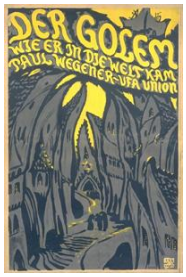
Home * Board Representation * Move Generation * **DirGolem**



The Golem [1]

DirGolem,
Direction-wise Generation of Legal Moves, is best done with vectors of bitboards applying AltiVec, SSE2, AVX2 or similar SIMD architectures with appropriate register sets. Legal moves are stored as target-bitboards for all 16 move directions as a kind of unsorted move list inside the state of a node. That are four orthogonal directions with rook-, queen- and king-moves including castling and vertical pawn-pushes, four diagonal directions with bishop-, queen- and king-moves and pawn-captures including en passant, and eight knight directions. Each target square of each direction set has an unique one-to-one relation to it's source square.

Table of Contents

Prospect

To get the idea, the following description with scalar C++ code intended as pseudo code, is a rough translation of HansDamf's intrinsic SSE2 code by Gerd Isenberg. It assumes White to move within a color flipper approach. More information for a set-wise SEE, determining check giving moves including discovered checks, etc. may be collected and considered. There are zillions of implementation nuances to utilize up to 16 128-bit XMM-, 16 256-bit YMM (AVX2), or even AVX-512 32 512-bit ZMM-SIMD registers [2]. The generation is completely branch-less, does not use huge lookup tables, and is intended to hide the latency of a prefetched TT probe. However, picking any moves or move proposals like hash- or killer moves from that list is more expensive and is typically done by a finite state machine to ensure proper move ordering considering MVV-LVA, hanging pieces, etc., as mentioned in pieces versus directions.

Basics

- King Attacks by Calculation
- Multiple Knight Attacks
- Multiple Sliding Pieces
- One Step Only
- Opposite Ray-Directions
- Pawn Attacks set-wise
- Pawn Pushes set-wise
- Pieces versus Directions

Code samples and bitboard diagrams rely on Little endian file and rank mapping.

Move Target Sets

To determine white absolutely pinned pieces and squares attacked by black pieces, pawns, and king, taboo for the white king, black attacks are generated in a first phase. Those attacks are also utilized to detect whether the own king is in check. Aggregated collected information is then used in a second phase of legal move target generation.

Black Attacks

Sliders & Super King

First, all black sliding attacks are generated by Dumb7- or Kogge-Stone fill, unrolled for all eight ray directions, to aggregate them into a white king taboo bitboard, and to determine and keep line-wise in-between bitboards by intersection with opposite ray-direction attacks of the white king as sliding super piece, best generated by the classical approach, or alternatively along with the sliding pieces itself as additional SIMD data element processing two opponent fill direction in one run.

The occupancy for the black sliding attacks has the white king excluded, so that in case of check by a sliding piece the king is x-rayed to later avoid king moves to otherwise non attacked squares. The in-between sets intersected with own pieces leave pinned pieces, but are also handy to determine an interposing block target set in case of distant sliding checks. In total, seven bitboards are processed as used in the second part of the routine, four line-wise in-between sets, orthogonal and diagonal white king super piece attacks to later determine a possible sliding checking piece, and the aggregated black attacks.

```
U64 horInbetween, verInbetween, diaInbetween, antInbetween;
U64 wKSuperAttacksOrth, wKSuperAttacksDia , bAnyAttacks;
```

