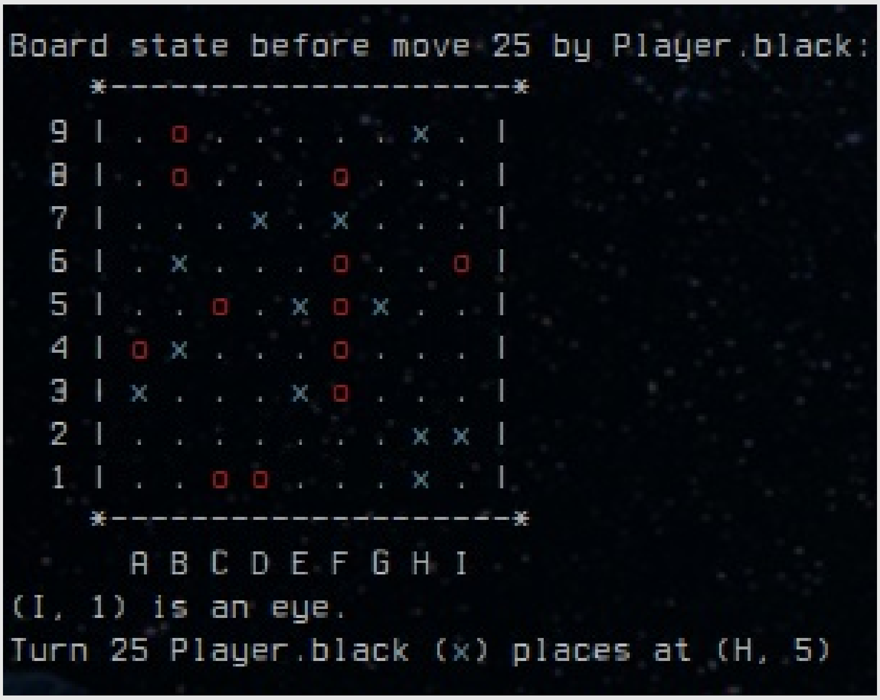
AlphaGo board implementation

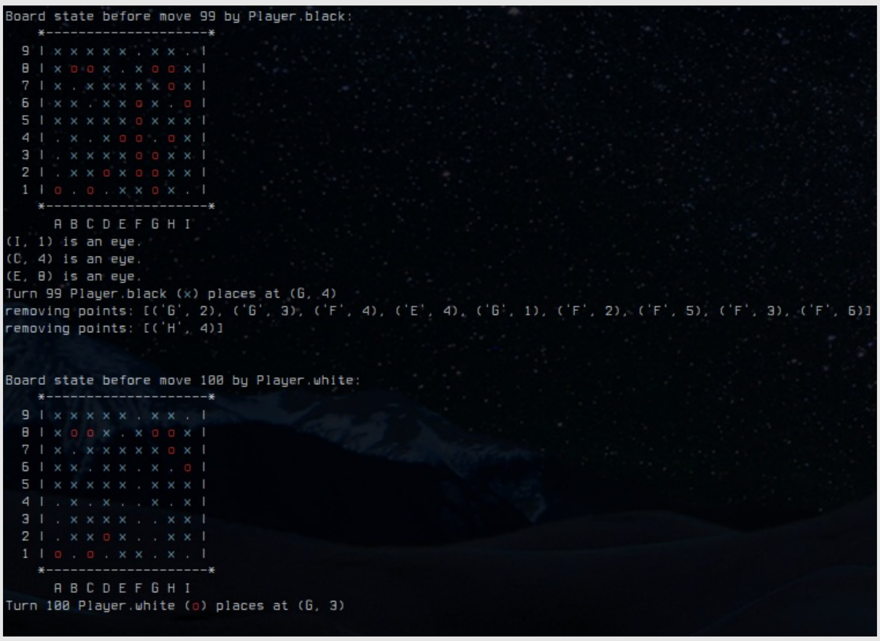
Mitchell Dodson - CS 430 - Dr. Menon

In this assignment, I implemented a Go board with alphago-compatible game states according to the guide by Zhijun Sheng on medium.com.

