COMP24011 Lab1: Reversi

Ian Pratt-Hartmann and Francisco Lobo

Academic session: 2024-25

Introduction

In this exercise, you will write a Java program to play the game of reversi, sometimes known by its trademark name of Othello. Reversi is played on an 8 by 8 board, initially set up as shown in Figure 1 on the left, with the players moving alternately by placing their own pieces (black or white) on the board, one at a time. By default Black always goes first.

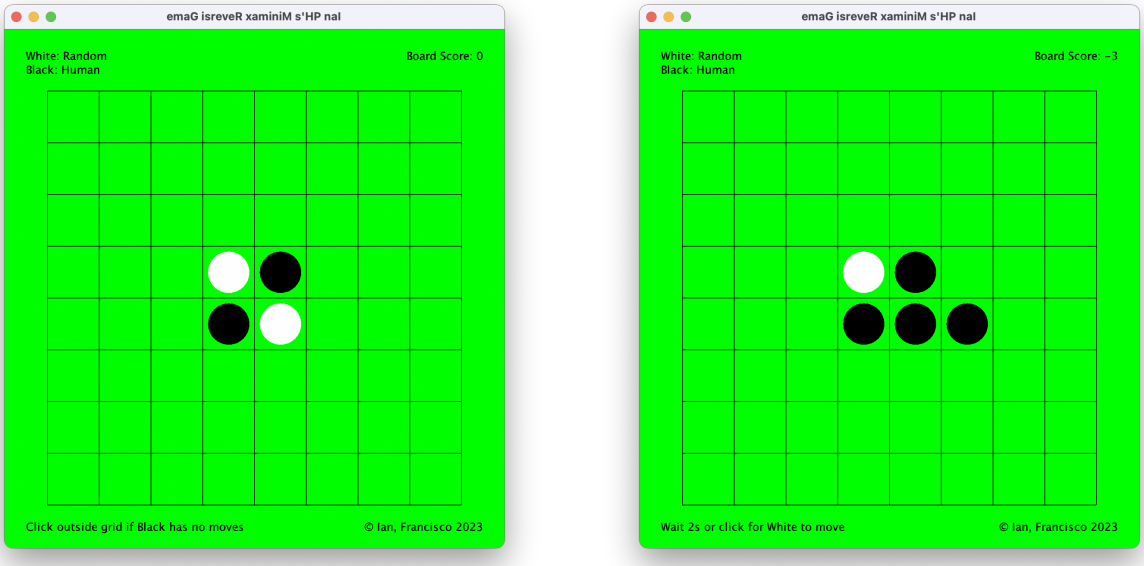

Figure 1: The opening position in a reversi game; Black moves to $(5, 4)$.

In the `lab1` branch of your `COMP24011_2024` GitLab repo you will find the Java code for the game of reversi in Figure 1. To compile and run this program you can use the following commands

```
$ javac *.java
$ java Othello
```

In addition to the game board, the main window also shows

- on the top left, the player agents that control the white and black pieces
- on the top right, the current board score
  i.e. the difference between the number of white and black pieces
- on the bottom left, a message indicating who's turn it is and how to proceed with the game
- on the bottom right, a copyright notice; please do not distribute the code for this lab exercise.

Note that the board score value will be positive when White is winning and negative when Black is winning. By default the Java program allows a human player (always Black) to play against a computer agent that moves randomly (always White). You can change the code in `Othello.java` to configure these game settings. In fact, you will **need to** do this to test your player agent!

The graphical interface of the Java reversi game is not very responsive and a bit basic, to put it mildly. For example:

- The human agent waits for the user to click on a square and then makes that move. There is a 2 seconds delay before non-human agents can move, but the user can also click to request the agent to move.
- Clicking on squares which do not represent legal moves just produces an irritating beep.
- If it is the turn of the human player and there is no legal move, then the user must click anywhere in the green area outside the grid of the board, to generate a skip move and allow play to pass to the other player.
- The interaction between mouse click events from the user and timer delay events that trigger non-human agents is not perfect... It's better not to click while the computer is 'thinking' ☺
- If you want to play another game, you need to close the window and re-run the program.

