CS1002 W09 Practical 170003011 Tutor: Terzic 17 November 2019

Overview

The practical essentially required the construction of an emulator of the well known board game draughts. It was supposed to use a textual representation of the board and characters and represent their positions through the use of arrays. Moves were only to be permitted if they were legal (i.e. diagonal,with any possible captures taken). Functionality for kings who can move both diagonally forwards and backwards was implemented. I attempted to complete all 4 parts, but only parts 1-3 were completed successfully.

Design

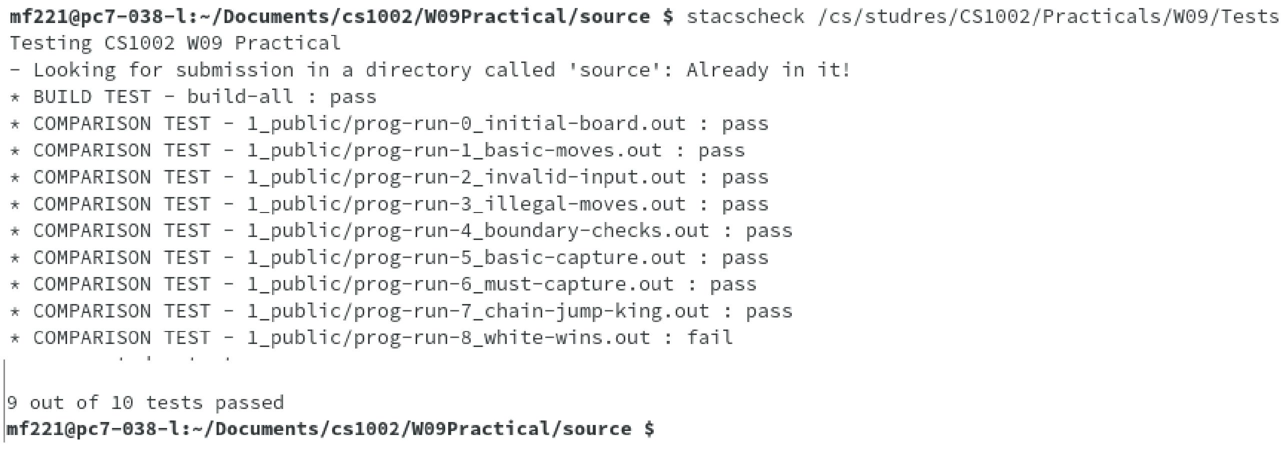
My program consists of 3 classes: Board, Game and W09Practical. The majority of my code was written in the board class as this was where the board array was created. This meant that I could easily access it when checking whether moves were legal etc. The main piece of programming in the Game class was a do-while loop which alternates between the players' turns. This loop was used as it allowed the game to be continue until game over. The main method was very simple; it just called the other 2 classes, and ran the do-while loop as instructed by the practical specification. Booleans were heavily employed throughout this project, in attempts to constantly check whether conditions were met. Arrays were used to return the indices of pieces between the Board and Game classes. The indices for these arrays were obtained by updating ascii values of their characters. I tried to make lots of methods which each do only a small amount of work - the idea was to reduce the likelihood of something going wrong this way by separating them. The drawback however, was that if something did go wrong it was difficult to locate the source of the problem – hence my trouble with part 4. I used the final constant DEFAULT_SIZE throughout my code rather than magic numbers to make the board size scalable, with a for loop initialising pieces relative to the edges of the board rather than at specific row numbers. Victory was detected when no more of the other player's pieces were left on the board. Following the link https://www.mastersofgames.com/rules/draughts-rules.htm , I saw that draughts can finish at any time by mutual agreement between the players. This was achieved through the checking for quit mentioned earlier.

Testing

Stacscheck was reasonably satisfied with my program. 9/10 tests were passed – I believe the final test was down to my program not recognising that players should get another turn if they have just successfully taken a piece and another piece is available to be taken.

The stacscheck output is given on the next page.