Output Screenshots



Early stages of a bot-vs-bot game

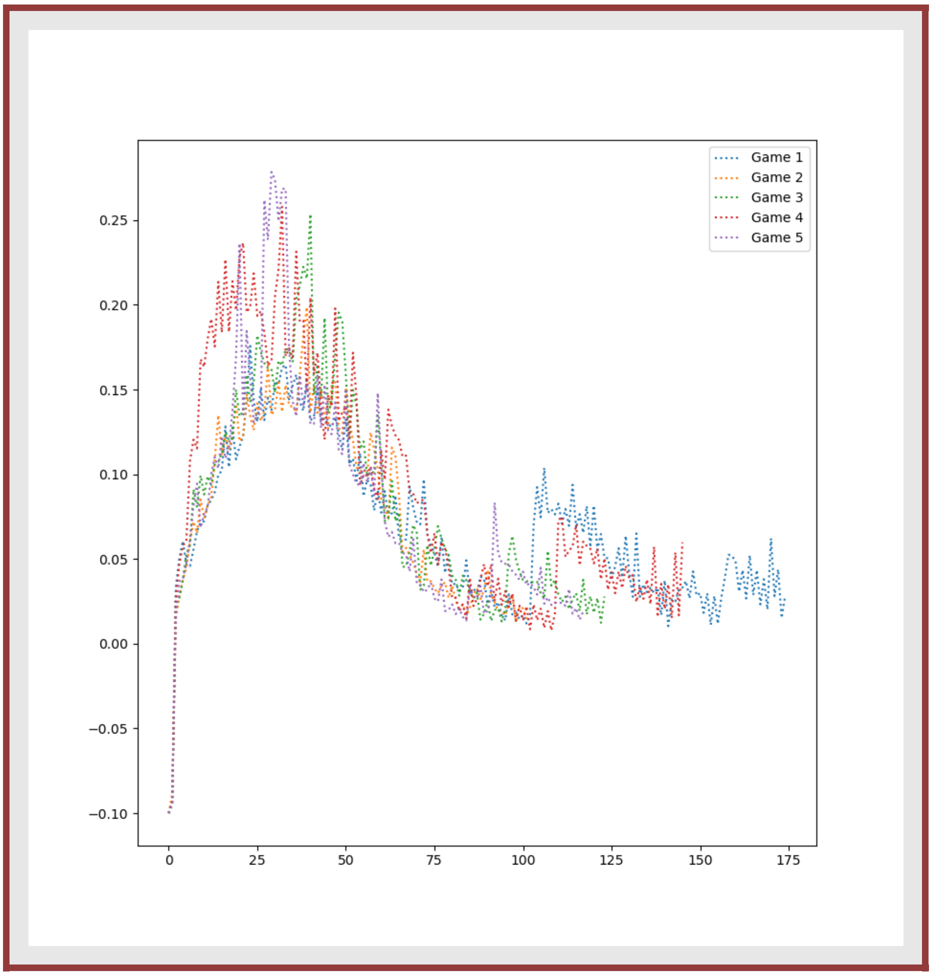


Double-capture made by Black in a testing round

Final state of a bot game on a 19x19 board.

Even with zobrist hashes, the combinatorial explosion of conditions that needed to be checked on each move slowed 19x19 bot games to a halt on my machine, so most of my testing involved 9x9 boards.

As the number of valid moves decreases in the late game, the combinatorial possibilities diminish, causing move frequency to speed back up. The following figure shows the interval of time between moves by each bot with respect to the total number of moves.



Turn time interval wrt move count

I also implemented a human-vs-bot version of the Go game according to the guide. The following is a sample of a game where I tried to claim the entire center of the board (despite the fact this is very inadvisable when playing Go for victory).

Analysis of AlphaGo architecture and comparison to human gameplay

Original implementations of AlphaGo by Google's DeepMind team were initially trained on human gameplay data before being transitioned to learn by playing against itself. The model is a residual convolutional network which uses a 19x19x17 binary matrix with a 19x19 board state matrix for each of the players, a 7-state feature map for each player's history, and an additional 19x19 grid containing a uniform bit indicating the turn.

The output feature vector of the network is used to generate a value representation - a float between zero and 1 - predicting the likelihood of winning the game given the current state, and a policy vector which evaluates the quality of all available moves.

After each real stone placement, the Monte-Carlo tree search component of AlphaGo selects several of the highest-probability moves from the policy vector and repeats the process of evaluating each of the options until ~1,600 hypothetical game states have been tried.

Ultimately, the network chooses the move that optimizes the future value representation.

