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 14

- Probabilistic simulation policies
- Repeated simulations as Bernoulli experiments
- Statistics background for bandit algorithms, UCB

Coursework

- Assignment 3:
  - Simulation-based player for the game of NoGo
  - Can start now. Related to Lectures 12-14
- Reading: parts of Browne et al, *A Survey of Monte Carlo Tree Search Methods*
- Activities
- Code
  - `prob_select.py` - probabilistic selection from a list
  - `bernoulli.py` and `mystery-bernoulli.py` for today's lecture and activities
- Quiz 8: improving simulations with rules and patterns, and probabilistic simulation policies.

Stochastic vs Deterministic Simulation Policies

- Simulation-based player:
  - Simulations need to be *stochastic*, randomize moves
  - Simulations need to explore different move sequences
- Opposite of stochastic: *deterministic*
  - Deterministic policy: all simulations from same start state play the same sequence, have the same result
  - This is useless!
  - All moves would have either a 0% or 100% winrate

Why Use Randomized Simulation Policies?

- Having variety in simulations is very important
- It gives us more information about the huge state space
- This is the main idea of **sampling**
- We hope that errors in simulation “average out” through randomness
- This is true if simulations have no *bias*
- The *variance* can be reduced by getting more samples
- Contrast with deterministic policy:
  it repeats exactly the same errors in each try

From Rule-Based to Probabilistic Simulation Policies

- We have seen two types of policies so far
- Uniform Random: all legal moves equally likely
- Rule-Based: all moves from a (short) list equally likely
- Now we introduce a third type of policies: Probabilistic

Motivation

- Rule-based policies work OK but are quite crude
- What if we want a better distribution over all moves?
- Example:
  - Play pattern moves with some higher probability
  - Play other moves with some other, smaller probability
- How do they work? Start with simpler example

`prob_select.py`: Probabilistic Selection from a List

- Imagine a large table with a selection of different drinks
- There are more of some drinks than others
- Random experiment: waiter randomly grabs one drink
- Implementation in `prob_select.py`
- Given probability of selecting each drink
  - `drinks = [("Coffee", 0.3), ("Tea", 0.2), ("OJ", 0.4),...]`
- Repeat random experiment many times
- Measure drink selection frequency empirically

`prob_select.py` Sample Run

```
python3 prob_select.py
Experiment 0: OJ
Experiment 1: OJ
Experiment 2: Tea
Experiment 3: Coffee
Experiment 4: OJ
...
Experiment 99: OJ
Coffee probability 0.3, empirical frequency 0.26
Tea probability 0.2, empirical frequency 0.22
OJ probability 0.4, empirical frequency 0.4
Milk probability 0.07, empirical frequency 0.08
RootBeer probability 0.03, empirical frequency 0.04
```

Probabilistic Simulation Policy

- Same idea as in `prob_select.py`
- Used for one move decision step in a simulated game
- Given a game position in a simulated game
- Position has n legal moves
- Move i chosen with probability p_i
- Probabilities sum to 1: $\sum_{i=0}^{n-1} p_i = 1$
- Idea: heuristic to give (probably) better moves a higher chance of being played
- Can also use as a "soft" filter: give (probably) bad moves a low probability

Probabilities in Simulation Policies - So Far

- So far we have seen two kinds of policies
- Both can be viewed as (simple) probabilistic policies
- Uniform random
  - n possible moves
  - Each chosen with probability $1/n$
- Rule-based policy
  - n possible moves
  - $m \leq n$ of them selected by a rule (e.g. patterns)
  - Each chosen with probability $1/m$
  - All $n - m$ other moves have probability 0

