1. Title of project

Option 1: Baroque Chess Agents with Zobrist Hashing

2. Names and roles of each teammate

- Yu Fan (fany23): basic static evaluation function; custom static evaluation function; defining *makeMove* and the move of each piece; debugging
- Dan Wang (daw1230): Zobrist hashing; IDDFS; minimax search and alpha-beta pruning; drafting the report; debugging

3. What the program is supposed to do

The program describes an agent that plays Baroque Chess without imitator.

4.Techniques used and brief description of how those techniques work

- Zobrist hashing. For a given board state, if there is a piece on a given cell, we use the random number of that piece from the corresponding cell in the table. When a player makes a move, it is not necessary to recalculate the hash value from scratch. Instead we simply use few XOR operations to recalculate the hash value.
- Minimax search. If looking ahead two moves, we will be considering the positions that our opponent can get to in one move, from each of the positions we can get to in one move. Assuming that the opponent is playing rationally, the opponent Min will minimize the value of the resulting board. Therefore, instead of using the static value at each successor of the current state, we examine the successors of each of those, computing their static values, and take the minimum of those as the value of our successor.
- IDDFS. IDDFS does a sequence of DFS executions, but with a depth limit for each execution. We make the depth limit increase by 1 in each execution, starting from 0.
- Alpha-beta pruning. An alpha (beta) cutoff occurs at a Maximizing (minimizing) node when it is known that the maximizing (minimizing) player has a move that results in a value alpha (beta) and, subsequently, when an alternative to that move is explored, it is found that the alternative gives the opponent the option of moving to a lower (higher) valued position. Any further exploration of the alternative can be canceled.

5. An interesting sample session

One transcript of our agent playing against *Gary_BC_Player.py* is also uploaded.

6. Brief demo instructions

In *BaroqueGameMaster_V02.py*, *makeMove* is called recursively in a competing Baroque chess game. As our most import function, *makeMove* uses almost all the AI techniques designed in this project, such as IDDFS, Zobrist hashing, custom static evaluation, time limit, and minimax search with alpha-beta pruning.

parameterized_minimax is for testing the basic capabilities of the agent. It can work for any combination of argument values: alpha-beta pruning enabled or not; Zobrist hashing enabled or not; useBasicStaticEval enabled or not.

7. Code excerpt showing Zobrist hashing and some explanation of it

From *DD_BC_module_zobrist_hashing.py*

```
import random

PIECES = {2:'p',3:'P',4:'c',5:'C',6:'l',7:'L',10:'w',11:'W',12:'k',13:'K',14:'f',15:'F'}
# We are not considering imitator (8:'i',9:'I') in DD_BC_Player.py
TABLE = []
zobrist_table = {}

# Assign a random number to initial board
def init_table():
    global TABLE
    for row in range(8):
        rowhash = []
        for column in range(8):
            cell = {}
            for piece in PIECES:
                cell[piece] = random.getrandbits(64)
            rowhash.append(cell)
        TABLE.append(rowhash)

# Return the hash value for a given state
def hash_state(state):
    global TABLE
    h = 0
    board = state.board
    for row in range(8):
        for column in range(8):
            piece = board[row][column]
            if piece != 0:
                h ^= TABLE[row][column][piece]
    return h
```

In Zobrist Hashing, for a given board state, if there is a piece on a given cell, we use the random number of that piece from the corresponding cell in the table. More bits the random number has, lesser chance a hash collision would occur. Therefore, 64-bit numbers are commonly used, and a hash collision is unlikely to occur with such large numbers. The TABLE is initialized once during the program execution. When a player makes a move, it is not necessary to recalculate the hash value from scratch. Instead we simply use few XOR operations to recalculate the hash value.

8. What we learn in this project

- Yu has learned how to define basic and customized static evaluation function, makeMove, and the move of each piece in Baroque chess.
- Dan has learned how to implement Zobrist hashing, IDDFS, mini-max search, and alpha-beta pruning.
- We both have learned how to cooperate a coding task as a team.

9. What to add if we had more time

We would definitely consider imitator move if we have more time.

