# Draughts in Python

Emma Parsley
40206111@live.napier.ac.uk
Edinburgh Napier University - Algorithms and Data Structures SET09117

## Abstract

This project aims to create playable Draughts in Python using the most useful and relevant algorithms and data structures. It should be possible to play this with two human players as well as with one human player against the computer. It should be possible to undo, redo and replay the games.

**Keywords –** Python, Algorithms, Data Structures, AI, Minimax

## 1 Introduction

This rules for draughts used in this project are those listed on the English Draughts Wikipedia[1]. These rules must be implement as such that a user can play a game without breaking these rules and will win or lose the game when win conditions have been met. An artificial intelligence should also be able to play this game against a human obeying all of the rules. At any point in the game the user should be able to undo back to the start of the game. If the user has undone anything provided they haven't changed anything since they should be able to redo back to where they were before they started undoing. Previous games should also be able to be replayed.

## 2 Design

### 2.1 Grid

**Squares** To create the board a class was created called grid which would take a width and height and create the equivalent of a 2D array. It is standard in Draughts to have an 8x8 board of pieces, however as python does not have 2D arrays (which would need to have their size defined at the start) the board squares where created as a list of lists meaning the board width and height could be set at any size.

Listing 1: Creating 2D array equivelant in Python

```
1
2 class Grid:
3     def __init__(self, width, height):
4         self.squares = []
5         for i in range(height):
6             self.squares.append([])
```

Restrictions do need to be put on the width and height sizes to keep the game fair and readable, for example having an odd width will cause one player to have more pieces than the other.

**Rows** In a standard game of Draughts each players pieces take up the first 3 rows of the board however as we can define the height of the board there are situations where this doesn't work, such as when the boards height is 6 or less, to fix this the rows must be set to be at least 1 and at most;

$$rows = (height/2) - 1$$

**Create Grid** Any square in the grid can contain a white space, a black space, player 1's piece or player 2's piece. A white space is a space that can never have a piece in it, a black space however can have a piece on it.

```
for i = 0 to height do
    for j = 0 to width do
        if (i is even and j is odd) or (i is odd and j is even)
         then
            square[i][j] contains white space
        end
        else if i < rows then
            square[i][j] contains player 2's piece
        else if i >= height - rows then
            square[i][j] contains player 1's piece
        else
            square[i][j] contains black space
        end
    end
end
```

**Algorithm 1:** FizzBuzz

Squares are stored in the list at squares[i][j] with i being the height of the grid at it's position and j being the width of the grid at it's position.

**Viewing Grid** A grid stored like this is really simple to view in console, just having a nested loop though the height and then width will print out what the squares contain.

To make the grid more user friendly the width numbers are turned into letters using ASCII values (which puts another restriction on the grid width as if it exceeds 26 non letter ASCII characters will start printing) and these are printed below and above the grid. 1 is added to the height numbers and they are printed to the left and right of the grid with a tab for separation (This puts restrictions on the height of the grid as if the height number exceeds the tab it will skew the grid).

For this project player 1's pieces are shown with the letter "w" player 2's pieces are shown with the letter "b" and both black and white spaces are shown with a space. See Figure 1.

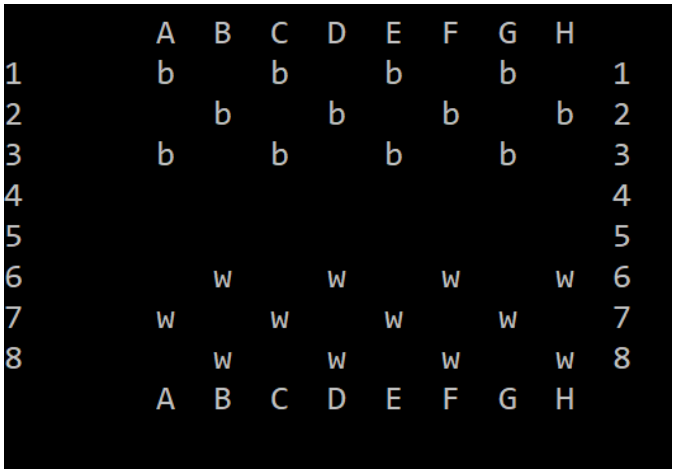


Figure 1: **Board** - This is how the grid ended up looking by default

## 2.2 Pieces

**Piece Class** Each piece in this project needs to know where it currently is, weather or not it has been taken, and weather or not it is a king.In order to help with the undoing and redoing it also knows every time it has moved.

**Data Structures** A piece's current position is stored in a tuple (i, j), with i being the height of the square and j being the width, so this can be easily transferred into grid square positions.

Whether or not a piece is taken is stored as an int, this is to help with undoing. When a piece is taken this is set to the turn it was taken on so when undoing the piece knows it needs to re-appear on the board on this turn. As it is impossible to take a piece on turn 0 this can be set to 0 when the piece has not been taken.

Whether or not a piece is a king is used in much the same way as a piece being taken, it is impossible for there to be a king on turn 0 so it is that by default and changes to the current turn when it becomes a king.

