Artificial Intelligence - Methods and Application

Assignment 1 - Othello

5DV181
Cyril Bos (CAS: cybo0001, CS: ens17cbs)

Teachers: Ola Ringdahl
Juan Carlos Nieves Sánchez

Contents

1 General information and test results

The assignment has been developed in Java 8. It uses the given helper code. The source code is in the directory `umu-othello/src`. To compile it, execute `javac Othello.java` from this directory.

I copied and modified the script othellostart into a new script `test_code/x-times-othellostart.sh`, which adds a number of games played parameter. This new script counts how many times each player wins and computes the average of score points on their wins (loosing does not change the score average).

The table below shows how many points my AI wins with when playing against the naive one. It seems there is a bit of performance loss when playing as white when the time limit increases. The AI might in fact plan too far ahead and "incorrectly" since the other AI does not reason the exact same way.

Time limit	1	2	3	4	5	6	7	8	9	10
Score as white	15	32	32	20	52	61	20	20	20	20
Score as black	50	34	60	62	61	22	22	40	54	54

Figure 1: Results from test runs

2 getMoves() and makeMove() algorithms

In the given pseudo-code algorithms, "player" refers to the player currently played by my AI, and "opponent" to the other player. So if `OthelloPosition.playerToMove` is true, player refers to the white player and opponent to the black player. The algorithm for `getMoves()` in pseudo-code is:

```
for every opponent disc:
    if there is an adjacent empty cell E:
        for every disc C in the opposite direction:
            if C is a player disc
                then add the possible move(E.row, E.column)
            else if C is an opponent disc or an empty cell
                then change to the next empty cell

return the list of possible moves
```

The algorithm of course checks and stops if going over the limit of the board, or encounters an empty cell in the opposite direction. It also checks that a move has not already been added, done by overriding `equals()` in class `OthelloAction`.

The adjacent empty cells are calculated by a double iteration on row and column value differences between -1 and 1 (ignoring row = 0 and column = 0).The opposite direction is then the opposite of these values. For instance, going north-west will be [-1, -1] and south-east will be [1,1].

The `makeMove()` method first checks if the move is a pass move (it does nothing if it is). It then throws an `IllegalMoveException` if the given `OthelloAction` row and column are outside the board, or if the corresponding cell is already occupied. Otherwise, it executes this algorithm given in pseudo-code:

```
for every direction around played cell:
    if the first cell in that direction contains an opponent disc:
        loop through the cells in that direction while they
           contain an opponent disc
        if the last cell contains the player disc
            loop again from the first cell, converting the
               opponent discs in between
```

The directions are represented with the same system as the `getMoves()` method. If no discs are captured, this is an illegal move and an exception is raised.

3 Board evaluation through heuristics

The class `BoardEvaluator` implements the interface `OthelloEvaluator` using several heuristics. The `evaluate()` method computes the difference of score between the white player (always max) and the black player (always mini). This score is an evaluation of the given board state, where each disc value is 1 except on specific locations:

- the corners, which are very valuable since they can not be taken back and help control the borders and the diagonals of the board ($score = 20$),
- the stable tokens, which can not be captured back by the opponent ($score = 5$),
- the C cells, which are located on the borders of the board, adjacent to the corners. They usually help the opponent capture the corners ($score = -10$),
- the X cells, which are located on the diagonals of the board, adjacent to the corners. They usually are even worse than the C cells because they can still be captured even if the corner is occupied by the player ($score = -15$).

The corners, the C cells, and the X cells are detected through almost hard-coded positions. A token is considered stable if for every adjacent empty cell, the board is full of player discs in the opposite direction of that empty cell. The evaluator checks for every disc first if it is a corner, else if it stable, else if it is on a C, else if it is on a X cell. This way, **stable** discs placed on X or C cells are not valued down.

