Iteration 2 - *Date 2/12/22*

Aims for this iteration

The main aim for this iteration is to create a working chess engine. I may also attempt to start building the frontend interface but this goal is secondary. I then should be able to create a basic chess game that the user can play against the bot.

My chess engine should:

- be functional for basic chess but will not include special moves such as on passant, promotion and castling.
- at least have some user interface which is at minimum console based and allows the user to play a game of chess.
- have a minimax function that is efficient enough to run at depth 2 in a reasonable time.
- It should be rigorously tested to ensure that it is working and that there aren't any nasty surprises later down the line.

Functionality that the prototype will have

So to meet my main goal and successfully build a chess engine my end product must:

- Be able to correctly determine the legal moves available to a player, accounting for the different movements of pieces and check
- The engine should be able to identify check
- The engine should be able to give a static evaluation of each board state
- The engine should allow for a move to be executed: creating a new child board state that can be examined
- The engine should be able to identify then the game is over, who has won and the outcome type (stalemate, checkmate)

My minimax function should:

- Be able to beat a randomly moving opponent
- Have some consideration of efficiency and efficacy

My unit tests should:

- Be fully automated allowing them to be easily re-run to diagnose problems
- Be a mix of data driven and logic driven tests where appropriate
- My unit tests for minimax should be able to finish and so some efficiency is needed as it will need to undertake many games at depth 2 or 3

Annotated code screenshots with description

Quick note: I have annotated my code by adding comments explaining it in depth. Further comments are made with reference to specific function or classes in word. I have not added comments explain my VUE GUI as it is now redundant and is no longer being developed (still useful as parts can be reused). I will also explain both the code and the tests in tandem as they were created in tandem (I test as I went along not at the end).

I first began by attempting to make my interface. This was not used in the final product but will be used in future iterations. I used VUE js to create the interface shown:

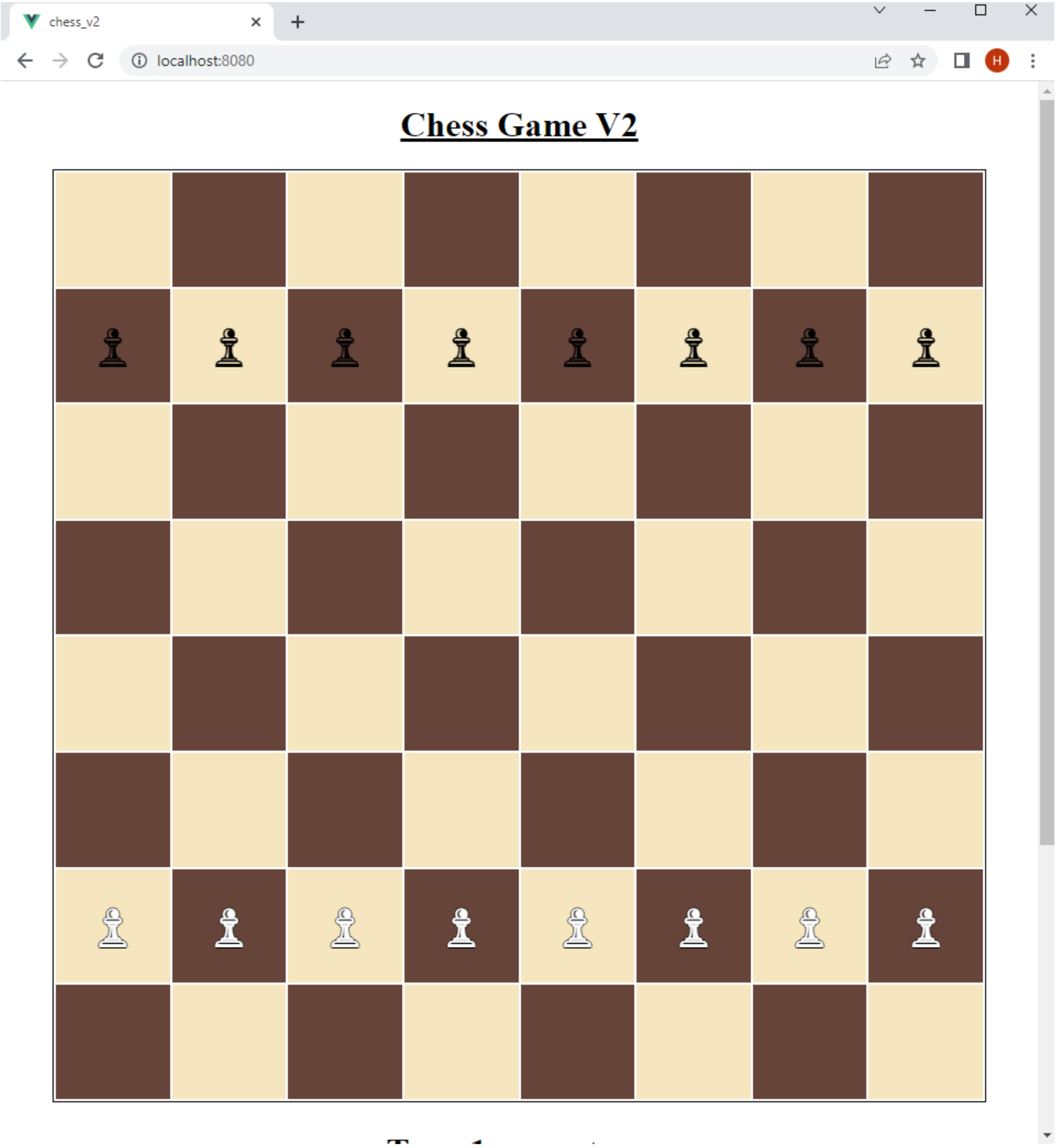
(screenshot of how I ran the code)

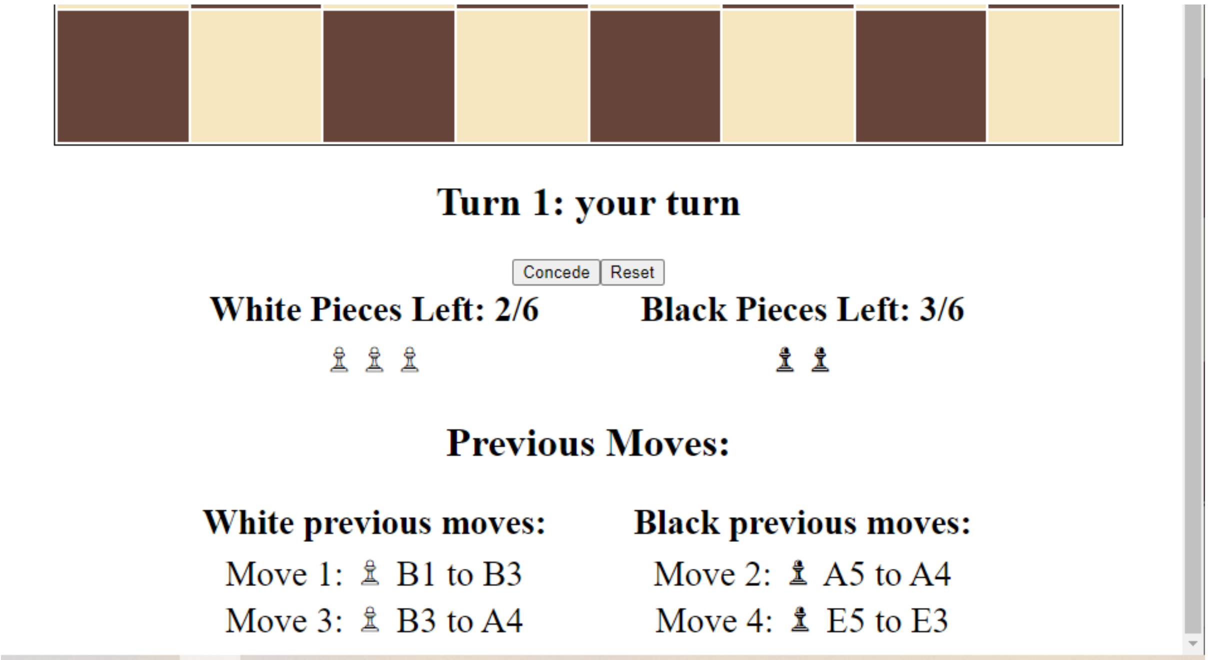
.

```
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.1 vue page> cd chess_v2
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.1 vue page\chess_v2> npm run serve
npm WARN config global `--global`, `--local` are deprecated. Use `--location=global` instead.

> chess_v2@0.1.0 serve
> vue-cli-service serve

```

```
 DONE  Compiled successfully in 7850ms


  App running at:
  - Local:   http://localhost:8080/
  - Network: http://192.168.1.50:8080/
```

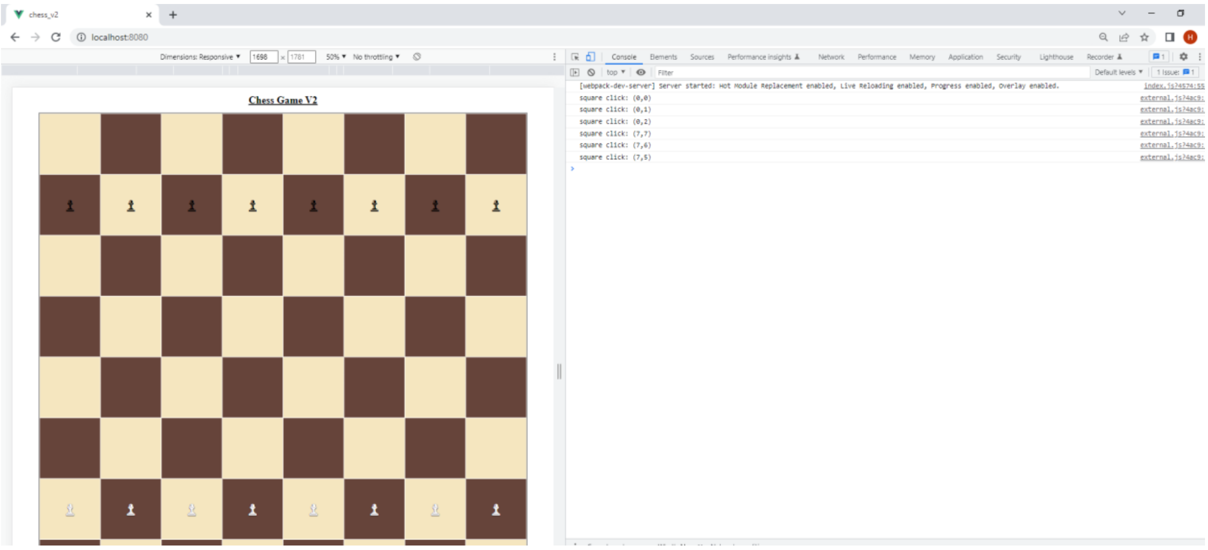
Going to the local host, the webpage is running locally on my computer


chess_v2
localhost:8080
Chess Game V2



On clicking a square, a message is printed to the console to show that the correct square click is registered

.



The code for this user interface is below

```
<template>
    <h1>Chess Game V2</h1>
    <div :style="cssVars" class="chess_board">
        <table>
            <tr v-for="[row_num, row] in board.entries()" :key="row">
                <td v-for="[col_num, square] in row.entries()"
:key="square" @click="square_click(row_num, col_num)">
                    <!-- {{row_num}}, {{col_num}} -->
                    <!-- {{square !== null? square: ""}} -->
```

```
                    <!-- playing around with white pieces -->
                    <!-- https://stackoverflow.com/questions/4772906/css-is-it-possible-to-add-a-black-outline-around-each-character-in-text -->
                    <span style="color:white; text-shadow: 1px 0 0 #000, 0 -1px 0 #000, 0 1px 0 #000, -1px 0 0 #000;" v-if="square == '♙'">♟</span>
                    <!-- <span style="color:black" v-if="square == '♙'">♙</span> -->
                    <!-- <span style="color:white" v-if="square == '♙'">♙</span> -->

                    <!-- <span style="color:black; text-shadow: 1px 0 0 #000, 0 -1px 0 #000, 0 1px 0 #000, -1px 0 0 #000;" v-else-if="square == '♟'">♟</span> -->
                    <span style="color:black" v-else-if="square == '♟'">♟</span>
                    <!-- <span v-else></span> -->
                </td>
            </tr>
        </table>
    </div>

    <h2>Turn {{turn_num}}: {{next_to_go == 'user'? "your turn": "computer's turn"}}</h2>

    <div class="option_button">
        <button>Concede</button>
        <button>Reset</button>
    </div>

    <div class="pieces_left_table">
        <table>
            <tr>
                <th>White Pieces Left: {{pieces_left.black}}/6</th>
                <th>Black Pieces Left: {{pieces_left.white}}/6</th>
            </tr>
            <tr>
                <td>{{ '♙'.repeat(pieces_left.white) }}</td>
                <td>{{ '♟'.repeat(pieces_left.black) }}</td>
            </tr>
        </table>
    </div>

    <h2>Previous Moves:</h2>

    <div class="previous_moves_table">
        <table>
            <tr>
                <th>White previous moves:</th>
```

```
                <th>Black previous moves:</th>
            </tr>

            <tr>
                <td>
                    <div v-for="move in previous_moves" :key="move.num">
                        <span v-if="move.player=='white'">
                            Move {{move.num}}: ♙ {{ move.from }} to {{
move.to }}
                        </span>
                    </div>
                </td>
                <td>
                    <div v-for="move in previous_moves" :key="move.num">
                        <span v-if="move.player=='black'">
                            Move {{move.num}}: ♟ {{ move.from }} to {{
move.to }}
                        </span>
                    </div>
                </td>


            </tr>
        </table>
    </div>

</template>

<script>
    import {handle_square_click} from '@/assets/scripts/external.js'
    // import * from '@/assets/scripts/external.js'
    const white_sq_color = '#f5e6bf';
    const black_sq_color = '#66443a';
    // white pawn: '♙'; black pawn: '♟'
    const pawn_white = '♙';
    const pawn_black = '♟';

    export default {
        name: "ChessGame",
        ready_for_next_move: false,
        // pawn characters
        // https://en.wikipedia.org/wiki/Chess_symbols_in_Unicode
        data(){return{
            board: [
                Array(8).fill(null),
                Array(8).fill(pawn_black),
                Array(8).fill(null),
                Array(8).fill(null),
```

```
                Array(8).fill(null),
                Array(8).fill(null),
                Array(8).fill(pawn_white),
                Array(8).fill(null)
            ],
            next_to_go: 'user',
            turn_num: 1,
            pieces_left: {
                black: 2,
                white: 3
            },
            previous_moves: [
                { num: 1, player: "white", from: "B1", to: "B3" },
                { num: 2, player: "black", from: "A5", to: "A4" },
                { num: 3, player: "white", from: "B3", to: "A4" },
                { num: 4, player: "black", from: "E5", to: "E3" },
            ],
        }},
        methods: {
            square_click(row, col) {
                // console.log(`square click: (${row},${col})`)
                handle_square_click(row, col)
            }
        },
        // https://www.telerik.com/blogs/passing-variables-to-css-on-a-
vue-component
        computed: {
            cssVars(){return{
                '--black_sq_color': black_sq_color,
                '--white_sq_color': white_sq_color,
            }}
        }
    }
</script>

<style scoped>
    /* :root {
        https://www.vectorstock.com/royalty-free-vector/chess-field-in-
beige-and-brown-colors-vector-24923385
        https://imagecolorpicker.com/en
        --black_sq_color: #66443a;
        --white_sq_color: #f5e6bf;
    } */

    h1 {
        text-align: center;
        text-decoration: underline;
    }
```

```
    h2 {
        text-align: center;
        font-size: 32px;
    }
    table {
        margin: auto;
        table-layout: fixed;
        text-align: center;
    }

    .option_button {
        margin: auto;
        text-align: center;
        /* margin-left: 0.125vw;
        margin-right: 0.125vw; */
        margin-left: 20px;
        margin-right: 20px;
    }

    .chess_board td {
        height: 10.5vh;
        width: 10.5vh;
        /* dimension so it is all in view for ipad air portrait  (820 *
1180) */
        /* height: 12vw;
        width: 12vw; */
        text-align: center;
        /* border: 3px black solid; */
        font-size: 50px;
        /* font-weight: 100; */

    }
    .chess_board table {
        border: 1px black solid;
    }

    /* these odd and even rules dictate the color of chess board squares
*/
    .chess_board tr:nth-child(2n-1) > td:nth-child(2n-1) {
    background-color: var(--white_sq_color);
        /* color: black; */
    }
    .chess_board tr:nth-child(2n-1) > td:nth-child(2n) {
        background-color: var(--black_sq_color);
        /* color: white; */
    }
    .chess_board tr:nth-child(2n) > td:nth-child(2n-1) {
        background-color: var(--black_sq_color);
```

```
        /* color: white; */
    }
    .chess_board tr:nth-child(2n) > td:nth-child(2n) {
        background-color: var(--white_sq_color);
        /* color: black; */
    }

    .previous_moves_table th, td {
        width: 35vw;
        /* border: 3px solid black; */
        font-size: 28px;
    }
    .pieces_left_table th, td {
        width: 35vw;
        /* border: 3px solid black; */
        font-size: 28px;
    }
</style>
```

It is not fully finished as the idea of developing a VUE js GUI was shelved.

On the one hand there are many benefits to a VUE GUI. Most importantly, the html content is dynamically updated from JavaScript variables. For example the board shown automatically updates with the content of the 2d array board. The main downside however is that the single page nature of a VUE webpage clashed with what I wanted the flask server to do. This is where one single HTTP request is sent and instead multiple pages are handled by JavaScript loading different components. I used this source and decided that I wanted to implement my user interface like this: https://www.digitalocean.com/community/tutorials/how-to-add-authentication-to-your-app-with-flask-login

I then moved on to implementing the backend chess engine in python. I knew that I intended to implement my chess engine using decision trees and specifically the minimax function. This function is suitable as chess is a complete information game as there is not information hidden such as a hand of cards. It is also a zero sum game as one player's good move worsens another player's chance of winning. It is also turn based.

To implement minimax I would need 2 functions. A utility function to determine a number for how favourable the current game state is to the maximiser (utility) and a child game state generating function that could provide all possible game states that could be achieved in one move from the current game state.

To implement the utility function I would need a robust game over function and a static evaluation function to provide an approximate guess as to the utility of a board state.

To implement the child game state generating function I will need to have a function to action a move and generate a child game state as well as a function to generate all legal moves of the current game state.

These function would need to be rigorously tested. If they contained logic errors, then the minimax function would fail without and easy way to determine why. However there functions themselves are hard to implement and so they would need to be broken down into simpler function. I clearly would also need to employ some high quality testing in order to produce sufficiently reliable code.

I decided that at its most abstracted, I would be solving this problem with a series objects built on vectors and rules. So I began by creating a vector class. It was sufficiently simple that it didn't need any major debugging. This would still provide me a good opportunity to get to grips with my unit-test process.
I decided to use a library called data driven tests (ddt) as it would let me write a single function to execute a single type of test, then run this test for every set of test data specified in a given file.

I thought that this would be better that generating the test data and expected outcome withing the test function as this method is more transparent and reduces the risk of logic errors within the tests themselves.

Here was my vector class (**vector.py**)

```
# import dataclass to reduce boilerplate code
from dataclasses import dataclass


# frozen = true means that the objects will be immutable
@dataclass(frozen=True)
class Vector():
    # 2d vector has properties i and j
    i: int
    j: int

    # code to allow for  +  -  and  *  operators to be used with vectors
    def __add__(self, other):
        assert isinstance(other, Vector), "both objects must be instances of
the Vector class"
        return Vector(
            i=self.i + other.i,
            j=self.j + other.j
        )
    def __sub__(self, other):
        assert isinstance(other, Vector), "both objects must be instances of
the Vector class"
        return Vector(
            i=self.i - other.i,
```

```
            j=self.j - other.j
        )

    def __mul__(self, multiplier: int):
        return Vector(
            i=self.i * multiplier,
            j=self.j * multiplier
        )

    # check if a vector is in board
    def in_board(self):
        """Assumes that the current represented vector is a position vector
        checks if it points to a square that isn't in the chess board"""
        return self.i in range(8) and self.j in range(8)

    # alternative way to create instance, construct from chess square
    @classmethod
    def construct_from_square(cls, to_sqr):
        """Example from and to squares are A3 -> v(0, 2) and to B4 -> v(1,
3)"""
        to_sqr = to_sqr.upper()
        letter, number = to_sqr

        # map letters and numbers to 0 to 7 and create new vector object
        return cls(
            i=ord(letter.upper()) - ord("A"),
            j=int(number)-1
        )

    # this function is the reverse and converts a position vector to a square
    def to_square(self) -> str:
        letter = chr(self.i + ord("A"))
        number = self.j+1
        return f"{letter}{number}"

    # this function checks if 2 vectors are equal
    def __eq__(self, other) -> bool:
        try:
            # assert same subclass like rook
            assert isinstance(other, type(self))
            assert self.i == other.i
            assert self.j == other.j

        except AssertionError:
            return False
        else:
            return True
```

```
    # used to put my objects in set
    def __hash__(self):
        return hash((self.i, self.j))
```

and here were my tests
starting with the python file containing the logic to run the unit tests (**test_vector.py)**

```
import ddt
import unittest

# from vector.vector import Vector
from vector import Vector

# function for path to vector related test data
def test_path(file_name):
    return f"./test_data/vector/{file_name}.yaml"

# augment my test case class with ddt decorator
@ddt.ddt
class Test_Case(unittest.TestCase):

    # performs many checks of construct form square
    @ddt.file_data(test_path("from_square"))
    def test_square_to_vector(self, square, expected_vector):
        self.assertEqual(
            Vector.construct_from_square(square),
            Vector(*expected_vector),
            msg=f"\nVector.construct_from_square('{square}')  !=  Vector(i={ex
pected_vector[0]}, j={expected_vector[1]})"
        )

    # performs many checks of adding vectors
    @ddt.file_data(test_path("vector_add"))
    def test_add_vectors(self, vector_1, vector_2, expected_vector):
        self.assertEqual(
            Vector(*vector_1) + Vector(*vector_2),
            Vector(*expected_vector),
            msg=f"\nVector(*{vector_1}) +
Vector(*{vector_2})  !=  Vector(*{expected_vector})"
        )

    # performs many checks of multiplying vectors
    @ddt.file_data(test_path("vector_multiply"))
    def test_multiply_vectors(self, vector, multiplier, expected):
        self.assertEqual(
            Vector(*vector) * multiplier,
            Vector(*expected)
        )
```

```
    # many tests of vector in board
    @ddt.file_data(test_path("vector_in_board"))
    def test_in_board(self, vector, expected):
        self.assertEqual(
            Vector(*vector).in_board(),
            expected,
            msg=f"\nVector(*{vector}).in_board()  !=  {expected}"
        )

    # many tests of vector out of board
    @ddt.file_data(test_path('vector_to_square'))
    def test_to_square(self, vector, expected):
        self.assertEqual(
            Vector(*vector).to_square(),
            expected
        )

if __name__ == '__main__':
    unittest.main()
```

Here are the test data files:

from_square.yaml

```
test1:
  square: 'G3'
  expected_vector: [6, 2]

test2:
  square: 'A8'
  expected_vector: [0, 7]

test3:
  square: 'H1'
  expected_vector: [7, 0]

test4:
  square: 'C4'
  expected_vector: [2, 3]

test5:
  square: 'F6'
  expected_vector: [5, 5]

# # invalid
# test6:
#   square: 'A5'
```

```yaml
#   expected_vector: [5, 5]
```

Vector_add.yaml

```yaml
test1:
  vector_1: [1, 4]
  vector_2: [4, 6]
  expected_vector: [5, 10]

test2:
  vector_1: [1, 7]
  vector_2: [4, 6]
  expected_vector: [5, 13]

test3:
  vector_1: [1, 3]
  vector_2: [0, 6]
  expected_vector: [1, 9]

test4:
  vector_1: [-1, 32]
  vector_2: [3, 6]
  expected_vector: [2, 38]

test5:
  vector_1: [6, 0]
  vector_2: [6, 7]
  expected_vector: [12, 7]

# # invalid delete me
# test6:
#   vector_1: [6, 0]
#   vector_2: [6, 7]
#   expected_vector: [0, 7]
```

Vector_in_board.yaml

```yaml
test1:
  vector: [0, 0]
  expected: True

test2:
  vector: [7, 0]
  expected: True
```

```yaml
test3:
  vector: [0, 7]
  expected: True
test4:
  vector: [7, 7]
  expected: True

test5:
  vector: [4, 6]
  expected: True

test6:
  vector: [3, 2]
  expected: True

test7:
  vector: [-1, -1]
  expected: False

test8:
  vector: [-5, 4]
  expected: False

test9:
  vector: [8, 7]
  expected: False

test10:
  vector: [-4, 4]
  expected: False

test11:
  vector: [10, 4]
  expected: False

# adding comment inside causes logic error and false pass on test
# # invalid delete me
# test:
#   vector: [4, 6]
#   expected: True
```

Vector_multiply.yaml

```yaml
test1:
  vector: [1, 1]
  multiplier: 1
  expected: [1, 1]
test2:
```

```yaml
  vector: [1, 1]
  multiplier: -1
  expected: [-1, -1]
test3:
  vector: [1, 1]
  multiplier: 5
  expected: [5, 5]
test4:
  vector: [5, 0]
  multiplier: 0
  expected: [0, 0]
```

vector_to_square.yaml

```yaml
test1:
  vector: [0, 0]
  expected: A1
test2:
  vector: [5, 7]
  expected: F8
test3:
  vector: [6, 2]
  expected: G3
test4:
  vector: [0, 6]
  expected: A7
test5:
  vector: [7, 7]
  expected: H8
```

as can be seen with the console:

```
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4> python -i test_vector.py
>>> list(filter(lambda name: name[:5] == "test_", dir(Test_Case)))
['test_add_vectors_1_test1', 'test_add_vectors_2_test2', 'test_add_vectors_3_test3', 'test_add_
vectors_4_test4', 'test_add_vectors_5_test5', 'test_in_board_01_test1', 'test_in_board_02_test2
', 'test_in_board_03_test3', 'test_in_board_04_test4', 'test_in_board_05_test5', 'test_in_board
_06_test6', 'test_in_board_07_test7', 'test_in_board_08_test8', 'test_in_board_09_test9', 'test
_in_board_10_test10', 'test_in_board_11_test11', 'test_multiply_vectors_1_test1', 'test_multipl
y_vectors_2_test2', 'test_multiply_vectors_3_test3', 'test_multiply_vectors_4_test4', 'test_squ
are_to_vector_1_test1', 'test_square_to_vector_2_test2', 'test_square_to_vector_3_test3', 'test
_square_to_vector_4_test4', 'test_square_to_vector_5_test5', 'test_to_square_1_test1', 'test_to
_square_2_test2', 'test_to_square_3_test3', 'test_to_square_4_test4', 'test_to_square_5_test5']

>>>
```

The decorators I added to my tests are unusual as they return many mutations of my original generic function that complete a specific test from my yaml file. This allows me to perform targeted testing

```
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4> python -m unittest
..........................................................................
----------------------------------------------------------------------
Ran 74 tests in 0.301s

OK
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4> python -m unittest test_vector
..............................
----------------------------------------------------------------------
Ran 30 tests in 0.005s

OK
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4> python -m unittest test_vector.Test_Case.test_to_square_4_test4
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4>
```

As is shown above, I can run all test, just vector tests or a specific test form a yaml file. I know the tests are working as I can change some of the tests data to create a test that I expect to fail.

