COMP30024 Artificial Intelligence

Project Part A:
Massacre

Sarah Erfani, Matt Farrugia, Chris Leckie
Last updated: 12 March 2018

In this part of the project, you will design and implement a Python program to analyse a board configuration for the game of *Watch Your Back!*. Before you read this specification, please make sure you have carefully read the "Rules of the Game" document.

The aims for Project Part A are for you and your project partner to refresh and extend your Python programming skills, explore some of the new algorithms we have met so far, and become more familiar with the mechanics of *Watch Your Back!*. This is also a chance for you to invest some time developing fundamental Python tools for working with the game: some of the functions and classes you create now may be helpful later when you are building your game playing agent for Project Part B.

Task

You must write a Python 3.6 program that reads a text-based board configuration from input, and depending on a further input command, calculates one of the following:

- The **number of available moves** for each player, or
- a **sequence of moves** for White pieces that would eliminate all Black pieces assuming the Black pieces are unable to move.

The input format, the detail of these calculations, and the output format are explained below.

Board input format

The first thing your program should do is read the board configuration from **standard input** (that is, the default input source accessible through the built-in function `input()`). The first 8 lines of standard input will contain a text-based representation of a board with four different symbols representing the different possible contents of each square:

- The character "-" (hyphen) is used to mark **unoccupied** squares
- The character "X" (capital *x*) is used to mark **corner** squares
- The character "O" (capital *o*) is used to mark squares containing a **White piece**
- The character "@" (*at* sign) is used to mark squares containing a **Black piece**

Each line of input corresponds to one row of the board. Within each line, each symbol is separated by a single whitespace character. Each line is terminated by a single line-feed character. You may assume that input will always be of the correct format and size.

The order of symbols in input corresponds directly to the coordinate positions of the board squares they represent. That is, the first three symbols on the first line represent squares (0,0), (1,0) and (2,0) respectively, the first three symbols on the second line represent squares (0,1), (1,1) and (2,1) respectively, and so on. See figure 1 for a complete example.

8 lines of standard input:

```
X - - - 0 0 - X
- - - 0 - 0 0 -
0 - 0 - - - - @
- - - 0 0 0 - @
0 0 0 - - @ @ -
- @ @ - @ @ @ -
- @ @ @ - - - -
X - - - - - - X
```

	0	1	2	3	4	5	6	7
0	**X**				**W**	**W**		**X**
1				**W**		**W**	**W**	
2	**W**		**W**					
3				**W**	**W**	**W**		**B**
4	**W**	**W**	**W**			**B**	**B**	
5		**B**	**B**		**B**	**B**	**B**	
6		**B**	**B**	**B**				
7	**X**							**X**

Figure 1: an example input (left) and the corresponding board configuration (right): Black pieces are marked with **B**, White pieces with **W**, and corner squares with **X**.

Analysing the board

Next, your program should read one more line of input. This 9th line will contain a string telling your program how it should analyse the board. The string will be either `"Moves"` or `"Massacre"` (both case-sensitive). Your program should respond as follows:

Calculating "Moves": If the 9th line of input is the string `"Moves"`, your program should count the total number of legal moves available for each player based on the board configuration read from input. That is, what is the total number of possible moves each player could make, according to the rules of the game, if it were their turn to move one of their pieces in the movement phase? This should include all regular moves and jumps available for all of the Player's pieces (even those which result in the piece being instantly eliminated).

Your program should print the resulting legal move counts to **standard output** (that is, the default output source accessible through the built-in function **`print()`**) on two lines. On the first line of standard output, your program should print the number of available moves for White player. On the second line, your program should print the number of available moves for Black player. Your program ***should not print anything else to standard output***. See figure 2 for an example.

Example of standard input:

```
X - - - - - - X
- - - - - - - -
- - - - - 0 0 -
- - - - @ 0 - -
- - - - - - - -
- - - - - 0 - -
- - - - @ - @ @
X - - - - - - X
Moves
```

Correct standard output:

```
16
13
```

Figure 2: An example of the desired output format for calculating "Moves".

