PREPRINT OF AN ARTICLE* IN: ICCA JOURNAL 21(3), PP. 156-167, SEP. 1998.

Efficient Interior-Node Recognition

Ernst A. Heinz†
Institute for Program Structures and Data Organization (IPD)
School of Computer Science, University of Karlsruhe
P.O. Box 6980, D-76128 Karlsruhe, F.R. Germany

August 18, 1998

Abstract

This article re-examines the implications of *interior-node recognition* when focussing on its efficient yet seamless integration into modern chess programs. By means of a thorough discussion of its fundamental principles, we reveal various problems related to the practical application of interior-node recognition. Then, the rest of the article presents an implementation framework for recognizers that solves all known problems and has already proven its practical viability in our high-speed chess program DARKTHOUGHT.[1]

Among others we introduce the new concept of *material signatures* which allow for a quick and easy classification of chess positions into different categories of material distribution. By including material signatures in the internal representation of the chess board, they can incrementally be updated during the execution of moves. This makes the computation of material signatures extremely cheap in practice.

1 Introduction

Without knowing about Slate's 1984 article we independently reinvented the concept of "interior-node score bounds" [23]. Based thereon, we developed our own so-called "interior-node recognizers" and have used them in our chess program DARKTHOUGHT [13] since its tournament début at the 8th World Computer Chess Championship (WCCC) in Hong Kong, May 1995.

DARKTHOUGHT is a fast yet sophisticated alpha-beta searcher using PVS/NEGASCOUT [7, 16] with state-of-the-art enhancements like extended and normal futility pruning [12, 20, 26], internal iterative deepening [2, 22],

*The published version of the article won the ICCA Journal Award 1998.
†Email = <heinze@ira.uka.de>, WWW = <http://wwwipd.ira.uka.de/~heinze/>
[1] WWW = <http://wwwipd.ira.uka.de/Tichy/DarkThought/>

dynamic move ordering (history+killer heuristic) [1, 9, 19, 21, 24], recursive null-move pruning [4, 8, 10], selective extensions [2, 3], and an extended transposition table [15, 24]. DarkThought routinely reaches 200K nodes per second (nps) and search depths of 11–13 plies in normal middle-game positions at tournament time-controls on a 500MHz DEC Alpha-21164a PC164 workstation. The search speed often doubles in endgame positions and actually peaks at 750K nps on the aforementioned hardware.

To the best of our knowledge, there exist no other published works about interior-node recognition than Slate's article written 14 years ago. At least somewhat related are two equally old articles by Beal who explored how to mix heuristic and perfect evaluations [5, 6]. In contrast to Slate's disjoint score bounds for these different kinds of evaluations (see Section 2), Beal's "nested minimax" solution requires the search to back up multiple values that indicate the degree of reliability for the resulting scores. But Beal's scheme combines poorly with alpha-beta pruning because it substantially reduces the number of possible cutoffs. Regarding research in another area, private communication with J. Schaeffer revealed that the reigning World Man-Machine Checkers Champion Chinook [17, 18] employs techniques that are similar in concept to interior-node recognition. Unfortunately, however, the Chinook team has not yet published anything as to this respect.

The strongest chess programs running on supercomputers of the early 1980s barely reached search depths of 8 plies in the middlegame while processing around 20K nps. Thence, it is hardly surprising that Slate's article does not mention any efficiency problems related to the implementation of interior-node recognizers. This seems to have been no big issue back in 1984 because slow searchers usually tend to invest much effort into accurate evaluations. But at today's speeds of several hundred thousand nodes per second, the efficiency of the whole recognizer implementation matters a lot.

In the following sections we discuss and explain in detail how we solved all the conceptual and efficiency problems related to interior-node recognition if applied by a modern high-performance chess program. We seamlessly integrated interior-node recognition with the rest of our search (e.g. transposition tables) while never comprising any speed, knowledge, or accuracy. Our design and implementation proved their practical viability under tough tournament conditions during the 15th World Microcomputer Chess Championship (WMCC) in Paris, October 1997. The version of DarkThought competing there contained a lot of recognizers for many different types of endgame positions. With all its recognizers enabled and running on a cooled 767MHz DEC/Kryotech Alpha-21164a workstation DarkThought surpassed the magic speed limit of 1M nps during this event [11, 14].

2 Fundamentals of Interior-Node Recognition

The basic idea of interior-node recognition is really trivial yet extremely powerful. It simply says: stop the search immediately and cut the whole subtree when reaching a position with a known game-theoretical value (draw, loss, or win). At a first glance, this theoretically sound cutoff scheme looks easy to implement. But the practical consequences of integrating it into a game-playing alpha-beta searcher are highly complex.

In case of losses and wins, for instance, you cannot just return the same loss/win values as search results for all lost/won positions. Otherwise, the game-playing program will not make any progress in these positions because there is nothing to distinguish between them. Consequently, appropriate scoring functions need to be supplied for all recognized positions. While the recognizer scores must not interfere with mate and static evaluation scores, they should still provide a good estimate of the recognized game-theoretical value. Slate's concept of interior-node score bounds [23] solves this problem by reserving two normally unused ranges of values for recognizer scores. Figure 1 shows how neatly the new ranges fit directly in-between the usual mate and static evaluation scores.