For example:

```yaml
# adding comment inside causes logic error and false pass on test
# invalid delete me
test:
  vector: [4, 6]
  expected: False
```

I have uncommented out this invalid test in the **vector_in_board.yaml** file
When I run the tests:

```
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4> python -m unittest test_vector
...............F.............
======================================================================
FAIL: test_in_board_12_test (test_vector.Test_Case)
test_in_board_12_test
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\henry\AppData\Local\Programs\Python\Python310\lib\site-packages\ddt.py", line 220, in wrapper
    return func(self, *args, **kwargs)
  File "C:\Users\henry\Documents\computing coursework\prototype 2\v2.4\test_vector.py", line 44, in test_in_board
    self.assertEqual(
AssertionError: True != False :
Vector(*[4, 6]).in_board()  !=  False

----------------------------------------------------------------------
Ran 31 tests in 0.007s

FAILED (failures=1)
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4>
```

There is a summary of which of the tests fail and a traceback, including a message to help determine why one of the tests failed.

As part of testing my tests in this way I realised that my tests weren't working. I determined that this was because I had names them all test in my yaml file. To fix this I named them test1, test2, ect.

I think it is important to document tests and code at the same time as they were developed in tandem. I didn't move to code the next stage until all of my vector untit tests were working. This was valuable as it allowed me to use the vector module with the assumption that it was working perfectly in unit tests and code for later modules that built upon my vectors module.

The next logic to code was the various rules for how pieces could move on a chess board. To do this I created a pieces module.

I wanted to make a class for each piece to describe its value (needed for the utility function) and the ways in which is can move within a chess board. As I knew that some logic would be repeated withing the piece classes I decided to have a parent class piece and a series of child classes that inherit form it. Since I also knew that I would want to do general operations on a set of pieces later on (such as iterate through them for movement vectors or to sum up their value) I decided to use an abstract base class. This ensures that all of the piece classes have some key attributes and methods (they share the same interface). This means that I won't need to differentiate between pieces and have different logic for each one (with the exception of the king).

I found a series of standard values and matrices for determining addition value based on location within the board on the following website https://www.chessprogramming.org/Simplified_Evaluation_Function

They were not in a useful form (e.g. csv download):

Bishops

```
// bishop
-20,-10,-10,-10,-10,-10,-10,-20,
-10,  0,  0,  0,  0,  0,  0,-10,
-10,  0,  5, 10, 10,  5,  0,-10,
-10,  5,  5, 10, 10,  5,  5,-10,
-10,  0, 10, 10, 10, 10,  0,-10,
-10, 10, 10, 10, 10, 10, 10,-10,
-10,  5,  0,  0,  0,  0,  5,-10,
-20,-10,-10,-10,-10,-10,-10,-20,
```

We avoid corners and borders. Additionally we prefe

To solve this I used a jupyter notebook to process them

```python
def dec_gen_to_list(generator_func):
    def wrapper(*args, **kwargs):
        return list(
            generator_func(*args, **kwargs)
        )
    wrapper.__name__ == generator_func.__name__
    return wrapper
```

Python

```python
@dec_gen_to_list
def str_to_matrix(str_matrix):
    @dec_gen_to_list
    def str_to_row(row):
        for e in row.split(","):
            if not e:
                continue
            e = e.strip()
            yield int(e)
            # print(f"'{e}'")
    for row in str_matrix.split("\n"):
        if not row:
            continue

        # print(f"row:   '{row}'")
        yield str_to_row(row)
```

Python

```python
pawn = str_to_matrix(""" 0,  0,  0,  0,  0,  0,  0,  0,
50, 50, 50, 50, 50, 50, 50, 50,
10, 10, 20, 30, 30, 20, 10, 10,
 5,  5, 10, 25, 25, 10,  5,  5,
 0,  0,  0, 20, 20,  0,  0,  0,
 5, -5,-10,  0,  0,-10, -5,  5,
 5, 10, 10,-20,-20, 10, 10,  5,
 0,  0,  0,  0,  0,  0,  0,  0""")
pawn
```

Python

```
[[0, 0, 0, 0, 0, 0, 0, 0],
 [50, 50, 50, 50, 50, 50, 50, 50],
 [10, 10, 20, 30, 30, 20, 10, 10],
 [5, 5, 10, 25, 25, 10, 5, 5],
 [0, 0, 0, 20, 20, 0, 0, 0],
 [5, -5, -10, 0, 0, -10, -5, 5],
 [5, 10, 10, -20, -20, 10, 10, 5],
 [0, 0, 0, 0, 0, 0, 0, 0]]
```

```python
knight = str_to_matrix("""-50,-40,-30,-30,-30,-30,-40,-50,
-40,-20,  0,  0,  0,  0,-20,-40,
-30,  0, 10, 15, 15, 10,  0,-30,
-30,  5, 15, 20, 20, 15,  5,-30,
-30,  0, 15, 20, 20, 15,  0,-30,
-30,  5, 10, 15, 15, 10,  5,-30,
-40,-20,  0,  5,  5,  0,-20,-40,
-50,-40,-30,-30,-30,-30,-40,-50,""")
knight
```

```
[[-50, -40, -30, -30, -30, -30, -40, -50],
 [-40, -20, 0, 0, 0, 0, -20, -40],
 [-30, 0, 10, 15, 15, 10, 0, -30],
 [-30, 5, 15, 20, 20, 15, 5, -30],
 [-30, 0, 15, 20, 20, 15, 0, -30],
 [-30, 5, 10, 15, 15, 10, 5, -30],
 [-40, -20, 0, 5, 5, 0, -20, -40],
 [-50, -40, -30, -30, -30, -30, -40, -50]]
```

(repeat for all pieces)

I then wrote them to a json file for future use.

```python
import json

with open("value_matrices.json", "w") as file:
    file.write(
        json.dumps(
            {
                'pawn': pawn,
                'knight': knight,
                'bishop': bishop,
                'castle': rook,
                'queen': queen,
                'king_start': king_start,
                'king_end': king_end
            },
        )
    )
```

With this strategy for valuing a piece and my testes vector library I created my pieces library

pieces.py

```python
# to do: make value and value matrix unchangeable of private
# https://stackoverflow.com/questions/31457855/cant-instantiate-abstract-class-with-abstract-methods

# import libraries and other local modules
import abc
from itertools import chain as iter_chain, product as iter_product
from typing import Callable
```

```python
from vector import Vector
from assorted import ARBITRARILY_LARGE_VALUE

# Here is an abstract base class for piece,
# it dictates that all child object have the specified abstract attributes
else and error will occur
# this ensures that all piece objects have the same interface


class Piece(abc.ABC):

    # required
    # color is public
    color: str | None
    # value and value matrix is protected
    # value is inherent value
    _value: int
    # value matrix is additional value based on location
    _value_matrix: tuple[tuple[float]]

    # must be given before init
    @abc.abstractproperty
    def _value_matrix(): pass

    @abc.abstractproperty
    def _value(): pass

    @abc.abstractproperty
    def color(): pass

    @abc.abstractmethod
    def symbol(self) -> str:
        """uses color to determine the appropriate symbol"""

    # not needed as abstract method as come classes will nor override
    def __init__(self, color):
        self.color = color
        self.last_move = None

    # this should use the position vector and value matrix to get the value of
the piece
    def get_value(self, position_vector: Vector):
        # flip if black as matrices are all for white pieces
        if self.color == "W":
            row, column = 7-position_vector.j, position_vector.i
        else:
            row, column = position_vector.j, position_vector.i

        # return sum of inherent value + value relative to positon on board
```

```python
        return self._value + self._value_matrix[row][column]

    # this function should yield all the movement vector tha the piece can
move by
    # this doesn't account for check and is based on rules specific to each
piece as well an checking if a vector is outside the board

    @abc.abstractmethod
    def generate_movement_vectors(self, pieces_matrix, position_vector):
        pass

    # when str(piece) called give the symbol

    def __str__(self):
        # return f"{self.color}{self.symbol}"
        return self.symbol()

    # standard repr method
    def __repr__(self):
        return f"{type(self).__name__}(color='{self.color}')"

    # logic that would be otherwise repeated in many of the child classes
    # determines the contents of a given square
    def square_contains(self, square):
        """returns 'enemy' 'ally' or None for empty"""
        # check if empty
        if square is None:
            return "empty"
        # else the square must contain a piece, so examine its color
        if square.color == self.color:
            return "ally"
        else:
            return "enemy"

    # again reduces repeated logic
    # checks the result of a position vector
    # if not illegal (out of board) the square contents is returned
    def examine_position_vector(self, position_vector: Vector, pieces_matrix):
        """returns 'enemy' 'ally' 'empty' or 'illegal'  """
        # check if the vector is out of the board
        if not position_vector.in_board():
            return 'illegal'
        # for the rest of the code I can assume the vector is in board

        # else get the square at that vector
        row, column = 7-position_vector.j, position_vector.i
        square = pieces_matrix[row][column]
```

```
        # examine its contents
        return self.square_contains(square)

    # equality operator
    def __eq__(self, other):
        try:
            # assert same subclass like rook
            assert isinstance(other, type(self))
            assert self.color == other.color

            # i am not checking that pieces had the same last move as I want
to compare kings without for the check function
            # assert self.last_move == other.last_move

            # value and value_matrix should never be changes
        except AssertionError:
            return False
        else:
            return True

    # making pieces hashable allows for pieces matrices to be hashed and
allows for pieces and pieces matrices to be put in sets,
    # also essential for piping data between python interpreter instances
(different threads) for multitasking
    def __hash__(self):
        return hash((self.symbol(), self.color, self.last_move))


# this class inherits from Piece an so it inherits some logic and some
requirements as to how its interface should be
# as many child classes are similar I will explain this one in depth and then
only explain notable features of others
class Pawn(Piece):
    # defining abstract properties, needed before init
    _value = 100
    _value_matrix: tuple[tuple[float]] = [
        [0, 0, 0, 0, 0, 0, 0, 0],
        [50, 50, 50, 50, 50, 50, 50, 50],
        [10, 10, 20, 30, 30, 20, 10, 10],
        [5, 5, 10, 25, 25, 10, 5, 5],
        [0, 0, 0, 20, 20, 0, 0, 0],
        [5, -5, -10, 0, 0, -10, -5, 5],
        [5, 10, 10, -20, -20, 10, 10, 5],
        [0, 0, 0, 0, 0, 0, 0, 0]
    ]
    color = None

    # define symbol method (str method)
    def symbol(self): return f"{self.color}P"
```

```
    # override init constructor
    def __init__(self, color):
        # perform super's instructor
        super().__init__(color)

        # but in addition...
        # decide the vectors that the piece can move now as it is based on
color
        multiplier = 1 if color == "W" else -1

        # method defined here as it only used here
        # decided if the pawn is allowed to move foreward 2 based on square
contents and last move
        def can_move_foreword_2(square):
            return square is None and self.last_move is None

        # tuple contains pairs of vector and contition that must be met
        # (in the form of a function that takes square and returns a boolean)
        self.movement_vector_and_condition: tuple[Vector, Callable] = (
            # v(0, 1) for foreword
            (Vector(0, multiplier), lambda square:
self.square_contains(square) == "empty"),
            # v(0, 2) for foreword as first move
            (Vector(0, 2*multiplier), can_move_foreword_2),
            # v(-1, 1) and v(1, 1) for take
            (Vector(1, multiplier), lambda square:
self.square_contains(square) == 'enemy'),
            (Vector(-1, multiplier), lambda square:
self.square_contains(square) == 'enemy'),
        )

    # generate movement vectors
    def generate_movement_vectors(self, pieces_matrix, position_vector):
        # iterate through movement vectors and conditions
        for movement_vector, condition in self.movement_vector_and_condition:
            # get resultant vector
            resultant_vector = position_vector + movement_vector
            # if vector_out of range continue
            if not resultant_vector.in_board():
                continue

            # get the contents of the square corresponding to the resultant
            row, column = 7-resultant_vector.j, resultant_vector.i
            piece = pieces_matrix[row][column]

            # if the condition is met, yield the vector
            if condition(piece):
```

```
                yield movement_vector


class Knight(Piece):
    _value = 320
    _value_matrix: tuple[tuple[float]] = [
        [-50, -40, -30, -30, -30, -30, -40, -50],
        [-40, -20, 0, 0, 0, 0, -20, -40],
        [-30, 0, 10, 15, 15, 10, 0, -30],
        [-30, 5, 15, 20, 20, 15, 5, -30],
        [-30, 0, 15, 20, 20, 15, 0, -30],
        [-30, 5, 10, 15, 15, 10, 5, -30],
        [-40, -20, 0, 5, 5, 0, -20, -40],
        [-50, -40, -30, -30, -30, -30, -40, -50]
    ]
    color = None

    # n for knight as king takes k
    def symbol(self): return f"{self.color}N"

    def generate_movement_vectors(self, pieces_matrix, position_vector):
        # this function yields all 8 possible vectirs
        def possible_movement_vectors():
            vectors = (Vector(2, 1), Vector(1, 2))
            # for each x multiplier, y multiplier and vector combination
            for i_multiplier, j_multiplier, vector in iter_product((-1, 1), (-
1, 1), vectors):
                # yield corresponding vector
                yield Vector(
                    vector.i * i_multiplier,
                    vector.j * j_multiplier
                )
        # iterate through movement vectors
        for movement_vector in possible_movement_vectors():
            # get resultant
            resultant_vector = position_vector + movement_vector
            # look at contents of square
            contents =
self.examine_position_vector(position_vector=resultant_vector,
pieces_matrix=pieces_matrix)
            # if square is empty yield vector
            if contents == "empty":
                yield movement_vector


class Bishop(Piece):
    _value = 330
    _value_matrix: tuple[tuple[float]] = [
        [-20, -10, -10, -10, -10, -10, -10, -20],
```

```
        [-10, 0, 0, 0, 0, 0, 0, -10],
        [-10, 0, 5, 10, 10, 5, 0, -10],
        [-10, 5, 5, 10, 10, 5, 5, -10],
        [-10, 0, 10, 10, 10, 10, 0, -10],
        [-10, 10, 10, 10, 10, 10, 10, -10],
        [-10, 5, 0, 0, 0, 0, 5, -10],
        [-20, -10, -10, -10, -10, -10, -10, -20]
    ]
    color = None

    def symbol(self): return f"{self.color}B"

    def generate_movement_vectors(self, pieces_matrix, position_vector):
        # sourcery skip: use-itertools-product

        # repeat for all 4 vector directions
        for i, j in iter_product((1, -1), (1, -1)):
            unit_vector = Vector(i, j)
            # iterate through length multipliers
            for multiplier in range(1, 8):
                # get movement and resultant vectors
                movement_vector = unit_vector * multiplier
                resultant_vector = position_vector + movement_vector

                # examine the contents of the square and use switch case to
decide behaviour
                match
self.examine_position_vector(position_vector=resultant_vector,
pieces_matrix=pieces_matrix):
                    case 'illegal':
                        # if vector extends out of the board stop extending
                        break
                    case 'ally':
                        # break of of for loop (not just match case)
                        # as cannot hop over piece so don't explore longer
vectors in same direction
                        break
                    case 'enemy':
                        # this is a valid move
                        yield movement_vector
                        # break of of for loop (not just match case)
                        # as cannot hop over piece so don't explore longer
vectors in same direction
                        break
                    case 'empty':
                        # is valid
                        yield movement_vector
                        # and keep exploring, don't break
```

```
class Rook(Piece):
    _value = 500
    _value_matrix: tuple[tuple[float]] = [
        [0, 0, 0, 0, 0, 0, 0, 0],
        [5, 10, 10, 10, 10, 10, 10, 5],
        [-5, 0, 0, 0, 0, 0, 0, -5],
        [-5, 0, 0, 0, 0, 0, 0, -5],
        [-5, 0, 0, 0, 0, 0, 0, -5],
        [-5, 0, 0, 0, 0, 0, 0, -5],
        [-5, 0, 0, 0, 0, 0, 0, -5],
        [0, 0, 0, 5, 5, 0, 0, 0]
    ]
    color = None

    # r for rook
    def symbol(self): return f"{self.color}R"

    # this code is very similar in structure to that of a bishop just with
different direction vectors
    def generate_movement_vectors(self, pieces_matrix, position_vector):
        # sourcery skip: use-itertools-product
        unit_vectors = (
            Vector(0, 1),
            Vector(0, -1),
            Vector(1, 0),
            Vector(-1, 0),
        )
        # for unit_vector, multiplier in iter_product(unit_vectors, range(1,
8)):
        for unit_vector in unit_vectors:
            for multiplier in range(1, 8):
                movement_vector = unit_vector * multiplier
                resultant_vector = position_vector + movement_vector

                # note cases that contain only break are not redundant, they
break the outer for loop
                match
self.examine_position_vector(position_vector=resultant_vector,
pieces_matrix=pieces_matrix):
                    case 'illegal':
                        # if vector extends out of the board stop extending
                        break
                    case 'ally':
                        # break of of for loop (not just match case)
                        # as cannot hop over piece so don't explore longer
vectors in same direction
                        break
```

```
                    case 'enemy':
                        # this is a valid move
                        yield movement_vector
                        # break of of for loop (not just match case)
                        # as cannot hop over piece so don't explore longer
vectors in same direction
                        break
                    case 'empty':
                        # is valid
                        yield movement_vector
                        # and keep exploring, don't break


class Queen(Piece):
    _value = 900
    _value_matrix: tuple[tuple[float]] = [
        [-20, -10, -10, -5, -5, -10, -10, -20],
        [-10, 0, 0, 0, 0, 0, 0, -10],
        [-10, 0, 5, 5, 5, 5, 0, -10],
        [-5, 0, 5, 5, 5, 5, 0, -5],
        [0, 0, 5, 5, 5, 5, 0, -5],
        [-10, 5, 5, 5, 5, 5, 0, -10],
        [-10, 0, 5, 0, 0, 0, 0, -10],
        [-20, -10, -10, -5, -5, -10, -10, -20]
    ]
    color = None

    def symbol(self): return f"{self.color}Q"

    # this code also uses a similar structure to the rook or bishop
    def generate_movement_vectors(self, pieces_matrix, position_vector):
        # unit_vectors = (Vector(i, j) for i, j in iter_product((-1, 0, 1), (-
1, 0, 1)) if i != 0 and j != 0)
        unit_vectors = (Vector(i, j) for i, j in iter_product((-1, 0, 1), (-1,
0, 1)) if i != 0 or j != 0)

        # for unit_vector, multiplier in iter_product(unit_vectors, range(1,
8)):
        for unit_vector in unit_vectors:
            for multiplier in range(1, 8):
                movement_vector = unit_vector * multiplier
                resultant_vector = position_vector + movement_vector
                match
self.examine_position_vector(position_vector=resultant_vector,
pieces_matrix=pieces_matrix):
                    case 'illegal':
                        # if vector extends out of the board stop extending
                        break
                    case 'ally':
```

```
                        # break of of for loop (not just match case)
                        # as cannot hop over piece so don't explore longer
vectors in same direction
                        break
                    case 'enemy':
                        # this is a valid move
                        yield movement_vector
                        # break of of for loop (not just match case)
                        # as cannot hop over piece so don't explore longer
vectors in same direction
                        break
                    case 'empty':
                        # is valid
                        yield movement_vector
                        # and keep exploring, don't break


class King(Piece):
    # not needed as static eval does't add up kings value
    _value = ARBITRARILY_LARGE_VALUE

    # there are 2 matrices to represent the early and late game for the king
    value_matrix_early: tuple[tuple[float]] = [
        [-30, -40, -40, -50, -50, -40, -40, -30],
        [-30, -40, -40, -50, -50, -40, -40, -30],
        [-30, -40, -40, -50, -50, -40, -40, -30],
        [-30, -40, -40, -50, -50, -40, -40, -30],
        [-20, -30, -30, -40, -40, -30, -30, -20],
        [-10, -20, -20, -20, -20, -20, -20, -10],
        [20, 20, 0, 0, 0, 0, 20, 20],
        [20, 30, 10, 0, 0, 10, 30, 20]
    ]
    value_matrix_late = [
        [-50, -40, -30, -20, -20, -30, -40, -50],
        [-30, -20, -10, 0, 0, -10, -20, -30],
        [-30, -10, 20, 30, 30, 20, -10, -30],
        [-30, -10, 30, 40, 40, 30, -10, -30],
        [-30, -10, 30, 40, 40, 30, -10, -30],
        [-30, -10, 20, 30, 30, 20, -10, -30],
        [-30, -30, 0, 0, 0, 0, -30, -30],
        [-50, -30, -30, -30, -30, -30, -30, -50]
    ]
    color = None

    # initially value matrix is the early one
    _value_matrix: tuple[tuple[float]] = value_matrix_early

    # def __init__(self, *args, **kwargs):
    #     self._value_matrix = self.value_matrix_early
```

```
    #     super().__init__(self, *args, **kwargs)

    def symbol(self): return f"{self.color}K"

    # based on total pieces, changes the value matrix
    def update_value_matrix(self, pieces_matrix):
        # counts each empty square as 0 and each full one as 1 then sums them
up to get total pieces
        # if total pieces less than or equal to 10: then late game
        if sum(int(isinstance(square, Piece)) for square in
iter_chain(pieces_matrix)) <= 10:
            self.value_matrix = self.value_matrix_late
        # else early game
        else:
            self.value_matrix = self.value_matrix_early

    # this generates the movement vectors for the king
    def generate_movement_vectors(self, pieces_matrix, position_vector):
        # take the opportunity to update the value matrix
        self.update_value_matrix(pieces_matrix)

        # all 8 movement vectors
        unit_vectors = (Vector(i, j) for i, j in iter_product((-1, 0, 1), (-1,
0, 1)) if i != 0 or j != 0)

        # for each movement vector, get the resultant vector
        for movement_vector in unit_vectors:
            resultant_vector = position_vector + movement_vector

            # examine contents of square and use switch case to decide
behaviour
            match
self.examine_position_vector(position_vector=resultant_vector,
pieces_matrix=pieces_matrix):
                case 'illegal':
                    continue
                case 'ally':
                    continue
                case 'enemy':
                    # this is a valid move
                    yield movement_vector
                case 'empty':
                    # is valid
                    yield movement_vector


# used by other modules to convert symbol to piece
PIECE_TYPES = {
    'P': Pawn,
```

```
    'N': Knight,
    'B': Bishop,
    'R': Rook,
    'K': King,
    'Q': Queen
}

# if __name__ == '__main__':
# ensures that all classes are valid (not missing any abstract properties)
whenever the module is imported
Pawn('W')
Knight('W')
Bishop('W')
Rook('W')
Queen('W')
King('W')
```

I can remove one of the required abstract methods form the Rook class to show you the error this causes.

