Computing Science (CMPUT) 455

Search, Knowledge, and Simulations

Martin Müller

Department of Computing Science
University of Alberta
mmueller@ualberta.ca

Fall 2022

455 Today - Lecture 15

- Probability of selecting right move vs different kinds of regret
- Upper confidence bound (UCB) algorithm and demo
- Code for today's lecture
  - `binomial-select.py` and `binomial-select-experiment.txt` - How often do bandits based on Bernoulli experiments make the wrong choice?
  - `ucb.py` - the UCB algorithm

Review - Story So Far

- Last time: Bernoulli experiments
- Results of repeated Bernoulli experiment follow a binomial distribution
- Next: Bandit Problems and UCB
- Questions:
- What is the probability of making a wrong choice?
- How do we measure the performance, i.e. how to quantify errors?
- How to design an algorithm that minimizes error?
- One popular answer: UCB

Bandit Problems

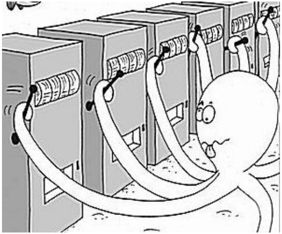


Image source: `https://blogs.mathworks.com/loren`

- Simulation-based players:
  - Run many simulations for each move as evaluation
  - Choose move with best winrate
- These decision problems are often called “bandit problems”. Why?
- “One-armed bandits” (slot machines in Casino)
- Each bandit has an arm we can pull
- Which arm has the best payoff?
- To find out, need to play and estimate winrates

Wrong Choices and Regret

- Scenario: play each arm a number of times
- Pick arm based on results, e.g. best empirical winrates
- We will make mistakes since we make decisions based on random experiments
- How to measure mistakes?
- (At least) three popular ways
  - Probability of making wrong choice
  - Simple regret
  - Cumulative regret (used in UCB)

Probability of making wrong choice, Simple Regret and Cumulative Regret

- Probability of making wrong choice
  - Arm i has best winrate p_i, but we choose arm j with $p_j < p_i$
  - Measure: what is the probability of that happening?
- Simple regret
  - Evaluate how bad our move choice j is
  - Compared to best choice i
  - Simple regret is the difference $p_i - p_j$
  - Simple regret is 0 if we pick a best move, > 0 otherwise
  - Simple regret is higher if we pick a really bad move
- Cumulative regret
  - Regret $p_i - p_j$ for every pull of an arm j
  - Cumulative regret is the sum of all these regrets

Example

- Three arms 1, 2, 3 with $p_1 = 0.8, p_2 = 0.5, p_3 = 0.1$
- Arm 1 is best (but we don't know that)
- We pull each arm once. Only arm 2 wins.

Example

- Three arms 1, 2, 3 with $p_1 = 0.8, p_2 = 0.5, p_3 = 0.1$
- Arm 1 is best (but we don't know that)
- We pull each arm once. Only arm 2 wins.
- We choose arm 2. Simple regret $p_1 - p_2 = 0.3$
- Cumulative regret 0 (pull arm 1) + 0.3 (pull arm 2) + 0.7 (pull arm 3) + 0.3 (second pull of arm 2)
- In terms of "making the wrong choice", both arm 2 and arm 3 are equally bad
- For simple regret, it is important that we choose arm 1 in the end. But choosing arm 2 is still better than arm 3.
- For cumulative regret, it is important that we choose arm 1 most of the time over the whole experiment

Types of Regret

- Regret: difference between expected value of best arm, and expected value of arm played
- Regret = 0 if you play a best arm
- Regret $>$ 0 if you don't
- Cumulative regret: each arm pull costs money
- Simple regret: can try out arms for free.
  Measure only regret of final arm selection

Types of Regret

- Regret: difference between expected value of best arm, and expected value of arm played
- Regret = 0 if you play a best arm
- Regret $>$ 0 if you don't
- Cumulative regret: each arm pull costs money
- Simple regret: can try out arms for free.
  Measure only regret of final arm selection
- **Question:** Which type of regret makes the most sense for using simulations to evaluate which move to make in a game? (i.e., "arms" are actions from the current state).