–M ... –M+N	–R–K ... –R	–E ... 0 ... +E	+R ... R+K	M–N ... M–1
Mated-in-N	Recog-Loss	Eval. ↑ Scores	Recog-Win	Mate-in-N

with

$$-M \leq -M+N < -R-K < -R < -E < 0 < +E < +R < R+K < M-N \leq M-1.$$

Figure 1: Slate's Interior-Node Score Bounds.

During the 8th WCCC in 1995, DarkThought used an equivalent scoring scheme for its interior-node recognizers which included access functions to Thompson's 5-piece endgame databases [25]. Such databases combine neatly with interior-node recognition because they contain game-theoretical values for elementary positions. Thompson's endgame databases employ a distance-to-conversion rather than a distance-to-mate metric and thus do not return exact mate scores. Therefore, the recognizers converted the raw database results into a special recognizer score accounting among others for the material balance of the position.

We abandoned the scheme of interior-node score bounds in 1996 when it became clear that disjoint scoring ranges for recognizers and the static evaluation tend to introduce frequent scoring inconsistencies during iteratively deepened searches of high depths. The inconsistencies are caused by incomplete recognizer coverage of positions that represent subgames of other positions for which recognizers exist. Pawn (under-) promotions often lead to such subgames, especially if any recognized positions contain more than one Pawn per side. This kind of recognizer incompleteness is hard to avoid in practice unless the implementation restricts recognizers to solely trivial cases. Hence, in our opinion Slate's theoretically sound scheme of interior-node score bounds with disjoint ranges for recognizer results and static evaluation scores carries only little practical value for modern high-speed chess programs.

Other difficult problems caused by interior-node recognizers relate to their efficient implementation. First and foremost, the efficient selection of recognizers proves to be a challenging task. The problem is to decide at each node in the search tree whether the current position lends itself to recognition at all and, if so, how to select the appropriate recognizer. Please note as an important constraint that the recognizer selection constitutes just a tiny fraction of the whole work done at each node with an overall frequency of several hundred thousand times per second. Further efficiency problems and general conceptual complications arise during the algorithmic encoding of the underlying chess knowledge into the final recognizer functions.

3 Recognizers and Transposition Tables

Independent of any implementation details, interior-node recognizers may be viewed as access functions to permanent databases of omniscient search results. In this rather abstract respect recognizers resemble transposition tables which also serve as databases of search results and are found in all decent chess programs. Therefore, one must integrate recognizers with transposition tables if planning to augment the search by interior-node recognition. But despite its obvious importance Slate's article does not mention anything about this task. Hence, we developed our own new scheme for the efficient yet seamless integration of recognizers and transposition tables. In the remainder of this section, we explain and discuss the peculiarities of our new scheme in more detail.

The central idea of our scheme is to handle interior-node recognition conceptually like an additional transposition-table access that gets executed after the normal ones. Thus, the search only falls back on interior-node recognition if all probes in the standard transposition tables fail. This greatly reduces the number of recognition trials whenever the rate of successful probes in the standard transposition tables is high (e.g. in endgames).

Treating interior-node recognition conceptually like an access to a transposition table provides for other advantages as well. In particular, our integration framework actually requires the results returned by recognizer calls to contain more information than just plain scores. Like entries in transposition tables they must at least comprise bound information, too. The following subsections discuss the peculiarities of our scheme in more detail.

3.1 Recognizer Results

In order make recognizer results compatible with the results of probes from standard transposition tables, we compose them of bound and score information. Incorporating bounds into their results further increases the usefulness of recognizers because it enables them to return a meaning of the kind "this is at least/at most a draw/loss/win for Black/White". Furthermore, recognizer results may contain an expense indicator that tells the controlling search whether to store them directly into the standard transposition tables. For the sake of clarity, the following descriptions elaborate on each of the three proposed result items.

Bound. The bound information determines if the recognizer score is exact, a lower bound, or an upper bound. Because of the intended similarity we suggest to adopt the same values for identification of the different bounds as those already in place within the transposition tables. We introduce the special new bound value $fail\_bound$ for the sole purpose of signalling unsuccessful recognitions.

Expense. In general the search does not store results of successful interior-node recognitions directly into the standard transposition tables because they do not have a best move associated with them. Moreover, recognizer calls are normally much cheaper than even pure quiescence searches such that it does not make much sense to clog precious transposition entries with them. In case of really expensive recognition results (e.g. external endgame database values), however, direct storing might be beneficial. To this end the recognizer supplies the expense information as part of its result. Based thereon, the controlling search decides if it deems direct storing of the recognition result worthwhile.

Score. The score information features just the according numerical value.

