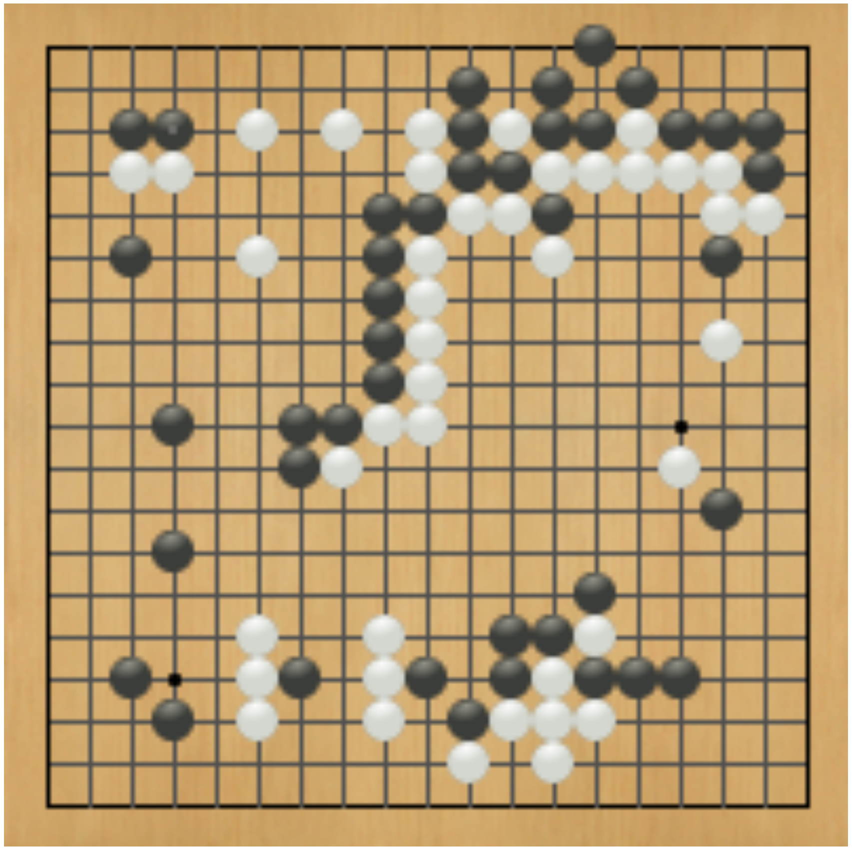
Lecture 33 – Reinforcement Learning for Two-Player Games

Mark Hasegawa-Johnson, 4/2020



Snapshot of a gnugo game, http://www.gnu.org/software/gnugo/

Outline

- Review: minimax and alpha-beta
- Move ordering: policy network
- Evaluation function: value network
- Training the value network
    - Exact training: endgames
    - Stochastic training: Monte Carlo tree search
- Case study: alphago

Minimax games

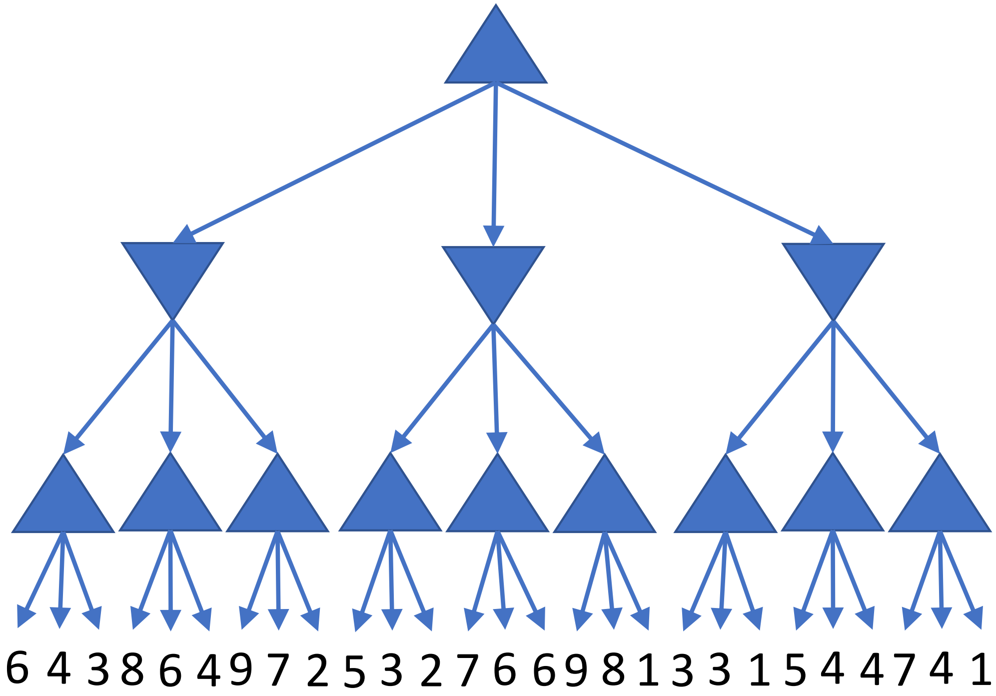
Let s be the state of the game: complete specification of the board, and a statement about whose turn it is.

