6-PIECE ENDGAMES[1]

K. Thompson[2]

New Jersey, USA

ABSTRACT

Currently I feel that it is well within modest means to solve endgames with two Kings and four pure pieces. Pawn endgames are still a little out of reach. This contribution outlines an approach for achieving endgame solutions and provides the current status of solving 6-piece endgames. Some code for the construction of endgame databases is given and some pitfalls to be avoided are described.

1. THE LAST OF THE CD-ROMS

In this contribution I address the topic of 6-piece endgames, though their solution is not a reality yet. For a starting point, Section 2 will give some of the history of endgame databases, which is well documented in the *ICCA Journal* (Van den Herik and Herschberg, 1985, 1986). Moreover, 6-piece endgames mean two Kings and four pure pieces. The previous work in the area of endgame databases is readily available. The actual results of the endgames are on a set of CD-ROMs, in three official volumes (cf. Thompson, 1991; The Editors (of the *ICCA Journal*), 1992, 1993) plus one volume that was circulated informally. I have been bringing these out over the last ten years or so, for essentially the cost of reproducing them, maybe for less. I have run out of these and I am not producing them anymore. The last ten are available right here [Thompson brought them to the conference and gave them to interested participants]. Now I am officially out of business!

2. THE TECHNOLOGY CURVE

If we look at what happened with the 5-piece endgames and use that as a basis for comparison, i.e., putting the size of the 5-piece endgame at 1, then the size of the 6-piece endgame is a factor of 64 times larger, multiplied by another factor of 1.5, stemming from the necessity of going from single precision to double precision addresses, yielding a ratio of some 100. This is per 6-piece endgame. There are many more 6-piece endgames than 5-piece endgames, probably a factor of five more. Finally, the lines of play in the 6-piece endgames tend to be longer (say twice as long) and the size of the endgame depends on how many moves it takes. Altogether, we can conjecture that building all 6-piece endgame databases is a factor of 1000 more time consuming than building all 5-piece endgame databases.

After my first 5-piece endgame result (KBBKN) in 1983 (Roycroft, 1984), I did all the 5-piece endgame databases around 1985 on a spare machine I had close by; it was not the biggest machine available at that time. The speed of that machine was about 1 MIP (million instructions per second), while the machine I have now for the construction of the 6-piece endgame databases is about 150 MIPs. The memory size, one of the critical factors in 1985, was 16Mb. This was not enough to fit the space required, so paging was

[1] This is an edited version of an invited presentation, delivered on June 27, 1996 at the *Advances in Computer Chess 8* conference in Maastricht, The Netherlands. The Editors thank Ken Thompson for scrutinizing their rendition. The article will be published in extended form in the Proceedings of the ACC8 conference.

[2] Bell Laboratories, 700 Mountain Ave., Room 2C-519, Murray Hill, New Jersey 07974-0636, USA.
Email: ken@plan9.bell-labs.com.

necessary. Today the memory size is 250 Mb, also not large enough, and hence paging is still required. Yet, the memory has been extended by a ratio of about 16. The size of the disks also has crept up with the development of the technology. The only thing that is really pathetic is that the I/O rates (as expressed in Mbps, megabytes per second) now are only roughly 10 times the I/O rates then. Of course, you can obtain huge I/O rates on some special machines, but at least I cannot afford them. In summary, all those numbers lead to such machines as specified in Table 1.

	1985	1996	ratio
cpu	1 MIP	150 MIP	150
mem	16 Mb	250 Mb	16
disk	500 Mb	50 Gb	100
I/O	1 Mbps	10 Mbps	10

Table 1: Specification of machines used for building 5-piece and 6-piece endgame databases.

