

From Deep Blue to Monte Carlo: An Update on Game Tree Research

Akihiro Kishimoto and Martin Müller



AAAI-14 Tutorial 5:
Monte Carlo Tree Search

Presenter:
Martin Müller, University of Alberta



Tutorial 5 – MCTS - Contents

Part 1:

- Limitations of alphabeta and PNS
- Simulations as evaluation replacement
- Bandits, UCB and UCT
- Monte Carlo Tree Search (MCTS)

Tutorial 5 – MCTS - Contents

Part 2:

- MCTS enhancements: RAVE and prior knowledge
- Parallel MCTS
- Applications
- Research challenges, ongoing work

Go: a Failure for Alphabeta

- Game of Go
- Decades of Research on knowledge-based and alphabeta approaches
- Level weak to intermediate
- Alphabeta works much less well than in many other games
- Why?

Problems for Alphabeta in Go

- Reason usually given: Depth and width of game tree
 - 250 moves on average
 - game length > 200 moves
- **Real reason: Lack of good evaluation function**
 - Too subtle to model: very similar looking positions can have completely different outcome
 - Material is mostly irrelevant
 - Stones can remain on the board long after they “die”
 - Finding safe stones and estimating territories is hard

Monte Carlo Methods to the Rescue!

- Hugely successful
 - Backgammon (Tesauro 1995)
 - Go (many)
 - Amazons, Havannah, Lines of Action, ...
- Application to deterministic games pretty recent (less than 10 years)
- Explosion in interest, applications far beyond games
 - Planning, motion planning, optimization, finance, energy management,...

Brief History of Monte Carlo Methods

- 1940's – now Popular in Physics, Economics, ... to simulate complex systems
- 1990 (Abramson 1990) expected-outcome
- 1993 Brüggmann, *Gobble*
- 2003 – 05 Bouzy, Monte Carlo experiments
- 2006 Coulom, *Crazy Stone*, **MCTS**
- 2006 (Kocsis & Szepesvari 2006) **UCT**
- 2007 – now *MoGo*, *Zen*, *Fuego*, many others
- 2012 – now MCTS survey paper (Browne et al 2012); huge number of applications

Idea: Monte Carlo Simulation

- No evaluation function? No problem!
- Simulate rest of game using random moves (easy)
- Score the game at the end (easy)
- Use that as evaluation (hmm, **but...**)

The GIGO Principle

- **Garbage In, Garbage Out**
- Even the best algorithms do not work if the input data is bad
- How can we gain any information from playing random games?

Well, it Works!

- For many games, anyway
 - Go, NoGo, Lines of Action, Amazons, Konane, Disconnect, ..., ..., ...
- Even random moves often preserve *some* difference between a good position and a bad one
- The rest is statistics...
- ...well, not quite.

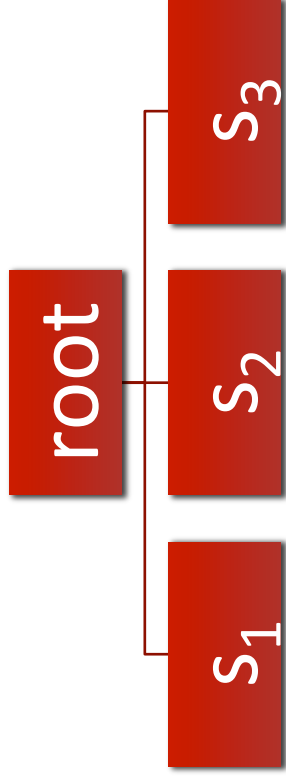
(Very) Basic Monte Carlo Search

- Play lots of random games
 - start with each possible legal move
- Keep winning statistics
 - Separately for each starting move
- Keep going as long as you have time, then...
- Play move with best winning percentage

Simulation Example in NoGo

- Demo using *GoGui* and *BobNoGo* program
- Random legal moves
- End of game when *ToPlay* has no move (loss)
- Evaluate:
 - +1 for win for current player
 - 0 for loss

Example – Basic Monte Carlo Search

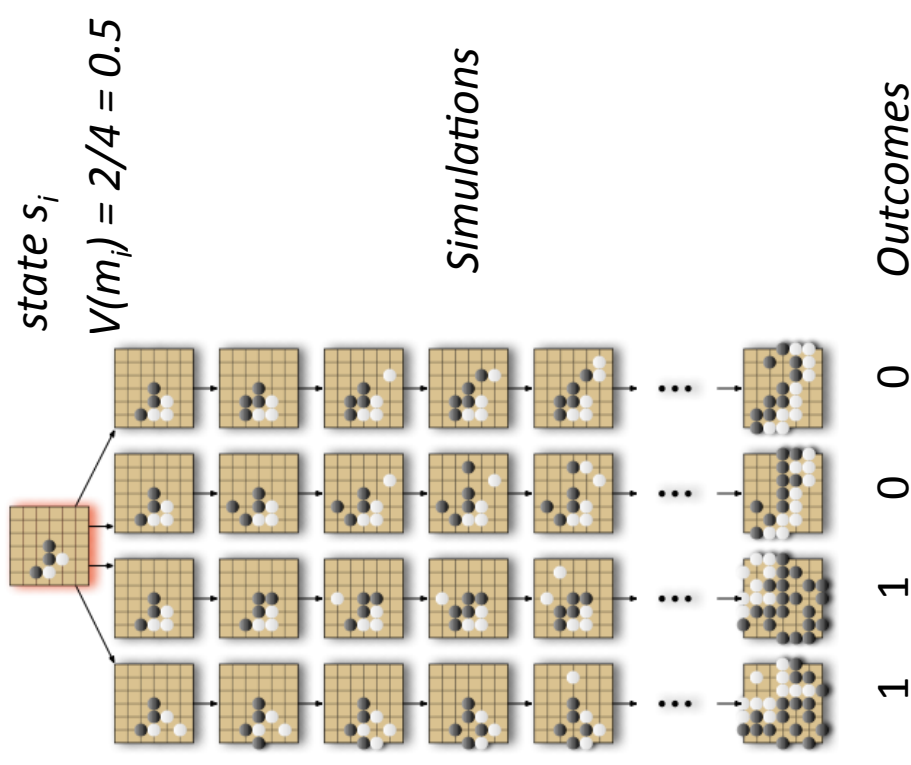


1 ply tree

root = current position

s_1 = state after move m_1

$s_2 = \dots$



Example for NoGo

- Demo for NoGo
- 1 ply search plus random simulations
- Show winning percentages for different first moves

Evaluation

- Surprisingly good e.g. in Go - much better than random or simple knowledge-based players
- Still limited
- Prefers moves that work “on average”
- Often these moves fail against the best response
- Likes “silly threats”

Improving the Monte Carlo Approach

- Add a game tree search (Monte Carlo Tree Search)
 - Major new game tree search algorithm
- Improved, better-than-random simulations
 - Mostly game-specific
- Add statistics over move quality
 - RAVE, AMAF
- Add knowledge in the game tree
 - human knowledge
 - machine-learnt knowledge

Add game tree search (Monte Carlo Tree Search)

- Naïve approach and why it fails
- Bandits and Bandit algorithms
 - Regret, exploration-exploitation, UCB algorithm
- Monte Carlo Tree Search
 - UCT algorithm

Naïve Approach

- Use simulations directly as an evaluation function for $\alpha\beta$
- Problems
 - Single simulation is very noisy, only 0/1 signal
 - running many simulations for one evaluation is very slow
 - Example:
 - typical speed of chess programs **1 million eval/second**
 - Go: 1 million moves/second, 400 moves/simulation, 100 simulations/eval = **25 eval/second**
- Result: Monte Carlo was ignored for over 10 years in Go

Monte Carlo Tree Search

- Idea: use results of simulations to guide growth of the game tree
- **Exploitation**: focus on promising moves
- **Exploration**: focus on moves where uncertainty about evaluation is high
- Two contradictory goals?
 - Theory of *bandits* can help

Bandits

- Multi-armed bandits
(slot machines in Casino)

- Assumptions:

- Choice of several *arms*

- each arm pull is independent of other pulls

- Each arm has *fixed, unknown average payoff*

- Which arm has the best average payoff?

