Coursework Report

Marco Moroni
40213873@live.napier.ac.uk
Edinburgh Napier University - Algorithms & Data Structures (SET09117)

Abstract

The goal of this coursework is to implement the classic board game of checkers in an arbitrary computer language demonstrating a correct use of data structures. The language chosen here is Python and the game can be played from the console.

Keywords – algorithms, data structures, Python, checkers, draughts

1 Introduction

This report aims to describe the implementation of a checkers game. It was important to use the correct data structures in the code, trying to make it as efficient as possible. The game was coded in using Python.

2 Design

2.1 Overwiew

The board is made by a 2D list, in which every item can be either empty or a piece.
Every piece know its *rank* (man or king) and what player they belong to.
Moves are recorded in a stack called `moves`, which is used when undoing. There is another similar stack, `redoMoves`, used to store moves that can be redone.
The game uses a while loop as a game loop. This loop runs until there is a winner.
Finally, the AI works by choosing random moves.

2.2 Board

This project started by creating a simple board made by a 2D array. In Python this is archived by using a list of lists, which means a list of rows where each row is a list os sqaures.
There are 8 rows and 8 columns and each sqaure of the board is initially empty (`None` in Python).

2.3 Pieces

In checkers a piece can be:

- either black or white;
- either a man or a king.

Considering that, the game uses a `Piece` class that knows:

- its player (of type `Player`);
- its *rank*, that is wheter it is a man or a king.

The ranks are enumerated elements of a class `PieceRank`: the values are `PieceRank.MAN` and `PieceRank.KING`. Initially, a piece also knew its position it had at the beginning of the game. This information could have been used in an early version of the `replay` funtion, where there was a board reset to its inital state. Later on this information became useless, because `replay` now resets the board by undoing every move.

2.4 Players

A player is normally identified by being either black or white. In this implementation, a player is an instance of a class `Player`. This class has:

- a dictionary of symbols (*symbols* are the characters printed in the console to represent a piece). The keys are `PieceRank`s and the values are a character (eg. a white/black dot for men, or a white/black sqaure for kings);
- a boolean `isFacingUp`. This is used in two occasions: when setting up the pieces (see section 2.12.1 *Prelude*) and when getting all the legal displacements of a piece (see section 2.11 *Getting legal moves*);
- a boolean `cpu`;
- all the functions used by AI (called when the player is not human) (see *AI*).

2.5 Printing the board

The function `printBoard` simply print a symbol for every sqaure where there is a `Piece`. It will print one of the two symbols (man or king) stored in the player class of the piece

2.6 Moves

Every move is an instance of the class `Move`. This class stores the `originPosition` of the moved piece, the `displacement` ($(\pm1, \pm1)$ or $(\pm2, \pm2)$), the eventual `pieceEaten` and a boolean `doesBecomeKing`.
Initially `pieceEaten` was not present, but knowing only if the displacement is $(\pm2, \pm2)$ is not enogh when undoing a move. It would be easy to just create another piece of the opponent player when recreating an eaten piece,

but there would be no way to know what rank it used to be.

2.7 Storing moves

Moves are stored in a Python list `moves` considered as a stack: when performed, a move gets pushed in and when undone, it gets popped out. Another stack has been added later, `redoMoves`, which is used to store moves that can be redone.
The program works with these 2 stacks by following 3 rules:

1. every time a move is performed, tis move gets pushed to `moves` and `redoMoves` is emptied;
2. when undoing the last move `m`, `m` get popped from `moves` and pushed to `redoMoves`;
3. when redoing a move `m`, `m` get popped from `redoMoves` and pushed to `moves`.