3. OUTLINE OF THE ALGORITHM

The algorithm for building 6-piece endgames uses a representation of a chess position on the chess-board and converts it to a unique number. So the conversion is from chess position to number; the number is used to index a table of all such chess positions. In particular, a set of positions is grouped as a bitmap. For instance, 10 positions give a very large bitmap with 10 bits set to 1. Moreover, it is necessary to be able to convert from a chess position to a number and *vice versa*, and to *do* (make) moves and to *undo* (retract) moves in a chess position, i.e., to do the required set manipulations in the bitmap representation. For 6-piece endgames we have the following. First, there are 462 combinations of placing the two Kings, taking into account reflections and rotations and legal positions. Then we consider the chess-piece combinations that are left: there are four more pieces and each one can be on 64 squares. Again, there are some illegitimate positions, but it is too hard to remove them and it is not worth doing the effort. This results in 64^4 (16M) piece combinations. Thus, exhausting all 6-piece positions, for one configuration of pieces, it combines to 7.75×10^9 (or roughly 8G) possible positions. If each position is stored as a single bit, some 970 Mbytes of memory are needed for one of these bitmaps, i.e., one set only.

The algorithm is published in Thompson (1986) and is fairly straightforward. We start with a large bitmap of all ~8G positions, with many zero-bits and a couple of 1-bits in it, representing all the positions with Black-to-move-and-lose in n moves. The idea is illustrated in Figure 1. Then we apply a mapping function on all the 1-bits, converting them into chess positions; for the chess positions we generate unmoves. Thereafter the resulting chess positions are converted back into indices and the corresponding bits for the unmoves are set. This new set of bits now represents positions with White-to-move-and-win in n+1 moves. Subsequently, we do a series of these manipulations, *ad nauseum*, until no more unmoves are generated. Everything not touched (i.e., left zero) is not a win; it represents either a draw or a loss, or an illegal position. This roughly describes the algorithm. Figure 2 provides more details.

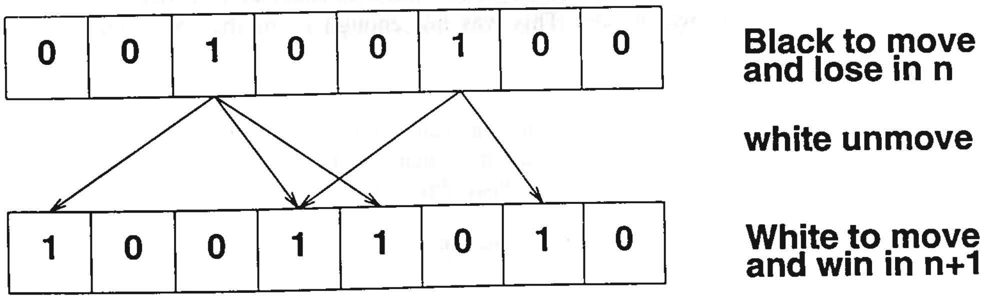

Figure 1: An example bitmap conversion.

```
1. initial mates over all positions

   B = btmmate(~0)
   BTM = 0
   WTM = 0
   n = 0

2. record last pass

   BTM |= B
   RES |= res(B, n)
   n += 1

3. White to move and win in n

   W = wunmove(B) & ~WTM
   WTM |= W

4. Black to move and might lose in n

   X = bunmove(W) & ~BTM

5. verify Black to move and lose in n

   B = bmove(X) => WTM
   goto 2
```

Figure 2: The algorithm.

In Figure 2, B is an 8G bitmap (see also below). In the algorithm, all strings with capital letters denote an 8G bitmap. BTM and WTM are so-called summation bitmaps, with the former including all Black-to-move-and-lose positions, and the latter all White-to-move-and-win positions, initially both having no bits set. The variable n is just a counter. Moreover, '~0' denotes a vector of all ones, i.e., all bits are set ('~' being the one-complement operation in the language C). Further '|=' is C's assignment operator for bitwise inclusive-OR, e.g., 'BTM |= B' is shorthand for 'BTM = BTM | B', denoting bitwise ORing BTM with B. Finally, '&' is C's bitwise AND operator.

