Lecture 7: Game Playing (Part 2)

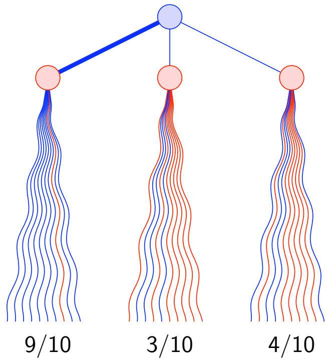
- Monte Carlo Tree Search (MCTS)
- Upper confidence bounds (optimism in the face of uncertainty again!)
- Scrabble
- Computer Go illustration
- Maybe: Poker and belief states

With thanks to David Silver

Recall: Game search

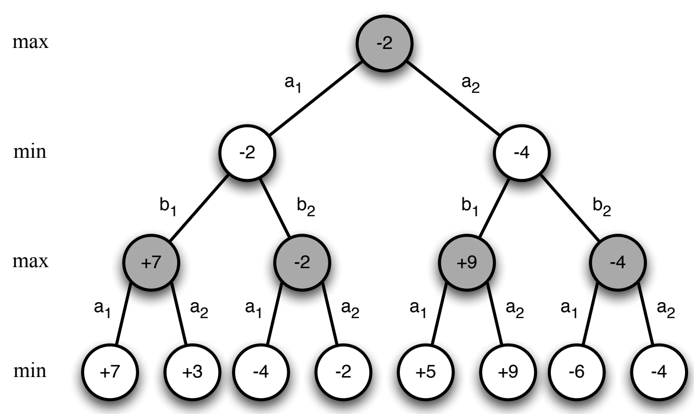
- We looked at perfect information, 2-player games
- α-β search can be used to cut off branching factor (but maybe not enough)
- Optimal play is guaranteed against an optimal opponent if search proceeds to the end of the game
- But the opponent may not be optimal!
- If heuristics are used, this assumption turns into the opponent playing optimally *according to the same heuristic function* as the player
- This is a very big assumption! What to do if the opponent plays very differently?

Monte Carlo tree search



- For each move, *sample possible continuation* using a randomized playing policy for both players
- Typically, the game is played until the end
- The value of the node is the *average* of the evaluations obtained at the end of the lines of play
- Pick the move with the best average value

