Sequential Decision-Making	**F2019**
Problem Set 1	
Handed Out: Sep 16^{th}, 2019	*Due: Oct* 7^{th}, 2019

In this assignment you will implement a variety of techniques for game playing as well as solving stochastic bandit problems. Game playing agents will be tested on the two games provided: Connect Four (connect4.py) and Breakthrough (breakthrough.py). Bandit solving agents will be tested on two Bernoulli bandits defined using test0.txt and test1.txt. You will submit a writeup that summarizes your results and all code as a zip file. The writeup should be in AAAI format and, generally, be no longer than 3 to 4 pages. Your writeups can be longer, this is just meant to be a rough guideline. You can find the AAAI style files on Canvas. Submit the writeup (with attached source code) to the Canvas submission locker before 11:59pm on the due date.

To aid you in your quest I've provided some base code to get you started. The general structure of the code is similar for both game playing and bandit solving. A controller will instantiate your agents and run your experiments. Included is a set of command line arguments that can be used to customize your experiments. I have also provided functioning random agent for both game playing and bandit solving, as well as skeleton code for each other agent you will be asked to implement. While you are not required to use this, I think that some of you may find this code helpful. I apologize in advance for subjecting everyone to my generally terrible code-writing skills.

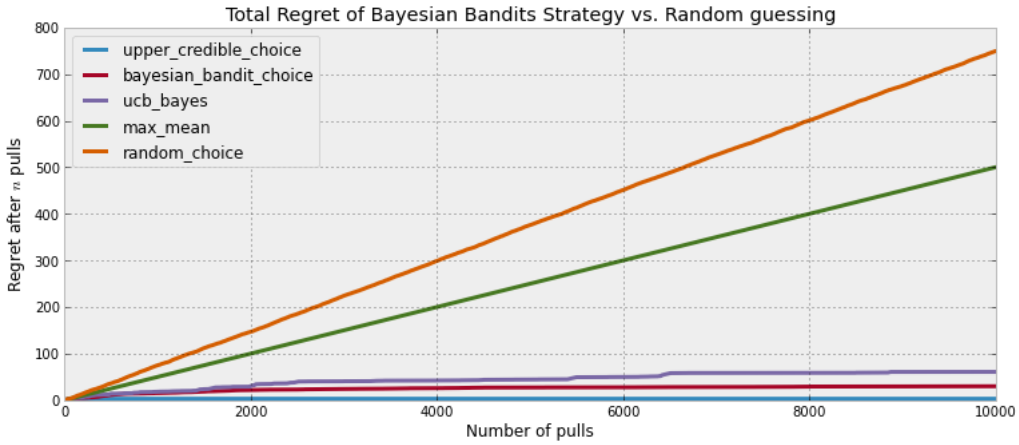
Bernoulli Bandits (20 points)

Your first task should be straight forward enough. I have provided (in the bandits-input folder) specifications (test0.txt and test1.txt) for two Bernoulli bandits with 10 arms each. Your task is to solve these two bandits using the approaches we've talked about in class. Specifically, use ϵ-greedy, UCB1, and Thompson sampling to maximize your reward gained for these problems over 10,000 steps. In addition, you should compare the performance of

each approach by comparing the cumulative regret achieved by each agent on each problem. In your writeup these should be presented as graphs.

Details:

- I am sure there is a library somewhere that has some of these approaches already implemented. Do not use them. Pretty much anything else should be fine. If you are not sure if a certain library is allowable, just ask and I will let you know.
- As mentioned at the start of class, all code should be written in Python 3.
- For ϵ-greedy feel free to hold ϵ constant or using a decaying schedule. Be sure to discuss the performance of your agent with respect to this choice.
- Explain any implementation choices you had to make in the final report. This includes the motivation behind using certain data structures or using certain libraries.
- Include a comparison of the regret gained over time for each algorithm. You should end up with a graph that looks something like the following figure.



Minimax Search (25 points)

Now let's get to some game playing. I've provided a random agent that can play the two games included in this assignment: Connect Four and Breakthrough. This agent is not very good, or very interesting. Your first task is to implement a suitable opponent for it. Implement an agent that uses minimax search to choose actions. In essence, this just means that you need to implement the *suggestAction* method in minMaxAgent.py.

