Computing Science (CMPUT) 455

Search, Knowledge, and Simulations

Martin Müller

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

Fall 2022

Part III

Simulations and Monte Carlo Tree Search

455 Today - Lecture 12

- **Midterm this week**, in LAC
- See midterm study guide
- Today - Finish Lecture 11 slides, start lecture 12

455 Today - Lecture 12

- Start Part 3 - Simulations and Monte Carlo Tree Search
- Simulation methods in computing science
- Early examples simulating physics
- Examples in heuristic search
- Flat Monte Carlo
- Simulation-based TicTacToe player
- Simulation-based Go player, `Go3`

Coursework

- Assignment 2
- Reading: Rémi Coulom, *Computing Elo Ratings of Move Patterns in the Game of Go.*
- Lecture 12 activities
- Quiz 7: Simulations

Simulation Methods

Simulation Methods

- Wikipedia definition:

  Simulation is the imitation of the operation of a real-world process or system over time.

- Huge number and variety of applications
- Here, we focus on Monte Carlo (MC) simulation
- Main idea: learn information from **random sequences** of decision steps
- First simple examples:
  - No real sequence, just a single random step
  - That step is repeated many times for the same decision problem

Example 1: Estimate π with Monte Carlo

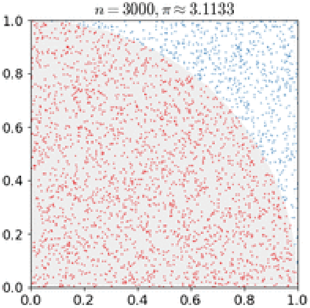


Image source: `https://upload.wikimedia.org/wikipedia`

- Generate random point (x, y) in $[0, 1) \times [0, 1)$ square
- Check if within circle: $x^2 + y^2 < 1$
- Repeat many times
- Fraction of points within circle is estimate for its area, $\pi/4$
- See code `estimate_pi.py` and do **Activity 12a**

Example 2: Numerical Integration with Monte Carlo Sampling

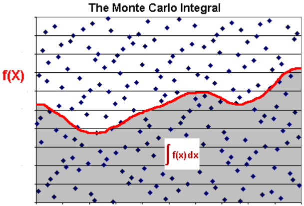


Image source:

`http://cedric-augonnet.com`

- Numerical Integration
- Similar idea as with π example
- Given arbitrary function *f*
- Count fraction of random points "under the curve"
- Need to find enclosing rectangle
- Deal with negative function values
- Demo of code `numerical_integration_MC.py`

Discussion - Monte Carlo Sampling for Numerical Integration

- Very general method
- No assumptions on type of function to integrate
- Can also use it in higher dimensions
  - Example in Activity 12a: volume of unit ball
  - Can estimate volume of irregularly shaped object
- All you need is:
  - Bounding box containing object
  - Random point generator
  - Reasonably fast method to check if point is inside or outside the object

Discussion continued - Limitations

Limitations

- Convergence of the basic method is slow
  - High variance (but no bias)
- Much faster methods exist if:
  - Functions have “nice” properties such as smoothness
- Decades of work on specialized MC methods
- Basic Monte Carlo (MC) is a fall-back
  for cases without “nice” structure

Origins of Monte Carlo Method - Manhattan Project

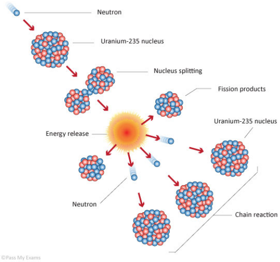


Image source: `http://www.nuclear-power.net`

- 1940's Manhattan Project: developed the first atomic bombs
- Extremely complex physics modeling required
- First computers were just becoming available
- One key problem: Neutron diffusion

Neutron Diffusion

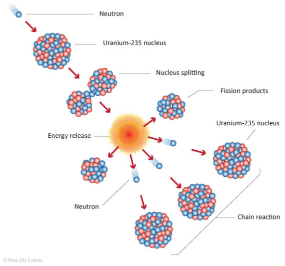


Image source: `http://www.nuclear-power.net`

- To obtain a nuclear chain reaction:
- Each neutron must create >1 neutron, before being absorbed
- Physicists could not solve this problem with “pure calculation”
- Ulam and von Neumann developed first *Markov Chain Monte Carlo* simulation methods
- Simulate many random neutrons flying through a substance
- Count how often new ones are generated in simulation

