Monte Carlo Tree Search and AlphaGo

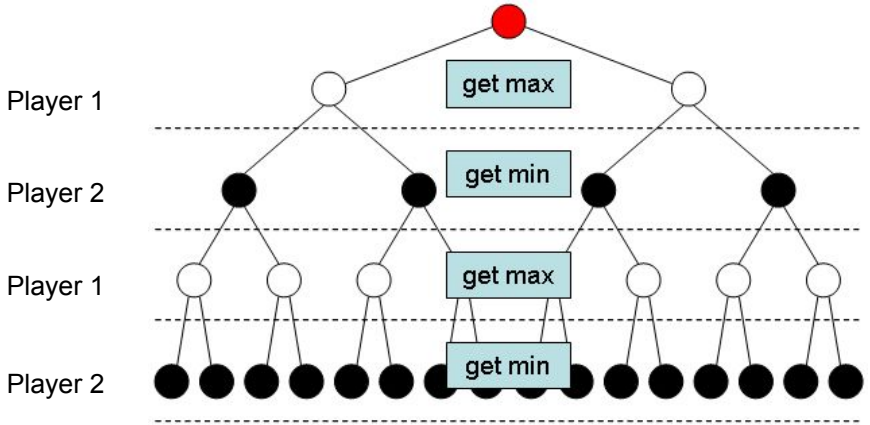
Suraj Nair, Peter Kundzicz, Kevin An, Vansh Kumar

Zero-Sum Games and AI

- A player's utility gain or loss is exactly balanced by the combined gain or loss of opponents:
  - E.g. - Given a pizza with 8 slices to share between person A and B.
    - A eats 1 slice.
      - A experiences +1 net utility.
      - B experiences -1 net utility.
- This is a powerful concept important to AI development for measuring the cost/benefit of a particular move.
- Nash Equilibrium.

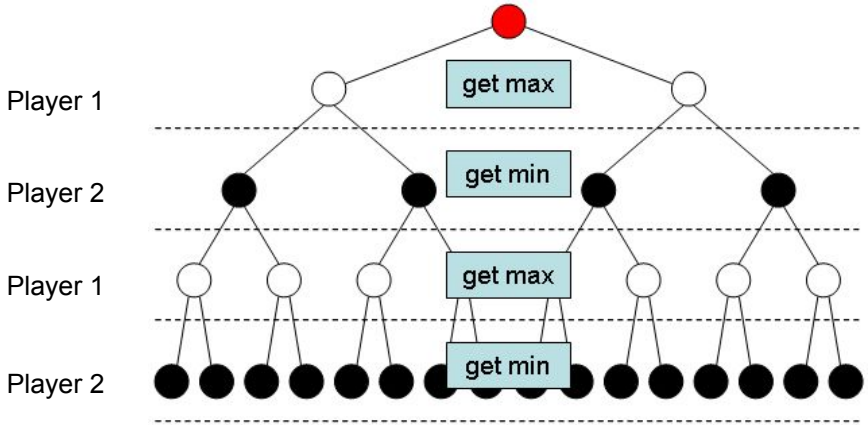
Games and AI

- Traditional strategy - **Minimax**:
  - Attempt to minimize opponent's maximum reward at each state (Nash Equilibrium)
  - Exhaustive Search



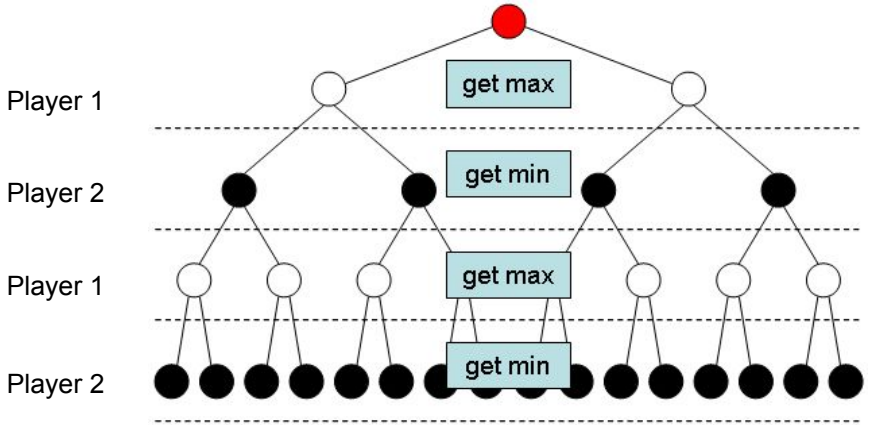
Drawbacks

- The number of moves to be analyzed quickly increases in depth.
- The computation power limits how deep the algorithm can go.



Alternative Idea

- Bandit-Based Methods
  - Choosing between K actions/moves.
  - Need to maximize the cumulative reward by continuously picking the best move.
- Given a game state we can treat each possible move as an action.
- Some problems / Further improvements:
  - Once we pick a move the state of the game changes.
  - The true reward of each move depends on subsequently possible moves.



Monte Carlo Tree Search

- Application of the Bandit-Based Method.
- Two Fundamental Concepts:
  - The true value of any action can be approximated by running several random simulations.
  - These values can be efficiently used to adjust the policy (strategy) towards a best-first strategy.
- Builds a partial game tree before each move. Then selection is made.
  - Moves are explored and values are updated/estimated.

General Applications of Monte Carlo Methods

- Numerical Algorithms
- AI Games
  - Particularly games with imperfect information
  - Scrabble/Bridge
  - Also very successful in Go (We will hear more about this later)
- Many other applications
  - Real World Planning
  - Optimization
  - Control Systems

Understanding Monte Carlo Tree Search

MCTS Overview

- Iteratively building partial search tree
- Iteration
  - Most urgent node
    - Tree policy
    - Exploration/exploitation
  - Simulation
    - Add child node
    - Default policy
  - Update weights

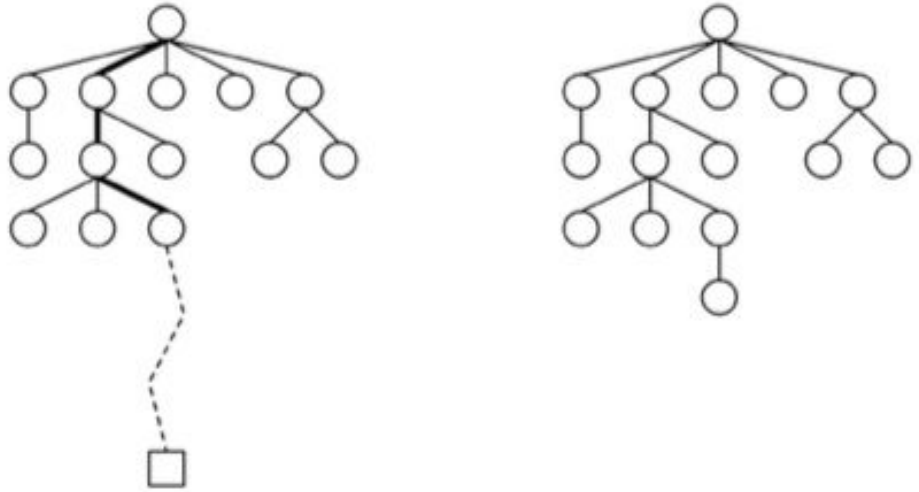


Fig. 1. The basic MCTS process [17].

Development of MCTS

- Kocsis and Szepesvári, 2006
  - Formally describing bandit-based method
  - Simulate to approximate reward
- Proved MCTS converges to minimax solution
- UCB1: finds optimal arm of upper confidence bound (UCT employed UCB1 algorithm on each explored node)