But these are details (which you are welcome to improve, though no extra marks are available). By default, as the game progresses there is some debug output printed on the console. If you close the window after making the move shown in Figure 1 on the right, the terminal will display

```
$ java Othello
Loading game...
Human: Move (5,4), valid, 0.2 secs, no error
Current boardState: {........|........|........|...wb...|...bbb..|........|....
....|........}
Current boardScore: -3
```

This reports that the user attempted to move to the square $(5, 4)$, that this move was valid and hence made. The program times how long the agent took to select a move in response to the mouse or timer event, in increments of 0.2 seconds, and if this generated any runtime error. Note that x and y move coordinates can only have values from 0 to 7, and the top left square is at $(0, 0)$.

When you close the window, an ASCII representation of the current state of the game is written. It shows the squares of the board row-by-row, with the letters 'w' and 'b' representing white and black pieces respectively, and dots representing empty squares. This representation is enclosed by curly braces or square brackets depending on whose turn it is to move, White or Black respectively. The Java program accepts these ASCII representations as an optional parameter on the command-line. If you execute

```
$ java Othello "{........|........|........|...wb...|...bbb..|........|........
|........}"
```

you will restart the game exactly where you've left off. You'll probably find this very useful while testing and debugging the Java code for your agent.

Let's review how reversi is played. In the following description, a *line segment* is a sequence of board squares forming a contiguous straight line (horizontal, vertical or diagonal). The rule for a player to place a piece is:

- 👉 The piece must be placed on an empty square such that there is a line segment passing through the piece played, then through one or more pieces of the opposing colour, and ending on a piece of the player's own colour.

When such a line segment exists, we say that the opponent's pieces on that line segment are *bracketed*. When a piece is played, bracketed pieces change colour according to the following rule:

- 👉 For every a line segment passing through the piece played, then through one or more pieces of the opposing colour, and ending on a piece of the player's own colour, the pieces of opposing colour through which that line segment passes are all changed to pieces of the player's own colour.

In Figure 2, the left-hand picture shows a game state where $(3,5)$ is one of the possible moves for White. This move brackets three of the opponent's pieces, namely $(3,6)$ horizontally, $(3,4)$ vertically, and $(4,4)$ diagonally, resulting in the position shown in the right-hand picture.

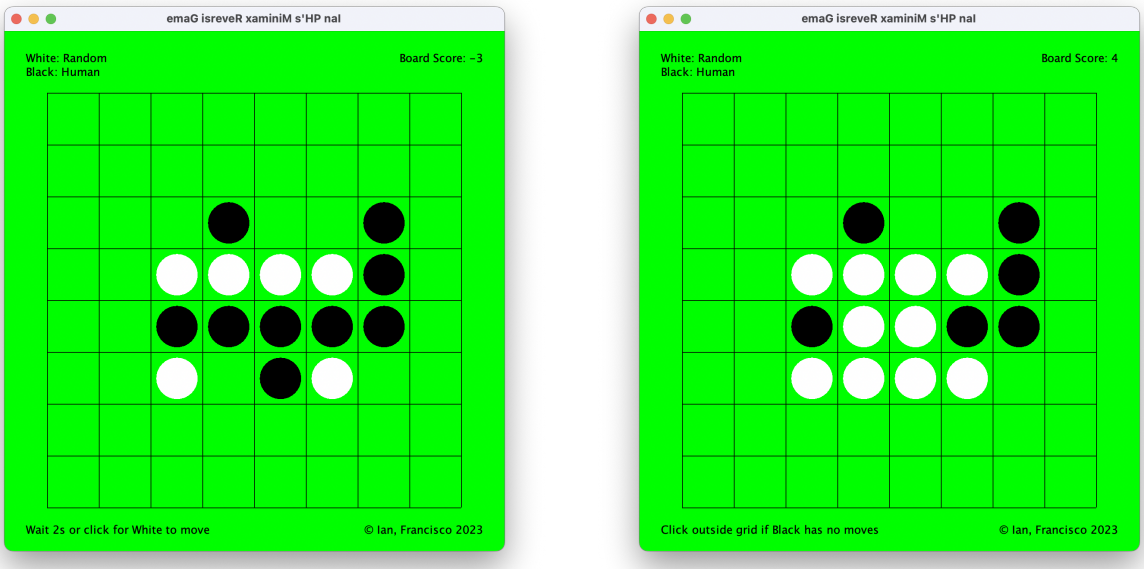

Figure 2: A move by White in a reversi game, and its result.