Recall: Minimax

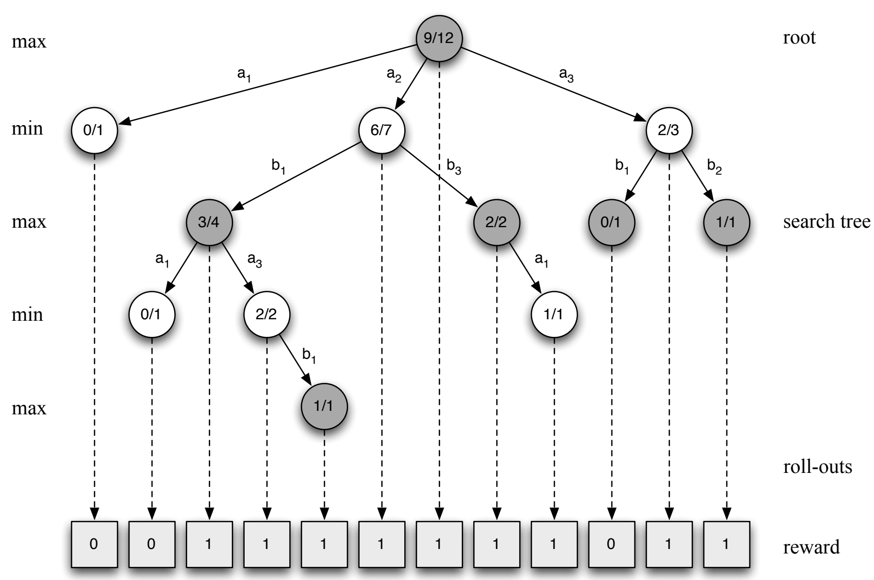


This would be "optimal" if we could do it

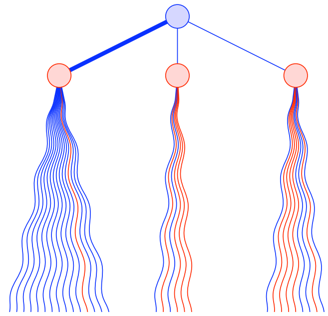
Main idea

- We can start with a completely randomized search
- In the beginning, we do minimax, but then go to Monte Carlo searches
- Accumulate statistics at the nodes
- As we get more information about the game, the "minimax" portion should grow and the Monte Carlo portion should get shorter

Example



Where to spend the search effort?



- If you limit the number of lines of play that will be generated per turn, these do not have to be allocated equally to every move.
- Intuitively, you should look more closely at the promising moves, since the others would not be picked
- Exact formulas can be established theoretically for this allocation

MCTS Algorithm Outline

We are going to grow a tree for the game, with each node having a value anotation

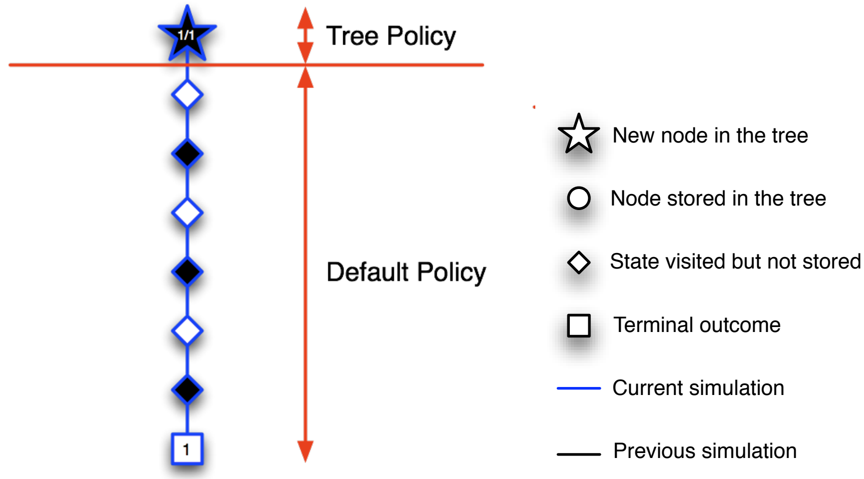
- Descent phase: Always pick the highest scoring move for both players (based on what you know)

  Score can be just the value of the child node, or can have extra information
- 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
- Update phase: statistics for all nodes visited during descent are updated
- Growth phase: the first state in the rollout is added to the tree and its statistics are initalized

