Square Attacked By (/Square+Attacked+By)

Max Ernst - L'Ange du Foyer, 1937 [1]

Square Attacked By,
determines whether a square is attacked and/or defended by various or specific pieces. So far, as elaborated in pawn-, knight- and king pattern, as well as sliding piece attacks, we are able to generate all attacks and target-sets of all pieces, sufficient to generate all pseudo legal moves. It is often useful to generate attacks **to** a certain square, or to look whether moves retrieved elsewhere are pseudo legal or legal. Some programs maintain incremental updated attack tables for that purpose. The techniques proposed on this page are intended to use on the fly.

Table of Contents

Attacks to a Square

By all Pieces

A common approach is to put a **super-piece** on the to-square, to look up all kind of piece-type attacks from there and to intersect them with all appropriate pieces able to attack that square. Note that white pawn attacks intersect black pawns and vice versa. Knights, kings and sliders are considered as union of both sides. The set of all attacking and defending pieces is the union of all piece-attack intersections. Assuming a C++ member function of a Bitboard Board-Definition class. Robert Hyatt further checks whether there are any sliding pieces on relevant rays at all, in order to save calling the attack getter in case there are none [2]:

```
U64 CBoard::attacksTo(U64 occupied, enumSquare sq) {
   U64 knights, kings, bishopsQueens, rooksQueens;
   knights        = pieceBB[nWhiteKnight] | pieceBB[nBlackKnight];
   kings          = pieceBB[nWhiteKing]   | pieceBB[nBlackKing];
   rooksQueens    =
   bishopsQueens  = pieceBB[nWhiteQueen]  | pieceBB[nBlackQueen];
   rooksQueens   |= pieceBB[nWhiteRook]   | pieceBB[nBlackRook];
   bishopsQueens |= pieceBB[nWhiteBishop] | pieceBB[nBlackBishop];

   return (arrPawnAttacks[nWhite][sq] & pieceBB[nBlackPawn])
        | (arrPawnAttacks[nBlack][sq] & pieceBB[nWhitePawn])
        | (arrKnightAttacks      [sq] & knights)
        | (arrKingAttacks        [sq] & kings)
        | (bishopAttacks(occupied,sq) & bishopsQueens)
        | (rookAttacks  (occupied,sq) & rooksQueens)
        ;
}
```

Any Attack by Side

If boolean information is required, whether a square is attacked by a side, one may use conditionals to return early. This might be useful to determine whether a king is in check. Assuming a C++ member function of a Bitboard Board-Definition class:

```
bool CBoard::attacked(U64 occupied, enumSquare square, enumColor bySide) {
   U64 pawns          = pieceBB[nWhitePawn   + bySide];
   if ( arrPawnAttacks[bySide^1][square]    & pawns )         return true;
   U64 knights        = pieceBB[nWhiteKnight + bySide];
   if ( arrKnightAttacks[square]            & knights )       return true;
   U64 king           = pieceBB[nWhiteKing   + bySide];
   if ( arrKingAttacks[square]              & king )          return true;
   U64 bishopsQueens  = pieceBB[nWhiteQueen  + bySide]
                      | pieceBB[nWhiteBishop + bySide];
   if ( bishopAttacks(occupied, square)     & bishopsQueens ) return true;
   U64 rooksQueens    = pieceBB[nWhiteQueen  + bySide]
                      | pieceBB[nWhiteRook   + bySide];
   if ( rookAttacks  (occupied, square)     & rooksQueens )   return true;
   return false;
}
```

Legality Test

One application inside a chess program, is to test whether a certain move is psuedo-legal. This could be a hash move probed from the transposition table, or a killer move supplied by the killer heuristic. In cut-nodes one may save the complete move-generation by one legality test.

In Between

Assuming otherwise legal from-to coordinates, it is about distant moves of sliding pieces (and double pawn push and castling) - whether a square between from and to is obstructed or not. One obvious solution is to switch on piece-type and call the appropriate attack- or move-target getter by from-square, to see whether the target bit is set. For a queen that may take quite some instructions for up to four sliding lines, thus there seems to be a cheaper solution.