- Want to minimize *regret* = loss from playing non-optimal arm



Example (1)

- Three arms A, B, C
- Each pull of one arm is either
 - a win (payoff 1) or
 - a loss (payoff 0)
- Probability of win for each arm is fixed but *unknown*:
 - $p(A \text{ wins}) = 60\%$
 - $p(B \text{ wins}) = 55\%$
 - $p(C \text{ wins}) = 40\%$
- A is best arm (but we don't know that)

Example (2)

- How to find out which arm is best? Which arm is best ????
- Play each arm many times
 - the empirical payoff will approach the (unknown) true payoff
- The only thing we can do
 - It is expensive to play bad arms too often
- Example:
 - Play A, win
 - Play B, loss
 - Play C, win
 - Play A, loss
 - Play B, loss
- How to choose which arm to pull in each round?

Applying the Bandit Model to Games

- Bandit arm \approx move in game
- Payoff \approx quality of move
- Regret \approx difference to best move

Explore and Exploit with Bandits

- *Explore* all arms, but also:
- *Exploit*: play promising arms more often
- Minimize *regret* from playing poor arms

Formal Setting for Bandits

- One specific setting, more general ones exist
- K arms (actions, possible moves) named $1, 2, \dots, K$
- $t \geq 1$ time steps
- X_i random variable, payoff of arm i
 - Assumed *independent of time* here
 - Later: discussion of *drift* over time, i.e. with trees
- Assume $X_i \in [0 \dots 1]$ e.g. $0 = \text{loss}$, $1 = \text{win}$
- $\mu_i = E[X_i]$ expected payoff of arm i
- r_t reward at time t
- realization of random variable X_i from playing arm i at time t

Formalization Example

- Same example as with A, B, C before, but use formal notation
- $K=3$.. 3 arms, arm 1 = A, arm 2 = B, arm 3 = C
- X_1 = random variable – pull arm 1
 - $X_1 = 1$ with probability 0.6
 - $X_1 = 0$ with probability $1 - 0.6 = 0.4$
 - similar for X_2, X_3
 - $\mu_1 = E[X_1] = 0.6, \mu_2 = E[X_2] = 0.55, \mu_3 = E[X_3] = 0.4$
- Each r_t is either 0 or 1, with probability given by the arm which was pulled.
 - Example: $r_1 = 0, r_2 = 0, r_3 = 1, r_4 = 1, r_5 = 0, r_6 = 1, \dots$

Formal Setting for Bandits (2)

- *Policy*: Strategy for choosing arm to play at time t
- given arm selections and outcomes of previous trials at times $1, \dots, t - 1$.
- $I_t \in \{1, \dots, K\}$.. arm selected at time t
- $T_i(t) = \sum_{s=1}^t \mathbb{I}(I_s = i)$
.. total number of times arm i was played from time $1, \dots, t$

Example

- Example: $l_1 = 2, l_2 = 3, l_3 = 2, l_4 = 3, l_5 = 2, l_6 = 2$
- $T_1(6) = 0, T_2(6) = 4, T_3(6) = 2$
- Simple policies:
 - Uniform - play a least-played arm, break ties randomly
 - Greedy - play an arm with highest empirical payoff
 - Question – what is a *smart* strategy?

Formal Setting for Bandits (3)

- Best possible payoff: $\mu^* = \max_{1 \leq i \leq K} \mu_i$
- Expected payoff after n steps: $\sum_{i=1}^K \mu_i \mathbb{E}[T_i(n)]$
- Regret after n steps is the difference:
$$n\mu^* - \sum_{i=1}^K \mu_i \mathbb{E}[T_i(n)]$$
- Minimize regret: minimize $T_i(n)$ for the non-optimal moves, especially the worst ones

Example, continued

- $\mu_1 = 0.6, \mu_2 = 0.55, \mu_3 = 0.4$
- $\mu^* = 0.6$
- With our fixed exploration policy from before:
 - $E[T_1(6)] = 0, E[T_2(6)] = 4, E[T_3(6)] = 2$
 - expected payoff $\mu_1^* \cdot 0 + \mu_2^* \cdot 4 + \mu_3^* \cdot 2 = 3.0$
 - expected payoff if always plays arm 1: $\mu^* \cdot 6 = 3.6$
 - Regret = $3.6 - 3.0 = 0.6$
- Important: regret of a policy is expected regret
 - Will be achieved in the limit, as average of many repetitions of this experiment
 - In any single experiment with six rounds, the payoff can be anything from 0 to 6, with varying probabilities

Formal Setting for Bandits (4)

- (Auer et al 2002)
- Statistics on each arm so far
- \bar{X}_i average reward from arm i so far
- n_i number of times arm i played so far (same meaning as $T_i(t)$ above)
- n total number of trials so far

UCB1 Formula (Auer et al 2002)

- Name UCB stands for Upper Confidence Bound
- Policy:
 1. First, try each arm once
 2. Then, at each time step:
 - choose arm i that maximizes the *UCB1 formula* for the upper confidence bound:

$$\bar{x}_i + \sqrt{\frac{2 \ln(n)}{n_i}}$$

UCB Demystified - Formula

$$\bar{X}_j + \sqrt{\frac{2 \ln(n)}{n_j}}$$

- Exploitation: higher observed reward \bar{X}_j is better
- Expect “true value” μ_j to be in some *confidence interval* around \bar{X}_j .
- “Optimism in face of uncertainty”:
choose move for which the upper bound of confidence interval is highest

UCB Demystified – Exploration Term

$$\bar{x}_i + \sqrt{\frac{2 \ln(n)}{n_i}}$$

- Interval is large when number of trials n_i is small.
Interval shrinks in proportion to $\sqrt{n_i}$
- High uncertainty about move
 - large exploration term in UCB formula
 - move is explored
- $\sqrt{\ln(n)}$ term, intuition:
explore children more if parent is important
(has many simulations)

Theoretical Properties of UCB1

- Main question: rate of convergence to optimal arm
- Huge amount of literature on different bandit algorithms and their properties
- Typical goal: regret $O(\log n)$ for n trials
- For many kinds of problems, cannot do better asymptotically (Lai and Robbins 1985)
- UCB1 is a simple algorithm that achieves this asymptotic bound for many input distributions

Is UCB What we Really Want???

