Chess program board representations

Note this is a draft, and that changes are likely.

Introduction

The first decision we make when we undertake writing a chess program is how to represent the chess board and the pieces that occupy squares on this board. Unfortunately, this decision is made before there is enough information available to choose the optimal representation. As a result, this turns out to be the single most often changed data structure in the entire program. As we work our way through generating moves, evaluating positions, attack detection and other such considerations, we soon discover the shortcomings of our representation, and then re-design this data structure to make the operations more efficient.

The purpose of this paper is to explain several ways of representing the chess board, so that it becomes possible to make an informed decision before stumbling over some unforseen drawback later on.

There are two distinctly different board representations in use today in current chess programs. The most common data structure is generally referred to as the offset (or array) board representation, which was the data structure mentioned by Shannon when he first described how to program a computer to play chess back in the early 1950's [check this date and give a citation also]. A newer approach uses a set of bit-vectors (also called bitmaps or bitboards) to represent the chess board. Each of these will be explained in detail, and both approaches have advantages and disadvantages that will be discusses.

Offset board representation

In the offset board representation, the chess board is representated by an array of elements with each square of the chess board mapping to one element of the array. In this approach, there are still several alternative ways to map the squares of the chess board to elements of the array.

Two-dimensional array representation

The first approach that comes to mind is a simple two-dimensional array, that I'll call board[8][8] (and I'll use ANSI C syntax exclusively when giving coding examples, as opposed to C++ or another programming language.) In this approach, we can map square a1 to board[0][0], h1 to board[0][7], a8 to board[7][0] and finally h8 to board[7][7]. This gives a simple to understand one-to-one mapping from chess board squares to elements of the array, and at first blush, this looks like a reasonable approach.

However, when we get to move generation, we immediately run into problems of efficiency that are not apparent at first. The first problem that even a beginning programmer should recognize is that to access any square of the chess board, it is necessary to take the rank, multiply by 8 and then add in the file, since all arrays are stored as linear lists of elements by the actual computer hardware. We'll look into how to use such an array to generate moves in a future paper. For now, notice that any reference to a square board[rank][file] requires that the compiler produce code to compute rank*8+file and then use this as an offset from the first element of the array (hence the name offset board representation). Even if the compiler is smart enough to turn the multiply by eight into a left shift 3 bits (in C, rank*8 == rank<<3) it still takes a couple of instructions.

To step down a diagonal, for example, starting from square e4 (rank=3, file=4 since arrays are indexed from 0-7 in C), we simply add one to the rank and file, then do the multiply/add and access that board element. For the square e4, moving toward h7, we get [3][4], [4][5], [5][6], and [6][7]. Of course we need to be careful and not access an element that is "off of the board" so each time we modify either rank or file, we have to make sure that the resulting value is >=0 and <=7. Of course, if we are adding to the rank or file, we only have to test for > 7, but one-half of the time, we are moving in the reverse direction and have to test for >= 0. This is one drawback of this particular offset representation. The number of times this test is repeated becomes a significant part of the loop. There are alternative ways to do the test, but it is always going to slow us down.

One-dimensional board representation

The first efficiency "hack" a serious chess programmer tries is to eliminate the multiply/add required for each square, by moving to a one-dimensional array, board[64]. Here we'll map a1 to board[0], h1 to board[7], a8 to board[56], and finally h8 to board[63]. What have we gained? First, to move down a diagonal, say in the direction from e4 toward h7, we simply add the constant 9, so we get [28], [37], [46] and [55]. Note that by adding 9, we save the multiply operation.

This approach solves the multiply/add or shift/add problem, which makes this faster, but we still have the same problem about testing each newly generated square number to make sure we don't step off the edge of the board in any direction. One example would be a bishop on square f1, which translates to [5] in the current notation. Stepping down the same diagonal direction will produce [5], [14], [23], [32] and without going any further we already see a problem in that we have gone from f1 to g2 to h3 to a5 (a5=32 in this notation.) Notice that this let us "wrap" from one side of the board to another, so we have to guard against this with the same rank/file test as before, so we only save the multiply/add, but still have to tolerate the conditional tests, which on modern RISC computers is far worse than the multiply/add we eliminated. Net gain so far, then, is negligable.

