CS441 WRITEUP

Yiming Lin

yl6@pdx.edu

1. Instruction of Running Program

Open three terminals and do the following things:

- In the first terminal, enter into directory server_and_negamaxplayer/:

```
cd server_and_negamaxplayer/
```

- Compile all the Java code and run the server:

```
javac *.java
```

```
java Gthd 0
```

- In the second terminal, enter directory server_and_negamaxplayer/:

```
cd server_and_negamaxplayer/
```

- Run negamax player by specifying the side black or white:

```
java Grossthello <side> local host 0 <depth>
```

- In the third terminal:

```
python3 game.py
```

Optional arguments:

`-s <side>:`	Specify the side of the player, default is **black**, choices are black and white
`--iterdeepening`	enable iterative deepening minimax search
`-m <number>`	specify the maximum number of states can be visited during iterative deepening minimax search, the default value is 3000
`--stats`	enable printing statistical information during minimax search
`--moveselection`	enable move selection during minimax search when there are multiple same returned values from recursive calls, the selection is based on the move that has largest number of liberties. If not enabled, the selection is randomly picked among all moves that have same returned value.
`-e <method>`	specify static evaluation function, there are two options: “eye” and “number”, the default is “number”, which counts the number of stones on each side. “eye” not only counts number, but also counts the number of eyes on each side
`--stonescore <number>`	specify the score for one stone
`--blackeyescore <number>`	specify the score for one black eye
`--whiteeyescore <number>`	specify the score for one white eye
`--depth <number>`	specify the depth limitation for minimax search, if `-iterdeepening` is not enabled. The default value is 4

Table 1 The table of optional arguments for game.py

2. Result of Experimentations and Evaluation

The following table shows 6 experimentations in this project. Each game results are stored in file "chessresult.txt" in the top-level directory.

2.1 Minimax player uses **black** stones vs. Negamax player uses **white** stones

	Negamax Player		***Minimax Player***		***Win rate of minimax***
#	Depth-limited	Evaluation method	Enable Iterative deepening	Select moves by liberties number	
1	3	Number	True	True	0.9
2	3	Number	True	False	0.7
3	3	Eye	True	False	0.8

Table 2 The result of experimentations

2.2 Minimax player uses **white** stones vs. Negamax player uses **black** stones

	Negamax Player		***Minimax Player***		***Win rate of minimax***
#	Depth-limited	Evaluation method	Enable Iterative deepening	Select moves by liberties number	
4	3	Number	True	True	0.8
5	3	Number	True	False	0.4
6	3	Eye	True	False	0.2

Table 3 The result of experimentations

Note: in each of all six experimentations, 10 games are run and the maximum number of states to visit during iterative deepening minimax search is set to 1000.

2.3 Commands for running experimentations

The following shell commends show how to run each of the experimentation:

Open one terminal:

```
java Gthd 0
```

The other two terminals, use the commands in the following table:

	Commands in each of the two terminals
#1	`python3 game.py -s black --iterdeepening -m 1000 --stats --moveselection` `java Grossthello white localhost 0 3`
#4	`python3 game.py -s white --iterdeepening -m 1000 --stats --moveselection` `java Grossthello black localhost 0 3`
#2	`java Grossthello white localhost 0 3` `python3 game.py -s black --iterdeepening -m 1000 --stats`
#5	`java Grossthello black localhost 0 3`

```
python3 game.py -s white --iterdeepening -m 1000 --stats
```

#3
```
java Grossthello white localhost 0 3
python3 game.py -s black --iterdeepening -m 1000 --stats -e eye
```

#6
```
java Grossthello black localhost 0 3
python3 game.py -s white --iterdeepening -m 1000 --stats -e eye
```

3. Minimax Algorithm

In this project, depth-limited minimax algorithm is implemented to find a move for the agent.