- No.
- UCB minimizes *cumulative* regret
- Regret is accumulated over all trials
- In games, we only care about the final move choice
 - We do not care about simulating bad moves
- *Simple regret*: loss of our final move choice, compared to best move
 - Better measure, but theory is much less developed for trees

The case of Trees: From UCB to UCT

- UCB makes a single decision
- What about sequences of decisions (e.g. planning, games)?
- Answer: use a lookahead tree (as in games)
- Scenarios
 - Single-agent (planning, all actions controlled)
 - **Adversarial** (as in games, or worst-case analysis)
 - Probabilistic (average case, “neutral” environment)

Our

Focus



Monte Carlo Planning - UCT

- Main ideas:
- Build lookahead tree (e.g. game tree)
- Use rollouts (simulations) to generate rewards
- Apply UCB – like formula in interior nodes of tree
 - choose “optimistically” where to expand next

Generic Monte Carlo Planning Algorithm

MonteCarloPlanning(state)

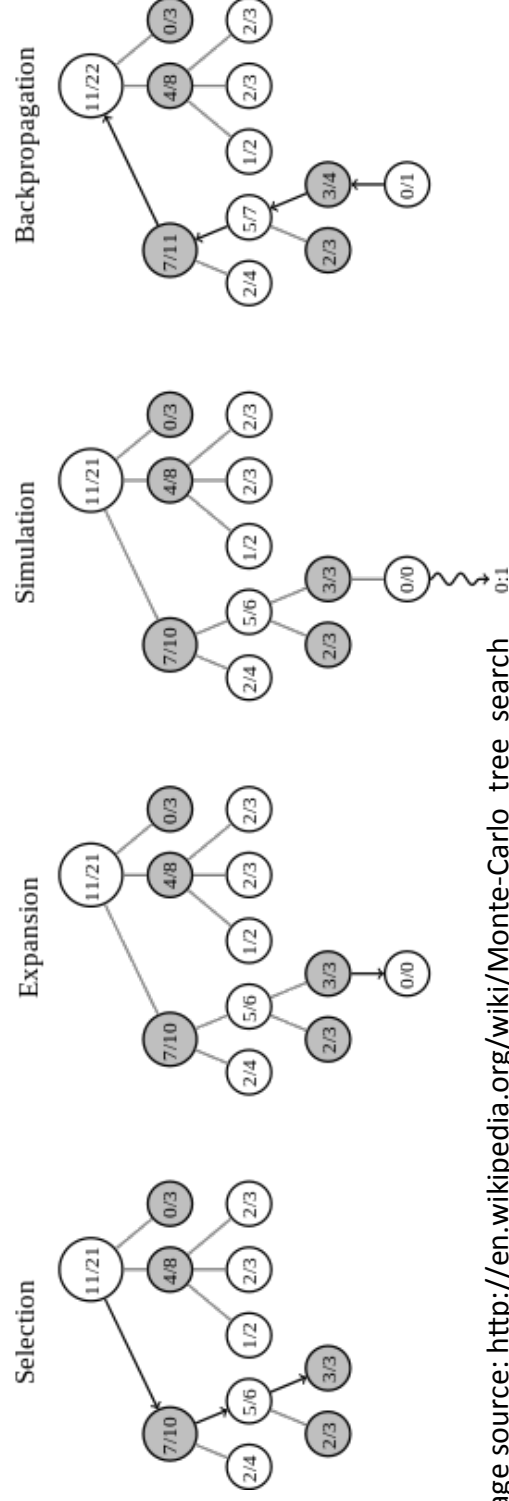
```
repeat search(state, 0) until Timeout
return bestAction(state,0)

    search(state, depth)
    if Terminal(state) then return 0
    if Leaf(state, depth) then return Evaluate(state)
    action := selectAction(state, depth)
    (nextstate, reward) := simulate (state, action)
    q := reward +  $\gamma$  search(nextstate, depth + 1)
    UpdateValue(state, action, q, depth)
return q
```

- Reinforcement-learning-like framework (Kocsis and Szepesvari 2006)
- Rewards at every time step
 - future rewards discounted by factor γ
- Apply to games:
 - 0/1 reward, only at end of game
 - $\gamma = 1$ (no discount)

Generic Monte Carlo Tree Search

- Select leaf node L in game tree
- Expand children of L
- Simulate a randomized game from (new) leaf node
- Update (or backpropagate) statistics on path to root



Drift

- In basic bandit framework, we assumed that payoff for each arm comes from a *fixed* (stationary) distribution
- If distribution changes over time, UCB will still converge under some relatively weak conditions
- In UCT, the tree changes over time
 - payoffs of choices within tree also change
 - Example: better move is discovered for one of the players

Convergence Property of UCT

- Very informal presentation here.
See (K+S 2006), Section 2.4 for precise statements.
- Assumptions:
 1. average payoffs converge for each arm i
 2. “tail inequalities”: probability of being “far off” is very small
- Under those conditions:
probability of selecting a suboptimal move approaches zero in the limit

Towards Practice: UCB1-tuned

- Finite-time Analysis of the Multiarmed Bandit Problem (Auer et al 2002)
- UCB1 formula simply assumes variance decreases with $1/\sqrt{\text{number of trials } n_i}$
- UCB1-tuned idea: take *measured variance* of each arm (move choice) into account
- Compute upper confidence bound using that measured variance
 - Can be better in practice
- We will see many more extensions to UCB ideas

MoGo – First UCT Go Program

- Original MoGo technical report (Gelly et al 2006)
- Modify UCB1-tuned, add two parameters:
 - *First-play urgency* - value for unplayed move
 - *exploration constant* c (called p in first paper) - controls rate of exploration
 $p = 1.2$ found best empirically for early MoGo

$$\bar{X}_j + p \sqrt{\frac{\log n}{T_j(n)} \min\{1/4, V_j(n_j)\}}$$

Formula from original MoGo report

Move Selection for UCT

- Scenario:
 - run UCT as long as we can
 - run simulations, grow tree
- When out of time, which move to play?
 - Highest mean
 - Highest UCB
 - **Most-simulated move**
 - later refinement: most wins

Summary – MCTS So Far

- UCB, UCT are very important algorithms in both theory and practice
- Well founded, convergence guarantees under relatively weak conditions
- Basis for extremely successful programs for games and many other applications

MCTS Enhancements

- Improved simulations
 - Mostly game-specific
 - We will discuss it later
- Improved in-tree child selection
 - General approaches
 - Review – the history heuristic
 - AMAF and RAVE
- Prior knowledge for initializing nodes in tree

Improved In-Tree Child Selection

- Plain UCT: in-tree child selection by UCB formula
 - Components: exploitation term (mean) and exploration term
- Enhancements: modify formula, add other terms
 - Collect other kinds of statistics – AMAF, RAVE
 - Prior knowledge – game specific evaluation terms
- Two main approaches
 - Add another term
 - “Equivalent experience” – translate knowledge into (virtual, fake) simulation wins or losses

Review - History Heuristic

- Game-independent enhancement for alphabeta
- Goal: improve move ordering (Schaeffer 1983, 1989)
- Give bonus for moves that lead to cutoff
Prefer those moves at other places in the search
- Similar ideas in MCTS:
 - all-moves-as-first (AMAF) heuristic, RAVE

Assumptions of History Heuristic

- Abstract concept of *move*
 - Not just a single edge in the game graph
 - identify *class of all moves* e.g. “Black F3” - place stone of given color on given square
- History heuristic: quality of such moves is correlated
 - tries to exploit that correlation
 - Special case of reasoning by similarity: in similar state, the same action may also be good
 - Classical: if move often lead to a beta cut in search, try it again, might lead to similar cutoff in similar position.
 - MCTS: if move helped to win previous simulations, then give it a bonus for its evaluation - will lead to more exploration of the move

All Moves As First (AMAF) Heuristic

- (Brügmann 1993)
- Plain Monte Carlo search:
 - no game tree, only simulations, winrate statistics for each first move
- AMAF idea: bonus for *all* moves in a winning simulation, not just the first.
 - Treat all moves like the first
 - Statistics in *global table*, separate from winrate
- Main advantage: statistics accumulate much faster
- Disadvantage: some moves good only if played right now - they will get a very bad AMAF score.