The one-dimensional array representation looks better, but when we implement it as given, it really saves very little since a good super-scalar processor will execute the multiply/add while waiting on the branch to complete. We now turn to trying to eliminate the tests for "edge of the board" to make this more efficient.

One dimensional board representation with border squares

The next step in board representaton evolution is to enclose the board inside a larger array, so that illegal squares are "off" the edge and are easily detectable.

Early chess programs typically used an array of 120 elements to represent the 64 valid board squares, plus a 2-square "fringe" or "border" around the valid set of board squares. Why do we need a border that is two squares wide? If you think about the knight, which moves either two ranks or two files in one move, the reason becomes obvious. For sliding pieces, a one-square border would suffice, but for the knight, we need two. In this representation, we'll map a1 to board[21], h1 to board[28], a8 to board[91] and h8 to board[98].

The efficency trick here is that suppose we choose to represent valid chess pieces by pawn=1, knight=2, bishop=3, rook=4, queen=5 and king=6, with positive values representing white pieces, negative values representing black pieces and a value of 0 representing empty squares. We will simply extend this definition so that a value of (say) 99 represents an illegal chess board square. If we initialize all the squares that are not part of the valid set of board squares to 99, we can detect the end of a rank, file or diagonal quite easily. In the above example, starting at square f1 [26], we simply increment by 11 to step down the same diagonal, and

produce the squares [37], [48] and [59]. When we access board[37] and board[48] we either get 0, or a valid piece code since those are valid board squares, but when we access board[59] we find our "invalid square" code of 99, and with one test we discover that we have reached the end of that direction and can quit.

This approach was used in the very early versions of Blitz/Cray Blitz before an even more efficient approach was discovered. The advantage of this approach is that a single test (board[n]==99) detects the end of a rank, file or diagonal in a simple manner. This also works for knights, because knights near the edge of the board will produce some moves that "jump off the edge" of the board.

One question you might ask is this representation has 12 ranks, which gives two ranks before the real board and two after the real board, as you've explained, but you only have one extra file on each side of the board. Why? When you study this, you will notice that the right-most "illegal" file is adjacent to the left-most illegal file. While we could have set the board up as a 12x12 array, it wouldn't really gain anything. We are left with using 120 words to store an 8x8 chess board. We have eliminated the multiply/add address calculation, and now we have reduced the illegal-square test to a single test from the two tests required for the first two approaches. The question is, can we eliminate anything else?

0X88 board representation

This representation has also been used for a long time, and is based on older computer systems that had little or no cache memory, which made memory accesses expensive operations. If you followed the last representation discussion, you know that to compute a new valid square from a prior valid square, we simply add some constant representing the direction we are interested in. We fetch the value of the board for that square and if the square is not legal, we can determine this by comparing the value we fetched to see if it is 99. If so, we can go no further in that direction. however, on many machines, this memory reference is expensive, and it is returning a value that we really do not need, except as an indication of the illegality of the new square value we just computed.

The way to eliminate this extra memory reference is to modify our board layout even further. We are now going to use an array board[128] to represent the chess board. We map a1 to board[0], h1 to board[7], a8 to board[112] and h8 to board[119]. If you visualize this, you will see that the board looks like this:

```
                                  |
112 113 114 115 116 117 118 119 | 120 121 122 123 124 125 126 127
 96  97  98  99 100 101 102 103 | 104 105 106 107 108 109 110 111
 80  81  82  83  84  85  86  87 |  88  89  90  91  92  93  94  95
 64  65  66  67  68  69  70  71 |  72  73  74  75  76  77  78  79
 48  49  50  51  52  53  54  55 |  56  57  58  59  60  61  62  63
 32  33  34  35  36  37  38  39 |  40  41  42  43  44  45  46  47
 16  17  18  19  20  21  22  23 |  24  25  26  27  28  29  30  31
  0   1   2   3   4   5   6   7 |   8   9  10  11  12  13  14  15
                                  |
```