Markov Chain Monte Carlo

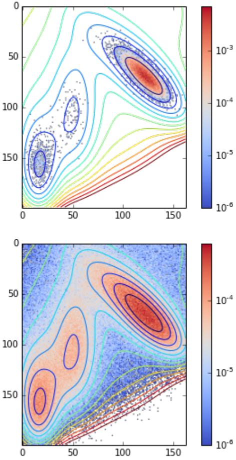


Image source:

`bougui505.github.io/`

`python/2014/11/17/`

`simple-markov-chain-monte-carlo-mcmc-algorithm-in-python.`

`html`

- Main idea:
  simulate *random walk* of particles
- Walk is biased by physics constraints
- Particle distribution approximates...
- ...true distribution of what should be measured
- Very popular in physics, engineering for modeling complex systems

Markov Chain Monte Carlo

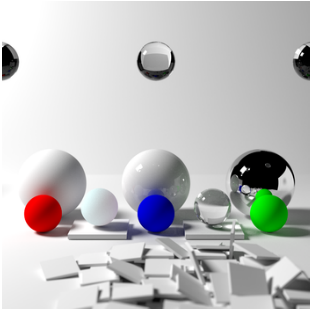


Image source:

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

- Application:
  image rendering by path tracing
- Each light source emits many photons
- Many light rays sent out in random directions from source
- Model the physics of reflection, absorption etc. of those particles
- Number of particles hitting an area gives its illumination

Simulation Model

- To simulate something we need a model
- Neutron diffusion: physics - laws of motion, speed of neutrons, absorption by different materials, radioactive decay of different uranium isotopes,...
- Path tracing: light sources, laws of optics, shadows, reflection/light scattering, indirect light, ...
- Games:
  legal moves, outcome at the end
- Games with chance:
  simulated dice throws,
  possible distributions of unknown cards,...

Garbage In - Garbage Out Principle (GIGO)

- A simulation can only be as good as the underlying model
- If you feed great data to an invalid model, you typically get garbage
- Examples:
  - Missing relevant physics
  - Wrong initial conditions
  - Numerical instability, cascading errors
  - Bugs in computer code and/or hardware
  - Not implementing the rules of Go properly

Simulation in Heuristic Search

Simulation and Random Walks in Heuristic Search

- Early application: GSAT and WalkSAT for Boolean Satisfiability Problem (SAT)
- Given a boolean formula
- Example: $(x_0 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$
- Check if it is *satisfiable*: an assignment of true and false to the variables which makes the formula true
- Example: Set $x_0 = \mathit{True}, x_1 = \mathit{False}, x_2 = \mathit{False}$
- Whole formula becomes true
- Solving large SAT formulas is a difficult problem (NP-Hard!)
- Best solvers use heuristic search

Solving SAT by Systematic Search

- Given SAT formula with n variables, $x_0, ...x_{n-1}$
- Can solve by systematic search, trial and error
- Set $x_0 = \mathit{True}$
  - Simplify formula
  - Solve SAT problem with $n - 1$ variables
- If fail: Set $x_0 = \mathit{False}$
  - Simplify formula
  - Solve SAT problem with $n - 1$ variables
- Worst-case cost of this procedure?
  - Exponential in n
  - Need to try many of the 2^n variable assignments
- Best case: guess good values for all variables, cost $O(n)$

GSAT and WalkSAT: Solving SAT by Biased Random Walk

- Local search methods developed in 1990's
- Start with random assignment of True or False to each variable
- If formula is true: stop, success
- If not, use heuristics to *flip* value of one variable
- Goal: Try to make all parts (*clauses*) of a formula true
- Restart if no progress for a while

GSAT and WalkSAT: Solving SAT by Biased Random Walk

- Which variables should we flip?
- Balance *exploitation* and *exploration*
  - Exploitation: flip a variable that makes the “largest possible improvement” of the formula
    - What could “improvement” mean?
  - Exploration: flip a random variable
- How does local search compare to systematic search?
  - No clear winner, different strengths/weaknesses
  - Modern SAT solvers use clause learning techniques