- If it’s the turn of the MAX player, and if $C(s)$ are the children of s (the set of states reachable in one move), then the value of the board is

$$U(s) = \max_{s' \in C(s)} U(s')$$

- If it’s MIN’s turn, then

$$U(s) = \min_{s' \in C(s)} U(s')$$



Minimax games

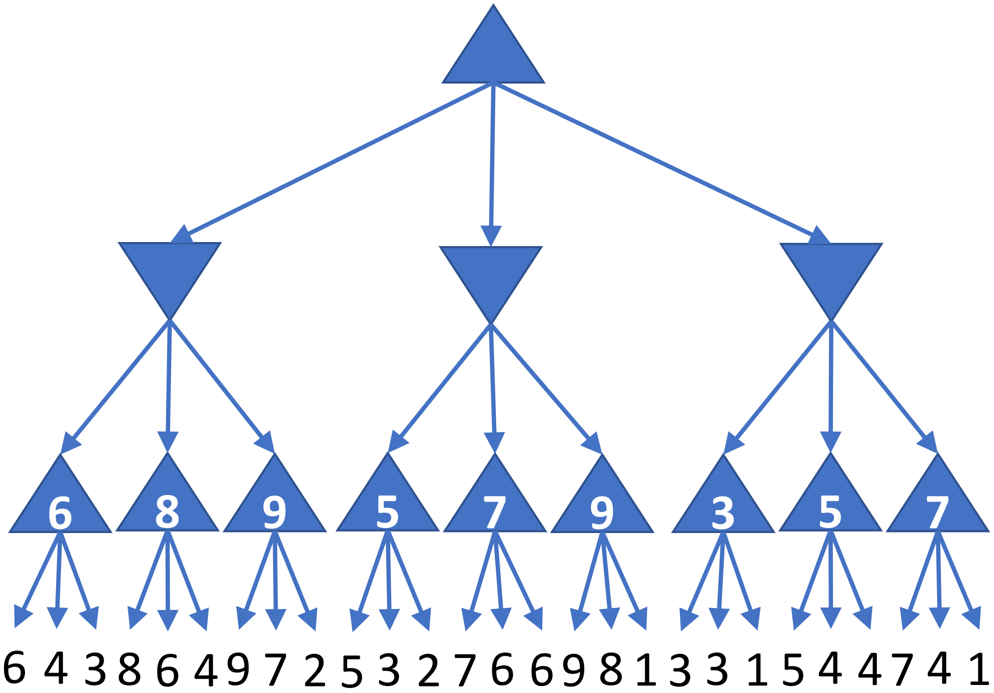
Let s be the state of the game: complete specification of the board, and a statement about whose turn it is.

- If it's the turn of the MAX player, and if $C(s)$ are the children of s (the set of states reachable in one move), then the value of the board is

$$\boxed{U(s) = \max_{s' \in C(s)} U(s')}$$

- If it's MIN's turn, then

$$U(s) = \min_{s' \in C(s)} U(s')$$



Minimax games

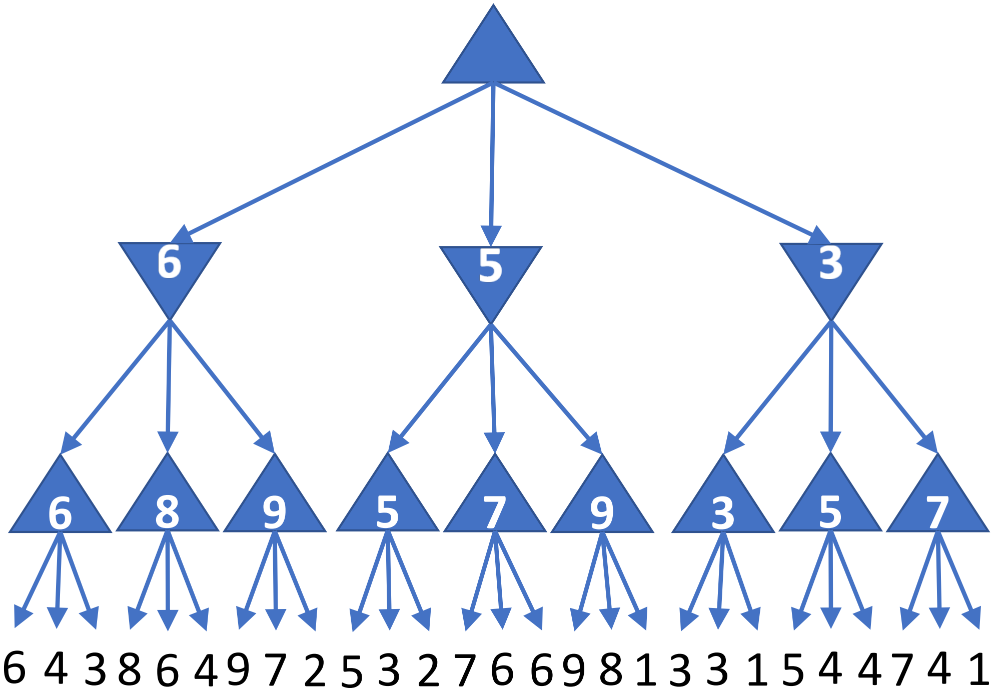
Let s be the state of the game: complete specification of the board, and a statement about whose turn it is.

