CS24011 Lab 1: Reversi

Ian Pratt-Hartmann
Francisco Lobo

Academic session: 2022-23

Git Tag: `24011-lab1-S-Games`

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, with the players moving alternately by placing their own pieces (black or white) on the board, one at a time. Black always goes first.

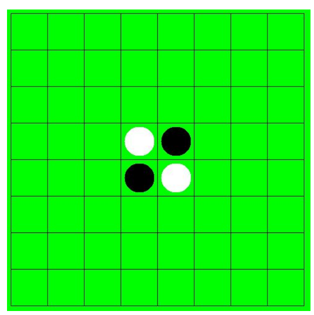

Figure 1: The opening position in a reversi game; black to move.

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 possible move for white, which brackets three of his opponent's pieces, resulting in the position shown in the right-hand picture.

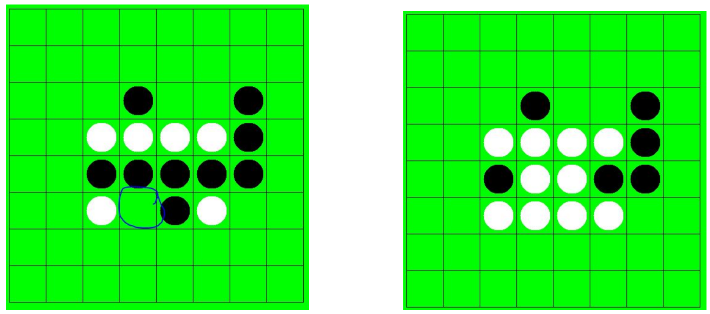

Figure 2: A move by white in a reversi game, and its result.

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

The materials for this lab are available in the `lab1` branch of your Gitlab repo

`COMP24011_2022/COMP24011_`*username*

which should exist in your Gitlab account. Follow the instructions on the coursework section of the CS Handbook to clone the repo. The top `README.md` file explains how the repo is organised, how access the `lab1` branch, and how to use the refresh script to fetch the lab materials.

To complete this lab, unzip the file `labCode.zip`. The unpacked directory contains a Java program to play a game of reversi. To compile the code, use `javac Othello.java`. The subsequent call `java Othello` will invoke a simple program that allows a human player (always black) to play against the computer (always white). The provided graphical interface is rather basic, to put it mildly: clicking on squares which do not represent legal moves just produces an irritating beep; if it is your go, but you have no legal move, then you must click anywhere on the board to allow play to pass to the computer; if you want to play another game, you need to close the window and re-run the program. Also, it's better not to click while the computer is 'thinking'. Finally, as a special irritation, if the game ends with the computer's move, then you have to click on the board to see the result. Try it out, and play a few games. You will see that the computer plays terribly. In fact, it computes the possible moves in the current position, but just selects the first one! Your task is to modify the program so that it uses minimax search with alpha-beta pruning to play a better game.

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 invididual 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. To make things easier for you, we have included the method `boolean checkLegalMove(int x, int y)`, which checks whether it is possible for the current player to move on square `(x,y)`, and `void makeLegalMove(int x, int y)`, which actually executes the move. In fact, to make things *really* easy, we have provided a method to return the list of all and only those legal moves for the current player. Here, we rely on a class `Move` to encode the actual moves; it has just two public fields, `x` and `y`; so that instances of this class represent moves (legal or illegal) in the obvious way. The method for retrieving the list of legal moves for the current player is ArrayList<Move> getLegalMoves(). (Make sure you understand generic types in Java!). *Notice that this list may be empty, because the current player may be unable to make a move.*

The computer player is located in the class `MoveChooser.java`, of which the main program creates an instance. The only thing that this class does is implement the static method `Move chooseMove(BoardState boardState)`. This is the method called when it's the computer's turn to move in the board state `boardState`. In its current version, this method just gets the legal moves and returns either `null` if that list is empty (remember what I said about there sometimes being no legal moves), and the first move in that list otherwise. The rest of the control is handled by the program. All you have to do is write a better move selection function than just picking the first. In fact, you must use minimax with $\alpha\beta$-pruning. The depth of the search should simply be controlled by the static final int `searchDepth`, which is currently set to 6 at the start of the `Othello` class. You may wish to set it to a lower value to speed up development. When you get everything working, try setting the search depth 8. This should be enough to thrash most human players, and won't be too slow.