In the first step every bit is checked to see if it represents a position with Black mated. Next, we have the steps 2 to 5. In step 2, we just record the last pass, ORing the mate positions in the summation bitmap and keeping a record of when (at what iteration) positions are set in the vector RES. We increment n. In step 3, we back up one move as illustrated in Figure 1 and arrive at a set of White-to-move-and-win positions. We AND these positions with the White-to-move positions not set so far, because you only want to investigate unknown positions, and store these in the summation map. In step 4, we back-up one more move, arriving at positions with Black to move and possibly lose in *n*; of course, again only taking into consideration still unknown positions. But Black does not want to lose, so in step 5 we have to verify that those positions are *forced* losses; we do that by investigating if any black move from such a position leads to a White-to-move-and-win position.

This algorithm requires the least amount of work. However, the I/O is a problem in this work: the machine speed has gone up by a factor of 150, but the I/O has just gone up by a factor of 10. So we really want to avoid the I/O, even more than we want to avoid computing. The key is that in this algorithm, in step 3, we go linearly through the B. For every chess position set we back-up one move, which accesses the W map *randomly*. And of course, random access jumps around, fetching different blocks at different times, all causing I/O. We cannot avoid I/O, but what you *can* avoid is to fetch randomly the WTM vector.

Instead of unmoving a Black-to-move position and fetching at random the WTM vector and ANDing it to decide if we are going to set a bit in the W vector or not, we *should* delay the AND operation to the next step. Hence, in step 3 we set more bits in the W vector, but that only costs some computation and we AND the W vector with the unknown position vector in step 4. There we only do it once, saving considerabe I/O. Of course, we should analogously delay the ANDing for the black unmoves (step 4 of Figure 2). The enhanced algorithm is given in Figure 3. The enhancement decreases the I/O drastically and should be done.

```
1. initial mates over all positions

   B = btmmate(~0)
   BTM = 0
   WTM = 0
   n = 0

2. record last pass

   BTM |= B
   RES |= res(B, n)
   n += 1

3. White to move and win in n

   W = wunmove(B)
   WTM |= W

4. Black to move and might lose in n

   X = bunmove(W & ~WTM)

5. verify Black to move and lose in n

   B = bmove(X & ~BTM) => WTM
   goto 2
```

Figure 3: The I/O-enhanced algorithm.

4. TWO ALTERNATE ALGORITHMS

There are two alternate algorithms to be considered. The first algorithm is essentially the CHINOOK algorithm (Lake, Schaeffer, and Lu, 1994). It requires 1 or 2 bits per position. The idea is as follows. You just walk over this bit; for every position that is not set you ask whether it is a mate or directly leads to a mate. If positive, you set the bit and you just go on. You repeat this sequential walk until nothing more happens. But you can walk over yourself. This means you can set a bit at one pass and later on you actually read that bit and decide that it is a mate. So you do not end with a nice clean set of positions with indications of win/lose in *n*, *n*+1, *n*+2, etc. In fact you do not even get an answer that you can actually use to play perfect chess to win a game. All you get is a set of vectors that indicate whether the game is won or not. The advantages are that it is real cheap (you have only 2 bits instead of 10 or 11, depending on how you count), it is fast and it is partitionable. The disadvantage, which from my point of view is crucial for chess, is that there is no result. There is no RES vector; there is no way to generate best play out of it; all there is, is a set of positions to win.

On the second alternate algorithm I spent a month, but I am embarrassed to say that it does not work (but I thought I will warn you). Consider a position in the RES vector recording the results. Before you actually find the answer to be stored you can use that position for something else. You can use it for the number of unproven White-to-move-and-win positions the position leads to. So any time you backup from a White-to-move-and-win position (step 4 of the algorithm), you do not have to go again forward and verify if the black position indeed is a loss. This essentially eliminates step 5 in the algorithm. Instead you just decrement the counter of unproven positions by one. As soon as the counter reaches zero it is known that the Black-to-move position indeed is a *forced* loss. This counter of unproven positions uses the same slot reserved for storing the result, since there is no result yet.