- If it's the turn of the MAX player, and if $C(s)$ are the children of s (the set of states reachable in one move), then the value of the board is

$$U(s) = \max_{s' \in C(s)} U(s')$$

- If it's MIN's turn, then

$$\boxed{U(s) = \min_{s' \in C(s)} U(s')}$$



Minimax games

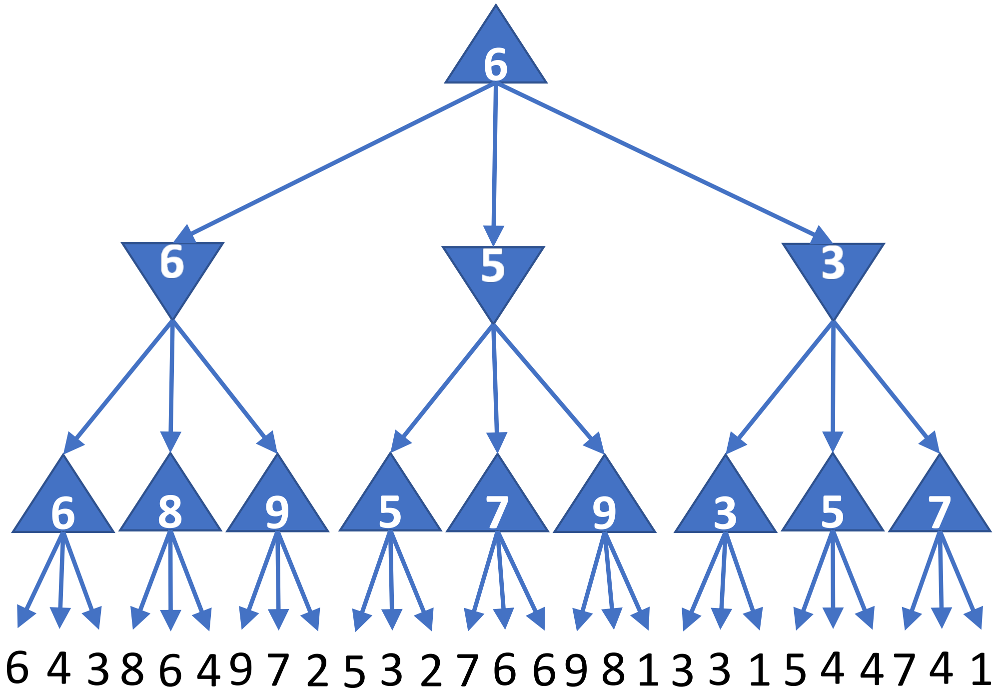
Let s be the state of the game: complete specification of the board, and a statement about whose turn it is.

- If it's the turn of the MAX player, and if $C(s)$ are the children of s (the set of states reachable in one move), then the value of the board is