To check that other facets of the game were functional, I tried playing a few moves through. *Figure 2* and *Figure 3* show what happens when a player mistakenly tries to go outside of their turn, the consequent error message and then the repetition of their turn. This was good as it means that even if people make mistakes in their input, the game can continue.



```
mf221@pc7-038-l:~/Documents/cs1002/W09Practical/source $ stacscheck /cs/studres/CS1002/Practicals/W09/Tests
Testing CS1002 W09 Practical
- Looking for submission in a directory called 'source': Already in it!
* BUILD TEST - build-all : pass
* COMPARISON TEST - 1_public/prog-run-0_initial-board.out : pass
* COMPARISON TEST - 1_public/prog-run-1_basic-moves.out : pass
* COMPARISON TEST - 1_public/prog-run-2_invalid-input.out : pass
* COMPARISON TEST - 1_public/prog-run-3_illegal-moves.out : pass
* COMPARISON TEST - 1_public/prog-run-4_boundary-checks.out : pass
* COMPARISON TEST - 1_public/prog-run-5_basic-capture.out : pass
* COMPARISON TEST - 1_public/prog-run-6_must-capture.out : pass
* COMPARISON TEST - 1_public/prog-run-7_chain-jump-king.out : pass
* COMPARISON TEST - 1_public/prog-run-8_white-wins.out : fail

9 out of 10 tests passed
mf221@pc7-038-l:~/Documents/cs1002/W09Practical/source $
```

Figure 1: Stacscheck output

Figure 1 shows the main problem with my code. It didn't recognise that a piece should get another turn if chain jumps were possible. This main problem was carried through in the other stacscheck test leading to its failure. Creating the board was scalable as mentioned previously through the use of a for loop in the initialisation. Stacscheck also proves that valid inputs are only those specified on the board.

Figure 2 shows that my program understands whose turn it is and recognises which moves are available to the player on their turn. If these moves don't match with the inputs by the players then the error message gets printed.

Figure 3 shows what happens in various scenarios. When the inputs are out of bounds it prints an error message. When you try to move to a square that is already occupied by another piece, it prints an error message. It also shows how captures are forced to be made if they are available.

Figure 4 shows what happens when you try to enter 2 valid inputs, but they are in the same place – obviously you cannot move to the position you're trying to move out of, so this needed to be accounted for.

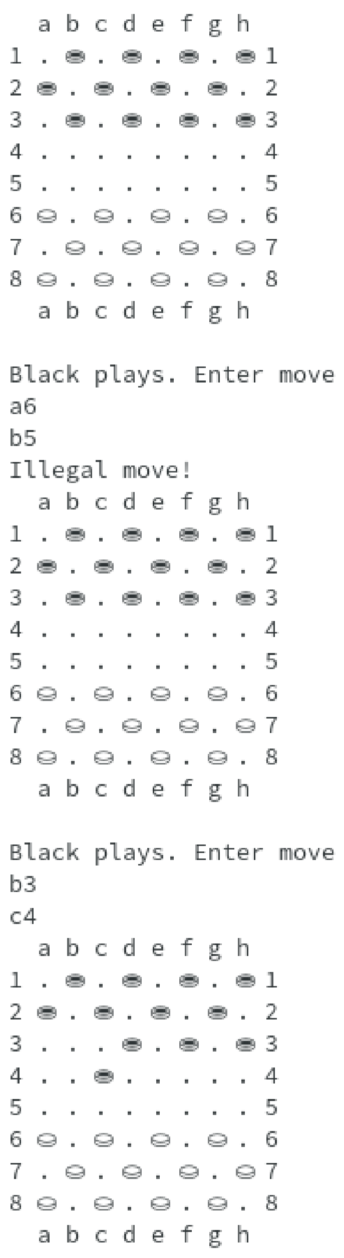
Figure 5 shows that when you are taking pieces, the same conditions hold for legality i.e. you still must move to an unoccupied square otherwise the move is invalid.