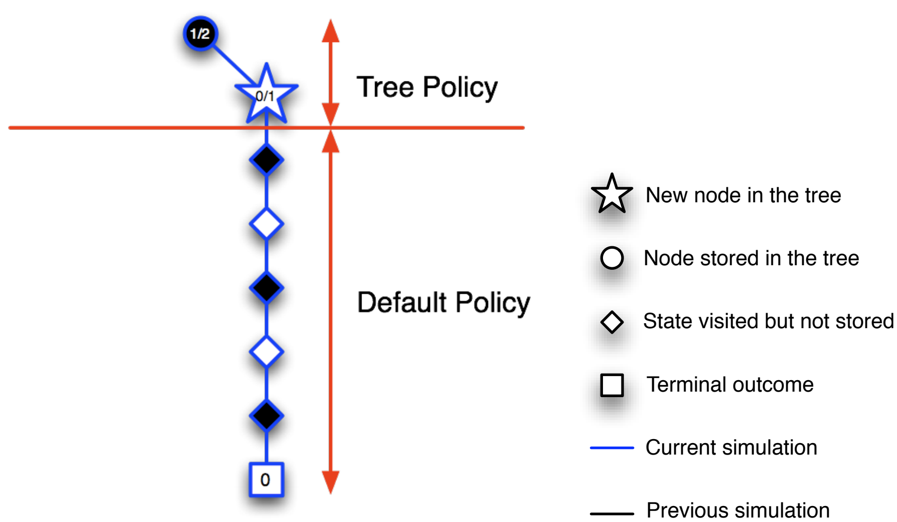
Example

Simulation 1

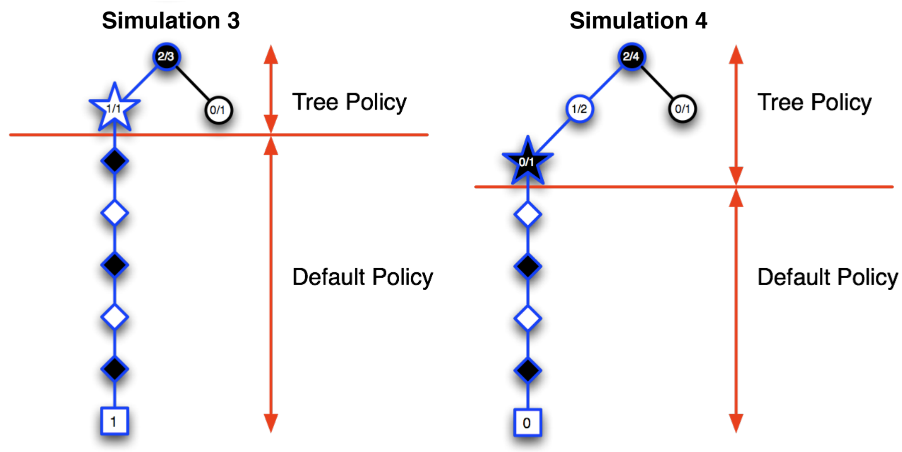


Example

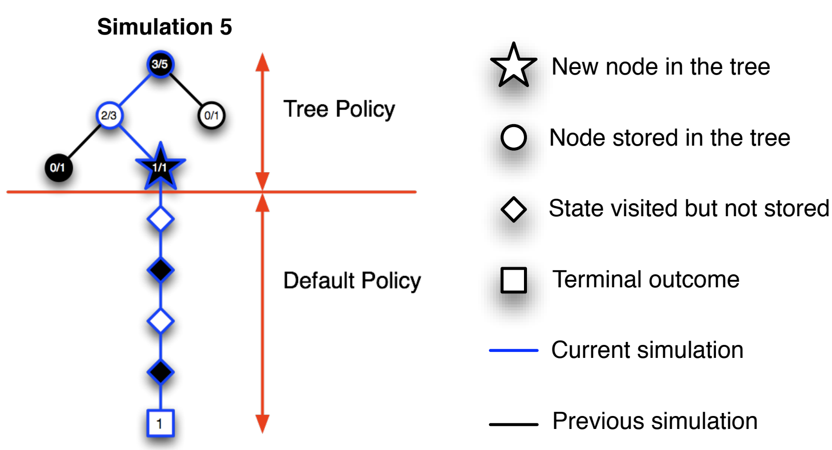
Simulation 2



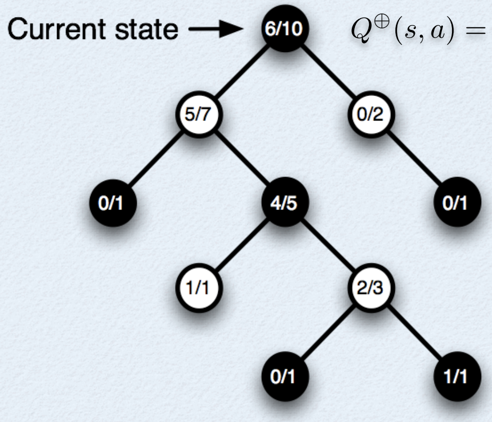
Example



Example

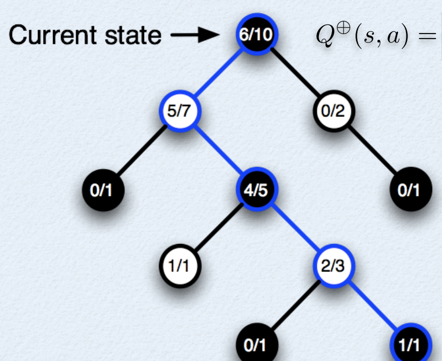


UCT (Upper Confidence Trees)

