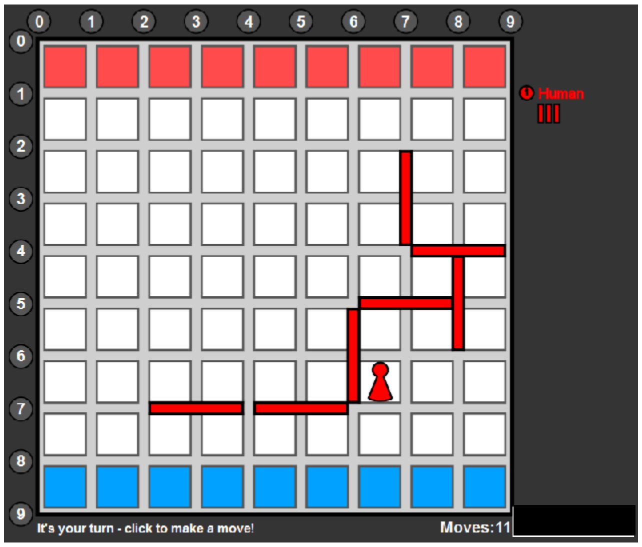
Data Structures for Problem Solving
Quoridor - Project Part 2

3/18/13

1 Problem

An important component of Part 2 of the Quoridor project is to figure out whether a wall is placed according to the rules of the game. As a student player, you will need to generate valid wall placements.

Consider the following board, with a single player who started at home position $(8, 4)$.



1. Consider a `PlayerMove` object that represents a wall placement (the `move` slot is `False`). What are the conditions one needs to determine if the move is valid?

2. Assume the above configuration as a starting point. Follow this sequence of wall placements and figure out whether the wall placement follows the rules of the game. If not, briefly explain why and do not add it to the board. The wall placements are given as r_1, c_1, r_2, c_2 specifying the start and end row/column coordinates of the wall.

   (a) 7,0,8,1
   (b) 3,3,3,4
   (c) 2,0,2,2
   (d) 8,8,10,8
   (e) 5,5,3,5
   (f) 0,0,0,2
   (g) 6,2,8,2
   (h) 6,3,8,3
   (i) 6,4,8,4
   (j) 4,6,6,6
   (k) 7,4,9,4

After you are finished with the above exercise, your group should break up into separate teams. Each team will hand in a separate solution to the remaining questions.

3. Discuss with your teammate what data structure(s) you will be using for your project. It could be one that one of you was using for part 1, or something new. You may want to consider the next two questions before deciding.

4. Sketch the algorithm that checks whether a given wall placement crosses any of the already placed walls.

5. Sketch the algorithm that checks whether a given wall placement blocks any of the pawns from reaching their destinations.

2 Implementation Deliverables for Students Developing Player Modules (80%)

2.1 Overview

The main objective of part 2 of the project for player module students is to be able to make moves that follow the rules of the game while being concerned neither about strategies nor about other players at all.

In the config file, you should set `PART_ONE` to `False`, and make sure your module is the only one specified for `PLAYER_MODULES`. Make sure that you are still able to run correctly even if valid `PRE_MOVE`s are specified!

Look at the `move` function template found in your `__init__.py` file. The function is given your player module's own game state (player data) and must return your player's choice for a move in a `PlayerMove` object. Notice that you are not returning your player data, therefore you should not be updating it inside the `move` function. (`last_move` will be called with your own move; see below.)

In general you only have to make sure to return a legal move, meaning a legal wall or pawn position. However, to make sure you are playing the game in a reasonable way and exercising all the logic we expect you to develop, there are some constraints.

1. Your player module must alternate between pawn moves and wall placements until you run out of walls.[1]
2. Your pawn must reach the goal line within 100 moves.
3. Each move your pawn makes must be along a shortest path to the goal line.

You have already written code for part 1. Here is what happens to those functions you have already developed.

- `get_neighbors`, although no longer called by the system, is most likely useful as an internal utility function for computing paths.
- `get_shortest_path` is no longer required, but again should be a valuable utility.
- `init` is still called, and will most likely need to be modified to at least record the location of your player's pawn, unless you already did this in part 1.
- `last_move` is still called by the system. In addition to pre-moves, it will also be called immediately after your `move` function is called so that you can record your own moves in your player data.

2.2 Grading

2.2.1 Grade Breakdown

- 40% correct functioning of the game without `PRE_MOVE`s
- 30% correct functioning of the game with arbitrary correct `PRE_MOVE`s added
- 5% style
- 5% submission instructions have been followed

1. This means that your player data has to maintain a Boolean flag or integer counter.

2.3 Submission

Now that you are in teams, *only one person should submit your assignment to the dropbox*. If you make this mistake and multiple teammates submit, please contact your SLI to let them know which one should be graded.

Once you have selected whose code you are going to work with, you should make a copy of the folder in Eclipse. Right click on the username folder, select `Copy`, then right click on the `StudentPlayers` folder and select `Paste`. You should rename the folder to your chosen team name. *Make sure that all the code you write or edit is inside your own folder!* To import other modules from within your own team folder, some special syntax is needed. Let's say you wrote a file named `mod.py` and from that file you need `x`, `y`, and `z`.

```
from .mod import x,y,z
```

or

```
from .mod import *
```

When you are ready to submit you should go through your file explorer window to the team name folder and create a zip file similar to how you did things in part 1.

Your submission is due by the deadline to the MyCourses dropbox. There is no late dropbox for project submissions. Make sure that you submit a `zip` file and not anything else in any other format (e.g. `rar`, `7z`, etc.). Do not submit your individual python files, they must be compressed into the zip file, whose top level is a single folder with your username.