Intelligent Search & Games Assignment Impasse

Foti Kerkeshi

Maastricht University

Outline

- → Board
- → Piece
- → Move
- → Evaluation
- → α-β Tree Search
- → Live Demo

Preliminaries

- coded in Python programming language
- inspired from python-chess [1]
- note: followed OOP principles, separating all the classes caused import errors, preferred python type checking for better code reading

Board

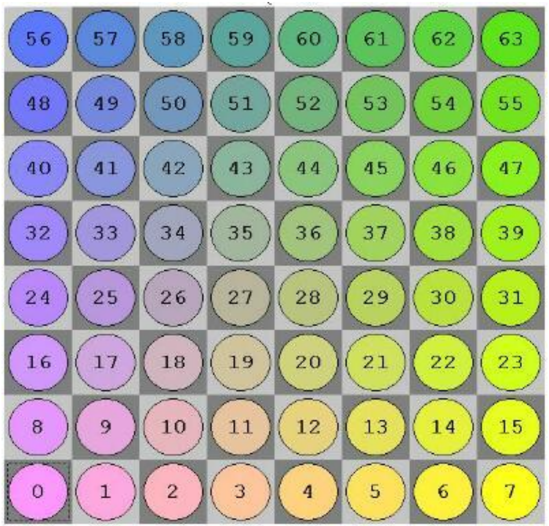
Board Representation

Bitboard

→ only data structure to keep the state of the board: **64-bit Integer**

→ 4 integers are enough to know the state of the board:

- bb_square_occupation_white
- bb_square_occupation_black
- bb_singles_squares
- bb_doubles_squares



state = bb_square_occupation_white | bb_square_occupation_black | bb_singles_squares | bb_doubles_squares

Board Representation

Advantages

- ➔ very very fast move generation
- ➔ memory efficient

Challenges

- ➔ complicated moves need more thought and sometimes are hard to generate with bitwise operators

Trial

- ➔ represent the board as a 32-bit integer instead (impasse uses only 32 squares)
- ➔ hard to generate moves, misalignment on the diagonal indices (8x4)
- ➔ Checker Bitboard Tutorial [2]

BoardState Class

```python
BoardT = TypeVar("BoardT", bound="Board")

class BoardState(Generic[BoardT]):
  def __init__(self, board: BoardT) -> None:
    self.singles = board.singles
    self.doubles = board.doubles

    self.occupied_w = board.occupied_co[WHITE]
    self.occupied_b = board.occupied_co[BLACK]
    self.occupied = board.occupied
    # self.occupied_co = [self.occupied_b, self.occupied_w]

    self.turn = board.turn
```

Board Class

```python
class Board:
  def __init__(self, board_state: BoardState = None) -> None:

    if board_state:
      self.move_stack : List[Move] = []
      self.stack : List[BoardState[BoardT]] = []

      self.turn = board_state.turn

      self.occupied_co = [board_state.occupied_b, board_state.occupied_w]

      self.singles = board_state.singles
      self.doubles = board_state.doubles

      self.occupied = board_state.occupied_b | board_state.occupied_w

    else:
      self.reset_board()
```

Board Functions

```python
def push(self, move: Move):
  ...

  if move.bear_off:
    self.set_piece_at(move.to_square, SINGLE, moving_piece.color)
    if move.transpose:
      self.set_piece_at(move.from_square, SINGLE, moving_piece.color)
  elif move.impasse:
    # if the removed checker was double, replace with a single
    if moving_piece.piece_type == DOUBLE:
      self.set_piece_at(move.from_square, SINGLE, moving_piece.color)
  elif move.transpose:
    self.set_piece_at(move.from_square, DOUBLE, moving_piece.color)
    self.set_piece_at(move.to_square, SINGLE, moving_piece.color)
  else:
    self.set_piece_at(move.to_square, moving_piece.piece_type, moving_piece.color)

  ...

  return self
```

Board Functions

```python
def pop(self: BoardT) -> Move:

  move = self.move_stack.pop()
  self.stack.pop().restore(self)

  return move
```

