Project Option 2

CSE 415: Introduction to Artificial Intelligence
The University of Washington, Seattle, Autumn 2019

OPTION 1: Baroque Chess Agents with Zobrist Hashing.

This option is might be attractive to students interested in a deeper experience with 2-person zero sum, non-stochastic game-playing.

A big part of this option's challenge is writing the move generator for Baroque Chess. The capturing rules for each piece are different. To make this a lot easier, we are making "Imitator" pieces optional in the game. So you do not have to handle imitators, which are the most complex of the pieces.

Starter code and a move-testing service are available for this option.

Your player's file name should be of the form [free_name]_BC_Player.py, where you get to choose an original name, but that name is followed by "_BC_Player.py". An example might be Magnifico_BC_Player.py. If you need to split your agent to use any additional modules, please name them using the convention: [free_name]_BC_module_[module_name].py. Here an example could be Magnifico_BC_module_static_eval.py.

Game rules: A good overview of the rules of Baroque Chess is given in the Wikipedia article on Baroque Chess. However, since there are a number of variations of Baroque Chess, we have to specify some version, and we will use an operational definition of the rules. "When in doubt, try it out, at the validation page."

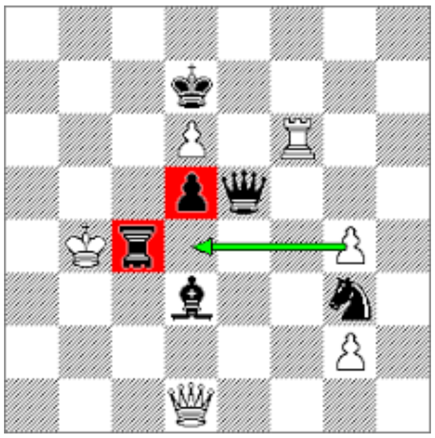

Image courtesy of Chess Guide at multionline.net.

It will be essential that your program use the specified representation of states, so that it is compatible with the other Baroque Chess programs developed in the class. Some code for representing states of Baroque Chess is shown near end of this page. (This is included as BC_state_etc.py in the Starter code.)

Your program should be designed to anticipate time limits on moves. There are two aspects to this: (1) use iterative deepening search, and (2) poll a clock frequently in order to return a move before time runs out.

Starter code is available. This code provides a game-master to handle the turn-taking between a pair of playing agents, and it provides a class definition for states of the game. Some simple agent stub files are included to help get the project going.

We are playing Baroque Chess with rules that: do not permit (a) any choice of center-counter symmetry (see the Wikipedia description of Baroque Chess), or (b) long-leaper double jumping (it's considered detrimental to having balanced and dynamic games), or (c) "suicide" moves. We will assume that the game ends when either player loses a king.

Your program should implement the following functions, so that they can be called by the game administration software:

- **`prepare(player2Nickname)`**. This function will be called by the game administration software before a game starts, and usually before any of the other functions below are called. If your agent needs to initialize any data structures, this is a good place for that to be done, as the time taken to execute this will not be "on the clock" that is running during game moves.

- **`parameterized_minimax(currentState, alphaBeta=False, ply=3, useBasicStaticEval=True, zobrist=True)`**. This function is for testing the basic capabilities of your agent. It should be able to work for any combination of values of arguments, within reason. For example, it should be OK with any legal state for `currentState`; either using or not using alpha-beta pruning; a specified number of ply (maximum depth) from 0 (statically evaluate the current state only) to perhaps 8 (which might not be too bad if there are only a few pieces left on the board); and Zobrist hashing enabled or not.

When `useBasicStaticEval` is true, you'll evaluate leaf nodes of your search tree with your own implementation of the following function: White pincers are worth 1, the White king is worth 100, and all other White pieces are worth 2. Black pieces have the same values as their white counterparts, but negative. When useBasicStaticEval is False, you should use your own, more discriminating function. The value of the function is the sum of the values of the pieces on the board in the given state.