Rectangular Lookup

The common approach is to lookup a two-dimensional 64 times 64 array, initialized with empty sets and appropriate in-between sets for distant squares on the same line. If the intersection of in-between sets with the occupancy is empty, there are no obstructions, and the move is considered pseudo legal. This is implicitly true for squares in king- and knight distance as well, since they already contain zero.

```
U64 arrRectangular[64][64]; // 4096*8 = 32KByte

U64 inBetween(enumSquare from, enumSquare to) {
   return arrRectangular[from][to];
}

bool mayMove(enumSquare from, enumSquare to, U64 occ) {
   return (inBetween(from, to) & occ) == 0;
}
```

That looks cheap, and likely is the fastest for recent processor architectures, but 32KByte is just another thing competing with the caches. Three further space-time tradeoffs are mentioned, triangular lookup, 0x88 difference and rotate, and pure calculation.

Triangular Lookup

Due to the commutative property of from- and to-squares, each in-between set is stored twice in the 64x64 array. Eugene Nalimov once suggested to roughly half the table-size by making the rectangle a triangle, and already noted "*Not sure that it will make sense, as index-code will be more complex*" [3]. Applying max and min by sign-mask and some packing with the triangular number, following index calculation may be used:


```
U64 arrTriangular[64*65/2];

int triangularIndex(int a, int b) {
   int d = a - b; /* difference */
   d &= d >> 31;  /* only if negative */
   b += d;        /* min */
   a -= d;        /* max */
   b *= b ^ 127;  /* min * (127-min)  ... */
   return (b>>1) + a; /* ... /2 + max */
}

U64 inBetween(enumSquare from, enumSquare to) {
   return arrTriangular[ triangularIndex(from, to) ];
}
```

0x88 Difference

What about a translation of one square (the smallest) to a1 and shifting the occupancy right? Unfortunately, the square difference is ambiguous according to the 8 ray- and 8 knight directions, +-7 occurs as rank- or anti-diagonal-difference, +-6 occurs as rank- and knight-difference. If we keep other stuff like distance, Manhattan-distance, ray-direction and so on - it might be worth to rely on Vector Attacks or the difference of 0x88 coordinates and their property of being unambiguous according to ray-direction and distance. By rotating left the pre-calculated obstructions, the order of squares don't cares. Don Dailey reported the 64x64 array lookup slightly faster, apparently inside Komodo [4].:

```
U64 arrInBetweenBy0x88Diff[240]; // 1920 bytes, 2KByte - 128 Byte

unsigned int x88diff(enumSquare f, enumSquare t) {
   return t - f + (t|7) - (f|7) + 120;
}

U64 inBetween(enumSquare from, enumSquare to) {
   return rotateLeft(arrInBetweenBy0x88Diff[x88diff(from,to)], from);
}
```

Pure Calculation

A branch-less solution without any lookups and some parallel gain is likely too expensive on the fly and may at most used for initialization purposes:

```
U64 inBetween(enumSquare sq1, enumSquare sq2) {
   const U64 m1   = C64(-1);
   const U64 a2a7 = C64(0x0001010101010100);
   const U64 b2g7 = C64(0x0040201008040200);
   const U64 h1b7 = C64(0x0002040810204080); /* Thanks Dustin, g2b7 did not work for c1-a3 */
   U64 btwn, line, rank, file;

   btwn  = (m1 << sq1) ^ (m1 << sq2);
   file  =   (sq2 & 7) - (sq1   & 7);
   rank  =  ((sq2 | 7) -  sq1) >> 3 ;
   line  =      (   (file  &  7) - 1) & a2a7; /* a2a7 if same file */
   line += 2 * ((   (rank  &  7) - 1) >> 58); /* b1g1 if same rank */
   line += (((rank - file) & 15) - 1) & b2g7; /* b2g7 if same diagonal */
   line += (((rank + file) & 15) - 1) & h1b7; /* h1b7 if same antidiag */
   line *= btwn & -btwn; /* mul acts like shift by smaller square */
   return line & btwn;   /* return the bits on that line in-between */
}
```

