag.* **41** (1950) 256–275.
18. D.J. Slate and L.R. Atkin, CHESS 4.5: The Northwestern University chess program, in: P.W. Frey, ed., *Chess Skill in Man and Machine* (Springer, New York, 1977, 2nd ed., 1983) 82–118.
19. K. Thompson and J.H. Condon, Belle chess hardware, in: M.R.B. Clarke, ed., *Advances in Computer Chess* **3** (Pergamon, Oxford, 1982).
20. D.E. Wilkins, Using patterns and plans to solve problems and control search, Ph.D. Thesis, Rept. STAN-CS-79-747, Stanford University, Stanford, CA (1979).