RAVE - Rapid Action Value Estimate

- Idea (Gelly and Silver 2007): compute separate AMAF statistics in *each node* of the MCTS tree
- After each simulation, update the RAVE scores of all ancestors that are in the tree
- Each move i in the tree now also has a RAVE score:
 - number of simulations $n_{i,RAVE}$
 - number of wins $v_{i,RAVE}$
 - RAVE value $x_{i,RAVE} = v_{i,RAVE}/n_{i,RAVE}$

RAVE Illustration

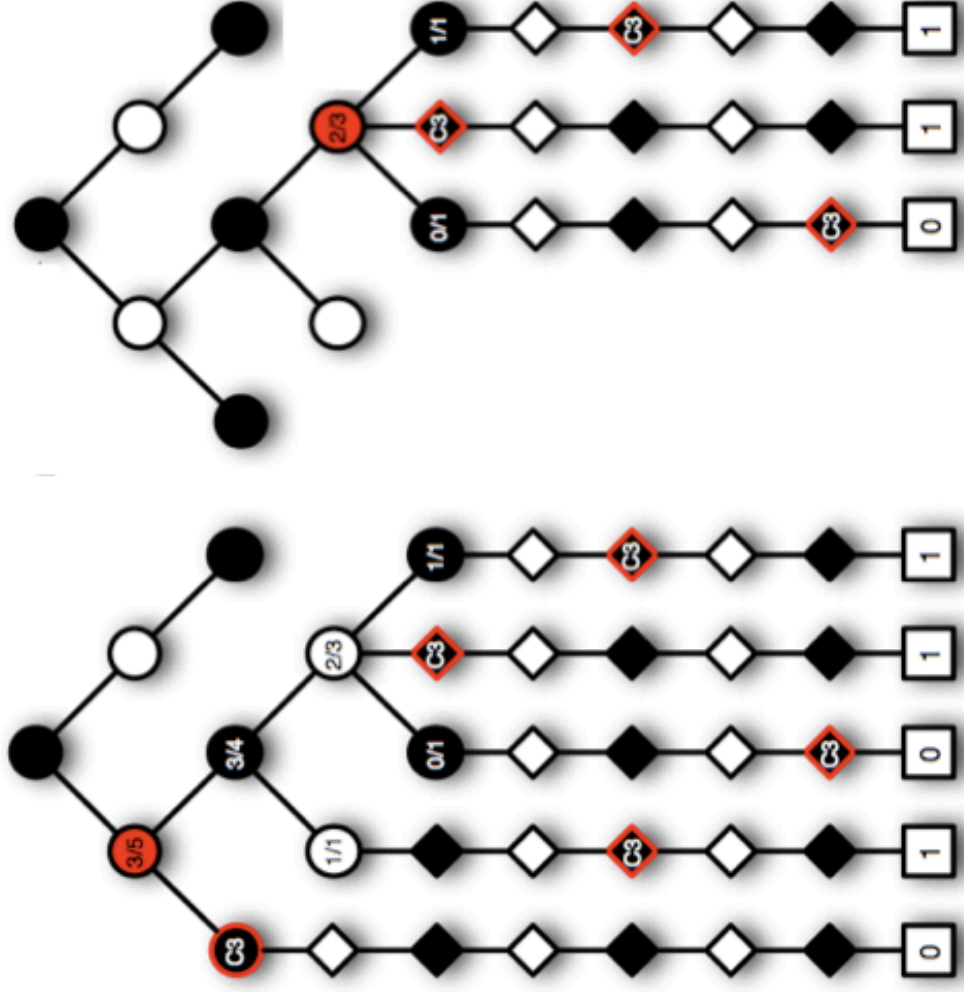


Image source: (Silver 2009)

Adding RAVE to the UCB Formula

- Basic idea: replace mean value x_j with weighted combination of mean value and RAVE value

$$\beta x_j + (1 - \beta) x_{j,RAVE}$$

- How to choose β ?
Not constant, depends on all statistics
- Try to find best combined estimator given x_j and $x_{j,RAVE}$

Adding RAVE (2)

- Original method in MoGo (Gelly and Silver 2007):
 - *equivalence parameter* k = number of simulations when mean and RAVE have equal weight
 - When $n_j = k$, then $\beta = 0.5$
 - Results were quite stable for wide range of $k=50\dots 10000$

➤ Formula

$$\beta(s, a) = \sqrt{\frac{k}{3n(s) + k}}$$

Adding RAVE (3)

- (Silver 2009, Chapter 8.4.3)
- Assume independence of estimates
 - not true in real life, but useful assumption
- Can compute optimal choice in closed form (!)
- Estimated by machine learning, or trial and error

Adding RAVE (4) – Fuego Program

- General scheme to combine different estimators
 - Combining mean and RAVE is special case
 - Very similar to Silver's scheme
- General scheme: each estimator has:
 1. *initial slope*
 2. *final asymptotic value*
 - Details: <http://fuego.sourceforge.net/fuego-doc-1.1/smartgame-doc/sguctsearchweights.html>

Using Prior Knowledge

- (Gelly and Silver 2007)
- Most nodes in the game tree are leaf nodes (exponential growth)
- Almost no statistics for leaf nodes - only simulated once
- Use domain-specific knowledge to initialize nodes
 - “equivalent experience” - a number of wins and losses
 - additive term (Rosin 2011)
- Similar to heuristic initialization in proof-number search

Types of Prior Knowledge

- (Silver 2009) machine-learned 3x3 pattern values
- Later Mogo and Fuego: hand-crafted features
- Crazy Stone: many features, weights trained by Minorization-Maximization (MM) algorithm (Coulom 2007)
- Fuego today:
 - large number of simple features
 - weights and interaction weights trained by *Latent Feature Ranking* (Wistuba et al 2013)

Example – Pattern Features (Coulom)