2.8 Undo

When the function `undo` is called the steps in algorithm 1. There was a problem with this function: if one of the

move = moves.pop() redoMoves.push(move) undo piece position **if** *move.displacement ==* $(\pm 2, \pm 2)$ **then**
| restore piece eaten in $(\pm 1, \pm 1)$
end
if *move.doesBecomeKing* **then**
| piece.undoBecomingKing
end

Algorithm 1: `undo`

players was not human the following scenario would happen:

1. it's turn 5: human player choose to undo;
2. it's now cpu player's turn: it moves a piece;
3. it's turn 5 again and human player did not actually undo her move;

To fix this situation `undo` can take one argument that corresponds to the number of moves a player wants to undo. The only change to implement this was to put algorithm 1 inisde a `for` loop.

2.9 Redo

`redo` is very similar to `undo`, as you can see in algorithm 2. And like `undo`, the whole algorithm is inside a for loop

move = redoMoves.pop() moves.push(move) redo piece position **if** *move.displacement ==* $(\pm 2, \pm 2)$ **then**
| eat piece in $(\pm 1, \pm 1)$
end
if *move.doesBecomeKing* **then**
| piece.becomesKing
end

Algorithm 2: `redo`

that iterates as many times as the player want to.

2.10 Replay

`replay` is quite simple. It first undo everything and then it redo everything (but in slightly different ways). As shown in alogrithm 3, while `undo` is called only once (it undo the whole game), `redo` gets called once for each move and every time the board is printed. In this way the user can see all the moves made in chronological order, just like the game is being replayed.

totalNumberOfMoves = len(moves)
undo(totalNumberOfMoves) printBoard() **for** $i = 0$ **to** *totalNumberOfMoves* **do**
| redo() printBoard() wait some time
end

Algorithm 3: `replay`

2.11 Getting legal moves

This is the function that implements most of the rules of the board game.
It is called `getLegalDisplacements` because it returns a list of all the legal displacements ($(\pm 1, \pm 1)$ or $(\pm 2, \pm 2)$) a piece can move by. It has 2 arguments:

- the coordinates of the piece to be considered;
- a boolean `mustEat`.

This is how the function works: it creates a list `possibleDisplacements` of the 8 possible displacements (all the sign combinations of $(\pm 1, \pm 1)$ and $(\pm 2, \pm 2)$), it deletes all the illegal displacements, and then it returns it (as the list of all legal displacements called `legalDisplacements`).
That was the big picture of `getLegalDisplacements`, but the implementation has been optimised and tweaked little bit as you can see in algorithm 4.
From there it can be noted that `possibleDisplacements` does not have all 8 displacements, but only the ones possible depending on rank and `mustEat`. Also, every row is multiplied by -1 if the player is facing the board so that the closest row to him is the 8th.
After that, only the legal displacements between those are kept and returned. Because element shouldn't be removed from a list while it is iterated, another list, `lagaleDisplacements`, had to be used, be the concept remains the same.

2.12 Game loop

The game is player in a while loop that runs until there is a winner. The steps inside this loop are the following:

1. get a list of all movable pieces;
2. select an action (move, undo, redo, replay, none (if no movement are available));
3. perform the action;
4. calculate next player.

2.12.1 Prelude

Before the game loop begins, the game has to be set up:

```
possibleDisplacements = []
// use a multiplier to change the rows depending on
   which side the playeris facing
mult = 1
if player.isFacingUp then
    mult = -1
end
// add displacements to possibleDisplacements
if not mustEat then
    possibleDisplacements.append((1 * mult, -1))
    possibleDisplacements.append((1 * mult, 1))
end
possibleDisplacements.append((2 * mult, -2))
possibleDisplacements.append((2 * mult, 2))
if piece.player is king then
    if not mustEat then
        possibleDisplacements.append((-1 * mult, -1))
        possibleDisplacements.append((-1 * mult, 1))
    end
    possibleDisplacements.append((-2 * mult, -2))
    possibleDisplacements.append((-2 * mult, 2))
end
// remove illegal displacements
legalDisplacements = []
for each displacement in possibleDisplacements do
    if destination is inside the board and is not
       occupied then
        if displacement is (±2, ±2) then
            if you eat an opponent piece then
                add displacement to
                   legalDisplacements
            end
        end
        else
            add displacement to legalDisplacements
        end
    end
end
return legalDisplacements
```

Algorithm 4: `replay`

1. the empty board is created;
2. the players are created;
3. the pieces are created and placed on the board (using the same algorithm for both sides, but tweaked with a row multiplier, like in algorithm 4);
4. the empty stacks `moves` and `redoMoves` are created;
5. one of the players is set as current player.