The advantages are that it is cheap, it takes the size of the RES vector (i.e., 8 bits) + 2 bits per position; 1 bit indicating whether the corresponding slot in the RES vector contains a result or contains a number of unproven positions and the other one being the W vector. Further, it is one step less per pass, and therefore faster. But an important fact is, again going back to the I/O issue, when you back-up the position instead of fetching a bit, you now have to fetch 10 bits, a factor of 10 in I/O. So, the major disadvantage is that the I/O rate goes up by an order of magnitude. Another disadvantage is the following: because of the transformation from chess position to number and back, the number of moves out of the position is not the same as the number of moves into the position, as a result of symmetries and transformations. So you must do an initialization pass to set the entries initially to the number of unmoves in the position instead of setting them to the number of moves out of the position; this requires an additional pass, on every position on the whole board. That takes about three days of computer time on my machine, just doing the initialization. A third disadvantage, actually also crucial for this algorithm, is that you cannot restart. If you interrupt, you have to start all over again, since you do not know by which amount the counters were decremented halfway during a pass. All in all, this was almost an algorithm. Unfortunately, it does not work.

5. THE ALGORITHM IN DETAIL

Below the algorithm of Section 3 (cf. Figure 2) is described in more detail, using C code. The board position is not represented as a 64-byte vector, since it would be a waste of space, dealing with six pieces only. Instead, it is a 6-byte vector indicating on which square each piece is. Each piece is uniquely known, piece 0 being the white King, piece 1 the black King, etc.

The next issue is the conversion of a chess position to and from a double-precision offset in the bitrange. The structure definitions are given in Figure 4. `KH` enumerates the number of King-King positions, whereas `PH` enumerates the number of 4-pure-piece configurations.

```
typedef
struct Posn
{
    char p[NPIECE];
} Posn;

typedef
struct Gird
{
    int kgird;
    long pgird;
} Gird;

enum
{
    KH = 462,
    PH = 1L << (6* (NPIECE-2))
};
```

Figure 4: Structure definitions of 6-piece endgames.

The core of the 6-piece-endgame database construction is a program I wrote, called `mksub`, generating all the specific move and unmove generators needed for a particular configuration of the 6 pieces. It is all done automatically and all the special cases are done once by the program. `mksub` generates all the routines necessary to run that particular configuration of pieces. Subsequently, the appropriate program is compiled and generates all the routines needed. Part of the output is given in Figure 5 for the 4-piece endgame KQKR.

The program `mksub` generates among other things that the number of pieces is 4, and that the pieces are WK, BK, WQ and BR. Then it puts them in canonical order. Thereafter it actually writes the code for six functions. The white and black functions are symmetric, so I describe only one side. The `wattack` function is a Boolean function that notices whether White attacks a square. It is used to find out whether Black is in check, by investigating if White attacks the position of the black King. Then there are the `wmove` and `wumove` functions. These are used as follows. Assume a given board position and a routine, say `wmove`, with a function as an argument is called. This routine modifies the board according to each of the legal moves and then calls the function with the board modified each time. So if there are 10 legal moves, the function argument that you pass to `wmove` will be called 10 times, once for each move. The function `wumove` performs likewise, for the positions that result from White retracting a move.

```
mksub q.r >sub.c
cc -c sub.c

#define NPIECE  4
char*   pname[] =
{
    "WK",
    "BK",
    "WQ",
    "BR",
};

extern  int wattack(int);
extern  int wmove(int (*) (void));
extern  int wumove(int (*) (void));

extern  int battack(int);
extern  int bmove(int (*) (void));
extern  int bumove(int (*) (void));
```

Figure 5: The move-and-unmove-generator generator.

Figure 6 presents the outline of the code for the black-unmove function, bumove. In the KQKR example, it is just a series of unmove King and unmove Rook. The function parameter, `(*f) (void)`, is called for each new board configuration, i.e., in bumove, on each board position after an unmove has been performed. If the function, `(*f)`, returns non-zero, then bumove is aborted at that point and bumove returns non-zero. If there is no abort and bumove executes to completion, then bumove returns a zero. The variables `fr, to, e, el,` and `l` are just temporary variables for the code fragments for each piece type.

```
int bumove(int (*f)(void))
{
    int fr, to, e, el;
    char *l;
    /* posn.p[1] is bk */
    ...
/* black-King-unmove code, see Figure 7 */

    /* posn.p[3] is br */
    if(posn.p[3] != GONE)   {
    ...
/* black-Rook-unmove code, see Figure 8 */
    }
    return 0;
}
```

Figure 6: Outline of the black-unmove function.