AlphaGo is especially effective against humans for a few reasons. First, Go is a perfect information game (each player has instant knowledge of the entire game state at every stage, which enables the network to fully simulate future game states without the influence of free uncertain parameters. Additionally, Go is a heavily combinatorial game, with more than 2.081E170 valid game states (for reference, chess has roughly 4.5E46 valid games states). In games like this, humans rely heavily on tradition and heuristics. AlphaGo, on the other hand, is simply able to evaluate vastly more future game states than humans are capable of. Even in early game stages when humans are playing standard opening moves, AlphaGo is known to make stone placement decisions that humans would rarely consider, and which impact the short-term future states of the game in ways that seem unpredictable to human players. For example, the latest iteration of AlphaGo (AlphaGo Zero) was trained entirely using gameplay against itself - no human data whatsoever. Initially, it rediscovered openings (like 4/5 and 3/3) commonly used by humans, but ultimately diverged from these and discovered abstract and highly state-sensitive openings that humans rarely if ever used, but which were shockingly more effective.

Implementation screenshots

Bot vs Bot

```python
from gogame.goboard import GameState, Move
from gogame.gotypes import Player, Point
from gogame.agent.naive import RandomBot
from gogame.utils import start_bots_game, update_times

# Board size and update frequency. Assumes board is square
size = 9
freq = .1
timefile = "/home/krttd/uah/22.s/cs430/hw3/turn_times.pkl"

capture_sequence = [
    Move.play(Point(row=5, col=10)),
    Move.play(Point(row=6, col=10)),
    Move.play(Point(row=5, col=11)),
    Move.play(Point(row=6, col=11)),
    Move.play(Point(row=6, col=9)),
    Move.play(Point(row=1, col=1)),
    Move.play(Point(row=6, col=12)),
    Move.play(Point(row=2, col=1)),
    Move.play(Point(row=7, col=10)),
    Move.play(Point(row=3, col=1)),
    Move.play(Point(row=7, col=11)),
    Move.play(Point(row=4, col=1))]

if __name__=="__main__":
    for i in range(5):
        times = start_bots_game(
            board_size=size,
            update_freq=freq,
            bot_a=RandomBot(size),
            bot_b=RandomBot(size),
            #sequence=capture_sequence
            )
        update_times(times, timefile)
```

Human vs Bot

```python
from gogame.agent.naive import RandomBot
from gogame.goboard import GameState, Move
from gogame.gotypes import Player
from gogame.utils import print_board, print_move, point_from_coords

def start_game():
    board_size = 9
    game = GameState.new_game(board_size)
    bot = RandomBot(board_size)
    c=1
    while not game.is_over():
        print(chr(27)+"[2J")
        print_board(game.board, False)
        if game.next_player == Player.black:
            human_move = input("-- ")
            point = point_from_coords(human_move.strip())
            move = Move.play(point)
        else:
            move = bot.get_move(game)
        print_move(game.next_player, move, c)
        game = game.apply_move(move)
        c+=1

if __name__=="__main__":
    start_game()
```