If using the code provided, you shouldn't really need to know how the games work to finish the assignment. It is, however, more fun if you do! So here's a brief overview of how each game works. In Connect Four, players take turns dropping pieces into a 2-dimensional array. Player 1, in this implementation, drops O's while Player 2 drops X's. The goal is for one player to create an uninterrupted line of length 4 or greater. This can be accomplished

horizontally, vertically, or diagonally. It is possible for the game to be a draw if no moves are possible.

Breakthrough is a variation of chess where all of the pieces are replaced with pawns. The board is a 2-dimensional array with the top two and bottom two rows occupied by pawns corresponding to a specific player. The goal of this game is for either one player to capture all of the other player's pawns, or to move one of their pawns to the opposite back row of the board. Pawns can move either one space forward, or one space diagonally in either direction. Pawns can capture opposing pawns only by moving diagonally. It is not possible for breakthrough to end in a draw.

Here, compare the performance of the minimax agent against the random agent in each game. Be sure to vary which agent goes first. To make this a little more reasonable, because the search space of each game could be large, you will be capped at either only looking 2-ply steps ahead or creating a tree of no more than 10,000 nodes. A ply is one set of player and opponent turns. Thus, 2-ply would mean exploring the outcome of your turn, your opponent's turn, then your next turn, and your opponent's next turn. I have provided heuristics for each game that can be used to evaluate the quality of an intermediate state game state. This is done to limit the size of your tree in memory.

Details:

- As with the bandit problems, please do not just find a library for minimax search and use it for this project.
- As mentioned at the start of class, all code should be written in Python 3.
- Explain any implementation choices you had to make in the final report. This includes the motivation behind using certain data structures or using certain libraries.
- Report the win percentage for the minimax agent for each game. Also include win rates for when the minimax agent goes first as well as second.

Monte-Carlo Tree Search (25 points)

If you are paying close attention to the game code, or at least reading the comments I've left, then I'm pretty sure the following throught crossed your mind at some point. "Brent, these heuristics you gave us are terrible!" Well, that was what I was going for. I really hope that I didn't accidentally give you a good heuristic. Anyway, in this section you will implement Monte-Carlo tree search! Recall that MCTS is very useful in situations where you don't have access to a good heuristic. Here, you'll be implementing the suggestMove method for the MCTSAgent such that it executes MCTS.

Once you have done this, compare your method against both the random agent and the minimax agent on both of the games provided. How did the MCTS agent perform? Was it able to outperform the random agent? What about the minimax agent? Does this behavior make sense to you? As with the minimax agent, report the win rates for the MCTSAgent.

Also assume that you can only go 2-ply deep when performing this search or create a tree of no more than 10,000 nodes.

Details:

- As with everything else, please do not just find a library for MCTS and use it for this project.
- As mentioned at the start of class, all code should be written in Python 3.
- Explain any implementation choices you had to make in the final report. This includes the motivation behind using certain data structures or using certain libraries. You do not need to repeat information, however. No need to go into great detail data structures if you have already talked about them. Just mention it and move on.
- Include win rates for the MCTSAgent against both the random agent and the minimax agent.

Presentation (20 points)

Your report must be complete and clear. A few key points to remember:

- AAAI Format: Your report must be written in AAAI format using either Word or LaTeX. I have included templates for each of these in the files for the course.
- Complete: the report does not need to be long, but should include everything that was requested.
- Clear: your grammar should be correct, your graphics should be clearly labeled and easy to read.
- Concise: I sometimes print out reports to ease grading, don't make figures larger than they need to be. Graphics and text should be large enough to get the point across, but not much larger.
- Credit (partial): if you are not able to get something working, or unable to generate a particular figure, explain why in your report. If you don't explain, I can't give partial credit.

Bonus (Required for CS 660): Alpha-Beta Pruning (10 points)

Even with depth limiting, it's possible that the game tree produced could be very large. For extra credit, implement an agent that uses minimax search in conjunction with alpha-beta pruning (alphaBetaAgent.py). Here, we are specifically concerned with comparing against

the minimax agent on both games in terms of the size of the resulting game tree. As before, you can either create a tree that only searches 2-ply forward, or one that is constrained to 10,000 nodes. If you use a node budget to constrain your tree, you should compare the depth of your created trees rather than the number of nodes expanded (because in this case, they'd both have 10,000 nodes). In your writeup, report the size of the tree (number of nodes expanded) and/or depth achieved for both the alpha-beta agent and the minimax agent. What effect did alpha-beta pruning have on tree size? Does this savings seem worth it?