Board Constants

```
Bitboard = int
BB_EMPTY = 0
BB_ALL = 0xffff_ffff_ffff_ffff

BB_SQUARES = [
  BB_A1, BB_B1, BB_C1, BB_D1, BB_E1, BB_F1, BB_G1, BB_H1,
  BB_A2, BB_B2, BB_C2, BB_D2, BB_E2, BB_F2, BB_G2, BB_H2,
  BB_A3, BB_B3, BB_C3, BB_D3, BB_E3, BB_F3, BB_G3, BB_H3,
  BB_A4, BB_B4, BB_C4, BB_D4, BB_E4, BB_F4, BB_G4, BB_H4,
  BB_A5, BB_B5, BB_C5, BB_D5, BB_E5, BB_F5, BB_G5, BB_H5,
  BB_A6, BB_B6, BB_C6, BB_D6, BB_E6, BB_F6, BB_G6, BB_H6,
  BB_A7, BB_B7, BB_C7, BB_D7, BB_E7, BB_F7, BB_G7, BB_H7,
  BB_A8, BB_B8, BB_C8, BB_D8, BB_E8, BB_F8, BB_G8, BB_H8,
] = [1 << sq for sq in SQUARES]

SQUARES_180 = [square_mirror(sq) for sq in SQUARES]
```

Board Constants

```
BB_FILES = [
  BB_FILE_A,
  BB_FILE_B,
  BB_FILE_C,
  BB_FILE_D,
  BB_FILE_E,
  BB_FILE_F,
  BB_FILE_G,
  BB_FILE_H,
] = [0x0101_0101_0101_0101 << i for i in range(8)]

BB_RANKS = [
  BB_RANK_1,
  BB_RANK_2,
  BB_RANK_3,
  BB_RANK_4,
  BB_RANK_5,
  BB_RANK_6,
  BB_RANK_7,
  BB_RANK_8,
] = [0xff << (8 * i) for i in range(8)]

BB_UPPER_HALF_RANKS = BB_RANK_5 | BB_RANK_6 | BB_RANK_7 | BB_RANK_8
BB_LOWER_HALF_RANKS = BB_RANK_1 | BB_RANK_2 | BB_RANK_3 | BB_RANK_4
```

Piece

Piece Constants

```
PieceType = int
PIECE_TYPES = [SINGLE, DOUBLE] = [1, 2]
# PIECE_SYMBOLS = (("None", "s", "D"), ("None", "s", "d"))
PIECE_SYMBOLS = (("None", "⛀", "⛁"), ("None", "⛂", "⛃"))
PIECE_NAMES = [None, "single", "double"]
```

Piece Class

```python
class Piece:

  def __init__(self, piece_type=SINGLE, color=True) -> None:
    self.color = color
    self.piece_type = piece_type

  def symbol(self) -> str:
    return PIECE_SYMBOLS[self.color][self.piece_type]

  def __repr__(self) -> str:
    return f"Piece: {self.symbol()}"

  def __str__(self) -> str:
    return f"{self.symbol()}"
```

Move

Move Representation

- Contains **from_square** and **to_square** board index
- Move types
  - boolean move types
    - bear_off
    - transpose
    - impasse
  - crown square index
    - crown

Move Class

```python
class Move:
  def __init__(
    self,
    from_square: Square,
    to_square: Square,
    bear_off: bool = False,
    transpose: bool = False,
    impasse: bool = False,
    crown: Optional[Square] = None,
    ) -> None:

      self.from_square = from_square
      self.to_square = to_square
      self.transpose = transpose
      self.bear_off = bear_off
      self.impasse = impasse
      self.crown = crown
```

Move Representation

Examples:

```
e1c3
d6c7[-><-]
d5d5[X][X]
a5e1[X]
g7f8[-><-][f8*e1]
```

```python
def uci(self) -> str:

  uci = f"{SQUARE_NAMES[self.from_square]}{SQUARE_NAMES[self.to_square]}"

  if self.transpose:
    uci += f"[-><-]"

  if self.impasse:
    uci += f"[X][X]"

  if self.bear_off:
    uci += f"[X]"

  if self.crown is not None:
    uci += f"[{SQUARE_NAMES[self.crown[0]]}*{SQUARE_NAMES[self.crown[1]]}]"

  return uci
```

Move Generation

- generate moves using bitwise operations AND (&), OR (|), XOR (^)
- bit shift from the current move square
- **Sliding move:** iterate over diagonal squares until occupied square is reached
- **Transposition move:** for each double checker check if there is adjacent single below
- **Bear-off move:** check if a checker reached the nearest row
- **Crown move:** (more complicated) after each move, peek if crown is available next