If, and only if, a player cannot move, but his opponent can, he misses a turn. The game ends when neither player can move. (This usually, but not always, happens because all the squares have been occupied.) The winner is the player with the greater number of pieces of his own colour on the board; if there is no such player, the result is a draw.

Assignment

Try out the Java reversi game, and play a few games. You will see that the computer is playing terribly. In fact, it computes the possible moves in the current position, and then simply selects one at random! Your task is to implement a player agent that uses the minimax search with alpha-beta pruning to play a better game.

Before starting this exercise, make sure you understand generic types in Java, e.g. things like `ArrayList<SomeClass>`. Otherwise, the Java code is very simple. To complete this exercise you do not need to understand the code in `OthelloDisplay.java` or `MoveStats.java` which runs the game UI and the collection of debug stats.

The main game logic is in the class `BoardState`. An instance of this class represents a situation in the game. The field `colour` encodes whose turn it is (1 for White; -1 for Black), while the low-level methods `int getContents(int x,int y)` and `void setContents(int x,int y,int piece)` allow retrieval and setting on the individual board squares. Here, a value of 1, -1 or 0 represents presence of, respectively, a white piece, a black piece or no piece at all at the square on rank x and file y.

A separate class `Move` is used to encode the actual moves; it has two public fields, `x` and `y`; as well as a public method `boolean isSkip()`. As long as `isSkip()` is false, `x` and `y` give the coordinates of a newly placed piece in the obvious way. If, on the other hand, `isSkip()` is true, then the move represented is the act of 'skipping one's go' (passing control to the opponent); and the values of the coordinate fields have no meaning. Note that an instance of `Move` is just a move — not necessarily a legal one. In particular, the skip move is legal if and only if the current player cannot put a piece down, but his opponent can.

Both of these classes include the code necessary to be used as Java `HashMap` keys. To make things easier for you, the `BoardState` class has the method **`boolean`** `checkLegalMove(Move m)`, which checks whether it is possible for the current player to make the given move, as well as the method **`void`** `makeLegalMove(Move m)`, which actually executes the move. In fact, to make things *really* easy, we have provided the method `ArrayList<Move> getLegalMoves()` that returns the list of all and only those legal moves for the current player. Notice that this list cannot be empty as long as the game is not over.

All game player agents correspond to implementations of the abstract Java class `MoveChooser`. The rest of the control is handled by the program. You are given 3 working implementations of this class:

- `MoveChooserHuman` that simply relays the moves chosen by the user on the UI.
- `MoveChooserFirst` is a computer player that always selects the first available legal move.
- `MoveChooserRandom` is a computer player that selects one of available legal moves at random.

To complete this exercise you need to implement a computer player called `MoveChooserAlphaBeta`. Note that the sample code that you're provided compiles but doesn't work. You cannot change the public interface of the sample code; the parameters for each method are documented in the source.

The constructor `MoveChooserAlphaBeta(`**`int`** `searchDepth)` needs to set a name for your player agent (to be used on the UI), and record the search depth that will control the minimax search algorithm. The method `Move chooseMove(BoardState boardState,Move hint)` is the most important that you need to implement. This is called when it's the computer's turn to move in the board state `boardState`. As you're coding a non-human player, the 2nd parameter giving the move `hint` of the triggering UI event should be ignored. All you have to do is write a move selection procedure that uses the minimax algorithm with $\alpha\beta$-pruning discussed in the lectures. Note that you are allowed to code auxiliary functions to achieve this. The depth of the minimax search should be controlled by the `searchDepth` field of your agent's instance.

