Computing Science (CMPUT) 455

Search, Knowledge, and Simulations

James Wright

Department of Computing Science
University of Alberta
james.wright@ualberta.ca

Fall 2021

455 Today - Lecture 16

- Today: Selective search and Monte Carlo Tree Search (MCTS)
- Finish discussion of UCB from Lecture 15
- Comparison and overview:
  Exact search
  Selective search
  Simulations
- Monte Carlo Tree Search framework
- UCT algorithm
- Enhancements of MCTS

Coursework

- Work on Assignment 3
- Reading: Pedro Domingos, A Few Useful Things to Know about Machine Learning
- Quiz 9 Monte Carlo Tree Search. Double length
- Activities

Exact Search, Selective Search, and Simulations

- Big-picture overview of algorithms so far
- For each method, focus on three questions:
  1. Which parts of the game tree does it visit?
  2. How does it back-up results to the root of the tree?
  3. Exact or selective?

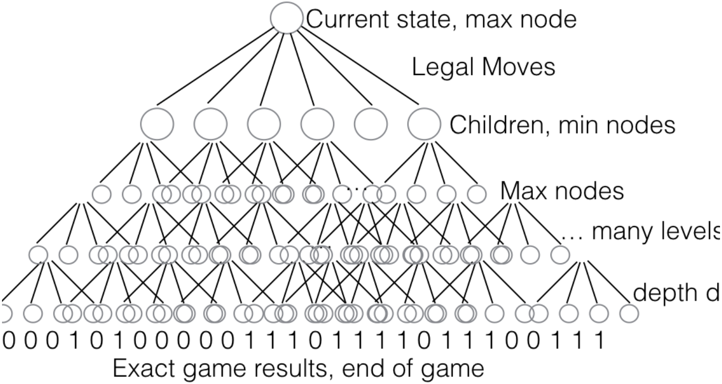
Review - (b,d) Tree Model and Solving a Game

- Search space in (b,d) tree model:
- Branching factor b, depth d
- Alternating min and max levels in tree
- b^d leaf nodes
- $(b^{d+1} - 1)/(b - 1) \approx (b * b^d)/(b - 1) \approx b^d$ nodes in whole tree
- Size of proof tree in best case: very roughly $b^{d/2}$
- Minimum amount of search to solve a game

Naive Minimax (or Negamax) - Exact Solver

1. Which parts of the game tree does it visit?
    - Explores the full game tree
    - All children searched in each node
2. How does it back-up results to the root of the tree?
    - Minimax:
      Minimum over children at min nodes,
      maximum at max nodes
    - Negamax is a different but equivalent formulation, same result
3. Exact or selective?
    - Exact
    - Terminal nodes are true end-of-game
    - Uses only exact scores at terminal nodes for evaluation
    - Result is proven correct

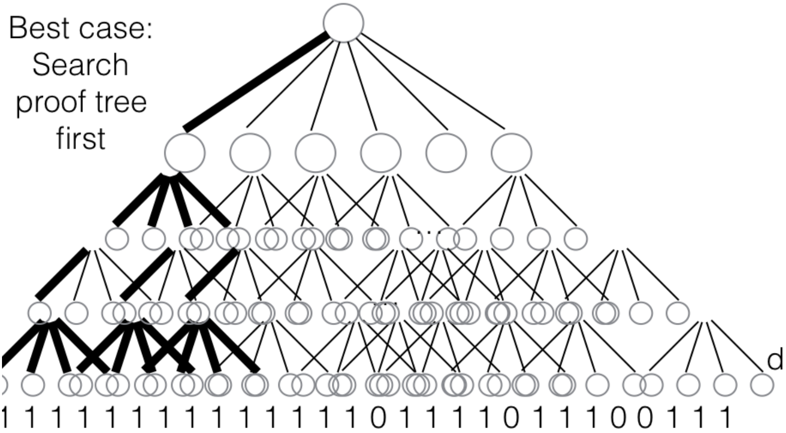
Naive Minimax - Exact Solver



Efficient Minimax (or Negamax) - Boolean Minimax, Alphabeta

1. Which parts of the game tree does it visit?
   - Some parts of tree may be cut by exact pruning rules
   - Best case: Visit only 1 child for winner
   - Needs to try all moves for loser
2. How does it back-up results to the root of the tree?
   - Minimax
   - For alphabeta, some back-up values are "good-enough" upper or lower bounds, not exact values
3. Exact or selective?
   - Exact.