The different score absolute values are not that important but should respect that order ($abs(corner) > abs(X) > abs(C) > abs(stable)$). This way, it prevents from playing towards a position with captured bad X and C cells but enough captured other cells to counterbalance this score loss. With these current absolute score values, a bad X cell is equivalent to 3 stable tokens. One could imagine a situation where the AI stops its search while thinking ”if I capture this bad X cell I will capture 4 stable tokens”, opening a corner for the opposite AI.

The board score is adjusted with the number of moves available to each player, which is especially useful in early game to avoid running out of moves.

4 Alpha Beta implementation

The class `AlphaBeta` (which implements the interface `OthelloEvaluator`) translates the AlphaBeta algorithms given during the lecture.

Both `maxValue()` and `minValue()` methods first test if the maximum depth has been reached, and if so evaluate the position score. Secondly, they test is if it is a leaf position, in which case it checks if the game is ended, returning a maximum value if the white player wins, and a minimum value if the black player wins. This is done by switching the `OthelloPosition.playerToMove` value (done by calling `makeMove()` with a pass move), then by counting the number of white and black discs on the board. This returns a pass action with a score of `Integer.`**`MAX`**`_VALUE` $(+\infty)$ if white wins, or `Integer.`**`MIN`**`_VALUE`$(+\infty)$ if black wins. Otherwise, the score is evaluated through the usual heuristics.

5 Iterative deepening search

To manage the iterative deepening search with the time limit, the `main` method in `Othello.java` parses the time limit, converts it to milliseconds and adds it to the current time. 100 milliseconds are arbitrarily subtracted to ensure the program does not go over the time limit. The search is first of depth 1 and uses iterative deepening. The program increments the depth of the search and stores the resulting move until interrupted by handling an `OutOfTimeException`:

```java
while (System.currentTimeMillis() < timeLimitStamp) {
    moveChooser.setSearchDepth(depth++);
    try {
        OthelloAction newMove = moveChooser.evaluate(position);
        //may return a blank action if there is no move possible
        if (!newMove.equals(new OthelloAction(0, 0))) {
            chosenMove = newMove;
        }

    } catch (OutOfTimeException exception) {
        //time is up
    }
}
```

This exception is thrown inside `maxValue()` and `minValue()` methods of class `AlphaBeta`, in the iteration over the possible moves at each depth.

6 Encountered problems

At first, I had troubles debugging my `getMoves()` and `makeMove()` methods. For instance, I forgot at first to ignore the direction [0,0] in my loop. It would have been wise to code unit tests from the start.

I encountered problems while implementing the Alpha Beta algorithm as it is easy to make small mistakes hard to debug since the code for max and min are almost the same.

Also, I did not implement the iterative deepening search correctly at first, as I was replacing the chosen move of depth-1 by the best move found during the current unfinished search. I needed to modify my code to be able to know if the search ended because of the time limit, and the easiest way was to use the exception system described above.

7 Possible evolutions

The usage of the heuristics could evolve depending on the advancement of the game. I did not because the number of turns is hard to determine with the functioning of the script (it would be needed to store and read it in a file for example). It could instead count the number of discs on the board. In order to do this efficiently, `OthelloPosition` should store two attributes (number of discs of each player) and update them in `makeMove()`. A simpler to implement but less efficient algorithm would be to iterate through the whole board. In end-game, it would be wiser to rely more on the pure number of tokens instead of their position (corner, C, X, ...). It might even be possible to brute-force every position instead of pruning.

The heuristics could also be enriched. The board evaluator could attribute a score to every cell in a matrix instead of considering only specific cells (corners, C and X cells). Against a real player or better AI, it might be better to balance out the board evaluation with the other heuristics (number of possible moves for example).

Some performance improvements include multithreading/multiprocessing, and that would benefit early move ordering. In the current implementation, the Alpha Beta algorithm uses depth-first search, thus possibly cutting the best move if it gets pruned at a lower depth. By searching each move at first depth (in parallel), it might be better to launch the first search down the branch with the highest evaluated score, giving different alpha and beta values when looking at the other branches later on.