Intended members associated with a position object have a m_-prefix, scratch or register variables an underscore:

```
/* black rooks and queens west */
_bAttacks = westAttacks (m_bRooks | m_bQueens, m_occ ^ m_wKbb);
bAnyAttacks         = _bAttacks;
_wKSuperAttacks     = slidingRayEastAttacks(m_wKsq, m_occ);
wKSuperAttacksOrth  = _wKSuperAttacks;
horInbetween        = _bAttacks & _wKSuperAttacks;
/* black rooks and queens east */
_bAttacks = eastAttacks (m_bRooks | m_bQueens, m_occ ^ m_wKbb);
bAnyAttacks        |= _bAttacks;
_wKSuperAttacks     = slidingRayWestAttacks(m_wKsq, m_occ);
```

```
  wKSuperAttacksOrth |= _wKSuperAttacks;
  horInbetween       |= _bAttacks & _wKSuperAttacks;
  /* black rooks and queens north */
  _bAttacks = nortAttacks (m_bRooks | m_bQueens, m_occ ^ m_wKbb);
  bAnyAttacks        |= _bAttacks;
  _wKSuperAttacks     = slidingRaySoutAttacks(m_wKsq, m_occ);
  wKSuperAttacksOrth |= _wKSuperAttacks;
  verInbetween        = _bAttacks & _wKSuperAttacks;
  /* black rooks and queens south */
  _bAttacks = soutAttacks (m_bRooks | m_bQueens, m_occ ^ m_wKbb);
  bAnyAttacks        |= _bAttacks;
  _wKSuperAttacks     = slidingRayNortAttacks(m_wKsq, m_occ);
  wKSuperAttacksOrth |= _wKSuperAttacks;
  verInbetween       |= _bAttacks & _wKSuperAttacks;

  /* black bishops and queens north east */
  _bAttacks = noEaAttacks (m_bBishops | m_bQueens, m_occ ^ m_wKbb);
  bAnyAttacks        |= _bAttacks;
  _wKSuperAttacks     = slidingRaySoWeAttacks(m_wKsq, m_occ);
  wKSuperAttacksDia   = _wKSuperAttacks;
  diaInbetween        = _bAttacks & _wKSuperAttacks;
  /* black bishops and queens south west */
  _bAttacks = soWeAttacks (m_bBishops | m_bQueens, m_occ ^ m_wKbb);
  bAnyAttacks        |= _bAttacks;
  _wKSuperAttacks     = slidingRayNoEaAttacks(m_wKsq, m_occ);
  wKSuperAttacksDia  |= _wKSuperAttacks;
  diaInbetween       |= _bAttacks & _wKSuperAttacks;
  /* black bishops and queens north west */
  _bAttacks = noWeAttacks (m_bBishops | m_bQueens, m_occ ^ m_wKbb);
  bAnyAttacks        |= _bAttacks;
  _wKSuperAttacks     = slidingRaySoEaAttacks(m_wKsq, m_occ);
  wKSuperAttacksDia  |= _wKSuperAttacks;
  antInbetween        = _bAttacks & _wKSuperAttacks;
  /* black bishops and queens south east */
  _bAttacks = soEaAttacks (m_bBishops | m_bQueens, m_occ ^ m_wKbb);
  bAnyAttacks        |= _bAttacks;
  _wKSuperAttacks     = slidingRayNoWeAttacks(m_wKsq, m_occ);
  wKSuperAttacksDia  |= _wKSuperAttacks;
  antInbetween       |= _bAttacks & _wKSuperAttacks;
```

None Sliding Pieces

Black pawn- and knight- attacks are best determined by disjoint direction-wise steps and the multiple knight attacks routine. Black king attacks may be determined by lookup or calculation:

```
  /* black knight attacks */
  bAnyAttacks  |= _knightAttacks  (m_bKnights);
  /* black pawn attacks */
  bAnyAttacks |= _bPawnEastAttacks(m_bPawns);
  bAnyAttacks |= _bPawnWestAttacks(m_bPawns);
  /* black king attacks */
  bAnyAttacks |= _kingAttacks(m_bKbb);
```

Generating White Moves

Now, with all black attacks and black sliding vs white king in-between sets processed, one generates all legal white moves into 16 move-target bitboards.

In Check?