Move Generation - sliding forward moves

Iterate over the ranks and files until you reach a blockage

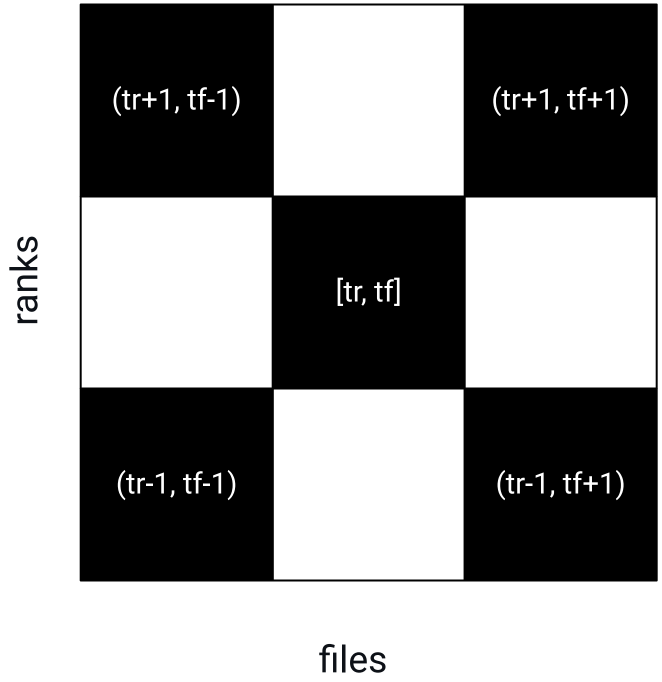
```python
def get_forward_moves(self, square: Square) -> Generator:
  tr, tf = square // 8, square % 8

  # up-right
  for r, f in zip(range(tr+1, 8), range(tf+1, 8)):
    square = r*8 + f
    if (1 << square) & self.occupied:
      break

    yield square

  # up-left
  for r, f in zip(range(tr+1, 8), range(tf-1, -1, -1)):
    square = r*8 + f
    if (1 << square) & self.occupied:
      break

    yield square
```



Move Generation - sliding backward moves

```python
def get_backward_moves(self, square: Square) -> Generator:
  tr, tf = square // 8, square % 8

  # down-right
  for r, f in zip(range(tr-1, -1, -1), range(tf+1, 8)):
    square = r*8 + f
    # if occupied square is faced then stop the slide
    if (1 << square) & self.occupied:
      break

    yield square

  # down-left
  for r, f in zip(range(tr-1, -1, -1), range(tf-1, -1, -1)):
    square = r*8 + f
    if (1 << square) & self.occupied:
      break
    yield square
```

Move Generation - transposition move

```python
def transpose_available(self) -> List:
  available_transpose = []

  bb_singles = set(scan_reversed(self.occupied_co[self.turn] & 
self.singles))
  bb_doubles = scan_reversed(self.occupied_co[self.turn] & self.
doubles)
  for d in bb_doubles:
    if self.turn:
      # down-left adjacent single
      if d-7 in bb_singles:
        available_transpose.append((d-7, d))
      # down-right adjacent single
      if d-9 in bb_singles:
        available_transpose.append((d-9, d))
    else:
      # up-right adjacent single
      if d+7 in bb_singles:
        available_transpose.append((d+7, d))
      # up-left adjacent single
      if d+9 in bb_singles:
        available_transpose.append((d+9, d))
```

Move Generation - bear off move

```python
def bearoff_available(self, to_square: Square) -> Bitboard:
    # a single checker cannot reach the nearest row
    if self.turn:
      return BB_SQUARES[to_square] & BB_RANK_1
    else:
      return BB_SQUARES[to_square] & BB_RANK_8
```

Move Generation - Generate Basic Moves

generate transposition, single and double moves

after each move we check for bear-off (except single checker moves) and then if crown is available

```
if self.turn:
  # TRANSPOSE
  if self.transpose_available():
    available_transposes = self.transpose_available()
    for from_square, to_square in available_transposes:
      if self.bearoff_available(from_square):
        move = Move(from_square, to_square, transpose=True, bear_off=True)
        # check if there is crown available
        crown_moves = self.peek_for_crown(move)
        if crown_moves is not None:
          for cm in crown_moves:
            move.crown = cm
            yield move
        else:
          yield move
      else:
        move = Move(from_square, to_square, transpose=True)
        # check if there is crown available
        crown_moves = self.peek_for_crown(move)
        if crown_moves is not None:
          for cm in crown_moves:
            move.crown = cm
            yield move
        else:
          yield move
```

Move Generation - Generate Crown Moves

```python
def peek_for_crown(self, move: Move) -> List:
  self.push(move)
  self.turn = not self.turn

  crown_moves = self.generate_crown_moves()

  self.pop()

  return crown_moves
```

Move Generation - Generate All Moves

```python
def generate_moves(self) -> List:
  legal_moves = list(self.generate_basic_moves())
  # generate impasse moves
  if not len(legal_moves):
    legal_moves = list(self.generate_impasse_moves())

  return legal_moves
```

