Student: Mark Klobukov

Course: CS 576 – Dependable Software Systems

Instructor: Dr. Vokolos

Report on: Chess Move Validation Engine

Introduction:

In this project, a move engine for chess was implemented. Program prompts for input like so:

```
WHITE: Rf1, Kg1, Pf2, Ph2, Pg3

BLACK: Kb8, Ne8, Pa7, Pb7, Pc7, Ra5

PIECE TO MOVE: Rf1
```

And returns valid positions for the given piece to move to:

```
LEGAL MOVES FOR Rf1: e1, d1, c1, b1, a1
```

The program is contained in the file called **main.py** and it was written in Python3. In order to run the program, either execute **python3 main.py** and manually enter configurations like above, or execute **python3 main.py < moves.txt** where the **moves.txt** file contains black configuration, white configuration, and the piece to move in the following format:

```
Rf1, Kg1, Pf2, Ph2, Pg3

Kb8, Ne8, Pa7, Pb7, Pc7, Ra5

Rf1
```

The program assumes that the input is valid and provides no check whether the user entered configurations correctly. Following implementation details are important to mention:

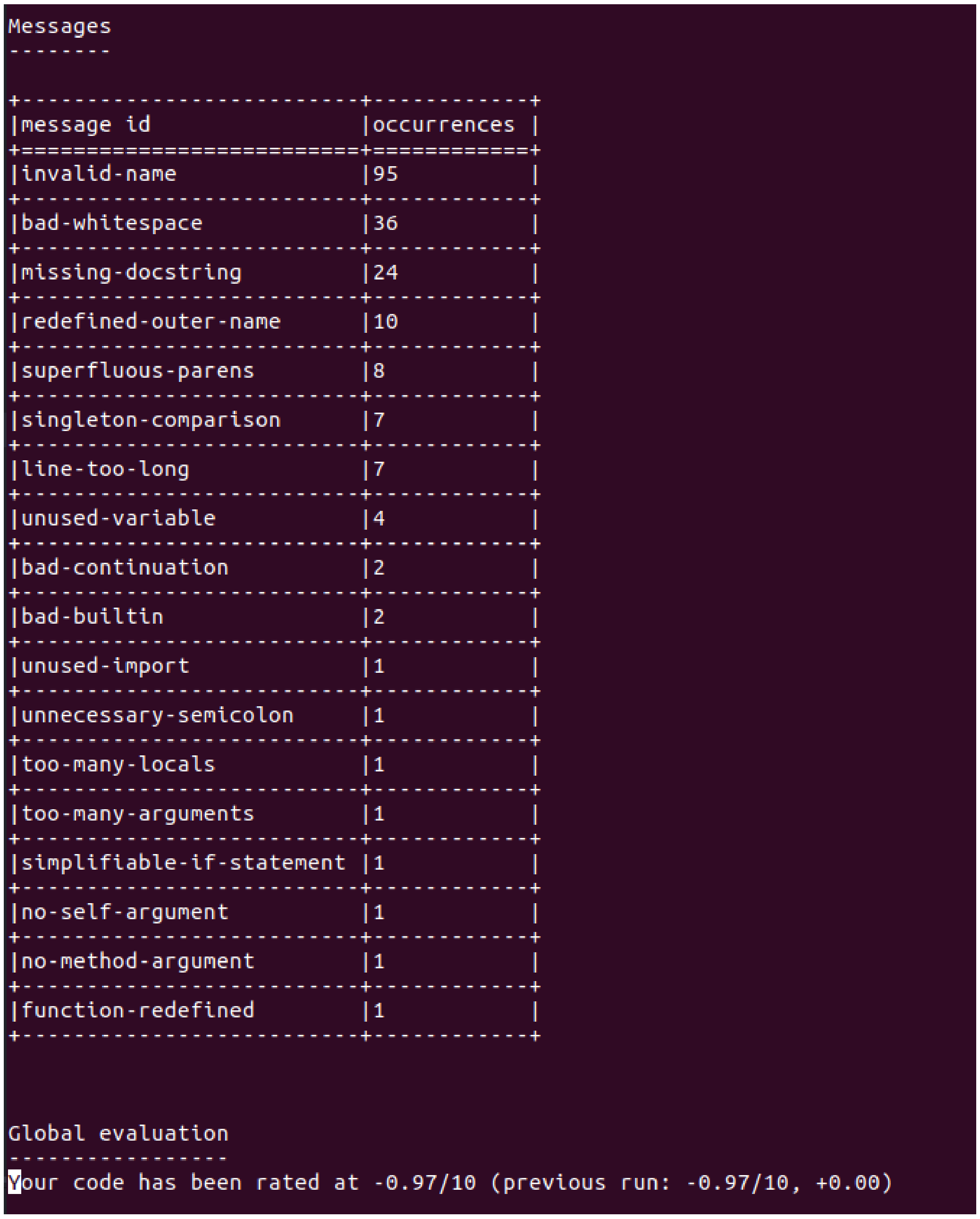
- The program outputs no valid moves if the opponent’s king is in check. For example, it does not make sense for any white piece to move if the black king is in check.
- When computing valid moves for a piece, a program checks if a given move will expose king to a check. If so, the move is deemed invalid.
- The program does not check for the possibility of castling