<u>Evaluation</u>

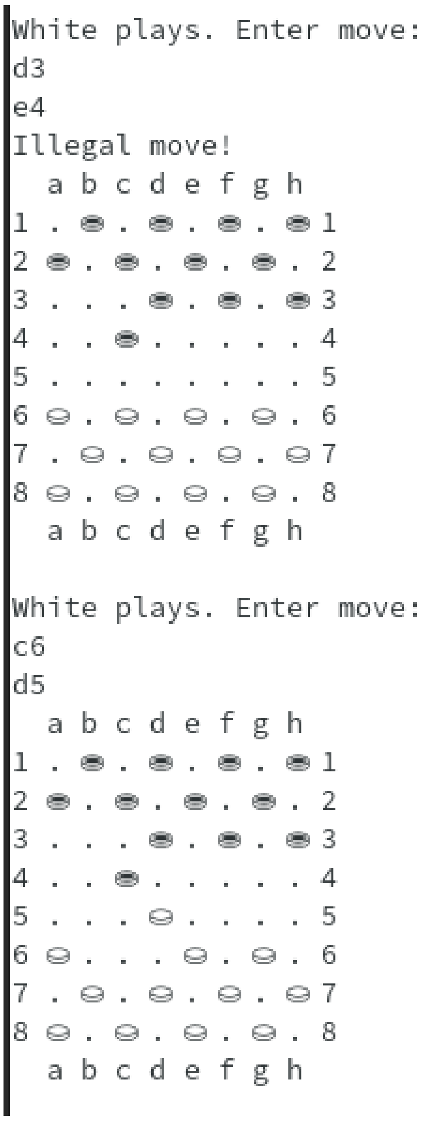
My program was reasonably successful as it completed the first 3 parts quite well. It was however an extremely long, quite complicated program with segments of code which were very, very similar. So I am quite certain there is a much cleaner solution which supports full functionality of the game. My program reached the same stacscheck success with about 30 lines less code (the code which I added trying to get the chain jumps to work properly), but I left them in to show which kind of lines I was thinking along. I feel like my logic was okay, I just didn't successfully implement it. I wasn't happy with the complexity of my program, it seemed way too long and there were so many things going on that it took a long time to try and debug, however I am not too disappointed with the output. Simplified games still seemed to work pretty nicely.

Conclusion

This practical was without question the most difficult so far. Due to extraneous circumstances, I didn't get to start it until much later than I typically would have liked. I received an extension, which gave me a little breathing room, and allowed me to produce the partial functionality that I did. However, if I had more time I would have approached this practical totally differently. I would have started with a pen and paper and not gone near the computer until I had my plan clear. I would have broken down the problem into a series of smaller problems (much smaller than the 4 parts given in the specification) and tackled each one in turn, rather than trying to solve everything at once. I achieved reasonable functionality with my program, pieces moved, only legal inputs were accepted, kings were recognised and their move abilities were updated accordingly. However, the chaining jumps didn't work as I hoped it would. If I had more time, I would try and debug this issue with a fine tooth comb to achieve the full functionality – I don't doubt that within the code I have written I already have the recipe for the fully functioning game, it would just be a matter of finding this final bug. Each stage presented its own difficulties – part 1 was probably the biggest hurdle as I had no idea where to start, I'd never heard of ascii values until I did some googling, so without their help to create indices then none of the rest of the functionality would have been possible. Part 4 was the other major source of difficulty, I managed to make captures (where possible) the only legal move and I know that if I had some more time I could get to the bottom of the chain-jumping problem. I achieved the majority of the desired functionality, I think there was just one issue that impeded full functionality – this issue of chain jumps. I tried to work out this problem right up until the moment I HAD to submit it! All in all, I achieved the majority of the desired functionality. It wasn't perfect, but I think I made a good attempt, and successfully completed the major aims of the practical. I feel less confused by arrays than when I started, and have gained quite a lot from the practical.