The black-King-unmove code is presented in Figure 7 and the black-Rook-unmove code in Figure 8. I will not go into much detail, but let the code mainly speak for itself. The array, edge, is a set of 8 bits for each of the 64 chess squares. There is a bit to tell if this square is within 1 or 2 squares of each of the four edges of the board. A move is tested by ANDing a set of edge bits (i.e., UP2 | RIGHT1 for a Knight) prior to adding in an offset. The moves of Knights and Kings are identical except for the tables. The moves for the sliding pieces are the same except the test and offset is repeated until it fails. Thus, the moves of Bishops, Rooks and Queens are identical except for edge/offset tables. The functions just generate specific code for the constellation of the pieces on the board and are quite efficient, much more efficient than general code.

```
fr = posn.p[1];
e = edge[fr];
for(l=ktab; el=l[0]; l+=2)
while(!(e & el)) {
    to = fr + l[1];
    if(to==posn.p[2]||to==posn.p[3]) break;
    posn.p[1] = to;
    if(!battack(posn.p[0]))
    if((*f)()) {
        posn.p[1] = fr;
        return 1;
    }

    /* posn.p[2] is wq */
    if(posn.p[2] == GONE)  {
        posn.p[2] = fr;
        if(!battack(posn.p[0]))
        if((*f)()) {
            posn.p[2] = GONE;
            posn.p[1] = fr;
            return 1;
        }
        posn.p[2] = GONE;
    }
    break;
}
posn.p[1] = fr;
```

Figure 7: The black-King-unmove code.

```
fr = posn.p[3];
for(l=rtab; el=l[0]; l+=2)
for(to=fr; !(edge[to]&el);)  {
    to += l[1];
    if(to==posn.p[0]||to==posn.p[1]||to==posn.p[2]) break;
    posn.p[3] = to;
    if(!battack(posn.p[0]))
    if((*f)()) {
        posn.p[3] = fr;
        return 1;
    }

    /* posn.p[2] is wq */
    if(posn.p[2] == GONE) {
        posn.p[2] = fr;
        if(!battack(posn.p[0]))
        if((*f)()) {
            posn.p[2] = GONE;
            posn.p[3] = fr;
            return 1;
        }
        posn.p[2] = GONE;
    }
}
posn.p[3] = fr;
```

Figure 8: The black-Rook-unmove code.

After this preparatory work we now arrive at the whole program. Having generated the move and unmove generators and having compiled them together with the `attack` functions, all initial mates are determined (step 1 of the main algorithm of Figure 3). Figure 9 provides the code. The program loops through all the king positions, loops through all the piece positions, converts them to piece types, skips illegal positions, skips positions with Black not in check, skips positions with White in check, and then calls `bmove` with the generation function `hpost`. If `hpost` is ever called, then there is at least one black move and therefore Black is not mated. The function `hpost` will return non-zero and `bmove` will abort and also return non-zero, and the position is not marked as a mate. If `hpost` is never called, then `bmove` will return zero and the position is a (black) mate. Thus, what remains are BTM mate positions; hence, the corresponding bits are set. The function `bitset` is an interface to the random I/O routines. It asks if the double-precision offset, `g`, into the bitmap/file, `B`, is set. If the bit is set, `bitset` returns non-zero, if the bit is not set, it sets the bit and returns zero.

```
/* find initial btm mates */

void main(void)
{
    Gird g;

    B = ioinit("B", SEQ|OUT);

    for(g.kgird=0; g.kgird<KH; g.kgird++)
    for(g.pgird=0; g.pgird<PH; g.pgird++) {
        gird = g;
        convg2p();
        if(illeg()) continue;
        if(!wattack(posn.p[1])) continue;
        if(battack(posn.p[0])) continue;
        if(bmove(hpost)) continue;
        bitset(B, &g);
    }
}

int hpost(void)
{
    return 1;
}
```

Figure 9: The main algorithm step 1: finding initial BTM mates.