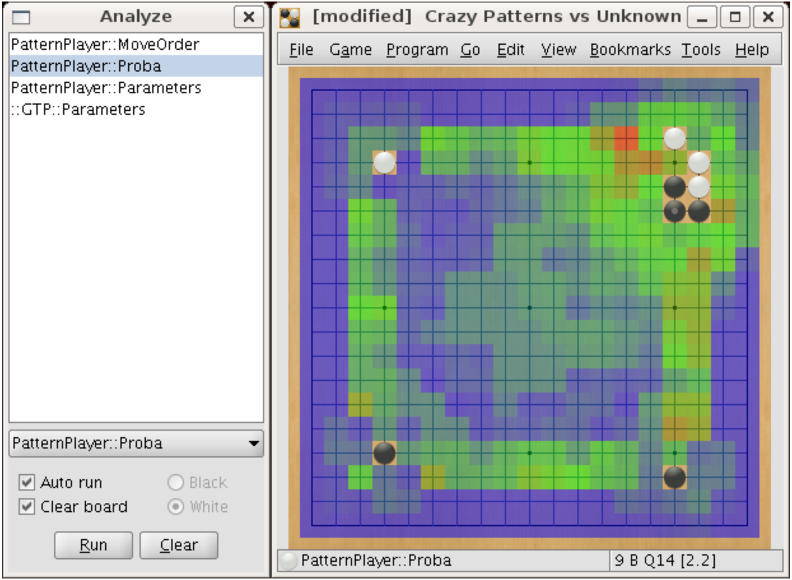
General Probabilistic Simulation Policy

- n moves
- Move i has probability p_i
- $\sum_{i=0}^{n-1} p_i = 1$
- Give moves that "look strong" a higher p_i than others
- The paper by Rémi Coulom explains one way to come up with such probabilities
- It is based on learning knowledge from game records

Visualizing Probabiliies

- Next slide shows “heat map”
- Moves on Go board encoded in different colors
- High probability moves in red/orange (probably good)
- Medium probability moves in green (probably mediocre)
- Low probability moves in blue (likely bad/meaningless)

Coulom Move Patterns, `https://www.remi-coulom.fr/Amsterdam2007/`



Rule-based vs Probabilistic Policies

- Which is better, rule-based or probabilistic policy?
- No clear answer
- Rules are easier to code efficiently
- Probabilities are better suited for many machine learning methods

Fuego: Mixing Rule-based and Probability-based Policies

- The Go program `Fuego` uses both rules and probability
- Most of its simulation policy is rule-based as in `Go3`
  - About a dozen different rules and filters
  - AtariCapture, AtariDefend, LowLiberty, Patterns,...,Random
- Probabilistic selection:
  - For 3x3 patterns
  - Fuego uses a pre-computed table of probabilities for each pattern
  - More urgent pattern moves chosen more often

Summary of Simulation Policies

- Looked at three types of simulation policies
- Uniform random
- Simple rules and filters
- Probability-based
- Where do probabilities come from?
- Answer: from machine-learned knowledge

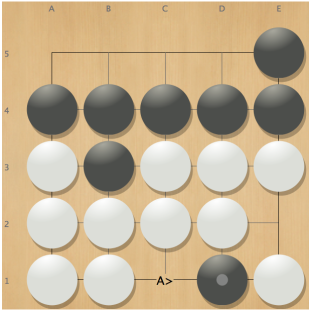
Statistical Analysis of Repeated Simulations

Next topic: Better Top-Level Algorithm

- So far, we focused on better simulations
- Random, rule-based, probabilistic
- Next, we focus on top level algorithm in FlatMC
- So far: uniform move selection
- Use same number *n* of simulations to evaluate each move
- This is not smart!
- See example on next slide

Example - Winrates of FlatMC

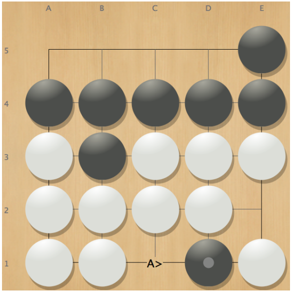
Winrates with 10, 100 and 1000 simulations per move.



- 10 Simulations/move winrates: [**('c1', 1.0)**, ('b5', 0.6), ('a5', 0.5), ('c5', 0.5), ('d5', 0.5), ('Pass', 0.4), ('e2', 0.0)]
- 100 Simulations/move winrates: [**('c1', 1.0)**, ('b5', 0.63), ('Pass', 0.58), ('c5', 0.53), ('a5', 0.49), ('d5', 0.46), ('e2', 0.06)]
- 1000 Simulations/move winrates: [**('c1', 1.0)**, ('Pass', 0.572), ('b5', 0.565), ('d5', 0.524), ('a5', 0.523), ('c5', 0.484), ('e2', 0.087)]

