CSC148, Assignment #2
due March 5th, 2015, 10 p.m.

deadline to declare your assignment team: February 26, 10 p.m.

overview

In assignment 2 you will build on the foundation laid in assignment 1 to add a new game, called tippy, and a new strategy, called minimax. Starter code that we provide is designed so that you should be able to "plug in" your new classes without re-writing existing code. You'll end up with a new game and a much stronger computer opponent. As well, you'll be able to play subtract square against the stronger computer opponent, or tippy against the existing weaker opponent.

In the process, you'll consolidate your ability to implement subclasses and recursive algorithms.

tippy

Tippy is a variation of tic-tac-toe. Players take turns placing either an **X** or an **O** on an $n \times n$ grid (where n is at least 3), with the goal of forming a tippy. In the four examples below, played on a 3×3 grid, **X** has won by forming a tippy:

X	X	O
O	X	X
O		

O	X	X
X	X	O
O		

X	O	O
X	X	
O	X	

O	O	X
	X	X
O	X	

Other tippies may be formed by taking one of the four above and shifting it on the grid, left or right, up or down. Tippies correspond to the green **Z** and red **S** tetrominos (AKA tetriminos) from tetris. On a 3×3 grid there are 8 possible tippies, and more on a larger grid.

Unlike tic-tac-toe there is a winning tippy strategy for whichever player moves first. In other words, if the first player always chooses the best move, she or he will always win the game of tippy. Compare this to the situation in tic-tac-toe where if both players choose the best possible move, the result is always a tie.

minimax

Minimax is a strong strategy that assumes that both players (let's call them **A** and **B**) have all the information and time they need to make the strongest possible move from any game state (AKA position). In order to choose the strongest possible move, players need a way to evaluate the best possible outcome, called the player's score, they can guarantee from each position.

If the game is over, **A**'s score is:

- 1 if **A** wins
- -1 if **A** loses
- 0 if it's a tie

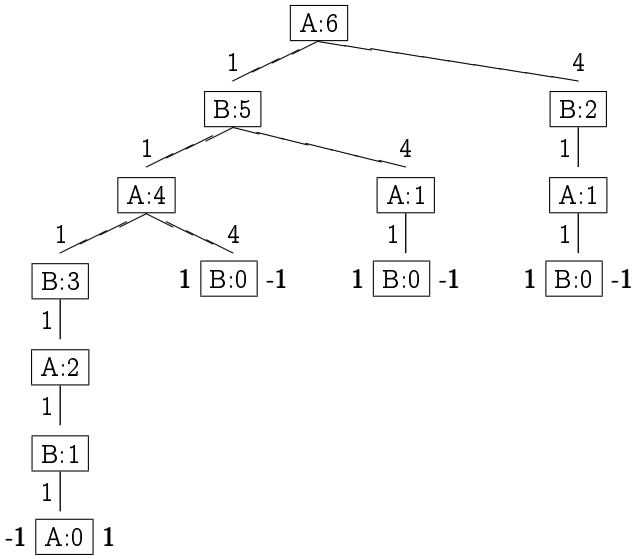
B's score is -1 times **A**'s score, since this is a zero-sum game: what's good for **A** is bad for **B**.

Suppose the game is not over, and the current player is **A**. Then **A**'s score is

- 1, if at least one **A**'s legal moves leads to a new position where **A**'s score is 1 (or **B**'s score is -1, since **B** is the current player in the new position)
- -1, if every legal move for **A** leads to a position where **A**'s score is -1 (or **B**'s score is 1, since **B** is the current player in the new position)
- 0, if at least one of **A**'s legal moves leads to a score of 0, but none lead to a score of 1.

This circular-sounding scoring process is guaranteed to provide an answer, assuming every sequence of moves eventually leads to the game ending. Indeed, it has a recursive solution, where the base case(s) corresponds to the end of the game.

Here's a diagram representing all sequences of moves for a game of Subtract Square, starting with value being 6 and current player **A**, so that position is labelled A:6. Leading out from that position are lines (edges) labelled with the possible moves, and then positions those moves lead to... and so on.