Change

```
    # # r for rook
    # def symbol(self): return f"{self.color}R"
```

Result

```
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4> python pieces.py
Traceback (most recent call last):
  File "C:\Users\henry\Documents\computing coursework\prototype 2\v2.4\pieces.py", line 466, in <module>
    Rook('W')
TypeError: Can't instantiate abstract class Rook with abstract method symbol
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4>
```

This behaviour is valid as it means that, if I am sure that a variable contains an object of type Piece, not matter which piece, I can be sure that the interface will be the same (e.g. I can assume they all have a symbol method).

For the King class I gave it a value which is arbitrarily high. This comes from a file called **assorted.py**

```
# this file is just a file of short assorted constants and functions that are
general in use
# It only contains small functions as I have tried to group large, similar
functions logically in there own file
```

```python
# this is used in the static evaluation and minimax process. It is used to represent infinity in a way that still allows comparrison
ARBITRARILY_LARGE_VALUE = 1_000_000

# this function is relatively redundant but allows for print statements in debugging
# in later iteration this may be replaced with logging.
# it is useful as it allows for DEBUG print statements without needing to remove them when finished
DEBUG = True
def dev_print(*args, **kwargs):
    if DEBUG:
        print(*args, **kwargs)


# this is an exception that allows for the game data to be bound to it
# this allows for the relevant chess game that caused the error to be examined afterwards
# it is a normal exception except the constructor has been modified to save the game data as a property
class __ChessExceptionTemplate__(Exception):
    def __init__(self, *args, **kwargs) -> None:
        # none if key not present
        self.game = kwargs.pop("game", None)

        super().__init__(*args, **kwargs)

# these are custom exceptions.
# they contain no logic but have distinct types allowing for targeted error handling
class InvalidMove(__ChessExceptionTemplate__):
    pass
class NotUserTurn(__ChessExceptionTemplate__):
    pass
```

The kings inherent value isn't relevant as both players will always have a king and so the value will always cancel out. I also used 2 value matrices for the king. I determined that the late game is when there are 10 or less total pieces though this is arbitrary.

The testing for the pieces module was as follows:

Test_pieces.py

```python
import unittest
import ddt
```

```python
import pieces
# as vector is already tested we can use it here and assume it won't cause any logic errors
from vector import Vector

def test_dir(file_name): return f"test_data/pieces/{file_name}.yaml"

EMPTY_PIECES_MATRIX = ((None,)*8,)*8

@ddt.ddt
class Test_Case(unittest.TestCase):

    # this test is based on testing the the movement vectors of a piece places at some position in an empty board are as expected
    @ddt.file_data(test_dir('test_empty_board'))
    def test_empty_board(self, piece_type, square, expected_move_squares):
        # deserialize
        position_vector = Vector.construct_from_square(square)
        piece: pieces.Piece = pieces.PIECE_TYPES[piece_type]('W')
        movement_vectors = piece.generate_movement_vectors(pieces_matrix=EMPTY_PIECES_MATRIX, position_vector=position_vector)
        resultant_squares = list(
            map(
                lambda movement_vector: (movement_vector+position_vector).to_square(),
                movement_vectors
            )
        )
        # assert as expected
        # can use sets to prevent order being an issue as vectors are hashable
        self.assertEqual(
            set(resultant_squares),
            set(expected_move_squares),
            msg=f"\n\nactual movement squares {sorted(resultant_squares)}  !=  expected movement squares {sorted(expected_move_squares)}\n{repr(piece)} at {square}"
        )


    # this test assert that that a pieces movement vectors are as expected when the piece is surrounded by other pieces
    @ddt.file_data(test_dir('test_board_populated'))
    def test_board_populated(self, pieces_matrix, square, expected_piece_symbol, expected_move_squares):
        # deserialize
        def list_map(function, iterable): return list(map(function, iterable))

        def descriptor_to_piece(descriptor) -> pieces.Piece:
```

```
            # converts WN to knight object with a color attribute of white
            if descriptor is None:
                return None

            color, symbol  = descriptor
            piece_type: pieces.Piece = pieces.PIECE_TYPES[symbol]
            return piece_type(color=color)

        def row_of_symbols_to_pieces(row): return
list_map(descriptor_to_piece, row)

        # update pieces_matrix replacing piece descriptors to piece objects
        pieces_matrix = list_map(row_of_symbols_to_pieces, pieces_matrix)

        position_vector = Vector.construct_from_square(square)
        row, column = 7-position_vector.j, position_vector.i

        # assert piece as expected
        piece: pieces.Piece = pieces_matrix[row][column]
        self.assertEqual(
            piece.symbol(),
            expected_piece_symbol,
            msg=f"\nPiece at square {square} was not the expected piece"
        )

        # assert movement vectors as expected
        movement_vectors =
piece.generate_movement_vectors(pieces_matrix=pieces_matrix,
position_vector=position_vector)
        resultant_squares = list(
            map(
                lambda movement_vector:
(movement_vector+position_vector).to_square(),
                movement_vectors
            )
        )
        self.assertEqual(
            set(resultant_squares),
            set(expected_move_squares),
                msg=f"\n\nactual movement squares
{sorted(resultant_squares)}  !=  expected movement squares
{sorted(expected_move_squares)}\n{repr(piece)} at {square}"
        )

if __name__ == '__main__':
    unittest.main()
```

test_board_populated.yaml

```
test1:
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [BP,   null, BP,   null, null, null, null, null],
    [null, WP,   null, null, null, null, null, null]
  ]
  square: B1
  expected_piece_symbol: WP
  expected_move_squares: [
    A2, C2, B2, B3
  ]
test2:
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, BP,   null, null, null, null],
    [null, null, null, null, BR,   null, null, null],
    [null, BR,   null, BQ,   null, null, WB,   null],
    [null, null, BP,   null, null, null, BP,   null],
    [null, WR,   null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null]
  ]
  square: D4
  expected_piece_symbol: BQ
  expected_move_squares: [
    D5,
    E4, F4, G4,
    E3, F2, G1,
    D3, D2, D1,
    C4,
    C5, B6, A7
  ]
test3:
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, BP,   null, null, null, null],
    [null, null, null, null, BR,   null, null, null],
    [null, BR,   null, BR,   null, null, WB,   null],
    [null, null, BP,   null, null, null, BP,   null],
    [null, WR,   null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null]
  ]
```

```
  square: D4
  expected_piece_symbol: BR
  expected_move_squares: [
    D5,
    E4, F4, G4,
    D3, D2, D1,
    C4,
  ]
test4:
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, BP, null, null, null, null],
    [null, null, WP, null, WP, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null]
  ]
  square: D7
  expected_piece_symbol: BP
  expected_move_squares: [
    D6, D5,
    C6, E6
  ]
test5:
  pieces_matrix: [
    [null, null, null, BK, null, null, null, null],
    [null, null, null, WP, null, null, null, null],
    [null, null, null, WK, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
  ]
  square: D8
  expected_piece_symbol: BK
  expected_move_squares: [
    C7, D7, E7,
    C8, E8
  ]
test6:
  pieces_matrix: [
    [BR,   BK,   BB,   BK,   BQ,   BB,   BK,   BR  ],
    [BP,   BP,   BP,   BP,   BP,   BP,   BP,   BP  ],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
```

```yaml
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [WP,   WP,   WP,   WP,   WP,   WP,   WP,   WP  ],
    [WR,   WN,   WB,   WK,   WQ,   WB,   WN,   WR  ]
  ]
  square: B1
  expected_piece_symbol: WN
  expected_move_squares: [A3, C3]
test7:
  pieces_matrix: [
    [BR,   BN,   BB,   BK,   BQ,   BB,   BN,   BR  ],
    [BP,   BP,   BP,   BP,   BP,   BP,   BP,   BP  ],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [WP,   WP,   WP,   WP,   WP,   WP,   WP,   WP  ],
    [WR,   WN,   WB,   WK,   WQ,   WB,   WN,   WR  ]
  ]
  square: G1
  expected_piece_symbol: WN
  expected_move_squares: [F3, H3]
```

test_empty_board.yaml

```yaml
test1:
  piece_type: 'N'
  square: E4
  expected_move_squares: [
    D2, F2, D6, F6, C5, C3, G5, G3
  ]
test2:
  piece_type: Q
  square: C3
  expected_move_squares: [
    A1, B2, D4, E5, F6, G7, H8,
    C1, C2, C4, C5, C6, C7, C8,
    A3, B3, D3, E3, F3, G3, H3,
    A5, B4, D2, E1
  ]
test3:
  piece_type: K
  square: B2
  expected_move_squares: [
    C1, C2, C3,
    B1, B3,
    A1, A2, A3,
  ]
```

```
test4:
  piece_type: P
  square: E3
  expected_move_squares: [
    E4, E5
  ]
test5:
  piece_type: R
  square: F6
  expected_move_squares: [
    F1, F2, F3, F4, F5, F7, F8,
    A6, B6, C6, D6, E6, G6, H6
  ]
test6:
  piece_type: B
  square: D7
  expected_move_squares: [
    C8, E6, F5, G4, H3,
    E8, C6, B5, A4,
  ]
test7:
  piece_type: K
  square: D8
  expected_move_squares: [
    C7, D7, E7,
    C8, E8
  ]
```

These tests are explained by the comments. They focus on testing the movement of the pieces (needed for the legal moves function).

This sub problem of deciding where a piece could move was made easier by abstracting away complications such as how check affects the legal moves. This additional logic will be added on by later modules that will rely on the pieces module. I wrote the tests as I coded the pieces module and so I was able to detect and fix issues as I developed the module until and the tests needed had been written and were working.

Evidence:

```
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4> python -m unittest test_pieces
..............
----------------------------------------------------------------------
Ran 14 tests in 0.004s

OK
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4>
```

There was, in hind sight a flaw with my testing of this module which I will get to later (mirroring where I was in my development journey when I discovered it).

Next I created a board state class. This class was to be responsible for keeping track of a single board state: a snapshot in the whole game. It would include a utility function, a move executer function and a generate legal moves function. It would be immutable. This was because I knew that the minimax function would want to make many different combinations of moves on a given board state without irreversibly mutating the object or needing to make a deep copy.

Here is the class I came up with:
Board_state.py

```
from copy import deepcopy
from dataclasses import dataclass
from itertools import product as iter_product

import pieces as pieces_mod
from assorted import ARBITRARILY_LARGE_VALUE
from vector import Vector

# this is a pieces matrix for the starting position is chess
# white is at the bottom as it is from the user's perspective and I am
currently assuming the user is white.
# I can change this in the frontend later
STARTING_POSITIONS: tuple[tuple[pieces_mod.Piece]] = (
    (
        pieces_mod.Rook(color="B"),
        pieces_mod.Knight(color="B"),
        pieces_mod.Bishop(color="B"),
        pieces_mod.Queen(color="B"),
        pieces_mod.King(color="B"),
        pieces_mod.Bishop(color="B"),
        pieces_mod.Knight(color="B"),
        pieces_mod.Rook(color="B")
    ),
    (pieces_mod.Pawn(color="B"),)*8,
    (None,)*8,
    (None,)*8,
    (None,)*8,
    (None,)*8,
    (pieces_mod.Pawn(color="W"),)*8,
    (
        pieces_mod.Rook(color="W"),
        pieces_mod.Knight(color="W"),
        pieces_mod.Bishop(color="W"),
        pieces_mod.Queen(color="W"),
        pieces_mod.King(color="W"),
        pieces_mod.Bishop(color="W"),
        pieces_mod.Knight(color="W"),
        pieces_mod.Rook(color="W")
    )
```

```
)

# frozen makes each instance of the board_state class is immutable
@dataclass(frozen=True)
class Board_State():
    next_to_go: str = "W"
    # the pieces matrix keeps track of the board positions and the pieces
    pieces_matrix: tuple[tuple[pieces_mod.Piece]] = STARTING_POSITIONS

    # this function outputs the board in a user friendly way
    # 8| BR  BN  BB  BQ  BK  BB  BN  BR
    # 7| ·   ·   BP  BP  BP  BP  BP  BP
    # 6| ·   ·   ·   ·   ·   ·   ·   ·
    # 5| BP  BP  ·   ·   ·   ·   ·   ·
    # 4| ·   ·   ·   WP  ·   WP  ·   ·
    # 3| ·   ·   ·   ·   ·   ·   ·   ·
    # 2| WP  WP  WP  ·   WP  ·   WP  WP
    # 1| WR  WN  WB  WQ  WK  WB  WN  WR
    #   (A   B   C   D   E   F   G   H )
    def print_board(self):
        # convert each piece to a symbol BP and replace none with dots
        # add numbers to left and add letters at the bottom
        numbers = range(8, 0, -1)
        letters = [chr(i) for i in range(ord("A"), ord("H")+1)]
        for row, number in zip(self.pieces_matrix, numbers):
            # clean up row of pieces
            symbols_row = map(lambda piece: piece.symbol() if piece else "· ",
row)
            pretty_row = f"{number}| {'  '.join(symbols_row)}"
            print(pretty_row)
        print(f"  ({'   '.join(letters)} )")

    def get_piece_at_vector(self, vector: Vector):
        # this function exists as it is a really common operation.
        # due to vector 0, 0 pointing the the bottom left not top left of the
2d array,
        # some correction is needed
        column, row = vector.i, 7-vector.j
        return self.pieces_matrix[row][column]

    # this function should yield all the pieces on the board
    def generate_all_pieces(self):
        # nested loop for i and j to iterate through all possible vectors
        for i, j in iter_product(range(8), range(8)):
            position_vector = Vector(i,j)
            # get the contents of the corresponding square
            piece = self.get_piece_at_vector(position_vector)
```

```
            # skip if none: skip if square empty
            if piece:
                yield piece, position_vector

    # this yields all pieces of a given color:
    # used when examining legal moves of a given player
    def generate_pieces_of_color(self, color=None):
        # be default give pieces of player next to go
        if color is None:
            color = self.next_to_go

        # return all piece and position vector pairs
        # filtered by piece must match in color
        yield from filter(
            # lambda piece, _: piece.color == color,
            lambda piece_and_vector: piece_and_vector[0].color == color,
            self.generate_all_pieces()
        )

    # determines if a given player is in check based on the player's color
    def color_in_check(self, color=None):
        # default is to check if the player next to go is in check
        if color is None:
            color = self.next_to_go

        # let is now be A's turn
        # I use this player a and b model to keep track of the logic here
        color_a = color
        color_b = "W" if color == "B" else "B"

        # we will examine all the movement vectors of B's pieces
        # if any of them could take the A's King then currently A is in check
as their king is threatened by 1 or more pieces (which could take it next
turn)
        # for each of b's pieces
        for piece, position_v in self.generate_pieces_of_color(color=color_b):
            movement_vs = piece.generate_movement_vectors(
                pieces_matrix=self.pieces_matrix,
                position_vector=position_v
            )
            # for each movement vector that the piece could make
            for movement_v in movement_vs:
                resultant = position_v + movement_v
                # check the contents of the square
                to_square = self.get_piece_at_vector(resultant)
                # As_move_threatens_king_A = isinstance(to_square,
pieces_mod.King) and to_square.color == color_a
                # if the contents is A's king then the a is in check.
```

```
                As_move_threatens_king_A = (to_square ==
pieces_mod.King(color=color_a))
                # if As_move_threatens_king_A then return true
                if As_move_threatens_king_A:
                    return True
        # if none of B's pieces were threatening to take A's king then A isn't
in check
        return False

    # this function is a generator to be iterated through.
    # it is responsible for yielding every possible move that a given player
can make
    # yields this as a position and a movement vector
    def generate_legal_moves(self):
        # iterate through all pieces belonging to player next to go
        for piece, piece_position_vector in
self.generate_pieces_of_color(color=self.next_to_go):
            movement_vectors = piece.generate_movement_vectors(
                pieces_matrix=self.pieces_matrix,
                position_vector=piece_position_vector
            )
            # iterate through movement vectors of this piece
            for movement_vector in movement_vectors:
                # examine resulting child game state
                child_game_state =
self.make_move(from_position_vector=piece_position_vector,
movement_vector=movement_vector)
                # determine if current player next to go (different to next to
go of child game state) is in check
                is_check_after_move =
child_game_state.color_in_check(color=self.next_to_go)
                # only yield the move if it doesn't result in check
                if not is_check_after_move:
                    yield piece_position_vector, movement_vector

    # determines if the game is over
    # returns over, winner
    def is_game_over_for_next_to_go(self):
        # sourcery skip: remove-unnecessary-else, swap-if-else-branches

        # in all cases, the game if over if a player has no legal moves left
        if not list(self.generate_legal_moves()):
            # if b in check
            if self.color_in_check():
                # checkmate for b, a wins
                winner = "W" if self.next_to_go == "B" else "B"
                return True, winner
            else:
```

```
                # stalemate
                return True, None
        # game not over
        return False, None

    # this function if responsible for generating a static evaluation for a
given board-stat
    # it should be used by a maximiser or minimiser
    # starting position should have and evaluation of 0
    def static_evaluation(self):
        # if over give an appropriate score for win loss or draw
        over, winner = self.is_game_over_for_next_to_go()
        if over:
            match winner:
                case None: multiplier = 0
                case "W": multiplier = 1
                case "B": multiplier = -1
            # return winner * ARBITRARILY_LARGE_VALUE
            return multiplier * ARBITRARILY_LARGE_VALUE

        # this function takes a piece as an argument and uses its color to
decide if its value should be positive or negative
        def get_piece_value(piece: pieces_mod.Piece, position_vector: Vector):
            # this function assumes white it maximizer and so white pieces
have a positive score and black negative
            match piece.color:
                case "W": multiplier = 1
                case "B": multiplier = (-1)
            value = multiplier * piece.get_value(position_vector)
            # print(f"{piece.symbol()} at {position_vector.to_square} has
value {value}")
            return value

        # for each piece, get the value (+/-)
        values = map(
            lambda x: get_piece_value(*x),
            self.generate_all_pieces()
        )
        # sum values for static eval
        return sum(values)


    def make_move(self, from_position_vector: Vector, movement_vector:
Vector):
        resultant_vector = from_position_vector + movement_vector

        # make a coppy of the position vector, deep coppy is used to ensure no
parts are shared be reference
        new_pieces_matrix = deepcopy(self.pieces_matrix)
```

```
        # convert to list
        new_pieces_matrix = list(map(list, new_pieces_matrix))

        # look at square with position vecotor
        row, col = 7-from_position_vector.j, from_position_vector.i
        # get piece thats moving
        piece: pieces_mod.Piece = new_pieces_matrix[row][col]
        # set from square to blank
        new_pieces_matrix[row][col] = None

        # update piece to keep track of its last move
        piece.last_move = movement_vector


        # set to square to this piece
        row, col = 7-resultant_vector.j, resultant_vector.i
        new_pieces_matrix[row][col] = piece

        # convert back to tuple
        new_pieces_matrix = tuple(map(tuple, new_pieces_matrix))

        # update next to go
        new_next_to_go = "W" if self.next_to_go == "B" else "B"

        # return new board state instance
        return Board_State(
            next_to_go=new_next_to_go,
            pieces_matrix=new_pieces_matrix
        )
```

I wrote the functions and tested them in the order of dependency.

Starting with:

- Get_piece_at_vecotor as it had not dependency to other functions

Then in order

- Generate_all_pieces
- Generate_pieces_of_color
- Color_in_check
- Generate_legal_moves
- Game_over
- Static_Evaluation

I had a significant issue in testing this module, specifically the generate legal moves function and the static evaluation function. I will show all the testing code and data and then explain the issue.