Due to the diversity of information that our recognizer results contain, their handling after successful recognitions proves to be somewhat involved. Figure 2 provides the skeleton of a normal search function which outlines the necessary additions. Because of its verbose ANSI-C style and the numerous embedded comments this search skeleton is quite self-explanatory. It assumes all not explicitly declared identifiers to be suitably defined and initialized. Please note how closely the actions resemble the handling of bounds and scores as read from standard transposition tables. Successful recognitions actually “massage” the search tree in a similar manner as successful accesses to transposition tables. Beside causing direct cutoffs by exact, fail-high, or fail-low scores, recognizer results may also shrink the alpha-beta window (raise alpha or lower beta) at the current node while the search still descends on its way down in the game tree. Upon lowering beta, the search skeleton carefully keeps track of the largest upper bound (denoted by *upper*) that it may return. It does so in the same way as it keeps track of the best score so far. Fail-high cutoffs respect this upper bound by returning the minimum of *upper* and the best score so far as their result. This works flawlessly because $beta \le upper$ always holds.

```
int search(int alpha, int beta, int depth, int move, node parent) {
    node current; int score = -infinite_val; int upper = -score;
    /* execute the opponent's move and determine the search extensions */
    make_move(parent, move, &current);
    depth += extensions(depth, move, current);
    ...
    /* call recognizer and analyze the result in case of recognition success */
    {   recog recog_func = ...;
        recog_result rresult = recog_func(current);
        if (recog_success(rresult)) {
            /* extract bound, expense, and score from recognizer result */
            int bound, expense, rscore;
            recog_extract(rresult, &bound, &expense, &rscore);
            /* store results of expensive recognitions in transpos. table */
            if (store_directly_into_tt(expense))
                store_recog_into_tt(current, depth, rresult);
            /* handle recognizer results that are exact scores */
            if (bound == exact_bound) return rscore;
            /* handle recognizer results that are lower bounds */
            if (bound == lower_bound) {
                /* recognizer fail-high beta cutoff */
                if (rscore >= beta) return min(rscore, upper);
                /* raise alpha and the best score so far */
                if (rscore > alpha) alpha = rscore;
                if (rscore > score) score = rscore;
            }
            /* handle recognizer results that are upper bounds */
            if (bound == upper_bound) {
                /* recognizer fail-low alpha cutoff */
                if (rscore <= alpha) return rscore;
                /* lower beta and the best upper bound so far */
                if (rscore < beta)  beta  = rscore;
                if (rscore < upper) upper = rscore;
            }
        }
    }
    ...
}
```

Figure 2: Search with Result Handling of Successful Recognitions.

3.2 Recognizer Scores

In view of the hardly avoidable incompleteness of the recognizer coverage (see Section 2) we suggest to employ recognizer scores that are compatible with static evaluation scores. Hence, we base our recognizer scores mainly on the material balances of the according positions if neither draws nor mates get recognized.[2] Compatibility with static evaluation scores has the further advantage of enabling the implementation to replace expensive evaluations by cheap recognitions. Simply skip the calls of the static evaluation function in case of successful recognitions and reuse the already computed recognizer scores as static evaluation scores.

In order to illustrate how scoring functions for interior-node recognition work, we develop such functions for the well-known classes KBBK (with differently coloured Bishops) and KBNK of elementary endgames. The winning strategy for KBBK with is to drive the lone King into any corner and mate it there by a concerted effort of all three pieces. The strong King must approach the scene, too, because two minor pieces cannot mate on their own. The same winning strategy applies to KBNK with the notable exception that the lone King has to be driven into a corner controlled by the Bishop of the strong side. Therefore, the distance between the lone King and the nearest target corner as well as the distance between the lone and the strong King are important traits of the respective positions. Beside these explicitly obvious relations of the two Kings the recognizer scores should also reflect the different degrees of difficulty involved in winning KBBK (with a maximal distance-to-mate of 19 moves) in contrast to winning the much harder KBNK (with a maximal distance-to-mate of 33 moves).

The below two scoring functions $KBBK(p)$ and $KBNK(p)$ implement the above guidelines for given positions p with minor pieces valued at 3.25 Pawns and $k\_corner\_dist()$, $kb\_corner\_dist()$, as well as $kk\_dist()$ suitably defined. Both scoring functions are efficient to compute and easy to implement. They let the search make progress in all recognized positions and channel it into greatly preferring KBBK over KBNK but not over positions where the winning side is up a Queen or more.

$$
\begin{aligned}
KBBK(p) &= material\_balance(p) + winning\_bonus(p) \\
&= 6.5 + 2.5 - 0.25 * k\_corner\_dist(p) - 0.125 * kk\_dist(p) \\
&= \boxed{9 - 0.25 * k\_corner\_dist(p) - 0.125 * kk\_dist(p)}
\end{aligned}
$$

[2] Please note that mates (as read from distance-to-mate endgame databases, for example) require special care, too, because standard quiescence searches (whether captures-only or checks included) can normally not return any real mate scores.