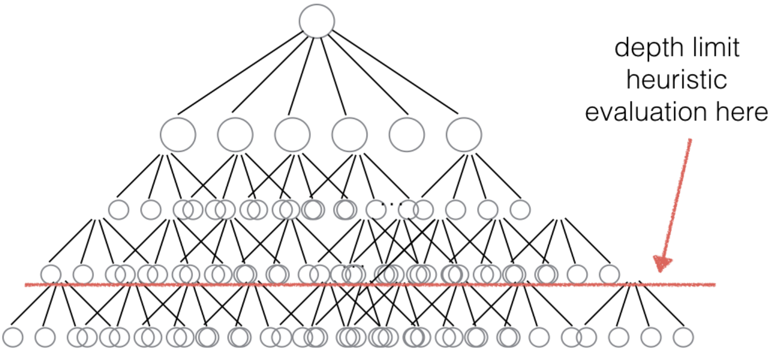
Efficient Minimax (or Negamax) - Boolean Minimax, Alphabeta



Depth-limited Alphabeta Search

1. Which parts of the game tree does it visit?
   - As in alphabeta, but only up to depth limit
2. How does it back-up results to the root of the tree?
   - Min and max
3. Exact or selective?
   - Selective
   - Heuristic evaluation at terminal nodes
   - Search process is exact, but evaluation of leaves is not
   - Source of error: heuristic evaluation in leaf nodes

Depth-limited Alphabeta Search



Selective Alphabeta Search with Fixed Time or Node Budget

- For many games even $O(b^{d/2})$ nodes for best-case proof is far too large
- In practice: fixed time or node limit for the search (e.g. 30 seconds, or 10^{12} nodes)
- What can we search within that budget?
- First answer was depth-limited search: reduce d until search fits within budget
- *New: Second answer - selective search*: reduce both b and d until search fits within budget

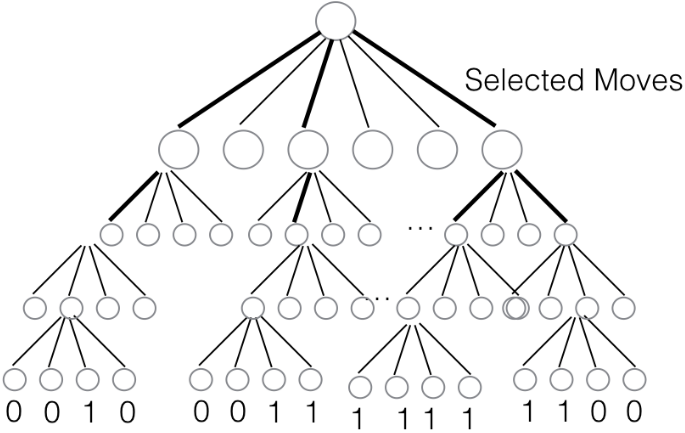
Selective Alphabeta Search - Methods

- How to do selective search?
- Search “interesting” moves much deeper than others
- Choice 1: Prune moves by using selective minimax algorithms such as ProbCut (Buro) or Nullmove pruning
- Choice 2: Prune moves using knowledge
  - Details: `https://www.chessprogramming.org/Selectivity`
- Choice 3: expand search tree selectively
  - Example: Monte Carlo Tree Search (MCTS)

Selective Alphabeta Search

1. Which parts of the game tree does it visit?
   - Does not consider all legal moves in each node
   - Often depth-limited as well
2. How does it back-up results to the root of the tree?
   - Min and max
3. Exact or selective?
   - Selective
   - Heuristic evaluation at terminal nodes
   - Skips some legal moves
   - Source of error: heuristic evaluation in leaf nodes
   - Source of error: may prune the best move from a node

Selective Alphabeta Search



Selective Alphabeta Search for Large Problems

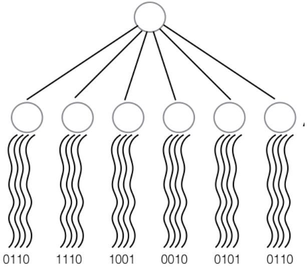
- Large problems (chess, checkers, ...)
- Reducing *b* not enough
- Reduce both *b* and *d* - selective search with heuristic evaluation
- Before Monte Carlo, this was the standard approach for most complex games

Selective Alphabeta Search for Go?

How about Go?

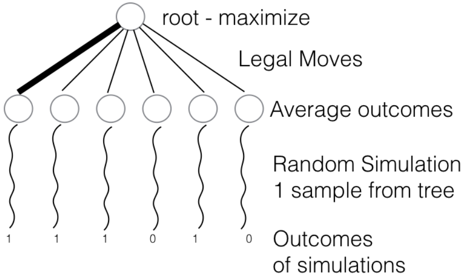
- $>$ 200 moves on average for 19x19 Go
- Usually, only 1-10 of them are good
- Can we reduce *b* down to this range, without missing important good moves?
- Many attempts failed in the past - too many good moves missed
- MCTS was the first approach that worked well
- Later, strong move selection heuristics based on neural nets also helped a lot
  - Neural nets were not tried much with alphabeta, since MCTS worked so well in Go

Simulation-based Players



- Review: simple simulation-based players (e.g. Go3)
- 1 ply search at the root
- Move selection - simple or UCB
- Simulations - (almost) random, rule-based, or probabilistic
- How do these algorithms compare to selective search?