To make in check or double check handling branch-less, a target mask for all non king moves is calculated, which is needed anyway to remove own white pieces from the attack sets. In case of check it only contains the check giving piece as capture target - distant sliding checks include the in-between set intersected with empty squares as target set for blocking the check. The rare case of double check is considered by a mask (_nullIfDblCheck) computed from the set of check giving pieces - which leaves empty if population greater one and the universe (~empty) otherwise. Note that the ones' decrement of any single populated bitboard is always positive and leaves the empty set if shifted right 63 arithmetically:

```
  U64 allInbetween = horInbetween | verInbetween |
 diaInbetween | antInbetween;
  _blocks      =  allInbetween & ~m_occ;
/* in case of distant check */
  _checkFrom   = (wKSuperAttacksOrth    & (m_bRooks  | m_bQueens) )
               | (wKSuperAttacksDia     & (m_bBishop | m_bQueens) )
               | (knightAttacks(m_wKbb) &  m_bKnights )
               | (wPawnAttacks (m_wKbb) &  m_bPawns )
               ;
  I64 _nullIfcheck    = ( (I64)( bAnyAttacks & m_wKbb ) - 1) >> 63;
/* signed shifts */
  I64 _nullIfDblCheck = ( (I64)( checkFrom & (checkFrom-1) ) - 1) >>
63;

  _checkTo     = _checkFrom | _blocks  | _nullIfcheck;
  targetMask   = ~m_wPieces & _checkTo & _nullIfDblCheck;
```

Sliding Pieces

Moves by white sliding pieces are computed with eight Dumb7 or Kogge-Stone fills. Pinned sliders are

considered partial pinned to move along the pinned direction:

```
  /* horizontal rook and queen moves */
  _sliders = (m_wRooks | m_wQueens) & ~(allInbetween ^ horInbetween);
  m_moveTargets[eWest] = westAttacks (_sliders, m_occ) & targetMask;
  m_moveTargets[eEast] = eastAttacks (_sliders, m_occ) & targetMask;
  /* vertical rook and queen moves*/
  _sliders = (m_wRooks | m_wQueens) & ~(allInbetween ^ verInbetween);
  m_moveTargets[eNort] = nortAttacks (_sliders, m_occ) & targetMask;
  m_moveTargets[eSout] = soutAttacks (_sliders, m_occ) & targetMask;
  /* diagonal bishop and queen moves */
  _sliders = (m_wBishops | m_wQueens) & ~(allInbetween ^
 diaInbetween);
  m_moveTargets[eNoEa] = noEaAttacks (_sliders, m_occ) & targetMask;
  m_moveTargets[eSoWe] = soWeAttacks (_sliders, m_occ) & targetMask;
  /* antidiagonal bishop and queen moves */
  _sliders = (m_wBishops | m_wQueens) & ~(allInbetween ^
 antInbetween);
  m_moveTargets[eNoWe] = noWeAttacks (_sliders, m_occ) & targetMask;
  m_moveTargets[eSoEa] = soEaAttacks (_sliders, m_occ) & targetMask;
```

None Sliding Pieces

Knight moves with their disjoint unique directions are generated by eight appropriate direction shifts with all white knights not pinned, using the same targetMask considering in check. Pawn captures and Pawn pushes also consider partial pins. A given en passant target square implies a strictly legal move, as already secured while making the triggering double pawn push.