Algorithm Overview

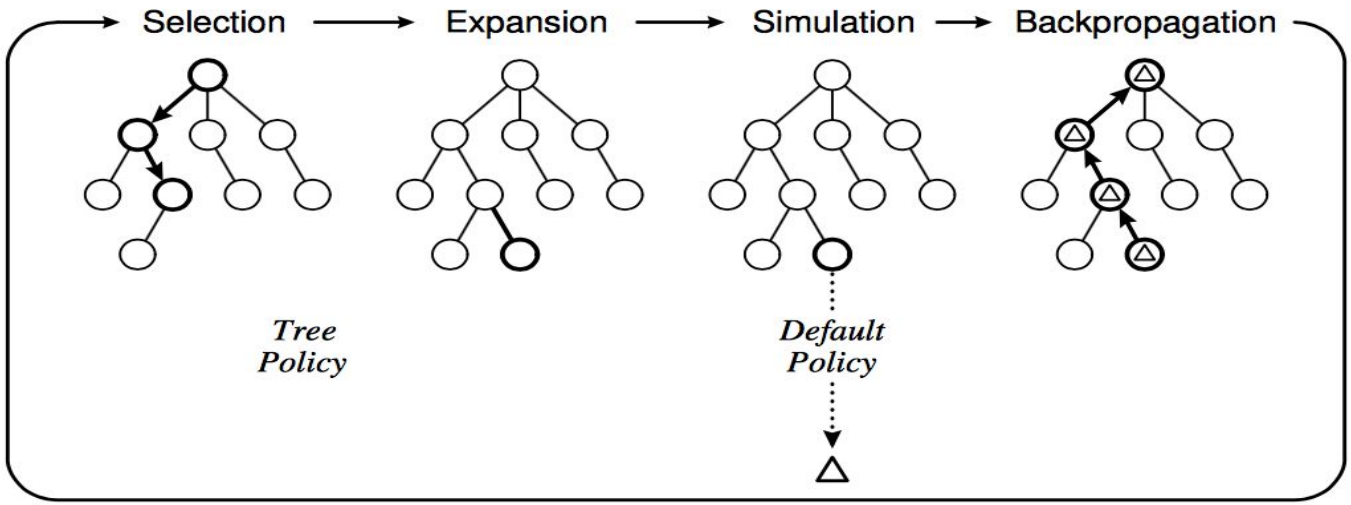


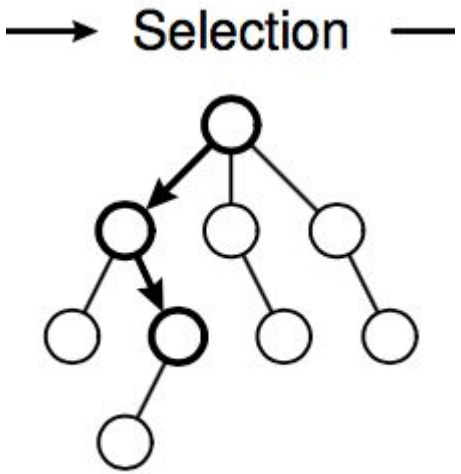
Fig. 2. One iteration of the general MCTS approach.

Policies

- Policies are crucial for how MCTS operates
- Tree policy
  - Used to determine how children are selected
- Default policy
  - Used to determine how simulations are run (ex. randomized)
  - Result of simulation used to update values

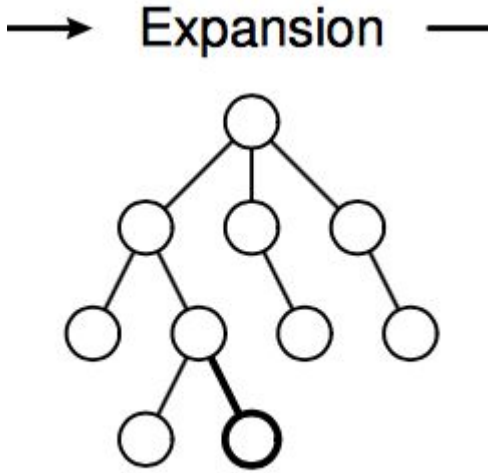
Selection

- Start at root node
- Based on Tree Policy select child
- Apply recursively - descend through tree
  - Stop when expandable node is reached
  - *Expandable* -
    - Node that is non-terminal and has unexplored children



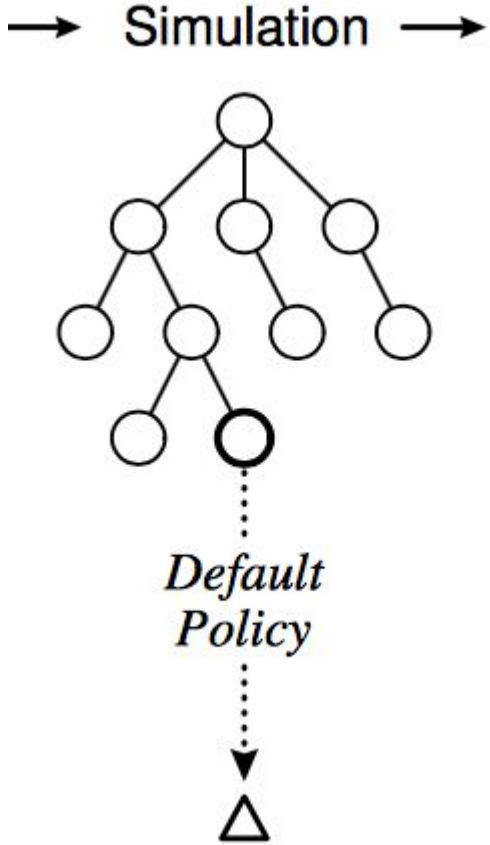
Expansion

- Add one or more child nodes to tree
  - Depends on what actions are available for the current position
  - Method in which this is done depends on Tree Policy



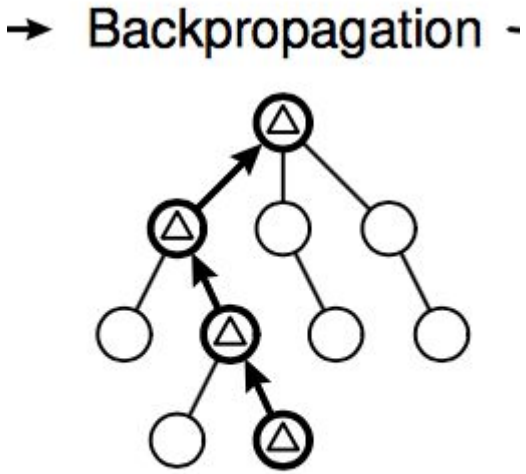
Simulation

- Runs simulation of path that was selected
- Get position at end of simulation
- Default Policy determines how simulation is run
- Board outcome determines value



Backpropagation

- Moves backward through saved path
- Value of Node
  - representative of benefit of going down that path from parent
- Values are updated dependent on board outcome
  - Based on how the simulated game ends, values are updated



Policies

- Tree policy
  - Select/create leaf node
  - Selection and Expansion
  - Bandit problem!
- Default policy
  - Play the game till end
  - Simulation
- Selecting the best child
  - Max (highest weight)
  - Robust (most visits)
  - Max-robust (both, iterate if none exists)

UCT Algorithm

- Selecting Child Node - Multi-Arm Bandit Problem
- UCB1 for each child selection
- UCT - $$UCT = \overline{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

- n - number of times current(parent) node has been visited
- n_j - number of times child j has been visited
- C_p - some constant > 0
- X_j - mean reward of selecting this position
  - [0, 1]

UCT Algorithm

$$UCT = \overline{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

- n_j = 0 means infinite weight
  - Guarantees we explore each child node at least once
- Each child has non-zero probability of selection
- Adjust C_p to change exploration vs exploitation tradeoff

Advantages/disadvantages of MCTS

- Aheuristic
  - No need for domain-specific knowledge
  - Other algos may work better if heuristics exists
    - Minimax for Chess
    - Better because chess has strong heuristics that can decrease size of tree.
- Anytime
  - Can stop running MCTS at any time
  - Return best action
- Asymmetric
  - Favor more promising nodes
- Ramanujan et al.
  - Trap states = UCT performs worse
  - Can't model sacrifices well (Queen Sacrifice in Chess)

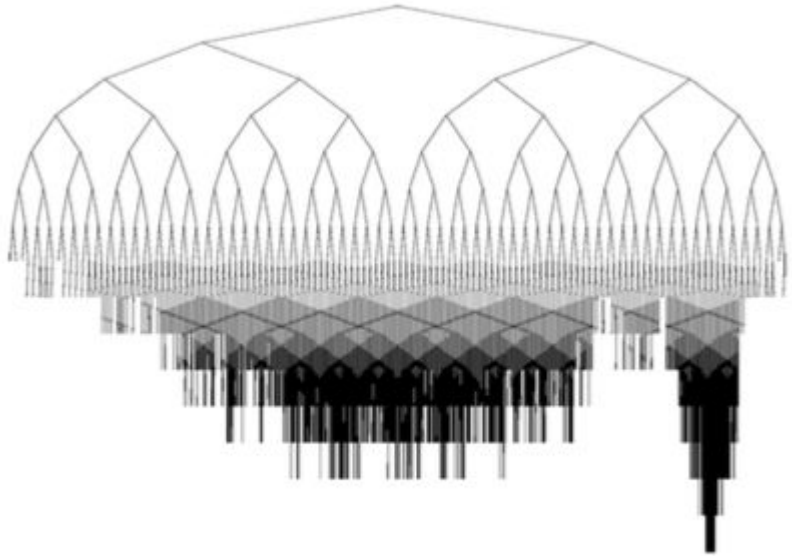
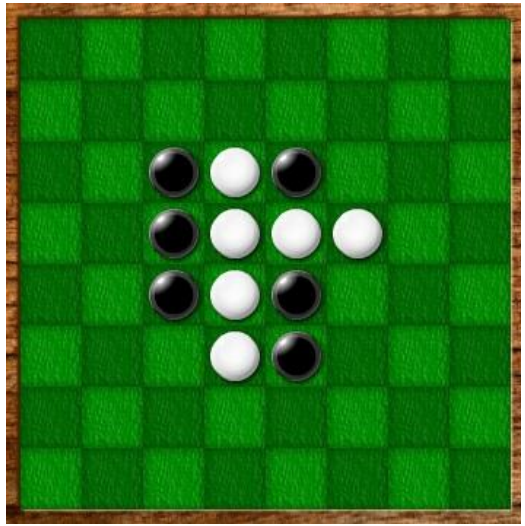


Fig. 3. Asymmetric tree growth [68].