We want you to proceed in 2 stages. In the first stage you implement a board evaluation function, using the following method. Each square in the playing area is assigned a number as follows.

```
120 -20 20  5  5 20 -20 120
-20 -40 -5 -5 -5 -5 -40 -20
 20  -5 15  3  3 15  -5  20
  5  -5  3  3  3  3  -5   5
  5  -5  3  3  3  3  -5   5
 20  -5 15  3  3 15  -5  20
-20 -40 -5 -5 -5 -5 -40 -20
120 -20 20  5  5 20 -20 120
```

These numbers reflect the value for a player of being on the respective square. Note that almost all squares on the edges have high value (since pieces here are hard to take) and squares in the corners have the highest value (since pieces here cannot be taken). By contrast, neighbouring squares have negative values, since a piece there will allow one's opponent to move onto a high-value square. (Incidentally, the above array of values was taken from an older textbook by Peter Norvig.) The value of a board position can then be defined by adding up the weights of all those squares occupied by white pieces and subtracting the weights of those squares occupied by black pieces. Thus, the value for the board is always counted from White's point of view. This value needs to be computed by the method **`int`** `boardEval(BoardState boardState)` of your `MoveChooserAlphaBeta` implementation.

In the second stage you implement `MoveChooserAlphaBeta.chooseMove()` so that it performs minimax search with $\alpha\beta$-pruning. If you find $\alpha\beta$-pruning hard, you may prefer first to try ordinary minimax, and then graduate to $\alpha\beta$-pruning once that's working. Your first challenge is to calculate minimax values for board positions. Note that your `MoveChooserAlphaBeta.boardEval()` can be

used when you reach the limit of the search depth. This is controlled by the field `searchDepth` of the `Othello` class, which by default is set to 6. You may wish to set it to a lower value to speed up development, and increase it when you get everything working.

It is worth thinking about what you'll need to do when searching through a tree of board positions. Remember that `BoardState` instances are Java objects. By calling `BoardState.makeLegalMove()` on these, you will update the board (and the current player). Therefore, if you want to get the daughters of a vertex `b0` in the search tree, you will need to create a fresh copy of this board state for each legal move, and then execute the move in question on that copy. We have provided a cloning method `BoardState.deepCopy()`, so that a fresh copy of the board state `b0` can be made with the call `b1 = b0.deepCopy();` subsequently modifying `b1` (e.g. by executing a legal move) does not then change `b0`.

The cost of cloning full board states will affect performance, and in some (but not all) cases this can be avoided. Once you know that your implementation is working, you can investigate improving performance by using the methods `BoardState.save()` and `BoardState.restore()` to replace some of the cloning calls in your algorithm. Note that if your agent runs into the 10 seconds timeout set in `MoveStats` for the collection of debug stats, then you probably have written inefficient code, but it is unlikely that this is due to uses of the `BoardState.deepCopy()` method.

When you start coding, change `Othello.java` so that `MoveChooserFirst` plays Black and your `MoveChooserAlphaBeta` plays White with `searchDepth` set to 4. Use the ASCII representations to help you debug any illegal moves or runtime errors that your code may generate. Once your player agent is beating `MoveChooserFirst`, it should be playing a decent game of reversi against a human. Try this out!

After that, increase `searchDepth` to 6 and have `MoveChooserRandom` play Black. If your implementation is correct, your agent should consistently beat a player choosing moves more or less at random, and should not take more than a few seconds for any move when run on an ordinary PC at this search depth. Now try setting the search depth to 8. This should be enough to thrash most human players, and won't be too slow.

Submission

Please follow the `README.md` instructions in your `COMP24011_2024` GitLab repo. Refresh the files of your `lab1` branch and develop your solution to the lab exercise. The solution consists of a single file called `MoveChooserAlphaBeta.java` which must be committed to your GitLab repo and tagged as `lab1_sol`. The `README.md` instructions that accompany the lab files include the `git` commands necessary to commit, tag, and then push **both** the commit and the tag to your `COMP24011_2024` GitLab repo. Further instructions on coursework submission using GitLab can be found in Appendix L of the CS Handbook, including how to change a `git` tag after pushing it.