Squares on the left half of this diagram represent the chess board, while squares on the right half are illegal. This isn't quite as revealing as when we look at this same mapping, but display the square numbers in a base-16 format as follows:

```
70 71 72 73 74 75 76 77 | 78 79 7a 7b 7c 7d 7e 7f
60 61 62 63 64 65 66 67 | 68 69 6a 6b 6c 6d 6e 6f
50 51 52 53 54 55 56 57 | 58 59 5a 5b 5c 5d 5e 5f
40 41 42 43 44 45 46 47 | 48 49 4a 4b 4c 4d 4e 4f
30 31 32 33 34 35 36 37 | 38 39 3a 3b 3c 3d 3e 3f
20 21 22 23 24 25 26 27 | 28 29 2a 2b 2c 2d 2e 2f
```

```
 10 11 12 13 14 15 16 17 | 18 19 1a 1b 1c 1d 1e 1f
  0  1  2  3  4  5  6  7 |  8  9  a  b  c  d  e  f
```

If you study this, you begin to notice one important feature of laying the board out like this. We are using 4 bits for the file and 4 bits for the rank, but notice that for valid squares, the high order bit of the 4 bit value is always 0. The way we use this is as follows. Again, take a bishop on square f1 [5]. To generate the same set of squares, we now add 17 to each sucessive square and get [22], [39] and [56] where we quit. We know that [56] is illegal because if we and it with 0X88 (a base-16 constant 88) and get a non-zero value, we just proved that this is an invalid square. [56] is [38] in base 16, and clearly anding this with 0X88 will produce a non-zero result. Notice, also, that we don't need to make the size of this array 256 words. Due to the 0X88 trick, we will never access the upper half of the array because all those squares have the high-order bit (0X80) set and would be caught as illegal before the memory reference is made. Of course, if memory is a concern, the right-half of the board is never accessed either and can be used to store anything you might choose.

Now why is this faster than the previous example? If we analyze the necessary assembly instructions, we find that both approaches have to add some integer constant to the previous square to produce a new square. Since we have this value handy (in a register) the and with 0X88 takes one machine cycle, rather than having to go to memory to fetch the value before we can determine that we have reached an illegal square.

Before going on, I should point out that this was used in very early versions of Cray Blitz, but we later went back to the previously described representation because a vector machine can often pipeline the memory references and actually executes faster using the non-0X88 representation. As microprocessors move into the world of vector processing, this trade-off can be important.

With that said, it seems that 0X88 is an improvement, albeit a modest one. It turns out that as we get further into implementing our chess program, this representation is going to offer other advantages. One simple example is that if the distance between any to real squares on the board is an exact multiple of any of our normal distance increments (17 in the bishop example) then we know the two squares are on the same ray, since the distance increment will either step from legal square to legal square, or else will step into the 0X88 part of the board which is illegal. As a result, the "wrap-around" issue is never going to be a problem. If you try this on the earlier representations that we covered, this will fail, because there are many squares that are separated by a multiple of our distance increment, but which are not on the same ray. For example, in the second approach which used an array of 64 words, we discover that squares 48 and 57 (a7 and b8) are a multiple of our increment which is 9. however, 39 is also a multiple of this distance yet it is on the other side of the board. The 0X88 approach eliminates this confusing problem.

[more here]

Representing pieces

The next decision we face is how the pieces are going to be represented on the chess board.

There are at least two ways to represent pieces. The most obvious is to use pawn=1, knight=2, bishop=3, rook=4, queen=5 and king=6, and use positive numbers for white and negative numbers for black. One minor caution here. Avoid using +=computer, -=opponent. This seems like a workable solution, but you will eventually want to execute tree searches for both sides, and since the search starts at ply=1, it is easy to fall into the above trap of using +=computer. However, this will make it difficult to use the search in a general way. It is simply much more understandable to let +=white and -=black and not identify pieces as "machine" or "human" in order to make the programming more general.