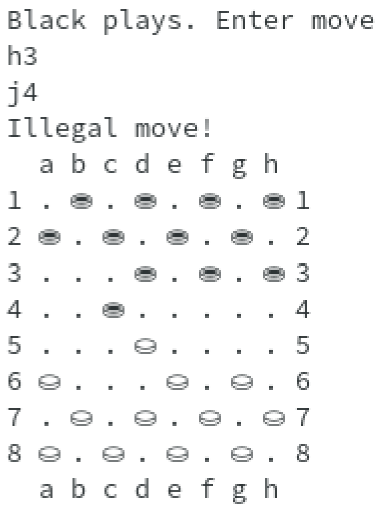


```
  a b c d e f g h
1 . ⛂ . ⛂ . ⛂ . ⛂1
2 ⛂ . ⛂ . ⛂ . ⛂ . 2
3 . ⛂ . ⛂ . ⛂ . ⛂3
4 . . . . . . . . 4
5 . . . . . . . . 5
6 ⛀ . ⛀ . ⛀ . ⛀ . 6
7 . ⛀ . ⛀ . ⛀ . ⛀7
8 ⛀ . ⛀ . ⛀ . ⛀ . 8
  a b c d e f g h

Black plays. Enter move
a6
b5
Illegal move!
  a b c d e f g h
1 . ⛂ . ⛂ . ⛂ . ⛂1
2 ⛂ . ⛂ . ⛂ . ⛂ . 2
3 . ⛂ . ⛂ . ⛂ . ⛂3
4 . . . . . . . . 4
5 . . . . . . . . 5
6 ⛀ . ⛀ . ⛀ . ⛀ . 6
7 . ⛀ . ⛀ . ⛀ . ⛀7
8 ⛀ . ⛀ . ⛀ . ⛀ . 8
  a b c d e f g h

Black plays. Enter move
b3
c4
  a b c d e f g h
1 . ⛂ . ⛂ . ⛂ . ⛂1
2 ⛂ . ⛂ . ⛂ . ⛂ . 2
3 . . . ⛂ . ⛂ . ⛂3
4 . . ⛂ . . . . . 4
5 . . . . . . . . 5
6 ⛀ . ⛀ . ⛀ . ⛀ . 6
7 . ⛀ . ⛀ . ⛀ . ⛀7
8 ⛀ . ⛀ . ⛀ . ⛀ . 8
  a b c d e f g h
```

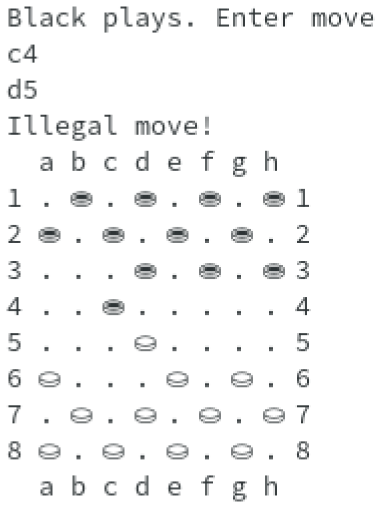


```
White plays. Enter move:
d3
e4
Illegal move!
  a b c d e f g h
1 . ⛂ . ⛂ . ⛂ . ⛂1
2 ⛂ . ⛂ . ⛂ . ⛂ . 2
3 . . . ⛂ . ⛂ . ⛂3
4 . . ⛂ . . . . . 4
5 . . . . . . . . 5
6 ⛀ . ⛀ . ⛀ . ⛀ . 6
7 . ⛀ . ⛀ . ⛀ . ⛀7
8 ⛀ . ⛀ . ⛀ . ⛀ . 8
  a b c d e f g h

White plays. Enter move:
c6
d5
  a b c d e f g h
1 . ⛂ . ⛂ . ⛂ . ⛂1
2 ⛂ . ⛂ . ⛂ . ⛂ . 2
3 . . . ⛂ . ⛂ . ⛂3
4 . . ⛂ . . . . . 4
5 . . . ⛀ . . . . 5
6 ⛀ . . . ⛀ . ⛀ . 6
7 . ⛀ . ⛀ . ⛀ . ⛀7
8 ⛀ . ⛀ . ⛀ . ⛀ . 8
  a b c d e f g h
```

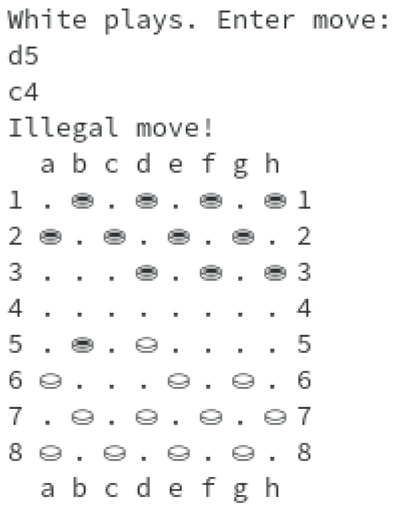
Figure 2: When wrong player tries to play and result