Simulation in Game Tree Search - Backgammon

Image sources:

`www.backgammoned.net`

- Early success of simulation methods in games: Backgammon
- “Rollout” games with many different sequences of dice throws
- Play move that is most successful in these rollouts
- Backgammon was also an early success story for neural networks
  We will discuss that story later

Simulation in Backgammon

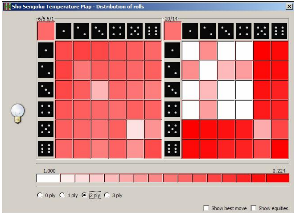


Image sources: `www.bkgm.com/gnu/AllAboutGNU.html`

- Picture: simulation result of all possible dice throws
- white = good winrate, red = bad
- Right side: risky move
    - Many throws lead to sure win
    - Many other throws lead to sure loss
- Left side: safe move
    - Outcomes more similar
    - Here, this move is better in expectation
- Knowing this distribution allows you to make better decisions

From Games With Chance Elements to Games With No Chance

- Games with chance element (dice, hidden cards)
  - Using random simulations is a natural idea
  - Tried even in the earliest programs
- Games without chance element (chess, checkers, Go,...)
  - Using random simulations is much less natural
  - It took a lot longer to develop those methods
  - Often, it also works very well
  - Our first example: TicTacToe
  - Second example: Go

Random Simulation in TicTacToe

- From given state, finish game
  with moves selected uniformly at random
- In TicTacToe, all empty squares are legal moves
- End simulation when game is over by rules:
  - Three in a row
  - Board full
- Implementation: method `TicTacToe.simulate()`
  - In file `tic_tac_toe.py`

Method TicTacToe.simulate()

```
def simulate(self):
    allMoves = self.legalMoves()
    random.shuffle(allMoves)
    i = 0
    while not self.endOfGame():
        self.play(allMoves[i])
        i += 1
    return self.winner(), i
```

Method TicTacToe.simulate()

```
def simulate(self):
    allMoves = self.legalMoves()
    random.shuffle(allMoves)
    i = 0
    while not self.endOfGame():
        self.play(allMoves[i])
        i += 1
    return self.winner(), i
```

Implementation note:

- For random play in TicTacToe, it's enough to shuffle the list of moves once, then play in that order
- Not true in Go (**why?**)

Use Simulations as Evaluation Function

Evaluate: how good is a game state?

- Exact answer:
  - Run solver
  - Compute minimax value
- Heuristic, first (old) answer:
  - Run depth-limited alphabeta search
  - At depth limit: call heuristic evaluation function
  - Compute minimax value
  - Problem: how to create evaluation function?
- **Heuristic, second (new) answer:**
  - Run simulations, score final result
    - Win = 1, loss = 0 (draw = 0.5)
  - Compute *winrate* over all simulations

Simulation-Based Player

- Uses 1-step (1-ply) lookahead to evaluate all moves
- For each legal move:
  - Run *n* simulations
  - Measure the winrate (winning percentage) for these simulations
- After all simulations:
  - Play move with highest winrate
- Implementation: `simulation_player.py`

Flat Monte Carlo

- The method based on 1-ply lookahead + simulations is sometimes called *Flat Monte Carlo*
- **Monte Carlo** method: uses random simulations
- **Flat:** does not build a deep tree,
  only 1 ply (1 move) lookahead
- Contrast: Monte Carlo Tree Search
  builds a (often very deep) tree

Simulation Player Implementation - simulate

`SimulationPlayer.simulate(self, state, move)`

- Play `move` from given `state` - changes state
- Evaluate state after the move:
- Run `self.numSimulations` from it
- After simulations: `undoMove` to restore previous state
- Evaluation of move:
  average outcome of these simulations

SimulationPlayer.simulate

```
def simulate(self, state, move):
    stats = [0] * 3
    state.play(move)
    moveNr = state.moveNumber()
    for _ in range(self.numSimulations):
        winner, _ = state.simulate()
        stats[winner] += 1
        state.resetToMoveNumber(moveNr)
    state.undoMove()
    eval = (stats[BLACK] + 0.5 * stats[EMPTY])
           / self.numSimulations
    if state.toPlay == WHITE:
        eval = 1 - eval # Negamax view
    return eval
```