Simulation-Based Player as Selective Minimax Search



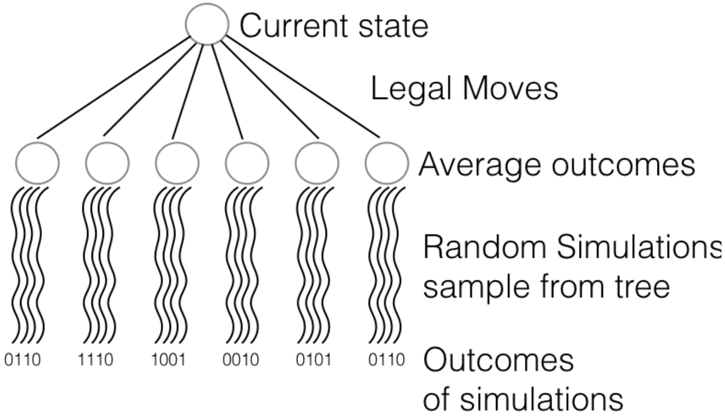
- Extreme case of selective search:
- Simulation-based player with one simulation per move
- Branching factor *b* at root, complete search
- Branching factor 1 at all later levels...

Simulation-Based Player with Repeated Sampling

Repeated sampling in Simulation Player

- With small number of samples
  - Samples a few moves close to the start
  - does not improve the branching factor lower in the tree
- With large, unlimited number of samples
  - Eventually samples all nodes in the full (b,d) tree infinitely often
  - With selective policy (e.g. patterns, filters), samples some subtree infinitely often

Simulation-based Player - Uniform Random Simulation Policy



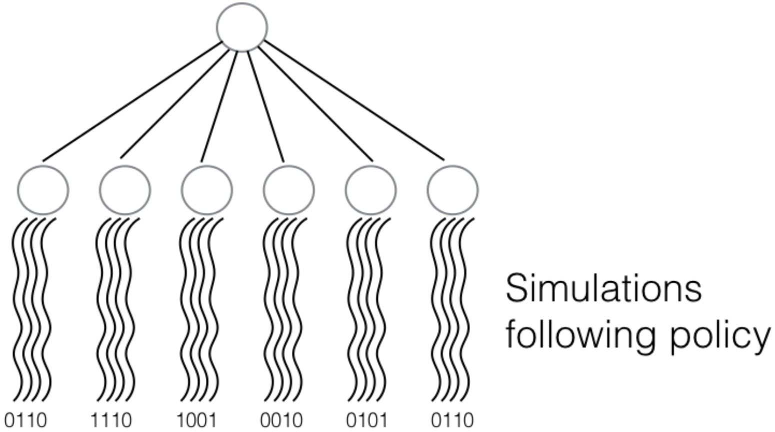
Simulation-based Player - Uniform Random Simulation Policy

1. Which parts of the game tree does it visit?
   - Eventually, visits all nodes
2. How does it back-up results to the root of the tree?
   - Max at root only
   - Average over all simulations
3. Exact or selective?
   - Selective
   - Not exact because of averaging instead of minimax
   - Source of error/risk: bias - average may be far from min, max
   - Source of error: variance - large uncertainty with small number of samples

Simulation-based Player - Non-Uniform Simulation Policy

1. Which parts of the game tree does it visit?
   - All, except subtree below moves that are never selected by policy
2. How does it back-up results to the root of the tree?
   - Same as Uniform Random: 1-ply max + average over simulations
3. Exact or selective?
   - Selective
   - Similar to Uniform Random
   - Strength: average over better samples may be closer to min, max
   - Risk: can miss totally by hard-pruning all good moves

Simulation-based Player - Non-Uniform Simulation Policy



- Some nodes in tree may never be sampled:
- If some move on path to node never selected by policy