We have labelled the positions where the game is over with **A**'s score on the left, and **B**'s score on the right. Work up from those positions, writing **A**'s and **B**'s scores beside each position, until you reach the top.

You were on the right track if you ended up with a score of **1** for **A** at position A:6. You can score any position in the games we're working with using this technique. Here is an algorithm for determining the score for the current position if you are the next player (the one about to play):

- If the game is over there are no moves available, your score is either 1, 0, or -1, depending on whether you win, tie, or lose.
- If the game is not over, there are legal moves available. Consider each available move, and the position it takes the game to. Determine **your opponent**'s score in the new position (after all, your opponent is the next player in the new position). Multiply that score by -1 to determine your score in that position.

  Your highest possible score among all the new positions is your score for the current position.

Of course, you will also want to record the move that gets you the highest score, so you might as well bundle them together in some suitable data structure.

Notice that you and your opponent are each solving different instances of the same problem, what your score is. That's what makes the process recursive.

your job

- Read over the starter code we have provided, and copy it into a subdirectory where you will work on your code. This gives you the interface for the code you will write, as well as a text-based **Subtract Square** against a weak computer opponent.
- Implement class **TippyGameState**, a subclass of **GameState**, including overriding all unimplemented methods from the superclass. Your code should be able to represent the state of an $n \times n$ tippy game, for any n 3 or greater. You may get some insight by reading our implementation of **SubtractSquareState**. This is worth 45% of the assignment mark.
- Implement class **TippyMove**, a subclass of **Move**, to ensure that it works with your **TippyGameState**. You may get some insight by reading our implementation of **SubtractSquareMove**. This is worth 5% of the assignment mark.
- Implement class **StrategyMinimax**, a subclass of **Strategy**, and implement all unimplemented methods. This is worth 40% of the assignment.
- Add or modify a few (less than a dozen) lines of **game_view.py** so that your new game and strategy become available when a user evaluates this file in **Wing**, while the existing game (Subtract Square) and strategy remain. Do not change any other part of the starter code. This is worth 5% of the assignment mark.
- Follow CSC108 style guidelines unless your CSC148 instructors say otherwise. This includes running **pep8** on your *.py files and fixing the errors it reports:

  1. Save pep8.py in the same directory as your *.py files
  2. Open each of your *.py files in Wing, click the green arrow
  3. In Wing's Python shell import pep8
  4. Type: `pep8.Checker('some_file.py', ignore=('W2','W3')).check_all()`
     Of course, replace `some_file.py` with the name of the *.py file you are checking.

  This is worth 5% of the assignment mark.

Declaring your assignment team

You may do this assignment alone or in a team of either 2 or 3 students. Your partner(s) may be from any section of the course on St George campus. You must declare your team (even if you are working solo) using the MarkUs online system.

Navigate to the MarkUs page for the assignment and find "Group Information". If you are working solo, say so. If you are working with other(s):

First: one of you needs to "invite" the other(s) to be partners, providing MarkUs with their cdf user name(s).

Second: the invited student(s) must accept the invitation.

Important: there must be **only** one inviter, and other group members accept **after** being invited, if you want MarkUs to set up your group properly.

To accept an invitation, find "Group Information" on the appropriate Assignment page, find the invitation listed there, and click on "Join".

Submitting your work

Submit all your code on MarkUs by 10 p.m. March 5th. Click on the "Submissions" tab near the top. Click "Add a New File" and either type a file name or use the "Browse" button to choose one. Then click "Submit". You can submit a new version of a file later (before the deadline, of course); look in the "Replace" column.

Only one team member should submit the assignment. Because you declared your team, all of you will get credit for the work.

Be sure to hand in `tippy_game_state.py`, `tippy_move.py`, `game_view.py`, and `strategy_minimax.py`. Once you have submitted, check that you have submitted the correct version. There is a 5% deduction for each hour late any part of your submission is. A good strategy is to begin submitting the parts of your assignment that work **early**, and keep submitting improved versions as the deadline approaches. Only the last version of each file is graded.