Monte-Carlo Search in Go

By David Anderson
SZTAKI (Budapest, Hungary)
WPI D2009
Motivation

- 1997, Deep Blue won against Kasparov
- Average workstation can defeat best Chess players
- Computer Chess no longer “interesting”
- Go is much harder for computers to play
  - Branching factor is ~50-200 versus ~35 in Chess
  - Positional evaluation inaccurate, expensive
  - Game cannot be scored until the end
- Beginners can defeat best Go programs
Playing Go

- Two-player, total information
- Players take turns placing black and white stones on grid
- Board is 19x19 (13x13 or 9x9 for beginners)
- Object is to surround empty space as territory
- Pieces can be captured, but not moved
- Winner determined by most points (territory plus captured pieces)
Go looks like...

Image from http://ict.ewi.tudelft.nl/~gineke/
Minimax/α-β algorithms require huge trees
  - Tree depth cannot be cut easily
Monte-Carlo now more popular
  - Simulate random games from the game tree
  - Use results to pick best move
Two areas of optimization
  - Discovery of good paths in the game tree
  - Intelligence of random simulations
    - Random games are usually bogus
Monte-Carlo Search

- Need to balance between exploration...
  - Discovering and simulating new paths
- And exploitation...
  - Simulating the most optimal path
- Best method is currently UCT given by Levente Kocsis and Csaba Szepesvári.
Say you have a slot machine with a probability of giving you money. You can infer this probability through experimentation.
What if there are three slot machines, and each has a different probability?
You need to choose between experimenting (exploration) and getting the best reward (exploitation).
Bandit Problem

UCB algorithm balances these problems to minimize loss of reward.
UCT applies UCB to games like Go, deciding which move to explore next by treating it like the bandit problem.
UCT Demonstration

- Starts with one-level tree of legal board moves
UCT Demonstration

- Picks best move according to UCB algorithm
RUNS MONTE-CARLO SIMULATION, UPDATE NODE’S WIN/LOSS.

This is one iteration of the UCT process.
If node gets visited enough times, start looking at its child moves
UCT Dives Deeper, Each Time Picking the Most "Interesting" Move.
Eventually, UCT has built a large tree of simulation information
State of UCT and Computer Go

- UCT is now in most major competitive programs
- “MoGo” used UCT to defeat a professional
  - Used 800-node grid and a 9 stone handicap
- Much research now focused on improving simulation intelligence
Policy decides which move to play next in a random game simulation

High stochasticity makes UCT less accurate
  ▪ Takes longer to converge to correct move

Too much determinism makes UCT less effective
  ▪ Defeats purpose of Monte-Carlo search
  ▪ Might introduce harmful selection bias
Good and Bad Shapes

- Certain shapes in Go are good
  - “Hane” here is a strong attack on B
    
    ![Diagram of Hane attack]

- Others are quite bad!
  - B’s “empty triangle” is too dense and wasteful
    
    ![Diagram of empty triangle]
Pattern Knowledge

- MoGo uses pattern knowledge with UCT
  - Hand-crafted database of 3x3 interesting patterns
  - Doubled simulation win-rate according to authors

- Can pattern knowledge be trained automatically via machine learning?
Paper “Monte-Carlo Simulation Balancing”
- (by David Silver and Gerald Tesauro)
- Policies accumulate error with each move
- Strong policies minimize this error, but not the whole-game error
- Proposes algorithms for minimizing whole-game error with each move
- Authors tested on 5x5 Go using 2x2 patterns
  - Found that balancing was more effective over raw strength
The Project

- Implemented pattern-learning algorithms in “Monte-Carlo Simulation Balancing”
  - Strength: Apprenticeship
  - Strength: Policy Gradient Reinforcement
  - Balance: Policy Gradient Simulation Balancing
  - Balance: Two-Step Simulation Balancing
- Used 9x9 Go with 3x3 patterns
Experiments

- Used amateur database of 9x9 games for training
- Mention-worthy metrics:
  - Simulation winrate against purely random
  - UCT winrate against UCT purely random
  - UCT winrate against GNU Go
Apprenticeship Learning

- Simplest algorithm
- Looks at every move of every game in the training set
  - High preference for chosen moves
  - Low preference for unchosen moves
- Strongly favored good patterns
- Over-training; poor error compensation
  - Values converge to infinity
Apprenticeship Results

Apprenticeship vs Pure Random

Winrate (%)

Game Type

Playout

UCT vs libEGO

UCT vs GNU Go

Pure Random

Apprenticeship
Reinforcement Learning

- Plays random games from the training set
- If the simulation matches the original game result, patterns get higher preference
- Otherwise, lower preference
- Results were promising
Reinforcement Results

Reinforcement vs Pure Random

Winrate (%)

Game Type

Playout

UCT vs libEGO

UCT vs GNU Go

Pure Random

Reinforcement
Simulation Balancing

- For each training game...
  - Plays random games to estimate win rate
  - Plays more random games to determine which patterns win and lose
  - Gives preferences to patterns based on error between actual game result and observed winrate
Simulation Balancing

- Usually, strong local moves
- Seemed to learn good pattern distribution
- Aggressively played useless moves hoping for an opponent mistake
- Poor consideration of the whole board
Global perspective: not good!
Simulation Balancing Results

Simulation Balancing versus Pure Random

Winrate (%)

Game Type

Playout

UCT vs libEGO

UCT vs GNU Go

Pure Random

Simulation Balancing
Picks random game states
Computes score estimate of every move at 2-ply depth
Updates pattern preferences based on these results, using actual game result to compensate for error
Two Step Balancing Problems

- Game score is hard to estimate, usually inaccurate
- Extremely expensive; 10-30 sec to estimate score
- Game score doesn’t change meaningfully for many moves
- Probably does not scale as board size grows
Two Step Balancing Results

**Graph:**

- **Game Type:**
  - Playout
  - UCT vs libEGO
  - UCT vs GNU Go

- **Winrate (%):**
  - Pure Random
  - Two Step Balancing

Values:
- Playout: Two Step Balancing > Pure Random
- UCT vs libEGO: Two Step Balancing > Pure Random
- UCT vs GNU Go: Pure Random > Two Step Balancing
Very bad play
Result Summary

Algorithm Results

- Pure Random
- Apprenticeship
- Reinforcement
- Simulation Balancing
- Two Step Balancing

Winrate (%)

Game Type
- Playout
- UCT vs libEGO
- UCT vs GNU Go
Conclusions

- Reinforcement strongest
- All algorithms capable of very deterministic policies
- Higher playout winrates were too deterministic and thus usually bad with UCT
- Go may be too complex for these algorithms
  - Optimizing self-play doesn’t guarantee good moves
Special thanks to...

- Levente Kocsis
- SZTAKI
- Professors Sárközy and Selkow
Questions, Comments, Queries?
Move Selection

- Algorithm generates list of patterns
- Each pattern has a weight/value
- Policy looks at open positions on the board
- Gets the pattern at each open position
- Uses weights as a probability distribution