Do incorporate Zobrist hashing in your agent. However, it should be disabled while running the `parameterized_minimax` function if the parameter `zobrist` has value `False`. Your implementation of Zobrist hashing should include both basic functionality and instrumentation. Basic functionality includes the `put`, and `get` methods, which store and retrieve items from the hash table. A typical item is a tuple of the form `( state, ply_used, value, n_static_evals)`. The state is a BC_state object. The ply_used item is an integer telling how many ply of lookahead were used to determine the value; it should be 0 if the value is the static value of state. The value is the saved result of minimax, alpha-beta, or static-eval. The n_static_evals is an int giving the number of leaf nodes of this subtree for which the static-evaluation function was applied. If ply_used is 0, then n_static_evals should be 1. The instrumentation component of your Zobrist hashing implementation is simply keeping track, during each turn or each call to parameterized minimax, and overall within each game, how much computation is being performed or saved, due to Zobrist hashing. Each time a game is started the following counts should be initialized to 0: `game_put_count`, `game_get_success_count`, `game_get_failure_count`, `game_collision_count`, and `game_static_evals_saved_count`. Similarly, each time a move computation starts or a call is started to `parameterized_minimax`, the following counts should be initialized to 0: `move_put_count`, `move_get_success_count`, `move_get_failure_count`, `move_collision_count`, and `move_static_evals_saved_count`. Each time a `put` operation is performed, each of the two "put-count" variables should be incremented. Handle `get` operation counts similarly, except that the successful and unsuccessful operations should be handled separately. Collisions can happen when putting or getting, and the counts should include both kinds. The varible counting static evals saved will total up the numbers of static evaluations from the last field of the tuples whenever a `get` operation is performed.

In order for `parameterized_minimax` to be a good testing function, the order for move generation must be controlled. Although your agent may generate successors in any order when playing a game, in `parameterized_minimax` it must use a prescribed order. Here is that order, based on the source and destination squares for each move. This order should be respected whether the agent is playing White or Black. Moves of a piece at (a, 1) come before moves of a piece at (a, 2), which come before moves of a piece at (a, 3), etc., which come before moves at (b, 1), etc. Among all moves of a piece from (M, N), we break ties according to the destination squares, using the same ordering as for source squares. Here is the board layout:

a b c d e f g h
8 7 6 5 4 3 2 1

Although your `makeMove` function should use IDDFS to be able to come up with a move no matter how much time is given, you should NOT use IDDFS in `parameterized_minimax`.

The return value of `parameterized_minimax` should be a `dict` object with the following attributes, in addition to hashing counts already mentioned:

- 'CURRENT_STATE_STATIC_VAL': The static eval value of the current_state as determined by your minimax search
- 'N_STATE_EXPANSIONS': The number of state expansions as part of your minimax search. An expansion takes place when the set of successors of a state are computed.
- 'N_STATIC_EVALS': The number of static evals performed as part of your minimax search
- 'N_CUTOFFS': The number of cutoffs that occurred during the minimax search (0 if alpha-beta was not enabled)

- **`introduce()`**. This function will return a multiline string that introduces your player, giving its full name (you get to make that up), the names and UWNetIDs of its creators (you), and (optionally) some words to describe its character.

- **`nickname()`**. This function should return a short version of the playing agent's name (16 characters or fewer). This name will be used to identify the player's moves in game transcripts.

- **`makeMove(currentState, currentRemark, timeLimit=10)`**. This is probably your most important function. It should return a list of the form `[[move, newState], newRemark, plies_completed, zobrist_stats]`. The move is a data item describing the chosen move, of the form ((r, c),(r1, c1)). Here (r, c) gives the starting row and column coordinates of the piece that is being moved, and (r1, c1) give the coordinates of the square where it ends up. The `zobrist_stats` value is a dict giving the various counts for that move that are described above. They can be useful when you are assessing the benefits of the Zobrist hashing.

The `newState` is the result of making the move from the given currentState. It must be a complete state (an instance of class `BC_state`) and not just a board.

The `currentRemark` argument is a string representing a remark from the opponent on its last move. You may (optionally) use this as input to your own agent's remark.

The `timeLimit` represents the number of seconds available for computing and returning the move.

The `newRemark` to be returned must be a string. During a game, the strings from your agent and its opponent comprise a dialog. These remarks should reflect a "personality" of your agent. They could be canned remarks from a list of 10 or more, but you may make this more elaborate and context-sensitive, commenting on the current state or the direction in which the game seems to be heading.

Your `makeMove` function should use IDDFS so that it can implement an "anytime" version of the mimimax search, and it should check the clock often enough to make sure it can return its best move found so far, if there is little time left.

- **`staticEval(state)`**. This function will perform a static evaluation of the given state. The value returned should be high if the state is good for WHITE and low if the state is good for BLACK. This function is not the same as your basic static evaluation function mentioned above in association with `parameterized_minimax`, but should be custom-designed by your team to help your agent be a strong player. The staff plans to test your `staticEval` function separately at some point during the grading, so it should be possible to use it by executing code such as the following. (This example assumes your agent is named "The_Roman_BC_Player".)

```
import The_Roman_BC_Player as player
staticResult = player.staticEval(some_state)
```

The quality of your agent's playing will probably depend heavily on how well your staticEval function works.