goboard code

```python
class Board():
    def __init__(self, row_count, col_count):
        self._nrows = row_count
        self._ncols = col_count
        self._grid = {}
        self._hash = zobrist.EMPTY_BOARD

    def get_row_count(self):
        return self._nrows

    def get_col_count(self):
        return self._ncols

    def put_stone(self, player, point, real_move):
        #  Make sure the provided point is unoccupied and on the grid
        assert self.is_on_grid(point)
        assert self._grid.get(point) is None
        adjacent_same_color = []
        adjacent_opposite_color = []
        liberties = []
        for neighbor in point.neighbors():
            #  Don't consider points off the grid.
            if not self.is_on_grid(neighbor):
                continue
            neighbor_string = self._grid.get(neighbor)
            if neighbor_string is None:
                liberties.append(neighbor)
            #  Append neighbors of the same color to the appropriate array
            elif neighbor_string.color == player:
                if neighbor_string not in adjacent_same_color:
                    adjacent_same_color.append(neighbor_string)
            #  Append neighbors of opposite color to the appropriate array
            else:
                if neighbor_string not in adjacent_opposite_color:
                    adjacent_opposite_color.append(neighbor_string)
        #  Make a new gostring for this point, assigning its liberties.
        new_string = GoString(player, [point], liberties)
        for same_color_string in adjacent_same_color:
            new_string = new_string.merged_with(same_color_string)
        #  Set all stones in this group to the new merged string
        for new_string_point in new_string.stones:
            self._grid[new_string_point] = new_string
        self._hash ^= zobrist.HASH_CODE[point, player]
        #  Remove the liberty of this point from the adjacent opposite group
        for other_color_string in adjacent_opposite_color:
            replacement = other_color_string.without_liberty(point)
            if replacement.num_liberties:
                self._replace_string(replacement)
            else:
                self._remove_string(other_color_string, real_move)
            other_color_string.remove_liberty(point)
        """
        #  Remove adjacent opponent strings with no liberties remaining.
        for other_color_string in adjacent_opposite_color:
            if other_color_string.num_liberties == 0:
                self._remove_string(other_color_string, real_move)
        """

    def _replace_string(self, new_string):
        """ Replace an entire former string with a new one. """
        for point in new_string.stones:
            self._grid[point] = new_string
```

```python
    def _remove_string(self, string, real_move):
        """ Remove a string once it has been surrounded. """
        if real_move: print("removing points:",[
            (s.col_letter(), s.row) for s in string.stones])
        for point in string.stones:
            for neighbor in point.neighbors():
                neighbor_string = self._grid.get(neighbor)
                if neighbor_string is None:
                    continue
                if neighbor_string is not string:
                    self._replace_string(neighbor_string.with_liberty(point))
                    #neighbor_string.add_liberty(point)
            self._grid[point] = None
            self._hash ^= zobrist.HASH_CODE[point, string.color]

    def zobrist_hash(self):
        return self._hash

    def get_stone(self, point):
        string = self._grid.get(point)
        if string is None: return None
        return string.color

    def is_on_grid(self, point):
        return 1<=point.row<=self._nrows and 1<=point.col<=self._ncols

    def get_go_string(self, point):
        """ Return the contiguous string associated with a point """
        string = self._grid.get(point)
        if string is None: return None
        return string
```

```python
class GameState():
    def __init__(self, board, next_player, previous, last_move):
        self.board = board
        self.next_player = next_player
        self.previous_state = previous
        self.last_move = last_move
        if self.previous_state is None:
            self.previous_states = frozenset()
        else:
            self.previous_states = frozenset(previous.previous_states |
                    {(previous.next_player, previous.board.zobrist_hash())})

    @classmethod
    def new_game(cls, board_size):
        if isinstance(board_size, int):
            board_size = (board_size, board_size)
        board = Board(*board_size)
        return GameState(board, Player.black, None, None)

    def apply_move(self, move):
        """ Apply a Move class to the board. """
        #  If the move places a stone, make a new board with the chosen
        #  stone placement and generate a new GameState
        if move.is_play:
            next_board = copy.deepcopy(self.board)
            next_board.put_stone(self.next_player,
                    move.point, real_move=True)
        #  If the next move doesn't involve placing a stone, the new game
        #  state is the same as the onld one.
        else:
            next_board = self.board
        #  Generate a new GameState with the updated board, marking this
        #  board as the previous one.
        return GameState(next_board, self.next_player.opponent, self, move)

    def is_over(self):
        #  The game can't be over on the first move.
        if self.last_move is None:
            return False
        #  If the opponent resigned, the game is over.
        if self.last_move.is_resign:
            return True
        #  If both players pass, the game is stalemated.
        second_last_move = self.previous_state.last_move
        if second_last_move is None:
            return False
        return self.last_move.is_pass and second_last_move.is_pass

    def is_valid_move(self, move):
        if self.is_over(): return False
        if move.is_pass or move.is_resign:
            return True
        return self.board.get_stone(move.point) is None \
                and not self.is_move_self_capture(self.next_player, move) \
                and not self.does_move_violate_ko(self.next_player, move)

    def is_move_self_capture(self, player, move):
        """ Determine if a move results in suicide, which is illegal """
        if not move.is_play: return False
        next_board = copy.deepcopy(self.board)
        next_board.put_stone(player, move.point, real_move=False)
        new_string = next_board.get_go_string(move.point)
        return not new_string.num_liberties
```

```python
    @property
    def situation(self):
        """ Return a tuple wrapping the player and the board state """
        return (self.next_player, self.board)

    def does_move_violate_ko(self, player, move):
        """ Determine if a move sequence has been repeated """
        if not move.is_play: return False
        next_board = copy.deepcopy(self.board)
        next_board.put_stone(player, move.point, real_move=False)
        next_sit = (player.opponent, next_board.zobrist_hash())
        return next_sit in self.previous_states
```

```python
class Move():
    """ Class representing all possible game actions """
    def __init__(self, point=None, is_pass=False, is_resign=False):
        assert (point is not None) ^ is_pass ^ is_resign
        self.point = point
        self.is_play = self.point is not None
        self.is_pass = is_pass
        self.is_resign = is_resign

    @classmethod
    def play(cls, point):
        """ Return a move putting a stone at the provided point """
        return Move(point=point)

    @classmethod
    def pass_turn(cls):
        """ Return a pass move. """
        return Move(is_pass=True)

    @classmethod
    def resign(cls):
        """ Return a resignation move. """
        return Move(is_resign=True)

class Board():
    def __init__(self, row_count, col_count):
        self._nrows = row_count
        self._ncols = col_count
        self._grid = {}
        self._hash = zobrist.EMPTY_BOARD

    def get_row_count(self):
        return self._nrows

    def get_col_count(self):
        return self._ncols
```