One alternative to this that we used in later versions of Cray Blitzs and in Crafty as well, is to represent pieces like this: pawn=1, knight=2, king=3, bishop=5, rook=6 and queen=7. The advantage of this is that if you AND the piece value with 4, and get a non-zero result, you know that is a sliding piece. If the result of the AND is zero, this is not a sliding piece. Next, if this is a sliding piece and you AND the piece value with a 1, and the result is non-zero, you have a piece that can slide diagonallyl if you AND the piece value with a 2 and the result is non-zero, you have a piece that slides vertically/horizontally like a rook. Note that the queen has both of these bits set and that it can slide in all 8 directions. This may or may not be useful in your program, but since it still only requires 3 bits to represent a piece, there is nothing to lose by using this.

One minor issue is the decision to use +/- to represent white/black pieces. This seems rather obvious at first, but it might result in an efficiency problem that goes unnoticed. When generating moves later, we will need to the value of the captured piece. If this piece can be either plus or minus, we end up using the ABS() function, which introduces a branch for those machines that don't have a conditional assignment assembly instruction. As a result, it is possible to simply use values 1-6 for white pieces and 7-12 for black pieces. This won't eliminate the branch in some places, but it might make things like a switch statement in C more efficient. This simply has to be tested on your machine to see if it helps or hurts performance.

The next issue is how to use the array to represent the board. That is, if a white rook is on square a1, assuming the 0X88 offset representation, then what value do we store in board[0]? If we use the sliding-piece representation from above, we might say board[0]=6. However, this might not be the most efficient way to store things. One immediate problem is how will we locate every white piece when we want to generate moves? We certainly don't want to have to loop over all of the 64 possible squares (or even more in the 0X88 board, since we have 8 useless squares between each rank of the board.)

One possible approach is an array of piece records for each side. A record might be defined like this:

```
struct piece_entry {
          int piece_value;     /* 1=pawn, ..., 7=queen */
          int piece_location;  /* where the piece is on the board */
          ...                  /* other things to be added later */
}
```

This has one immediate advantage. If we define an array of 16 of these piece records for white and 16 for black, the loop to touch each piece only has to execute 16 iterations (maximum) because each of the piece records tells exactly which square that piece is on. We immediatly make locating each of our pieces four times faster.

We might store the index to the piece records in the board[] array now, rather than the simple piece code. If we are given a square, and want to know what piece is on that square, board[square] gives us an index into the piece records, so piece_entry[board[square]].piece_value would give us the value of the piece on that square. In reality, we would probably want to define the board as

piece_entry *board[128];

and then simply store the address of the piece entry in each element of the board array that is occupied, and NULL (0) in the unoccupied squares. The advantage in speed of using pointers rather than subscripts is noticable, although the reason why it is faster is best explained in any good book on programming in C.

Another advantage of this approach is that we can store other things along with the square of the piece and the value of the piece. For example, when we get to the evaluation paper and talk about "lazy evaluation," if we know the largest positional score a piece can produce we can store it in this entry to help decide whether a full

piece evaluation will help or not (this will be explained in detail when evaluation is discussed.)

What we end up with, then, is that board[square] points to the piece record for the piece that stands on square. Given a square, we can find the piece that stands on that square, or given a pointer to a piece record, we can find what square that piece stands on. We now have a compact array of piece records that we can scan through without having to skip over a large number of useless information (remember that the chess board is 1/2 empty and gets progressively more empty as the game is played.)

The array of piece records can also be ordered in any fashion. For example, a good chess program will discover that most pawn moves are bad, because once a pawn is moved, it can never back up. If the list of piece records is ordered so that pieces appear before pawns in the list, then the move generator will generate piece moves before pawn moves, since it scans the piece list from front to back. At the root of the tree, we might want to sort this list so that pieces on the first rank are placed first, making the move generator produce developing moves before producing moves for pieces that have already been moved once.