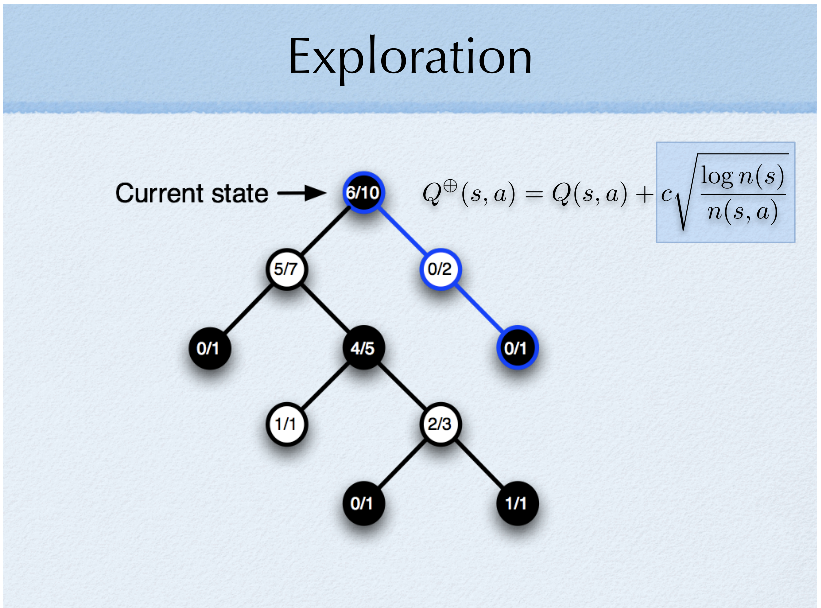


$$Q^{\oplus}(s,a) = Q(s,a) + c\sqrt{\frac{\log n(s)}{n(s,a)}}$$

Exploitation



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



Scrabble

- Stochastic (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
- *Strategy* is difficult: what letters to keep?

Maven

- Best player in the world (beat Adam Logan 9-5)
- Evaluates moves by score + value of rack
- Uses a binary-linear evaluation function of the rack left after the move
- 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, observing the final value of the game.

Monte Carlo Tree Search in Maven

1. For each legal move:
   (a) Roll-out, i.e. imagine n steps of self-play (dealing tiles at random to both players)
   (b) Evaluate resulting position by score + value of rack (according to the evaluation function)
   (c) The score of the move it the average evaluation over the rollouts
   (d) Note that this can be done incrementally after each rollout:

$$V_{k+1} = \frac{1}{k+1}\sum_{i=1}^{k+1} R_i = \frac{1}{k+1}\sum_{i=1}^{k} R_i + \frac{1}{k+1}R_{k+1} = \frac{k}{k+1}V_k + \frac{1}{k+1}R_{k+1}$$

2. Play the move with the highest score

The Game of Go

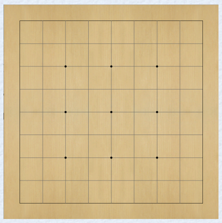
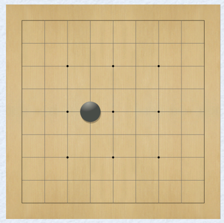
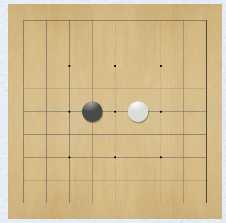
- ~10^{170} unique positions
- ~200 moves long
- ~200 branching factor
- ~10^{360} complexity

The Story of Go

- The ancient oriental game of Go is 2000 years old
- Considered to be the hardest classic board game
- Considered to be a grand challenge task for AI *(e.g. John McCarthy)*
- *Traditional approaches to game-tree search have failed in Go*

