Introduction to Game Trees

- Play Tic-Tac-Toe with your neighbor.
- Play Tic-Tac-Toe with your neighbor for exactly 5 moves (XOXOX), then draw the game tree under the current board.
- Score 1.0 for X win, -1.0 for O (or Y) win, 0.0 for a draw.
- Each node is a board as viewed by one of the players. It is eventually scored with 1.0, 0.0 or -1.0 depending on the best score obtainable for THAT player playing on that board against a perfect opponent.
Let's draw a game tree..
Games

- Artificial Intelligence mini-overview.
- General computer game loop.
- Tic-Tac-Toe Board class
  - Model of the playing board
  - Example of an instantiated class (methods and data members are NOT static.)
- Computer player
  - random
  - static evaluation
  - (recursive) backtracking search for best move.
Classical Artificial Intelligence

- Invent a *formal model* for a task that seems to require intelligence. (Mathematical, ... )
- Program a model simulator.
- Program an evaluator for a numeric measure of how good a solution or partial solution is.
- Program a move lister or generator.
- Apply a general purpose problem solving strategy (GPS-Newell&Simon, 1970's):
  - Search for a best move, use backtracking perhaps.
  - Model that move and repeat until happy.
Criticism – Rodney Brooks (80-90's)

- Triumphs for classical AI: Chess, diagnosis, led to expert systems, language parsers, etc.
  - IBM's Big Blue chess-playing machine sometimes beat world class champions. (1990's)
  - Task becomes “unintelligent” when an algorithmic solution becomes known.

- Brooks' “intelligent” mobile robots use direct interaction and combinations of simple response algorithms in place of an internal model. He invented “subsumption” (remember Kumar and our robot competition lab.)
We will use classical AI for Tic-Tac-Toe

- Board simulator: DONE for you following Kumar's Python version. (class Board)
  - Stores and prints the board, presents legal moves, changes stored board after doing a given move, tests whether the board is a win for one player, a draw, or can still be played.

- Game Loop and user interface: Also follows Kumar's Python version. (class TTTFramework)
  - The method for the computer move calls the method for the person's move.
def play():
    # Initialize board
    board = makeBoard()

    # set who moves first/next: X always moves first
    player = 'X'

    # Display the initial board
    display(board)

    # The game loop
    while (not gameOver(board)):
        move(board, player)
        display(board)
        player = opponent(player)

    # game over, show outcome
    winningPiece = winner(board)

    if winningPiece != 'Tie':
        print winningPiece, "won."
    else:
        print "It is a tie."

    Tic Tac Toe Rules

    Two players, O and X, take turns filling the squares in a 3×3 grid. X always going first. The player who succeeds in placing three pieces in a horizontal, vertical or diagonal row wins the game.
1. `makeBoard()`: Returns a fresh new board representing the start of the game. For Tic Tac Toe, this function will return an empty board representing the nine squares.

2. `displayBoard(board)`: Displays the board on the screen for the user to see. The display can be as simple or elaborate as you wish. It is good to start with the easiest one you can write. Later you can make it fancier.

3. `opponent(player)`: Returns the opponent of the current player/piece. In Tic Tac Toe, if the player is X, it will return an O, and vice versa.

4. `move(board, player)`: Updates the board by making one move for the player. If the player is the user, it will input the move from the user. If the player is the computer, it will decide how to make the best move. This is where the smarts will come in.

5. `gameOver(board)`: Returns `True` if there are no more moves left to be made, `False` otherwise.

6. `winner(board)`: Examines the board and returns the winning piece or that the game is not yet over, or that it is a tie. In Tic Tac Toe, it will return either an X, O, a blank (representing game is not over yet), or a TIE.
Your Assignment-3 parts

1. Program a computer move method that picks one move from the legal set randomly.

2. Program the static evaluator as described by Kumar. Program a loop that for each legal move,
   1. Make a copy of the current board.
   2. Do the legal move on the copy.
   3. Pick the first (or last) move that evaluates best for the computer player.

Make the computer move be that best move.
Classes to be instantiated

- Usually the programmer writes a constructor.
- Usually the data members and methods are NOT static.
- Non-static method calls (activations) are ALWAYS associated with one instance of the class.
- New instances are created by the new operator.
- MANY instances can co-exist in the computer.
My Board class.

- `new Board();` returns a new, blank board. Only one Board is used in the manually playing version TTTFramework.

- (Obviously?) You need to have a Board variable to refer to the top level Board:
  - `Board topLevelBoard = new Board();`

- Manual moves are ALL done on this board.

- (for parts 2 and 3 only) You must code
  - `Board trialBoard = someBd.copy();`

to get a copy on which to try tentative moves.
Tree illustrating static evaluation
Part 3: Backtracking Search

- A recursive method makes the computer's move.
- The activation tree will BE the game tree.
- The copy method of class Board is crucial so copies of the top level board (and copies of copies, etc.) can be made to try out moves (and move sequences!)
  - We don't want the move picker's calculations to mess up the current board.
Challenging Final Project..

- 50% of grade is based on a written report showing your progress in learning from the project work, including Java.

- Write it WHEN YOU DO THE PROJECT WORK.

- Projects this complex are normally assigned mid-semester in CSI310, but this is an honors course. You can still get an A even if you don't do part 3. (Note: UA's style is to make you code the infrastructure.)