Simulation-based Player - Simple vs UCB Move Selection

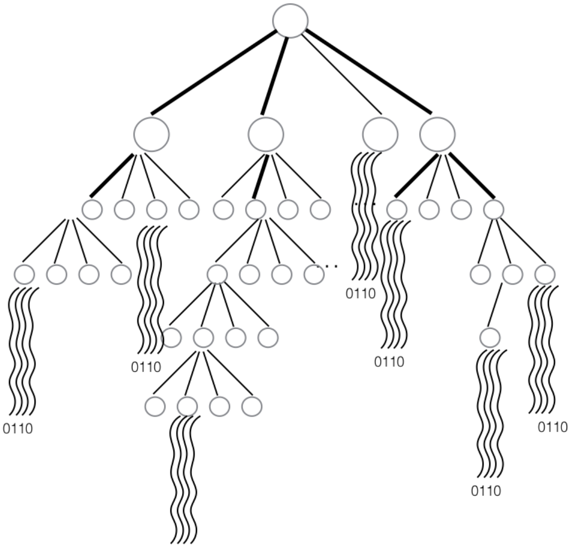
- Move selection:
  both simple and UCB behave the same in principle
- Both compute average over all simulations
- Difference: in UCB, average is taken:
  - Over more simulations for good move
  - Over fewer simulations for bad move
  - With a tree, in MCTS with UCT, this will be important
- Another difference:
  - UCB selects most-simulated move (**Why?**)
  - Simple selects move with highest winrate
  - These moves are usually, but not always the same

Monte Carlo Tree Search (MCTS)

Next algorithm: Monte Carlo Tree Search (MCTS)

1. Which parts of the game tree does it visit?
   - Tree search at the start, simulations to finish
2. How does it back-up results to the root of the tree?
   - Weighted averages over children
   - Weight of child = number of simulations for that child
   - Approaches min, max if best child has much higher weight than rest
3. Exact or selective?
   - Selective
   - Much deeper search for moves with better winrates
   - Converges to exact if given enough time to grow whole tree
   - Weighted average

Monte Carlo Tree Search



Monte Carlo Tree Search

- Weakness of simulation-based players so far:
- No tree search after move 1
- Everything from move 2 is random(ized) simulations only

MCTS + UCT approach

- Add selective tree search
- Adapt UCB idea to work in trees - UCT algorithm
- UCT = Upper Confidence bounds on Trees
- Run simulation from leaf of tree for evaluation

Adding a Game Tree to Simulation-Based Player

- First idea: combine what we have:
  - Depth-limited alphabeta
  - Evaluation by simulation
- This fails miserably.
  - Too noisy - need many simulations to get reasonably stable evaluation
  - Too slow - even 1-ply simulation-based player is slow
  - Result: Simulation-based approaches were ignored for over 10 years in Go
- Smarter way to combine search and simulation
  - MCTS, UCT

Monte Carlo Tree Search(MCTS) Model

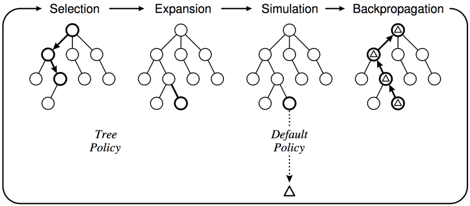


Fig. 2. One iteration of the general MCTS approach.

Image source: Browne et al, A Survey of Monte Carlo Tree Search Methods

Four steps, repeated many times

- Selection - traverse existing tree using formula such as UCT to select a child in each node
- Expansion: add node(s) to tree
- Simulation: follow randomized policy to end of game
- Backpropagation: update winrates along path to root

Using MCTS to Play Games

- To play one move:
  - Run MCTS search from current state
  - After search: select best move at root, play it
- To play a whole game:
  - Run MCTS every time it is the program's turn
  - May store and re-use parts of tree from previous search

Monte Carlo Tree Search(MCTS) Model

Algorithm 1 General MCTS approach.

function MCTSSEARCH(s_0)
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow$ TREEPOLICY(v_0)
    $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
    BACKUP(v_l, Δ)
  return a(BESTCHILD(v_0))

Image source: Browne et al, A Survey of Monte Carlo Tree Search Methods

- v_l = leaf node in tree
- Δ = result of simulation
- a(..) = action to move to best child