The next two sections of the report outline my experience with static analysis and code coverage tools for testing and verifying the engine.

Part 1: Static analysis.

The project was completed in Python 3, and the leading static analysis tool for this language is Pylint.

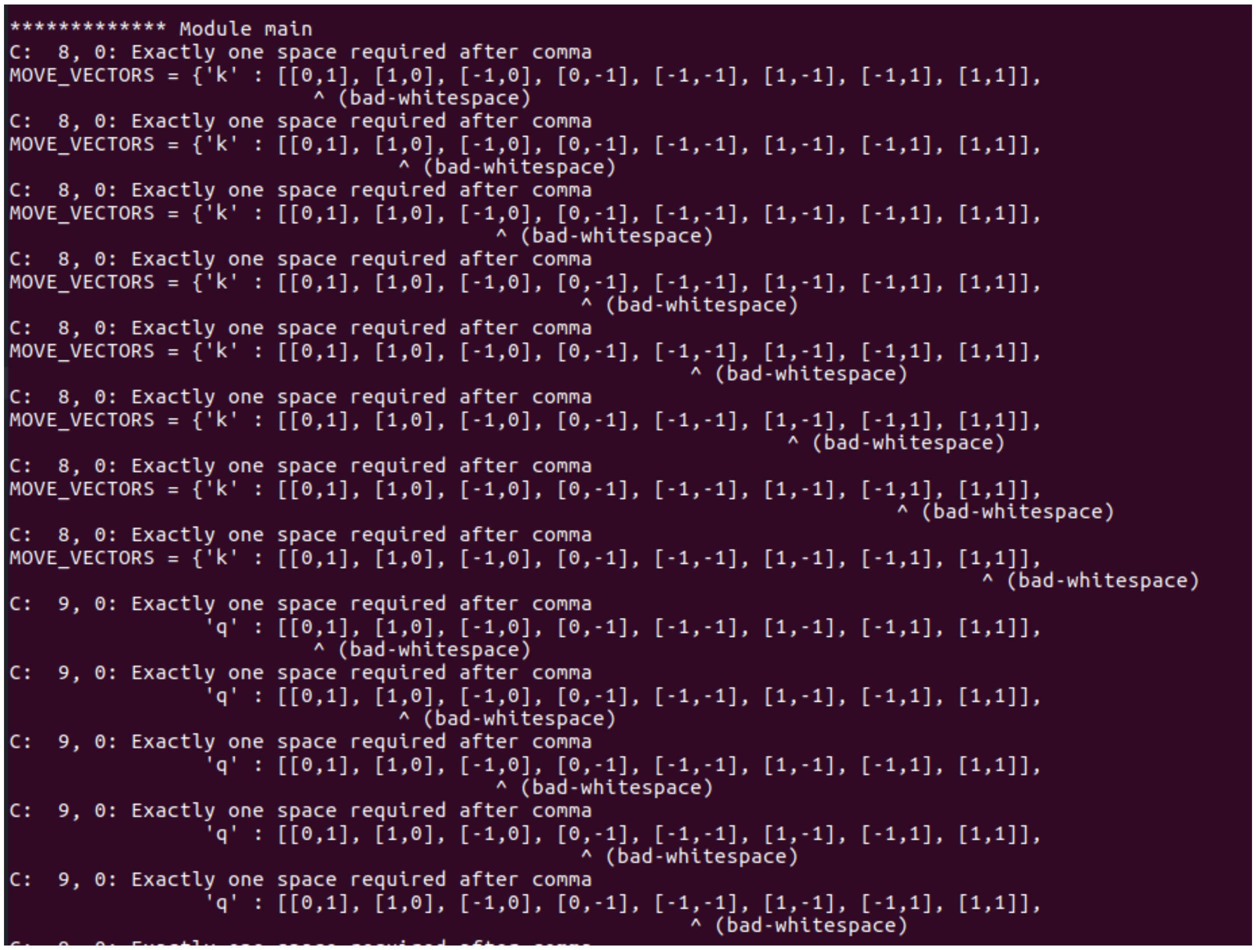
After the very first run of Pylint, I got 285 warnings (which are all provided in the file called **firstOutput)**. The summary of results was as follows (with an extremely low score of -0.97/10):



```
Messages
--------

+-------------------------+------------+
|message id               |occurrences |
+=========================+============+
|invalid-name             |95          |
+-------------------------+------------+
|bad-whitespace           |36          |
+-------------------------+------------+
|missing-docstring        |24          |
+-------------------------+------------+
|redefined-outer-name     |10          |
+-------------------------+------------+
|superfluous-parens       |8           |
+-------------------------+------------+
|singleton-comparison     |7           |
+-------------------------+------------+
|line-too-long            |7           |
+-------------------------+------------+
|unused-variable          |4           |
+-------------------------+------------+
|bad-continuation         |2           |
+-------------------------+------------+
|bad-builtin              |2           |
+-------------------------+------------+
|unused-import            |1           |
+-------------------------+------------+
|unnecessary-semicolon    |1           |
+-------------------------+------------+
|too-many-locals          |1           |
+-------------------------+------------+
|too-many-arguments       |1           |
+-------------------------+------------+
|simplifiable-if-statement |1           |
+-------------------------+------------+
|no-self-argument         |1           |
+-------------------------+------------+
|no-method-argument       |1           |
+-------------------------+------------+
|function-redefined       |1           |
+-------------------------+------------+



Global evaluation
-----------------
Your code has been rated at -0.97/10 (previous run: -0.97/10, +0.00)
```