Test_board_state.py

```python
import unittest
import ddt

from board_state import Board_State
# tested so assumed correct
import pieces
from vector import Vector

def test_dir(file_name): return f"test_data/board_state/{file_name}.yaml"

# code used to deserialize
# code repeated from test pieces, opportunity to reduce redundancy
def list_map(function, iterable): return list(map(function, iterable))
def tuple_map(function, iterable): return tuple(map(function, iterable))

def descriptor_to_piece(descriptor) -> pieces.Piece:
    # converts WN to knight object with a color attribute of white
    if descriptor is None:
        return None

    color, symbol = descriptor
    piece_type: pieces.Piece = pieces.PIECE_TYPES[symbol]
    return piece_type(color=color)

def deserialize_pieces_matrix(pieces_matrix, next_to_go="W") -> Board_State:
    def row_of_symbols_to_pieces(row):
        return list_map(descriptor_to_piece, row)

    # update pieces_matrix replacing piece descriptors to piece objects
    pieces_matrix = list_map(row_of_symbols_to_pieces, pieces_matrix)
    board_state: Board_State = Board_State(pieces_matrix=pieces_matrix,
next_to_go=next_to_go)

    return board_state



@ddt.ddt
class Test_Case(unittest.TestCase):

    # this test isn't data driven,
    # it tests that the static evaluation is 0 for starting positions
    def test_static_eval_starting_positions(self):
        self.assertEqual(
            Board_State().static_evaluation(),
```

```
            0
        )

    # this test is testing that a list of all pieces and there position
vectors can be generated
    @ddt.file_data(test_dir('generate_all_pieces'))
    def test_generate_all_pieces(self, pieces_matrix, pieces_and_squares):
        pieces_and_squares = tuple_map(tuple, pieces_and_squares)

        board_state: Board_State = deserialize_pieces_matrix(pieces_matrix)
        # use set as order irrelevant
        all_pieces: set[pieces.Piece, Vector] =
set(board_state.generate_all_pieces())


        # convert square to vector, allowed as this is tested
        def deserialize(data_unit):
            # unpack test data unit
            descriptor, square = data_unit
            # return [descriptor_to_piece(descriptor),
Vector.construct_from_square(square)]
            return (descriptor_to_piece(descriptor),
Vector.construct_from_square(square))

        def serialize(data_unit):
            piece, vector = data_unit
            # return [piece.symbol(), vector.to_square()]
            return (piece.symbol(), vector.to_square())


        all_pieces_expected: set[pieces.Piece, Vector] = set(map(deserialize,
pieces_and_squares))
        # legal_moves_expected: list[pieces.Piece, Vector] = sorted(
        #     map(deserialize, pieces_and_squares),
        #     key=repr
        # )

        self.assertEqual(
            all_pieces,
            all_pieces_expected,
            msg=f"\nactual {list_map(serialize, all_pieces)}  !=  expected
{list_map(serialize, all_pieces_expected)}"
        )

    # this test is to test the function responsible for getting the piece at a
given position vector
    @ddt.file_data(test_dir('piece_at_vector'))
    def test_piece_at_vector(self, pieces_matrix, vectors_and_expected_piece):
        board_state: Board_State = deserialize_pieces_matrix(pieces_matrix)
```

```
        # could use all method and one assert but this would have been less
readable, also hard to make useful message,
        # wanting different messages implies multiple asserts should be
completed
        for vector, expected_piece in vectors_and_expected_piece:
            # deserialize vector / cast to Vector
            vector: Vector = Vector(*vector)
            # repeat for piece
            expected_piece: pieces.Piece = descriptor_to_piece(expected_piece)

            # actual
            piece: piece.Piece = board_state.get_piece_at_vector(vector)
            msg = f"Piece at vector {repr(vector)} is {repr(piece)} not
expected piece {repr(expected_piece)}"

    # this test ensures that all the pieces belonging to a specific color and
there position vectors can be identified
    @ddt.file_data(test_dir('generate_pieces_of_color'))
    def test_generate_pieces_of_color(self, pieces_matrix, color,
pieces_and_squares):
        pieces_and_squares = tuple_map(tuple, pieces_and_squares)

        board_state: Board_State = deserialize_pieces_matrix(pieces_matrix)
        # use set as order irrelevant
        legal_moves_actual: set[pieces.Piece, Vector] =
set(board_state.generate_pieces_of_color(color))

        # convert square to vector, allowed as this is tested
        def deserialize(data_unit):
            # unpack test data unit
            descriptor, square = data_unit
            # return [descriptor_to_piece(descriptor),
Vector.construct_from_square(square)]
            return (descriptor_to_piece(descriptor),
Vector.construct_from_square(square))

        def serialize(data_unit):
            piece, vector = data_unit
            # return [piece.symbol(), vector.to_square()]
            return (piece.symbol(), vector.to_square())


        legal_moves_expected: set[pieces.Piece, Vector] = set(map(deserialize,
pieces_and_squares))

        self.assertEqual(
            legal_moves_actual,
```

```
            legal_moves_expected,
            msg=f"\nactual {list_map(serialize,
legal_moves_actual)}  !=  expected {list_map(serialize,
legal_moves_expected)}"
        )

    # this test ensures that the chess engine can determine if a specified
player is currently in check
    @ddt.file_data(test_dir('color_in_check'))
    def test_color_in_check(self, pieces_matrix, white_in_check,
black_in_check):
        board_state: Board_State = deserialize_pieces_matrix(pieces_matrix)

        self.assertEqual(
            board_state.color_in_check("W"),
            white_in_check,
            msg=f"white {'should' if white_in_check else 'should not'} be in
check but {'is' if board_state.color_in_check('W') else 'is not'}"
        )
        self.assertEqual(
            board_state.color_in_check("B"),
            black_in_check,
            msg=f"black {'should' if black_in_check else 'should not'} be in
check but {'is' if board_state.color_in_check('B') else 'is not'}"
        )

    # this test ensures that a game over situation can be identified and its
nature discerned
    @ddt.file_data(test_dir("game_over"))
    def test_game_over(self, pieces_matrix, expected_over, expected_outcome,
next_to_go):
        board_state: Board_State = deserialize_pieces_matrix(pieces_matrix,
next_to_go=next_to_go)

        self.assertEqual(
            board_state.is_game_over_for_next_to_go(),
            (expected_over, expected_outcome),
            msg=f"\nactual {board_state.is_game_over_for_next_to_go()} !=
expected {(expected_over, expected_outcome)}"
        )

    # this is one of the most important functions to test
    # this function determines all the possible legal moves a player can make
with their pieces, accounting for check
    # this test checks this for many inputs
    @ddt.file_data(test_dir("generate_legal_moves"))
    def test_generate_legal_moves(self, pieces_matrix, next_to_go,
expected_legal_moves):
```

```
        board_state: Board_State =
deserialize_pieces_matrix(pieces_matrix=pieces_matrix, next_to_go=next_to_go)
        actual_legal_moves = set(board_state.generate_legal_moves())

        # convert square to vector, allowed as this is tested
        def deserialize_expected_legal_moves():
            # unpack test data unit
            for test_datum in expected_legal_moves:
                from_square, all_to_squares = test_datum
                for to_square in all_to_squares:
                    position_vector: Vector =
Vector.construct_from_square(from_square)
                    movement_vector: Vector =
Vector.construct_from_square(to_square) - position_vector

                    yield (position_vector, movement_vector)

        expected_legal_moves = set(deserialize_expected_legal_moves())
        self.assertEqual(
            actual_legal_moves,
            expected_legal_moves,
        )

if __name__ == '__main__':
    unittest.main()
```

Color_in_check.yaml

```
test1:
  pieces_matrix: [
    [BR,   BK,   BB,   BK,   BQ,   BB,   BK,   BR  ],
    [BP,   BP,   BP,   BP,   BP,   BP,   BP,   BP  ],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [WP,   WP,   WP,   WP,   WP,   WP,   WP,   WP  ],
    [WR,   WK,   WB,   WK,   WQ,   WB,   WK,   WR  ]
  ]
  white_in_check: false
  black_in_check: false
test2:
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
```

```
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, WK, WQ, BK],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],

  ]
  white_in_check: false
  black_in_check: true

test3:
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, BQ],
    [null, null, null, null, null, null, null, null],
    [null, null, null, WP, WP, null, null, null],
    [null, null, null, WB, WK, WB, null, null],
  ]
  white_in_check: true
  black_in_check: false

test4:
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, WK, null, BK],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, WR]
  ]
  white_in_check: false
  black_in_check: true
test5:
  pieces_matrix: [
    [null, null, null, BK, null, null, null, null],
    [null, null, null, WP, null, null, null, null],
    [null, null, null, WK, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
  ]
```

```
  white_in_check: false
  black_in_check: false
test6:
  pieces_matrix: [
    [null, null, BK, null, null, null, null, null],
    [null, null, null, WP, null, null, null, null],
    [null, null, null, WK, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
  ]
  white_in_check: false
  black_in_check: true
test7:
  pieces_matrix: [
    [null, null, null, null, BK, null, null, null],
    [null, null, null, WP, null, null, null, null],
    [null, null, null, WK, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
  ]
  white_in_check: false
  black_in_check: true
test8:
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, WP, BK, null, null, null],
    [null, null, null, WK, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
  ]
  white_in_check: true
  black_in_check: true
test9:
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, BK, WP, null, null, null, null],
    [null, null, null, WK, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
```

```yaml
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
  ]
  white_in_check: true
  black_in_check: true
```


Game_over.yaml


```yaml
test1:
  next_to_go: W
  pieces_matrix: [
    [BR,   BK,   BB,   BK,   BQ,   BB,   BK,   BR  ],
    [BP,   BP,   BP,   BP,   BP,   BP,   BP,   BP  ],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [WP,   WP,   WP,   WP,   WP,   WP,   WP,   WP  ],
    [WR,   WK,   WB,   WK,   WQ,   WB,   WK,   WR  ]
  ]
  expected_over: false
  expected_outcome: null
test2:
  next_to_go: B
  pieces_matrix: [
    [null, null, null, BK,   null, null, null, null],
    [null, null, null, WP,   null, null, null, null],
    [null, null, null, WK,   null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
  ]
  expected_over: true
  expected_outcome: null
test3:
  next_to_go: B
  pieces_matrix: [
    [WR,   null, null, null, null, null, BK,   null],
    [null, WR,   null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
```

```yaml
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, WK  ],
  ]
  expected_over: true
  expected_outcome: W
test4:
  next_to_go: B
  pieces_matrix: [
    [null, null, null, BK,   null, null, null, null],
    [null, null, null, WP,   null, null, null, null],
    [null, null, WP,   WK,   null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
  ]
  expected_over: true
  expected_outcome: null
test5:
  next_to_go: B
  pieces_matrix: [
    [null, null, null, BK,   null, null, null, null],
    [null, null, WP,   WP,   null, null, null, null],
    [null, null, null, WK,   null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
  ]
  expected_over: true
  expected_outcome: W
test6:
  next_to_go: W
  pieces_matrix: [
    [BR,   null, null, null, null, BR,   BK,   null],
    [BP,   null, null, null, null, BP,   BP,   null],
    [null, BP,   null, null, null, null, null, null],
    [null, null, BQ,   null, null, null, null, null],
    [null, null, null, null, null, null, null, WQ  ],
    [null, null, null, null, null, null, WP,   null],
    [WP,   null, null, null, null, WP,   WB,   null],
    [WR,   null, null, null, null, null, null, WR  ],
  ]
  expected_over: false
```

```
  expected_outcome: null
test7:
  next_to_go: B
  pieces_matrix: [
    [BR,   null, null, null, null, BR,   BK,   WQ  ],
    [BP,   null, null, null, null, BP,   BP,   null],
    [null, BP,   null, null, null, null, null, null],
    [null, null, BQ,   null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, WP,   null],
    [WP,   null, null, null, null, WP,   WB,   null],
    [WR,   null, null, null, null, null, null, WR  ],
  ]
  expected_over: true
  expected_outcome: W
```

generate_all_pieces.yaml

```
test1:
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [BP,   null, BP,   null, null, null, null, null],
    [null, WP,   null, null, null, null, null, null]
  ]
  pieces_and_squares: [
    [WP, B1],
    [BP, A2],
    [BP, C2]
  ]

test2:
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, BP,   null, null, null, null],
    [null, null, null, null, BR,   null, null, null],
    [null, BR,   null, BR,   null, null, WB,   null],
    [null, null, BP,   null, null, null, BP,   null],
    [null, WR,   null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null]
  ]
  pieces_and_squares: [
    [WR, B2],
```

```
    [BR, B4],
    [BP, C3],
    [BR, D4],
    [BP, D6],
    [BP, G3],
    [WB, G4],
    [BR, E5]
  ]
```

Generate_legal_moves.yaml

```yaml
test1:
  next_to_go: B
  pieces_matrix: [
    [null, null, null, BK, null, null, null, null],
    [null, null, null, WP, null, null, null, null],
    [null, null, null, WK, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
  ]
  expected_legal_moves: []
test2:
  next_to_go: W
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [BP,   null, BP,   null, null, null, null, null],
    [null, WP,   null, null, null, null, null, null]
  ]

  expected_legal_moves: [
    [B1, [B2, B3, A2, C2]]
  ]
test3:
  next_to_go: B
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
```

```
    [null, null, null, null, null, null, null, null],
    [BP,   null, BP,   null, null, null, null, null],
    [null, WP,   null, null, null, null, null, null]
  ]
  expected_legal_moves: [
    [A2, [A1, B1]],
    [C2, [C1, B1]],
  ]
test4:
  next_to_go: W
  pieces_matrix: [
    [BR,   BN,   BB,   BK,   BQ,   BB,   BN,   BR  ],
    [BP,   BP,   BP,   BP,   BP,   BP,   BP,   BP  ],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [WP,   WP,   WP,   WP,   WP,   WP,   WP,   WP  ],
    [WR,   WN,   WB,   WK,   WQ,   WB,   WN,   WR  ]
  ]
  expected_legal_moves: [
    [A2, [A3, A4]],
    [B2, [B3, B4]],
    [C2, [C3, C4]],
    [D2, [D3, D4]],
    [E2, [E3, E4]],
    [F2, [F3, F4]],
    [G2, [G3, G4]],
    [H2, [H3, H4]],
    [B1, [A3, C3]],
    [G1, [F3, H3]]
  ]
test5:
  next_to_go: W
  pieces_matrix: [
    [BK, BP, null, null, null, null, WP, WK],
    [BP, BP, null, null, null, null, WP, WP],
    [null, null, null, BP,   null, null, null, null],
    [null, null, null, null, BR,   null, null, null],
    [null, BR,   null, BR,   null, null, WB,   null],
    [null, null, BP,   null, null, null, BP,   null],
    [null, WN,   null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null]
  ]
  expected_legal_moves: [
    [B2, [A4, C4, D1, D3]],
    [G4, [
      H3, F5, E6, D7, C8,
```

```
      H5, F3, E2, D1
    ]],
  ]
```

Generate_pieces_of_color.yaml

```yaml
test1:
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [BP,   null, BP,   null, null, null, null, null],
    [null, WP,   null, null, null, null, null, null]
  ]
  color: W
  pieces_and_squares: [
    [WP, B1]
  ]
test2:
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [BP,   null, BP,   null, null, null, null, null],
    [null, WP,   null, null, null, null, null, null]
  ]
  color: B
  pieces_and_squares: [
    [BP, A2],
    [BP, C2]
  ]

test3:
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, BP,   null, null, null, null],
    [null, null, null, null, BR,   null, null, null],
    [null, BR,   null, BR,   null, null, WB,   null],
    [null, null, BP,   null, null, null, BP,   null],
    [null, WR,   null, null, null, null, null, null],
```

```
    [null, null, null, null, null, null, null, null]
  ]
  color: W
  pieces_and_squares: [
    [WR, B2],
    [WB, G4]
  ]
test4:
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, BP,   null, null, null, null],
    [null, null, null, null, BR,   null, null, null],
    [null, BR,   null, BR,   null, null, WB,   null],
    [null, null, BP,   null, null, null, BP,   null],
    [null, WR,   null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null]
  ]
  color: B
  pieces_and_squares: [
    [BR, B4],
    [BP, C3],
    [BR, D4],
    [BP, D6],
    [BP, G3],
    [BR, E5]
  ]
```

Piece_at_vector.yaml

```
test1:
  pieces_matrix: [
    [BR,   BN,   BB,   BK,   BQ,   BB,   BN,   BR  ],
    [BP,   BP,   BP,   BP,   BP,   BP,   BP,   BP  ],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [WP,   WP,   WP,   WP,   WP,   WP,   WP,   WP  ],
    [WR,   WN,   WB,   WK,   WQ,   WB,   WN,   WR  ]
  ]
  vectors_and_expected_piece: [
    [[0, 0], WR],
    [[1, 1], WP],
    [[2, 4], null],
    [[7, 3], null],
    [[5, 7], BB],
```

```
    [[2, 6], BP]
  ]

test2:
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, BP,   null, null, null, null],
    [null, null, null, null, BR,   null, null, null],
    [null, BR,   null, BR,   null, null, WB,   null],
    [null, null, BP,   null, null, null, BP,   null],
    [null, WR,   null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null]
  ]
  vectors_and_expected_piece: [
    [[0, 0], null],
    [[1, 1], WR],
    [[1, 3], BR],
    [[0, 7], null],
    [[7, 0], null],
    [[7, 7], null],
    [[3, 3], BR],
    [[3, 5], BP],
    [[6, 3], WB],
    [[6, 1], null]
  ]
```

The tests now work fine:

```
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4> python -m unittest test_board_state
..............................
----------------------------------------------------------------------
Ran 30 tests in 0.277s

OK
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4>
```

But I was extremely stumped as I was testing my legal moves generator function when I got this issue:

```
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4> python -m unittest test_board_state
F....
======================================================================
FAIL: test_generate_legal_moves_1_test1 (test_board_state.Test_Case)
test_generate_legal_moves_1_test1
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\henry\AppData\Local\Programs\Python\Python310\lib\site-packages\ddt.py", line 220, in wrapper
    return func(self, *args, **kwargs)
  File "C:\Users\henry\Documents\computing coursework\prototype 2\v2.4\test_board_state.py", line 177, in test_generate_legal_moves
    self.assertEqual(
AssertionError: Items in the first set but not the second:
(Vector(i=3, j=7), Vector(i=1, j=0))
(Vector(i=3, j=7), Vector(i=1, j=-1))
(Vector(i=3, j=7), Vector(i=0, j=-1))
(Vector(i=3, j=7), Vector(i=-1, j=-1))
(Vector(i=3, j=7), Vector(i=-1, j=0))

----------------------------------------------------------------------
Ran 5 tests in 0.048s

FAILED (failures=1)
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4>
```

(other tests disables to show only this issue)

The test that had failed was (**generate_legal_moves.yaml**)

```yaml
test1:
  next_to_go: B
  pieces_matrix: [
    [null, null, null, BK, null, null, null, null],
    [null, null, null, WP, null, null, null, null],
    [null, null, null, WK, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
  ]
  expected_legal_moves: []
```

There are supposed to be no legal moves as black in in checkmate. Instead the restriction of check was ignored and my function said that the black king could move to all 5 adjacent squares. I added this test to the check tests:

(**color_in_check.yaml**)

```yaml
test5:
  pieces_matrix: [
    [null, null, null, BK, null, null, null, null],
    [null, null, null, WP, null, null, null, null],
    [null, null, null, WK, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
```

```
    [null, null, null, null, null, null, null, null],
  ]
  white_in_check: false
  black_in_check: false
test6:
  pieces_matrix: [
    [null, null, BK, null, null, null, null, null],
    [null, null, null, WP, null, null, null, null],
    [null, null, null, WK, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
  ]
  white_in_check: false
  black_in_check: true
test7:
  pieces_matrix: [
    [null, null, null, null, BK, null, null, null],
    [null, null, null, WP, null, null, null, null],
    [null, null, null, WK, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
  ]
  white_in_check: false
  black_in_check: true
test8:
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, null, WP, BK, null, null, null],
    [null, null, null, WK, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
  ]
  white_in_check: true
  black_in_check: true
test9:
  pieces_matrix: [
    [null, null, null, null, null, null, null, null],
    [null, null, BK, WP, null, null, null, null],
    [null, null, null, WK, null, null, null, null],
```

```
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
    [null, null, null, null, null, null, null, null],
  ]
  white_in_check: true
  black_in_check: true
```

All these tests passed which really confused me. The check function correctly determined that the child game states resulting from each of these moves were check for black.

To further analyse this I created a jupyter notebook. I tried to replicate the exact function call being made by the test with a version of the legal moves generator that included a lot of print statements to reveal the inner workings.

I was able to recreate the contradictory outcomes

Cell 1:

```python
from copy import deepcopy
from dataclasses import dataclass
from itertools import product as iter_product
from pprint import pprint


import pieces as pieces_mod
from assorted import ARBITRARILY_LARGE_VALUE, dev_print
from vector import Vector
```

✓ 0.7s Python

Cell 2 (copied as too big to screenshot)

```python
STARTING_POSITIONS: tuple[tuple[pieces_mod.Piece]] = (
    (
        pieces_mod.Rook(color="B"),
        pieces_mod.Knight(color="B"),
        pieces_mod.Bishop(color="B"),
        pieces_mod.Queen(color="B"),
        pieces_mod.King(color="B"),
        pieces_mod.Bishop(color="B"),
        pieces_mod.Knight(color="B"),
        pieces_mod.Rook(color="B")
    ),
    (pieces_mod.Pawn(color="B"),)*8,
    (None,)*8,
    (None,)*8,
```

```python
    (None,)*8,
    (None,)*8,
    (pieces_mod.Pawn(color="W"),)*8,
    (
        pieces_mod.Rook(color="W"),
        pieces_mod.Knight(color="W"),
        pieces_mod.Bishop(color="W"),
        pieces_mod.Queen(color="W"),
        pieces_mod.King(color="W"),
        pieces_mod.Bishop(color="W"),
        pieces_mod.Knight(color="W"),
        pieces_mod.Rook(color="W")
    )
)

@dataclass(frozen=True)
class Board_State():
    pieces_matrix: tuple[tuple[pieces_mod.Piece]]
    next_to_go: int = "W"
    # pieces_matrix: tuple[tuple[pieces_mod.Piece]] = STARTING_POSITIONS

    def print_board(self, print_function=pprint):
        print_function(
            list(map(
                lambda row: list(map(
                    lambda piece: None if piece is None else piece.symbol(),
                    row
                )),
                self.pieces_matrix
            ))
        )

    def get_piece_at_vector(self, vector: Vector):
        column, row = vector.i, 7-vector.j
        # column, row = vector.i, vector.j
        return self.pieces_matrix[row][column]

    def generate_all_pieces(self):
        # for i, j in zip(range(8), range(8)):
        # should be product
        for i, j in iter_product(range(8), range(8)):
            position_vector = Vector(i,j)
            piece = self.get_piece_at_vector(position_vector)
            # skip if none
            if piece:
                yield piece, position_vector
            # if piece:
            #     dev_print(f"not skipping {position_vector}")
```

```
            #     yield piece, position_vector
            # else:
            #     dev_print(f"skipping {position_vector}")

    def generate_pieces_of_color(self, color=None):
        if color is None:
            color = self.next_to_go

        def same_color(data_item):
            piece, _ = data_item
            return piece.color == color

        yield from filter(
            same_color,
            self.generate_all_pieces()
        )

    def color_in_check(self, color=None):
        if color is None:
            color = self.next_to_go

        # it is now A's turn
        color_a = color
        color_b = "W" if color == "B" else "B"

        # we will examine all the movement vectors of B's pieces
        # if any of them could take the A's King then currently A is in check
as their king is threatened by 1 or more pieces (which could take it next
turn)
        for piece, position_v in self.generate_pieces_of_color(color=color_b):
            movement_vs = piece.generate_movement_vectors(
                pieces_matrix=self.pieces_matrix,
                position_vector=position_v
            )
            for movement_v in movement_vs:
                resultant = position_v + movement_v
                to_square = self.get_piece_at_vector(resultant)
                # print(f"to_square   -->   {to_square!r}")
                # dev_print(f"piece at square {resultant.to_square()} is
{to_square.symbol() if to_square else '<empty>'}")
                # dev_print(f"piece at square {resultant.to_square()} is
{repr(to_square) if to_square else '<empty>'}")
                # As_move_threatens_king_A = isinstance(to_square,
pieces_mod.King) and to_square.color == color_a
                As_move_threatens_king_A = (to_square ==
pieces_mod.King(color=color_a))
```

```
                # print(f"to_square == pieces_mod.King(color='B')   --
>   {to_square!r} == {pieces_mod.King(color='B')!r}   -->    {to_square ==
pieces_mod.King(color='B')}")
                # dev_print(f"therefore king {'IS' if As_move_threatens_king_A
else 'NOT'} threatened as square {'DOES' if As_move_threatens_king_A else
'DOES NOT'} contain {pieces_mod.King(color=color_a)!r} instead containing
{repr(to_square) if to_square else '<empty>'}")
                # if As_move_threatens_king_A break out of all 3 loops
                if As_move_threatens_king_A:
                    # dev_print(f"color_in_check(color='{color}') returning
True")
                    # dev_print(f"piece {piece.symbol()} at
{position_v.to_square()} moving to {resultant.to_square()} IS threatening
king")
                    return True
                # else:
                    # dev_print(f"piece {piece.symbol()} at
{position_v.to_square()} moving to {resultant.to_square()} NOT threatening
king")
        # dev_print(f"color_in_check(color='{color}') returning False")
        return False


    def generate_legal_moves(self):
        # dev_print(f"analysing legal moves for next to go {self.next_to_go}")
        for piece, piece_position_vector in
self.generate_pieces_of_color(color=self.next_to_go):
            # dev_print(f"analysing moves for piece: {piece.symbol()}")
            movement_vectors = piece.generate_movement_vectors(
                pieces_matrix=self.pieces_matrix,
                position_vector=piece_position_vector
            )
            for movement_vector in movement_vectors:
                # dev_print(f"\tpiece {piece.symbol()}:_analysing movement
vector {repr(movement_vector)}")
                child_game_state: Board_State =
self.make_move(from_position_vector=piece_position_vector,
movement_vector=movement_vector)
                # dev_print(f"\t\tmovement vector {repr(movement_vector)}:
results in child game state")
                # child_game_state.print_board(
                #     print_function=lambda rows: list(map(
                #         lambda item: print(f"\t\t{item}"),
                #         rows
                #     ))
                # )
                # dev_print(repr(child_game_state))
```

```
                is_check_after_move =
child_game_state.color_in_check(color=self.next_to_go)
                dev_print(f"\t\tThis game state in check? for
{self.next_to_go}:  {is_check_after_move}")

                if not is_check_after_move:
                    yield piece_position_vector, movement_vector

    # def generate_legal_moves(self):
    #     for piece, piece_position_vector in
self.generate_pieces_of_color(color=self.next_to_go):
    #         movement_vectors = piece.generate_movement_vectors(
    #             pieces_matrix=self.pieces_matrix,
    #             position_vector=piece_position_vector
    #         )
    #         for movement_vector in movement_vectors:
    #             child_game_state =
self.make_move(from_position_vector=piece_position_vector,
movement_vector=movement_vector)
    #             is_check_after_move =
child_game_state.color_in_check(color=self.next_to_go)
    #             if not is_check_after_move:
    #                 yield piece_position_vector, movement_vector

    def is_game_over_for_next_to_go(self):
        # check if in checkmate
        # for player a
        # if b has no moves
        if not list(self.generate_legal_moves()):
            # if b in check
            if self.color_in_check():
                # checkmate for b, a wins
                return True, self.next_to_go
            else:
                # stalemate
                return True, None

        return False, None

    def static_evaluation(self):
        """Give positive static evaluation if white is winning"""
        def generate_all_pieces():
            for i, j in iter_product(range(8), range(8)):
                piece_position_vector = Vector(i, j)
                piece: pieces_mod.Piece =
self.get_piece_at_vector(piece_position_vector)
                if not piece:
                    continue
```

```
                yield piece, piece_position_vector

        over, winner = self.is_game_over_for_next_to_go()
        if over:
            match winner:
                case None: multiplier = 0
                case "W": multiplier = 1
                case "B": multiplier = -1
            return winner * ARBITRARILY_LARGE_VALUE

        else:
            return sum(piece.get_value(position_vector) * multiplier for
piece, position_vector in generate_all_pieces())

    def make_move(self, from_position_vector: Vector, movement_vector:
Vector):
        to_position_vector = from_position_vector + movement_vector
        # poor code, this below line can cause infinite recursion when legal
moves generator called post check changes
        # assert (from_position_vector, movement_vector) in
self.generate_legal_moves()

        new_pieces_matrix = deepcopy(self.pieces_matrix)
        # convert to list
        new_pieces_matrix = list(map(list, new_pieces_matrix))

        # set from to blank
        row, col = 7-from_position_vector.j, from_position_vector.i
        piece: pieces_mod.Piece = new_pieces_matrix[row][col]

        piece.last_move = movement_vector

        new_pieces_matrix[row][col] = None

        # set to square to this piece
        row, col = 7-to_position_vector.j, to_position_vector.i
        new_pieces_matrix[row][col] = piece

        # convert back to tuple
        new_pieces_matrix = tuple(map(tuple, new_pieces_matrix))
        new_next_to_go = "W" if self.next_to_go == "B" else "B"

        return Board_State(
            next_to_go=new_next_to_go,
            pieces_matrix=new_pieces_matrix
        )
```