Monte Carlo Tree Search Example

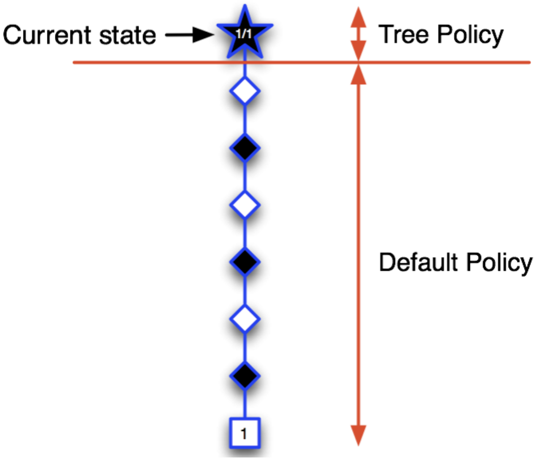


Image source: David Silver

Monte Carlo Tree Search Example

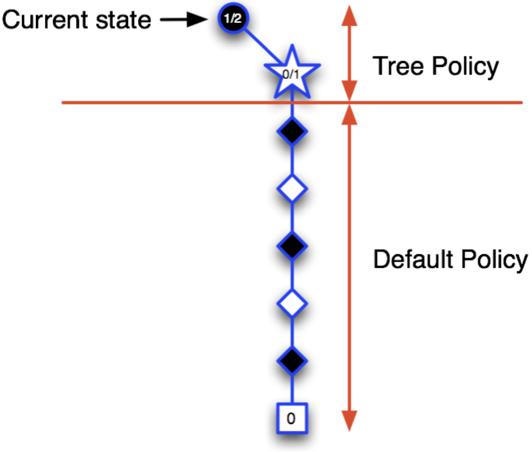


Image source: David Silver

Monte Carlo Tree Search Example

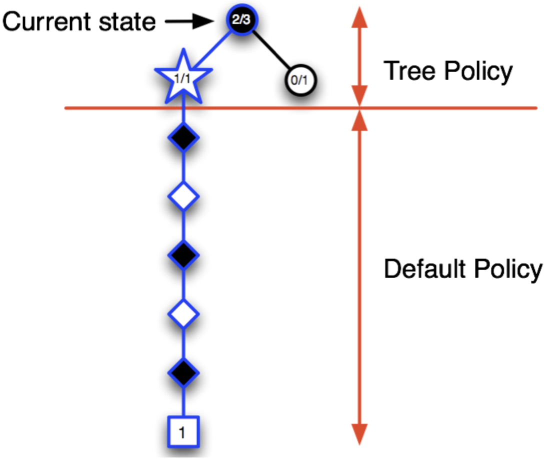


Image source: David Silver

Monte Carlo Tree Search Example

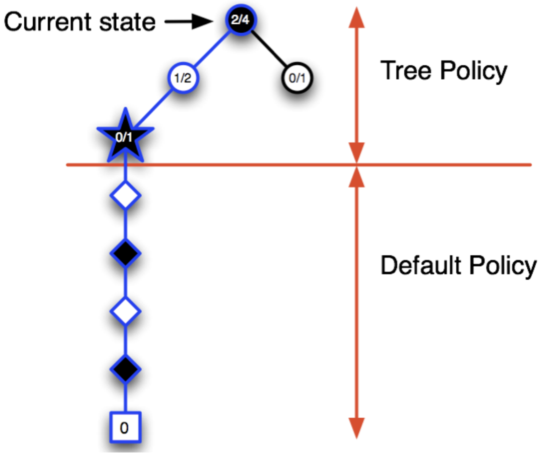


Image source: David Silver

Monte Carlo Tree Search Example

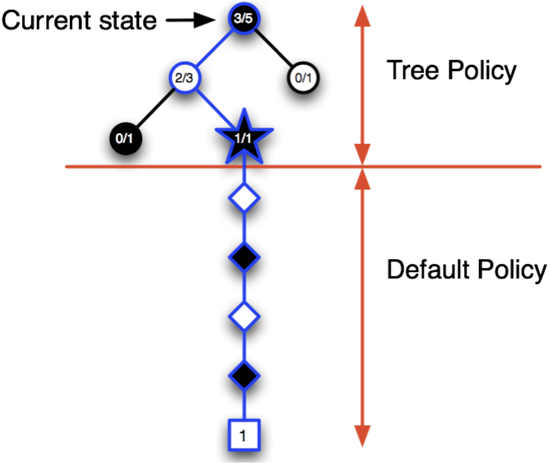


Image source: David Silver

MCTS Tree Traversal

- Start from root of tree
- Repeat:
  - Go to best child
  - Until reached leaf node in tree
- What is the best child?
- Use a formula to evaluate all children
  - UCT is popular (see next slides)
- Many other extended formulas are possible
  - Example: add knowledge-based term

From UCB to UCT

- UCT algorithm by Kocsis and Szepesvari (2006)
- It is still *the* classic algorithm for Monte Carlo Tree Search
- It is not the first child selection algorithm used in MCTS ...
- ... but it is the first based on sound theory
- Worked better in practice than earlier ad hoc algorithms
- Original paper has over 2200 citations -
  hugely influential

UCT Algorithm Main Ideas

- Algorithm for child selection in Monte Carlo Tree Search
- Name UCT is often used for MCTS with this algorithm
- Combines tree search with simulations
- Uses results of simulations to guide growth of the game tree
- Uses UCB-like rule to select “best” child of a tree node
- Goal: select a good path in the tree to explore/exploit next
- Grows the tree over time
- Stores winrate statistics in each node, used for child selections