Figure 1: Summary of Pylint results on the first run

A large portion of results warned me about an absent space after a comma in my vector declarations:



```
************* Module main
C:  8, 0: Exactly one space required after comma
MOVE_VECTORS = {'k' : [[0,1], [1,0], [-1,0], [0,-1], [-1,-1], [1,-1], [-1,1], [1,1]],
                         ^ (bad-whitespace)
C:  8, 0: Exactly one space required after comma
MOVE_VECTORS = {'k' : [[0,1], [1,0], [-1,0], [0,-1], [-1,-1], [1,-1], [-1,1], [1,1]],
                                ^ (bad-whitespace)
C:  8, 0: Exactly one space required after comma
MOVE_VECTORS = {'k' : [[0,1], [1,0], [-1,0], [0,-1], [-1,-1], [1,-1], [-1,1], [1,1]],
                                        ^ (bad-whitespace)
C:  8, 0: Exactly one space required after comma
MOVE_VECTORS = {'k' : [[0,1], [1,0], [-1,0], [0,-1], [-1,-1], [1,-1], [-1,1], [1,1]],
                                               ^ (bad-whitespace)
C:  8, 0: Exactly one space required after comma
MOVE_VECTORS = {'k' : [[0,1], [1,0], [-1,0], [0,-1], [-1,-1], [1,-1], [-1,1], [1,1]],
                                                        ^ (bad-whitespace)
C:  8, 0: Exactly one space required after comma
MOVE_VECTORS = {'k' : [[0,1], [1,0], [-1,0], [0,-1], [-1,-1], [1,-1], [-1,1], [1,1]],
                                                                ^ (bad-whitespace)
C:  8, 0: Exactly one space required after comma
MOVE_VECTORS = {'k' : [[0,1], [1,0], [-1,0], [0,-1], [-1,-1], [1,-1], [-1,1], [1,1]],
                                                                         ^ (bad-whitespace)
C:  8, 0: Exactly one space required after comma
MOVE_VECTORS = {'k' : [[0,1], [1,0], [-1,0], [0,-1], [-1,-1], [1,-1], [-1,1], [1,1]],
                                                                                ^ (bad-whitespace)
C:  9, 0: Exactly one space required after comma
                'q' : [[0,1], [1,0], [-1,0], [0,-1], [-1,-1], [1,-1], [-1,1], [1,1]],
                         ^ (bad-whitespace)
C:  9, 0: Exactly one space required after comma
                'q' : [[0,1], [1,0], [-1,0], [0,-1], [-1,-1], [1,-1], [-1,1], [1,1]],
                                ^ (bad-whitespace)
C:  9, 0: Exactly one space required after comma
                'q' : [[0,1], [1,0], [-1,0], [0,-1], [-1,-1], [1,-1], [-1,1], [1,1]],
                                        ^ (bad-whitespace)
C:  9, 0: Exactly one space required after comma
                'q' : [[0,1], [1,0], [-1,0], [0,-1], [-1,-1], [1,-1], [-1,1], [1,1]],
                                               ^ (bad-whitespace)
C:  9, 0: Exactly one space required after comma
                'q' : [[0,1], [1,0], [-1,0], [0,-1], [-1,-1], [1,-1], [-1,1], [1,1]],
                                                        ^ (bad-whitespace)
```