Cell 3:

```
# tested so assumed correct
import pieces
from vector import Vector


def test_dir(file_name): return f"test_data/board_state/{file_name}.yaml"

# code repeated from test pieces, opportunity to reduce redundancy


def list_map(function, iterable): return list(map(function, iterable))
def tuple_map(function, iterable): return tuple(map(function, iterable))


def descriptor_to_piece(descriptor) -> pieces.Piece:
    # converts WN to knight object with a color attribute of white
    if descriptor is None:
        return None

    color, symbol = descriptor
    piece_type: pieces.Piece = pieces.PIECE_TYPES[symbol]
    return piece_type(color=color)


def deserialize_pieces_matrix(pieces_matrix, next_to_go="W") -> Board_State:
    def row_of_symbols_to_pieces(row):
        return list_map(descriptor_to_piece, row)

    # update pieces_matrix replacing piece descriptors to piece objects
    pieces_matrix = list_map(row_of_symbols_to_pieces, pieces_matrix)
    board_state: Board_State = Board_State(pieces_matrix=pieces_matrix,
next_to_go=next_to_go)

    return board_state
```

cell 4 though 6

```python
next_to_go = "B"
pieces_matrix = [
    [None, None, None, "BK", None, None, None, None],
    [None, None, None, "WP", None, None, None, None],
    [None, None, None, "WK", None, None, None, None],
    [None, None, None, None, None, None, None, None],
    [None, None, None, None, None, None, None, None],
    [None, None, None, None, None, None, None, None],
    [None, None, None, None, None, None, None, None],
    [None, None, None, None, None, None, None, None]
]
expected_legal_moves = []
```

[4] ✓ 0.6s Python

```python
board_state: Board_State = deserialize_pieces_matrix(pieces_matrix=pieces_matrix, next_to_go=next_to_go)
```

[5] ✓ 0.4s Python

```python
board_state.print_board()
```

[6] ✓ 0.6s Python

```
[[None, None, None, 'BK', None, None, None, None],
 [None, None, None, 'WP', None, None, None, None],
 [None, None, None, 'WK', None, None, None, None],
 [None, None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None, None],
 [None, None, None, None, None, None, None, None]]
```

cell 7-8

```python
set(board_state.generate_legal_moves())
```

[7] 0.6s Python

```
This game state in check? for B:  False
This game state in check? for B:  False
This game state in check? for B:  False
This game state in check? for B:  False
This game state in check? for B:  False

{(Vector(i=3, j=7), Vector(i=-1, j=-1)),
 (Vector(i=3, j=7), Vector(i=-1, j=0)),
 (Vector(i=3, j=7), Vector(i=0, j=-1)),
 (Vector(i=3, j=7), Vector(i=1, j=-1)),
 (Vector(i=3, j=7), Vector(i=1, j=0))}
```

```python
board_state = Board_State(
    next_to_go="W",
    pieces_matrix= [
        [None, None, pieces.King(color="B"), None, None, None, None, None],
        [None, None, None, pieces.Pawn(color="W"), None, None, None, None],
        [None, None, None, pieces.King(color="W"), None, None, None, None],
        [None, None, None, None, None, None, None, None],
        [None, None, None, None, None, None, None, None],
        [None, None, None, None, None, None, None, None],
        [None, None, None, None, None, None, None, None],
        [None, None, None, None, None, None, None, None]
    ]
)
```

[8] 0.7s Python

Cell 9-11

```python
board_state.color_in_check("W")
```

[9] 0.5s Python

```
False
```

```python
board_state.color_in_check("B")
```

[10] 0.6s Python

```
True
```

```python
pieces.King("B") == pieces.King("B")
```

[11] 0.5s Python

```
True
```

This highlights the issue and apparent contradiction in logic. My chess program is checking each child game state and reporting that black isn't in check, yielding all 5 moves. Then for one of the child game states the color in check functions seen to directly contradict this by correctly identifying that black is in check.

I later realised the issue and why all my unit tests for the check function had worked and yet a practical test vie the legal moves generator function failed.

The line that was the issue what part of the Piece.__eq__ method

```
105     # equality operator
106     def __eq__(self, other):
107         try:
108             # assert same subclass like rook
109             assert isinstance(other, type(self))
110             assert self.color == other.color
111 
112             # i am not checking that pieces had the same last move as I want to compare kings without for the check function
113             # assert self.last_move == other.last_moves
114 
115             # value and value_matrix should never be changes
116         except AssertionError:
117             return False
118         else:
119             return True
```

Line 113 was the issue. Shown commented out.

This line meant that a king that had been moved was not equal to a king that had just been initialised for the purpose of a test. This is because they would have different last moves attributes. The comparison in the check function is to a king of the same color but with last_moves set to None. This worked for the unit tests as the kings in the tests also had not moved. Once this line was removed, all the tests passed.

This was an issue with the pieces testing ad my assumption that the module was robust, but it was fixed.

Next I moved onto a module that would contain a game class that was able to keep track of the whole game.

It should enable user moves to be implemented and computer moves to be generated and implemented (I will get to minimax).

Here was my code:

Game.py

```
# import other modules
from board_state import Board_State
from minimax import minimax
from vector import Vector
from assorted import ARBITRARILY_LARGE_VALUE, dev_print, NotUserTurn,
InvalidMove

# the game class is used to keep track of a chess game between a user and the
computer
class Game(object):
```

```
    # it keeps track of:
    # the player's color's
    player_color_key: dict
    # the difficulty or depth of the game
    depth: int
    # the current board state
    board_state: Board_State
    # the number of moves so far
    move_counter: int
    # a table of game state hashes and there frequency
    piece_matrix_occurrence_hash_table: list

    # this adds a new game state to the frequency table
    def add_new_piece_matrix_to_hash_table(self, piece_matrix):
        # the pieces matrix is hashed
        matrix_hash = hash(piece_matrix)

        # if this pieces matrix has been encountered before, add 1 to the
frequency
        if matrix_hash in self.piece_matrix_occurrence_hash_table.keys():
            self.piece_matrix_occurrence_hash_table[matrix_hash] += 1
        # else set frequency to 1
        else:
            self.piece_matrix_occurrence_hash_table[matrix_hash] = 1

    # determines if there is a 3 repeat stalemate
    def is_3_board_repeats_in_game_history(self):
        # if any of the board states have been repeated 3 or more times:
stalemate
        return any(value >= 3 for value in
self.piece_matrix_occurrence_hash_table.values())

    # constructor for game object
    def __init__(self, depth=2, user_color="W") -> None:
        # based on user's color, determine color key
        self.player_color_key = {
            "W": 1 if user_color=="W" else -1,
            "B": -1 if user_color=="W" else 1
        }
        # set depth property from parameters
        self.depth = depth

        # set attributes for game at start
        self.board_state = Board_State()
        self.move_counter = 0
        self.piece_matrix_occurrence_hash_table = {}
```

```python
    # this function validates if the user's move is allowed and if so, makes
it
    def implement_user_move(self, from_square, to_square) -> None:
        # check that the user is allowed to move
        which_player_next_to_go = self.player_color_key.get(
            self.board_state.next_to_go
        )
        if which_player_next_to_go != 1:
            raise NotUserTurn(game=self)

        # unpack move into vector form
        # invalid square syntaxes will cause a value error here
        try:
            position_vector = Vector.construct_from_square(from_square)
            movement_vector = Vector.construct_from_square(to_square) -
position_vector
        except Exception:
            raise ValueError("Square's not in valid format")

        # if the move is not in the set of legal moves, raise and appropriate
exception
        if (position_vector, movement_vector) not in
self.board_state.generate_legal_moves():
            raise InvalidMove(game=self)

        # implement move
        self.board_state =
self.board_state.make_move(from_position_vector=position_vector,
movement_vector=movement_vector)
        # adjust other properties that keep track of the game
        self.move_counter += 1
        self.add_new_piece_matrix_to_hash_table(self.board_state.pieces_matrix
)

        return (position_vector, movement_vector),
self.board_state.static_evaluation()

    # this function determines if the game is over and if so, what is the
nature of the outcome
    def check_game_over(self):
        # returns: over: bool, winning_player: (1/-1), classification: str
        # check for 3 repeat stalemate
        if self.is_3_board_repeats_in_game_history():
            return True, None, "Stalemate"

        # determine if board state is over for next player
        over, winner = self.board_state.is_game_over_for_next_to_go()
```

```
        # switch case statement to determine the appropriate values to be
returned in each case
        match (over, winner):
            case False, _:
                victory_classification = None
                winning_player = None
            case True, None:
                victory_classification = "Stalemate"
                winning_player = None
            case True, winner:
                victory_classification = "Checkmate"
                winning_player = self.player_color_key[winner]
        # return appropriate values
        return over, winning_player, victory_classification

    # this function determines and implements the computer move
    def implement_computer_move(self, best_move_function=None):
        # for use with testing bots, a best move function can be provided for
use, but minimax if the default

        # get next to go player (1/-1)
        which_player_next_to_go = self.player_color_key.get(
            self.board_state.next_to_go
        )
        # check that is it the computer's turn
        if which_player_next_to_go != -1:
            raise ValueError(f"Next to go is user:
{self.board_state.next_to_go} not computer")

        # if no function provided, default to minimax
        if best_move_function is None:
            score, best_child, best_move = minimax(
                board_state = self.board_state,
                # assume white / user always maximizer
                # is_maximizer = (self.board_state.next_to_go == "W"),
                is_maximizer = False,
                # depth is based of difficulty of game based on depth
parameter
                depth = self.depth,
                # default values for alpha and beta
                alpha = (-1)*ARBITRARILY_LARGE_VALUE,
                beta = ARBITRARILY_LARGE_VALUE
            )
        else:
            # otherwise use provided function,
            # the provided function should take game as an argument and then
return data in the same format as the minimax function
            score, best_child, best_move = best_move_function(self)
```

```
        # adjust properties that keep track of the game state
        self.board_state = best_child
        self.move_counter += 1
        self.add_new_piece_matrix_to_hash_table(self.board_state.pieces_matrix
)

        # incase is it wanted for a print out ect, return move and score
        return best_move, score
```

in tandem with this I created a minimax function which I will go into more detail about shortly.

Rather than make a test for this I created a rudementry console chess game to play chess using this library.

```
from game import Game
from assorted import InvalidMove


# create new chess game
# difficulty set to depth 2
game = Game(depth=2)

# this function informs the user of the details when the game is over
def handle_game_over(winner, classification):
    print(f"The game is over, the {'user' if winner==1 else 'computer'} has
won in a {classification}")

# print out the starting board
game.board_state.print_board()
print()

# keep game going until loop broken
while True:
    # user goes first
    print("Your go USER:")

    # while loop and error checking used to ensure move input
    while True:
        try:
            print("Please enter move in 2 parts")
            from_square = input("From square:  ")
            to_square = input("To square:   ")
            # check
            game.implement_user_move(from_square, to_square)
        except InvalidMove:
            print("This isn't a legal move, try again")
```

```
        except ValueError:
            print("This isn't valid input, try again")
        else:
            # if it worked break out of the loop
            break

    # if move results in check then output this
    if game.board_state.color_in_check():
            print("CHECK!")
    # print out the current board_state
    game.board_state.print_board()
    print()

    # check if game over after user's move
    over, winner, classification = game.check_game_over()
    # if over, handle it.
    if over:
        handle_game_over(winner=winner, classification=classification)
        break

    # alternate, it is now the computers go
    print("Computer's go: ")

    # get the computers move
    move, _ = game.implement_computer_move()

    # print out the board again
    game.board_state.print_board()

    # print out the computer's move in terms of squares
    position_vector, movement_vector = move
    resultant_vector = position_vector + movement_vector
    piece_symbol =
game.board_state.get_piece_at_vector(resultant_vector).symbol()
    print(f"Computer Moved  {piece_symbol}:  {position_vector.to_square()} to
{resultant_vector.to_square()}")

    # print check if applicable
    if game.board_state.color_in_check():
            print("CHECK!")

    # check if the game is over, if so handle it
    over, winner, classification = game.check_game_over()
    if over:
        handle_game_over(winner=winner, classification=classification)
        break
    # create a new line to separate for the user's next move
    print()
```

This allowed the user to play check against the computer. The game wasn't configurable as the user was white against depth 2 minimax but it was playable.

Here is an example of it running:

```
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4> python
console_chess.py
8| BR  BN  BB  BQ  BK  BB  BN  BR
7| BP  BP  BP  BP  BP  BP  BP  BP
6| ·   ·   ·   ·   ·   ·   ·   ·
5| ·   ·   ·   ·   ·   ·   ·   ·
4| ·   ·   ·   ·   ·   ·   ·   ·
3| ·   ·   ·   ·   ·   ·   ·   ·
2| WP  WP  WP  WP  WP  WP  WP  WP
1| WR  WN  WB  WQ  WK  WB  WN  WR
  (A   B   C   D   E   F   G   H )

Your go USER:
Please enter move in 2 parts
From square:  D2
To square:   D4
8| BR  BN  BB  BQ  BK  BB  BN  BR
7| BP  BP  BP  BP  BP  BP  BP  BP
6| ·   ·   ·   ·   ·   ·   ·   ·
5| ·   ·   ·   ·   ·   ·   ·   ·
4| ·   ·   ·   WP  ·   ·   ·   ·
3| ·   ·   ·   ·   ·   ·   ·   ·
2| WP  WP  WP  ·   WP  WP  WP  WP
1| WR  WN  WB  WQ  WK  WB  WN  WR
  (A   B   C   D   E   F   G   H )

Computer's go:
8| BR  ·   BB  BQ  BK  BB  BN  BR
7| BP  BP  BP  BP  BP  BP  BP  BP
6| ·   ·   BN  ·   ·   ·   ·   ·
5| ·   ·   ·   ·   ·   ·   ·   ·
4| ·   ·   ·   WP  ·   ·   ·   ·
3| ·   ·   ·   ·   ·   ·   ·   ·
2| WP  WP  WP  ·   WP  WP  WP  WP
1| WR  WN  WB  WQ  WK  WB  WN  WR
  (A   B   C   D   E   F   G   H )
Computer Moved  BN:  B8 to C6

Your go USER:
Please enter move in 2 parts
From square:  rubbish that is invalid input
To square:   more rubbish
```

```
This isn't valid input, try again
Please enter move in 2 parts
From square:  a0
To square:   i9
This isn't a legal move, try again
Please enter move in 2 parts
From square:  e1
To square:   e2
This isn't a legal move, try again
Please enter move in 2 parts
From square:  E2
To square:   E3
8| BR  ·   BB  BQ  BK  BB  BN  BR
7| BP  BP  BP  BP  BP  BP  BP  BP
6| ·   ·   BN  ·   ·   ·   ·   ·
5| ·   ·   ·   ·   ·   ·   ·   ·
4| ·   ·   ·   WP  ·   ·   ·   ·
3| ·   ·   ·   ·   WP  ·   ·   ·
2| WP  WP  WP  ·   ·   WP  WP  WP
1| WR  WN  WB  WQ  WK  WB  WN  WR
  (A   B   C   D   E   F   G   H )

Computer's go:
8| BR  ·   BB  BQ  BK  BB  ·   BR
7| BP  BP  BP  BP  BP  BP  BP  BP
6| ·   ·   BN  ·   ·   BN  ·   ·
5| ·   ·   ·   ·   ·   ·   ·   ·
4| ·   ·   ·   WP  ·   ·   ·   ·
3| ·   ·   ·   ·   WP  ·   ·   ·
2| WP  WP  WP  ·   ·   WP  WP  WP
1| WR  WN  WB  WQ  WK  WB  WN  WR
  (A   B   C   D   E   F   G   H )
Computer Moved  BN:  G8 to F6

Your go USER:
Please enter move in 2 parts
From square:  F2
To square:   F3
8| BR  ·   BB  BQ  BK  BB  ·   BR
7| BP  BP  BP  BP  BP  BP  BP  BP
6| ·   ·   BN  ·   ·   BN  ·   ·
5| ·   ·   ·   ·   ·   ·   ·   ·
4| ·   ·   ·   WP  ·   ·   ·   ·
3| ·   ·   ·   ·   WP  WP  ·   ·
2| WP  WP  WP  ·   ·   ·   WP  WP
1| WR  WN  WB  WQ  WK  WB  WN  WR
  (A   B   C   D   E   F   G   H )
```

```
Computer's go:
8| BR  ·   BB  BQ  BK  BB  ·   BR
7| BP  BP  BP  ·   BP  BP  BP  BP
6| ·   ·   BN  ·   ·   BN  ·   ·
5| ·   ·   ·   BP  ·   ·   ·   ·
4| ·   ·   ·   WP  ·   ·   ·   ·
3| ·   ·   ·   ·   WP  WP  ·   ·
2| WP  WP  WP  ·   ·   ·   WP  WP
1| WR  WN  WB  WQ  WK  WB  WN  WR
  (A   B   C   D   E   F   G   H )
Computer Moved  BP:  D7 to D5

Your go USER:
Please enter move in 2 parts
From square:  C2
To square:   C3
8| BR  ·   BB  BQ  BK  BB  ·   BR
7| BP  BP  BP  ·   BP  BP  BP  BP
6| ·   ·   BN  ·   ·   BN  ·   ·
5| ·   ·   ·   BP  ·   ·   ·   ·
4| ·   ·   ·   WP  ·   ·   ·   ·
3| ·   ·   WP  ·   WP  WP  ·   ·
2| WP  WP  ·   ·   ·   ·   WP  WP
1| WR  WN  WB  WQ  WK  WB  WN  WR
  (A   B   C   D   E   F   G   H )

Computer's go:
8| BR  ·   ·   BQ  BK  BB  ·   BR
7| BP  BP  BP  ·   BP  BP  BP  BP
6| ·   ·   BN  ·   BB  BN  ·   ·
5| ·   ·   ·   BP  ·   ·   ·   ·
4| ·   ·   ·   WP  ·   ·   ·   ·
3| ·   ·   WP  ·   WP  WP  ·   ·
2| WP  WP  ·   ·   ·   ·   WP  WP
1| WR  WN  WB  WQ  WK  WB  WN  WR
  (A   B   C   D   E   F   G   H )
Computer Moved  BB:  C8 to E6

Your go USER:
Please enter move in 2 parts
From square:  D1
To square:   D4
This isn't a legal move, try again
Please enter move in 2 parts
From square:  D1
To square:   A4
8| BR  ·   ·   BQ  BK  BB  ·   BR
7| BP  BP  BP  ·   BP  BP  BP  BP
```

```
6| ·   ·   BN  ·   BB  BN  ·   ·
5| ·   ·   ·   BP  ·   ·   ·   ·
4| WQ  ·   ·   WP  ·   ·   ·   ·
3| ·   ·   WP  ·   WP  WP  ·   ·
2| WP  WP  ·   ·   ·   ·   WP  WP
1| WR  WN  WB  ·   WK  WB  WN  WR
  (A   B   C   D   E   F   G   H )

Computer's go:
8| BR  ·   ·   BQ  BK  BB  ·   BR
7| BP  BP  BP  ·   ·   BP  BP  BP
6| ·   ·   BN  ·   BB  BN  ·   ·
5| ·   ·   ·   BP  BP  ·   ·   ·
4| WQ  ·   ·   WP  ·   ·   ·   ·
3| ·   ·   WP  ·   WP  WP  ·   ·
2| WP  WP  ·   ·   ·   ·   WP  WP
1| WR  WN  WB  ·   WK  WB  WN  WR
  (A   B   C   D   E   F   G   H )
Computer Moved  BP:  E7 to E5

Your go USER:
Please enter move in 2 parts
From square:  A4
To square:   B4
8| BR  ·   ·   BQ  BK  BB  ·   BR
7| BP  BP  BP  ·   ·   BP  BP  BP
6| ·   ·   BN  ·   BB  BN  ·   ·
5| ·   ·   ·   BP  BP  ·   ·   ·
4| ·   WQ  ·   WP  ·   ·   ·   ·
3| ·   ·   WP  ·   WP  WP  ·   ·
2| WP  WP  ·   ·   ·   ·   WP  WP
1| WR  WN  WB  ·   WK  WB  WN  WR
  (A   B   C   D   E   F   G   H )

Computer's go:
8| BR  ·   ·   BQ  BK  ·   ·   BR
7| BP  BP  BP  ·   ·   BP  BP  BP
6| ·   ·   BN  ·   BB  BN  ·   ·
5| ·   ·   ·   BP  BP  ·   ·   ·
4| ·   BB  ·   WP  ·   ·   ·   ·
3| ·   ·   WP  ·   WP  WP  ·   ·
2| WP  WP  ·   ·   ·   ·   WP  WP
1| WR  WN  WB  ·   WK  WB  WN  WR
  (A   B   C   D   E   F   G   H )
Computer Moved  BB:  F8 to B4

Your go USER:
Please enter move in 2 parts
```

```
From square:  C3
To square:   B4
8| BR  ·   ·   BQ  BK  ·   ·   BR
7| BP  BP  BP  ·   ·   BP  BP  BP
6| ·   ·   BN  ·   BB  BN  ·   ·
5| ·   ·   ·   BP  BP  ·   ·   ·
4| ·   WP  ·   WP  ·   ·   ·   ·
3| ·   ·   ·   ·   WP  WP  ·   ·
2| WP  WP  ·   ·   ·   ·   WP  WP
1| WR  WN  WB  ·   WK  WB  WN  WR
  (A   B   C   D   E   F   G   H )

Computer's go:
8| BR  ·   ·   BQ  BK  ·   ·   BR
7| BP  BP  BP  ·   ·   BP  BP  BP
6| ·   ·   BN  ·   BB  BN  ·   ·
5| ·   ·   ·   BP  ·   ·   ·   ·
4| ·   WP  ·   BP  ·   ·   ·   ·
3| ·   ·   ·   ·   WP  WP  ·   ·
2| WP  WP  ·   ·   ·   ·   WP  WP
1| WR  WN  WB  ·   WK  WB  WN  WR
  (A   B   C   D   E   F   G   H )
Computer Moved  BP:  E5 to D4

Your go USER:
Please enter move in 2 parts
From square:  E1
To square:   D2
8| BR  ·   ·   BQ  BK  ·   ·   BR
7| BP  BP  BP  ·   ·   BP  BP  BP
6| ·   ·   BN  ·   BB  BN  ·   ·
5| ·   ·   ·   BP  ·   ·   ·   ·
4| ·   WP  ·   BP  ·   ·   ·   ·
3| ·   ·   ·   ·   WP  WP  ·   ·
2| WP  WP  ·   WK  ·   ·   WP  WP
1| WR  WN  WB  ·   ·   WB  WN  WR
  (A   B   C   D   E   F   G   H )

Computer's go:
8| BR  ·   ·   BQ  BK  ·   ·   BR
7| BP  BP  BP  ·   ·   BP  BP  BP
6| ·   ·   BN  ·   BB  BN  ·   ·
5| ·   ·   ·   BP  ·   ·   ·   ·
4| ·   WP  ·   ·   ·   ·   ·   ·
3| ·   ·   ·   ·   BP  WP  ·   ·
2| WP  WP  ·   WK  ·   ·   WP  WP
1| WR  WN  WB  ·   ·   WB  WN  WR
  (A   B   C   D   E   F   G   H )
```

```
Computer Moved  BP:  D4 to E3
CHECK!

Your go USER:
Please enter move in 2 parts
From square:  D2
To square:   E3
8| BR  ·   ·   BQ  BK  ·   ·   BR
7| BP  BP  BP  ·   ·   BP  BP  BP
6| ·   ·   BN  ·   BB  BN  ·   ·
5| ·   ·   ·   BP  ·   ·   ·   ·
4| ·   WP  ·   ·   ·   ·   ·   ·
3| ·   ·   ·   ·   WK  WP  ·   ·
2| WP  WP  ·   ·   ·   ·   WP  WP
1| WR  WN  WB  ·   ·   WB  WN  WR
  (A   B   C   D   E   F   G   H )

Computer's go:
8| BR  ·   ·   BQ  BK  ·   ·   BR
7| BP  BP  BP  ·   ·   BP  BP  BP
6| ·   ·   BN  ·   BB  BN  ·   ·
5| ·   ·   ·   ·   ·   ·   ·   ·
4| ·   WP  ·   BP  ·   ·   ·   ·
3| ·   ·   ·   ·   WK  WP  ·   ·
2| WP  WP  ·   ·   ·   ·   WP  WP
1| WR  WN  WB  ·   ·   WB  WN  WR
  (A   B   C   D   E   F   G   H )
Computer Moved  BP:  D5 to D4
CHECK!

Your go USER:
Please enter move in 2 parts
From square:  E3
To square:   D4
This isn't a legal move, try again
```

You can see 1 problem and some sophisticated behaviours.