Finally king move targets are generated by eight one step direction shifts, masking off target squares occupied by own pieces, or attacked by black, including x-rays through the white king in case of check by a sliding piece. Castling is left to the ambitious reader. In orthodox chess, west or east castling can simply distinguished from ordinal king moves due to the double king step, but requires some more tinkering in Chess960.

```
  /* knight moves */
  _knights = m_wKnights & ~allInbetween;
  m_moveTargets[eNoNoEa] = noNoEa(_knights) & targetMask;
  m_moveTargets[eNoEaEa] = noEaEa(_knights) & targetMask;
  m_moveTargets[eSoEaEa] = soEaEa(_knights) & targetMask;
  m_moveTargets[eSoSoEa] = soSoEa(_knights) & targetMask;
  m_moveTargets[eNoNoWe] = noNoWe(_knights) & targetMask;
  m_moveTargets[eNoWeWe] = noWeWe(_knights) & targetMask;
  m_moveTargets[eSoWeWe] = soWeWe(_knights) & targetMask;
  m_moveTargets[eSoSoWe] = soSoWe(_knights) & targetMask;
```

```
  /* pawn captures and en passant */
  _targets = ( m_bPieces & targetMask) | (C64(1) << m_epTarget);
  _pawns   = m_wPans & ~(allInbetween ^ diaInbetween);
  m_moveTargets[eNoEa] |=  noEaOne (_pawns) & _targets ;
  _pawns   = m_wPans & ~(allInbetween ^ antInbetween);
  m_moveTargets[eNoWe] |= noWeOne(_pawns) & _targets;
  /* pawn pushes ... */
  _pawns   = m_wPans  & ~(allInbetween ^ verInbetween);
  _pawnPushs = nortOne (_pawns) & ~m_occ;
  m_moveTargets[eNort] |= _pawnPushs & targetMask;
  /* and double pushs */
  _rank4 = C64(0x00000000FF000000);
  m_moveTargets[eNort] |= nortOne (_pawnPushs) & ~m_occ &
 targetMask & _rank4;
  /* king moves */
  targetMask = ~(m_wPieces | bAnyAttacks);
  m_moveTargets[eWest] |= westOne (m_wKbb) & targetMask;
  m_moveTargets[eEast] |= eastOne (m_wKbb) & targetMask;
  m_moveTargets[eNort] |= nortOne (m_wKbb) & targetMask;
  m_moveTargets[eSout] |= soutOne (m_wKbb) & targetMask;
  m_moveTargets[eNoEa] |= noEaOne (m_wKbb) & targetMask;
  m_moveTargets[eSoWe] |= soWeOne (m_wKbb) & targetMask;
  m_moveTargets[eNoWe] |= noWeOne (m_wKbb) & targetMask;
  m_moveTargets[eSoEa] |= soEaOne (m_wKbb) & targetMask;
```

Finally

Additionally, all 16 move target bitboards might be aggregated into a union set, so that checkmate and stalemate is indicated by the zero flag for early returns, wasting a TT prefetch. Further, some disjoint and aggregated attack and pinned piece information might be utilized in evaluation, maybe best associated with thread local memory shared by all nodes searched, and allocated at the bottom of the stack referred by a (this) pointer, with the limited validity from generation until evaluation or the first (null) move is deepened.

See also

- AVX2
- AVX2 Dumb7Fill
- AVX2 Knight Attacks
- Checks and Pinned Pieces (Bitboards)
- Dumb7Fill
- Fill by Subtraction
- Fill Algorithms

- GPU
- Kogge-Stone Algorithm
- Pigeon
- SSE2

Forum Posts

- Is there such a thing as branchless move generation? by John Hamlen, CCC, June 07, 2012

External Links

- Golem from Wikipedia
- Golem (disambiguation) from Wikipedia

References

1. ^ The Golem: How He Came Into the World, from Wikipedia
2. ^ AVX-512 instructions | Intel® Developer Zone, by James Reinders, July 23, 2013

What links here?

Page	Date Edited
AltiVec	Jun 7, 2016
Avoiding Branches	Dec 16, 2016
AVX2	Aug 8, 2017
Bitboards	Nov 14, 2017
Check	Feb 1, 2018
Checks and Pinned Pieces (Bitboards)	Aug 14, 2013
DirGolem	Jun 5, 2016
Golem	May 18, 2017
HansDamf	Jun 27, 2012
Legal Move	Feb 16, 2017
Move Generation	Jan 29, 2018
Pieces versus Directions	Oct 6, 2016
SSE2	Feb 27, 2018

Up one Level