Of course, this is parallelizable. We only divide the main for loop into little pieces; there is no random access (cf. also Stiller, 1989).

The second step of the main algorithm of Figure 3 is just some bookkeeping; no special explanation is needed.

```
/* white unmove */

void main(void)
{
    B = ioinit("B", SEQ|IN);
    W = ioinit("W", RAND|OUT);

    for(;;) {
        if(getseq(B, &gird)) break;
        convg2p();
        wumove(hpost);
    }
}

int hpost(void)
{
    convp2g();
    bitset(W, &gird);
    return 0;
}
```

Figure 10: The main algorithm step 3: White-to-move-and-win in *n*.

Figure 10 provides the code of the third step, which fetches a previous B vector; it does a White unmove on B, and sets a W vector. We provide a brief explanation. We have a B vector and a W vector, where the first one is sequentially accessed and the second one randomly, which is very important. Random means we take all the memory and make a big hash table out of it. The latter type of access occurs only once per pass. Until vector B is exhausted we take sequentially the next position out of vector B, and we convert it to a piece vector and call wumove with a routine that simply converts it back into a bit vector, sets the bit and returns 0. The function getseq is an interface into the sequential I/O routines. It will search for the next set bit in the file, B, and set the double-precision offset of that bit in the argument, gird. The return value of getseq is an end-of-file indication.

The routines `convg2p` and `convp2g` will convert between board representations and offset representations and vice versa. Board manipulations, such as unmove, are best done with piece/location representation while set manipulations, such as AND/OR, are best done in offset representation.

Figure 11 shows the code of step 4 of Figure 3. It is exactly the same as the previous step, except that the `X` vector is set and black unmoves are done instead of white unmoves.

```
/* black unmove */
void main(void)
{
    W = ioinit("W", SEQ³IN);
    WTM = ioinit("WTM", SEQ³IN);
    X = ioinit("X", RAND³OUT);

    for(;;)  {
        if(getseq(W, &gird)) break;
        if(bittst(WTM, &gird)) continue;
        convg2p();
        bumove(hpost);
    }
}

int hpost(void)
{
    convp2g();
    bitset(X, &gird);
    return 0;
}
```

Figure 11: The main algorithm step 4: Black-to-move-and-might-lose in *n*.

```
/* black move */

void main(void)
{
    Gird g;

    X = ioinit("X", SEQ|IN);
    BTM = ioinit("BTM, RAND|IN);
    B = ioinit("B", SEQ|OUT);

    for(;;) {
        if(getnum(X, &g)) break;
        if(bittst(BTM, &g) continue;
        gird = g;
        convg2p();
        if(!bmove(force)) bitset(B, &g);
    }
}

int
force(void)
{
    convp2g();
    if(bittst(WTM, &gird)) return 0;
    return 1;
}
```

Figure 12: The main algorithm step 5: verify Black-to-move-and-lose in *n*.

Finally, we have the last step in Figure 12. We read the first vector out of X, which is done sequentially, and test it for a new Black-to-move position and return if it is not the case. We convert the new Black-to-move position to a piece vector and, for every possible black move, call the function `force`, which checks whether the position is a win (returning a 0), or not (returning a 1). If they all return 0, it is a forced loss for Black and the corresponding bit is set, but if anyone of these is 1, it is no forced loss for Black. Essentially, that is all.