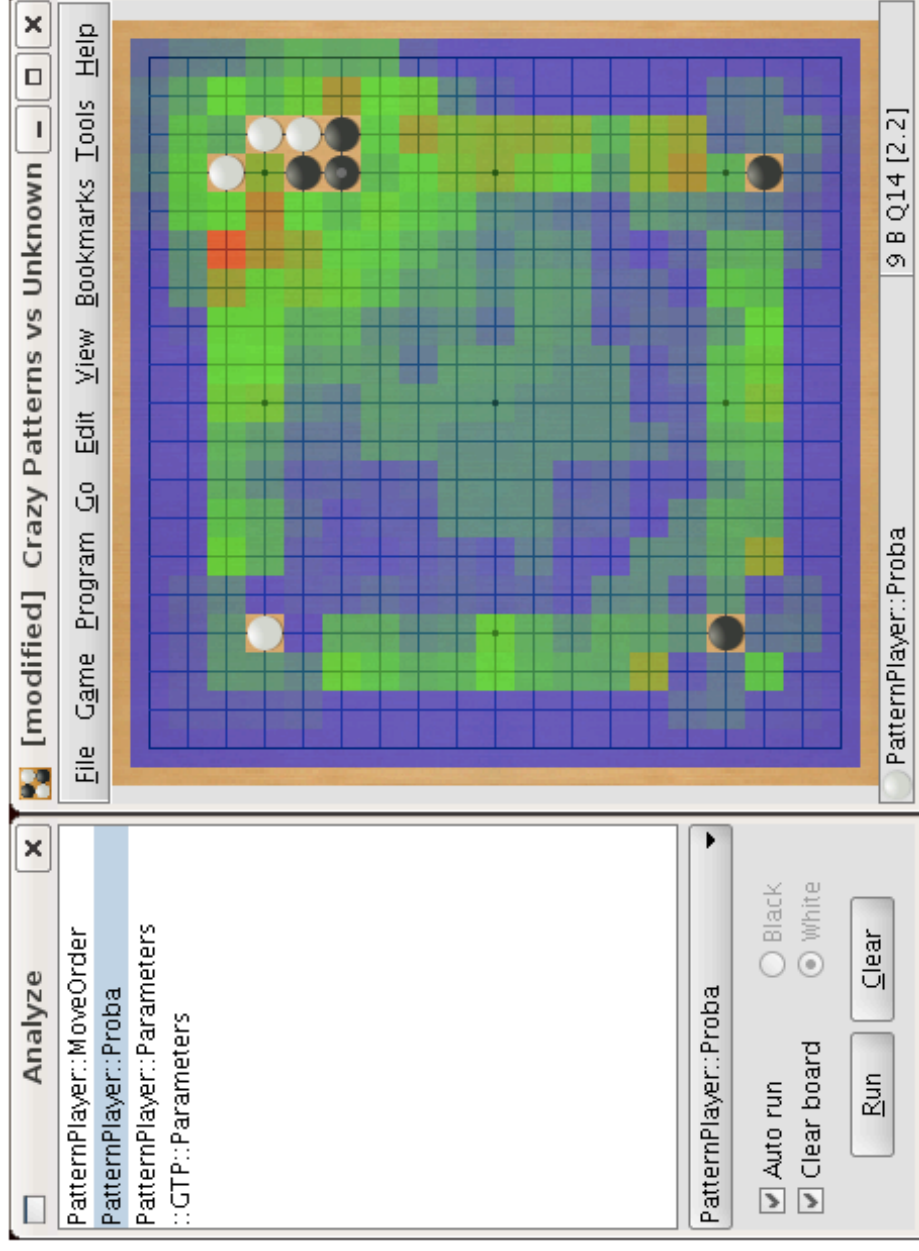


Image source: Remi Coulom

Improving Simulations

- Goal: strong correlation between initial position and result of simulation
- Preserve wins and losses
- How?
 - Avoid blunders
 - “Stabilize” position
 - Go: prefer local replies
 - Go: urgent pattern replies

Improving Simulations (2)

- Game-independent techniques
 - If there is an immediate win, then take it (1 ply win check)
 - Avoid immediate losses in simulation (1 ply mate check)
 - Avoid moves that give opponent an immediate win (2 ply mate check)
 - Last Good Reply – next slide

Last Good Reply

- Last Good Reply (Drake 2009),
Last Good Reply with Forgetting (Baier et al 2010)
- Idea: after winning simulation, store (opponent move, our answer) move pairs
 - Try same reply in future simulations
 - Forgetting: delete move pair if it fails
- Evaluation: worked well for Go program with simpler
playout policy (Orego)
 - Trouble reproducing success with stronger Go programs
- Simple form of adaptive simulations

Hybrid Approaches

- Combine MCTS with “older” ideas from the alphabeta world
- Examples
 - Prove wins/losses
 - Use evaluation function
 - Hybrid search strategy MCTS+alphabeta

Hybrids: MCTS + Game Solver

- Recognize leaf nodes that are wins/losses
- Backup in minimax/proof tree fashion
- Problem: how to adapt child selection if some children are proven wins or losses?
 - At least, don't expand those anymore
- Useful in many games, e.g. Hex, Lines of Action, NoGo, Havannah, Konane,...

Hybrids: MCTS + Evaluation

- Use evaluation function
 - Standard MCTS plays until end of game
 - Some games have reasonable and fast evaluation functions, but can still profit from exploration
 - Examples: Amazons, Lines of Action
- Hybrid approach (Lorentz 2008, Winands et al 2010)
 - run short simulation for fixed number of moves (e.g. 5-6 in Amazons)
 - call static evaluation at end, use as simulation result

Hybrids: MCTS + Minimax

- 1-2 ply lookahead in plyouts (discussed before)
 - Require strong evaluation function
- (Baier and Winands 2013) add minimax with no evaluation function to MCTS
 - Plyouts
 - Avoid forced losses
 - Selection/Expansion
 - Find shallow wins/losses

Towards a Tournament-Level Program

- Early search termination – best move cannot change
- Pondering – think in opponent's time
- Time control – how much time to spend for each move
- Reuse sub-tree from previous search
- Multithreading (see later)
- Code optimization
- Testing, testing, testing,...

Machine Learning for MCTS

- Learn better knowledge
 - Patterns, features (discussed before)
- Learn better simulation policies
 - Simulation balancing (Silver and Tesauro 2009)
 - Simulation balancing in practice (Huang et al 2011)
- Adapt simulations online
 - Dyna2, RLGo (Silver et al 2012)
 - Nested Rollout Policy Adaptation (Rosin 2011)
 - Last Good Reply (discussed before)
 - Use RAVE (Rimmel et al 2011)

Parallel MCTS

- MCTS scales well with more computation
- Currently, hardware is moving quickly towards more parallelism
- MCTS simulations are “embarrassingly parallel”
- Growing the tree is a sequential algorithm
 - How to parallelize it?

Parallel MCTS - Approaches

- root parallelism
- shared memory
- distributed memory
- New algorithm: depth-first UCT (Yoshizoe et al 2011)
 - Avoid bottleneck of updates to the root

Root Parallelism

- (Cazenave and Jouandeau 2007, Soejima et al. 2010)
- Run n independent MCTS searches on n nodes
- Add up the top-level statistics
- Easiest to implement, but limited
- Majority vote may be better

Shared Memory Parallelism

- n cores together build one tree in shared memory
- How to synchronize access? Need to write results (changes to statistics for mean and RAVE), add nodes, and read statistics for in-tree move selection
- Simplest approach: lock tree during each change
- Better: lock-free hash table (Coulom2008) or tree (Enzenberger and Müller 2010)
- Possible to use spinlock

Limits to Parallelism

- Loss of information from running n simulations in parallel as opposed to sequentially
- Experiment (Segal 2010)
 - run single-threaded
 - delay tree updates by $n - 1$ simulations
- Best-case experiment for behavior of parallel MCTS
- Predicts upper limit of strength over 4000 Elo above single-threaded performance

Virtual Loss

- Record simulation as a loss at start
 - Leads to more variety in UCT-like child selection
- Change to a win if outcome is a win
- Crucial technique for scaling
- With virtual loss, scales well up to 64 threads
- Can also use *virtual wins*

Fuego Virtual Loss Experiment

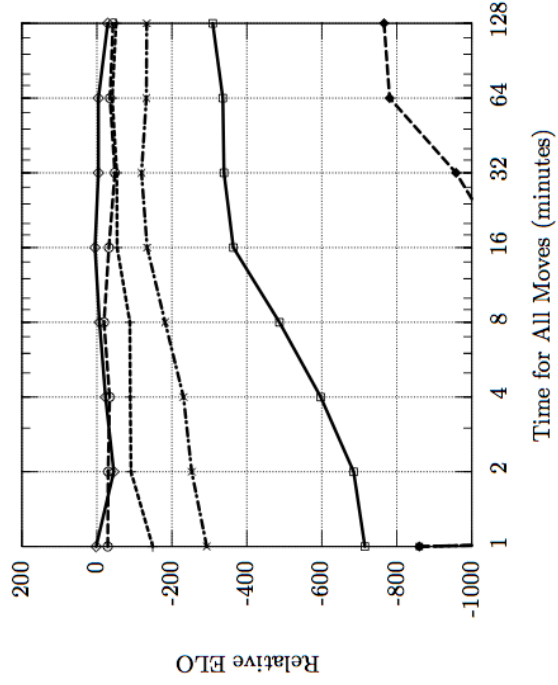


Fig. 2. Self-play of N threads against a uni-processor with equal total computation.