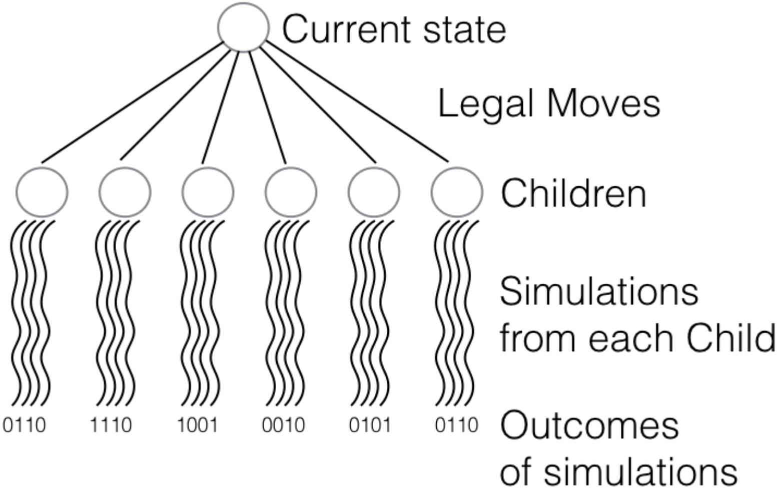
Simulation Player Implementation - genmove

- `SimulationPlayer.genmove(self, state)`
- For each move: Evaluate it by simulation
- Collect and compare winrates for all moves
- Pick the move with best winrate

SimulationPlayer.genmove

```
def genmove(self, state):
    moves = state.legalMoves()
    numMoves = len(moves)
    score = [0] * numMoves
    for i in range(numMoves):
        move = moves[i]
        score[i] = self.simulate(state, move)
    bestIndex = score.index(max(score))
    best = moves[bestIndex]
    return best
```

Simulation-Based Player



Match Simulation-Based Player vs Perfect and Random Players

- `simulation_player.py`
- `perfect_player.py`
  solves game at each step
- `random_player.py`
  selects move uniformly at random
- `play_match.py` run test games, print statistics
- How do these players compare?
- How does the strength of the Simulation Player change if we increase the number of simulations?

Match 1: 10 simulations/move, 100 games each color

- Results in table:
  - Black player (X, name on left side)
  - Result vs White player (O, name on top)
- Perfect player never loses with either color
- Going first is a big advantage
  (unless both are perfect)
- Note: numbers will change if re-run,
  but results will be similar with high probability

Table: Wins/Draws/Losses (W/D/L), 10 simulations/move, 100 games each color.

Black	Sim(10)	Perfect	Random
Sim(10)	**62W**/21D/17L	0W/**76D**/24L	**97W**/3D/0L
Perfect	**77W**/23D/0L	0W/**100D**/0L	**100W**/0D/0L
Random	9W/5D/**86L**	0W/20D/**80L**	**64W**/7D/29L

Scaling of Simulation Player vs Perfect

- Vary number of simulations 1, 10, 100, 1000
- Separate stats as Black, as White
- Results for Random and Perfect added for comparison
- Increasing simulations clearly helps
- 1000 simulations/move seem to play almost perfectly?
- Activity 12b: re-try this experiment, run more games
- TicTacToe is simple. In Go, Sim(1000) still plays poorly

Player	As Black	%	As White	%
Random	0W/20D/**80L**	10%	0W/0D/**100L**	0%
Sim(1)	0W/19D/**81L**	9.5%	0W/7D/**93L**	3.5%
Sim(10)	0W/**80D**/20L	40%	0W/24D/**76L**	12%
Sim(100)	0W/**100D**/0L	50%	0W/**77D**/23L	38.5%
Sim(1000)	0W/**100D**/0L	50%	0W/**100D**/0L	50%
Perfect	0W/**100D**/0L	50%	0W/**100D**/0L	50%

Scaling Simulation Player vs Random

- Vary number of simulations 1, 10, 100, 1000
- Separate stats as Black, as White
- Results for Random and Perfect added for comparison
- Increasing simulations clearly helps
- Sim(1000) as white better than perfect???