Every time a piece moves it stores the (i, j) tuple of coordinates that it moved to in a dictionary with the key being the turn it made this move. Using this data structure means pieces only have to remember a turn if they moved on it and as keys are unique they can't somehow have more than one move per turn.

**Viewing Pieces** Pieces are viewed by overriding the to string method in the class to return the symbol for either the default piece or the king of that piece if it's a king. See Figure 2.

## 2.3 Movement

There are 3 ways a piece can move in draughts;

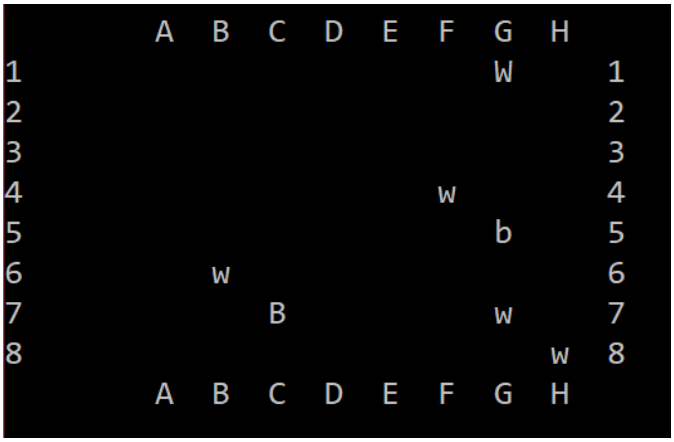


Figure 2: **Different Pieces**- Kings are capital letters of the regular pieces

"forwards" - *Towards the other players starting side moving diagonally.*

"forwards and backwards" - *If a piece makes it to the edge of the opposite side of the board to where it starts it becomes a king, kings are allowed to move one space in all diagonal direction.*

"jumping" - *If there is an opponents piece one space away diagonally from the player's piece in a way that it can move it may jump over this piece if the next diagonal piece is empty. The piece may continue to do this from the space it has jumped to until there are no more valid spaces it can jump to. When a piece is jumped over it is "taken" and is removed from the board.*

## 2.4 User Movement

To take movement from a human user, the program must first check if any pieces can jump. If a piece can jump another piece it must so the program most first check if the user must do this and if they must force them to. Then input must be taken from the user of the coordinates of piece they would like to move and it should be converted into a form that can be used by the program. If the user is allowed to can move this piece the program should then take in the coordinated of the square the piece should be moved to and convert this into the form the program is using.

**Normal Movement** If a user is simply moving one of it's pieces forward moving this piece is simple. First in order to check if the coordinates given to move to are valid, when a piece is given the program should look for all the valid places it can move (it will check for empty squares in front of it diagonally and if it's a king it will also check behind it). All the valid spaces it can move to are added to a set and then the program simply needs to check if the given coordinates to move to are in that set. A set should be used rather than a list because all valid spaces to move to should be unique and sets don't have any duplicates and also because set's are quicker to search than doing a linear search through a list.

To complete the move the program simply swaps the

piece from it's current square to the given square, sets the pieces current coordinates to the new coordinates, adds this move to the pieces list of moves and checks if the piece has reached the other side. If the piece has reached the other side and it is not already king it sets it's turn kinged variable to the current turn.

**Jumps** If a user can jump they must jump, so when checking if the current players pieces can jump any other pieces all the pieces that can jump added to a set (again because they will all be unique and sets are quick to search through). If this set contains items a jump has been forced so this time to find the valid places for the given piece the program looks for all the places this piece can jump to and adds any spaces it can jump to in multiple ways to a set.

To complete this move is more complicated than a normal move as the route the user wants to take isn't always clear. If the space the user has given to move to is only 2 spaces away and there is an opponents piece in-between it and there it simply needs to jump over that piece, add the same information to the piece as with a normal move and also tell the piece in-between that it's been taken that turn and remove it from the board. The program then needs to check if any more jumps can happen and if they can it takes user input again and repeats this until no more can happen.

If the user inputs a space further than 2 spaces away or the opponents piece is not in-between then the program works out what pieces it would have to jump to get there unless any pieces are in the set made earlier of spaces with multiple ways to get to in which case the program can't work it out and asks the user to input one jump at a time.

# 3 Memory

A memory class is used to store all pieces moved on a turn and the turn you can redo up to. When undo is called the memory class checks the take turns, movements, and turns kinged of all pieces in that turn of it's list and resets them to before that happened. Redo works doing the opposite.

# 4 AI

The AI is implemented using a minimax algorithm which can look to any depth but is by default at a depth of 5. If this algorithms max/min returns multiple of the same score it will choose a random one of these moves.

## 4.1 Take Tree

The AI uses an n-ary tree to store the possible routes it can take to take the opponents pieces. The take tree is stored in the same file as the Memory class as both are very short classes so having a file each is a waste of space.[2])

# References


[1] Wikipedia, "English draughts." https://en.m.wikipedia.org/wiki/English_draughts#Rules.

[2] TicTacTo, "Minimax." http://neverstopbuilding.com/minimax.