Function Decision(Board, Max-depth):

    Depth ← Max-depth

    Return Max-value(Board, Depth)

Function Max-value(Board, Depth):

    If terminal-test(Board):

        Return Board.Evaluate()

    Moves ← Board.Generate-possible-moves()

    If no Moves:

        Return Board.Evaluate()

    Values ← []

    For each Move in Moves:

        B ← Board.Does-move(Move)

        V ← Min-value(B, Depth – 1)

        Values.append(V)

    Max-index ← Index-of(Max-of(Values))

    Return Max-of(Values) and Moves[Max-index]

Function Min-value(Board, Depth):

    If terminal-test(Board):

        Return Board.Evaluate()

    Moves ← Board.Generate-possible-moves()

    If no Moves:

        Return Board.Evaluate()

    Values ← []

    For each Move in Moves:

        B ← Board.Does-move(Move)

        V ← Max-value(B, Depth – 1)

        Values.append(V)

    Min-index ← Index-of(Min-of(Values))

    Return Min-of(Values) and Moves[Min-index]

The function terminal-test() above returns “True” if the board has resulted in a winner or draw; it also tests for depth, if it is less than 0, it will return “True” to terminate recursion. Otherwise, it will return “False”.

4. Alpha-beta Pruning

This project also implemented alpha-beta pruning. Alpha-beta pruning is used to reduce visiting unnecessary states during the agent making a decision by visiting all possible moves. When the agent is deciding whether to continue visiting another move in recursion, the agent will do the following things:

In max node: after the agent visit the first “child node”(the state that is generated by trying the first move of all possible moves), the agent checks beta first, if the returned value of previous recursive call is greater than beta, it will return that value directly, rather than trying another possible move.

In min node: the agent checks alpha first after the agent visit the first “child node”, it the returned value of previous recursive call is less than alpha, it returns directly, so that the other recursive calls (incurred by the other possible moves) are pruned.

Updating of alpha and beta: if it is not pruned after recursive call (trying possible move) returned, in max node, the alpha is updated to the greater one between current alpha value and returned value; in min node, the beta is updated to the smaller one between current beta value and returned value.

Thus, the minimax algorithm is improved as following:

```
Function Decision(Board, Max-depth):
        Depth ← Max-depth
        Return Max-value(Board, Depth, -inf, inf)

Function Max-value(Board, Depth, Alpha, Beta):
        … Terminal tests and generate moves …
        Moves ← a list of possible moves based on Board
        Values ← []
        For each Move in Moves:
                B ← Board.Does-move(Move)
                V ← Min-value(B, Depth – 1)
                If V >= Beta:
                        Return V and Move
                Values.append(V)
                Alpha = Max-of(Alpha, V)
        Max-index ← Index-of(Max-of(Values))
        Return Max-of(Values) and Moves[Max-index]

Function Min-value(Board, Depth. alpha, beta):
```

```
… Terminal tests and generate moves …
Moves ← a list of possible moves based on Board
Values ← []
For each Move in Moves:
        B ← Board.Does-move(Move)
        V ← Max-value(B, Depth – 1)
        If V <= Alpha:
                Return V and Move
        Values.append(V)
        Beta = Min-of(Beta, V)
Min-index ← Index-of(Min-of(Values))
Return Min-of(Values) and Moves[Min-index]
```

---

5. Transposition Table

In this project, a transposition table is used to store visited states, since there are some possibilities that there are different paths lead to the same state. Using a transposition table to store that state as well as its score, it will help searching to be faster, because the agent will not recursively visit that state, instead the agent uses the score in the transposition table directly.

The special case in this project is that with the depth going deeper, the number of stones cannot become smaller, thus it will not be the case where a state in a deeper depth is identical to a state in a shallower depth. Hence, in this transposition table, the depth will not be maintained. And in this project, each time the agent is going to make a decision, the transposition table will be cleared.

The data structure for storing states is a hash table. And the hash key for mapping states is Zobrist key.

5.1 Zobrist Hashing

5.1.1 Zobrist table initialization

To use Zobrist hashing, a Zobrist table is initialized at the very beginning as following:

We have 5 × 5 = 25 position to put stones, also we have 2 color for the stones, so we have 50 different positions. Then, we can have a 2-dimensional array to represent each position with some random value (I use uuid.uuid4().int in Python 3.7.2 uuid module to large generate random value). Usually 64-bit bitstring is a common choice.

	pos1	pos2	…	pos25
black	n1	n2	…	n25
white	n26	n27	…	n50

Figure 1 Zobrist table representation

5.1.2 Generate Hash Key

Since Zobrist table has been initialized, then hash key can be generated. The hash key(h) is initialized to be 0. Each time we find a stone on the board, we can obtain its coordinate on the board, and its color. The we can find related number in the Zobrist table by using coordinate and color: in figure 1, if coordinate is (a, b), and color is white, the related number is Zobrist-table[5 × a + b][1]. Then using XOR operator to "XOR into" h.