In the code example above, the starting board is shown using ASCII text, and the encoding is as follows: (lower case for black, upper case for WHITE):

```
p: pincer
l: leaper
i: imitator
w: withdrawer
k: king
c: coordinator
f: freezer
-: empty square on the board
```

Here are the contents of the starter code file `BC_state_etc.py`. It's presented here for convenient references, but it is part of the .tar archive linked above.

```
BLACK = 0
WHITE = 1

INIT_TO_CODE = {'p':2, 'P':3, 'c':4, 'C':5, 'l':6, 'L':7, 'i':8, 'I':9,
  'w':10, 'W':11, 'k':12, 'K':13, 'f':14, 'F':15, '-':0}

CODE_TO_INIT = {0:'-',2:'p',3:'P',4:'c',5:'C',6:'l',7:'L',8:'i',9:'I',
  10:'w',11:'W',12:'k',13:'K',14:'f',15:'F'}

def who(piece): return piece % 2

def parse(bs): # bs is board string
  '''Translate a board string into the list of lists representation.'''
  b = [[0,0,0,0,0,0,0,0] for r in range(8)]
  rs9 = bs.split("\n")
  rs8 = rs9[1:] # eliminate the empty first item.
  for iy in range(8):
    rss = rs8[iy].split(' ');
    for jx in range(8):
      b[iy][jx] = INIT_TO_CODE[rss[jx]]
  return b

INITIAL = parse('''
c l i w k i l f
p p p p p p p p
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
P P P P P P P P
F L I W K I L C
''')

INITIAL_NO_IMITATORS = parse('''
c l - w k - l f
p p p p p p p p
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
P P P P P P P P
F L - W K - L C
''')


class BC_state:
  def __init__(self, old_board=INITIAL, whose_move=WHITE):
    new_board = [r[:] for r in old_board]
    self.board = new_board
    self.whose_move = whose_move;

  def __repr__(self):
    s = ''
    for r in range(8):
      for c in range(8):
        s += CODE_TO_INIT[self.board[r][c]] + " "
      s += "\n"
    if self.whose_move==WHITE: s += "WHITE's move"
    else: s += "BLACK's move"
    s += "\n"
    return s

  def __eq__(self, other):
    if not (type(other)==type(self)): return False
    if self.whose_move != other.whose_move: return False
    try:
      b1 = self.board
      b2 = other.board
      for i in range(8):
        for j in range(8):
          if b1[i][j] != b2[i][j]: return False
      return True
    except Exception as e:
      return False

def test_starting_board():
  init_state = BC_state(INITIAL, WHITE)
  print(init_state)

test_starting_board()
```

What to Turn In:

Turn in your files via Canvas. Do not zip up the files.

1. Your agent file(s). See the naming requirements at the beginning of this page. If Canvas renames your files, we will fix them.
2. **Report.pdf**: In addition to the items required in the general project report, include these option-specific items: Intended personality of the agent, if any; Status of agent -- features that work and any that don't; Design retrospective of about 200 to 500 words describing how you designed your agent. Zobrist hashing details. Where in your move-making did you include put and get calls?

How much did your program benefit from the using of the hashing? Was your agent ever able to search one or more ply deeper within the same time limit, because of Zobrist hashing? ; Optional partnership retrospective including how you operated as a team, what issues you had in terms of collaboration, and how you did or did not overcome those issues, and what if anything you learned in terms of collaboration. A thoughtful parnership retrospective is worth 3 to 5 points of extra credit, and a 5-point one will typically include a group portion and an individual statement of several lines of text from each of the two partners.

3. **Transcript.txt**: One game transcript of your player playing against another team's agent. To qualify for entering the tournament, you need to turn in a transcripts, representing a game with a different player, and your agent must be the winner. Note: If your agent A, plays agent B several times, winning at least once, you can use the transcript of that winning game as your qualifying transcript. One additional requirement to qualify for the tournament is that your agent must not make illegal moves in any games, whether in the transcripts or not. (It might not always make a move when a move exists, and that could possibly mean losing a game, but failing to make a legal move when one exists is not in itself disqualifying.)

Updates and Corrections: Any changes will be described here and/or in ED.