$$\boxed{U(s) = \max_{s\prime \in C(s)} U(s')}$$

- If it's MIN's turn, then

$$U(s) = \min_{s\prime \in C(s)} U(s')$$

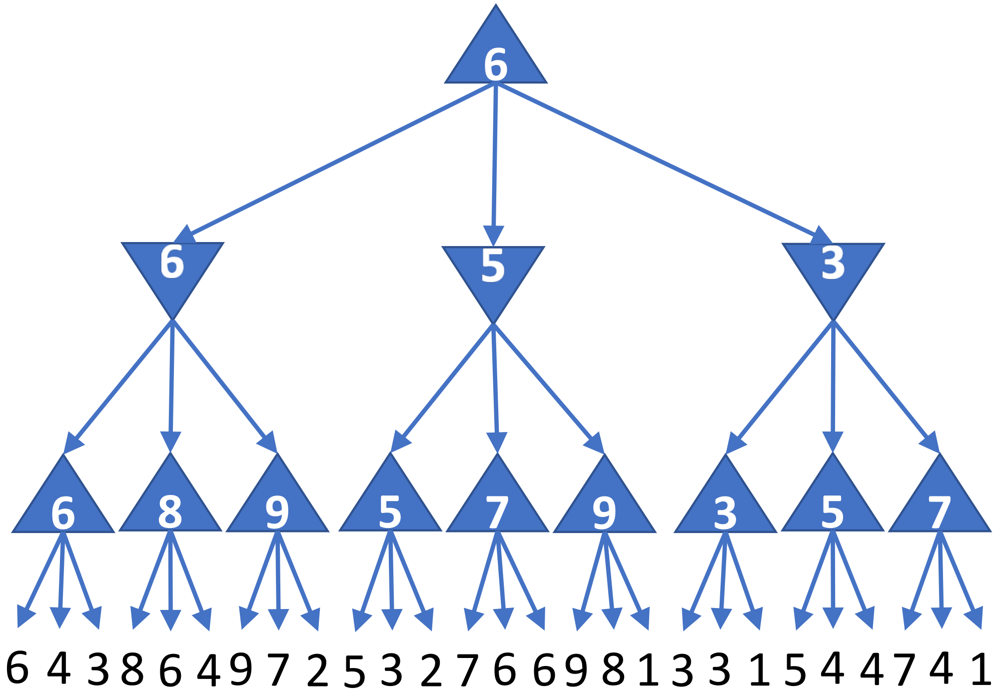


Minimax complexity

b =branching factor

d =search depth

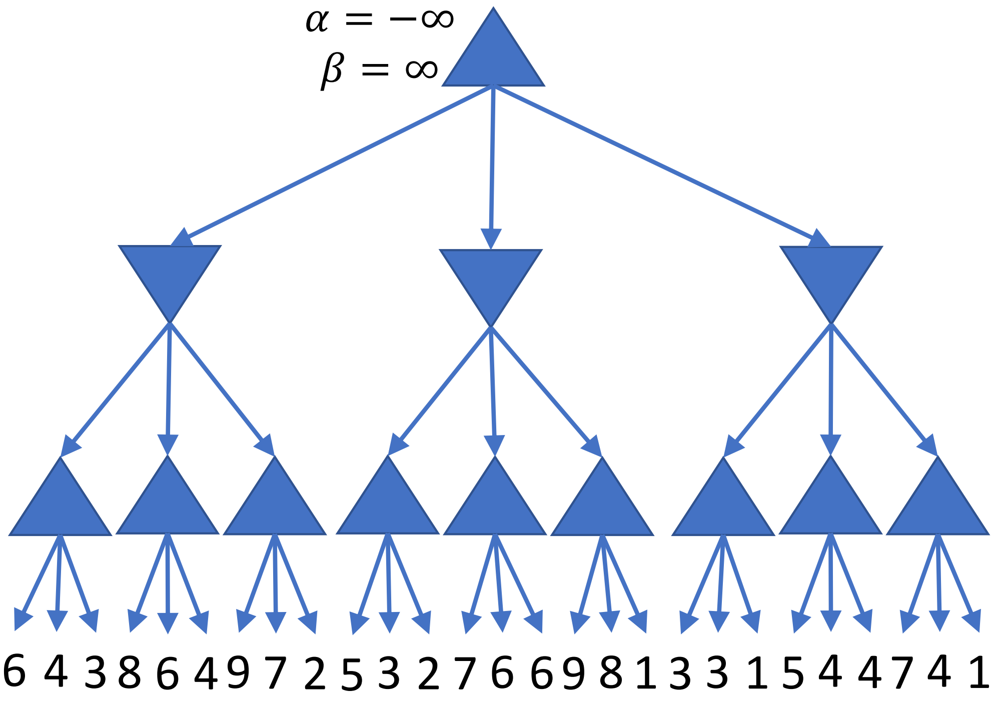
Complexity = $O\{b^d\}$



Alpha-Beta Pruning

Each node has two internal meta-parameters, initialized from its parent:

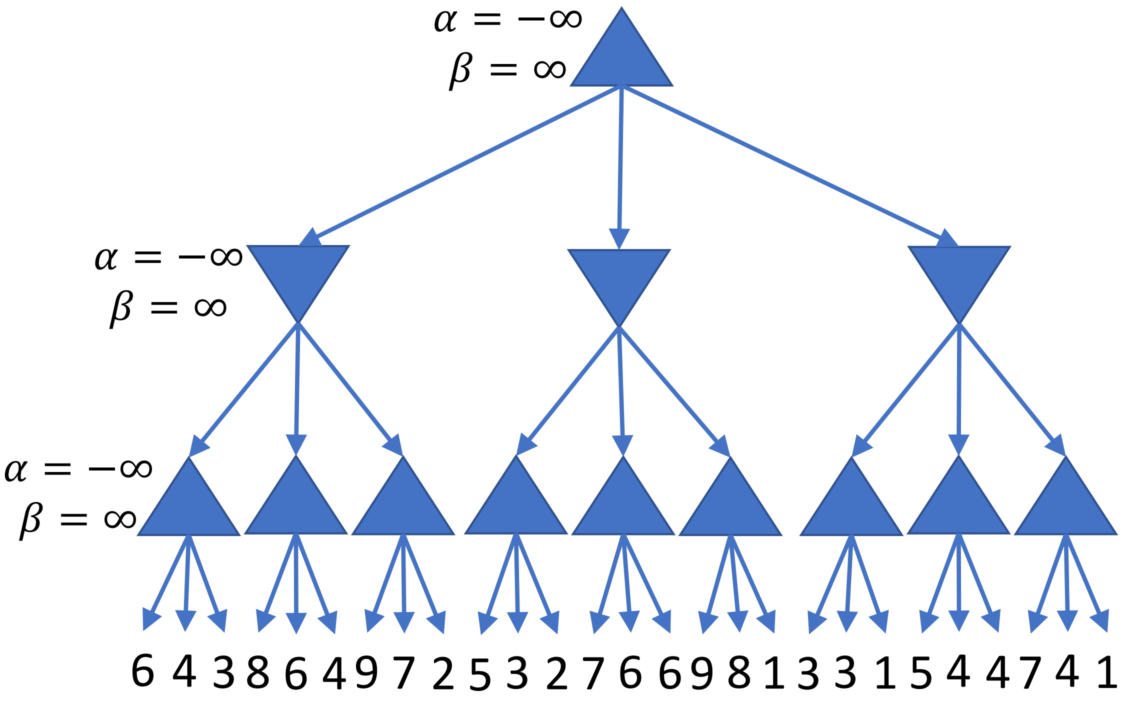
- $\alpha =$ highest value that MAX knows how to force MIN to accept
- $\beta =$ lowest value that MIN knows how to force MAX to accept
- $\alpha \leq \beta$
- Initial values: $\alpha = -\infty,\ \beta = \infty$