10. References

- Wikipedia article
  https://en.wikipedia.org/wiki/Baroque_chess
  We learn an introduction of Baroque Chess from the website.
- Logic Mazes
  http://www.logicmazes.com/games/wgr.html
  We learn what is and what is wrong with Ultima.
- GeeksforGeeks
  https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-5-zobrist-hashing/
  We learn Zobrist hashing from the website.
- Course website
  https://courses.cs.washington.edu/courses/cse415/19au/
  The course website provides a lot of information regarding minimax search, alpha-beta pruning, IDDFS, and Zobrist hashing.

11. Partners' Reflections

- Yu Fan
  This is general interesting project. I mainly did formulate moves and static evaluation method as well as debugging and testing. The collaborative developing experience with Dan is fun and productive. We use github for the version control, and it works fine for both of us. Dan is an extraordinary teammate to work with.
- Dan Wang
  I'm mainly in charge of drafting the final report, debugging, and the design of Zobrist hashing, IDDFS, mini-max search, and alpha-beta pruning. Cooperating with Yu is very delightful. We all have own pace, but it works out perfectly. Also, we use GitHub to synchronize our work, which is very helpful.
- Group
  As a first step, we partition the project into small tasks and assign each member some specified tasks to be completed. It is efficient for us to first focus on our own tasks and do some coding work alone. But it is always helpful when you know there is someone you can discuss with. As a team, we work together to integrate separate tasks and debug. We learn a lot, especially teamwork, from this project.

12. Option-specific items

i) Intended personality of the agent

Our agent would move based on the hash value or static evaluation. We would call it a predictor.

ii) Status of agent -- features that work and any that don't

Features that work include minimax search, alpha-beta pruning, IDDFS, Zobrist hashing, basic static evaluation, and custom static evaluation.

iii) Design retrospective of about 200 to 500 words describing how you designed your agent.

Our agent is built upon the one that plays Toro-Tile Straight in assignment 5. We use here the same ideas of minimax search and alpha-beta pruning. Starting from the given scratch, we first figure out the basic structure of the code, including functions such as *prepare*, *parameterized_minimax*, *makeMov*e, *staticEval*, *findAllMove*, etc. Then we go over the file to update the names of agent and designers in *introduce* and *nickname*. We spend some time learning the rules of Baroque chess, which is essential in coding functions to find all possible moves for a given state: *nextMove*, *findAllMove*, *PincerMove*, *PincerCapture*, *CoordinatorMove*, etc. From there, we design the static evaluation functions (*basicStaticEval* and *staticEval*) to guide the search. We implement Zobrist hashing in an additional module called *DD_BC_module_zobrist_hashing.py*. After that, we integrate in the function *minimax* some techniques, such as minimax search, alpha-beta pruning, Zobrist hashing, and time limit. Adapting the code of minimax search from Assignment 5 to this project is a little trickier in, for example, class variables. In addition, we design the *parameterized_minimax* that returns the desired data. Furthermore, as our most important function, *makeMove* uses almost all the AI techniques designed in this project, such as IDDFS, Zobrist hashing, custom static evaluation, and minimax search with alpha-beta pruning. As expected, debugging costs some additional time. The final step is drafting the report.

iv) Zobrist hashing details. Where in your move-making did you include put and get calls?

For the module of Zobrist hashing, please see 7.

In function *minimax*, we use the following syntaxes to implement Zobrist hashing.

```
if g_useZobristHashing:
    hashValue = zh.hash_state(currentState)
    if hashValue in zh.zobrist_table:
        val = zh.zobrist_table[hashValue]
        n_zobrist_hasing += 1
    else:
        if g_useBasicStaticEval:
            val = basicStaticEval(currentState)
        else:
            val = staticEval(currentState)
        zh.zobrist_table[hashValue] = val
        n_static_evals += 1
else:
    if g_useBasicStaticEval:
        val = basicStaticEval(currentState)
    else:
        val = staticEval(currentState)
    n_static_evals += 1
```

v) 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?

Because of the using of the hashing, when a player makes a move, we don't need to recalculate the hash value from scratch. Instead we simply use few XOR operations to recalculate the hash value. Within the same time limit, our agent could search more deeper plies.

vi) Optional partnership retrospective

Please see 11.