UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG
School of Computer Science & Applied Mathematics

Advanced Analysis of Algorithms
2021 Assignment: Part 5

1 Introduction

The previous part resulted in a minimax agent with a very simple evaluation function. In this section, we will implement an an alpha-beta pruning agent with improved evaluation function.

2 Evaluation Function

Because we will not be able to search to the end of the game (the search tree is too large), we must use an evaluation function. This will allows us to search N moves into the future and evaluate the resulting position. An evaluation function is simply a function that accepts as input a board position, and returns a value that measures how good the position is for whoever it is to play. Importantly, the evaluation function is static — it does not consider moves into the future. The first submission will have us create a more complex evaluation function.

3 Search Algorithm

Once we have our evaluation, implementing the search function is fairly standard. In this submission, we will alpha-beta pruning.

4 Input Hint

Hint: a reminder not to be careful when mixing `cin` with `getline`. An example of doing so is below:

```
int N;
cin >> N;
cin.ignore(); //NB!
for (int i = 0; i < N; ++i) {
    string fen;
    getline(cin, fen);
}
```

Submission 1: Advanced Evaluation

Write a C++ program that accepts a FEN string, sets up the board, and then outputs an evaluation of the position from the perspective of the player to move next. Our evaluation function here will be more complex than before, and is described by the following formula:

$$rawScore = \Delta material + \Delta mobility + \Delta attack \tag{1}$$

In the above, Δ refers to the difference between the scores for white and black (e.g. $\Delta material$ is equivalent to the $rawScore$ we calculated in Part 4). Each term in the above is calculated as follows

Material Score

The material score for each side is calculated using the instructions and table in Part 4. Compute the delta by taking white's score and subtracting black's.

Mobility Score

Given a position, the mobility score is the number of moves each side is able to play. Imagine we are given a position with white to play. The mobility score for white is simply the number of moves it has available. To calculate the score for black, "pretend" it is black to play instead, and calculate how many moves they would have available. Compute the delta by taking white's score and subtracting black's.

Attack Score

Given a position, the attack score is the number of opposite pieces that a player is threatening to capture. Imagine we are given a position with white to play. The attack score for white is simply the number of moves whose target square contains a black piece. In addition, for each move whose target square contains the black lion, add 10 to the score. The pseudocode describing this is below

```
attackScore = 0
for each valid move:
    if move attacks enemy piece:
        attackScore = attackScore + 1
        if attacked piece is enemy lion:
            attackScore = attackScore + 10
```

To calculate the score for black, "pretend" it is black to play instead, and then follow the exact same rules to compute their attack score. Compute the delta by taking white's score and subtracting black's.

Overall score

To compute the raw score, we again follow rules that we've implemented before:

1. If the board contains only a black and white lion and no other pieces, then the raw score is 0.
2. Otherwise, if the black lion is missing, then white has won and the raw score is 10000.
3. Otherwise, if the white lion is missing, then black has won and the raw score is -10000.
4. Otherwise, add up the deltas of the above three scores.

The raw score above is the score from white's point of view. But perhaps it is black to play! Therefore, to compute the final score, we must consider whose turn it is. If it is white to play, multiply the raw score by 1. Otherwise, multiply it by -1. This is the final value that should be returned by our evaluation function.

Input

The first line of input is N, the number of input positions given as FEN strings. N lines follow, with each line containing a single FEN string.

Output

For each FEN string, output the evaluation of that position according to the above evaluation function.

Example Input-Output

Sample Input

```
3
2ele1z/ppppppp/7/7/7/PPP1PPP/2ELE1Z w 4
1z5/pPp1lP1/5ep/4P1e/4L1p/2p2pP/7 b 12
1z5/pPp1lP1/5ep/4P1e/4L1p/2p2pP/7 w 12
```

Sample Output

```
-102
920
-920
```

Submission 2: Search

Write a C++ program that uses the evaluation function from the previous submission and implements alpha-beta pruning. The program should accept a FEN string, set up the board, and then output the value returned when searching to a depth of **4**. Pseudocode for the search algorithm is provided below:

```
function alphaBeta(currentState, depth, alpha, beta):
    if isGameOver(currentState) or depth <= 0:
        return evaluate(currentState)
    moves = generateMoves(currentState)
    for each move in moves:
        nextState = makeMove(currentState, move)
        eval = -alphaBeta(nextState, depth - 1, -beta, -alpha)
        if eval >= beta:
            return beta
        if eval > alpha:
            alpha = eval
    return alpha
```

Note: in the above, the game is over when one of the lions has been captured.

Input

The first line of input is N, the number of input positions given as FEN strings. N lines follow, with each line containing a single FEN string.

Output

For each FEN string, output the alpha-beta evaluation of that position for $depth = 4$ using the aforementioned evalation function.

Example Input-Output

Sample Input

```
3
2ele1z/ppppppp/7/7/7/PPP1PPP/2ELE1Z w 4
1z5/pPp1lP1/5ep/4P1e/4L1p/2p2pP/7 b 35
1z5/pPp1lP1/5ep/4P1e/4L1p/2p2pP/7 w 12
```

Sample Output

```
-103
832
-914
```