Using Regret In Algorithms

- UCB is designed to minimize cumulative regret
- For simulations in games, simple regret would make more sense:
  - Trying bad moves in simulation does not cost us anything
  - It is useful since it helps identifying a bad move
  - Only the final move decision is important
- Still, UCB-based algorithms work well
- Most used in practice
- Ongoing research on algorithms for simple regret

Wrong Choice in Bandits

- Code in `binomial-select.py`
- How often do bandits based on Bernoulli experiments make the wrong choice?
- Code implements special case:
  only two arms, exact probability calculations
- Error probability depends on how many simulations we do
- More simulations give lower error probability
- Result **strongly** depends on how close the two arms are in winrate
- See experiments in python code and
  `binomial-select-experiment.txt`

Error Rate - Theory vs Practice

- In practice, this exact error calculation is not used (**why?**)
  - We don't know the true winrates
  - It gets too complex with more than two arms or more simulations
- In most applications simple or cumulative regret is used instead

UCB Algorithm

- Our simulation players so far used simple move selection strategy
- All first moves were simulated equally often
- We saw that this is wasteful
- UCB does better
- UCB allocates simulations to moves in a smart way
- It is designed to minimize *cumulative regret*
- UCB demo from `http://mdp.ai/ucb/`
- Written by UofA grad student Eugene Chen

UCB Demo on `http://mdp.ai/ucb/`

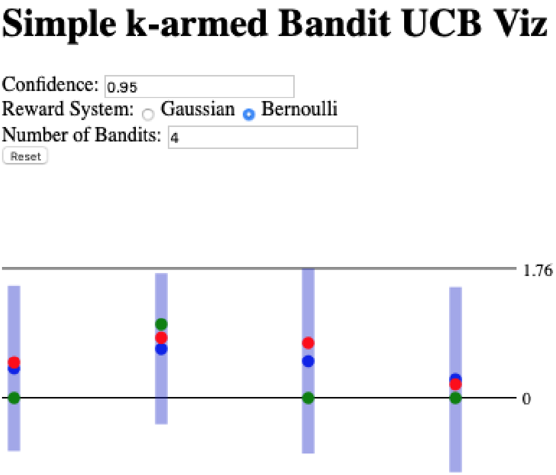


Image source: Eugene Chen, `http://mdp.ai/ucb/`

Notation for UCB Algorithm

- Goal: select best of k moves m_i, $0 \leq i \leq k - 1$
- n_i: Number of times move i has been tried
- Total number of simulations so far: $N = \sum n_i$
- w_i: number of wins for move i among n_i tries
- Empirical winrate of move i:
  $\hat{\mu}_i = w_i / n_i$

UCB Formula

- UCB stands for **U**pper **C**onfidence **B**ound
- Define Upper Confidence Bound for move i by

$$UCB(i) = \hat{\mu}_i + C\sqrt{\frac{\log N}{n_i}}. \tag{1}$$

- C is the *exploration constant*
- Larger C: require higher confidence level

UCB Algorithm For Bandit Problems

$$UCB(i) = \hat{\mu}_i + C\sqrt{\frac{\log N}{n_i}} \tag{1}$$

$$move = \arg\max_{i \in \text{moves}} UCB(i) \tag{2}$$

- Loop:
  - Compute $UCB(i)$ for all moves i
  - Pick a move i for which $UCB(i)$ is largest
  - Run one Bernoulli experiment for move i
  - Increase w_i if the experiment was a win
  - Increase n_i and N
- At end: play the **most-pulled** arm

UCB Illustration

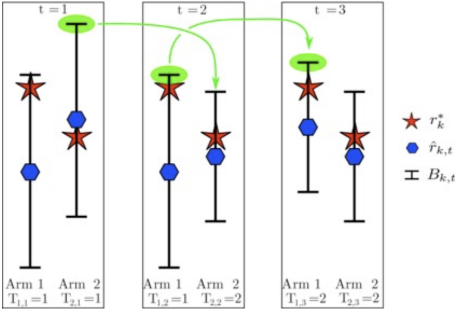
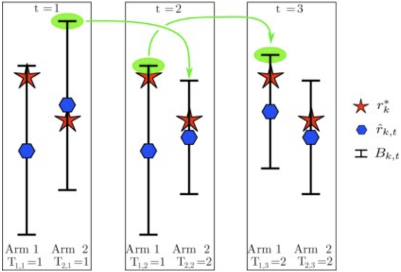