Example - Othello

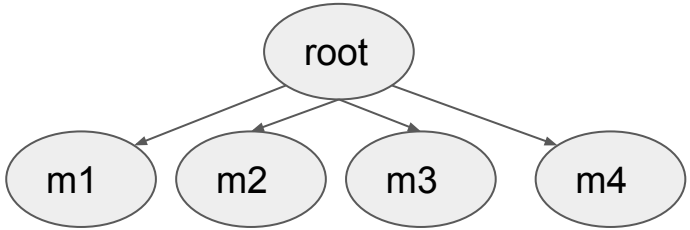
Rules of Othello

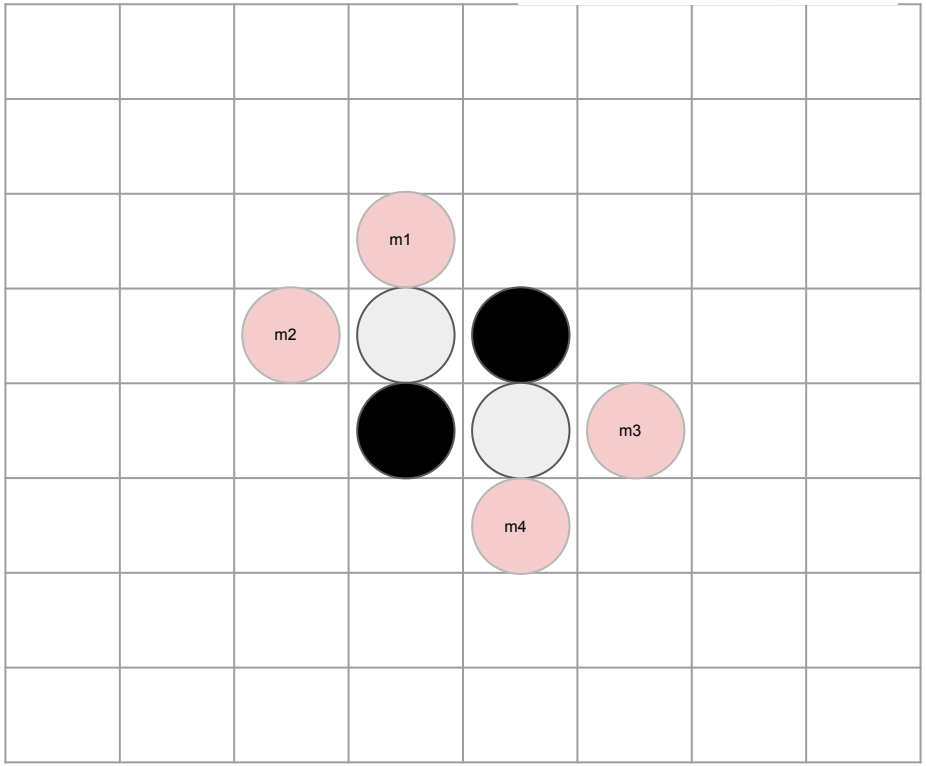
- Alternating turns
- You can only make a move that sandwiches a continuous line of your opponent's pieces between yours
    - Color of sandwiched pieces switches to your color
- Ends when board is full
- Winner is whoever has more pieces



Example - The Game of Othello

$$UCT = \overline{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$





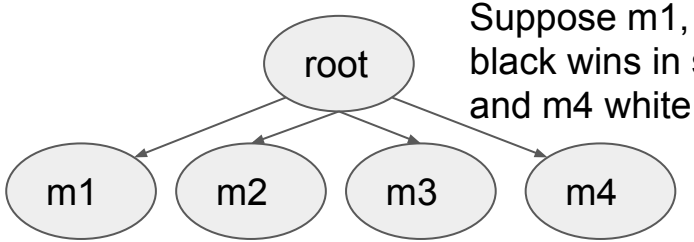
- n_j - initially 0
  - all weights are initially infinity
- n - initially 0
- C_p - some constant > 0
  - For this example
  - C = (1 / 2√2)
- X_j - mean reward of selecting this position
  - [0, 1]
  - Initially N/A

Example - The Game of Othello cont.

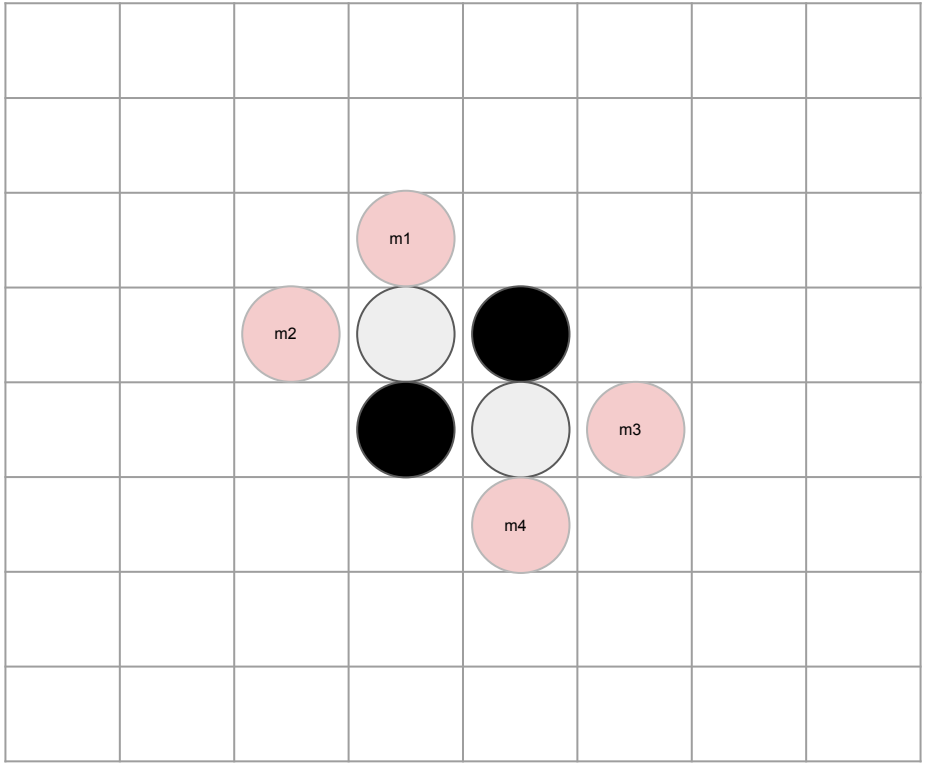
$$UCT = \overline{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

(X_j, n, n_j) - (Mean Value, Parent Visits, Child Visits)

After first 4 iterations:
Suppose m1, m2, m3 black wins in simulation and m4 white wins



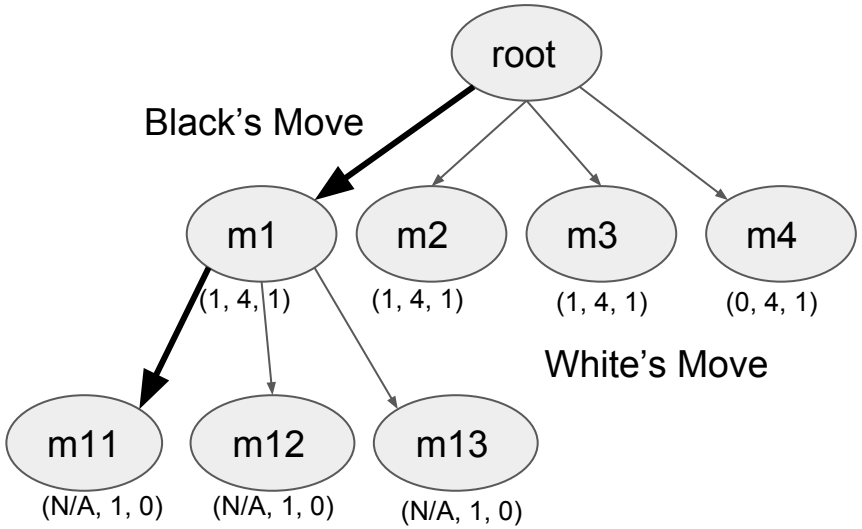
	X_j	n	n_j
m1	1	4	1
m2	1	4	1
m3	1	4	1
m4	0	4	1