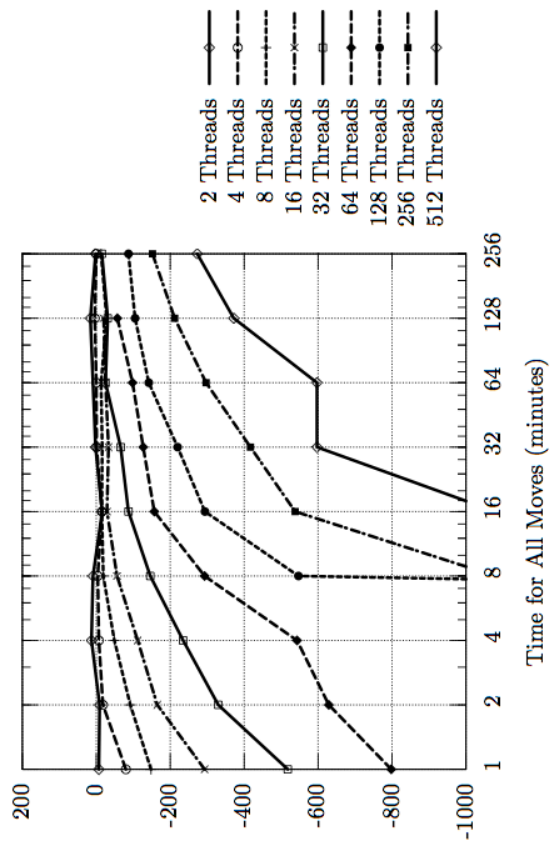


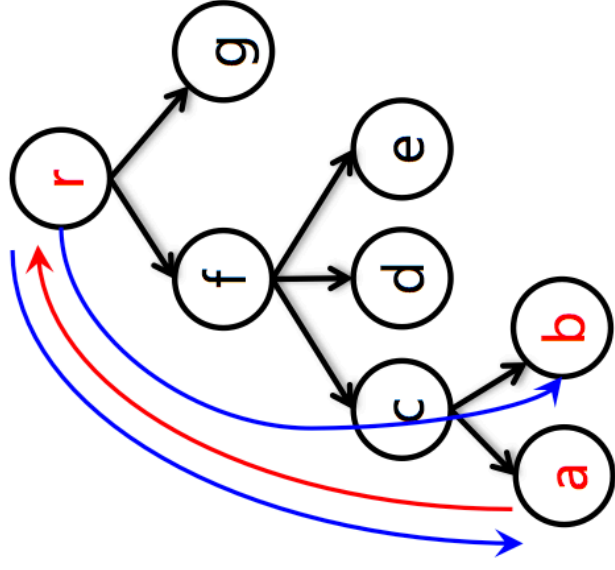
Fig. 4. Self-play of N threads against a uni-processor and virtual loss enabled.

Image source: (Segal 2010)

Distributed Memory Parallelism

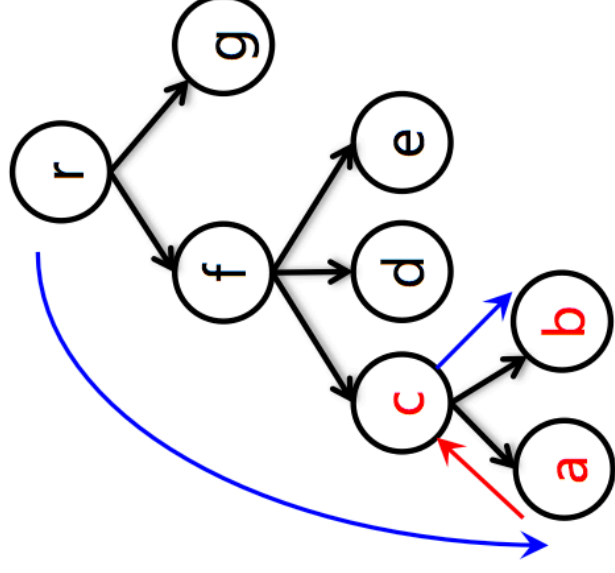
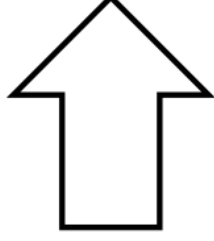
- Many copies of MCTS engine, one on each compute node
- Communicate by message passing (MPI)
- MoGo model:
 - synchronize a few times per second
 - synchronize only “heavy” nodes which have many simulations
- Performance depends on
 - hardware for communication
 - shape of tree
 - game-specific properties, length of playouts

Normal UCT vs. Depth-first UCT



Normal UCT

always return to root



Depth First UCT

returns only if needed

Depth-first UCT

- Bottleneck of updates to “heavy” nodes including root
- Depth-first reformulation of UCT
 - stay in subtree while best-child selection is unlikely to change
 - about 1 - 2% wrong child selections
 - Delay updates further up the tree
 - Similar idea as df-pn
 - Unlike df-pn, sometimes the 3rd-best (or worse) child can become best

