COMP 424 - Artificial Intelligence
Lecture 8: Monte-Carlo Tree Search

Instructor: Joelle Pineau (jpineau@cs.mcgill.ca)

Class web page: www.cs.mcgill.ca/~jpineau/comp424

Unless otherwise noted, all material posted for this course are copyright of the instructor, and cannot be reused or reposted without the instructor's written permission.

Quiz 2

1. Apply Minimax to this tree. Which action (left or right) will be chosen at node A?

2. If nodes are generated left-to-right, which nodes would be pruned by alpha beta pruning?

3. If nodes are generated right-to-left, which nodes would be pruned by alpha beta pruning?
Quick recap

• Last class:
  – Perfect information, 2-player games.
  – Alpha-beta search to cutoff branching factor.
  – Optimal play is guaranteed against an optimal opponent if search proceeds to the end of the game. But opponent may not be optimal!
  – If heuristics are used: this assumptions means opponent playing optimally according to the same heuristic function as the player.

• Today:
  – Monte Carlo Tree Search (MCTS)
  – Upper confidence bounds (optimism in the face of uncertainty!)
  – Scrabble, Computer Go
Monte Carlo Tree Search (MCTS)

1. Consider all the next possible moves.

2. Sample possible continuations using a randomized policy for both players (typically until the end of the game).
   - Policy: mapping $S \rightarrow A$, for all states

3. Value of a move is the average of the evaluations from its sampled lines.

4. Pick the move with the best average (i.e. expected) value.
Recall Minimax

This would be “optimal” if we could do it

Main idea

• Near the top of the tree do Minimax.

• Then go to Monte Carlo to allow searching deeply.
Main idea

- Near the top of the tree do Minimax.
- Then go to Monte Carlo to allow searching deeply.
- Accumulate statistics at the nodes.
- As we get more information about the game, the Minimax portion can grow and the Monte Carlo portion can get shorter.

Example
Where to spend the search effort?

- If you limit the number of lines of play generated per turn, these do not have to be allocated equally to every move.

- Intuitively, look more closely at the promising moves, since the others would not be picked.

- Exact formulas exist to optimally control this allocation.

---

MCST Algorithm outline

1. **Descent phase**: Pick the highest scoring move for both players (based on what you know.)
   - Score can be the value of the child node, or use extra information.
MCST Algorithm outline

1. **Descent phase**: Pick the highest scoring move for both players (based on what you know.)
   - Score can be the value of the child node, or use extra information.

2. **Rollout phase**: When a leaf is reached, use Monte Carlo simulation to the end of the game (or to an affordable depth).
   - This uses a fixed, stochastic policy for both players.

3. **Update phase**: Update statistics for all nodes visited during descent.
MCST Algorithm outline

1. **Descent phase**: Pick the highest scoring move for both players (based on what you know.)
   - Score can be the value of the child node, or use extra information.

2. **Rollout phase**: When a leaf is reached, use Monte Carlo simulation to the end of the game (or to an affordable depth).
   - This uses a fixed, stochastic policy for both players.

3. **Update phase**: Update statistics for all nodes visited during descent.

4. **Growth phase**: First state in the rollout is added to the tree and its statistics are initialized.

---

Example

---

*Simulation 1*

- **Tree Policy**
- **Default Policy**

- New node in the tree
- Node stored in the tree
- State visited but not stored
- Terminal outcome
- Current simulation
- Previous simulation
Example

Simulation 2

Tree Policy

Default Policy

New node in the tree
Node stored in the tree
State visited but not stored
Terminal outcome
Current simulation
Previous simulation

Example

Simulation 3

Tree Policy

Default Policy

Simulation 4

Tree Policy

Default Policy
Example

Upper Confidence Trees (UCT)

\[ Q^\text{UCT}(s,a) = Q(s,a) + c \sqrt{\frac{\log N(s)}{m(s,a)}} \]
Upper bound on the value of taking action $a$ in state $s$

$$Q^\Theta(s, a) = Q(s, a) + c \sqrt{\frac{\log n(s)}{n(s, a)}}$$

Exploitation ($=$ current estimate)

$$Q^\Theta(s, a) = Q(s, a) + c \sqrt{\frac{\log n(s)}{n(s, a)}}$$
**UCT (Upper Confidence Trees)**

\[ Q^*(s, a) = Q(s, a) + c \sqrt{\frac{\log n(s)}{n(s, a)}} \]

---

**Scrabble**

- Stochastic game (letters drawn randomly).
- Imperfect information (can’t see opponent’s hand).
- Computers can have an advantage due to dictionary (move generation is easy).