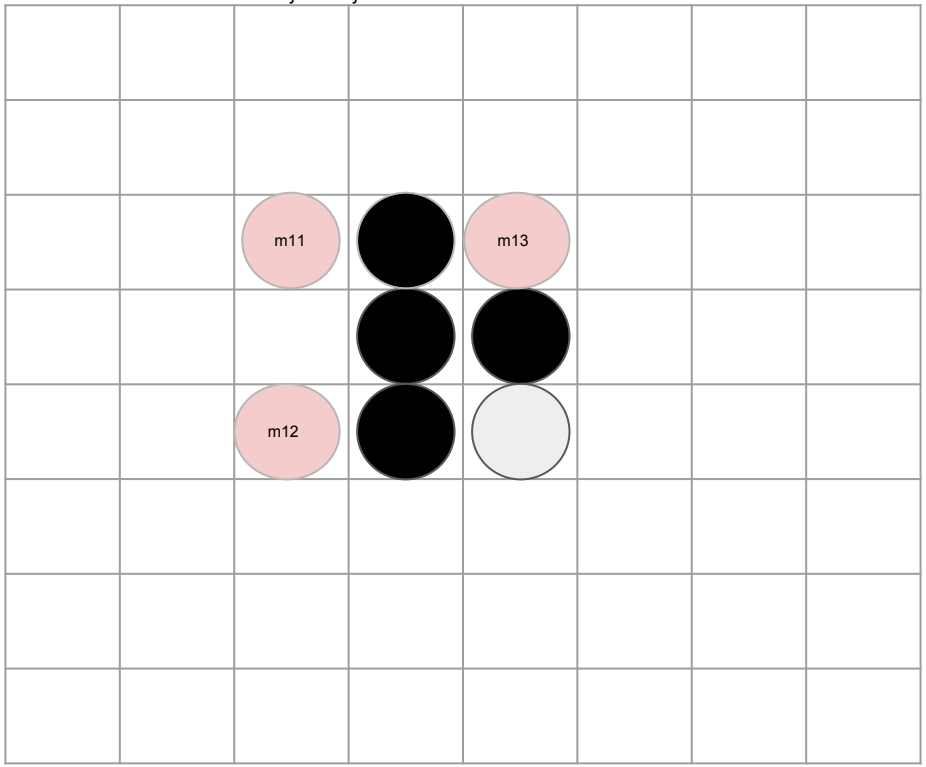


Example - The Game of Othello Iter #5

$$UCT = \overline{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

(X_j, n, n_j) - (Mean Value, Parent Visits, Child Visits)



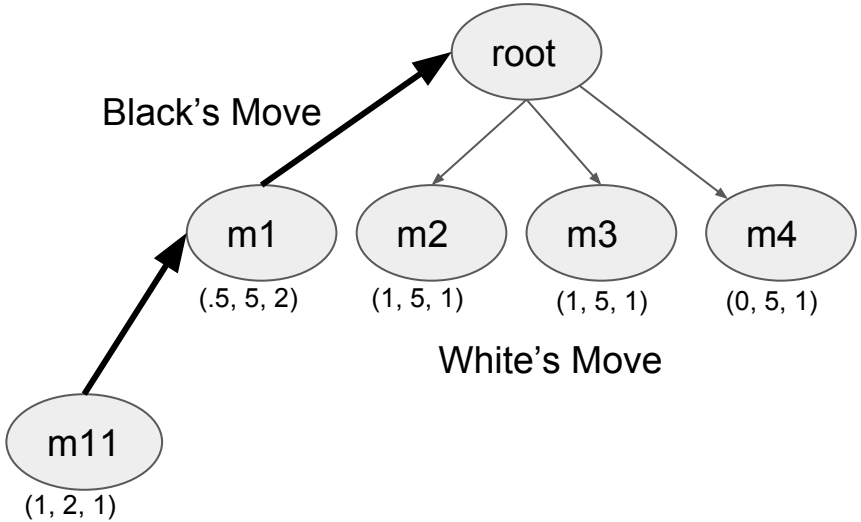


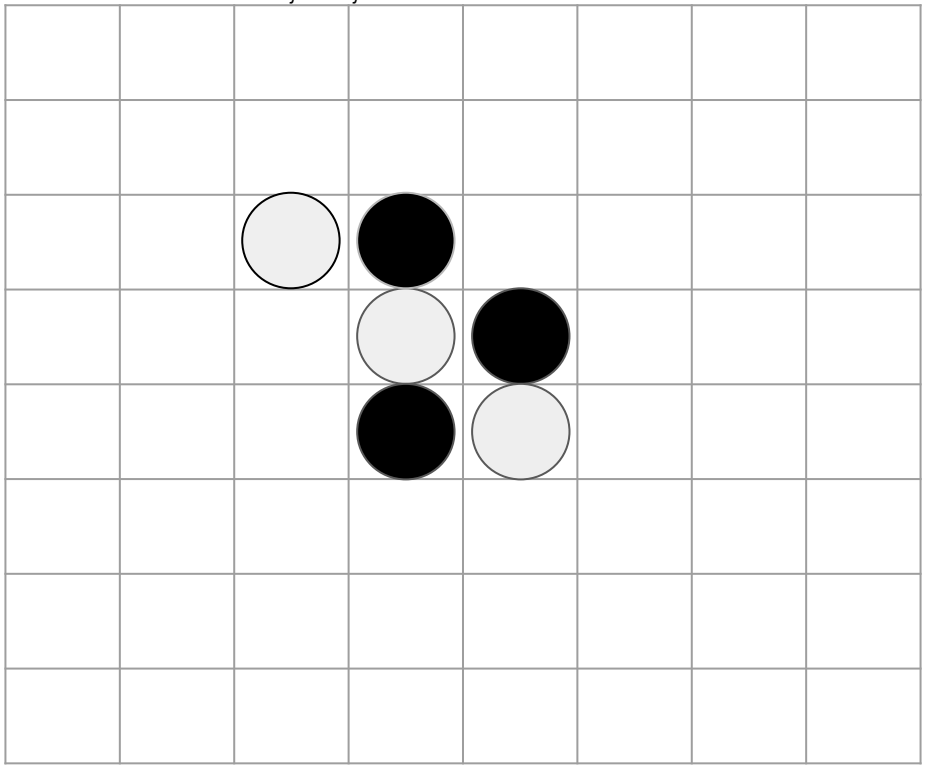
- First selection picks m1
- Second selection picks m11

Example - The Game of Othello Iter #5

$$UCT = \overline{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

(X_j, n, n_j) - (Mean Value, Parent Visits, Child Visits)