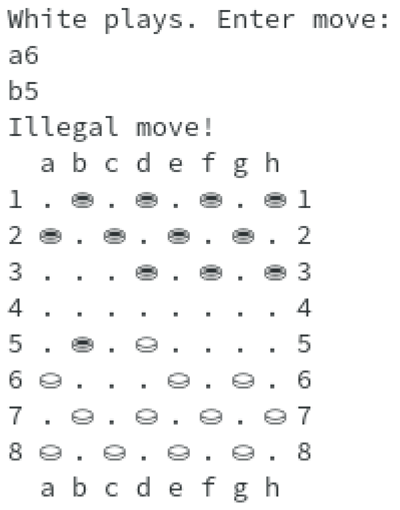
```
Black plays. Enter move
h3
j4
Illegal move!
  a b c d e f g h
1 . ⛂ . ⛂ . ⛂ . ⛂1
2 ⛂ . ⛂ . ⛂ . ⛂ . 2
3 . . . ⛂ . ⛂ . ⛂3
4 . . ⛂ . . . . . 4
5 . . . ⛀ . . . . 5
6 ⛀ . . . ⛀ . ⛀ . 6
7 . ⛀ . ⛀ . ⛀ . ⛀7
8 ⛀ . ⛀ . ⛀ . ⛀ . 8
  a b c d e f g h
```



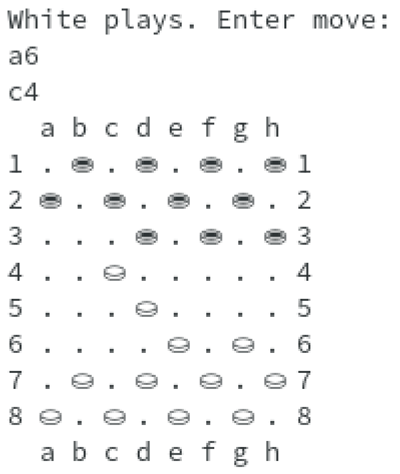
```
Black plays. Enter move
c4
d5
Illegal move!
  a b c d e f g h
1 . ⛂ . ⛂ . ⛂ . ⛂1
2 ⛂ . ⛂ . ⛂ . ⛂ . 2
3 . . . ⛂ . ⛂ . ⛂3
4 . . ⛂ . . . . . 4
5 . . . ⛀ . . . . 5
6 ⛀ . . . ⛀ . ⛀ . 6
7 . ⛀ . ⛀ . ⛀ . ⛀7
8 ⛀ . ⛀ . ⛀ . ⛀ . 8
  a b c d e f g h
```



```
White plays. Enter move:
d5
c4
Illegal move!
  a b c d e f g h
1 . ⛂ . ⛂ . ⛂ . ⛂1
2 ⛂ . ⛂ . ⛂ . ⛂ . 2
3 . . . ⛂ . ⛂ . ⛂3
4 . . . . . . . . 4
5 . ⛂ . ⛀ . . . . 5
6 ⛀ . . . ⛀ . ⛀ . 6
7 . ⛀ . ⛀ . ⛀ . ⛀7
8 ⛀ . ⛀ . ⛀ . ⛀ . 8
  a b c d e f g h
```

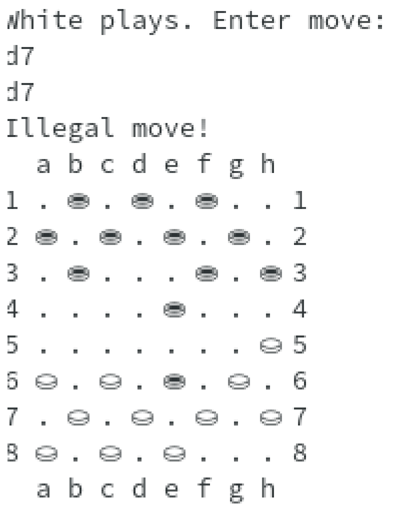


```
White plays. Enter move:
a6
b5
Illegal move!
  a b c d e f g h
1 . ⛂ . ⛂ . ⛂ . ⛂1
2 ⛂ . ⛂ . ⛂ . ⛂ . 2
3 . . . ⛂ . ⛂ . ⛂3
4 . . . . . . . . 4
5 . ⛂ . ⛀ . . . . 5
6 ⛀ . . . ⛀ . ⛀ . 6
7 . ⛀ . ⛀ . ⛀ . ⛀7
8 ⛀ . ⛀ . ⛀ . ⛀ . 8
  a b c d e f g h
```