Example - Comparing Winrates

10 Sim:
c1: 1.0, e2: 0.0
100 Sim:
c1: 1.0, e2: 0.06
1000 Sim:
c1: 1.0, e2: 0.087



- Do we really need 1000 simulations to be convinced that c1 is better than e2? No.
- Smarter algorithm:
- *Explore* all moves in the beginning
- Focus much more on a few highest-percentage moves soon
- This leads to better decisions, less wasted time
- Example: the famous **UCB** algorithm

Statistical Analysis of Repeated Simulations

- We study some concepts from statistics
- Needed to understand the UCB algorithm for move selection
- Law of Large Numbers
- Bernoulli distribution
- Benefits and limits of doing more simulations
- More concepts: Binomial Distribution, confidence intervals, confidence level

Borel's Law of Large Numbers

- There are several Laws of Large Numbers
  - A group of theorems in probability theory
  - General idea: repeating experiments many times will get results close to expectation
- Borel's law:
- An event E has probability p
- E occurs x times in n experiments
- As $n \to \infty$:

  $$x/n \to p$$

- Empirical frequency x/n approaches probability p
- Consequence: can *use* x/n to estimate an unknown p
- This estimate will be very rough when n is small
- Improves as n gets larger, and approaches true p

Bernoulli Distribution

- Bernoulli distribution (Jacob Bernoulli, 1655 - 1705)
- One of the simplest probability distributions
- Random variable X with two different values
  - 0 (loss) or 1 (win)
- Example: coin flip
- Example: win/loss outcome of a single simulation in Go
- Not a Bernoulli distribution:
  outcome of a simulation in TicTacToe (*why not?*)

Bernoulli Distribution

- Bernoulli distribution (Jacob Bernoulli, 1655 - 1705)
- One of the simplest probability distributions
- Random variable X with two different values
  - 0 (loss) or 1 (win)
- Example: coin flip
- Example: win/loss outcome of a single simulation in Go
- Not a Bernoulli distribution:
  outcome of a simulation in TicTacToe (*why not?*)
  - three outcomes, win/loss/draw

Bernoulli Distribution (2)

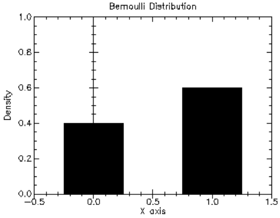


Image source:

`http://planet.racket-lang.org/package-source/williams/science.plt/3/1/planet-docs/science/random-distributions.html`

- Given fixed probability p with $0 \leq p \leq 1$
- (Wikipedia says $0 < p < 1$, but in games p can be equal to 0 or 1)
- Probabilities for outcomes 1 and 0
  $Pr(X = 1) = p$
  $Pr(X = 0) = 1 - p$
- Sometimes, q is written for $1 - p$
- Example: $p = 0.6, q = 1 - p = 0.4$

Bernoulli Experiment

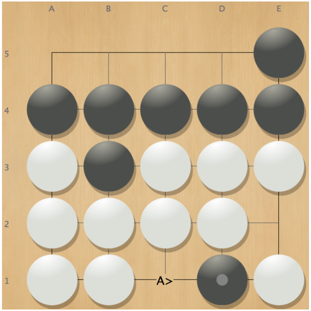
Image source: `https://en.wikipedia.org/wiki/Coin_flipping`

- Random experiment, typically repeated many times, same fixed p
- Each single experiment draws from Bernoulli distribution for p
- Example: coin flip with fair coin, $p = q = 0.5$
- Implementation: `bernoulli.py` - also see Activity

```
def bernoulli(p, limit):
    wins = 0
    for _ in range(limit):
        if random.random() < p:
            wins += 1
    return wins / limit
```

Simulation-based Evaluation as Bernoulli Experiments - Example

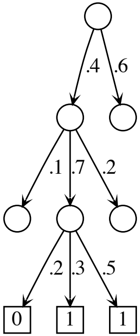
Running a fixed simulation policy from a fixed Go position is a Bernoulli experiment



- Example: winrates after playing each move, with 10, 100, 1000 sim.
- For each move: converges to a fixed probability

Move	10	100	1000
c1	1.0	1.0	1.0
b5	0.6	0.63	0.565
a5	0.5	0.49	0.523
c5	0.5	0.53	0.484
d5	0.5	0.46	0.524
Pass	0.4	0.58	0.572
e2	0.0	0.06	0.087

Simulation is a Bernoulli Experiment

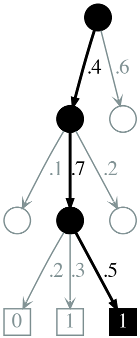


Why is random sampling from a game tree a Bernoulli experiment?