First, the in-between set as superset of the possible line-bits is calculated, excluding the "greater" square but including the "smaller", e.g. f6 and c3.

```
                                        btwn including c3
-1<<f6              -1<<c3                   excluding f6
1 1 1 1 1 1 1 1     1 1 1 1 1 1 1 1     . . . . . . . .
1 1 1 1 1 1 1 1     1 1 1 1 1 1 1 1     . . . . . . . .
. . . . . 1 1 1     1 1 1 1 1 1 1 1     1 1 1 1 1 . . .
. . . . . . . .     1 1 1 1 1 1 1 1     1 1 1 1 1 1 1 1
. . . . . . . .  ^  1 1 1 1 1 1 1 1  =  1 1 1 1 1 1 1 1
. . . . . . . .     . . 1 1 1 1 1 1     . . 1 1 1 1 1 1
. . . . . . . .     . . . . . . . .     . . . . . . . .
. . . . . . . .     . . . . . . . .     . . . . . . . .

b2g7             *  LS1B(btwn)       =  line
. . . . . . . .     . . . . . . . .     . . . . . . . 1
. . . . . . 1 .     . . . . . . . .     . . . . . . 1 .
. . . . . 1 . .     . . . . . . . .     . . . . . 1 . .
. . . . 1 . . .     . . . . . . . .     . . . . 1 . . .
. . . 1 . . . .  *  . . . . . . . .  =  . . . 1 . . . .
. . 1 . . . . .     . . 1 . . . . .     . . . . . . . .
. 1 . . . . . .     . . . . . . . .     . . . . . . . .
. . . . . . . .     . . . . . . . .     . . . . . . . .

line             &  btwn             =  inBetween
. . . . . . . 1     . . . . . . . .     . . . . . . . .
. . . . . . 1 .     . . . . . . . .     . . . . . . . .
. . . . . 1 . .     1 1 1 1 1 . . .     . . . . . . . .
. . . . 1 . . .     1 1 1 1 1 1 1 1     . . . . 1 . . .
. . . 1 . . . .  &  1 1 1 1 1 1 1 1  =  . . . 1 . . . .
. . . . . . . .     . . 1 1 1 1 1 1     . . . . . . . .
. . . . . . . .     . . . . . . . .     . . . . . . . .
. . . . . . . .     . . . . . . . .     . . . . . . . .
```

If both squares share either a rank, file, diagonal or anti-diagonal, one byte-difference of the four lines becomes zero, which is (zero) extended to a bitboard, otherwise a value 1 ... 255. Subtracting '1' either leaves all bits (-1) set, otherwise 0..254. The a1-scaled mask is shifted left left by the "smallest" square, which can be done by multiplication with the isolated LS1B of the in-between set. The final intersection leaves the obstructed bits between two squares.

Attacked by Piece on Square

The obstructed lookup idea may be advanced to determine one particular attack by one piece. To lookup whether a certain square is attacked by that piece on another square. Thus, we need a second obstructed-array dimension by piece code. All legal attacks by none sliding pieces are initialized by the empty set zero, all others with the universal set -1. Any intersection with the occupancy (which we need for the sliding pieces anyway) is either empty or not. Sliding pieces have the appropriate obstructed bits set for squares on a common diagonal for bishops and queens - and a common orthogonal for rooks and queens. To safe some memory we rely on the 0x88 trick.

Except pawns all white and black pieces have the same attacks - thus based on the piece enumeration mentioned in the Bitboard Board-Definition, one may divide the piece code by two to shift right the least significant piece color bit: {1,2,3,4,5,6} for {pawn, knight, bishop, rook, queen, king}. In case of a black pawn we subtract one, to get a 0..6 range:

```
U64 attacksBy0x88DiffAndPiece[7][256];  // 14KByte

/* is square <to> attacked by <piece> from square <from> */
bool isAttacked(enumSquare from, enumSquare to, enumPiece piece, U64 occ) {
```

```
  int isBlackPawn = (piece ^ nBlackPawn) - 1;
  isBlackPawn >>= 31; /* -1 if black pawn, otherwise 0 */
  return (attacksBy0x88DiffAndPiece[piece/2][isBlackPawn][x88diff(from,to)]
          & rotateRight (occ, from) ) == 0;
}
```


See rotateRight.

If one considers white and black pawns as disjoint pieces even without color information, one may safe that isBlackPawn calculation obtaining the same table size. With a 11 or 12 array-range one may index by piece-2 or similar according to the piece definition.