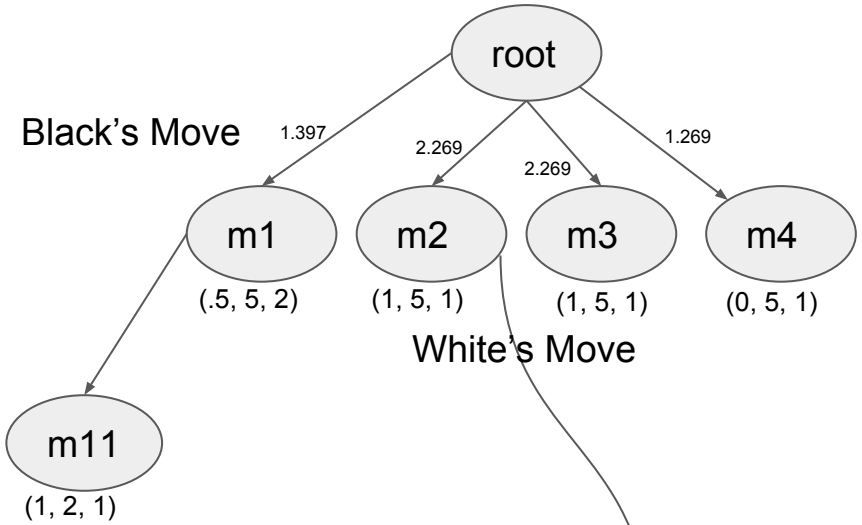


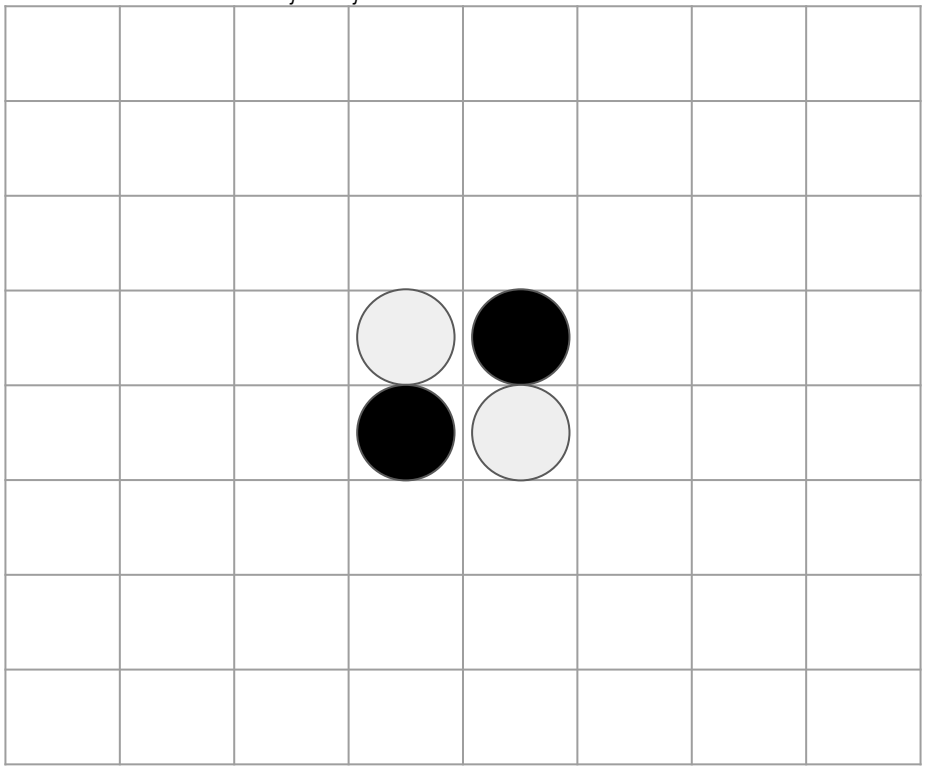
- Run a simulation
- White Wins
- Backtrack, and update mean scores accordingly.

Example - The Game of Othello Iter #6

$$UCT = \overline{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

(X_j, n, n_j) - (Mean Value, Parent Visits, Child Visits)



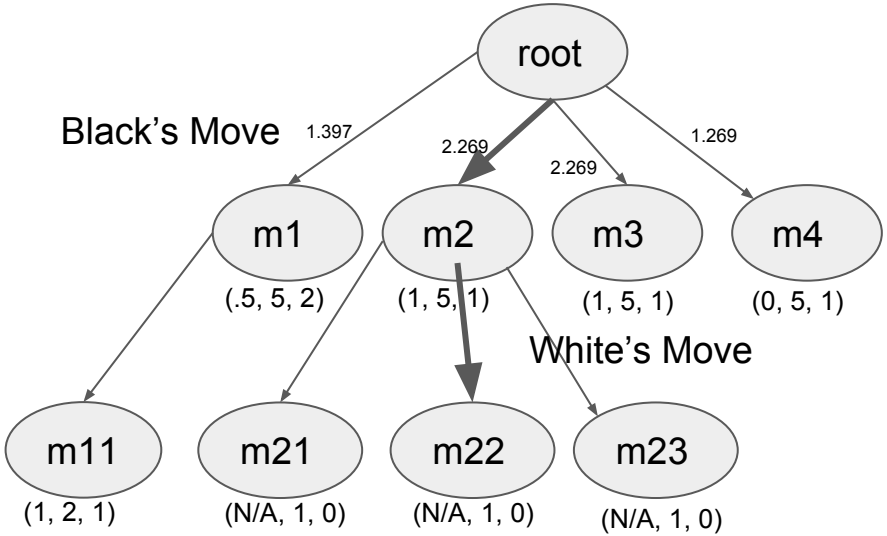


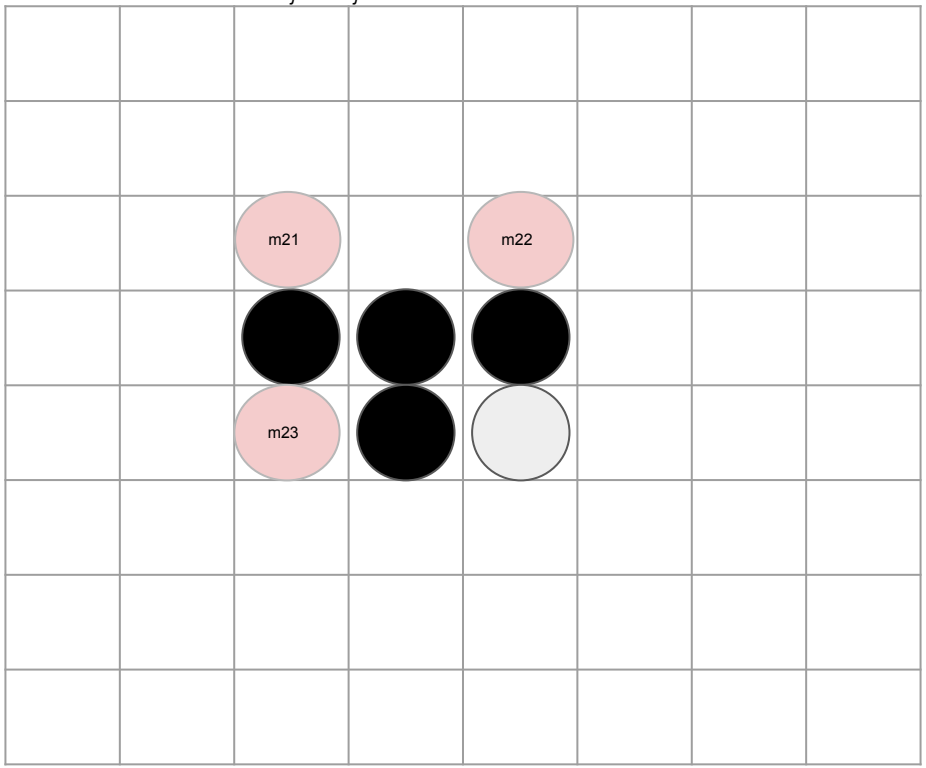
- Suppose we first select m2

Example - The Game of Othello Iter #6

$$UCT = \overline{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

(X_j, n, n_j) - *(Mean Value, Parent Visits, Child Visits)*



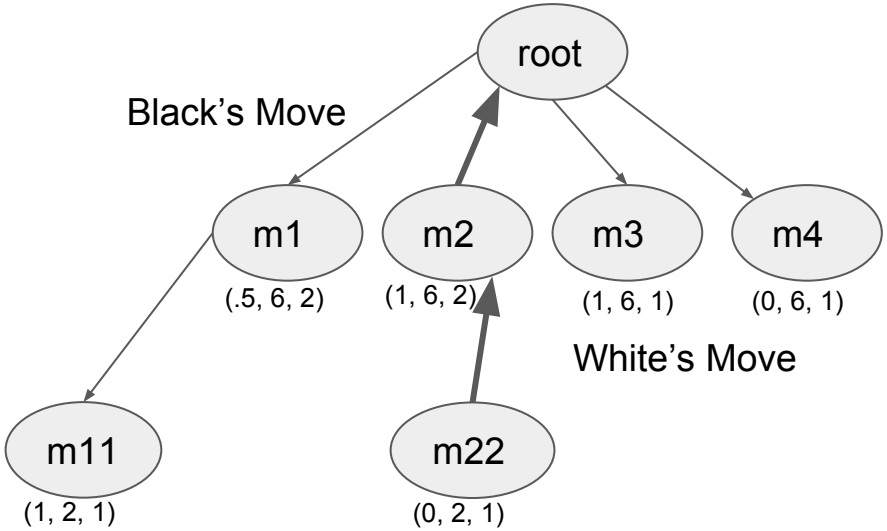


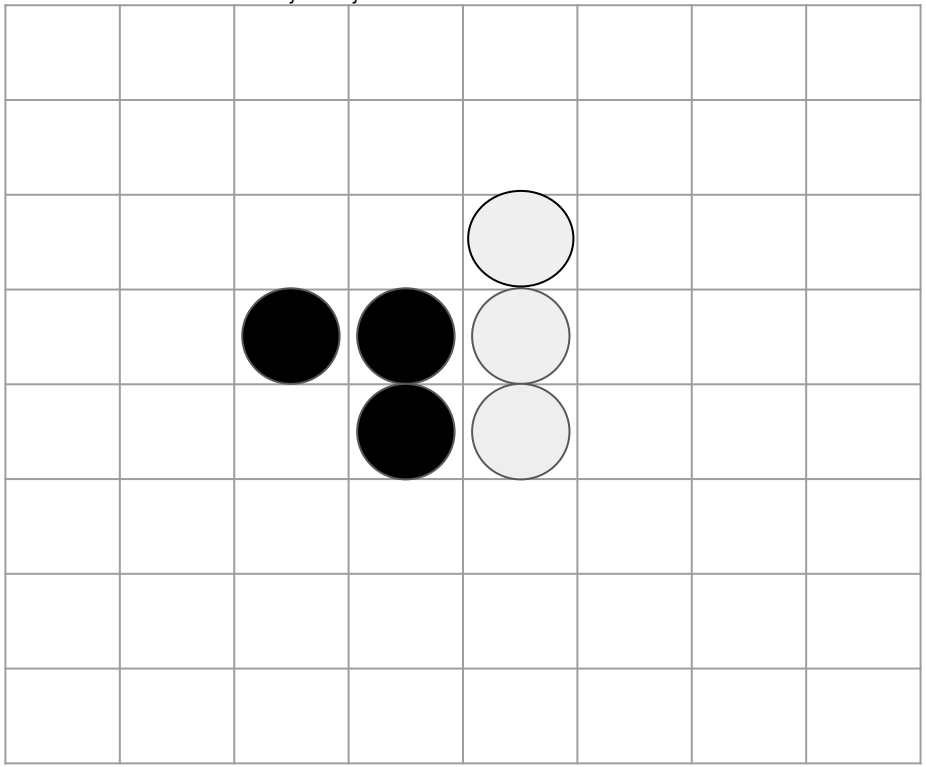
- Suppose we pick m22

Example - The Game of Othello Iter #6

$$UCT = \overline{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

(X_j, n, n_j) - (Mean Value, Parent Visits, Child Visits)