There are many variations on this theme in current chess programs. The one underlying theme is to avoid iterating over 64 squares, when one side can only occupy at most sixteen squares on the board. As pieces are removed, this will let the move generation proceed faster if the piece list is compressed at the root of the tree so that empty positions are put at the end and marked with zero to signify that the list is "short", rather than continually iterating over the 64 (or more) squares on the chess board.

This concludes the discussion of offset board representations. If move generation and attack detection were the most time-consuming parts of the typical chess program, the 0X88 approach would be difficult to improve on.

Bitmap board representation

Once we get to the point of writing a complete chess program, we soon discover that the offset representation of the chess board has some severe performance disadvantages that tend to minimize the advantages gained by the 0X88 method.

The bitmap (bitboard) board representation was described in "Chess Skill in Man and Machine" in the chapter written by Slate and Atkin. This chapter described the implementation of Chess 4.0, which used the bitmap board representation.

The approach is actually quite simple. Using a single 64-bit word, we can store the positions of all of one particular type of piece, by representing the squares on the board as bits in the 64-bit bitmap. For this discussion, I'll map chess board squares to bits using the methodology developed and used in Crafty. Here we map a1=0, h1=7, a8=56 and h8=63. These numbers represent bit numbers in the 64-bit bitmap, with bit 0 being the leftmost or most significant bit, and bit 63 being the right-most of least significant bit. It should be noted that while this was an optimal numbering scheme for the Cray family of computers, it is backward for the Intel family. In the X86 architecture, the BSF/BSR instructions number the bits as 0=LSB or rightmost bit, 63=MSB or leftmost bit. This is therefore a better numbering scheme as it eliminates the mapping function f(bit#)=63-bit#.

The simplest implementation is to use 12 of these bitmaps (bitmap=64 bit word from this point forward) to represent the locations of the twelve different pieces (king, queen, rook, bishop, knight and pawn for each side.)

At first, this looks quite elegant, because it obviously takes significantly less memory to store the entire board, even if you use chars (1 byte) in C to store the previous representations. At second glance, however, we begin to wonder just how we are going to do some of the operations necessary to actually use this information. This has likely been the primary reason that bitmaps have not been widely used; they are different and take some time to "sink in" to the point where they become easy to use. I would speculate that it took me roughly a year before I was able to get past the mental gymnastics of designing an algorithm using offset representation ideas and then working out how to modify the idea to fit the bitmap approach. After two years of working only with bitmaps, it has become natural and easy, and it's unlikely that I will ever "go back."

So far, we have 12 bitmaps, six for black, six for white. In Crafty, they are labeled WhitePawns, WhiteKnights, WhiteBishops, WhiteRooks, WhiteQueens, and WhiteKing, BlackPawns, BlackKnights, BlackBishops, BlackRooks, BlackQueens and BlackKing. As we get further into using these bitmaps, we will find that we will often want the set of squares occupied by either black or white. While we can compute this by ORing the appropriate bitmaps together, it turns out to be more efficient to add at least two more bitmaps, one called WhitePieces and one called BlackPieces. These two bitmaps are the logical sum of the corresponding bitmaps, which will let us avoid lots of ORing later when we need them. We will add a few more special-purpose bitmaps once we reach the stage of generating moves.

That is all there is to the bitmap approach. There are many more details that we will have to work out when we get to move generation, but as you will see, detecting when a piece tries to move off the edge of the board will not be a problem, because the move generation techniques we will look at simply don't generate moves in the same way (there will be no loops to slide a piece down a diagonal for example.)