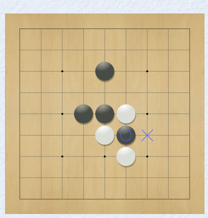
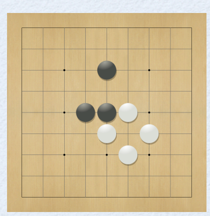
The Rules of Go

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

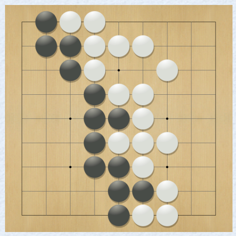
Capturing

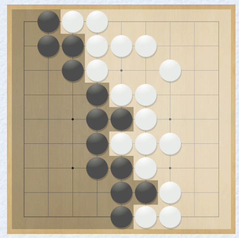
- If stones are completely surrounded they are captured

Winner

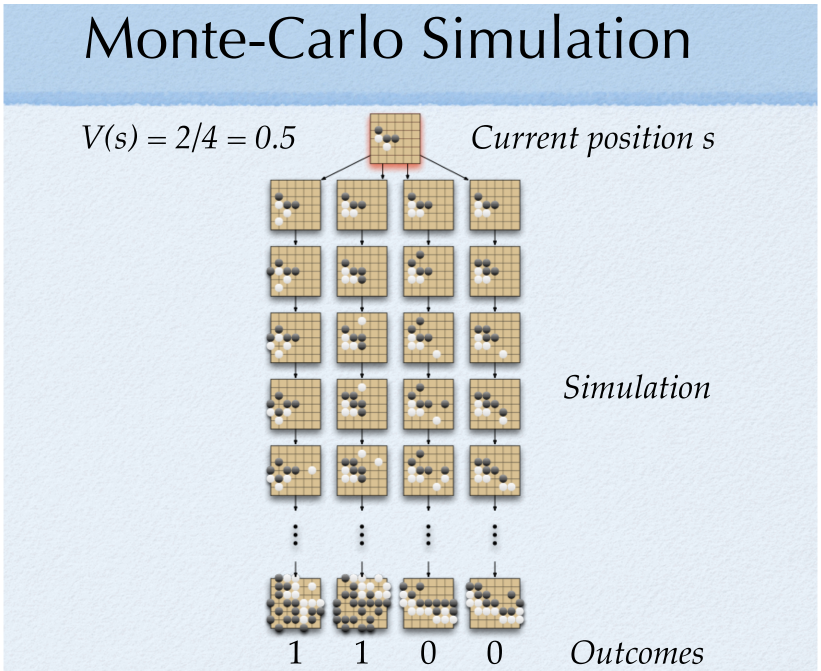
- The game is finished when both players pass
- The intersections surrounded by each player are known as *territory*
- The player with more territory wins the game





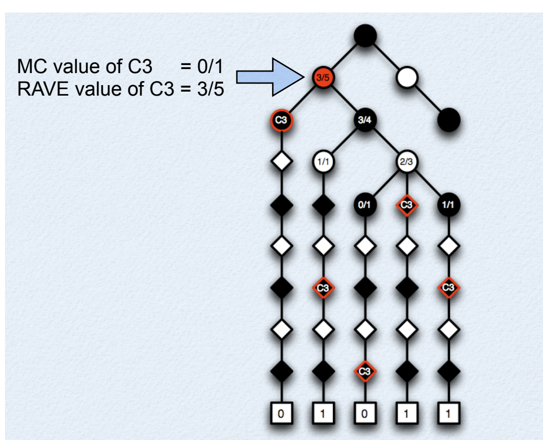
Position Evaluation

- Game outcome z
  - Black wins $z=1$
  - White wins $z=0$
- Value of position s
  - $V^{\pi}(s) = E^{\pi}[z \mid s]$ <= Monte-Carlo simulation
  - $V^{*}(s) = minimax_{\pi}\ V^{\pi}(s)$ <= Tree search