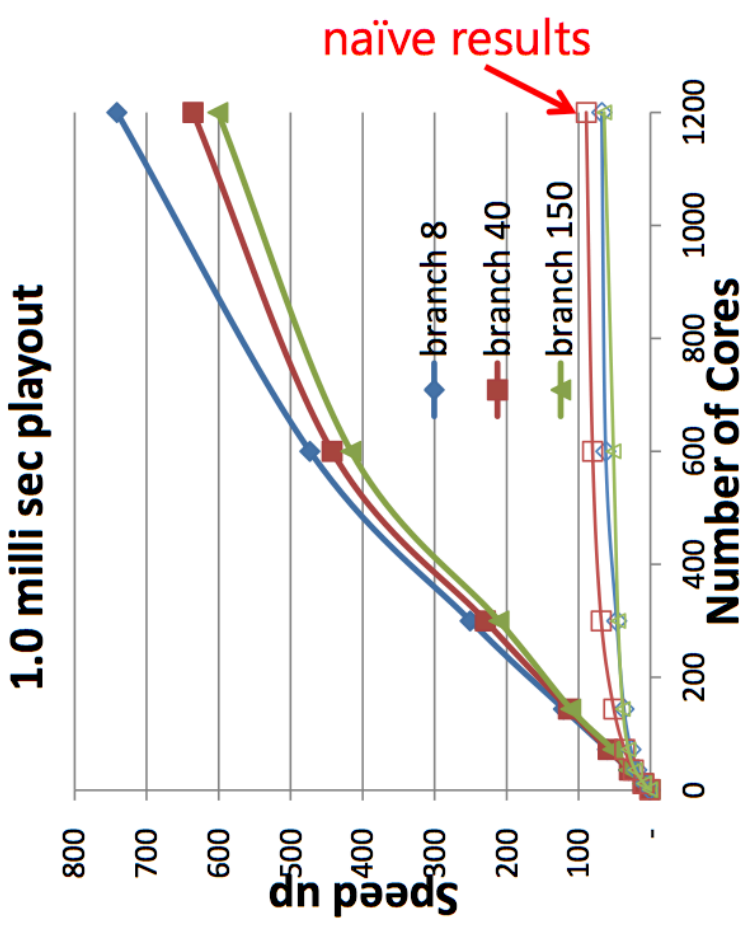
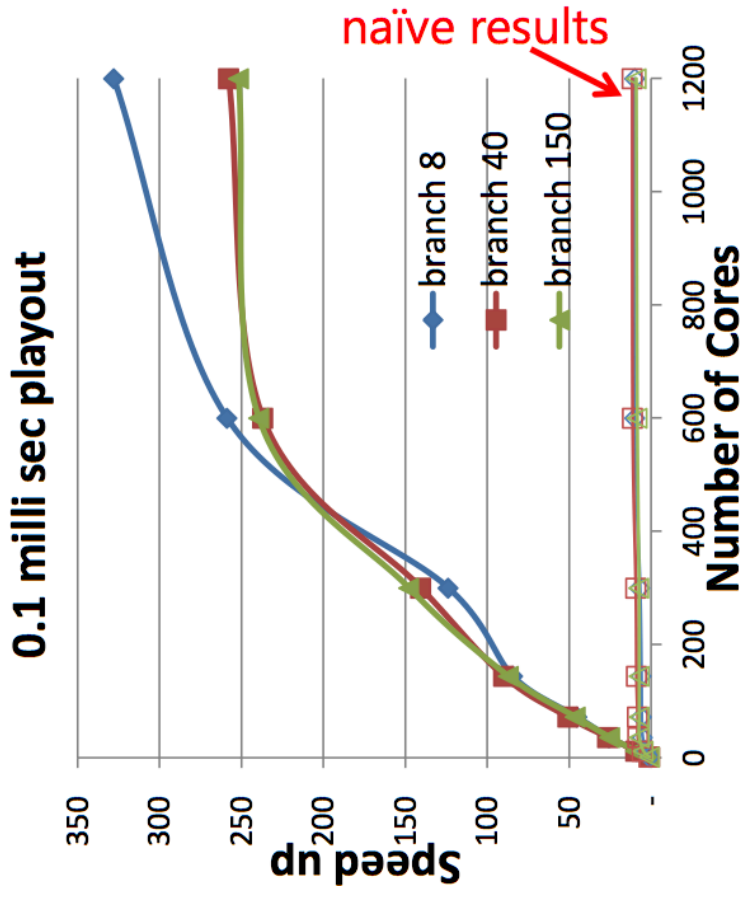
Distributed Memory: TDS

- TDS – Transposition Table Driven Scheduling (Romein et al 1999)
- Single global hash table
 - Each node in tree owned by one processor
 - Work is sent to the processor that owns the node
 - In single-agent search, achieved almost perfect speedup on mid-size parallel machines

TDS-df-UCT

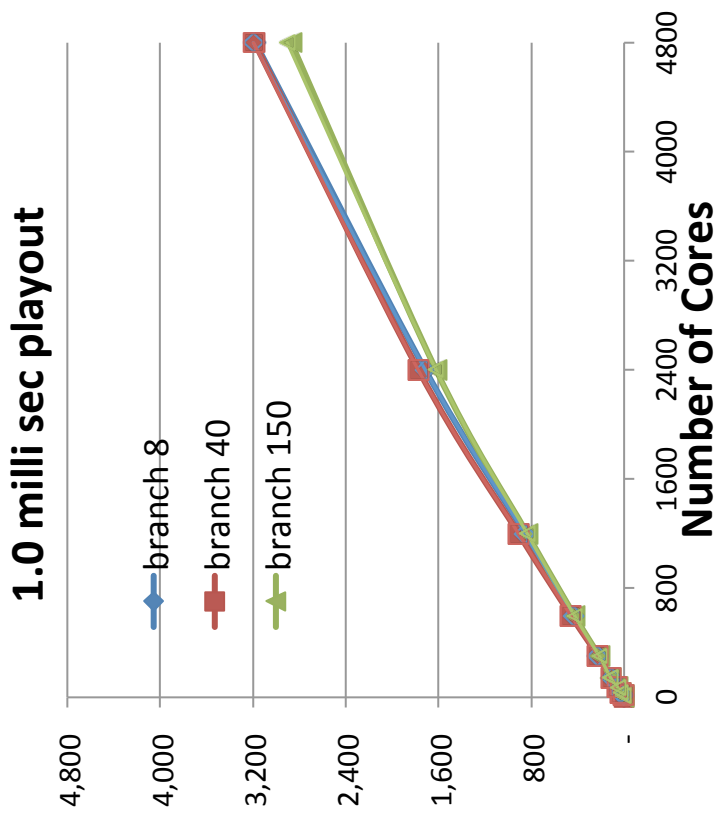
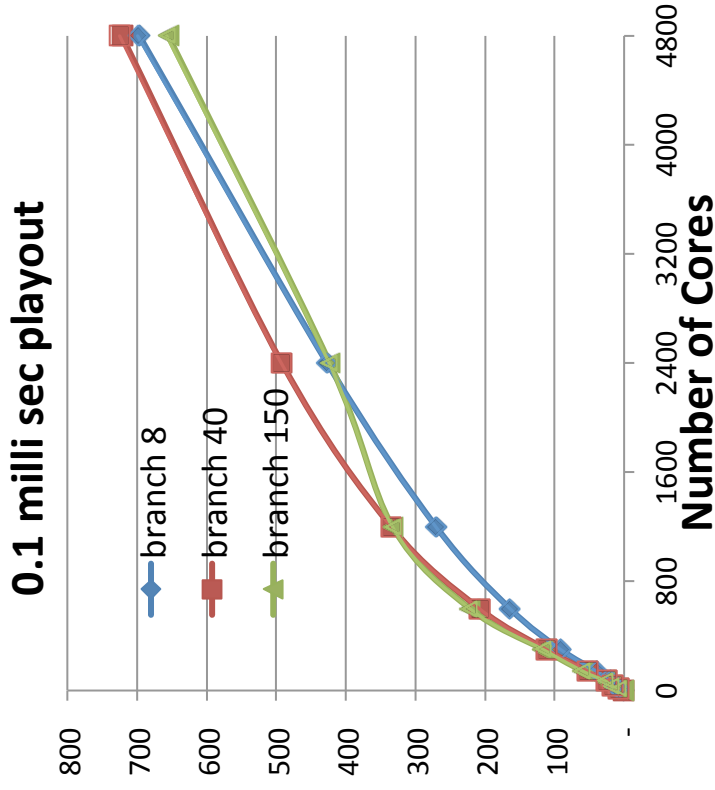
- Use TDS approach to implement df-UCT on (massively) parallel machines
 - TSUBAME2 (17984 cores)
 - SGI UV-1000 (2048 cores)
- Implemented artificial game (P-game) and Go (MP-Fuego program)
 - In P-game: measure effect of playout speed (artificial slowdown for fake simulations)