gotypes code

```python
import enum
from collections import namedtuple

class Point(namedtuple("Point", "row col")):
    def neighbors(self):
        return [
            Point(self.row-1, self.col),
            Point(self.row+1, self.col),
            Point(self.row, self.col-1),
            Point(self.row, self.col+1)
            ]
    def col_letter(self):
        return chr(ord('@')+self.col)

class Player(enum.Enum):
    black = 1
    white = 2

    @property
    def opponent(self):
        return Player.black if self==Player.white else Player.white
```

utils code

```python
# Maps each player to their stone character
stone_to_char = {
        None:'.',
        Player.black:Fore.BLUE+'x'+Fore.WHITE,
        Player.white:Fore.RED+'o'+Fore.WHITE,
        }

# Print the current board state. Assumes board is square
def print_board(board, clear_on_move=True):
    cols = [chr(ord('@')+n) for n in range(1,board.get_row_count()+1)]
    if clear_on_move: print(chr(27)+"[2J")
    print(4*' '+'*'+(board.get_col_count()+1)*'--'+"*")
    for i in range(board.get_row_count(), 0, -1):
        line = []
        for j in range(1, board.get_col_count()+1):
            stone = board.get_stone(Point(i,j))
            line.append(stone_to_char[stone]+" ")
        print(f"{i:3} | {''.join(line)}|")
    print(4*' '+'*'+(board.get_col_count()+1)*'--'+"*")
    print(6*' '+' '.join(cols))

# Print this player and their move.
def print_move(player, move, turn):
    """ print a string describing the next move. """
    if move.is_pass: move_str = "passes this turn"
    elif move.is_resign: move_str = "resigns"
    else: move_str = f"({move.point.col_letter()}, {move.point.row})"
    print(f"Turn {turn} {player} ({stone_to_char[player]}) places at {move_str}")

def update_times(turn_times, timefile):
    with open(timefile, "rb") as timefp:
        times = pkl.load(timefp)
    times.append(turn_times)
    with open(timefile, "wb") as timefp:
        pkl.dump(times, timefp)
```

```python
# Initialize the game with bot a and bot b
def start_bots_game(board_size, update_freq, bot_a, bot_b,
        sequence=None, capture_turntime=False):
    bots = { Player.white:bot_a, Player.black:bot_b }
    game = GameState.new_game(board_size)
    c = 1
    times = [0]
    last_time = time.time()
    while not game.is_over():
        # Determine which player's turn it is.
        print(f"\n\nBoard state before move {c} by {game.next_player}:")
        print_board(game.board, False)

        # Get this player's next move and play it
        if sequence == None:
            move = bots[game.next_player].get_move(game)
        elif sequence != None and len(sequence):
            move = sequence.pop(0)
        else:
            break
        print_move(game.next_player, move, c)
        game = game.apply_move(move)
        new_time = time.time()
        times.append(new_time-last_time)
        last_time = new_time
        time.sleep(update_freq)
        c+=1
    return times

def point_from_coords(coords):
    return Point(int(coords[1:]), ord(coords[0].lower())-96)
```