2.12.2 Getting all movable pieces

At the beginning of the game loop a list of `(pieceCoordinate, [displacements])` is created by using the function `getLegalDisplacements` for each current player's pieces. When creating this list the program also remembers two things (as booleans): if at least one piece can eat (`mustEat`) and if there are no possible moves (`canMove`).
In checkes, if a player have the possibility to eat an opponent piece she must do so: if `mustEat` is true the movable pieces are recalculated, but tis time the function `getLegalDisplacements` will be forced to return displacements of $(\pm 2, \pm 2)$.
The list of all movable pieces is used not only for an easier check of the player's input, but also to help the player deciding what to do, by printing a board that highlights all the pieces that can be moved.

2.12.3 Selecting an action (move, undo, redo, etc.)

The player is now able to select an action: move a selected piece (if `canMove == True`), undo (if `len(moves) > 0`), redo (if `len(redoMoves) > 0`), replay or nothing (if `canMove == False`).
An action is stored as an enumarated value of the class `ActionType`.
If the player is not human there are only two possible choices: move or nothing.

2.12.4 Perform the action

Every action is then performed depending on the player's choice:

- move: player must select a legal destination and the move is stored (also, `redoMoves` is emptyed). If the player can eat again after the moving he must do so;
- undo , redo, replay: simply call the corresponding function;
- nothing.

Initally, when the only action coded was *move*, the next player would have been very easy to caculalte. Algorithm 5 shows how it was done. This had to be changed by re-

```
while there is not a winner do
    for each player do
        select move
        perform move
    end
end
```

Algorithm 5: Old game loop

moving the for each loop and calculating the next player every time an action is performed (see section 2.12.5 *Next player*) .

2.12.5 Next player

Because this program supports undo and redo, calculating who the next player is is not straightforward.
This is how it is calculated depending on the action performed (note that this is executed *after* the action):

- move: the other player;
- undo: the player of the last move in `redoMoves`;
- redo: the player of the last move in `moves`;
- replay: the same player, because when the replay is over it will still be the turn of the player who called the function;
- nothing: the other player.

2.13 AI

The `Player` class has two functions that return the coordinates of a random valid piece that can move and a random valid move. This functions are called when the cpu player is choosing its move.

3 Enhancements

The board may be implemented by using a graph insted of a 2D list. Because the pieces in checkers can only be in black squares, it would make sense to consider each black square a node which edges connect to the (up to) 4 diagonal black squares. In other words, a sqaure will have a reference to its `topLeftSquare`, `topRightSquare`, `bottomRightSquare` and `bottomLeftSquare`.
For example, the funcion that checks if there is a winner can be updated. In its current state, it loop thorugh each square, even the white ones. If the board was a graph as described before, it could recursively check each node, halving the sqaures to check.
Another example can be with anything involved with piece movements. Insted of using displacementes or destination coordinates, it can be more efficient to utilise the edeges directly (e.g. `topLeftSquare` instead of `(-1, -1)`).

4 Critical evaluation

Using displacements instead of coordinates was an ealy development idea though to simplfy the program. Considering the final code, it may have been simplier to use coordinates.
Undoing and redoing is managed efficiently, using simple but very effient stacks that are filled with `Move`s.
It should also be not to difficult to use this program as a template to build a different board game because not many things have been hardcoded.

5 Personal evaluation

...

6 Conclusion

It have been very useful to plan the data structures used ahead, because they can be difficult to amend once in the code, expecially if the language used is not Python, which uses lists only.