Player	As Black	%	As White	%
Random	64W/7D/29L	67.5%	29W/7D/64L	32.5%
Sim(1)	82W/9D/9L	86.5%	63W/15D/22L	70.5%
Sim(10)	97W/1D/2L	97.5%	78W/8D/14L	82%
Sim(100)	99W/1D/0L	99.5%	88W/9D/3L	92.5%
Sim(1000)	97W/3D/0L	98.5%	**91W/5D/4L**	93.5%
Perfect	100W/0D/0L	100%	**80W/20D/0L**	90%

Comments on Experiments

- 100 games is not enough to get precise numbers
  - Still large statistical error
  - Enough to get a rough first idea
- Benefit of more simulations is clear
- Does it play perfectly?
  - In TicTacToe, maybe close to perfect
  - In harder games like Go, not at all

Comments on Experiments (2)

- Sim(1000) can exploit Random better than the perfect player
- Confirmed with 1000 games - see below
- Probable reason:
- Tie-breaking towards moves that are more successful in random simulations
- Optional activity: write a perfect player with simulation-based tiebreaking

Player	As Black	%
Sim(1000)	988W/12D/0L	99.4%
Perfect	991W/9D/0L	99.55%

Player	As White	%
Sim(1000)	908W/59D/33L	**93.75%**
Perfect	799W/201D/0L	**89.95%**

Go3 - Simulation-Based Go Players

- `Go3` implements several variations of simulation-based players
- All choose their best move based on success in simulations
- `Go3` implements two different **simulation** policies
- `Go3` implements two different **move selection** algorithms at the root
- `Go4` will have even more simulation policies

Simulations in `Go3`

- As in TicTacToe, simulations used as state evaluation
- Use simulations to finish game many times from current position
- Keep winrate statistics to evaluate state
- How to do simulation in Go?
- Two simulation methods implemented in `Go3`
    - Almost-random
    - Rule-based (discussed later)

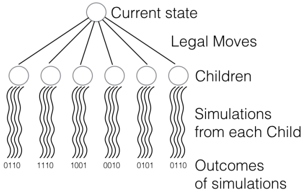
Almost-Random Simulations in `Go3`

- Remember `Go1` and `Go2`, random Go players
- Selected moves uniformly at random
  - *Except:* do not fill one-point eyes
- Almost-random simulation in `Go3` works the same way
- It will choose almost-random moves in its simulations
- Filter eye-filling moves only
- Pick all other moves with equal probability
- Pass in simulation only if all board moves are eye-filling

Move Selection in `Go3`

- Two algorithms: simple and UCB
- Simple is the same as in `simulation_player.py`
    - For each move, try *n* simulations starting with this move
- Second algorithm is UCB (later)
    - Smarter choice of which moves to simulate more often

Simple Move Selection Details



- For each legal move m_i
  - Play m_i
  - Run n random simulations
  - Undo move m_i
  - Count number of wins w_i
  - Compute winrate w_i/n
- Play the move with maximum winrate
- move $= \arg\max_i w_i/n$
- Difference to TicTacToe:
  - Legal moves include pass

Pass in Simulation vs Pass in Game

Simulations

- Regarding passing, behave like `Go1` and `Go2`
- No pass except at very end to avoid filling eyes

Move selection for player

- `Go3` player move selection is different from simulations, `Go1` and `Go2`
- In Go, pass is always legal
- `Go3` player can pass earlier if it has the best winrate
- Examples:
  - All moves on board are bad tactically
  - All moves on board are in own or opponent territory

Simulation Speed in Go vs TicTacToe

- Speed in Go is quite slow
- Simulations take much longer than in TicTacToe
- Max. 9 moves in TicTacToe
- Roughly $n \times n$ on board size n in the opening
- Example: 7×7 Go
- Simulation can be longer than 50 moves
- Reason:
    - Capture large blocks
    - Play back onto those newly empty points

Summary

- Simulation methods:
  - Approximate a quantity that is difficult to compute otherwise
  - Example: physics
  - Example: evaluation of game states
- Monte Carlo simulation - random sequences of actions
- First example - TicTacToe
  - Approaches (almost?) perfect play with enough simulations
- Second example - Go
  - Much better than random, but far from a good player