$$
\begin{aligned}
KBNK(p) &= material\_balance(p) + winning\_bonus(p) \\
&= 6.5 - 0.5 - 0.25 * kb\_corner\_dist(p) - 0.125 * kk\_dist(p) \\
&= \boxed{6 - 0.25 * kb\_corner\_dist(p) - 0.125 * kk\_dist(p)}
\end{aligned}
$$

4 Efficient Detection and Selection of Recognizers

In order to select an appropriate recognizer for a given position you must first decide whether the position lends itself to recognition at all. This is an absolutely non-trivial task if executed at every node in the search tree with an overall frequency of several hundred thousand times per second. Moreover, it constitutes only a tiny fraction of the whole work done at each node. For the sake of simplicity and practical relevance let us assume that both sides can have at most 8 Pawns and 2 pieces of each kind. Under these simplified assumptions there are still $(3^4*9)*(3^4*9) = 3^{12} = 531441$ different constellations of material distribution for Black and White to decide between (four kinds of 0 to 2 pieces and 0 to 8 Pawns per side).

Of course, a lookup table with 531441 pointer elements referring to the according recognizers of the different material constellations provides a straightforward solution to the detection problem. Just mark all elements in the lookup table with the special value NULL if there are no recognizers available for their constellations. The major drawback of this easy yet naive solution springs from its huge memory consumption. The lookup table requires $531441 * sizeof(pointer)$ bytes of memory which roughly tranlsates to 2 MBytes in case of 4-byte pointers and to 4 MBytes in case of 8-byte pointers. Frequently accessed lookup tables of such large sizes are very inefficient because they tend to thrash the data caches and other data paths of modern computers, thus causing the whole application to slow down considerably.

The remainder of this section discusses how to refine the straightforward approach into a viable solution by greatly reducing the size of the lookup table. During the course of our work we discovered that the most effective way to do so is by means of shrinking the index space of the table. The implementation of our final solution in DarkThought needs less than 1 KBytes of memory. It runs at a blazing speed and does not affect any other part of the program through cache thrashing or related effects.

4.1 Material Signatures

Our first refinement to the index scheme of the lookup table replaces the exact piece counts by 1-bit flags that solely indicate the presence of any Pawns or pieces of the respective kinds. The refined scheme treats both

Pawns and pieces in an identical way. The new index function comprises ten 1-bit parameters which are simply concatenated to yield the index number. The ten 1-bit parameters naturally break up into two groups of five bits, namely one each for Pawns and the four kinds of pieces per side. The two groups of five 1-bit values represent the "material footprints" of their sides. The separate concatenation of the five 1-bit values on their own results in 5-bit strings that we use as signatures for the material states of both sides. Below we summarize some important properties of these novel *material signatures*.

Properties of Material Signatures in Chess

$$mstate(side,\, pos) = \sum_{k\,=\,0}^{4} \Big(2^k * has\_piece(side,\, pos,\, k)\Big) \;\in\; \{0 \,\ldots\, 31\}$$

$$\Downarrow \qquad k \in \{0 \ldots 4\}$$

$$\begin{aligned} piece\_nr(side,\, pos,\, k) \neq 0 \;&\Longleftrightarrow\; has\_piece(side,\, pos,\, k) \\ &\Longleftrightarrow\; \Big(mstate(side,\, pos) \;\;\&\;\; 2^k\Big) \neq 0 \\ &\Longleftrightarrow\; \Big((mstate(side,\, pos) \gg k) \;\;\&\;\; 1\Big) = 1 \end{aligned}$$

The integer values of material signatures range from 0 to $2^5 - 1 = 31$. Bit $k \in \{0 \ldots 4\} = \{Bishop, \ldots, Pawn\}$ of a material signature gets set if the side in question has at least one man of kind k on the board. Like other important state information in computer chess, material signatures lend themselves to efficient calculation by incremental updates during the execution of moves by the search. To this end DarkThought includes two material signatures in its internal representation of the chess board, namely one for Black and one for White.

With the material signatures of Black and White spanning its index space, the refined lookup table contains $2^{10} = 32{*}32 = 1024$ elements overall. The elements require $1024 * sizeof(pointer)$ bytes of memory altogether which tranlsates to 4 KBytes in case of 4-byte pointers and to 8 KBytes in case of 8-byte pointers. Hence, the adoption of material signatures for indexing reduces the memory consumption of the lookup table more than 500-fold as compared with the initial naive approach from above. But even the refined lookup table still wastes a lot of precious data-cache space due to poor information density. According to extensive practical experiences with recognizers in DarkThought, most entries of the table tend to be NULL.

4.2 Further Empirical Refinements

Our aforementioned practical experiences gave rise to an empirical investigation of all positions for which we knew or at least imagined automatic interior-node recognition to be possible. The ensuing characterization of potentially recognizable positions showed the following.

- Nearly all potentially recognizable positions feature at most two pieces per side excluding Pawns.[3]
- Many potentially recognizable positions feature equal or at least partially inclusive material signatures for both sides.
- Many potentially recognizable positions feature a lone King.