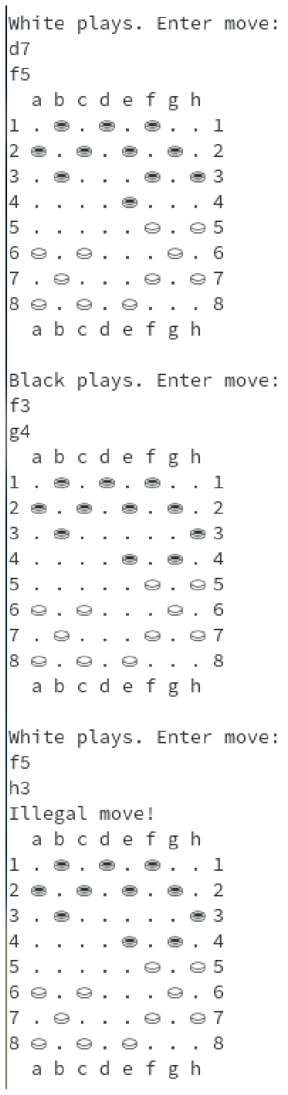
```
White plays. Enter move:
a6
c4
  a b c d e f g h
1 . ⛂ . ⛂ . ⛂ . ⛂1
2 ⛂ . ⛂ . ⛂ . ⛂ . 2
3 . . . ⛂ . ⛂ . ⛂3
4 . . ⛀ . . . . . 4
5 . . . ⛀ . . . . 5
6 . . . . ⛀ . ⛀ . 6
7 . ⛀ . ⛀ . ⛀ . ⛀7
8 ⛀ . ⛀ . ⛀ . ⛀ . 8
  a b c d e f g h
```

Figure 3: illegal inputs and consequences both colours and force captures in action



```
White plays. Enter move:
d7
d7
Illegal move!
  a b c d e f g h
1 . ● . ● . ● . . 1
2 ● . ● . ● . ● . 2
3 . ● . . . ● . ● 3
4 . . . . ● . . . 4
5 . . . . . . . ⊖ 5
6 ⊖ . ⊖ . ● . ⊖ . 6
7 . ⊖ . ⊖ . ⊖ . ⊖ 7
8 ⊖ . ⊖ . ⊖ . . . 8
  a b c d e f g h
```

Figure 4: when inputs are the same



```
White plays. Enter move:
d7
f5
  a b c d e f g h
1 . ● . ● . ● . . 1
2 ● . ● . ● . ● . 2
3 . ● . . . ● . ● 3
4 . . . . ● . . . 4
5 . . . . . ⊖ . ⊖ 5
6 ⊖ . ⊖ . . . ⊖ . 6
7 . ⊖ . . . ⊖ . ⊖ 7
8 ⊖ . ⊖ . ⊖ . . . 8
  a b c d e f g h

Black plays. Enter move:
f3
g4
  a b c d e f g h
1 . ● . ● . ● . . 1
2 ● . ● . ● . ● . 2
3 . ● . . . . . ● 3
4 . . . . ● . ● . 4
5 . . . . . ⊖ . ⊖ 5
6 ⊖ . ⊖ . . . ⊖ . 6
7 . ⊖ . . . ⊖ . ⊖ 7
8 ⊖ . ⊖ . ⊖ . . . 8
  a b c d e f g h

White plays. Enter move:
f5
h3
Illegal move!
  a b c d e f g h
1 . ● . ● . ● . . 1
2 ● . ● . ● . ● . 2
3 . ● . . . . . ● 3
4 . . . . ● . ● . 4
5 . . . . . ⊖ . ⊖ 5
6 ⊖ . ⊖ . . . ⊖ . 6
7 . ⊖ . . . ⊖ . ⊖ 7
8 ⊖ . ⊖ . ⊖ . . . 8
  a b c d e f g h
```

Figure 5: output when trying to take by moving into an occupied square