Exploration vs Exploitation

- Like UCB, UCT tries to balance Exploration and Exploitation
- Exploitation: focus on most promising moves
- Exploration: focus on moves where uncertainty about evaluation is high
- Difference: evaluate UCT formula in every node along a path in the search tree

From UCB to UCT

- Review - UCB formula

$$UCB(i) = \hat{\mu}_i + C\sqrt{\frac{\log N}{n_i}}. \quad (1)$$

- UCT is very similar: UCT value of move i from parent p:

$$UCT(i) = \hat{\mu}_i + C\sqrt{\frac{\log n_p}{n_i}}. \quad (2)$$

- Only difference in exploration term
  - UCB: uses global count of all simulations N
  - UCT: uses simulation count of parent n_p
- For root, UCT is identical to UCB
  - N = simulation count of root

MCTS Tree Expansion

- How to grow the tree?
- Simplest case: add one node per iteration
- Add one node from current simulation
- Tree grows very selectively
  - paths with strong moves become much deeper than others
- If memory fills too quickly:
  - Use an *expansion threshold* t_e
  - Only add a node if the leaf has at least t_e visits
  - Example: Fuego program, default $t_e = 3$

MCTS Simulations

- Run one simulation from the leaf node of tree
- Can use any simulation policy
  - Uniform random, rule-based, or probabilistic
- Result of simulation is win (1) or loss (0)
- Can run more than one simulation from each leaf node
  - Tradeoff between speed and accuracy
  - Tradeoff between time spent in updating tree vs running simulations
  - Example: for Fuego, on some hardware 2 simulations per leaf works better than 1

MCTS Backpropagation - Update Statistics

- Update wins and visit counts along path to root
- Negamax style implementation - flip wins/losses at each step
- `value = 1-value` changes from wins to losses and back

```
def backprop(node, value):
    while node:
        node._wins += value
        node._n_visits += 1
        value = 1 - value
        node = node._parent
```

MCTS Move Selection

- Run as many iterations of MCTS as you can
- Then select move to play at root
- How?
- Browne's paper mentions several approaches
- We discuss the main ones

MCTS Move Selection

- Max child: child with highest number of wins
- Robust child: Select the most visited root child. (This is popular)
- Highest winrate
  - Not a good/stable method with MCTS
  - Why not stable: see next slides
- Max-Robust child (see later slide)

Dangers of Selecting Move by Winrate in MCTS

- MCTS usually expands the move with best winrate (exploitation)
- But sometimes, it explores an inferior-looking move
- This can lead to trouble for selecting a move by best winrate
- A move with low simulation count and high uncertainty about its value might get selected
- See example next slide

Dangers of Selecting Move by Winrate in MCTS

- Example: two moves A and B
- A 78 wins / 100 visits, winrate 78%
- B 6 wins / 8 visits, current winrate 75%
- Assume B has higher UCT score, so we explore B
- B gets a win, now has 7 wins / 9 visits, current winrate 77.8%
- Explore B again
- B gets another win, now has 8 wins / 10 visits, current winrate 80%
- Assume we stop search now

Dangers of Selecting Move by Winrate in MCTS

- A 78 wins / 100 visits, winrate 78%
- B 8 wins / 10 visits, **winrate 80%**
- If we select B because of highest winrate:
- High risk of being wrong
- The value of A is much more certain
- The value of B still has much higher variance
- Remember discussion of binomial distribution of simulations
- Probability of error is high

Max-Robust Child: Extending Search

- What if most-simulated move and highest winrate move are different?
- Search may just have found a new best move
  - B is really better than A
- Or B may be a fluke
  - B got some "lucky" wins, but is worse than A in the long run
- Very little evidence to decide which is true
- One solution: extend the search in such cases

Max-Robust Child: Extending Search

- Extending the search can distinuish two cases:
- If B is really good:
  - B will now receive many more simulations soon, stabilize value
- If B's recent wins were a fluke:
  - Its winrate and upper confidence bound will drop quickly with more simulations
- Extending search in this way is called "Max-Robust child" in the paper

Improving MCTS

- Many ways to improve:
- Adding knowledge in tree or in simulation
- Modify in-tree selection
- Modify or replace simulations
- We will discuss several good options when we talk about machine learning and AlphaGo

Summary

- Overview of game tree search and simulation
- Discussed Monte Carlo Tree Search
- After all the preparation, MCTS mostly combines previously discussed concepts
- 4+1 steps of MCTS
    - Repeat: select, expand, simulate, backpropagate
    - Finally: select move to play