zobrist

```python
from .gotypes import Player, Point

__all__ = ['HASH_CODE', 'EMPTY_BOARD']

HASH_CODE = {
    (Point(row=1, col=1), Player.black): 7212332594980478893,
    (Point(row=1, col=1), Player.white): 6059206809261739078,
    (Point(row=1, col=2), Player.black): 5806050773891761297,
    (Point(row=1, col=2), Player.white): 5259041124309758012,
    (Point(row=1, col=3), Player.black): 477090242065025506,
    (Point(row=1, col=3), Player.white): 6584207647330890939,
    (Point(row=1, col=4), Player.black): 6894800654608549930,
    (Point(row=1, col=4), Player.white): 3637721230678135479,
    (Point(row=1, col=5), Player.black): 4276519716013619409,
    (Point(row=1, col=5), Player.white): 2299044066014256902,
    (Point(row=1, col=6), Player.black): 5537841095575436401,
    (Point(row=1, col=6), Player.white): 4933952713071900042,
    (Point(row=1, col=7), Player.black): 6391828484223924812,
    (Point(row=1, col=7), Player.white): 5707451722261733567,
    (Point(row=1, col=8), Player.black): 4680415496171866016,
    (Point(row=1, col=8), Player.white): 447972796156814551,
    (Point(row=1, col=9), Player.black): 8234134455310916159,
    (Point(row=1, col=9), Player.white): 8560110109950909353,
    (Point(row=1, col=10), Player.black): 5729324340100812510,
    (Point(row=1, col=10), Player.white): 4147802520603030650,
    (Point(row=1, col=11), Player.black): 842466009092063129,
    (Point(row=1, col=11), Player.white): 839309629037804159,
    (Point(row=1, col=12), Player.black): 6098026410620661069,
    (Point(row=1, col=12), Player.white): 3768770469021103446,
    (Point(row=1, col=13), Player.black): 4406425757099865654,
    (Point(row=1, col=13), Player.white): 2902360772062761228,
    (Point(row=1, col=14), Player.black): 7979590031999474709,
    (Point(row=1, col=14), Player.white): 7909150663214206314,
    (Point(row=1, col=15), Player.black): 3109297293426772277,
    (Point(row=1, col=15), Player.white): 4449806350097963898,
    (Point(row=1, col=16), Player.black): 6659405935430007340,
    (Point(row=1, col=16), Player.white): 691150618054807320,
    (Point(row=1, col=17), Player.black): 4360404964072595076,
    (Point(row=1, col=17), Player.white): 3578928140360233120,
    (Point(row=1, col=18), Player.black): 807213122545822075,
    (Point(row=1, col=18), Player.white): 4319259250607696416,
    (Point(row=1, col=19), Player.black): 9123139402105021339,
    (Point(row=1, col=19), Player.white): 4726219459777586225,
    (Point(row=2, col=1), Player.black): 416153434590373417,
    (Point(row=2, col=1), Player.white): 891067034413293237,
    (Point(row=2, col=2), Player.black): 4661950264302637853,
    (Point(row=2, col=2), Player.white): 7553060695651571216,
    (Point(row=2, col=3), Player.black): 756734769702167786,
    (Point(row=2, col=3), Player.white): 616641553161291071,
    (Point(row=2, col=4), Player.black): 4044278316241465689,
    (Point(row=2, col=4), Player.white): 1503432405700872709,
    (Point(row=2, col=5), Player.black): 7851008194925520208,
    (Point(row=2, col=5), Player.white): 3576544079062910919,
    (Point(row=2, col=6), Player.black): 1313084040340167561,
    (Point(row=2, col=6), Player.white): 3425209199765721605,
    (Point(row=2, col=7), Player.black): 2802108139305482073,
    (Point(row=2, col=7), Player.white): 8506417616025360340,
    (Point(row=2, col=8), Player.black): 580050731873902171,
    (Point(row=2, col=8), Player.white): 8680690093303335843,
    (Point(row=2, col=9), Player.black): 3617607699978604931,
    (Point(row=2, col=9), Player.white): 1126537609054539541,
    (Point(row=2, col=10), Player.black): 4784491323483384969,
    (Point(row=2, col=10), Player.white): 5317263329063746477,
    (Point(row=2, col=11), Player.black): 349987563015686602,
    (Point(row=2, col=11), Player.white): 7722729721104486794,
```