Based on the last two observations above, we further refined the lookup table and achieved another 16-fold reduction of its index space in our current version of DarkThought. This decreased the number of elements in the lookup table to mere 64, resulting in a cache-friendly memory consumption of 256 bytes with 4-byte pointers and 512 bytes with 8-byte pointers.

The main idea of the new index scheme is to separate positions with lone Kings from the rest because only one of their two material signatures conveys interesting information. Hence, we split the lookup table into two halfs. In positions with a lone King the lower 32 entries correspond to the material signatures of the strong sides. The remaining positions get mapped to the upper 32 entries by combining the material signatures of both sides into a single value through a logical OR operation. With the material signatures of Black and White given by *b_mstate* and *w_mstate* respectively, the new index function allows for its efficient calculation as detailed below. Modern computers execute the six necessary operations in a few clock cycles after loading both material signatures into registers of their CPUs.

Efficient Index Calculation for Recognizer Lookup Table

$$\underbrace{(b\_mstate \mid w\_mstate)}_{\in \{0 \ldots 31\}} \quad + \quad \underbrace{32 * ((b\_mstate \neq 0) \;\&\&\; (w\_mstate \neq 0))}_{\text{(either 0 or 32, selects between lower and upper half)}}$$

A minor drawback of the efficient index scheme springs from its mapping behaviour. The logical OR operation combines several distinct material signatures into one and the same result. Thence, the new index function maps different constellations of material distribution to identical entries of the lookup table. The constellations KBKN, KBBKN, KBBKBN, KBBKNN, KBNKB, and KBNKBN are good examples for this peculiar mapping behaviour because logical OR combinations of their material signatures always

[3] The only valid yet practically irrelevant counterexamples that we know of are positions with more than two equally coloured bishops for any side.

yield BN = Bishop + Knight. Consequently, the recognizer functions referred to in the lookup table do not know the exact material constellations of the positions they receive in advance. Instead, they must dynamically determine the actual material constellations during each call in order to select the appropriate recognition procedure (e.g. KXXK versus KXK for any kind of piece X other than a King). Our implementation assists the dynamic selection process by means of efficiently accessible 4-bit counters for Pawns and each kind of pieces per side. DarkThought keeps the counters in its internal representation of the chess board and incrementally updates them whenever the search executes a move.

Furthermore, we make sure to avoid any senseless function calls in unrecognizable positions. Many such calls would normally ensue from the mapping behaviour of the new index scheme: KBBKBN, KBBKNN, and KBNKBN, for instance, are currently not recognizable but they get mapped to the same function pointer of the lookup table as KBKN for which an endgame database enables the perfect recognition. An additional lookup table with 1024 flag bits occupying just 128 bytes of memory provides an elegant solution for this problem. The new table employs the index scheme of the last section with fully concatenated material signatures of both sides. Each flag bit of this second lookup table indicates whether the function referred to by the main lookup table can really recognize positions with material constellations that correspond exactly to the full index of the second table. Hence, the additional lookup table allows for the quick and easy detection of superfluous function calls altogether. But it does not eliminate the minor drawback of mapping similar material constellations to the same entry of the recognizer lookup table (see above).

Thanks to the ample descriptions above, the implementation details of our final solution for the efficient detection and selection of recognizers should now be ready to understand. Figure 3 presents the skeleton of a normal search function that outlines the necessary additions. Its verbose ANSI-C style and the various embedded comments make this search skeleton quite self-explanatory. It assumes all not explicitly declared identifiers to be suitably defined and initialized. The search skeleton omits the handling of successful recognitions because we already outlined this Figure 2. What deserves special attention, however, is the design of the second lookup table hosting the 1024 availability flags. For the sake of efficient storage and access, we use an array of 32-bit unsigned integers to implement the table. While Black's material signature selects the array element, White's material signature extracts the desired bit of the selected value.

```
/* global declaration of tables for availability flags and recognizer functions */
unsigned int32 recog_avail[32];
recog recog_table[64];

/* macro definition for the detection of available recognizers */
#define recog_available(b_mstate, w_mstate) \
    ((recog_avail[(b_mstate)] >> (w_mstate)) & 1)

/* macro definition for the calculation of a recognizer index */
#define recog_index(b_mstate, w_mstate) \
    (((b_mstate) | (w_mstate)) + (32 * ((b_mstate) && (w_mstate))))

/* implementation of the central alpha-beta search function */
int search(int alpha, int beta, int depth, int move, node parent) {
    node current; tt_entry tt_ref; int hit, score = -infinite_val;
    /* execute the opponent's move and determine the search extensions */
    make_move(parent, move, &current);
    depth += extensions(depth, move, current);
    ...
    /* probe the transposition tables at the current node */
    hit = probe_transposition_tables(current, depth, &tt_ref);
    /* if the probe failed try to apply an appropriate recognizer */
    if (!hit) {
        /* determine material signatures for Black and White */
        unsigned b_mstate = black_mstate(current);
        unsigned w_mstate = white_mstate(current);
        /* detect the availability of an appropriate recognizer */
        if ((b_pieces(current) <= 2) && (w_pieces(current) <= 2)
              && recog_available(b_mstate, w_mstate)) {
            /* select the according recognizer function */
            recog recog_func = recog_table[recog_index(b_mstate,
                                                       w_mstate)];
            /* call the selected recognizer function */
            if (recog_success(recog_func(current)))
                {...}  /* handle the result of a successful recognition */
        }
    }
    ...
}
```

Figure 3: Search with Detection and Selection of Recognizers.