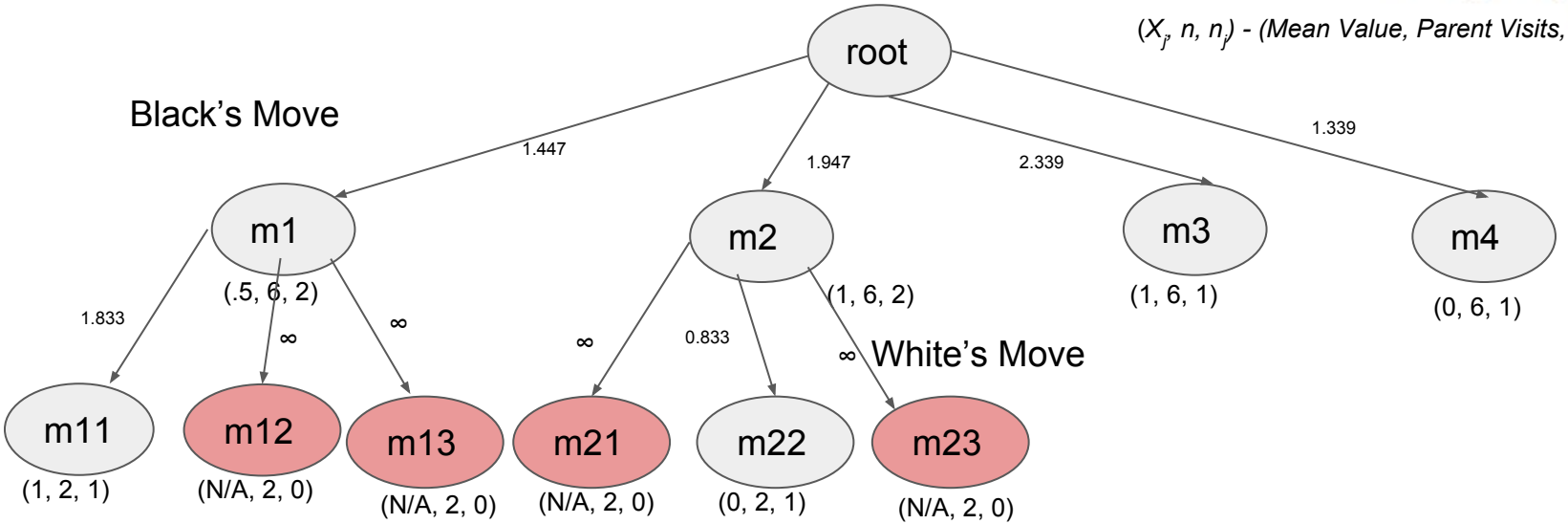


- Run simulated game from this position.
- Suppose black wins the simulated game.
- Backtrack and update values

Example - The Game of Othello Iter #6

$$UCT = \overline{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

(X_j, n, n_j) - (Mean Value, Parent Visits, Child Visits)



- This is how our tree looks after 6 iterations.
- Red Nodes not actually in tree
- Now given a tree, actual moves can be made using max, robust, max-robust, or other child selection policies.
- Only care about subtree after moves have been made

MCTS - Algorithm Recap

- Applied to solve Multi-Arm Bandit problem in a tree structure
  - UCT = UCB1 applied at each subproblem
- Due to tree structure same move can have different rewards in different subtrees
- Weight to go to a given node:
  - Mean value for paths involving node
  - Visits to node
  - Visits to parent node
  - Constant balancing exploration vs exploitation
- Determines values from Default Policy
- Determines how to choose child from Tree Policy
- Once you have acomplete tree - number of ways to pick moves during game - Max, Robust, Max-Robust, etc.

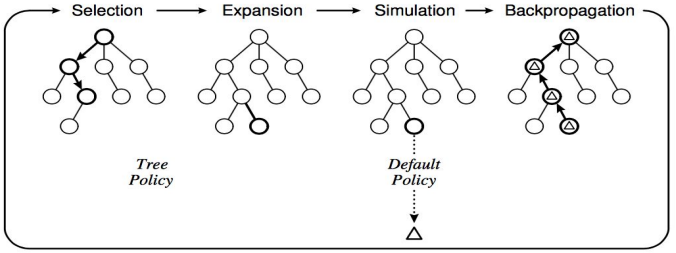


Fig. 2. One iteration of the general MCTS approach.

Analysis of UCT Algorithm

UCT Algorithm Convergence

- UCT is an application of the bandit algorithm (UCB1) for Monte Carlo search
- In the case of Go, the estimate of the payoffs is non-stationary (mean payoff of move shifts as games are played)
- Vanilla MCTS has not been shown to converge to the optimal move (even when iterated for a long period of time) for non-stationary bandit problems
- UCT Algorithm does converge to optimal move at a polynomial rate at the root of a search tree with non-stationary bandit problems
- Assumes that the expected value of partial averages converges to some value, and that the probability that experienced average payoff is a $\sqrt{n \ln(1/\delta)}$ factor off of the expected average is less than delta if we play long enough

UCT Algorithm Convergence

- Builds on earlier work by Auer (2002) who proved UCB1 algorithm converged for stationary distributions
- Since UCT algorithm views each visited node as running a separate UCB1 algorithm, bounds are made on expected number of plays on suboptimal arms, pseudo-regret measure, deviation from mean bounds, and eventually proving that UCB1 algorithm plays an suboptimal arm with 0 probability giving enough time
- Kocsis and Szepesvári's work was very similar, with additions of ε-δ type arguments using the convergence of payoff drift to remove the effects of drift in their arguments, especially important in their regret upper bounds

UCT Algorithm Convergence

- After showing UCB1 correctly converges to the optimal arm, the convergence of UCT follows with an induction argument on search tree depth
- For a tree of depth D, we can consider the all children of the root node and their associated subtrees.
- Induction hypothesis gives probability of playing suboptimal arm goes to 0 (base case is just UCB1), and the pseudo-regret bounds and deviation from partial mean bounds ensures the drift is accounted for
- The most important takeaway is when a problem can be rephrased in terms of multi-armed bandits (even with drifting average payoff), similar steps can be used to show failure probability goes to 0

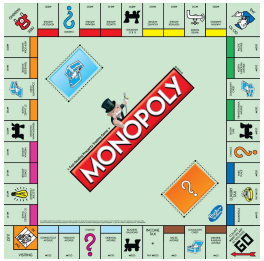
Variations to MCTS

Applying MCTS to different game domains

Go and other Games

- Go is a combinatorial game.
  - Zero-sum, perfect information, deterministic, discrete and sequential.

- What happens when some of these aspects of the game change?



Multi-player MCTS

- The central principle of minimax search:
  - The searching player seeks to find the move to maximize their reward while their opponent seeks to minimize it.
  - In the case of two players: each player seeks to maximize their own reward.
- Not necessarily true in the case of more than two players.
  - Is the loss of player 1 and gain of player 2 necessarily a gain for player 3?

Multi-player MCTS

- More than 2 players does not guarantee zero-sum game.
- No perfect way to model reward/loss among all players
- Simple suggestion - max^n idea:
  - Nodes store a vector of rewards.
  - UCB then seeks to maximize the value using the appropriate vector component depending.
  - Components of vector used depend on the current player.
  - But how exactly are these components combined?

MCTS in Multi-player Go

- Cazenave applies several variants of UCT to Multi-player Go.
  - Because players can have common enemies he considers the possibility of “coalitions”
- Uses max^n, but takes into account the moves that may be adversarial towards coalition members.
- Changes scoring to include the coalition stones as if they were the player’s own.

MCTS in Multi-player Go

- Different ways to treat coalitions:
  - *Paranoid UCT*: player assumes all other players are in coalition against him.
    - Coalition Reduction
    - Usually better than Confident.
  - *Confident UCT:* searches are completed with the possibility of coalition with each other one player. Move is selected based on whichever coalition could prove most beneficial.
    - Better when algorithms of other players are known.
  - Etc.
- No known perfect way to model strategy equilibrium between more than two players.