Alpha-Beta Pruning

Each node has two internal meta-parameters, initialized from its parent:

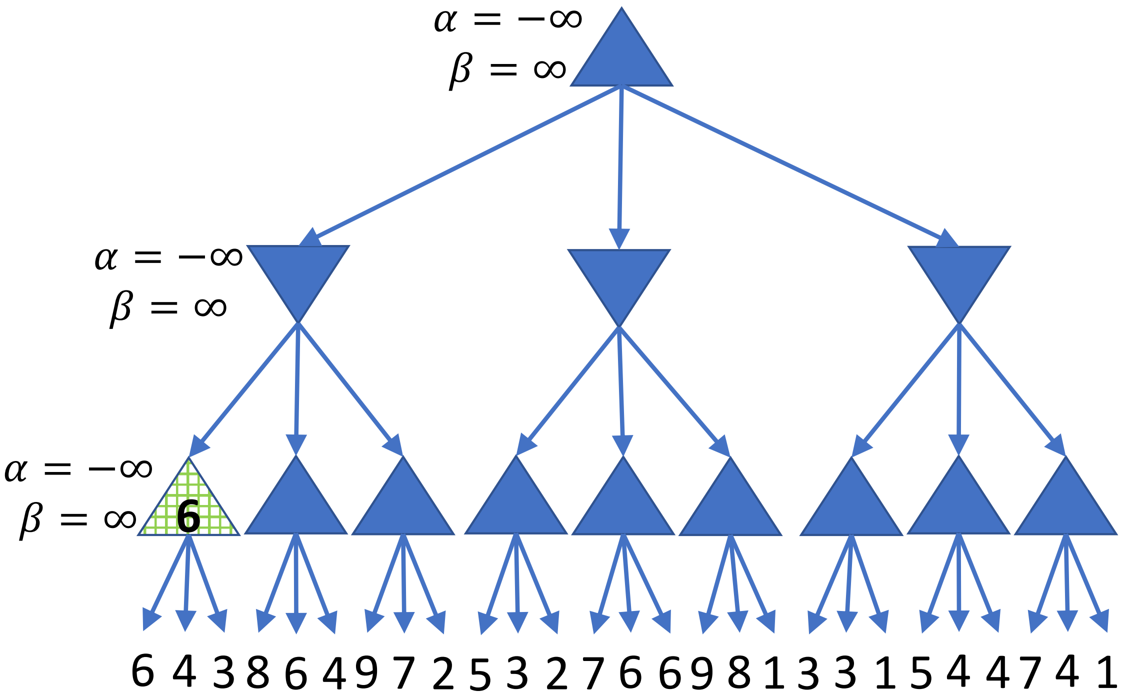
- $\alpha =$ highest value that MAX knows how to force MIN to accept
- $\beta =$ lowest value that MIN knows how to force MAX to accept
- $\alpha \leq \beta$
- Initial values: $\alpha = -\infty,\ \beta = \infty$



Alpha-Beta Pruning

If s is a ***MAX*** node, then:

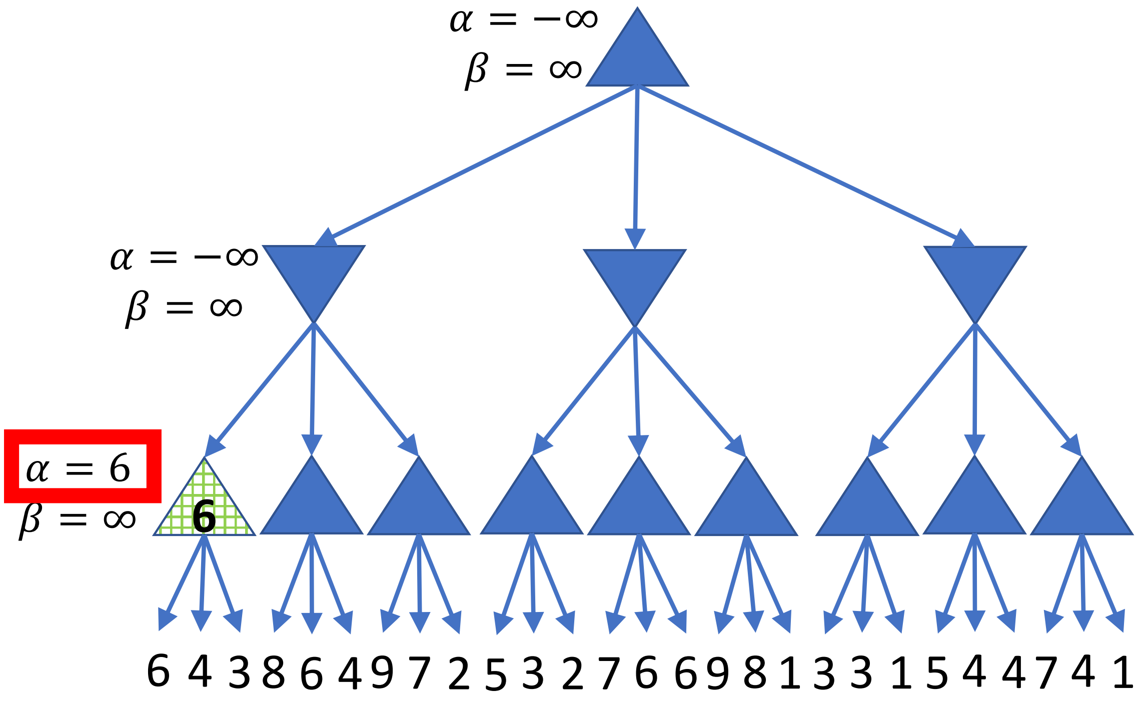
- For each child $s' \in C(s)$:
  - If you realize that $U(s') > \beta(s)$ then prune all remaining children of s: MIN will never let us reach this node.
  - Otherwise, if $U(s') > \alpha(s)$, then set $\alpha(s) = U(s')$. MIN might still choose s (because $U(s') \leq \beta(s)$), then MAX can choose s'.



