Announcements

- Course presentations on the 25th
- 2 hours, 3 presentations; 40 minutes each
- 30 minutes of talking
- 10 minutes of questions
- Papers posted this afternoon

n-armed bandit

- Suppose that we have \( n \) actions that we can take.
- Each action has a different payoff.
- How can we maximize our long term payoff?
- Exploration/exploitation problem

Regret

- Suppose you play a RPS competition
  - After the competition, you look back at the opponents play
  - Do you wish you could have taken a different action?
    - eg regret taking the actions we did?
  - How could we guarantee that in the long term we don’t regret our policy?
    - Imagine if you are selling a solution to a customer!
Different types of regret

- **External Regret**
  - Regret relative to always performing a single action
- **Internal Regret**
  - Regret relative to swapping a single action with another
- **Swap Regret**
  - Regret relative to swapping each action for a different one

(One possible) Formal definition of regret

- Let $u_t(a)$ be the utility (payoff) of the action taken at step $t$
- Let $\sigma_t(a)$ be the probability of taking action $a$
  - Recall that a strategy is written as $\sigma$ (Lecture 3)
- Internal regret at time $T$ of not taking action $a$, $R_T^a$, is:
  - $\sum_{t=1}^{T} (u_t(a) - \sum_b \sigma_t(b)u_t(b) )$
  - The second part is the actual payoff returned
  - The first part is the payoff that we would get if we could swap the action we took with the action $A$

Regret minimizing algorithms

- If we have an algorithm which minimizes regret, in the long-term we can provide performance guarantees
  - We are assuming that we are playing against a stationary opponent

Example 1

- RPS
  - Three actions: Rock, Paper, Scissors
  - If I have an algorithm which minimizes regret, it means my opponent did not have a bias towards R/P/S which I did not exploit.
  - But, if my opponent plays the sequence R/P/S/R/P/S...
    - We won’t exploit this opponent in practice
Regret minimizing algorithm: UCB

- Finite-time Analysis of the Multiarmed Bandit Problem
  Auer, Cesa-Bianchi, Fischer

\[
\text{Regret bound: } \sum_{i=1}^{K} \left( \frac{\ln n}{A_i} \right) + \left( 1 + \frac{\pi^2}{3} \right) \left( \sum_{i=1}^{K} A_i \right)
\]

Deterministic policy: UCB1.
Initialization: Play each machine once.
Loop:
- Play machine \( j \) that maximizes \( \bar{x}_j + \sqrt{\frac{2 \ln n}{n_j}} \), where \( \bar{x}_j \) is the average reward obtained from machine \( j \), \( n_j \) is the number of times machine \( j \) has been played so far, and \( n \) is the overall number of plays done so far.

RPS & UCB

- Use more complicated actions
- Make each action a strategy:
  - random play
  - fictitious play [6 versions]
  - mimic [6 versions]
- Now, we are guaranteed to not regret always playing random; eg. in the long term we won’t lose
- If any of our strategies can exploit the opponent, it will be able to do that

UCB1

- Note: more complicated versions of UCB with better bounds, but we’ll stick with the simple approach
- How does UCB help us?
  - It depends on how we define the problem
  - Simulate with just R/P/S actions

Main procedure

```c
int chooseStrategy(int *payoff, int *plays, int round)
{
    double util[gNumStrategies];
    for (int x = 0; x < gNumStrategies; x++)
    {
        if (plays[x] == 0)
            return x;
        util[x] = (double)payoff[x]/(double)plays[x]+sqrt(2*log(round)/plays[x]);
    }
    int best = 0;
    for (int x = 1; x < gNumStrategies; x++)
        if (util[x] > util[best])
            best = x;
    return best;
}
```
A different(?) topic

- Monte-Carlo Tree Search
  - Suppose I can't write a good evaluation function
  - Let's just randomly sample moves and play the game thousands of times
  - Choose the move with the best average results
  - Suppose that I know how to choose moves, but not how to evaluate state
  - This will be easier than constructing an evaluation function
  - Probably can do smarter sampling

Playing two- (multi-) player games

- The UCB process has been extended from $n$-arm bandits to trees, where each decision in the tree is considered to be a $n$-arm bandit
- Bandit Based Monte-Carlo Planning
  - L. Kocsis, Cs. Szepesvári

UCT

- Replace the UCB term with:

  $$2C_p \sqrt{\frac{\ln t}{s}}$$

- Where $t$ is the total number of plays at this branch of the tree.
- $s$ is the # plays of this ‘arm’.
- $C_p$ is an ‘appropriate’ constant

Using UCT in practice with MCTS

- Keep a small tree in memory
- At each step:
  - Explore the tree according to the UCT rule
  - When you reach the leaf of the tree, do a random sample until the game is over (MCTS)
  - Expand the leaf of the tree that was sampled
  - Return rewards back into the tree