TDS-df-UCT Speedup - 1200 Cores



330 fold speedup for 0.1 ms playoff
740 fold speedup for 1.0 ms playoff

P-game 4,800 Cores



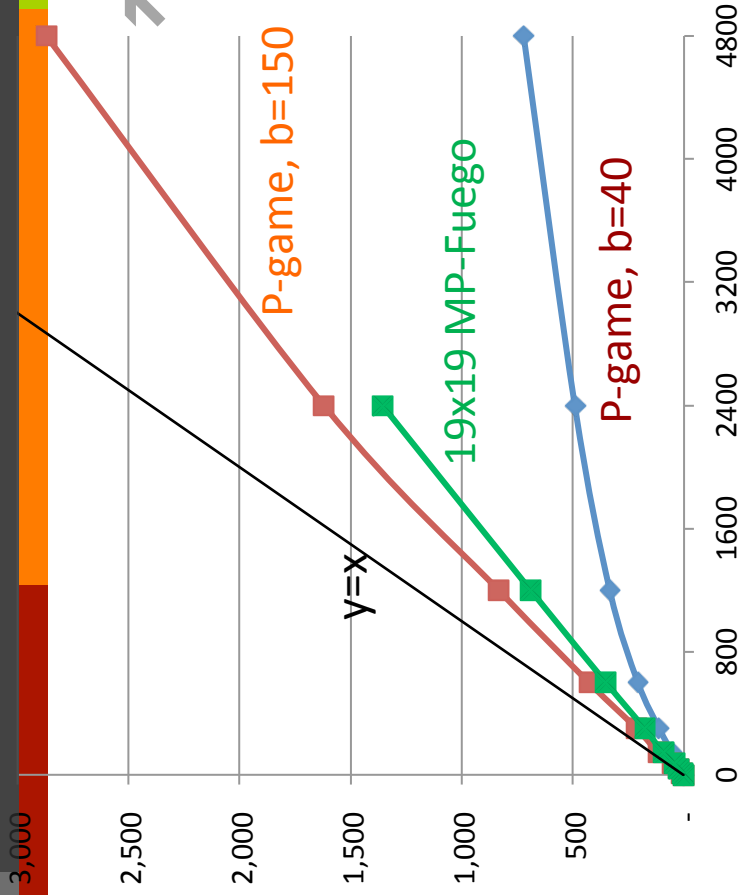
job number
= cores x 10

700-fold for 0.1 ms playoff
3,200-fold for 1.0 ms playoff

TDS-df-UCT = TDS + depth first UCT

Speedup including Go

Speedup



MP-Fuego

- ↗ 2 playouts at leaf
- ↗ (approx. 0.8 ms playout)
- ↗ 5 jobs/core



Hardware1: TSUBAME2 supercomputer

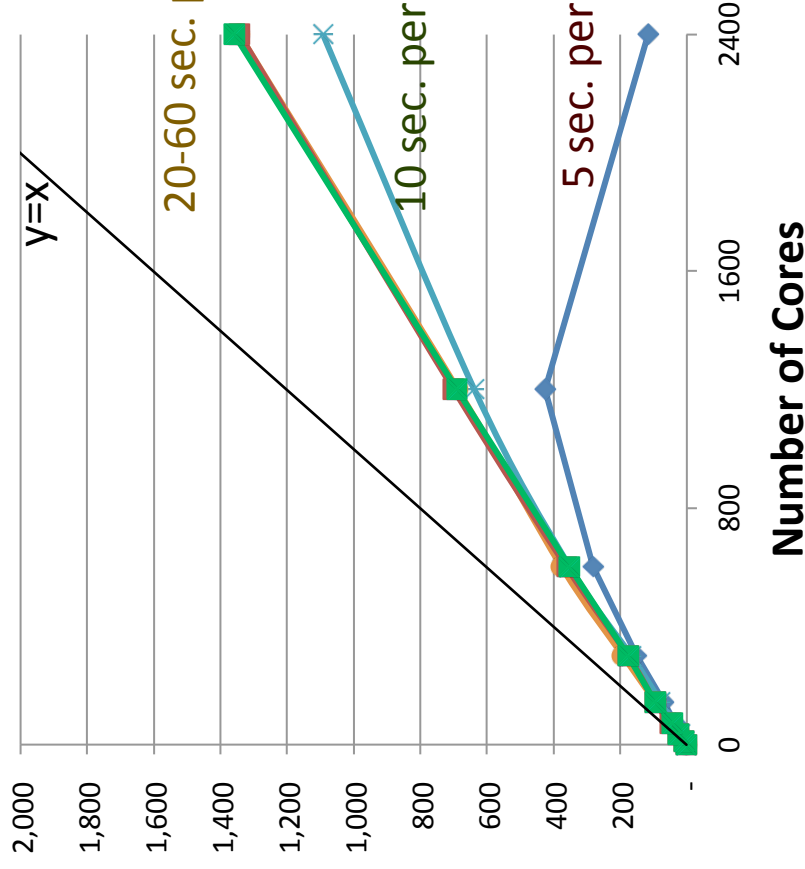


Hardware2: SGI UV1000 (Hungabee)

Image source: K. Yoshizoe

Search Time and Speedup

MP-Fuego speedup (19x19)



↗ Short thinking time =
slower speedup

↗ One major difficulty in
massive parallel search

Summary – MCTS Tutorial so far...

- Reviewed algorithms, enhancements, applications
 - Bandits
 - Simulations
 - Monte Carlo Tree Search
 - AMAF, RAVE, adding knowledge
 - Hybrid algorithms
 - Parallel algorithms
- Still to come: impact of MCTS, research topics

Impact - Applications of MCTS

- Classical Board Games
 - Go, Hex
 - Amazons
 - Lines of Action, Arimaa, Havannah, NoGo, Konane,...
- Multi-player games, card games, RTS, video games
- Probabilistic Planning, MDP, POMDP
- Optimization, energy management, scheduling, distributed constraint satisfaction, library performance tuning, ...

Impact – Strengths of MCTS

- Very general algorithm for decision making
- Works with very little domain-specific knowledge
 - Need a simulator of the domain
- Can take advantage of knowledge when present
- Successful parallelizations for both shared memory and massively parallel distributed systems

Current Topics in MCTS

- Recent progress, Limitations, random half-baked ideas, challenges for future work,...
- Dynamically adaptive simulations
- Integrating local search and analysis
- Improve in-tree child selection
- Parallel search
 - Extra simulations should never hurt
 - Sequential halving and SHOT

Dynamically Adaptive Simulations

- Idea: adapt simulations to specific current context
- Very appealing idea, only modest results so far
- Biasing using RAVE (Rimmel et al 2010) – small improvement
- Last Good Reply (with Forgetting) (Drake 2009, Baier et al 2010)

Integrating Local Search and Analysis

- Mainly For Go
 - Players do much local analysis
 - Much of the work on simulation policies and knowledge is about local replies
- Combinatorial Game Theory has many theoretical concepts
- Tactical alphabeta search (Fuego, unpublished)
- Life and death solvers

Improve In-tree Child Selection

- Intuition: want to maximize if we're certain, average if uncertain
- Is there a better formula than average weighted by number of simulations? (My intuition: there has to be...)
- Part of the benefits of iterative widening may be that the max is over fewer sibling nodes – measure that
 - Restrict averaging to top n nodes

Extra Simulations Should Never Hurt

- Ideally, adding more search should never make an algorithm weaker
- For example, if you search nodes that could be pruned in alphabeta, it just becomes slower, but produces the same result
- Unfortunately it is not true for MCTS
- Because of averaging, adding more simulations to bad moves hurts performance - it is worse than doing nothing!

Extra Simulations Should Never Hurt (2)

- Challenge: design a MCTS algorithm that is robust against extra search at the “wrong” nodes
- This would be great for parallel search
- A rough idea: keep two counters in each node - total simulations, and “useful” simulations
- Use only the “useful” simulations for child selections
- Could also “disable” old, obsolete simulations?

Sequential Halving, SHOT

- Early MC algorithm: successive elimination of empirically worst move (Bouzy 2005)
- Sequential halving (Karnin et al 2013):
 - Rounds of uniform sampling
 - keep top half of all moves for next round
- SHOT (Cazenave 2014)
 - Sequential halving applied to trees
 - Like UCT, uses bandit algorithm to control tree growth
 - Promising results for NoGo
 - Promising for parallel search