Variation Takeaway

- Game Properties:
  - Zero-sum: Reward across all players sums to zero.
  - Information: Fully or partially observable to the players.
  - Determinism: Chance Factors?
  - Sequential/Simultaneous actions.
  - Discrete: Whether actions are discrete or applied in real-time.
- MCTS is altered in order to apply to different games not necessarily combinatorial.

AlphaGo

Go

- 2 player
- Zero-sum
- 19x19 board
- Very large search tree
  - Breadth ≈ 250, depth ≈ 150
  - Unlike chess
- No amazing heuristics
  - Human intuition hard to replicate
- Great candidate for applying MCTS
  - Vanilla MCTS not good enough

How to make MCTS work for Go?

Idea 1: Value function to truncate tree -> shallower MCTS search

Idea 2: Better tree & default policies -> smarter MCTS search

- Value function
  - Expected future reward from board *s* assuming we play perfectly from that point
- Tree policy
  - Selecting which part of the search tree to expand
- Default policy
  - Determine how simulations are run
  - Ideally, should be perfect player

Before AlphaGo

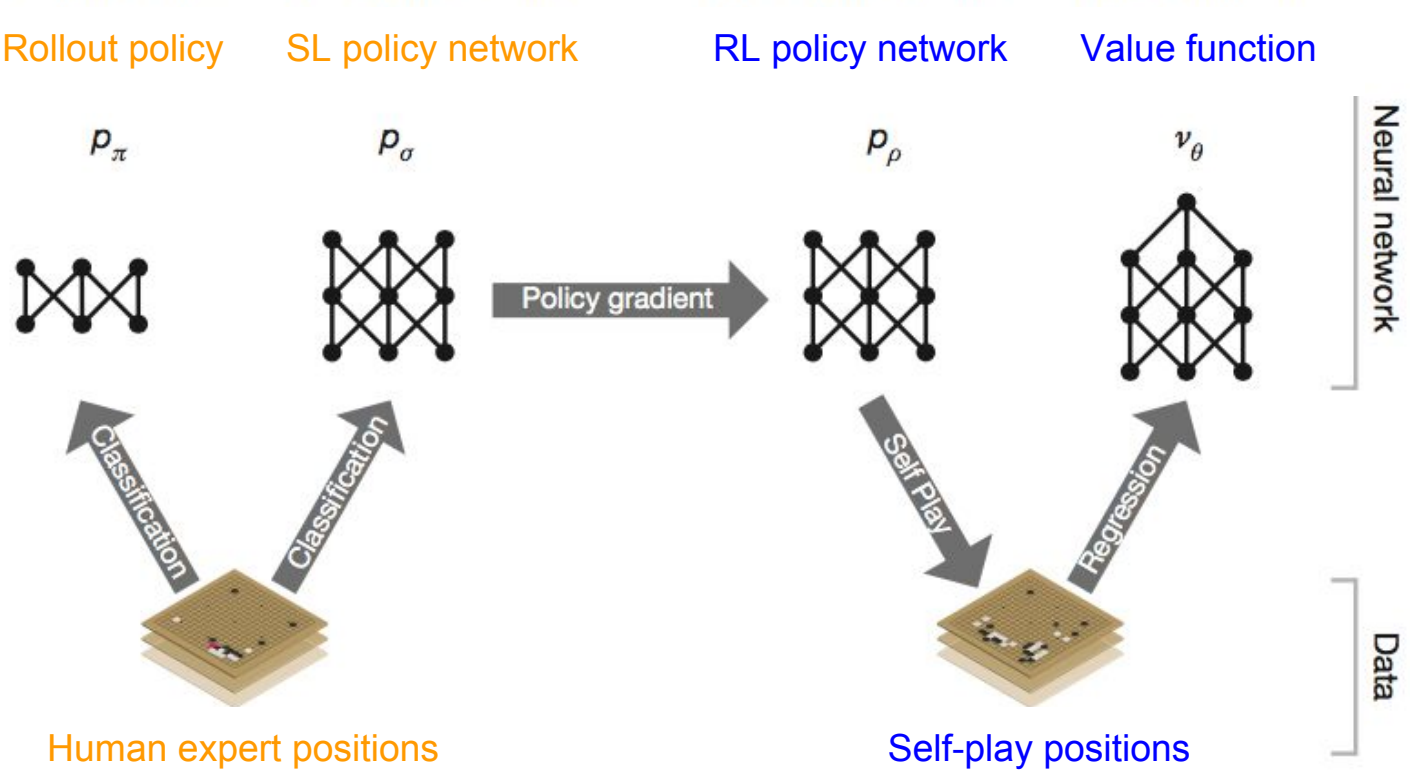
- Strongest programs
  - MCTS
  - Enhanced by policies predicting expert moves
    - Narrow search tree
  - Limitations
    - Simple heuristics from expert players
    - Value functions based on linear combinations of input features
  - Cannot capture full breadth of human intuition
  - Generally only looking a few moves ahead
  - Local v global approach to reward

AlphaGo - Training

- AlphaGo
  - Uses both ideas for improving MCTS
  - Two resources
    - Expert data
    - Simulator (self-play)
  - Value function
    - Expected future reward from a board *s* assuming we play perfectly from that point
  - Tree & Default Policy networks
    - Probability distributions over possible moves *a* from a board *s*
    - Distribution encodes reward estimates

Main idea: For better policies and value functions, train with deep convolutional networks

AlphaGo - Training

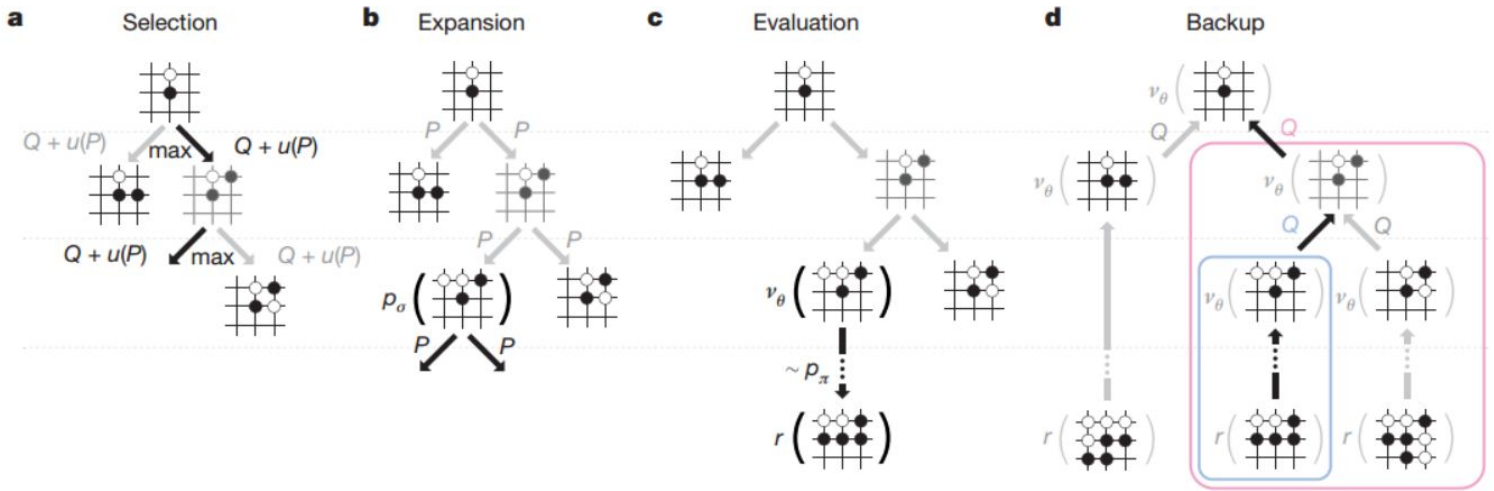


AlphaGo - Training

- Supervised Learning network p_σ
  - Slow to evaluate
  - Goal = predict expert moves well, prior probabilities for each move
- Fast rollout network p_π
  - Default policy
  - Goal = quick simulation/evaluation
- Reinforcement Learning network p_ρ
  - Play games between current network and randomly selected previous iteration
  - Goal = optimize on game play, not just predicting experts
- Value function $v^P(s)$
  - Self-play according to optimal policies p_r for both players from p_ρ
  - Default policy
  - Function of a board, not probability distribution of moves
  - Goal = get expected future reward assuming our best estimate of perfect play

AlphaGo - Playing

- Each move
  - Time constraint
  - Deepen/build our MCTS search tree
  - Select our optimal move and only consider subtree from there