Figure 2: Absent after-comma space warnings from Pylint

There were also several warnings about unnecessary parentheses in the if-statement conditions and lack of spaces surrounding logical operator “==”. Pylint does not seem to accommodate the change of **print** from statement to a function in Python 3 and warns about “unnecessary” parentheses after **print**. So I had to add a special flag in the configuration file for Pylint in order to disable the warning.

Quite a few errors were about global constants’ names not being capitalized. This was corrected. Another common naming error was about variable and function names. It turns out that Pyint checks whether the variables and functions are snake case by default. All of mine follow the camel case pattern, and I was not willing to change it, so in the subsequent runs of Pylint I changed the configuration to accept camel case as well.

An important correction that Pylint suggested is to change “==” to “is” when checking whether an element of a matrix is None or checking whether a Boolean value if True/False.

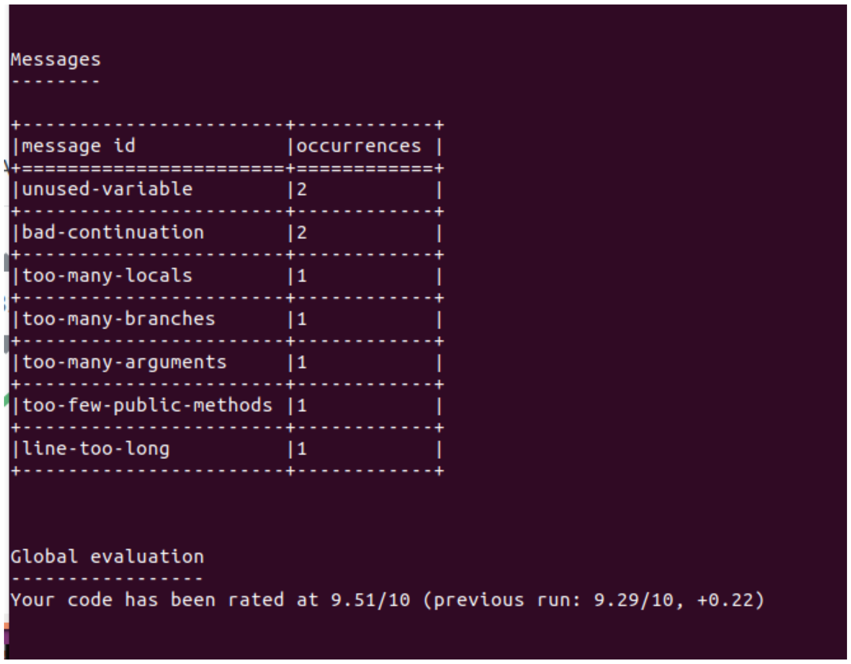
Pylint also helped identify unused code. There were several functions that I started writing but then changed my mind about implementation. The functions were not used anymore but they were still left in the code. Pylint directed my attention to them, as well as unused imports. Also I realized that I had a duplicate “inbounds” function, so I got rid of the extraneous copy.

Another source of warnings is that I did not provide docstring for my functions. I then added a Google-style docstring to each function.

One big problem that Pylint helped uncover is a typo in a variable that referred to the list of black pieces. With the typo, validation of moves for the black pieces would have been erroneous in some cases.

There were also several lines where Pylint pointed out a more elegant way of coding. Instead of returning True or False based on evaluation of condition for boundary checking, for example, Pylint suggested that I return **bool(condition)**. This resulted in reduction of about 5 lines without loss of readability.