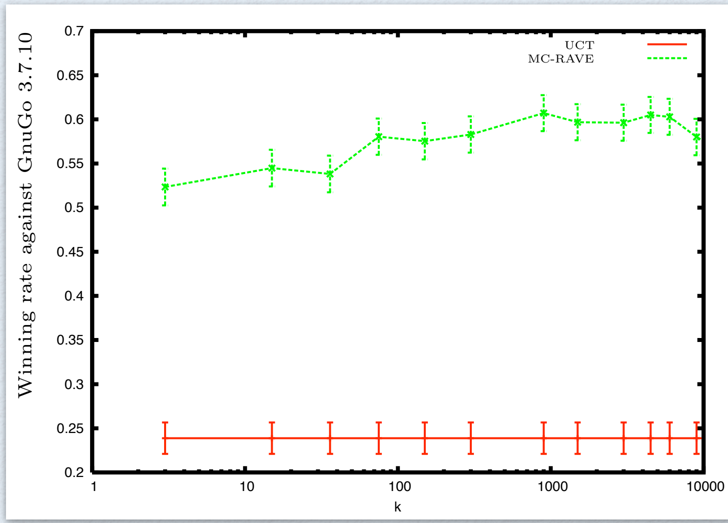


Rapid Action-Value Estimate (RAVE)

- Assume the value of the move is the same no matter when the move is played
- This will introduce a "bias" (simplification) in thinking but will reduce some variability



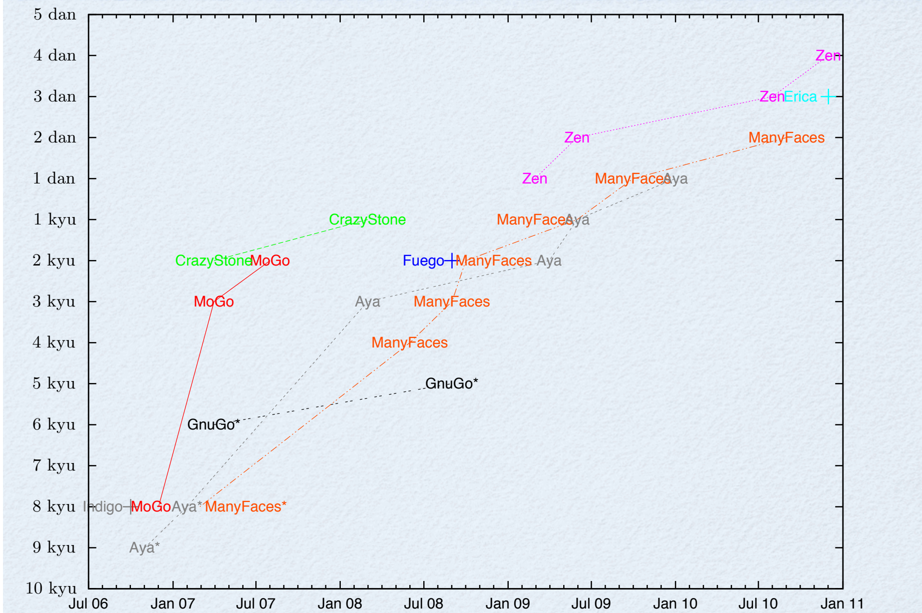
MC-RAVE in *MoGo*



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 9-dan professional player (9x9)

Progress in 19x19 Computer Go



Monte Carlo tree search vs. α-β search

- Not as pessimistic as α-β
- Converges to the minimax solution in the limit
- *Anytime algorithm*: performance increases with number of lines of play
- *Unaffected by branching factor*:
  - We control the number of lines of play, so a move can always be made on time
  - If the branching factor is huge, search can go much deeper, which is a big gain
- It is easy to parallelize the search
- We may miss on optimal play (because we will not even see all moves at deeper nodes)
- The policy used to generate the candidate plays is very important!

  E.g. can use an opponent model, or just make sure there is enough randomization.