Alpha-Beta Pruning

If s is a ***MAX*** node, then:

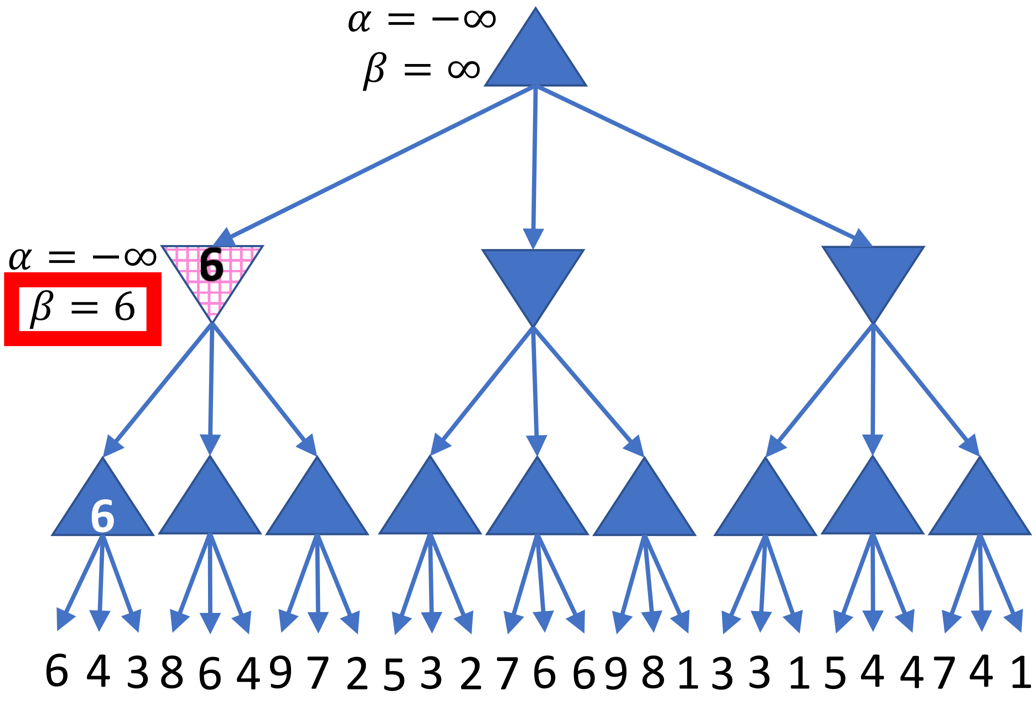
- For each child $s' \in C(s)$:
  - If you realize that $U(s') > \beta(s)$ then prune all remaining children of s: MIN will never let us reach this node.
  - Otherwise, if $U(s') > \alpha(s)$, then set $\alpha(s) = U(s')$. MIN might still choose s (because $U(s') \leq \beta(s)$), then MAX can choose s'.



Alpha-Beta Pruning

If s is a ***MIN*** node, then:

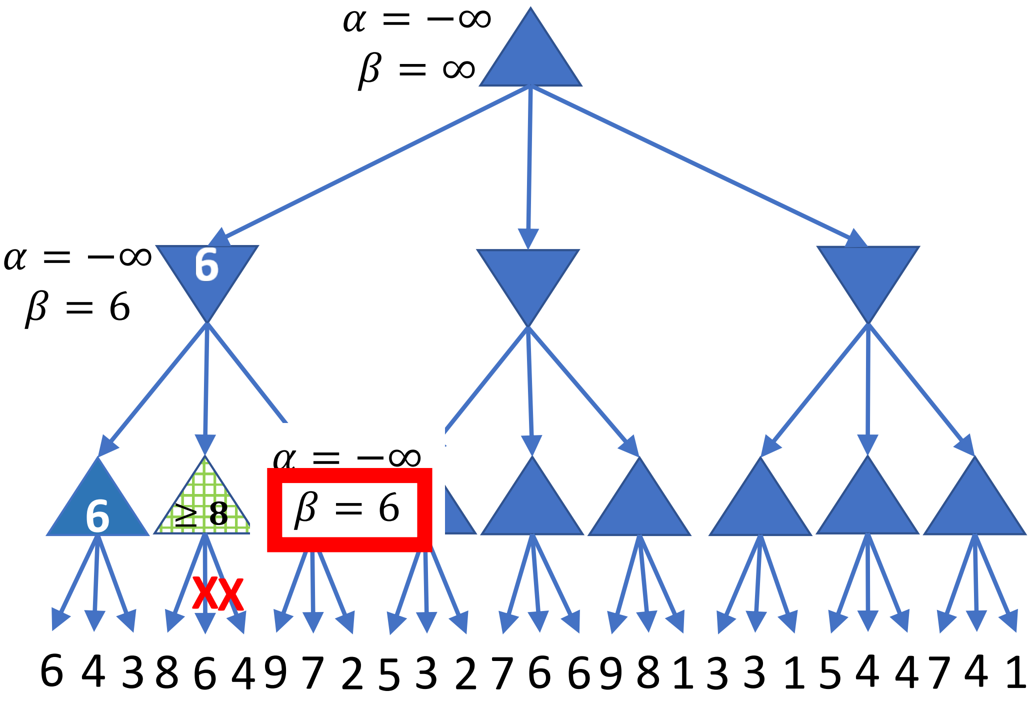
- For each child $s' \in C(s)$:
  - If you realize that $U(s') < \alpha(s)$ then prune all remaining children of s: MIN will never let us reach this node.
  - Otherwise, if $U(s') < \beta(s)$, then set $\beta(s) = U(s')$. MAX might still choose s (because $U(s') \geq \alpha(s)$), then MIN can choose s'.



Alpha-Beta Pruning

If s is a ***MAX*** node, then:

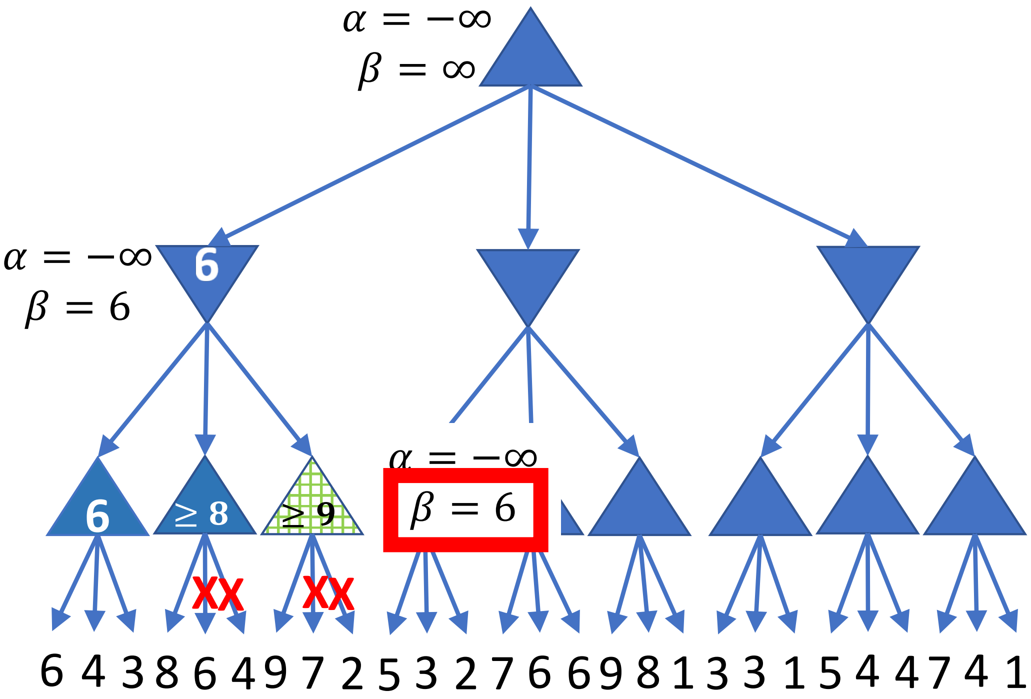
- For each child $s' \in C(s)$:
  - If you realize that $U(s') > \beta(s)$ then prune all remaining children of s: MIN will never let us reach this node.
  - Otherwise, if $U(s') > \alpha(s)$, then set $\alpha(s) = U(s')$. MIN might still choose s (because $U(s') \le \beta(s)$), then MAX can choose s'.