See also

- Attack and Defend Maps
- InBetween

Forum Posts

- Two questions for bitboard experts by Tord Romstad, Winboard Forum, November 06, 2004 » Piece-Lists
- Bitboard of squares between by Onno Garms, Winboard Forum, June 15, 2007
- bit boards - what is betwen by Don Dailey, CCC, December 02, 2012
- Extract direction (ray) informations from two squares by Mathieu Pagé, CCC, June 18, 2013 » Rays, Direction
- AttacksTo() bitboard by Tony Soares, CCC, December 29, 2013
- help with bitboards by stefano.c... , FishCooking, December 06, 2017

References

1. ^ Max Ernst from Wikipedia
2. ^ Re: AttacksTo() bitboard by Robert Hyatt, CCC, December 29, 2013
3. ^ Re: Bitboard user's information request by Eugene Nalimov, CCC, October 06, 1999
4. ^ Re: bit boards - what is betwen by Don Dailey, CCC, December 04, 2012

What links here?

Page	Date Edited
0x88	Nov 28, 2016
Array	Dec 1, 2016
Attack and Defend Maps	Nov 5, 2016
Bitboards	Nov 14, 2017
Check	Feb 1, 2018
Checks and Pinned Pieces (Bitboards)	Aug 14, 2013
CHEOPS	Apr 18, 2015
Chess 0.5	Nov 20, 2016
Direction	Oct 6, 2016
DirGolem	Jun 5, 2016
Discovered Attack	Nov 8, 2010
Discovered Check	Feb 1, 2018
Double Check	Apr 4, 2013
Eye Movements	Jul 22, 2015
Guard Heuristic	Nov 18, 2015
InBetween	Jan 21, 2018
Interception	May 21, 2011
Interference	May 20, 2011
Joker IT	Sep 16, 2017
Lachex	Jan 7, 2016
Legal Move	Feb 16, 2017
Mathieu Pagé	Feb 14, 2018
Onno Garms	Jul 19, 2013
Origin Square	Oct 10, 2016
Overloading	May 5, 2017
Perceiver	Nov 21, 2017
Piece-Lists	Feb 13, 2017
Pieces versus Directions	Oct 6, 2016
Pseudo-Legal Move	May 23, 2015
Rasjid Chan	Nov 26, 2014
Rays	Oct 20, 2016
Searcher	Sep 26, 2016
SEE - The Swap Algorithm	Jun 5, 2017
Sliding Piece Attacks	May 27, 2016
Space-Time Tradeoff	Jun 17, 2015
Square Attacked By	Jan 20, 2018
Square Control	Sep 15, 2016
Squares	Feb 15, 2015
Target Square	Oct 26, 2017
Thor's Hammer	Nov 23, 2013
Tinker	Aug 29, 2015
Tord Romstad	Dec 9, 2017
Undermining	Jun 10, 2012
Vector Attacks	Dec 15, 2017
WChess	Jan 7, 2016

Up one Level

I think 32Kbyte is quite a lot of wasted memory just to check whether the squares in between are obstructed.

I prefer to use:
((from - 1) ^ (to - 1)) & ray & occ & ~(from | to)

ray depends on the direction:
eg if col(from) == col(to) ray = bbcol[col(from)]

What do you think?

regards,
Edmund

(https://www.wikispaces.com/user/view/Edmund_Moshammer) Edmund_Moshammer (https://www.wikispaces.com/user/view/Edmund_Moshammer) *Apr 19, 2009*
I didn't read the text careful enough and missed your point about direct calculation. Anyway, I still think direct calculation can be competitive. Especially if you know whether a rank, file or diagonal is shared.

(https://www.wikispaces.com/user/view/GerdIsenberg) GerdIsenberg (https://www.wikispaces.com/user/view/GerdIsenberg) *Apr 19, 2009*
Hi Edmund,
as always with computation versus memory trade-offs, it depends on what you do elsewhere and one has to try how it interacts with surrounding code. With recent cpus and caches, I think pre-calculated obstructed arrays are fine for a decent L1 hit-rate, as long from-to squares are not totally random. My wild guess is that 10%-20% reads from L2 would still favor the pure lookup.

Cheers,
Gerd