Last but not least, we like to sketch the proper initialization of the two lookup tables for availability flags and recognizers. Figure 4 shows the skeleton of an according initialization function containing all the necessary actions. As before, its verbose ANSI-C style and the embedded comments make the code easy to comprehend. The initialization skeleton implicitly reuses the global definitions of macros and variables from Figure 3. It of course assumes all other not explicitly declared identifiers to be suitably defined and initialized. As for the installation of new recognizers, it is important to note that the efficient index scheme of the recognizer lookup table is commutative in its parameters: $recog\_index(x, y) = recog\_index(y, x)$. Because such commutativity does not hold for the index scheme of the availability lookup table we must set two different flag bits therein for each new recognizer. These flag bits correspond to the two material constellations which result from interpreting two given material signatures $mstate\_1$, $mstate\_2$ as representing either Black and White or White and Black respectively.

5 Recognizer Functions

The latest version of DarkThought features high-speed recognizers for the following classes of endgame positions with little material on both sides.

- KK, KBK, KNK (always drawn),
- KBKB, KBKN, KNKN (normally drawn but exceptional wins possible for any side by mates in 1 move),
- KBBK, KBNK (normally won for the strong side but exceptional draws possible by stalemate or by conversion to KBK and KNK),
- KNNK, KNNKB, KNNKN (normally drawn but exceptional wins possible for any side by mates in ≤ 7 moves),
- KQX*K, KRX*K (normally won for the stronger side but exceptional draws possible by stalemate or by conversion to drawn subgames, X* = arbitrary major and minor pieces),
- KPPP*K (sophisticated detection of draws, P* = one or more Pawns),
- KBP*K, KBP*KB, KBP*KN (sophisticated detection of draws for bishops of wrong colour, P* = one or more Pawns),
- KPK, KQK, KRK (3-piece endgame databases),

```
/* installs the given func. as a recognizer for the desired material constellation */
void new_recog(recog func, unsigned mstate_1, unsigned mstate_2) {
    /* set the according flag bits in the availability lookup table */
    recog_avail[mstate_1] |= (1 << mstate_2);
    recog_avail[mstate_2] |= (1 << mstate_1);
    /* enter the given function into the recognizer lookup table */
    recog_table[recog_index(mstate_1, mstate_2)] = func;
}
/* initializes the lookup tables of availability flags and recognizer functions */
void init_recognizers(void) {
    /* initialize both lookup tables with their respective zero values */
    unsigned i; for (i = 0; i <= 31; i++) {
        recog_avail[i] = msig(0,0,0,0,0);   /* B, N, Q, R, P = 0 */
        recog_table[i] = NULL;
    }
    /* install the available recognizer functions in both lookup tables */
    /*          recog. func.          B N Q R P          B N Q R P  */
    new_recog(kb_k_recog,  msig(1,0,0,0,0), msig(0,0,0,0,0));
    new_recog(kq_kr_recog, msig(0,0,1,0,0), msig(0,0,0,1,0));
    ...
}
```

Figure 4: Initialization of Lookup Tables for Flags and Recognizers.

- KBKP, KBPK, KNKP, KNPK, KPKP, KPPK, KQKB, KQKN, KQKP, KQKQ, KQKR, KRKB, KRKN, KRKP, KRKR (4-piece endgame databases).

It goes without saying that we continously search for new classes of positions that allow for their efficient interior-node recognition. We definitely want to add many more recognizer functions in the near future. In this respect, database recognizers for 5-piece endgames are promising candidates. According to our experiences with past versions of DarkThought, however, the recognition of and accesses to 5-piece endgame databases require careful fine-tuning until they perform at acceptably high speeds.

5.1 Implementation Example

In general recognizer functions must distinguish between three different types of recognitions corresponding to the different modes of their results (bound, exact, or fail). For any KB[B]K recognizer that does not access endgame databases taken as an example, this implies to detect the three conditions below.

1. KB[B]K is drawn if the Bishops cover squares of a single colour only. Then, the strong side is unable to mate the lone King and the KB[B]K recognizer can return an exact result with a draw score.

2. The lone King might still draw if it attacks a bishop (possible conversion to drawn KBK) or if it stands on any edge of the board with the strong King near (possible stalemate). In both cases, the KB[B]K recognizer has to back off and return a recognition failure as its result.

3. All other KB[B]K positions feature a winning KBB side and the recognizer can return bounded scores that indicate the great advantage.