However, there is one additional advantage to bitmaps that I have come to call "bit-parallel" operations. A simple example illustrates the idea best. We are trying to determine if the white pawn on square e4 is passed. To do this, we have to check the squares d5,d6,d7,e5,e6,e7,f5,f6 and f7 to see if any are occupied by black pawns. With the offset board representation, we will either have to test each of these 9 squares, one at a time, or else incrementally maintain the status of each pawn as to whether or not it is passed (which, as you will see later is going to complicate MakeMove() in a significant way.) Using bitmaps makes this not only easy, but actually trivial and fast. We simply precompute an array of 64 bitmaps, one for each valid square a pawn can occupy (plus, obviously 16 squares where a pawn will never exist). For the bitmap for square 28 (e4) we simply set bits for the nine squares listed, and save this bitmap for use anytime we want to ask is the white pawn on e4 passed. To answer the question, we simply AND this bitmap with the locations of all black pawns (BlackPawns) and if the result is non-zero, we know a black pawn stands on one of the squares that makes this white pawn not passed. If you compare the speed of this to the speed of the offset representation loop, you get excited, quickly. I coined the term bit-parallel to explain this phenomenon, because we are basically asking nine questions with one AND operation, which is effectively asking nine questions in parallel and getting the combined answer in one cycle.

What you are going to see, later on, is that bitmaps offer an amazing capability in the evaluation of chess positions. Even more importantly, when we get to move genration, you will find that bitmaps are actually more efficient at generating moves than the 0X88 approach.

Before concluding this paper, you are probably asking "if bitmaps are so great, why isn't everyone using them?" That is a fair question of course. At present the issue is speed. On 32-bit microprocessors, many of the operations we want to do on bitmaps are not efficient because they have to be done piece-meal using 32-bit registers. For AND, OR, XOR and similar operations, this doesn't hurt much, but for shifting, and, more importantly, finding the first one-bit that is set, current machines don't offer any features to make these operations efficient. However, 64-bit microprocessors are here, and they offer instant performance

improvements to bitmap programs because they do the very operations on 64-bit words that a bitmap program needs. It is highly likely that future chess programs will migrate to this representation or be left behind in performance, because the very thing that makes bitmaps unattractive at present is on the verge of becoming a non-issue when new PC platforms are all 64-bit machines. Crafty is already about as fast as most commercial chess programs, even though it is handicapped by current 32-bit architectures. It will see immediate improvement on 64-bit machines while the offset programs will not, because they have no need for 64-bit integers. Use shorter integers you say? Of course you can, but you will discover that this will actually slow things down because 64-bit processors like to load/store 64-bit values, not 16-bit or 32-bit values.

For now (2004) the bitmap and offset approaches seem to be nearly equal, with the one exception that the evaluation using bitmaps is always going to be significantly faster as you will see when we get to the evaluation paper. The bitmap approach does have one significant disadvantage, that is the lack of clarity in the programming methodology. Everyone understands arrays of integers, but arrays of bits, where you can not use a subscript to get to a particular bit takes some acclimation time. Possibly a LOT of time.

It is highly likely that if you were to ask me what I would recommend to a beginning programmer, I would respond with the offset approach. It is easier to understand and get used to. Writing a bitmap program takes time to do because the bitmap learning curve is steep at first. Several have tried it, but gave up before their proficiency reached a point where they could actually use bitmaps efficiently. In my case, I was sorely tempted on many occasions to return to the old offset approach for speed reasons, but I made a commitment to myself early in the bitmap development cycle that I was simply not going to revert back to the offset representation until I was completely convinced that bitmaps would not produce a program that could play as well as an offset program. I am now convinced that this bitmap approach works, and works well. You just have to be prepared to invest the mental training time necessary to become proficient at bitmap programming.

One note is that we will likely decide to use an offset board representation to complement a bitmap approach. One reason for this is move generation. When we generate a target square for a piece, and discover this piece is occupied by an opponent's piece, we might want to store the captured piece as part of the move. Unfortunately, figuring out what we are capturing is not exactly trivial, because we have to create a bitmap with the target square bit set, and then AND this with each of the bitmaps for the opponent's pieces until we get a non-zero result. Not very efficient. It would be much better if we had a board[] offset representation as well, where board[square] contains a piece code for the piece that occupies that square. It seems redundant, and it really is, but it turns out to be more efficient to do this, although we'll look into this problem more later.

Now that we've seen the various approaches to representing the chess board, it is time to find out how to generate chess moves from this data. The paper on rotated bitmaps explains how to do this with bitmaps, while the offset approach has been previously explained in this paper.