After somewhere between 10 and 15 re-runs of static analysis, the following statistics were achieved:



```
Messages
--------

+-----------------------+------------+
|message id             |occurrences |
+=======================+============+
|unused-variable        |2           |
+-----------------------+------------+
|bad-continuation       |2           |
+-----------------------+------------+
|too-many-locals        |1           |
+-----------------------+------------+
|too-many-branches      |1           |
+-----------------------+------------+
|too-many-arguments     |1           |
+-----------------------+------------+
|too-few-public-methods |1           |
+-----------------------+------------+
|line-too-long          |1           |
+-----------------------+------------+



Global evaluation
-----------------
Your code has been rated at 9.51/10 (previous run: 9.29/10, +0.22)
```

Figure 3: Final statistics for static analysis

There are only 1 or 2 warnings in each category remaining, and I was not willing to rewrite logic or further modularize functions just to get rid of these. 9.51/10 versus -0.97/10 is still a significant improvement.

I am providing the untouched code file before static analysis. It is called “before_static_analysis.py”, and the full Pylint report is provided in the file called “Pylint_report”

Static analysis not only uncovered inconsistencies in my code's naming conventions and spacing; it also showed me where several bugs were hiding. It would be quite difficult to find these without Pylint.

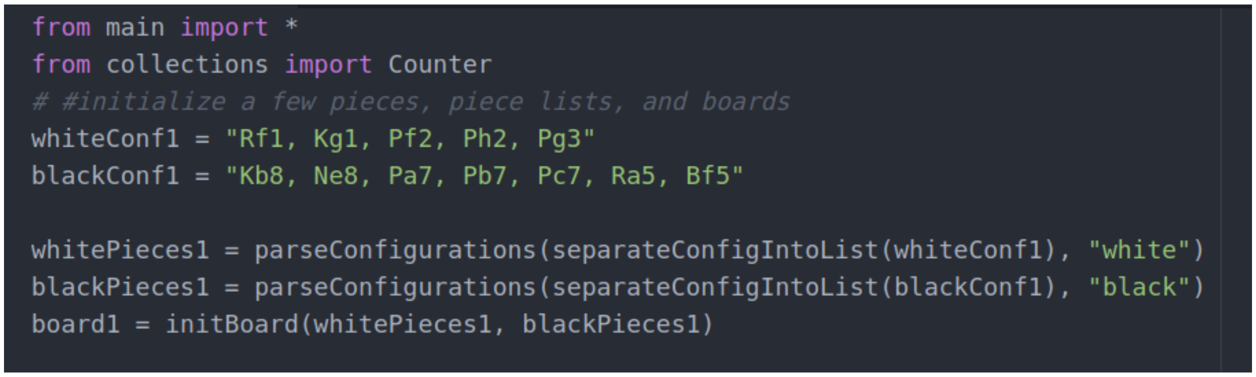
Part 2: Code coverage

For code coverage, I used Pytest with the coverage plugin. All of the tests for the program are in the file called "chess_tests.py". In order to run the tests, execute the following command:

py.test --cov-report term --cov-branch --cov=. chess_tests.py

The --cov-branch flag enables branch coverage analysis.

For testing purposes, five boards are created using the lines similar to the following:



```
from main import *
from collections import Counter
# #initialize a few pieces, piece lists, and boards
whiteConf1 = "Rf1, Kg1, Pf2, Ph2, Pg3"
blackConf1 = "Kb8, Ne8, Pa7, Pb7, Pc7, Ra5, Bf5"

whitePieces1 = parseConfigurations(separateConfigIntoList(whiteConf1), "white")
blackPieces1 = parseConfigurations(separateConfigIntoList(blackConf1), "black")
board1 = initBoard(whitePieces1, blackPieces1)
```

The following images show configurations of each of the five boards used for testing:

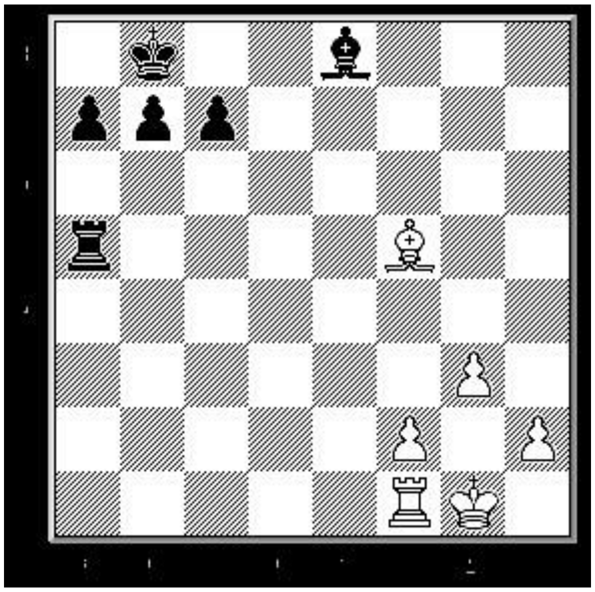


Figure 4: Test Board 1

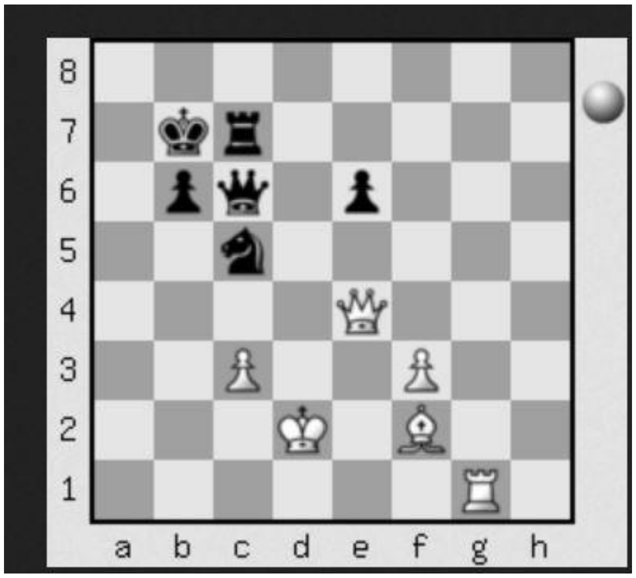


Figure 5: Test board 2

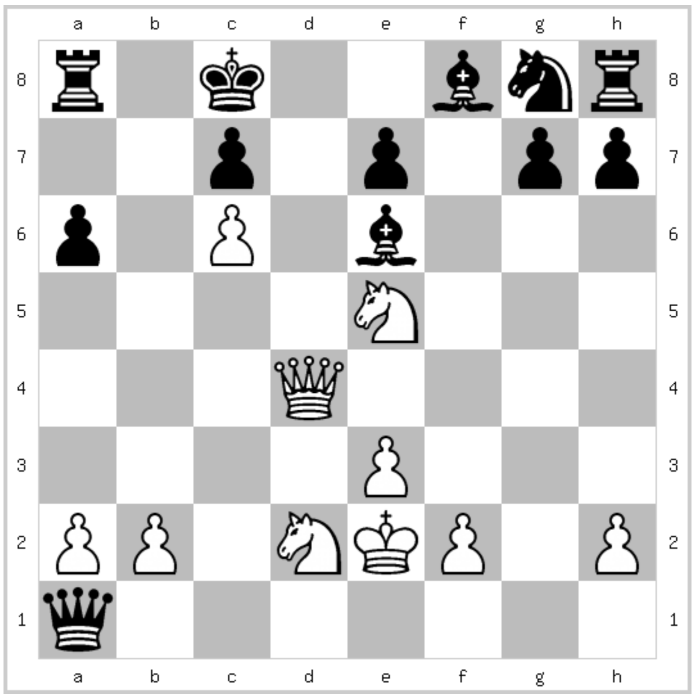


Figure 6: Test board 3

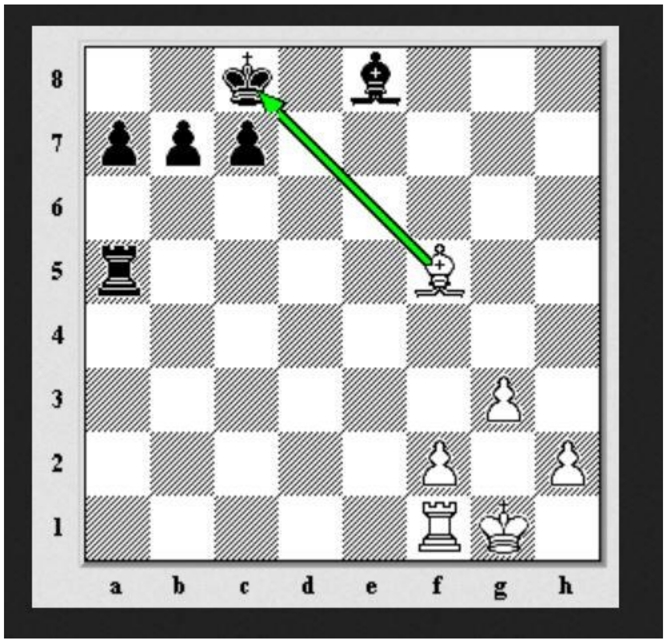
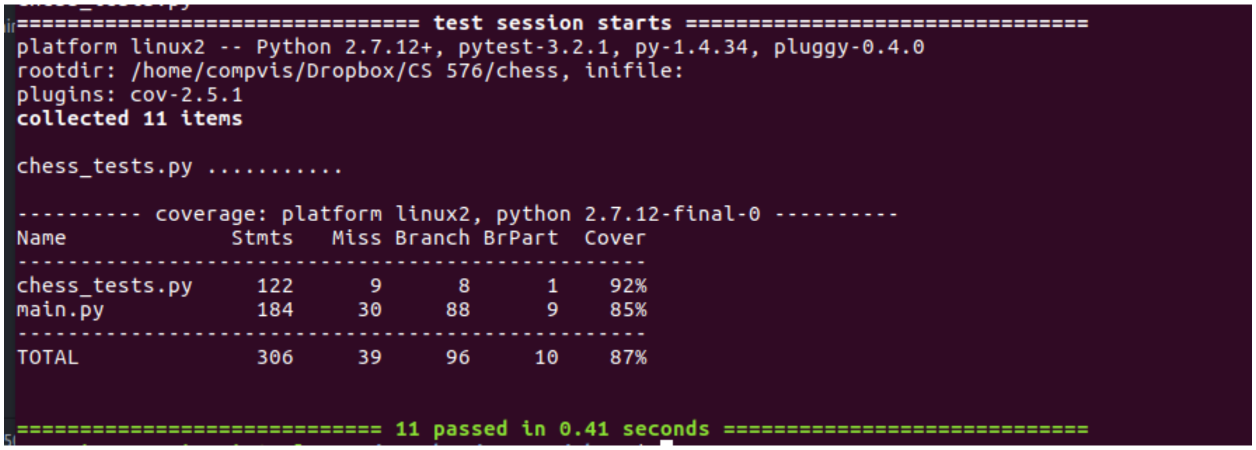


Figure 7: Test board 5

There is no diagram for test board 4, it is extremely simple. The board, along with board 5, is used to verify checks.

Various possibilities were tested for the game situations as well as individual functions in the code. The final code coverage achieved was 85%.



```
================================ test session starts ================================
platform linux2 -- Python 2.7.12+, pytest-3.2.1, py-1.4.34, pluggy-0.4.0
rootdir: /home/compvis/Dropbox/CS 576/chess, inifile:
plugins: cov-2.5.1
collected 11 items

chess_tests.py ...........

---------- coverage: platform linux2, python 2.7.12-final-0 ----------
Name              Stmts   Miss Branch BrPart  Cover
---------------------------------------------------
chess_tests.py      122      9      8      1    92%
main.py             184     30     88      9    85%
---------------------------------------------------
TOTAL               306     39     96     10    87%


============================= 11 passed in 0.41 seconds =============================
```

Figure 8: Code coverage results (all tests passed)

The line with **chess_tests.py** is to be disregarded. Test results are reported for all **.py** files in the current directory.

Conclusion:

This project involved thinking. There are many details in the game of chess that come naturally to humans but must be explicitly programmed on a computer.

Static analysis part of the project showed me that I was using an inconsistent style. It helped me achieve more consistency in naming and padding, as well as pointed out several instances of unused variables – which turned out to be hidden bugs in the code.

The code coverage and testing taught me how to test my functions, which I have never done before. In addition, it showed one of the drawbacks of using non-pure functions that modify global variables: they introduce problems when importing code into another file and testing. This encouraged me to rewrite some of my functions to make them pure and better modularize my code.

Files submitted:

main.py – move validation engine

moves.txt – text file with moves from the example in the assignment instructions. Used to execute the following command: **python3 main.py < moves.txt**

backup_before_static.py – code before any changes suggested by static analysis were made

Pylint_first_report – first report of the static analysis tool

Pylint_report – final report of the static analysis tool

chess_tests.py – tests for the engine. Uses py.test

code_coverage_report – final code coverage report