6. DOING IT IN WAVES

In Section 5 I have described the whole program; there is nothing more. Now reality sets in. The program allows us to find the terminal mates, to back them up, and to find how to force the mates. But then the question of the objective to be reached arises. It has been posed over and over in the literature, mostly as follows: "is it really the objective to play to mate?" The answer is "No!". We typically want to quit on the first capture. Certainly, that is the 50-move rule definition. Thus, we want to do the mating procedure in waves. I use the variable *w* for wave number. It is the variable indicating the number of pieces of solved subsets of positions. Hence, w=3 means all subsets of 3 pieces have been solved and once done we want to use the results as the basis for a winning set of moves. Then we do the whole program again for 4 pieces, i.e., for *w* set to 4. Then *w* is set to 5, and finally *w* is set to 6. Hopefully, if the machine is fast enough the w = 3 to 5 is quickly solved. The program knowing the value of *w* will only generate moves and positions for the relevant subsets of the *w* number of pieces. So if w = 3, the program creates the initial B exactly as it did before and it solves this problem: the problem is restricted to three pieces. For the KQKR endgame it first solves the KQK endgame. Then *w* is set to 4, and the program continues in a way to be described below, such that the program also handles Q and R moves. When the Rook is captured, the program uses the KQK subset. Of course, when going from four to five and then to six pieces, the program just builds on each of those subsets. Thus, we do the construction in waves. Between the waves we just increment the *w* variable. What we essentially use is the combination of the new Black-to-move-and-mate positions for *w* pieces plus all the Black-to-move-and-lose positions from the previous wave. We then set the summation vectors BTM and WTM to zero and start it all over again. Instead of the original initialization step, before each wave we now use the initialization step sketched in Figure 13.

```
w += 1
B = btmmate(~0)
B |= BTM
BTM = 0
WTM = 0
n = 0
```

Figure 13: Outline of the new wave-initialization step.

All this is fine and intuitive, except that there is a counterexample (see Diagram 1). This is a Black-to-move-and-lose position. But now Black forced the conversion to the previous wave, by being made to play 24. ... Kxg6.

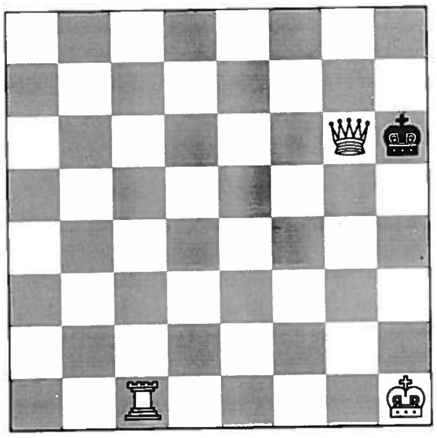

Diagram 1: Position after 24. Qg6+.

The position from Diagram 1 does not enter the set of B. It is *not* in the previous BTM, because they only contain wave-3 positions, and this is a wave-4 position **and** it is not a mate. The position resulting after 24. ... Kxg6 also is not in B though, since only Black-to-move positions were added from the vector BTM, and not White-to-move positions. What we must do is: add the mates from BTM, **and** then we must do one white unmove to pick up the positions where Black causes the conversion. Hence, instead of doing between waves the simple initialization step sketched in Figure 13, we must do the enhanced step from Figure 14 between each of the waves. If we do not do so, we will have a few positions – though I am not sure that they really matter – that mess up our counts. They may in fact back up into giving bad answers. Though they are pretty esoteric, we have to take them into account and we actually thus have to add one white-unmove pass.

```
w += 1
B = btmmate(~0)
B |= BTM
X = bunmove(WTM)
B |= bmove(X)  => WTM
BTM = 0
WTM = 0
n = 0
```

Figure 14: Outline of the enhanced wave-initialization step.

There is also a problem here in that for these new positions the count is one ply different from the normal positions. See again Diagram 1. Typically Black-to-move-and-lose means Black is going to make a move and then White is going to win. Here, Black is going to make the move and the conversion is one ply different. There should be a unified version of this type of position for the counts in the literature in order to compare if everyone arrives at the same answer. Unfortunately, there is not a unified version. I count this the same as if White captures on this move instead of Black captures on this move, which results in a one-ply difference. I guess, in some sense, it really does not matter much.