Image source: `http://iopscience.iop.org/article/10.1088/1741-2560/10/1/016012`

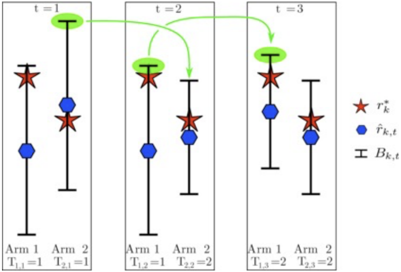
- Graphics show 3 steps in running UCB
- Red star: unknown true value
- Blue circle: empirical mean
- Black line: confidence interval
- Green: select arm with highest UCB

UCB Illustration Step 1



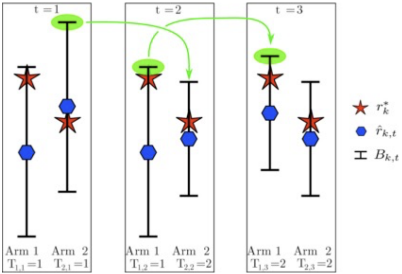
- Leftmost picture
- Arm 1 is best arm (highest true value = red star)
- Arm 1 was unlucky so far
- Its empirical mean is far below true mean
- Arm 2 has higher UCB (green)
- Step 1: select arm 2

UCB Illustration Step 2



- Arm 2 was selected
  - Consequence: Confidence interval for arm 2 shrinks
- Arm 2 lost in the new simulation
  - Consequence: Mean of arm 2 drops
- Results shown in middle picture
- Both consequences lower the UCB of arm 2
- Arm 1 now has highest UCB
- Step 2: Arm 1 selected

UCB Illustration Step 3



- Rightmost picture
- Arm 1 was selected
  - Consequence: Confidence interval for arm 1 shrinks, its UCB drops
- Arm 1 won in the new simulation
  - Consequence: Mean of arm 1 increases, UCB increases more than the drop from shrinking interval
- Arm 1 remains best by UCB, gap larger than before
- Step 3: Arm 1 selected again

UCB Code Main Loop

- `stats[move][0]` = number of wins (w_i)
- `stats[move][1]` = number of simulations (n_i)

```
stats = [[0,0] for _ in range(arms)]
for n in range(maxSimulations):
    move = findBest(stats, C, n)
    if simulate(move):
        stats[move][0] += 1 # win
    stats[move][1] += 1
```

UCB Code `ucb` and `findBest`

```
def findBest(stats, C, n):
    best = -1
    bestScore = -INFINITY
    for i in range(len(stats)):
        score = ucb(stats, C, i, n)
        if score > bestScore:
            bestScore = score
            best = i
    return best

def ucb(stats, C, i, n):
    if stats[i][1] == 0:
        return INFINITY
    return mean(stats, i)
         + C * sqrt(log(n) / stats[i][1])
```

Three Details of UCB

1. What if $n_i = 0$ at the beginning? Divide by zero problem
   - Answer 1: simulate each move once at the start, so $n_i = 1$
   - Answer 2: in my code I return a large constant INFINITY, so such moves will be chosen first

Three Details of UCB

2. How to choose exploration constant C?
   - In practice, we tune that constant for best results
   - Theory (later) shows us which choices are safe

Three Details of UCB

3. When does the loop end?
   - Can use fixed limit on total number of simulations, `maxSimulations` in code
   - Can stop if one move is “clearly best”, i.e. with high confidence
   - If there is a single best move, it eventually gets a very high percentage of all simulations

UCB vs Simple Simulation Player

$$UCB(i) = \hat{\mu}_i + C\sqrt{\frac{\log N}{n_i}}. \quad (1)$$

- UCB is much more efficient
- UCB will quickly focus almost all of its effort on small number of most promising moves
- UCB will never stop exploring other moves because of the $\log N$ term
- UCB will try the really bad-looking moves only very rarely

Exploration vs Exploitation in UCB

$$UCB(i) = \hat{\mu}_i + C\sqrt{\frac{\log N}{n_i}}. \tag{1}$$

- Exploitation: $\hat{\mu}_i$. Prefer moves with high winrate
- Exploration: $1/n_i$ term. Prefer moves with large uncertainty, small n_i
- Exploration: $\log N$ term. Never stop exploring, try bad-looking moves again eventually