Proof sketch

- Finite tree, finitely many leaves, finitely many paths to leaves
- Each leaf has fixed value 0 or 1
- In each node, the simulation policy has a fixed distribution over its children
- We can compute the probability of choosing each path as the product of the probabilities of choosing each move on the path

Simulation is a Bernoulli Experiment #2



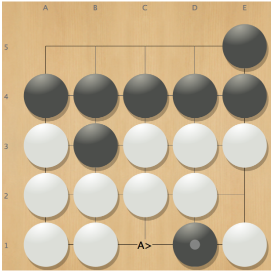
.4 * .7 * .5 = .014

- The probability of choosing some specific path to a leaf is a (small) constant
- The winning probability at the root is just the probability of choosing a path leading to a win
- This is a sum of (a huge number of) constants, so is a constant p
- Each simulation is like a Bernoulli experiment with parameter p
- We win by choosing a winning path, which happens with probability p

Analysis

- Analyse “flat” simulation player (1 ply tree)
- Runs *n* simulations on each child *c* of the root
- Focus on one child *c* now
- If we increase *n*, run more and more simulations, the winrate for *c* will stabilize
  - Reason: law of large numbers
- Limit, infinite number of simulations:
  - Winrate will converge to the “true winrate”
  - For one particular random simulation policy
  - For one particular start state
  - Winrate may be far from true minimax value
    - Reason: bias of simulation policy

Simple Move Selection is Inefficient



1000 Simulations/move
winrates: **[('c1', 1.0)**, ('Pass', 0.572), ('b5', 0.565), ('d5', 0.524), ('a5', 0.523), ('c5', 0.484), ('e2', 0.087)]

- We really do not need 1000 simulations to figure out that e2 is bad
- Huge gap between winrates of bad move e2 and best move c1
- Very limited gain from running more and more simulations on worst moves
- Very inefficient use of time
- We need to *explore* all moves, but...
- We should *focus* most effort on the most likely good moves

`mystery-bernoulli.py`: Guessing the Winrate

- Activity: experiment with `mystery-bernoulli.py`
- Program generates a random `p`
- Runs a number of Bernoulli experiments
- Outputs the empirical winrate
- How well can you guess the true `p`?
- How does the number of simulations affect it?

Optional Activity: Mystery game

- Program your own simulation-based game
- First, choose number of moves n
- Next, generate the true winrates p_i for each move i
- Next, ask the user for number of simulations/move
- Run that many simulations and collect empirical winrates (as in Go3)
- Print out the empirical winrates
- Let the user guess the best move
- Now, let the user know the true winrates and true best move
- Discuss: when is this game easy? When is it hard?

Benefits of More Simulations

Benefits of running more simulations:

- Reduce variance
- Better selection when several moves are almost tied for first
- Rule out “unlucky” cases which occur with low number of simulations:
  - Bad move wins many simulations - estimated winrate too high
  - Good move loses many simulations - estimated winrate too low

Optimize What Simulations Tell Us

- Goal: a smarter way to decide:
- Which moves do we most need to evaluate better by running more simulations?
- Let's study the results of doing many simulations on one move
- The outcomes follow a *binomial distribution*

Binomial Distribution

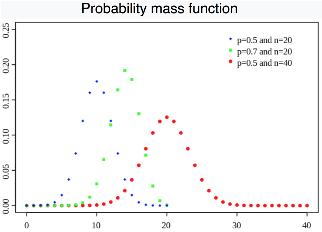
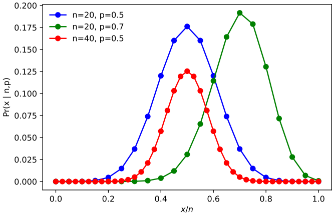


Image source:

`https://en.wikipedia.org/wiki/Binomial_distribution`

- Repeat the same Bernoulli experiment many times
- Number of wins is binomially distributed
- $B(n, p)$ = number of wins in n tries, where each has win probability p
- Expected value of $B(n, p)$: np
- As n grows, distribution of $B(n, p)/n$ becomes more narrowly centered around p
- Probability of being far from p decreases as n grows

Binomial Distribution



- Repeat the same Bernoulli experiment many times
- Number of wins is binomially distributed
- $B(n, p)$ = number of wins in n tries, where each has win probability p
- Expected value of $B(n, p)$: np
- As n grows, distribution of $B(n, p)/n$ becomes more narrowly centered around p
- Probability of being far from p decreases as n grows

Bandit Algorithms and UCB - Motivation

- Consider top 2 moves from example
- After 10 simulations: **('c1', 1.0), ('b5', 0.6)**
- After 100 simulations: **('c1', 1.0), ('b5', 0.63)**
- How sure are we that c1 is better?
- We want to compare the two *true means* of c1 and b5
- We only have the *empirical means* for moves c1 and b5
- In theory, b5 (or another even lower-ranked move) could still be better
- It is extremely unlikely given the results so far - so we could ignore that move...
- How unlikely? We need some more statistics to answer that

Confidence Interval

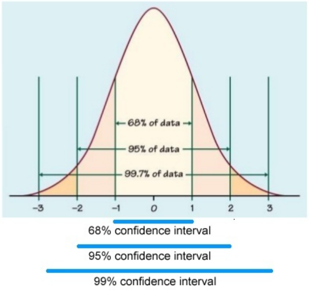


Image source:

`https://www.quora.com`

- Confidence Interval in statistics:
  - A range in which the true value is estimated to be
- Confidence level:
  - Probability that the range contains the true value

Confidence Interval for Repeated Bernoulli Experiment

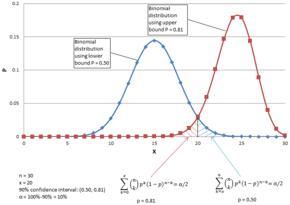


Image source:

`http://www.biyee.net`

- Repeated Bernoulli experiment with unknown win probability p
- Given empirical data
  Example: 20 wins in 30 tries
- From this, we need to estimate the unknown true mean p
- For any mean p, distribution of number of wins out of 30 experiments is the binomial distribution $B(30, p)$
- p is likely to be close to the empirical mean $20/30 = 0.66..$

Confidence Interval for repeated Bernoulli experiment

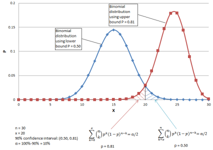
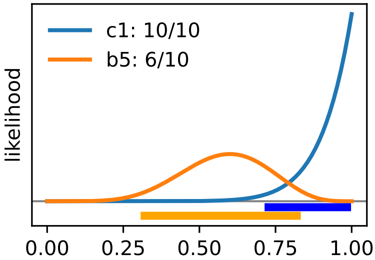


Image source:

`http://www.biyee.net`

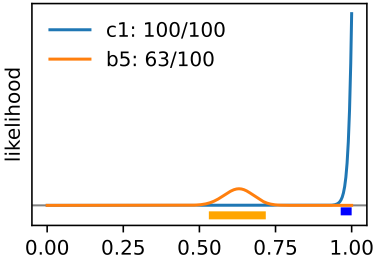
- For a given confidence level, we can define an interval around the empirical mean that likely contains the true mean
  - Example: empirical mean 0.666..
- For lower confidence level, the intervals are smaller - more chance of error
- For higher confidence level, the intervals are larger - less chance of error
- Example: 90% confidence interval around 0.666 = $(0.50, 0.81)$
- For any value of p in $0.50 < p < 0.81$:
  - The empirical result 20/30 is within the "middle 90%" of outcomes

Back to Finding the Best Move



- Given different moves, each with empirical winrate
- We can compute confidence intervals for true mean of each move
- Goal: separate best move from all others
- Separation means:
- The whole confidence interval for the best move
- ... is above the intervals of all other moves

Back to Finding the Best Move



- In practice, that often takes far too long
- In the UCB algorithm, we use the upper confidence bound instead - the upper end of the confidence interval
- Details: next lecture