Memory-Augmented Monte Carlo Tree Search

- Paper by Chenjun Xiao, Jincheng Mei and Martin Müller
- An improvement of MCTS
- Outstanding paper award at the 2018 AAAI conference
- Here: short, nontechnical summary of the ideas
  - Credit: most pictures and some bullet points taken from Chenjun's AAAI talk
- Interested in technical details?
  - Read the paper on Martin's publications page `http://webdocs.cs.ualberta.ca/~mmueller/publications.html`
  - Look at the technical talk on Martin's talks page `https://webdocs.cs.ualberta.ca/~mmueller/talks.html`

Main Idea

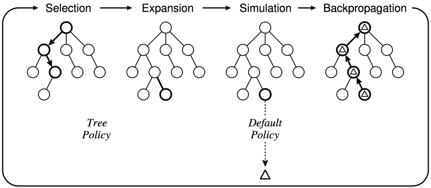
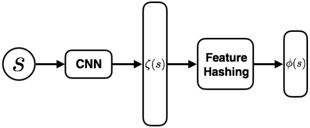


Fig. 2. One iteration of the general MCTS approach.

- Problem of MCTS:
- Most nodes are leaves or near leaf
- Most nodes have few simulations
- Evaluation is noisy

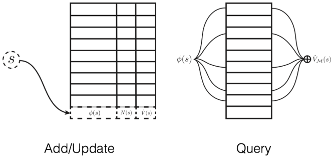
- Can we improve it?
- Approach: find similar states
- Use values of similar states to improve evaluation

Feature Representation for States



- How to define *similar* states?
- Represent state as vector of features
- States are similar if they share lots of features
- In this paper, features are defined by
  - Using a layer of a neural net
  - Using an unbiased hashing technique to reduce number of features

Memory



- Store for state *s*:
  - Feature vector of *s*
  - Pointer back to *s* to lookup its value (wins / visits)
  - As we do more search, value becomes better
- Lookup new state *s*:
  - Compute memory value as weighted sum of similar states in memory

Finding Similar States in Memory

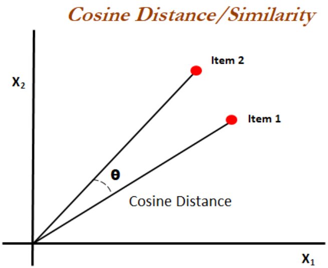
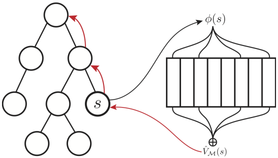


Image source: `https://www.safaribooksonline.com/library/view/statistics-for-machine`

- Compare two feature vectors
- Similar if they "point in similar direction"
- Measure: cosine similarity
- A standard similarity measure in machine learning
- Larger is better, similarity 1 if they have same direction
- Math: see `https://en.wikipedia.org/wiki/Cosine_similarity`

Using Memory with MCTS

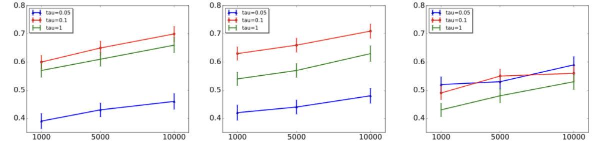


- Selection: compute state value by linear combination of state value $\hat{V}_s$ and memory value $\hat{V}_{\mathcal{M}}$

$$V(s) = (1 - \lambda_s)\hat{V}_s + \lambda_s \hat{V}_{\mathcal{M}}$$

- Evaluation: evaluate state by both Monte Carlo and memory
- Backup: update MC value and memory value in tree

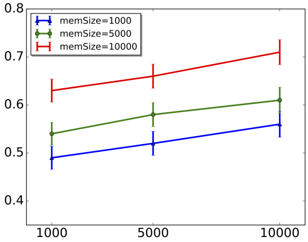
Experiment 1



M=20 M=50 M=100

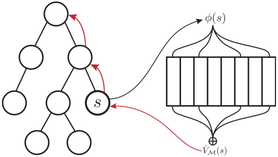
- Play games Fuego + M-MCTS against normal Fuego
- Vary neighbourhood size M
- τ is a “temperature” parameter in the algorithm
- X-axis: number of simulations/move
- Y-axis: winrate against Fuego

Experiment 2



- Varying Memory Size
- Keep neighborhood size M and τ constant

Summary of M-MCTS



- MCTS has very few samples on most nodes near the leaves
- We can "interpolate" the value of similar nodes
- This gives a better evaluation
- Not in this summary (read the paper...):
- Math. framework and proof that this gives better values with high probability