- It prints out the board after each move, makes it clear who’s turn it is and what move the computer has made.
- We can see that the validation works as I try in my second move to move in invalid ways or to input jargon and the program asks me again.
- We can also see that, when given the opportunity the computer sacrificed a rook to take a queen.
- We see that I cannot move in a way that causes check. This is after I moved out my king to present an easy check. We also see that it printed check which is useful

The issue we see is when the computer moves a pawn from E7 to E5 over the top of a bishop. This isn't a legal move but I forgot to check that the middle square and more to squares were empty when coding in the pawns movement. **This will be corrected for future.**

I created the following minimax function

```
# import local modules
# cannot import game as causes circular import, if necessary put in same file
from board_state import Board_State
from assorted import ARBITRARILY_LARGE_VALUE
from vector import Vector

# my minimax function takes as arguments:
# Board_State, is_maximiser, alpha and beta (used for pruning) and
check_extra_depth (produces better outcome but slower)
# it returns
# score, child, move

def minimax(board_state: Board_State, is_maximizer: bool, depth, alpha, beta,
check_extra_depth=True):
    # sourcery skip: low-code-quality, remove-unnecessary-else, swap-if-else-
branches
    # assume white is maximizer
    # when calling, if give appropriate max min arg

    # base case
    # if over or depth==0 return static evaluation
    over, _ = board_state.is_game_over_for_next_to_go()
    if depth == 0 or over:
        # special recursive case 1
        # examine terminal nodes that are check to depth 2 (variable depth)

        # to avoid goose chaises, extra resources are allowed if check not
already explored
        if board_state.color_in_check() and check_extra_depth and not over:
            # print(f"checking board state {hash(board_state)} at additional
depth due to check")
            return minimax(
                board_state=board_state,
                is_maximizer=is_maximizer,
                depth=2,
                alpha=alpha,
                beta=beta,
                check_extra_depth=False
            )
```

```
        else:
            # static eval works for game over to
            return board_state.static_evaluation(), None, None

    # define variables used to return more that just score (move and child)
    best_child_game_state: Board_State | None = None
    best_move_vector: Vector | None = None

    # function yields move ordered by how favorable they are (low depth minimax approximation)
    def gen_ordered_child_game_states():
        # this function does a low depth minimax recursive call (special recursive case 2) to give a move a score
        def approx_score_move(move):
            child_game_state = board_state.make_move(*move)

            return minimax(
                board_state=child_game_state,
                depth=depth-2,
                is_maximizer=not is_maximizer,
                alpha=alpha,
                beta=beta,
                check_extra_depth=False
            )[0]
            # print(f"approx_score_move(move={move!r})  ->  {result!r}")

        # if depth is 1 or less just yield moves form legal moves
        if depth <= 1:
            yield from board_state.generate_legal_moves()
        # else sort them
        else:
            # sort best to worse
            # sort ascending order if minimizer, descending if maximizer
            yield from sorted(
                board_state.generate_legal_moves(),
                key=approx_score_move,
                reverse=is_maximizer
            )


    if is_maximizer:
        # set max to -infinity
        maximum_evaluation = (-1)*ARBITRARILY_LARGE_VALUE

        # iterate through moves and resulting game states
        for position_vector, movement_vector in gen_ordered_child_game_states():
```

```
            child_game_state =
board_state.make_move(from_position_vector=position_vector,
movement_vector=movement_vector)

            # evaluate each one
            # general recursive case 1
            evaluation, _, _ = minimax(
                board_state=child_game_state,
                is_maximizer=not is_maximizer,
                depth=depth-1,
                alpha=alpha,
                beta=beta,
                check_extra_depth=check_extra_depth
            )

            # update alpha and max evaluation
            if evaluation > maximum_evaluation:
                maximum_evaluation = evaluation
                best_child_game_state = child_game_state
                best_move_vector = (position_vector, movement_vector)
                alpha = max(alpha, evaluation)

            # where possible, prune
            if beta <= alpha:
                # print("Pruning!")
                break
        # once out of loop, return result
        return maximum_evaluation, best_child_game_state, best_move_vector

    else:
        minimum_evaluation = ARBITRARILY_LARGE_VALUE

        for position_vector, movement_vector in
gen_ordered_child_game_states():
            child_game_state =
board_state.make_move(from_position_vector=position_vector,
movement_vector=movement_vector)
            evaluation, _, _ = minimax(
                board_state=child_game_state,
                is_maximizer=not is_maximizer,
                depth=depth-1,
                alpha=alpha,
                beta=beta,
                check_extra_depth=check_extra_depth
            )

            if evaluation < minimum_evaluation:
                minimum_evaluation = evaluation
```

```
                best_child_game_state = child_game_state
                best_move_vector = (position_vector, movement_vector)
                beta = min(beta, evaluation)

            if beta <= alpha:
                # print("Pruning!")
                break
        return minimum_evaluation, best_child_game_state, best_move_vector
```

it features some efficiency upgrades:

- Alpha beta pruning
- Child nodes examined best first to enhance pruning
- Variable depth in check to close out games

I then created a test for the minimax function

```
# this test is responsible for testing various mutations of the minimax
function and how they play, it is not a data driven test

# imports
from random import choice as random_choice
import unittest
# from functools import wraps
import multiprocessing
import os.path
import csv

from game import Game
from minimax import minimax
from assorted import ARBITRARILY_LARGE_VALUE
# from board_state import Board_State
# from vector import Vector

# # was not needed in the end, this decorator would have repeated a given
function a given number of times
# def repeat_decorator_factory(times):
#     def decorator(func):
#         @wraps(func)
#         def wrapper(*args, **kwargs):
#             for _ in range(times):
#                 func(*args, **kwargs)
#         return wrapper
#     return decorator
```

```python
# this is a utility function that maps a function across an iterable but also
converts the result to a list data structure
def list_map(func, iter):
    return list(map(func, iter))

# this function is used to serialize a pieces matrix for output in a message
# it converts pieces to symbols
def map_pieces_matrix_to_symbols(pieces_matrix):
    return list_map(
        lambda row: list_map(
            lambda square: square.symbol() if square else None,
            row
        ),
        pieces_matrix
    )

# this functions updates a CSV file with the moves and scores of a chess game
for graphical analysis in excel
def csv_write_move_score(file_path, move, score):
    # convert move to a pair of squares
    position_vector, movement_vector = move
    resultant_vector = position_vector + movement_vector

    from_square = position_vector.to_square()
    to_square = resultant_vector.to_square()

    # if file doesn't exist, create is and add the headers
    if not os.path.exists(file_path):
        with open(file_path, "w", newline="") as file:
            writer = csv.writer(file, delimiter=",")
            writer.writerow(("from_square", "to_square", "score"))

    # add data as a new row
    with open(file_path, "a", newline="") as file:
        writer = csv.writer(file, delimiter=",")
        writer.writerow((from_square, to_square, score))

# this contains the majority of the logic to do a bot vs bot test with the
game class
# it is a component as it isn't the whole test
def minimax_test_component(description, good_bot, bad_bot, success_criteria,
write_to_csv=False):
    # sourcery skip: extract-duplicate-method

    # good bot and bad bot make decisions about moves,
    # the test is designed to assert that good bot wins (and or draws in some
cases)
```

```
    # generate csv path
    if write_to_csv:
        # not sure why but the description sometimes contains an erroneous
colon, this is caught and removed
        # was able to locate bug to here, as it is a test I added a quick fix
        # bug located, some description stings included them
        csv_path = f"test_reports/{description}.csv".replace(" ",
"_").replace(":", "")

    # start a new blank game
    # depth irrelevant as computer move function passed as parameter
    game: Game = Game()

    # keep them making moves until return statement breaks loop
    while True:
        # get move choice from bad bot
        _, _, move_choice = bad_bot(game)

        # serialised to is can be passed as a user move (reusing game class)
        position_vector, movement_vector = move_choice
        resultant_vector = position_vector + movement_vector
        from_square, to_square = position_vector.to_square(),
resultant_vector.to_square()

        # implement bad bot move and update csv
        move, score = game.implement_user_move(from_square=from_square,
to_square=to_square)
        if write_to_csv:
            csv_write_move_score(
                file_path=csv_path,
                move=move,
                score=score
            )

        # see if this move causes the test to succeed or fail or keep going
        success, msg, board_state = success_criteria(game,
description=description)
        if success is not None:
            return success, msg, board_state

        # providing good bot function, implement good bot move and update csv
        move, score =
game.implement_computer_move(best_move_function=good_bot)
        if write_to_csv:
            csv_write_move_score(
                file_path=csv_path,
                move=move,
                score=score
```

```
            )

        # again check if this affects the test
        success, msg, board_state = success_criteria(game,
description=description)
        if success is not None:
            return success, msg, board_state

        # # if needed provide console output to clarify that slow bot hasn't
crashed
        # if game.move_counter % 10 == 0 or depth >= 3:
        # print(f"Moves {game.move_counter}:  static evaluation ->
{game.board_state.static_evaluation()}, Minimax evaluation -> {score} by turn
{description}")


# below are some function that have been programmed as classes with a __call__
method.
# these are basically fancy functions that CAN BE HASHED.
# I had to manually do this under the hood hashing as it is needed to allow
communication between the threads
# a job must be hashable to be piped to a thread (separate python instance)


class Random_Bot():
    # picks a random move
    def __call__(self, game):
        # determine move at random
        legal_moves = list(game.board_state.generate_legal_moves())
        assert len(legal_moves) != 0
        # match minimax output structure
        # score, best_child, best_move
        return None, None, random_choice(legal_moves)
    def __hash__(self) -> int:
        return hash("I am random bot, I am a unique singleton so each instance
can share a hash")




# picks a good move
# has constructor to allow for configuration
class Good_Bot():
    # configure for depth and allow variable depth
    def __init__(self, depth, check_extra_depth):
        self.depth = depth
        self.check_extra_depth = check_extra_depth

    # make minimax function call given config
```

```
    def __call__(self, game):
        return minimax(
            board_state=game.board_state,
            is_maximizer=(game.board_state.next_to_go == "W"),
            depth=self.depth,
            alpha=(-1)*ARBITRARILY_LARGE_VALUE,
            beta=ARBITRARILY_LARGE_VALUE,
            check_extra_depth=self.check_extra_depth
        )
    def __hash__(self) -> int:
        return hash(f"Good_Bot(depth={self.depth},
check_extra_depth={self.check_extra_depth})")


# used to look at a game and decide if the test should finish
class Success_Criteria():
    # constructor allow config for stalemates to sill allow test to pass
    def __init__(self, allow_stalemate_3_states_repeated: bool):
        self.allow_stalemate_3_states_repeated =
allow_stalemate_3_states_repeated
    def __call__(self, game: Game, description):
        # returns: success, message, serialised pieces matrix

        # call game over and use a switch case to decide what to do
        match game.check_game_over():

            # if 3 repeat stalemate, check with config wether is is allows
            case True, None, "Stalemate":
                # game.board_state.print_board()
                if game.is_3_board_repeats_in_game_history and
self.allow_stalemate_3_states_repeated:
                    # game.board_state.print_board()
                    # print(f"Success: Stalemate at {game.moves} moves in test
{description}: 3 repeat board states, outcome specify included in allowed
outcomes")
                    return True, f"Success: Stalemate at {game.move_counter}
moves in test {description}: 3 repeat board states, outcome specify included
in allowed outcomes",
map_pieces_matrix_to_symbols(game.board_state.pieces_matrix)
                else:
                    # game.board_state.print_board()
                    # print(f"FAILURE: ({description}) stalemate caused (3
repeats?  -> {game.is_3_board_repeats_in_game_history()})")
                    return False, f"FAILURE: ({description}) stalemate caused
(3 repeats?  -> {game.is_3_board_repeats_in_game_history()})",
map_pieces_matrix_to_symbols(game.board_state.pieces_matrix)
            # good bot loss causes test to fail
            case True, 1, "Checkmate":
                # game.board_state.print_board()
```

```
                # print(f"Failure: ({description}) computer lost")
                return False, f"Failure: ({description}) computer lost",
map_pieces_matrix_to_symbols(game.board_state.pieces_matrix)
            # good bot win causes test to pass
            case True, -1, "Checkmate":
                # game.board_state.print_board()
                # print(f"SUCCESS: ({description}) Game has finished and been
won in {game.move_counter} moves")
                return True, f"SUCCESS: ({description}) Game has finished and
been won in {game.move_counter} moves",
map_pieces_matrix_to_symbols(game.board_state.pieces_matrix)
            # if the game isn't over, return success as none and test will
continue
            case False, _, _:
                return None, None, None

    def hash(self):
        return
hash(f"Success_Criteria(allow_stalemate_3_states_repeated={self.allow_stalemat
e_3_states_repeated})")




# given a test package (config for one test), carry it out
def execute_test_job(test_data_package):
    # deal with unexplained bug where argument is tuple / list length 1
containing relevant dict (quick fix as only a test)
    # I was able to identify that this is where it occurs and add a correction
but I am not sure what the cause of the bug is
    if any(isinstance(test_data_package, some_type) for some_type in (tuple,
list)):
        if len(test_data_package) == 1:
            test_data_package = test_data_package[0]

    # print(f"test_data_package   -->   {test_data_package}")

    # really simple, call minimax test component providing all keys in package
as keyword arguments
    return minimax_test_component(**test_data_package)


# this function takes an iterable of hashable test_packages
# it all 8 logical cores on my computer to multitask to finish the test sooner
def pool_jobs(test_data):
    # counts logical cores
    # my CPU is a 10th gen i7
    # it has 4 cores and 8 logical cores due to hyper threading
    # with 4-8 workers I can use 100% of my CPU
```

```
    cores = multiprocessing.cpu_count()

    # create a pool
    with multiprocessing.Pool(cores) as pool:
        # map the execute_test_job function across the set of test packages
using multitasking
        # return the result
        return pool.map(
            func = execute_test_job,
            iterable = test_data
        )

# test case contains unit tests
# multitasking only occurs within a test, tests are themselves executed
sequentially
# I could pool all tests into one test function but this way multiple failures
can occur in different tests
# (one single test would stop at first failure)

class Test_Case(unittest.TestCase):

    # this function takes the results of the tests from the test pool and
checks the results with a unittest
    # a failure is correctly identified to correspond to the function that
called this function
    # reduces repeated logic
    def check_test_results(self, test_results):
        for success, msg, final_pieces_matrix in test_results:
            # print()
            # for row in final_pieces_matrix:
            #     row = "   ".join(map(
            #         lambda square: str(square).replace("None", ". "),
            #         row
            #     ))
            #     print(row)
            # print(msg)

            # i choose to iterate rather than assert all as this allows me to
have the appropriate message on failure
            self.assertTrue(
                success,
                msg=msg
            )

    # tests basic minimax vs random moves
    def test_vanilla_depth_1_vs_randotron(self):
        # 10 trials as outcome is linked to a random behaviour
        trials = 10
```

```
        # test package generated to include relevant data and logic (bots and
success criteria)
        test_data = {
            "description": "test: depth 1 vanilla vs randotron",
            "good_bot": Good_Bot(depth=1, check_extra_depth=False),
            "bad_bot": Random_Bot(),
            "success_criteria":
Success_Criteria(allow_stalemate_3_states_repeated=True),
            "write_to_csv": False
        }

        # only trial will write to a csv
        test_data_but_to_csv = test_data
        test_data_but_to_csv["write_to_csv"] = True

        self.check_test_results(
            pool_jobs(
                (trials-1) * [test_data] +  [test_data_but_to_csv]
            )
        )

    def test_advanced_depth_1_vs_randotron(self):
        trials = 10

        test_data = {
            "description": "test: depth 1 advanced vs randotron",
            "good_bot": Good_Bot(depth=1, check_extra_depth=True),
            "bad_bot": Random_Bot(),
            "success_criteria":
Success_Criteria(allow_stalemate_3_states_repeated=True),
            "write_to_csv": False
        }

        test_data_but_to_csv = test_data
        test_data_but_to_csv["write_to_csv"] = True

        self.check_test_results(
            pool_jobs(
                (trials-1) * [test_data] + [test_data_but_to_csv]
            )
        )


    def test_depth_2_vs_randotron(self):
        trials = 10

        test_data = {
            "description": "test: depth 2 advanced vs randotron",
```

```
            "good_bot": Good_Bot(depth=2, check_extra_depth=True),
            "bad_bot": Random_Bot(),
            "success_criteria":
Success_Criteria(allow_stalemate_3_states_repeated=False),
            "write_to_csv": False
        }

        test_data_but_to_csv = test_data
        test_data_but_to_csv["write_to_csv"] = True

        self.check_test_results(
            pool_jobs(
                (trials-1) * [test_data] + [test_data_but_to_csv]
            )
        )


    def test_depth_1_advanced_vs_depth_1_vanilla(self):
        # only one trial needed as outcome is deterministic
        trials = 1

        test_data = {
            "description": "test: depth 1 vanilla vs depth 1 variable check",
            "good_bot": Good_Bot(depth=1, check_extra_depth=True),
            "bad_bot": Good_Bot(depth=1, check_extra_depth=False),
            # they aren't different enough in efficacy to guarantee no draws
            "success_criteria":
Success_Criteria(allow_stalemate_3_states_repeated=True),
            "write_to_csv": True
        },

        self.check_test_results(
            pool_jobs(trials*[test_data])
        )

    def test_depth_2_vs_depth_1(self):
        trials = 1

        test_data = {
            "description": "test: depth 2 vs depth 1",
            "good_bot": Good_Bot(depth=2, check_extra_depth=True),
            "bad_bot": Good_Bot(depth=1, check_extra_depth=True),
            "success_criteria":
Success_Criteria(allow_stalemate_3_states_repeated=False),
            "write_to_csv": True
        }

        self.check_test_results(
```

```
            pool_jobs(trials*[test_data])
        )

    def test_depth_3_vs_depth_2(self):
        trials = 1

        test_data = {
            "description": "test: depth 3 vs depth 2",
            "good_bot": Good_Bot(depth=3, check_extra_depth=True),
            "bad_bot": Good_Bot(depth=2, check_extra_depth=True),
            "success_criteria":
Success_Criteria(allow_stalemate_3_states_repeated=False),
            "write_to_csv": True
        }

        self.check_test_results(
            pool_jobs(trials*[test_data])
        )

    def test_depth_3_vs_depth_1(self):
        trials = 1

        test_data = {
            "description": "test: depth 3 vs depth 1",
            "good_bot": Good_Bot(depth=3, check_extra_depth=True),
            "bad_bot": Good_Bot(depth=1, check_extra_depth=True),
            "success_criteria":
Success_Criteria(allow_stalemate_3_states_repeated=False),
            "write_to_csv": True
        }

        self.check_test_results(
            pool_jobs(trials*[test_data])
        )

    def test_depth_3_vs_randotron(self):
        trials = 4

        test_data = {
            "description": "test: depth 3 vs randotron",
            "good_bot": Good_Bot(depth=3, check_extra_depth=True),
            "bad_bot": Random_Bot(),
            "success_criteria":
Success_Criteria(allow_stalemate_3_states_repeated=False),
            "write_to_csv": False
        }

        test_data_but_to_csv = test_data
```

```
        test_data_but_to_csv["write_to_csv"] = True

        self.check_test_results(
            pool_jobs(
                (trials-1) * [test_data] + [test_data_but_to_csv]
            )
        )




if __name__ == '__main__':
    unittest.main()
```

I went to great effort to improve the tests efficiency and this wasn't wasted. It allowed me to perform tests with many trials simultaneously in order to fully utilise my CPU.

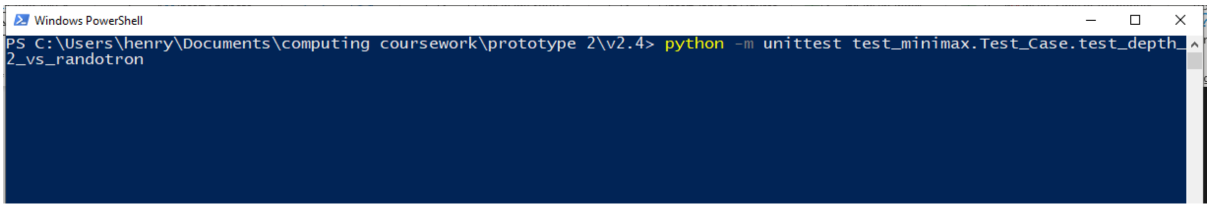
Currently all the test I have run today were successful **but** even given a whole afternoon I haven't been able to run all the tests. This is due to the depth 3 tests. Due to variable depth, the number of static evaluations can be (branching factor)^5 (really slow). This combinatorial explosion means that it takes more than 4 hours to run the tests.