Thus, the hash key can be generated like this:

```
Function Zobrist-key(Board, Zobrist-table):
        Key ← 0
        Loop x for 0..4:
                Loop y for 0..4:
                        Key ← Key ^ Zobrist-table[x * 5 + y][color-of(Board[x][y])]
        Return Key
```

5.1.2 Update Hash Key

To update hash key, it is not necessary to traverse all stones on the board, which is the advantage of Zobrist hashing. To remove a stone on the board, the only thing needs to do is to "XOR out" that value in the Zobrist table. Thus, once there is a stone captured from black to white, operate as following:

```
Key ← Key ^ Zobrist-table[x * 5 + y][color-of(black)]        -- (1)
Key ← Key ^ Zobrist-table[x * 5 + y][color-of(white)]        -- (2)
```

(1) is what is called "XOR out", which output a value that recover to the key that the board doesn't have a black stone on (x, y); (2) is what is called "XOR in", which output a value that is a key of the board that have a white stone on (x, y).

5.2 Using Transposition Table

The use of transposition table in this project is straight forward. In both min node and max node, I checked the transposition table first, if the Zobrist key for this board is in transposition table, return value directly. Otherwise, the transposition table will be updated during recursive calls.

6. Iterative Deepening Minimax Search

The last feature I implemented in this project is iterative deepening minimax search. Start with 1 ply deep, then record the path to the best state. Next, use the recorded path to re-order the moves using search. Doing repeatedly until the maximum number of states visited is reached.

7. Discussion

7.1 Iterative Deepening vs. Fixed Depth-limited Searching

The first thing I learned in this project is that iterative deepening has some advantages compared to

fixed depth-limited search. When the stone number on the board is small (usually during serial 1, 2, 3, and 4), it will search lots of states in order to find a good move, thus it is very hard to search into 6 depth, 7 depth, even 8 depth without using advanced computation techniques (I didn't use parallelism, or concurrency in this project, should be a future work), since there are lots of possible moves. However, when the number of stones on the board becomes larger, there are less possible moves to generate, thus, it is relatively easy to search 6 depth, 7 depth, even to final state.

In iterative deepening minimax search, it is hard to reach 5000, even 10000 states (a setting of maximum number of state to visit for making a decision) when there are 16 stones on the board with the help of alpha-beta pruning, thus the agent have the capability to search more deeper when the game is near ending. However, in fixed depth-limit minimax search, when the depth is set to 3 or 4 (for a reasonable short time to search), its searching depth is still 3 or 4 when the game is near ending.

Furthermore, in textbook the author mentions the concept "killer move", which is used to inform the move ordering during next iteration of iterative deepening search, which is helpful to prune more nodes during alpha-beta pruning, since killer move usually have a greater or smaller evaluated value, and it can quickly update the alpha-beta window.

7.2 Randomization vs. Ordered by Number of Liberties

It is very common that at the beginning of the game, the returned values of recursive calls are same, which leads to a situation that needs to randomly pick up a move. This is what our course git repo's negamax player does. I found that at the beginning of the game, especially when server's serial number is 1, 2, 3, even to 5 and 6, it is hard to find a move that is significantly better than others based on the evaluation function that counts number of stones. Thus, in some sense, it is more like randomly pick up a position and make a move. I have no chess previous experience, but in this project, I try to pick the move that has the largest number of liberties among all the moves that have the same best score, I found that it is working pretty good, as the experiment results in 2.1 and 2,2 experiment #1 and experiment #4.

8. Future Work

In the future, I would try to build a reinforcement learning system (Q-learning to be specific) by using a multi-level perceptron and CNN. I "guess" CNN can have a surprisingly effect, since a board is a 2-d plane, rather than a 1-d vector.