- Quite complex!
  - ~700 branching factor
  - ~25 depth for a game
  - Rough complexity: \(10^{70}\) search states

- Strategy is difficult! What letters to keep???
Maven

- Win against national champion (9 games to 5) in 1997.
- Evaluates moves by score + value of tiles on player’s rack.
- Uses a linear evaluation function of the rack.
- Features: presence of 1, 2, and 3-letter combinations (allows detecting frequent pairs, like QU, and triples of hard-to-place letters).
- Weights trained by playing many games by itself and using Monte-Carlo rollouts.

Monte Carlo Tree Search in Maven

- For each legal move:
  - Roll-out, i.e. imagine $n$ steps of self-play (dealing times at random to both players).
  - Evaluate resulting position by score + value of rack (according to the evaluation function).
  - The score of the move is the average evaluation over rollouts.
  - Note that this can be done incrementally after each rollout:
    \[
    V_{k+1} = \frac{1}{k+1} \sum_{i=1}^{k+1} R_i = \frac{1}{k+1} \sum_{i=1}^{k} R_i + \frac{1}{k+1} R_{k+1} = \frac{k}{k+1} V_k + \frac{1}{k+1} R_{k+1}
    \]
- Play the move with the highest score.
The Story of Go

- The ancient oriental game of Go is 2000 years old.
- Considered to be the hardest classic board game.
- Long-considered to be a grand challenge task for AI.
- Traditional approaches to game-tree search are insufficient to win in Go.

The game of Go

- Game characteristics:
  - $\sim 10^{170}$ unique positions
  - $\sim 200$ moves long
  - $\sim 200$ branching factor
  - $\sim 10^{360}$ complexity
The Rules of Go

- Usually played on 19x19 board, also 13x13 or 9x9 board.
- Simple rules, complex strategy.
- Black and white place down stones alternately.

Capturing

- If stones are completely surrounded they are captured.
Winner

- The game is finished when both players pass.
- Intersections surrounded by each player are known as territory.
- The player with more territory wins the game.

Position Evaluation for AI

- Game outcome, $z$
  - Black wins, $z=1$
  - White wins, $z=0$

- Value of position, $s$
  - $V^\pi(s) = E^\pi [z | s]$ <= Monte-Carlo simulation
  - $V^*(s) = \text{minimax}_\pi V^\pi(s)$ <= Tree search
Monte-Carlo Simulation

\[ V(s) = \frac{2}{4} = 0.5 \]

Current position \( s \)

Simulation

Outcomes

1 1 0 0

Rapid Action-Value Estimate (RAVE)

- Assume the value of the move is the same no matter when the move is played.
- This introduces a “bias” (simplification) in thinking but will reduce some variability in the Monte-Carlo estimates.
**MC-RAVE in MoGo**

![Graph showing MC-RAVE in MoGo](image)

**MoGo (2007)**

- MoGo = heuristic MCTS + MC-RAVE + handcrafted default policy.
- 99% winning rate against best traditional programs.
- Highest rating on 9x9 and 19x19 Computer Go Server.
- Gold medal at Computer Go Olympiad.
- First victory against professional player on 9x9 board.
Progress in 19x19 Computer Go

Monte-Carlo tree search (vs $\alpha$-$\beta$ search)

**Advantages?**

**Disadvantages?**
Monte-Carlo tree search (vs $\alpha$-$\beta$ search)

**Advantages?**
- Not as pessimistic.
- Converges to the Minimax solution in the limit.
- Anytime algorithm: performance increases with number of lines of play.
- Unaffected by the branching factor (i.e. #actions):
  - Control the number of lines of play, so move can always be made in time.
  - If branching factor is huge, search can go much deeper, which is big gain.
- Easy to parallelize.

**Disadvantages?**
- May miss optimal play (because won’t see all moves at deeper nodes.)
- Policy used to generate candidate plays is very important. E.g. can use an opponent model, or just simple randomization.
Course project

- Modified version of a game called Hus (similar to Mancala).
- 4x16 board, 2 players, each controls 32 pits placed in two rows.
- Each player begins with 24 seeds arranged in a pre-specified pattern.
- Play by picking up seeds in one of your pits and “sowing” them one by one in their own pits moving counter-clockwise.
- The last player to make a move wins!

- **Goal:** Design the best possible AI player.
- Instructions, back-end code will be provided.