You will require a static evaluation function and there several choices available. For this assignment you *need* to desing yours as follows. Each square in the playing area is assigned a number:

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 the squares on the edges have high value (since pieces here are hard to take) and squares in the corners have an even higher value (since pieces here cannot be taken). By contrast, neighbouring squares have negative values, since a piece here will allow one's opponent to move onto a high- value square. 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 is always counted from white's point of view. (Incidentally, the above array of values was taken from an older textbook by Peter Norvig).

You will also have to write a program to calculate minimax values of board positions. 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. It is worth thinking a bit about what you will need to do when searching through a tree of board positions. Remember that instances of `BoardState` are Java objects. When you call `BoardState.makeLegalMove(x, y)`, that will update the board (and the current player). If you want to get the daughters of a vertex `boardState` in the search tree, therefore, you will need to create, for each legal move, a fresh copy of `boardState`, and then execute the move in question on that copy. The method `Boardstate.deepCopy()`, which we have provided in `BoardState` is a cloning method, so that a call `boardstate1 = boardstate.deepCopy()` sets `boardState1` to be a fresh copy of `boardstate`; subsequently modifying `boardstate1` (e.g. by executing a legal move) does not then change `boardstate`. When computing the daughters of a vertex in the game tree, be careful about two corner cases: one is where the computer has no legal move. In that case, control passes to the other player. In terms of the search tree, this means that the vertex representing the board state in question has a single daughter, with all the pieces the same, but the current player changed.

The top-level call is a little different to subsequent calls, because you have to select the move that yields the best daughter (from the point of view of the computer player), rather than simply evaluating the daughter. It is important that, *when and only when there are no moves available*, you return the move `null`. This is fine: our code will understand what that means.

Try the program `Othello.java` out with your version of `MoveChooser.java`, and with the constant `Othello.searchDepth` set to the value 6. The resulting program should play a decent game of Othello against a human, and should certainly beat a player choosing moves more or less at random. Moreover, the program should not take more than about a few seconds for any move when run on an ordinary PC at that search depth; if so, you have probably written inefficient code.

Submission

The `README.md` instructions explain how to upload your files to the `lab1` branch of your Gitlab repo. One of these files **must be** your re-witten version of `MoveChooser.java`; you may include other `.java` files (for example, containing subsidiary classes). However, you may not include any files with other extensions than `.java`; nor may you modify the files provided in the exercise (with the exception of `MoveChooser.java`, of course). Any such files or changes to existing files **will be ignored**. Important: **do not zip** the source files of your solution. Remember: you will **need to** tag your commit with the command

```
git tag 24011-lab1-S-Games
```

You will most likely have done this for other courses as well. General instructions on coursework submission using Gitlab can be found on the CS Handbook.

The deadline for submission is **18:00 on Monday, 17th October**. The lab will be **auto-marked** offline. The marking program will download your code from Gitlab and test it against a reference implementation on random board configurations. To avoid any impacts of such random boards, the auto-marking will be done **3 times** and your final grade will be the **median score**. (This will take time so expect marking to take at least 2 weeks.)

The agent of the reference implementation has three settings: easy, medium, and hard. Each test starts on a random board and plays your implementation against the reference agent for a fixed number of moves. Before and after each test is run, the marking program calls `BoardState.result()` and records this board score. The marking program will use these board scores to measure how much your implementation improves the position of the player it is controlling. It will also keep track of the moves your implementation makes to check if it correctly follows the minimax algorithm with the prescribed evaluation function. Finally, the marking program measures the speed at which your implementation chooses its moves.

Mark Scheme

A total of 20 marks is available. The marking scheme is as follows:

1 mark Your code completes all test runs without runtime errors.

(18 marks) For each of the three reference agent settings:

- **3 marks** Depending on proportion of tests in which your implementation improves the board score for the agent it is controlling. (You need 75% for 3 marks, 50% for 2 marks, and 25% for 1 mark.)
- **3 marks** Depending on proportion of moves in which your implementation correctly follows the minimax algorithm and the prescribed evaluation function. (You need 100% for 3 marks, 75% for 2 marks, and 50% for 1 mark.)

1 mark The speed of your implementation is as fast as the reference agent in 50% of tests.