Exploration vs Exploitation in UCB

$$UCB(i) = \hat{\mu}_i + C\sqrt{\frac{\log N}{n_i}}. \tag{1}$$

- **Exploitation:** $\hat{\mu}_i$. Prefer moves with high winrate
- Exploration: $1/n_i$ term. Prefer moves with large uncertainty, small n_i
- Exploration: $\log N$ term. Never stop exploring, try bad-looking moves again eventually

Exploration vs Exploitation in UCB

$$UCB(i) = \hat{\mu}_i + C\sqrt{\frac{\log N}{n_i}}. \tag{1}$$

- Exploitation: $\hat{\mu}_i$. Prefer moves with high winrate
- **Exploration:** $1/n_i$ term. Prefer moves with large uncertainty, small n_i
- Exploration: $\log N$ term. Never stop exploring, try bad-looking moves again eventually

Exploration vs Exploitation in UCB

$$UCB(i) = \hat{\mu}_i + C\sqrt{\frac{\log N}{n_i}}. \quad (1)$$

- Exploitation: $\hat{\mu}_i$. Prefer moves with high winrate
- Exploration: $1/n_i$ term. Prefer moves with large uncertainty, small n_i
- **Exploration:** $\log N$ term. Never stop exploring, try bad-looking moves again eventually

Optimism in the Face of Uncertainty

*Principle of **optimism in the face of uncertainty**: assume the best plausible outcome for each move*

- Using the upper confidence bound implements this principle in UCB
- What exactly does *plausible* mean here?
- Upper confidence bound represents the best *plausible* value of a move

Exploration vs Exploitation Tradeoff

$$UCB(i) = \underbrace{\hat{\mu}_i}_{\text{exploitation}} + C \underbrace{\sqrt{\frac{\log N}{n_i}}}_{\text{exploration}}.$$

- How to trade off between exploring and exploiting?

Exploration vs Exploitation Tradeoff

$$UCB(i) = \underbrace{\hat{\mu}_i}_{\text{exploitation}} + C \underbrace{\sqrt{\frac{\log N}{n_i}}}_{\text{exploration}}.$$

- How to trade off between exploring and exploiting?
- Exploration constant C
- C small: focus on exploitation, $\hat{\mu}_i$ term is most important
- C large: focus on exploration, $1/n_i$ term is most important
- C very large: UCB becomes very similar to the simple uniform exploration strategy

Code `ucb.py` and Examples

- `ucb.py` implements UCB algorithm and two examples
- Two cases
- Easy case: difference in arms quite large
- 10 arms, true winrates 0, 0.1,...,0.8, **0.9**
- Hard case: top two arms very close together
- payoff = [0.5, **0.61, 0.62**, 0.55]
- The difficulty of a bandit problem depends mainly on the gap between winrates of best and second-best moves

UCB in `Go3`

- Switch on with command line option
- `moveselect=UCB`
- Select *average* number of simulations/move with `-sim`
- Example: 50 simulations/move
- Assume we have 20 legal moves in total
- `moveselect=simple` will run *exactly* 50 sim. on each move, total 1000 sim.
- `moveselect=UCB` will also run 1000 sim. in total
- It will choose the first move in each simulation by UCB
- Effect: much more focus on strongest moves
- You can change the exploration parameter C

A Small Scale Test of UCB in `Go3`

- Two versions of `Go3` against each other
- `moveselect=simple` vs `moveselect=UCB`
- 5x5 board
- 50 simulations/move
- movefilter=false, simulations=random
- Win rate: 74% ($\pm$ 4.4) for UCB

Summary and Limitations of UCB

- UCB fixes an efficiency problem of the simulation player
- It does not waste much time on hopeless moves
- It does *not* fix any other problem of the simulation player
- It reaches the performance limits of simple simulation-based play more quickly

Summary and Limitations of UCB

- Main limitation: still only 1 ply deep "tree search"
- We only look for differences in the first move
- Below that (move 2, 3, ...) we only use the simulation policy
- We are still vulnerable to all biases in the simulation policy
- After move 1, still plays randomly for both opponent and player
- Only deeper tree search can fix that

Summary and Next Topics

Summary:

- Bandit problems
- From confidence bounds to UCB algorithm
- Strengths and limitations of UCB

Next Topics:

- High-level overview of search and simulation-based algorithms so far
- Selective search
- Monte Carlo Tree Search (MCTS) framework
- UCT Algorithm: Combines MCTS with UCB