Thanks to the KBBK scoring function from Section 3.2, we already know how to calculate the bounded scores for a winning KBB side. Now we simply put everything together in order to implement a working function for our KB[B]K example. Figure 5 presents an according skeleton for a KB[B]K recognizer function in ANSI-C. The skeleton outlines the decision procedure of the recognizer that evaluates the above three conditions and returns the appropriate results. As usual it assumes all not explicitly declared identifiers to be suitably defined and initialized.

6 Discussion and Conclusion

Position #23 (FEN "8/7p/8/p4p2/5K2/Bpk3P1/4P2P/8 w – –") of the well-known test suite BT-2630 exemplifies what interior-node recognizers are capable of. This endgame position features little material on both sides (KBPPP vs. KPPPP) and it requires White to play "g4" immediately in order to win. The table below shows that with the help of its recognizers DarkThought manages to solve the position after roughly a second in iteration #10 while running on a 500MHz DEC Alpha-21164a PC164 workstation. Without the recognizers, however, it takes DarkThought 2:24 minutes, 47 million additional nodes, and 8 more plies of search to do so.

```
/* interior-node recognizer for all KB[B]K endgame positions */
recog_result kb_k_recog(node kb_k_pos) {
    int recog_bound, recog_score;
    /* return an exact draw score if the KB[B] side is unable to mate */
    if (kb_cannot_mate_k(kb_k_pos))
        return recog_value(exact_bound, low_expense, draw_score);
    /* return a recognition failure if the lone King might still draw */
    if (k_attacks_kb_bishop(kb_k_pos) || kb_may_stale_k(kb_k_pos))
        return recog_value(fail_bound, 0, 0);
    /* calculate the KB[B] score and initialize the recognizer bound */
    recog_bound = lower_bound;
    recog_score = 9 * pawn_val - kk_dist(kb_k_pos) * (pawn_val / 8)
                    - k_corner_dist(kb_k_pos) * (pawn_val / 4);
    /* invert both the bound and the score if the lone King is to move */
    if (lone_k_to_move(kb_k_pos)) {
        recog_bound = upper_bound;
        recog_score = -recog_score;
    }
    /* return a bounded recognizer result if the KB[B] side definitely wins */
    return recog_value(recog_bound, low_expense, recog_score);
}
```

Figure 5: Recognizer Function for KB[B]K Endgame Positions.

BT-2630: Pos. #23	**With Recognizers**	**Without Recognizers**
Solved in Iteration	10	18
Time to Solution	1 sec	144 sec
#Nodes to Solution	351,263	47,531,238
Search Speed	325K nps	327K nps

These impressive differences spring to a large part from the added knowledge of the recognizers. Any chess expert who knows about the great importance of knowledge in endgames will agree that this example is not artificially constructed. The recognizers of DarkThought behave like this in countless other endgame positions with little material on both sides, too. But comparisons of different versions of the same program with and without recognizers do not allow for a sound quantification of the recognizer framework alone. They rather measure the combined merits of both the recognizer implementation and the added recognizer knowledge. Because this knowledge could also be implemented by other means than interior-node recognition (e.g. in the static evaluation function), we need to compare our recognizer implementation with full-fledged alternative implementations of the whole recognizer knowledge in order to measure the effectiveness of the recognizer framework per se.

We deem the usefulness of such comparisons far too low as to justify their substantial implementation efforts. The advantages of recognizers over other forms of the respective knowledge are quite obvious in our opinion. We are convinced that most readers will agree with us on this issue after reading the following brief summary about recognizers and their distinguishing features.

- Interior-node recognition applies the knowledge at the earliest possible time during the search. It can cause direct cutoffs of complete subtrees and it "massages" the game tree already on the way down of the search by tightening the alpha-beta bounds.

- Bounded recognizer results enable the painless implementation of inexact knowledge like "this position is at least/at most a draw/loss/win for Black/White".

- Recognizer functions encapsulate handy chunks of knowledge that are completely independent of each other.

- Accesses to endgame databases integrate perfectly with interior-node recognition (simply code them into appropriate recognizer functions).

- Successful recognitions reduce the number of expensive static evaluations by reusing the normally much cheaper recognizer scores instead.

- Efficient implementation schemes make our recognizer framework extremely fast such that it does not slow down the search at all.

- Material signatures are useful for many other important tasks beside the detection and selection of recognizers.

Last but not least, we invite all interested readers and especially the still skeptic ones to take a closer look at the latest test games of DarkThought on its WWW page at <http://wwwipd.ira.uka.de/Tichy/DarkThought/>. In many of these games DarkThought easily outplayed its strong commercial opponents in endgames with little material on both sides. The portrayed strength of DarkThought in such positions clearly springs from its high-speed endgame recognizers. They turned many objectively lost games into draws and correspondingly drawn games into wins. The strength of the recognizers also surfaced during several electronic discussions on the Internet. DarkThought routinely solved difficult endgame positions much faster than most other strong chess programs involved in the testing.