7. THE CURRENT STATUS

For about three months now (27 June, 1996) I have been working on KRBKBN. I spent about a month and a half using the algorithm that did not work, the second alternate algorithm described in Section 4. That was very hard, because you do not want to recognise that it was a bad idea. It looks as though the current version of the program is actually generating correct answers. It has been running for about one month and a half. I thought it would been done in time for this talk. When I left it was on pass 70 of wave 6. Back in New Jersey, I found that the KRBKBN database was finished at the day of the conference. So, it took about three months from start to finish. The maximin (max-to-win) agrees with Stiller's (1995), being 98 moves. An example is White: Ka8, Rb6, Ba1; Black: Kd7, Bh7, Ne7; WTM. At the moment, the BTM-lose position counts are not double checked. They will be published in *Advances in Computer Chess 8* (Van den Herik and Uiterwijk, 1997). I tried a few times to compress the outcome. The results were disappointing. As of now, I have no idea how to compress them into something portable. Maybe I have to use the new CD types, the video CDs. Certainly in the next 5 years such new types of CDs will come out. Anyway, it will be made available with access to something like Java or whatever.

8. CONCLUSION

Finally let me mention some previous work in 6-piece endgames. Stiller (1991, 1992) ran many 6-piece endgames on the CM-5 machine, but he could not save the answers; so essentially only summarizing results are available, like max-to-wins. He then ran the KRBKNN endgame, on a machine in Los Alamos, I believe, making use of the Knight-Knight symmetry, thereby reducing the required memory to half the size of a normal 6-piece endgame. He saved the result, then announced his retirement, but I do not believe it. The endgame has a 223 move maximin. Together with some more information it is shown at the following WEB site: `http://symmetry.xo.com`.

9. REFERENCES

Herik, H.J. van den and Herschberg, I.S. (1985). The Construction of an Omniscient Endgame Data Base. *ICCA Journal*, Vol. 8, No. 2, pp. 66-87.

Herik, H.J. van den and Herschberg, I.S. (1986). A Data Base on Data Bases. *ICCA Journal*, Vol. 9, No. 1, pp. 29-34.

Herik, H.J. van den and Uiterwijk, J.W.H.M. (eds.) (1997). *Advances in Computer Chess 8*. Universiteit Maastricht, Maastricht, The Netherlands. (In preparation).

Lake, R., Schaeffer, J. and Lu, P. (1994). Solving Large Retrograde-Analysis Problems Using a Network of Workstations. *Advances in Computer Chess 7* (eds. H.J. van den Herik, I.S. Herschberg, and J.W.H.M. Uiterwijk), pp. 135-162. University of Limburg, Maastricht, The Netherlands. ISBN 90-6216-1014.

Roycroft, A.J. (1984). Two Bishops against Knight. *EG*, Vol. 5, No. 75, pp. 249-253.

Stiller, L.B. (1989). Parallel Analysis of Certain Endgames. *ICCA Journal*, Vol. 12, No. 2, pp. 55-64.

Stiller, L.B. (1991). Some Results from a Massively Parallel Retrograde Analysis. *ICCA Journal*, Vol. 14, No. 3, pp. 129-134.

Stiller, L.B. (1992). KQNKRR. *ICCA Journal*, Vol. 15, No. 1, pp. 16-18.

Stiller, L.B. (1995). *Exploiting Symmetry of Parallel Architectures*. Ph.D. thesis. The John Hopkins University, Baltimore, Maryland.

The Editors (1992). Thompson: All About Five Men. *ICCA Journal*, Vol. 15, No. 3, pp. 140-143.

The Editors (1993). Thompson: Quintets with Variations. *ICCA Journal*, Vol. 16, No. 2, pp. 86-90.

Thompson, K. (1986). Retrograde Analysis of Certain Endgames. *ICCA Journal*, Vol. 9, No. 3, pp. 131-139.

Thompson, K. (1991). Chess Endgames Vol. 1. *ICCA Journal*, Vol. 14, No. 1, p. 22.