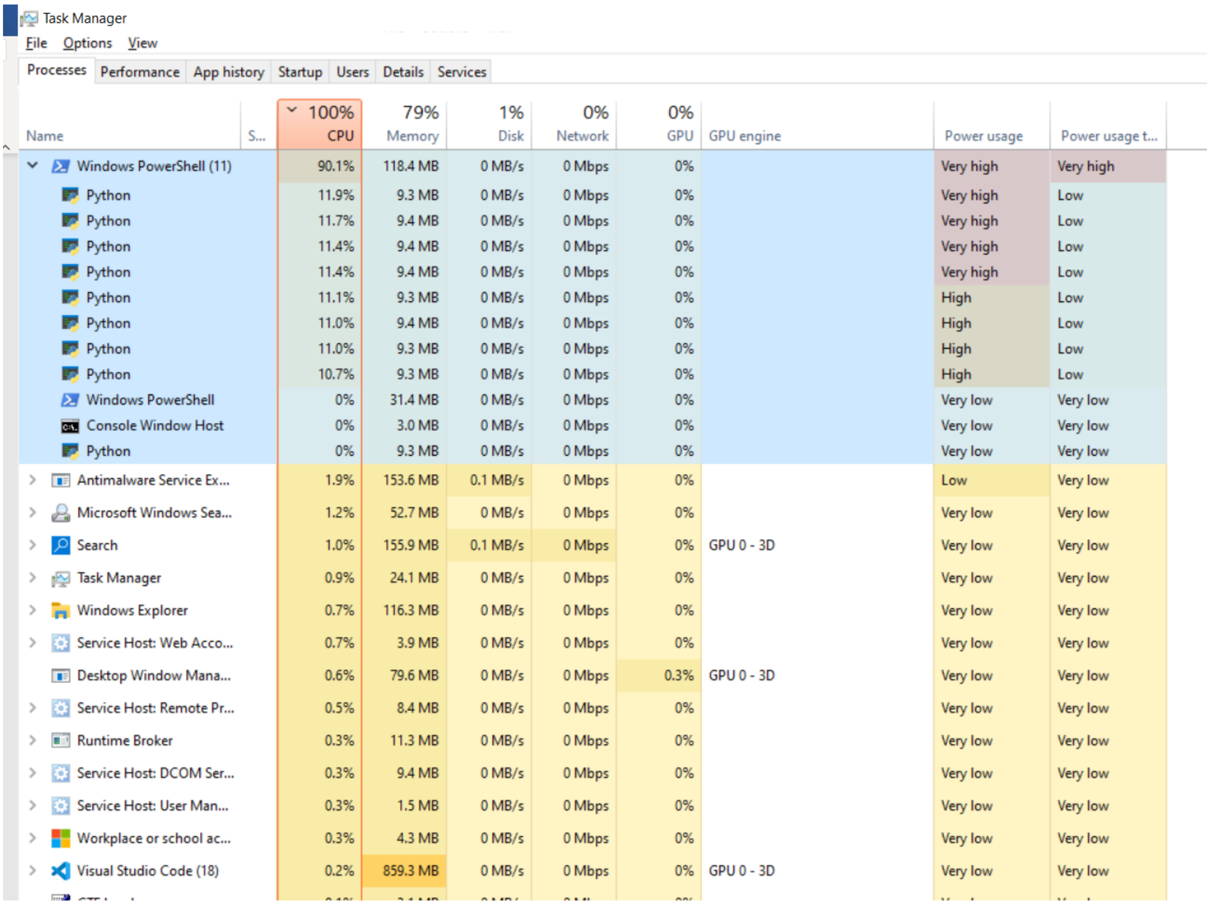
The tests I have run successfully are:

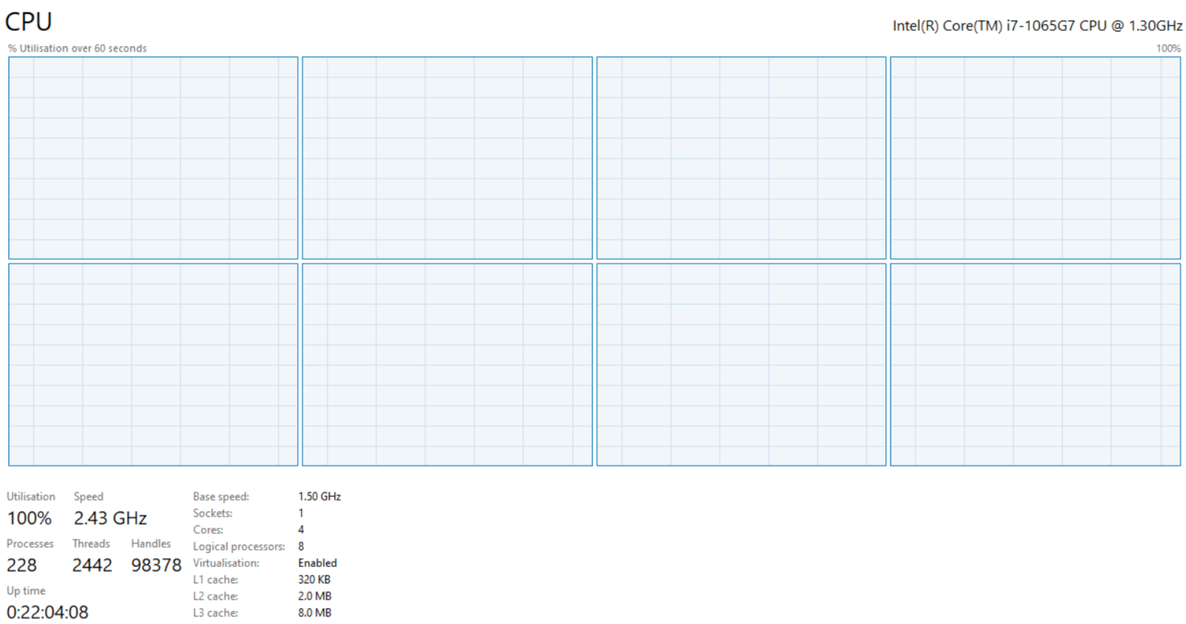
- test_depth_1_vanilla_vs_depth_1_variable_check
- test_depth_2_vs_depth_1
- test_depth_2_advanced_vs_randotron
- test_depth_3_vs_depth_1
- test_depth_1_advanced_vs_randotron

Here is me showing how multitasking has allowed me to fully utilise my CPU

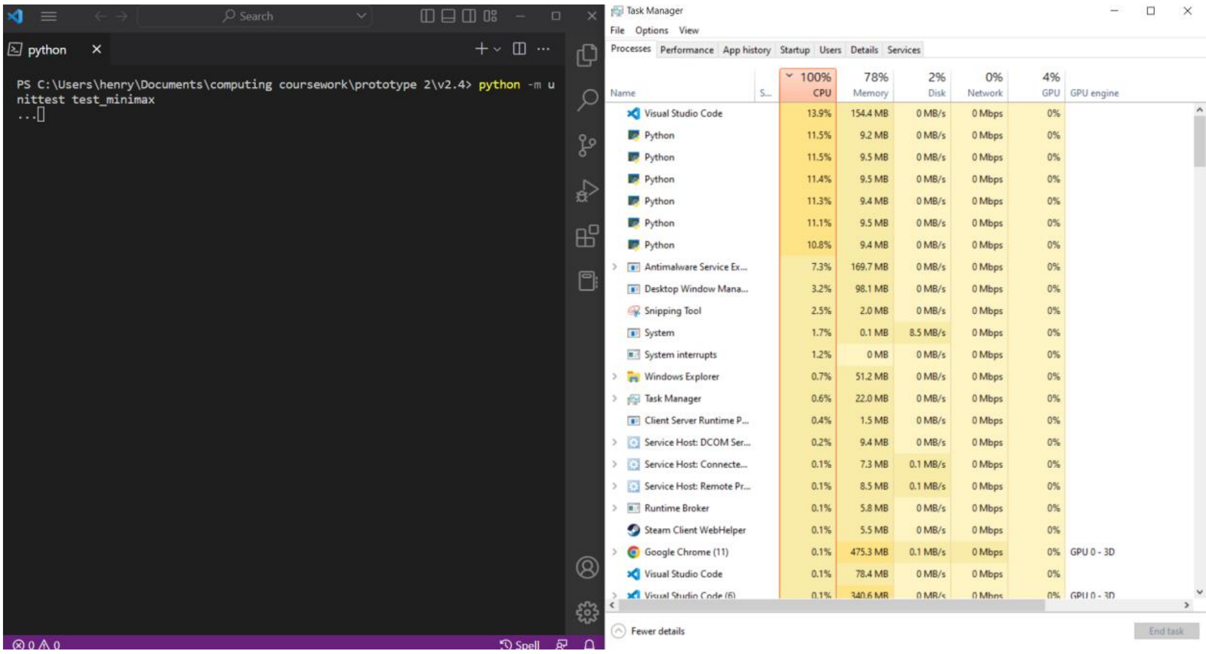
Let me run depth 2 vs randotron with 10 trials to demonstrate:

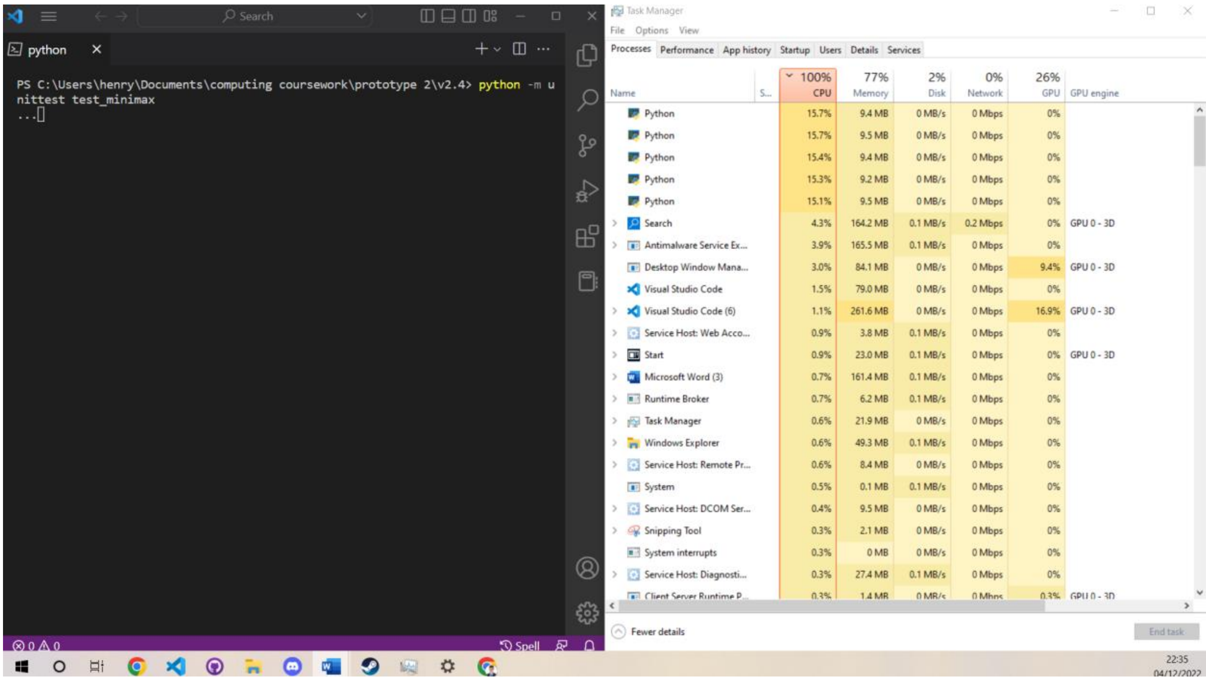


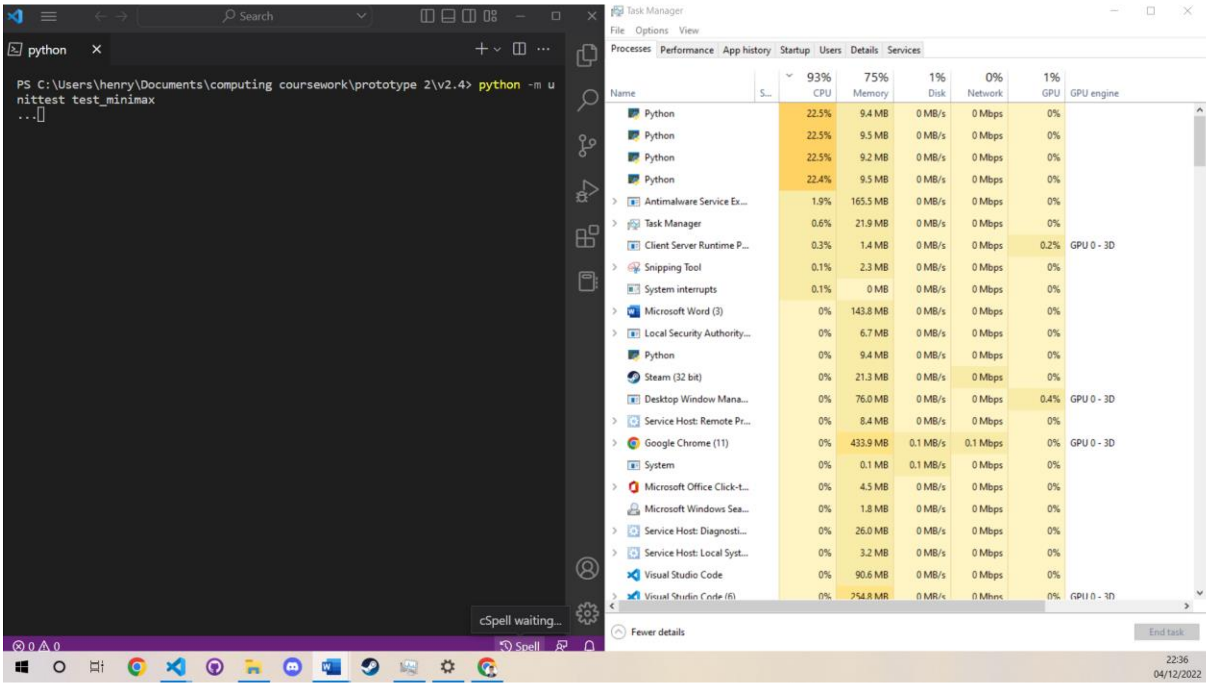
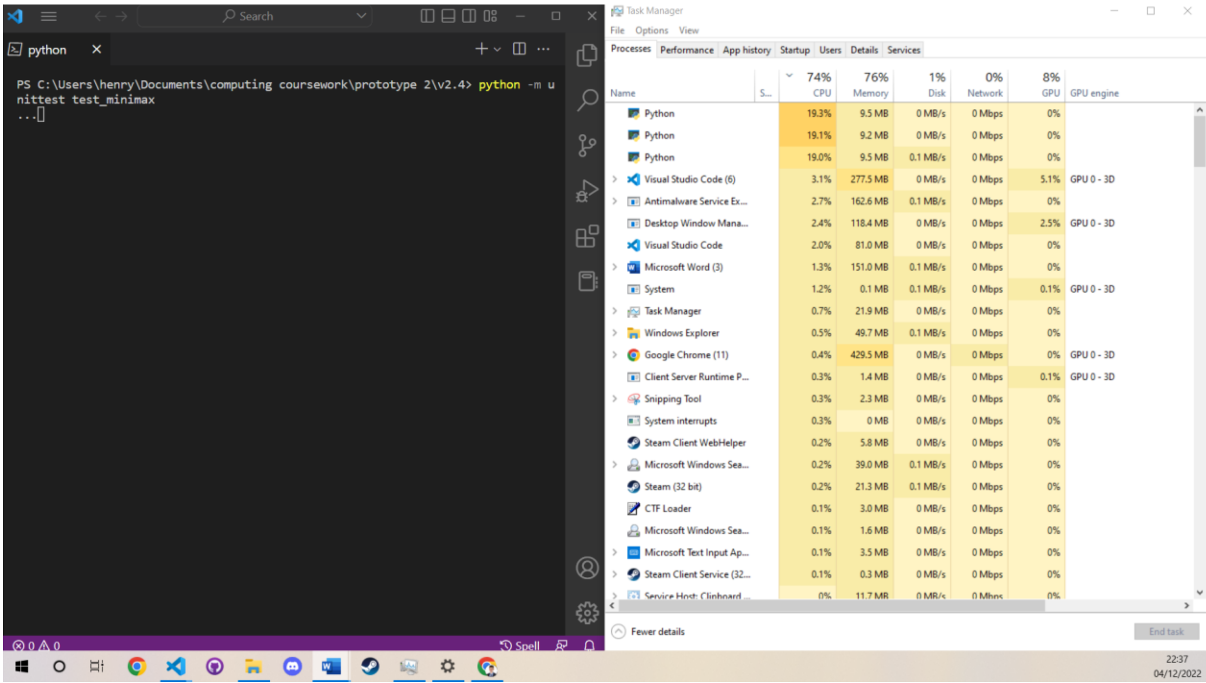


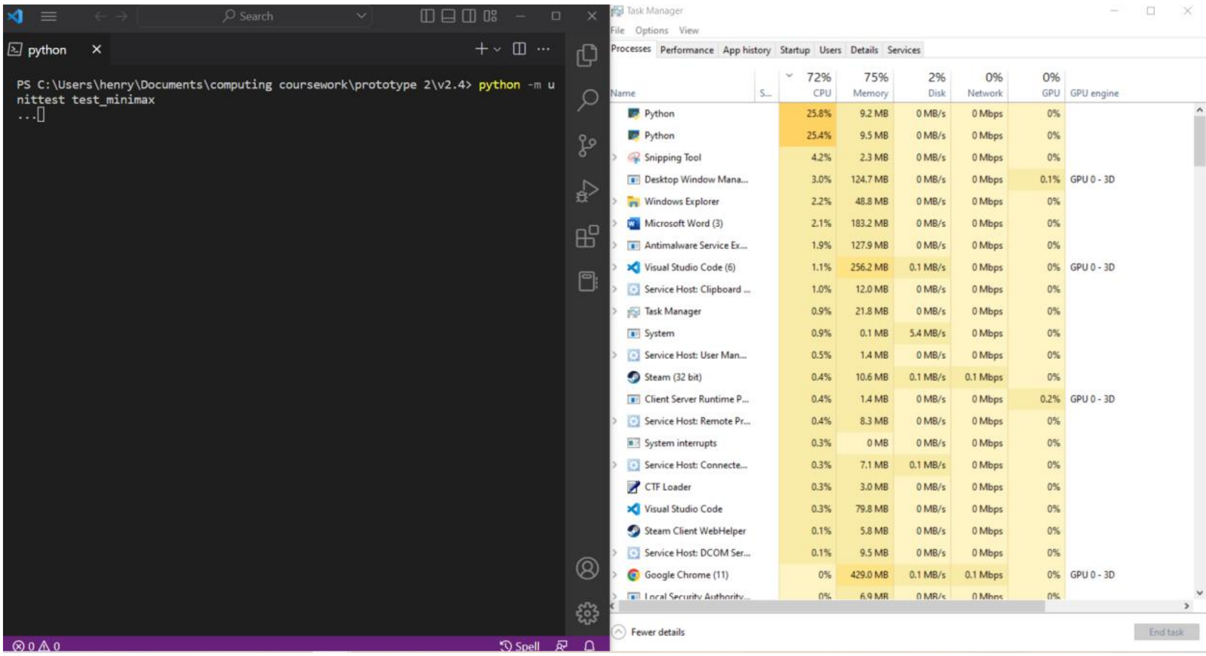


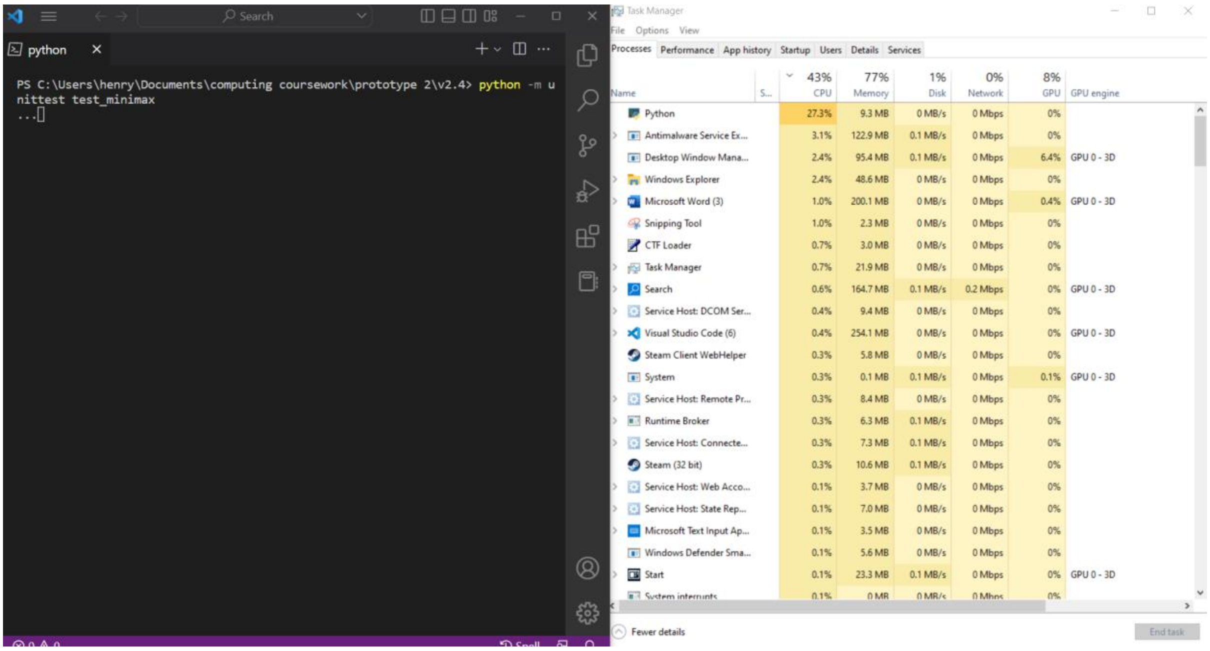
Before doing this my CPU was only ever able to devote 25% power to my python program.


python
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4> python -m unittest test_minimax
...
Task Manager
File Options View
Processes Performance App history Startup Users Details Services
100% CPU 78% Memory 2% Disk 0% Network 4% GPU GPU engine
Fewer details
End task


python
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4> python -m unittest test_minimax
...
Task Manager
File Options View
Processes Performance App history Startup Users Details Services
100% CPU 77% Memory 2% Disk 0% Network 26% GPU GPU engine
Fewer details
End task
22:35


python
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4> python -m unittest test_minimax
...
Task Manager
File Options View
Processes Performance App history Startup Users Details Services
93% CPU 75% Memory 1% Disk 0% Network 1% GPU GPU engine
22:36
04/12/2022

python
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4> python -m unittest test_minimax
...
Task Manager
File Options View
Processes Performance App history Startup Users Details Services
74% CPU 76% Memory 1% Disk 0% Network 8% GPU GPU engine
22:37
04/12/2022





As tasks finish the number of workers decreases. As my computer has 4 physical cores and 8 logical cores, near 100% cpu usage can continue until there are less than 4 threads. At this point each thread can only use 25% of the available CPU power. This means that the test doesn't use the CPU fully at the end. This is only problematic is one of the games is particularly long and tends to stalemate as the next test cannot begin until this one has finished.

The first time I created the randobot and ran the test for depth 1 I found the issue where the game was resulting in a stalemate where it was endlessly repeating. I looked it up and in chess if a game state is repeated 3 times a stalemate occurs. I implemented this using the hash and frequency table in the Game class. This does end the game when this occurs. However it does mean that that the minimax function is unaware that a stalemate can occur this way as the logic isn't happening in the board state class. This shouldn't be an issue however as, when an infinite loop occurs the minimax will anticipate that its mean evaluation won't change and so the stalemate wouldn't affect its behaviour greatly anyway.

I have added this to my unit test as bots of similar skill level may sometimes draw. This can be specified in the test criteria.

I also recorded the utility values of the various tests in CSV files so that it could be graphed. I had a problem with this where the workers from each thread were all adding to the file, making it a useless mess. The fix was to add a .copy()

```
    # tests basic minimax vs random moves
    def test_vanilla_depth_1_vs_randotron(self):
        # 10 trials as outcome is linked to a random behaviour
        # trials = 10
        trials = 5

        # test package generated to include relevant data and logic (bots and
success criteria)
        test_data = {
            "description": "test: depth 1 vanilla vs randotron",
            "good_bot": Good_Bot(depth=1, check_extra_depth=False),
            "bad_bot": Random_Bot(),
            "success_criteria":
Success_Criteria(allow_stalemate_3_states_repeated=True),
            "write_to_csv": False
        }

        # only trial will write to a csv
        test_data_but_to_csv = test_data.copy()
        test_data_but_to_csv["write_to_csv"] = True

        self.check_test_results(
            pool_jobs(
                (trials-1) * [test_data] +  [test_data_but_to_csv]
            )
        )
```

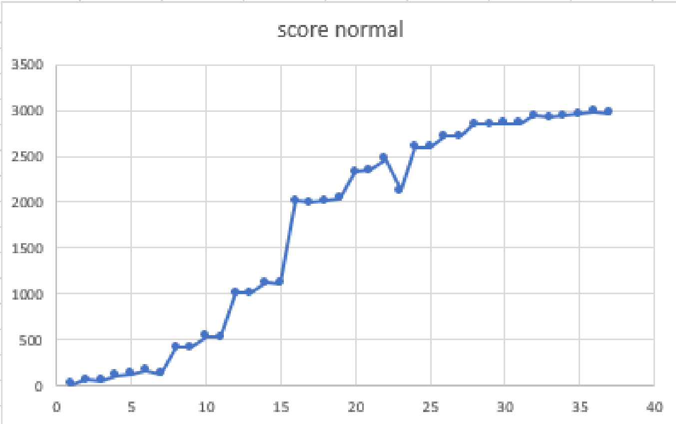
The issue was that I was passing the dictionary by reference to a new variable and so I wasn't mutating one test to use the CSV file but all of them.