Move Generation - Generate Impasse Moves

```python
def generate_impasse_moves(self) -> Generator:
  available_pieces = self.occupied_co[self.turn]  # pieces to remove

  for square in scan_reversed(available_pieces):
    if self.piece_type_at(square) == SINGLE:
      # single remove does not effect on crown status
      yield Move(square, square, impasse=True)
    elif self.piece_type_at(square) == DOUBLE:
      move = Move(square, square, impasse=True)
      # check if there is crown available
      crown_moves = self.peek_for_crown(move)
      if crown_moves is not None:
        for cm in crown_moves:
          move.crown = cm
          yield move
      else:
        yield move
    else:
      yield
```

Evaluation

Evaluation Heuristics

- → total nr. of checkers
- → total nr. of singles
- → total nr. of doubles
- → total nr. of singles in the uppermost half of the board
- → total nr. of doubles in the lowermost half of the board
- → singles disadvantage
- → doubles disadvantage
- → checkers disadvantage

Evaluation Function 1

```python
# Evaluation Function 1

BIAS_SINGLES_ADV = 0.3
BIAS_DOUBLES_ADV = 0.5
BIAS_UPPERMOST_SINGLES = 0.4
BIAS_LOWERMOST_DOUBLES = 0.8

h_value1 = (
    BIAS_SINGLES_ADV * self.total_singles(board_state)
  + BIAS_DOUBLES_ADV * self.total_doubles(board_state)
  + BIAS_UPPERMOST_SINGLES * self.singles_uppermost_halfboard(board_state)
  + BIAS_LOWERMOST_DOUBLES * self.doubles_lowermost_halfboard(board_state)
)
```

Evaluation Function 2

```
# Evaluation Function 2

BIAS_SINGLES_DIS = 0.5
BIAS_DOUBLES_DIS = 0.8
BIAS_CHECKERS_DIS = 0.6

h_value2 = BIAS_SINGLES_DIS * self.singles_disadvantage(board_state) + \
           BIAS_DOUBLES_DIS * self.doubles_disadvantage(board_state) + \
           BIAS_CHECKERS_DIS * self.checkers_disadvantage(board_state)
```

α-β Tree Search

Tree Search

```
def explore_leaves(self, state: Board, valuator: Valuator):
  start = time.time()

  self.valuator.reset()
  current_evaluation = valuator(state)
  search_evaluation, move_evaluation = self.alphabeta_minimax(state, valuator, 0, a=-10000, b=10000, pv=True)

  search_time = time.time() - start

  print(f"{current_evaluation:.2f} -> {search_evaluation:.2f}")
  print(f"Explored {valuator.count} nodes in {search_time:.3f} seconds {int(valuator.count/search_time)}/sec")

  return move_evaluation
```

Alphabeta search

```
    for move in [x[1] for x in moves]:
      state.push(move)
      tree_value = self.alphabeta_minimax(state, valuator, depth+1, a, b)
      state.pop()

      # backpropagate through the principal variation
      if pv:
        move_eval.append((tree_value, move))

      if turn == WHITE:
        ret = max(ret, tree_value)
        a = max(a, ret)
        if a >= b:
          break  # b cut-off
      else:
        ret = min(ret, tree_value)
        b = min(b, ret)
        if a >= b:
          break  # a cut-off

    if pv:
      return ret, move_eval
    else:
      return ret
```

Alphabeta search

```python
def alphabeta_minimax(self, state: Board, valuator: Valuator, depth: int, a, b, pv=False):
  MAX_DEPTH = 7
  if depth >= MAX_DEPTH or state.side_removed_all():
    return self.valuator(state)

  turn = state.turn
  if turn == WHITE:
    ret = -1000
  else:
    ret = 1000

  if pv:
    move_eval = []

  move_ordering = []
  for move in state.legal_moves:
    state.push(move)
    move_ordering.append((self.valuator(state), move))
    state.pop()

  moves = sorted(move_ordering, key=lambda x: x[0], reverse=state.turn)

  # get only top 10 moves if search depth goes beyond 3
  if depth >= 3:
    moves = moves[:10]
```

Live Demo

References

[1] - https://github.com/niklasf/python-chess - A chess library for Python
[2] - http://www.3dkingdoms.com/checkers/bitboards.htm#movegen - Checkers Bitboard Tutorial
[3] - https://youtube.com/playlist?list=PLmN0neTso3Jxh8ZIylk74JpwfiWNI76Cs - [chessprogramming.net] - Chess Game Engine in C

Thank you for your attention