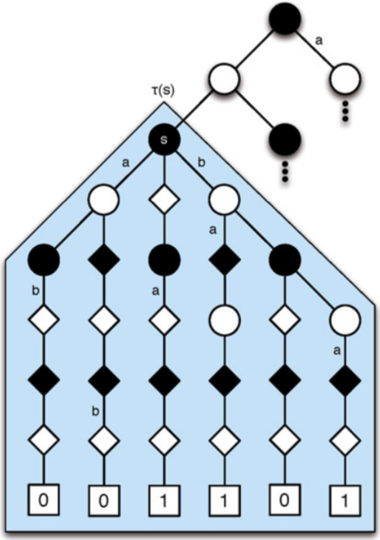
RAVE: Rapid Action Value Estimation

- Sylvain Gelly, David Silver, "Monte-Carlo tree search and rapid action value estimation in computer Go", 2011.
- One of many extension to MCTS
- Originally proposed for Go but can be generalized for other games
- Works especially well for NoGo

Main Idea: All Moves as First (AMAF)

- *All moves as first* is a heuristic that is useful for games that can be decomposed into independent subgames
  - Each move has its own value (as opposed to each move at a specific position having its own value)
  - The value for a move in one subgame is not affected by moves in other subgames
  - The subgames can be played in any order
- You can treat all moves played in a trajectory as if it is the first move

RAVE Example

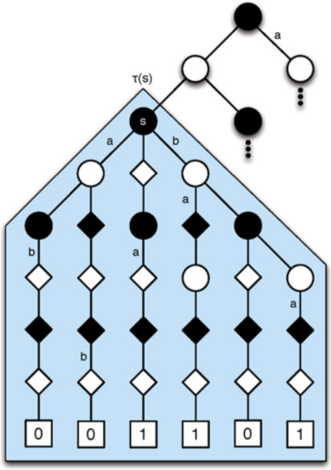


$$Q(s,a) = 0/2$$
$$Q(s,b) = 2/3$$
$$\tilde{Q}(s,a) = 3/5$$
$$\tilde{Q}(s,b) = 2/5$$

Terminology

- $\hat{\mu}_i = w_i/n_i$ is the win rate or MC value of the move i
- $\tilde{\mu}_i = \tilde{w}_i/\tilde{n}_i$ is the AMAF heuristic value of the move i
    - $\tilde{n}_i$ is incremented each time i is played in any trajectory within the same subtree
    - $\tilde{w}_i$ is incremented each time a win is the result of i being played in any trajectory within the same subtree

The RAVE Algorithm



$Q(s,a) = 0/2$
$Q(s,b) = 2/3$
$\tilde{Q}(s,a) = 3/5$
$\tilde{Q}(s,b) = 2/5$

- If you use the AMAF heuristic instead of the MC value of each node in your search tree, you get RAVE

RAVE Properties

- RAVE is similar to the *history heuristic* in alpha-beta search
  - History heuristic is a dynamic heuristic that keeps track of which moves cause the most beta-cuts and tries them first
  - RAVE also keeps track of which moves are most successful at different depths of the search
- RAVE gets more samples for each move *i*
  - Everytime *i* is played in the same subtree, it counts as if it is sampled once
  - You end up with many more samples of *i*, so it learns faster
  - This can actually be bad, because the same move played at different times can mean very different things

The MC-RAVE Algorithm

- If you combine the MC value with the AMAF heuristic, you get the MC-RAVE algorithm

$$\mu_i = (1 - \beta)\hat{\mu}_i + \beta\tilde{\mu}_i. \tag{3}$$

- β is a scheduling parameter that gives more or less emphasis to the two evaluations $\hat{\mu}$ (MC value) and $\tilde{\mu}$ (AMAF)
  - The general rule of thumb is to make β dynamic
  - Have β decay as the number of samples increase
  - In other words, use RAVE because it is useful but inaccurate, then transition over to the more accurate MC value as the number of samples increase

The UCT-RAVE Algorithm

- Lastly, you combine MC-RAVE (an evaluation scheme) with the UCT

$$UCT(i) = \mu_i + C\sqrt{\frac{\log n_p}{n_i}}. \tag{4}$$

- You can initialize the values with knowledge
  - For $\hat{\mu}_i$, $\tilde{\mu}_i$, 0.5 is a good reference
  - For n_i, you can set to 0
  - For $\tilde{n}_i$, you can set to 20 or 40

The Scheduling Constant β

- The β value is set to the following

$$\beta = \sqrt{\frac{k}{3n_p + k}} \tag{5}$$

- For $\beta = 1/2$, i.e. the point at which the MC value and the AMAF value has equal weighting, we can see that $k = n_p$
  - If we set $k = 100$, that means after p is sampled 100 times, MC and AMAF values are given equal weighting
- It is a heuristic that has been shown to be effective for Go and NoGo
- The original paper includes another scheduling method for β which has a stronger mathematical basis

UCT vs. MC-RAVE