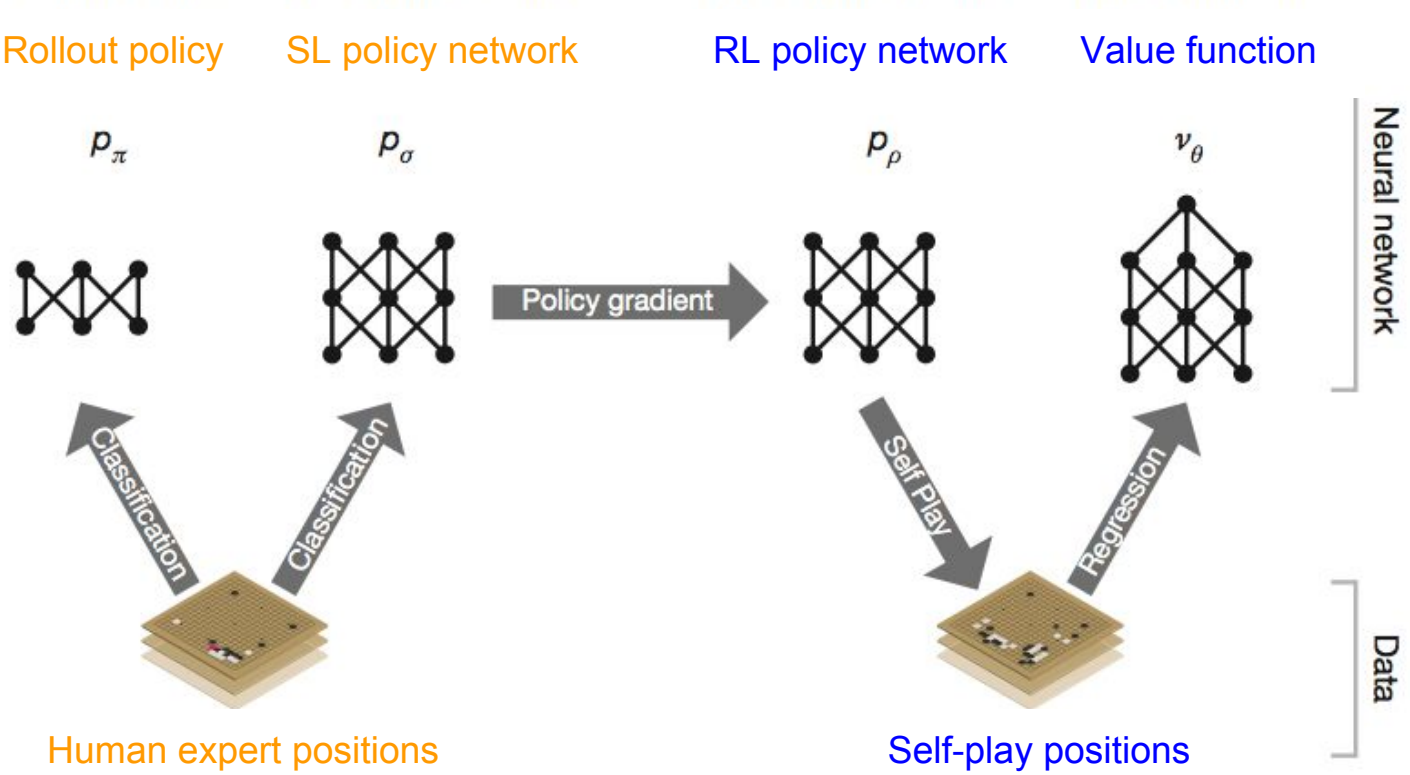
AlphaGo - Playing (Selection/Tree Policy)

$$a_t = \text{argmax}_a(Q(s_t, a) + u(s_t, a))$$

$$u(s, a) \propto \frac{P(s, a)}{1 + N(s, a)}$$

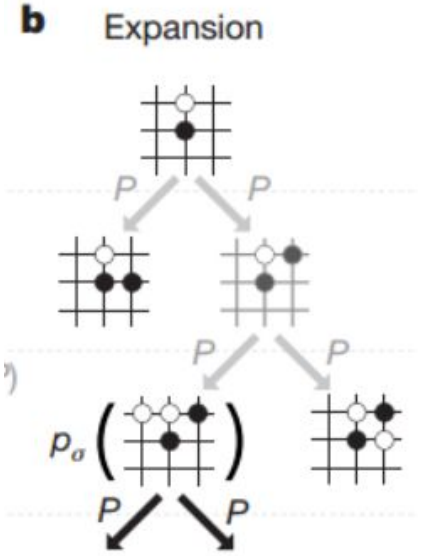
- a_t - action selected at time step *t* from board s_t
- $Q(s_t, a)$ - average reward for playing this move (exploitation term)
- $P(s, a)$ - prior expert probability of playing moving *a*
- $N(s, a)$ - number of times we have visited parent node
- *u* acts as a bonus value
    - Decays with repeated visits

AlphaGo - Playing (Policy Recap)



AlphaGo - Playing (Expansion)

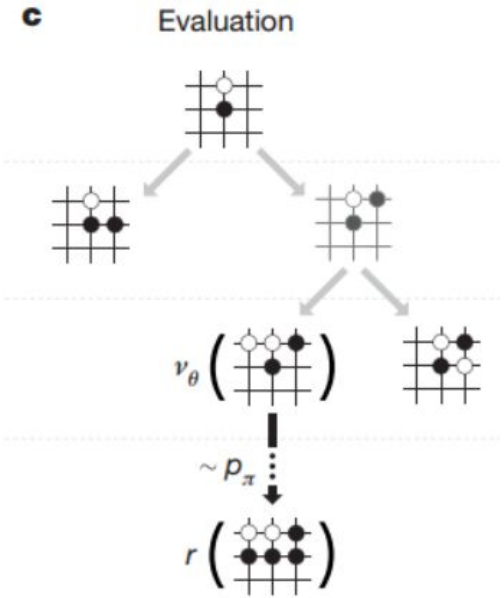
- When leaf node is reached, it has a chance to be expanded
- Processed once by SL policy network (p_σ) and stored as prior probs *P(s, a)*
- Pick child node with highest prior prob



AlphaGo - Playing (Evaluation/Default Policy)

$$V(s_L) = (1 - \lambda)v_\theta(s_L) + \lambda z_L$$

- Default policy, of sorts
- v_θ - value from value function of board position s_L
- z_L - Reward from fast rollout p_π
  - Played until terminal step
- λ - mixing parameter
  - Empirical

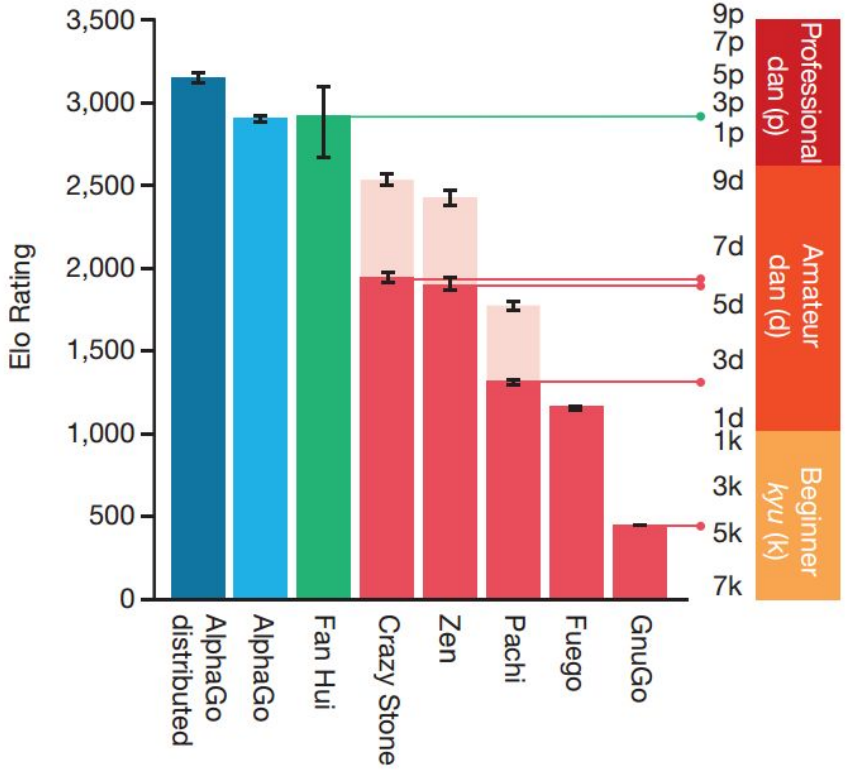


AlphaGo - Playing (Backup)

$$N(s,a) = \sum_{i=1}^{n} 1(s,a,i) \qquad Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{n} 1(s,a,i) V(s_L^i)$$

- Extra index *i* is to denote the i^{th} simulation, *n* total simulations
- Update visit count and mean reward of simulations passing through node
- Once search completes:
  - Algorithm chooses the most visited move from the root position

AlphaGo Results



AlphaGo Takeaway

- You should work for Google

- Tweaks to MCTS are not independently novel
- Deep learning allows us to train good policy networks
- Have data and computation power for deep learning
  - Can now solve a huge game such as Go
- Method applicable to other 2 player zero-sum games as well

Questions?