Calculating "Massacre": If the 9th line of input is the string `"Massacre"`, your program should calculate a sequence of legal moves for White pieces that, if carried out starting from the given board configuration, would lead to all Black pieces being eliminated (assuming that the Black pieces do not move from their starting positions). That is, if White player were allowed an unlimited number of turns in a row, how could they manoeuvre their pieces in a way to eliminate all of Black player's pieces?

Your program should print the resulting sequence of moves to **standard output**, one 'move' per line, in the order the moves should take place. Each move should be formatted as two pairs (two 2-tuples): first, a pair representing the starting square of the piece to move, and then a pair representing the finishing square of the piece (the square it is on after the move). The two pairs should be in the format `"(column, row)"`, and they should be separated by whitespace and an arrow: `" -> "`. For example, the line `(5, 3) -> (5, 1)` would represent the move: *The White piece on square (5,3) jumps to square (5,1)*. See figure 3 for a complete example.

Example of standard input:

```
X - - - - - - - X
- - - - - - - - -
- - - - - O - - -
- - - - @ O - - -
- - - - - - O - -
- - - - - O @ - -
- - - - - - - - @
X - - - - - - - X
Massacre
```

One correct standard output:

```
(5, 3) -> (3, 3)
(5, 2) -> (5, 3)
(6, 4) -> (7, 4)
(7, 4) -> (7, 5)
```

Figure 3: An example of the desired output format for calculating "Massacre". Note the spacing within the lines and the order of the output lines.

Note that for this calculation there are multiple possible correct outputs for each input. Any sequence of moves that will eliminate all Black pieces is acceptable. In particular, the sequence does not have to be the shortest possible sequence of moves.

Analysing your program

Finally, you must briefly discuss the structure and complexity of each of the main parts of your program in a separate file called **comments.txt** (to be submitted alongside your program). Your discussion should answer the following questions:

- For calculating the number of available moves, what is the time complexity and space complexity of your program? You should not count the time taken to read the board configuration from input.
- For calculating the sequence of moves to eliminate all enemy pieces, how have you modelled the problem as a search problem? What search algorithm does your program use? If applicable, what is the branching factor of your search tree? What is the maximum search depth? Does your algorithm always return the shortest possible sequence of moves that will eliminate all enemy pieces?

implies there is DFS, but maybe no BFS

Assessment

Project Part A will be marked out of 8 points, and contribute 8% to your final mark for the subject. Of the 8 points:

- 2 points will be for the quality of your code: its structure (including good use of functions and classes) and readability (including good use of code comments).
- 4 points will be for the correctness of your program, based on testing your program on a set of test cases. 2 of these points will be allocated to each of the two calculations your program must be able to perform.
- 2 points will be for the accuracy and clarity of the discussion in your **comments.txt** file.

Note that even if you don't have a program that works correctly by the time of the deadline, you should submit anyway. You may be awarded some points for a reasonable attempt at the project.

You may make use of the library of classes provided by the AIMA textbook website if you wish, provided you make appropriate acknowledgements that you have made use of this library. Otherwise, for this part of the project, your program should not require any tools from outside the Python Standard Library.

Please note that questions and answers pertaining to the project will be available on the LMS and will be considered as part of the specification for the project.

Academic integrity

There should be one submission per group. You are encouraged to discuss ideas with your fellow students, but your program should be entirely the work of your group. It is not acceptable to share code between groups, nor to use the code of someone else. You should not show your code to another group, nor ask another group to look at their code. **If your program is found to be suspiciously similar to someone else's or a third party's software, or if your submission is found to be the work of a third party, you may be subject to investigation and, if necessary, formal disciplinary action.**

Please refer to the 'Academic Integrity' section of the LMS, and to the university's academic honesty website http://academichonesty.unimelb.edu.au/ if you need any further clarification on these points.

Submission

The submission deadline for Part A is 4.00pm Friday 30th March 2018. Please see the separate document, partA-submit-2018.pdf, to be released closer to the deadline, for detailed information on submission.