This issue was particularly frustrating as I was wating hours for a set of corrupted data.

I have left my unit test running overnight and in 9 hours it still hasn't finished. The depth 3 vs depth 2 and depth 3 vs randotron tests haven't finished. This highlights an issue with the minimax testing as the depth 3 tests far too long. I should still be able to analyse some of the other data:

Here is the data to illustrate, I added columns D and E and then created a graph

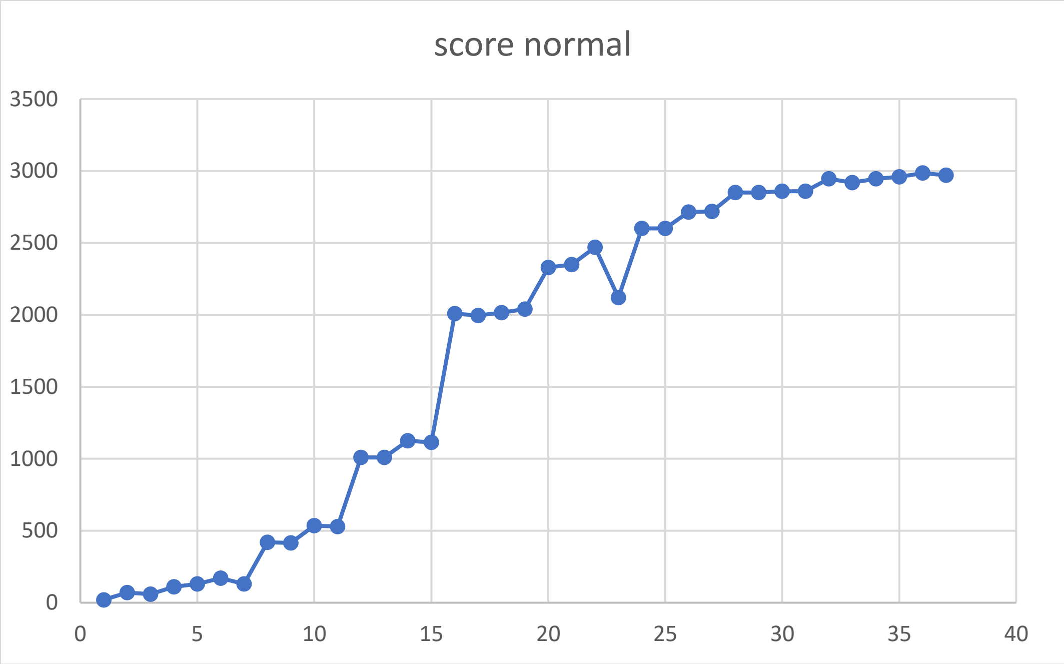
	A	B	C	D	E	F
1	from_sq	to_squar	score	move nu	score normal	
2	F2	F3	-20	1	20	
3	B8	C6	-70	2	70	
4	G1	H3	-60	3	60	
5	G8	F6	-110	4	110	
6	C2	C3	-130	5	130	
7	D7	D5	-170	6	170	
8	E2	E4	-130	7	130	
9	C8	H3	-420	8	420	
10	E4	E5	-415	9	415	
11	H3	G2	-535	10	535	
12	D1	E2	-530	11	530	
13	G2	H1	-1010	12	1010	
14	E1	F2	-1010	13	1010	
15	H1	F3	-1125	14	1125	
16	B1	A3	-1115	15	1115	
17	F3	E2	-2010	16	2010	
18	F1	G2	-1995	17	1995	
19	E7	E6	-2015	18	2015	
20	G2	H1	-2040	19	2040	
21	F8	A3	-2330	20	2330	
22	F2	E3	-2350	21	2350	
23	A3	B2	-2470	22	2470	
24	E3	E2	-2120	23	2120	
25	B2	A1	-2600	24	2600	
26	E2	F2	-2600	25	2600	
27	A1	C3	-2715	26	2715	
28	A2	A4	-2720	27	2720	
29	C3	E5	-2850	28	2850	
30	F2	E2	-2850	29	2850	
31	D5	D3	-2860	30	2860	
32	E2	E1	-2860	31	2860	
33	E5	H2	-2945	32	2945	
34	H1	G2	-2920	33	2920	
35	E6	E4	-2945	34	2945	
36	G2	H3	-2960	35	2960	
37	E4	E2	-2985	36	2985	
38	H3	F5	-2970	37	2970	
39	H2	G3	-1E+06	38	1000000	



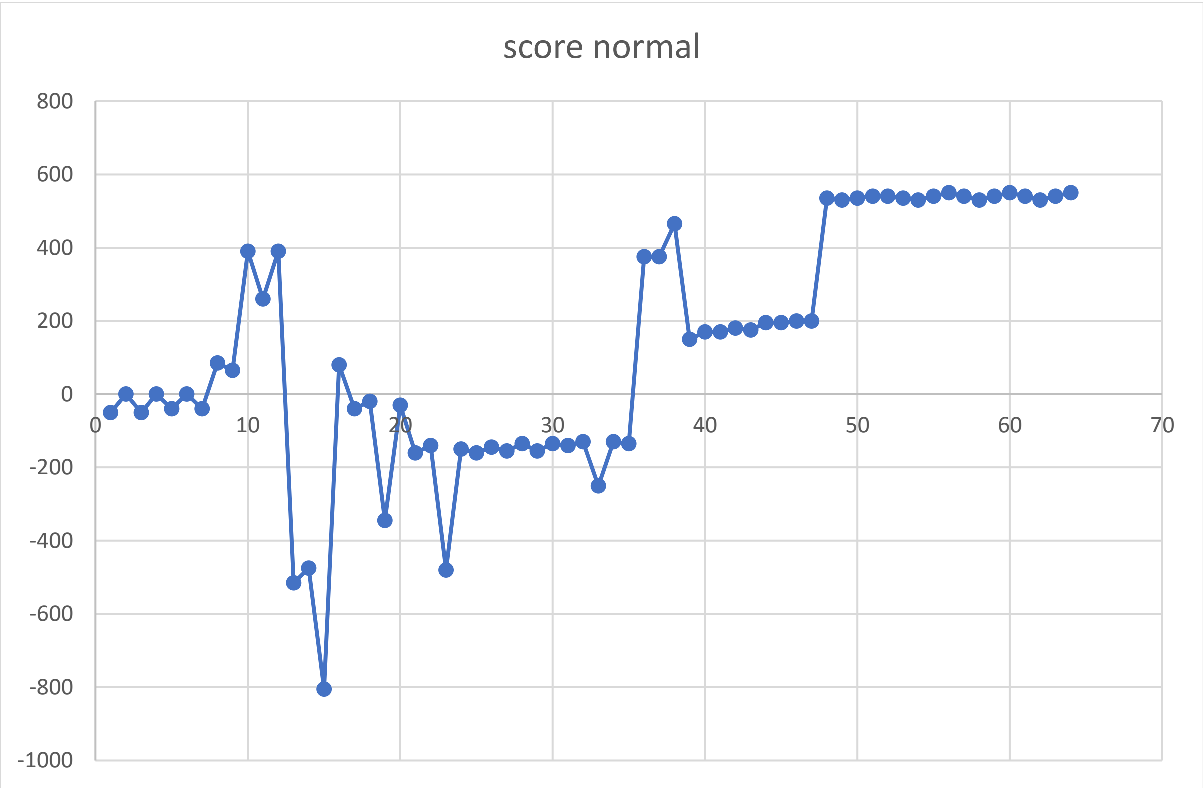
It shows the static evaluations and how it improves for the computer until a win

I will now show some of the graphs:

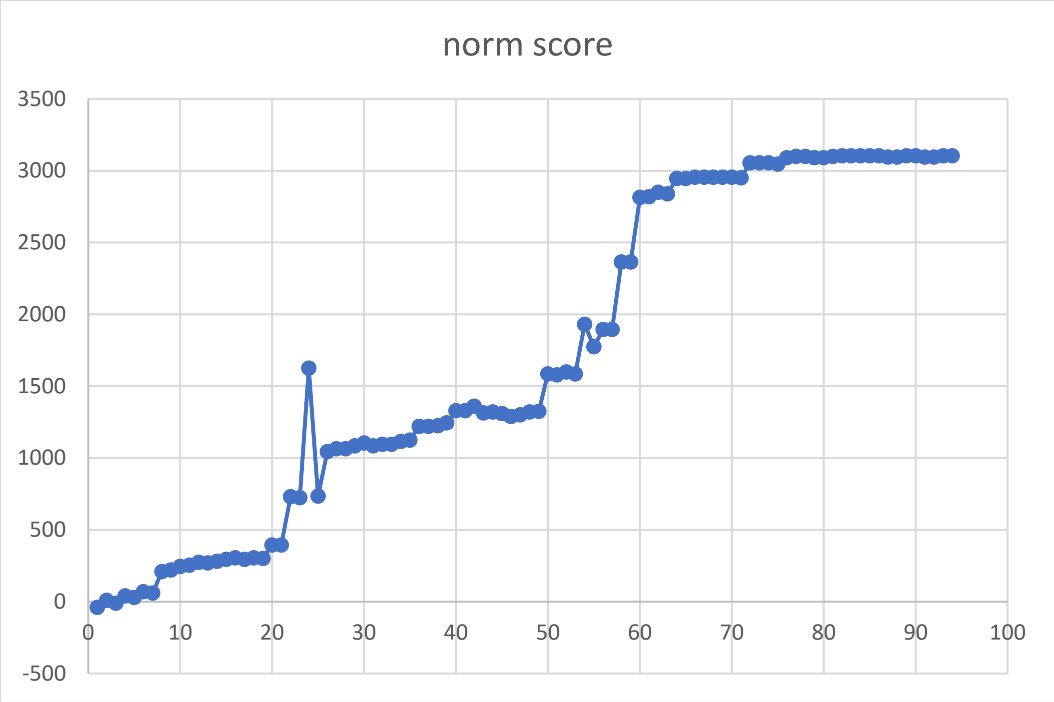
Depth 1 advanced vs Ranotron (ends in a win)



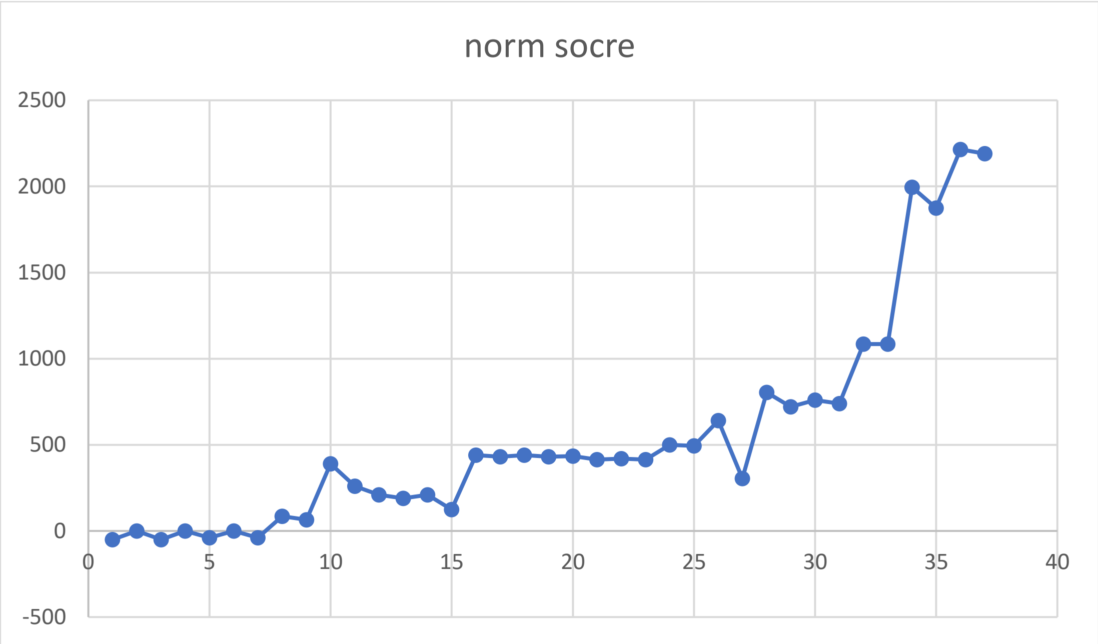
Depth 1 advanced vs depth 1 vanilla (ends in stalemate 3 repeat)



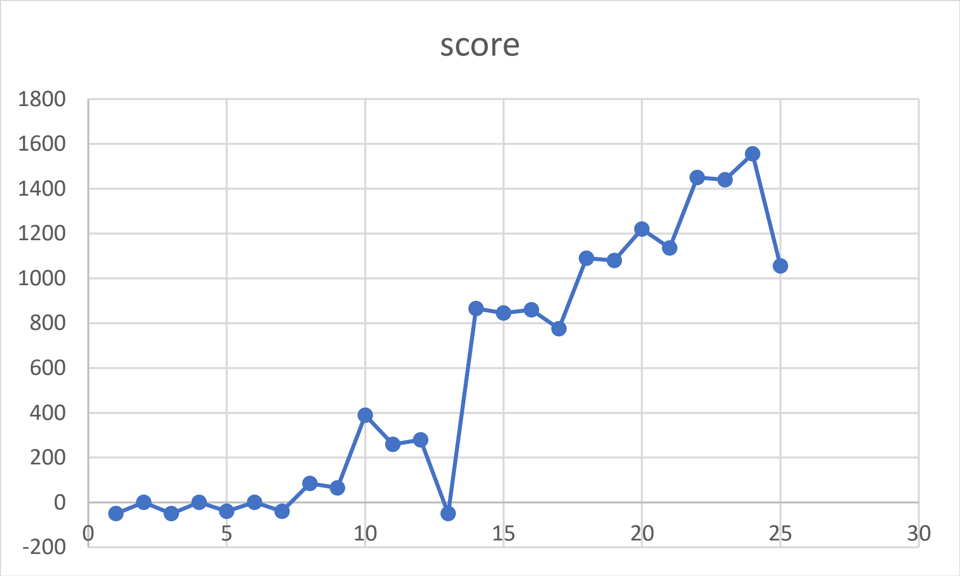
Depth 2 advanced vs Ranotron (ends in win):



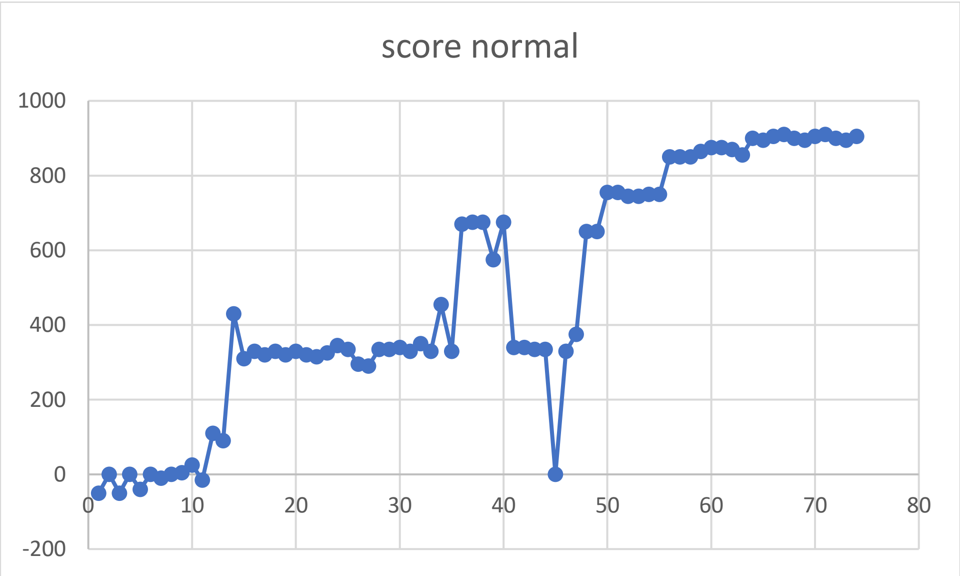
Depth 2 advanced vs depth 1 advanced (deterministic: win every time):



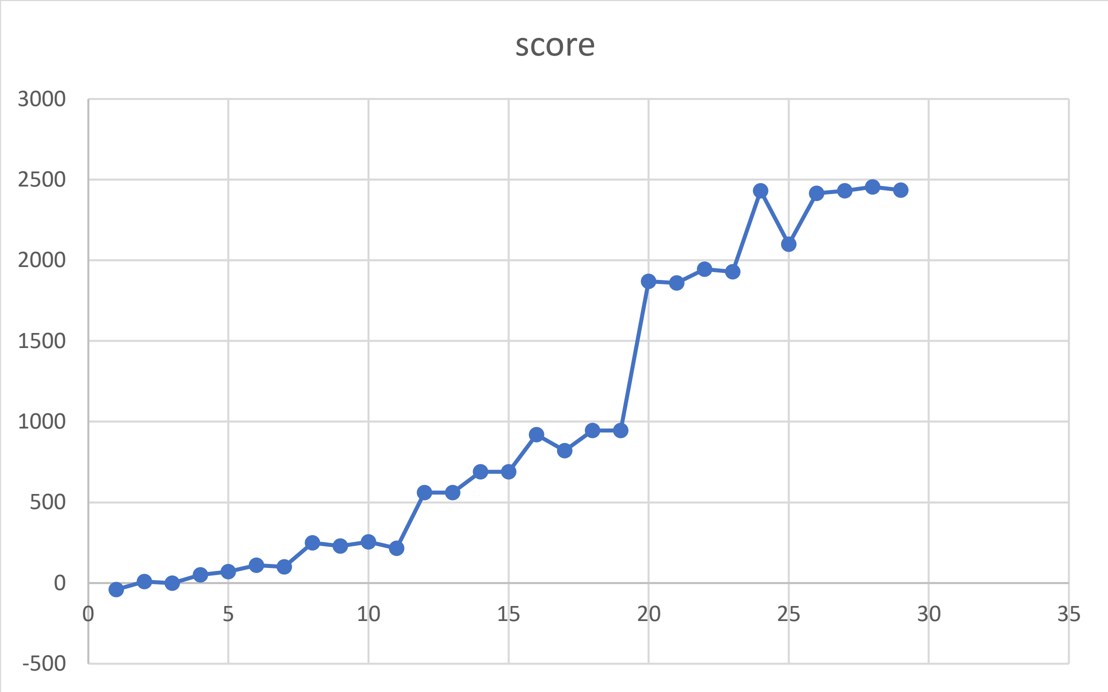
Depth 3 advanced vs depth 1 advanced (deterministic: win every time)



Depth 3 vs depth 2:



Depth 1 vanilla vs Ranotron:



I found analysing these graphs to be very interesting. It seems that both depth 1 and depth 2 can occasionally stalemate with Ranotron. It also seems that depth 1 vanilla vs advanced can stalemate. I believe that this is because they are quite well matched algorithms therefore the better one can only guarantee win or draw.

I thought It was also nice to see how depth 3 vs depth 1 got to checkmate quicker (25 moves) than depth 2 (37 moves). This shows that these algorithms are at there best when playing a rational opponent they can predict.

The antithesis of this is that, while depth 2 wins against depth 1, depth 1 wins quicker against Ranotron. This is because the minimax function assumes the opponent will play optimally. If the opponent isn't playing rationally then carefully selected move will result in the game lasting longer than necessary.

We also see that depth 3 vs depth 2 resulted in a stalemate however depth 3 was able to maximise its score. This shows how the algorithms are relatively similar and are a match for each other (able to draw)

I hope to get the graphs for the other tests. To do this I will run the tests that haven't finished again another time. However not finishing is still a outcome. It shows that it would not currently be possible to play a chess game vs depth 3 as it takes too long. This is due to the combinatorial explosion at a higher depth and the exponential time complexity of the algorithm.

We can also see that depth 1 vanilla beat Ranotron faster (29moves vs 37 moves) than vanilla depth 1. We also saw that even though these 2 algorithms drew in a 3 repeat stalemate, the advanced version was able to have a higher static evaluation.

All of these results show that:

- my minimax algorithm works
- a higher depth beat lower depth in effectiveness
- variable depth checking beats without in effectiveness

I also saw from manually running the algorithm with and without, that alpha beta pruning was improving performance. It is unclear weather pre-sorting child nodes has improved performance as this performance increase would only kick in at higher depths which I was unable to run before.

Validation:
With regards to validation, some was added in the game class and console chess program. This included:

- invalid input for users move (not chess squares) → value error
- illegal move → illegal move error
- not the users turn to go →not user turn error

This validation worked well enough to allow for the console chess game to be robust. However, the intended use for this engine is in a webserver as part of a full stack website. This means that I will need both client side and server side validation. The server side validation will be added to the flask server.

I intend to add more robust validation when I write the flask server, the chess engine will assume that the inputs it gets are valid.

Feedback from Stakeholder

I received feedback from a stakeholder who played the chess game:
They were very impressed that is could play chess and, in their game, the pawn forward 2 bug didn't arise.

Regarding the user interface they liked:

- the fact you could see the letters and numbers at the side of the board
- the fact that if printed out the computer's move
- the fact that it printed out check
- the fact that it correctly prevented illegal moves when in check
- the fact that it took pieces of high value when given the opportunity

They suggested:

- that I find a way of clearing the console / text output after the computer's move so that the user only sees the one relevant board and not many older ones if they scroll up.

What I took from this feedback:

I will keep this suggestion in mind if I am unable to get the GUI to work in the next protype. I will also consider adding letters and squares to the edges of the chess board in my GUI. While the user will input their moves by clicking, this will help contextualise the move history.

I did have an issue with the minimax function where the static evaluations were not what they should have been. I realised that the static evaluation for the starting positions was not 0. To correct this I made a test and then reviewed my code. The corrections was to flip the value matrices for black pieces as the matrices were not symmetrical and were from white's perspective. This corrections was easy as it could be made just in the pieces class.

```
    # this should use the position vector and value matrix to get the value of
the piece
    def get_value(self, position_vector: Vector):
        # flip if black as matrices are all for white pieces
        if self.color == "W":
            row, column = 7-position_vector.j, position_vector.i
        else:
            row, column = position_vector.j, position_vector.i

        # return sum of inherent value + value relative to positon on board
        return self._value + self._value_matrix[row][column]
```

I also attempted to refactor the King.movement_vecotor method, accidently breaking it. This was an easy fix as I ran the unit tests which highlighted the issue.

Changes/Fixes that I now plan to make to the design or code as a result of testing and feedback

- I intend to make changes to the command line interface (CLI) to make it clearer by removing old boards
- I intend to make the minimax function more efficient so that I can run at depth 3 in a reasonable time frame.
- I intend to fix the pawn forward 2 issue and add a test to ensure that it is fixed.

Evaluation

Overall I am extremely happy with this iteration.

I feel that in this iteration I accomplished my ambitions goals and created a robust chess engine complete with a chess bot that can beat me (I believe minimax depth 3 with variable depth surpasses my average ability). With odd exceptions such as the pawn move forward 2 bug, everything is fully functional and tested. This is a lesson in how testing isn't always perfect however, without the automated testing, I don't think I would have succeeded.

My stakeholders seem very impressed that my program can actually play chess in an intelligent manner. I am also pleasantly surprised by how well the final product has turned out.

While my testing was not a complete success (as expected) it was invaluable. My pieces module ended up having the most undiagnosed issues so in hindsight I should have tested it more thoroughly. I realised that most of the time testing was not writing tests but debugging functions. Because tests were quick to write I approached the problem like an engineer and wrote many. I was extremely happy with the results. They allowed me to debug essential function to allow my minimax function to work.

Evidence of testing (excluding minimax tests)

```
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4> python -m unittest
..........................................................................
----------------------------------------------------------------------
Ran 74 tests in 0.288s

OK
PS C:\Users\henry\Documents\computing coursework\prototype 2\v2.4>
```

The interface was not a focus of the project and so it received limited time investment. However, I was able to make the interface clear to my stakeholders by using standard chess symbols for pieces and standard coordinates. Because of this I think that the interface was also, considering requirements and expectations at this state, a success.

This prototype was a success as it was able to achieve the specified aims beyond the bare minimum. This will make producing the next prototype much easier as I have already created a robust chess engine.