References

[1] Akl, S. G. and Newborn, M. M. (1977). The principal continuation and the killer heuristic. *ACM '77 National Conference*, Proceedings, pp. 466–473, ACM Press.

[2] Anantharaman, T. S. (1991). Extension heuristics. *ICCA Journal*, Vol. 14, No. 2, pp. 47–65.

[3] Beal, D. F. and Smith, M. C. (1995). Quantification of search extension benefits. *ICCA Journal*, Vol. 18, No. 4, pp. 205–218.

[4] Beal, D. F. (1989). Experiments with the null move. *Advances in Computer Chess 5*, D. F. Beal (ed.), pp. 65–79, Elsevier Science, ISBN 0-444-87159-4.

[5] Beal, D. F. (1986). Selective search without tears. *ICCA Journal*, Vol. 9, No. 2, pp. 76–80.

[6] Beal, D. F. (1984). Mixing heuristic and perfect evaluations: Nested minimax. *ICCA Journal*, Vol. 7, No. 1, pp. 10–15.

[7] Campbell, M. S. and Marsland, T. A. (1983). A comparison of minimax tree search algorithms. *Artificial Intelligence*, Vol. 20, No. 4, pp. 347–367.

[8] Donninger, C. (1993). Null move and deep search: Selective search heuristics for obtuse chess programs. *ICCA Journal*, Vol. 16, No. 3, pp. 137–143.

[9] Gillogly, J. J. (1972). The technology chess program. *Artificial Intelligence*, Vol. 3, No. 1–3, pp. 145–163 [reprinted in *Computer Chess Compendium*, D. N. L. Levy (ed.), pp. 67–79, Springer, 1989, ISBN 0-387-91331-9].

[10] Goetsch, G. and Campbell, M. S. (1990). Experiments with the null-move heuristic. *Computers, Chess, and Cognition*, T. A. Marsland and J. Schaeffer (eds.), pp. 159–168, Springer, ISBN 0-387-97415-6/3-540-97415-6.

[11] Hamlen, T. and Feist, M. (1997). Report on the 15th World Microcomputer Chess Championship. *ICCA Journal*, Vol. 20, No. 4, pp. 254–256.

[12] Heinz, E. A. (1998). Extended futility pruning. *ICCA Journal*, Vol. 21, No. 2, pp. 75–83.

[13] Heinz, E. A. (1997). How DarkThought plays chess. *ICCA Journal*, Vol. 20, No. 3, pp. 166–176.

[14] van den Herik, H. J. (1997). Surpassing the speed limit of 1,000,000 nodes per second (photograph of DarkThought versus Toledo 2000). *ICCA Journal*, Vol. 20, No. 4, p. 273.

[15] Nelson, H. L. (1985). Hash tables in Cray Blitz. *ICCA Journal*, Vol. 8, No. 1, pp. 3–13.

[16] Reinefeld, A. (1983). An improvement to the Scout tree-search algorithm. *ICCA Journal*, Vol. 6, No. 4, pp. 4–14.

[17] Schaeffer, J. and Lake, R. and Lu, P. and Bryant, M. (1996). Chinook: The World Man-Machine Checkers Champion. *AI Magazine*, Vol. 17, No. 1, pp. 21–29.

[18] Schaeffer, J. and Culberson, J. and Treloar, N. and Knight, B. and Lu, P. and Szafron, D. (1992). A World Championship caliber checkers program. *Artificial Intelligence*, Vol. 53, No. 2–3, pp. 273–290.

[19] Schaeffer, J. (1989). The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 11, No. 11, pp. 1203–1212.

[20] Schaeffer, J. (1986). *Experiments in Search and Knowledge*. Ph.D. Thesis, University of Waterloo [reprinted as Technical Report TR 86–12, Department of Computing Science, University of Alberta].

[21] Schaeffer, J. (1983). The history heuristic. *ICCA Journal*, Vol. 6, No. 3, pp. 16–19.

[22] Scott, J. J. (1969). A chess-playing program. *Machine Intelligence 4*, B. Meltzer and D. Michie (eds.), pp. 255–265, Edinburgh University Press, 1973, ISBN 0-852-24062-7.

[23] Slate, D. J. (1984). Interior-node score bounds in a brute-force chess program. *ICCA Journal*, Vol. 7, No. 4, pp. 184–192.

[24] Slate, D. J. and Atkin, L. R. (1977). Chess 4.5 – The Northwestern University chess program. *Chess Skill in Man and Machine*, P. W. Frey (ed.), pp. 82–118, Springer, 2nd ed. 1983, ISBN 0-387-90790-4/3-540-90790-4.

[25] Thompson, K. (1986). Retrograde analysis of certain endgames. *ICCA Journal*, Vol. 9, No. 3, pp. 131–139.

[26] Ye, C. and Marsland, T. A. (1992). Experiments in forward pruning with limited extensions. *ICCA Journal*, Vol. 15, No. 2, pp. 55–66.