The deadline for submission is **09:00 on Monday 21st October**. In addition, no work will be considered for assessment and/or feedback if submitted more that **1 week** after the deadline. (Of course, these rules will be subject to any mitigating procedures that you have in place.)

The lab exercise will be **auto-marked** offline. The automarker program will download your submission from GitLab and test it against our solution agent `MoveChooserReference`. In each test your agent plays the reversi game against another agent starting on a random board position for a total of 20 moves (or until the game is over). The following statistics are collected.

- The change in board score. If the board score changes in favour of the player your agent controls, then your agent **beats** its opponent in that game.
- The number of times your agent **completes** `MoveChooserAlphaBeta.chooseMove()` within the 10 seconds time limit of `MoveStats`. If your agent times out and fails to complete a move, then `MoveChooserFirst.chooseMove()` will be used to decide your player's turn on that occasion.

- The number of times your agent performs minimax search with $\alpha\beta$-pruning **correctly**. This happens only when it chooses the same move as `MoveChooserReference.chooseMove()` for the particular board state **and** chosen search depth.
- The number of times your agent calculates the board evaluation value **correctly**. This happens only when it returns the same value as `MoveChooserReference.boardEval()` for the particular board state.
- The time your agent takes to complete **correct** moves, in increments of 0.2 seconds, will be compared to the time `MoveChooserReference` takes to complete the same moves. The number of times your agent is at least **as fast as** the reference agent is counted.

A total of 20 marks is available in this exercise. The marking scheme is as follows.

4 marks Your agent will be initialised as `MoveChooserAlphaBeta(4)` and play 4 tests as White against `MoveChooserRandom()`. You get 1 mark for each game your agent beats its opponent (or else makes only correct moves).

4 marks Your agent will be initialised as `MoveChooserAlphaBeta(8)` and play 4 tests as White against `MoveChooserReference(5)`. You get 1 mark for each game your agent beats its opponent (or else makes only correct moves).

4 marks Your agent will be initialised as `MoveChooserAlphaBeta(8)` and play 2 tests as White and 2 tests as Black against `MoveChooserReference(7)`. You get 1 mark for each game your agent beats its opponent (or else makes only correct moves).

2 marks If your agent correctly calculates board evaluation values while playing the above 12 test games. To obtain 1 mark your implementation needs to be correct 50% of the time.

4 marks If your agent correctly performs minimax search with $\alpha\beta$-pruning while playing the above 12 test games. To obtain 3 marks your implementation needs to be correct 80% of the time. To obtain 2 marks your implementation needs to be correct 65% of the time. To obtain 1 mark your implementation needs to be correct 50% of the time.

2 marks If your agent is as fast as the reference agent while playing the above 12 test games. To obtain 1 mark your implementation needs to be fast 50% of the time.

Important Clarifications

- If your code produces runtime errors under ordinary game conditions, then it will fail to complete moves. The automarker will try to finish test games by deferring the broken call to the `MoveChooserFirst` implementation, as described above. Such turns will contribute negatively when calculating the percentage of time your implementation was correct.
- It will be very difficult for you to circumvent time limits during test games, as the automarker will use its own version of `MoveStats.java`. If you try to do this, the most likely outcome is that the automarker will fail to receive return values from your implementation, which will have the same effect as your agent not completing the call. In any case, an additional time limit of 150 seconds per test game will be enforced; this is sufficient to allow your agent 10 seconds for each move.
- This lab exercise is fully auto-marked. If you submit code which does not compile on the department's lab machines, you will score 0 marks. We recommend that you make a backup of the Java source files provided for this lab, and ensure your `MoveChooserAlphaBeta.java` compiles with those original files before submission.
- It doesn't matter how you organise your `lab1` branch, but you should avoid having multiple files with the same name. The automarker will sort your directories alphabetically (more specifically, in ASCII ascending order) and find submission files using breadth-first search. It will mark the first `MoveChooserAlphaBeta.java` file it finds and ignore all others.
- Every file in your submission should only contain printable ASCII characters. If you include other Unicode characters, for example by copying and then pasting code from the PDF of the lab manuals, then the automarker is likely to reject your files.