Alpha-Beta Pruning

If s is a ***MAX*** node, then:

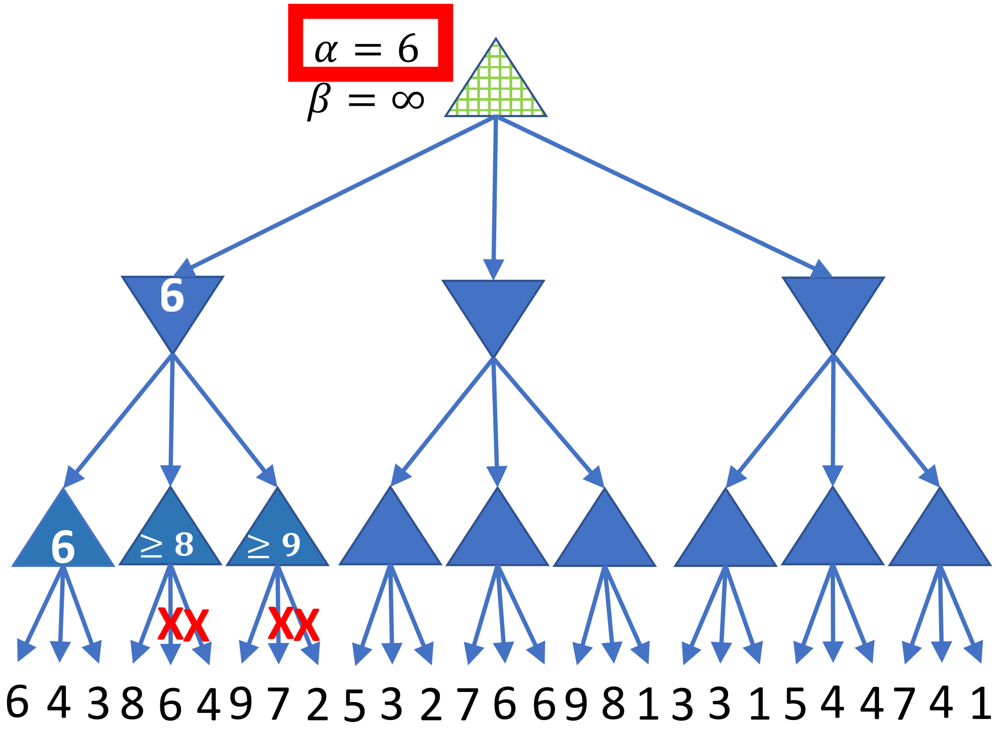
- For each child $s' \in C(s)$:
  - If you realize that $U(s') > \beta(s)$ then prune all remaining children of s: MIN will never let us reach this node.
  - Otherwise, if $U(s') > \alpha(s)$, then set $\alpha(s) = U(s')$. MIN might still choose s (because $U(s') \le \beta(s)$), then MAX can choose s'.



Alpha-Beta Pruning

If s is a ***MAX*** node, then:

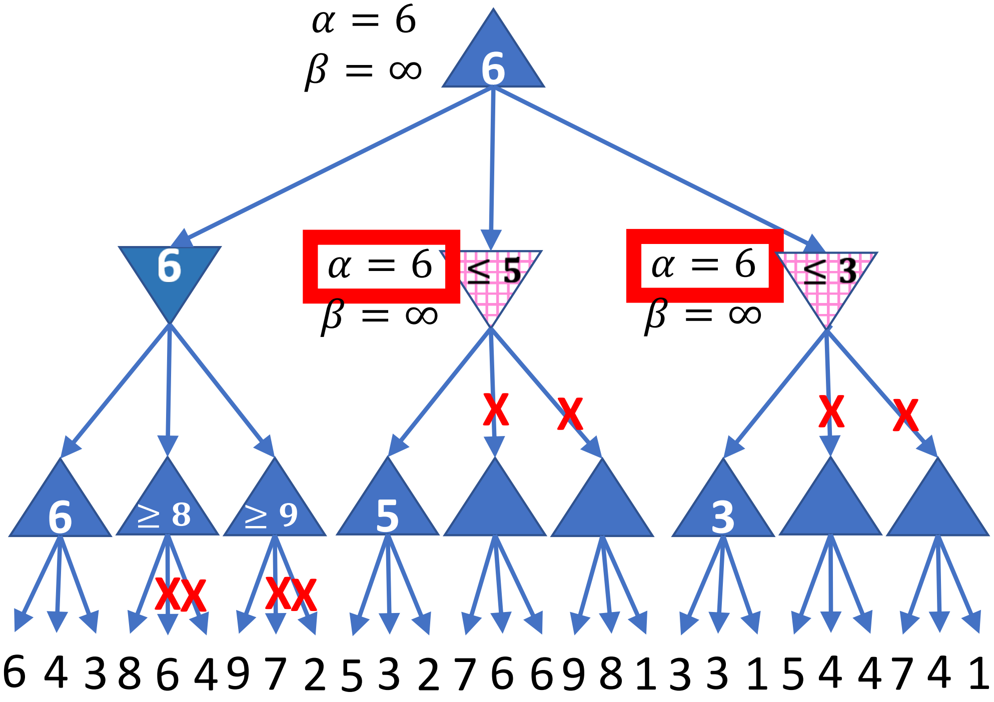
- For each child $s' \in C(s)$:
  - If you realize that $U(s') > \beta(s)$ then prune all remaining children of s: MIN will never let us reach this node.
  - Otherwise, if $U(s') > \alpha(s)$, then set $\alpha(s) = U(s')$. MIN might still choose s (because $U(s') \leq \beta(s)$), then MAX can choose s'.



Alpha-Beta Pruning

If s is a ***MIN*** node, then:

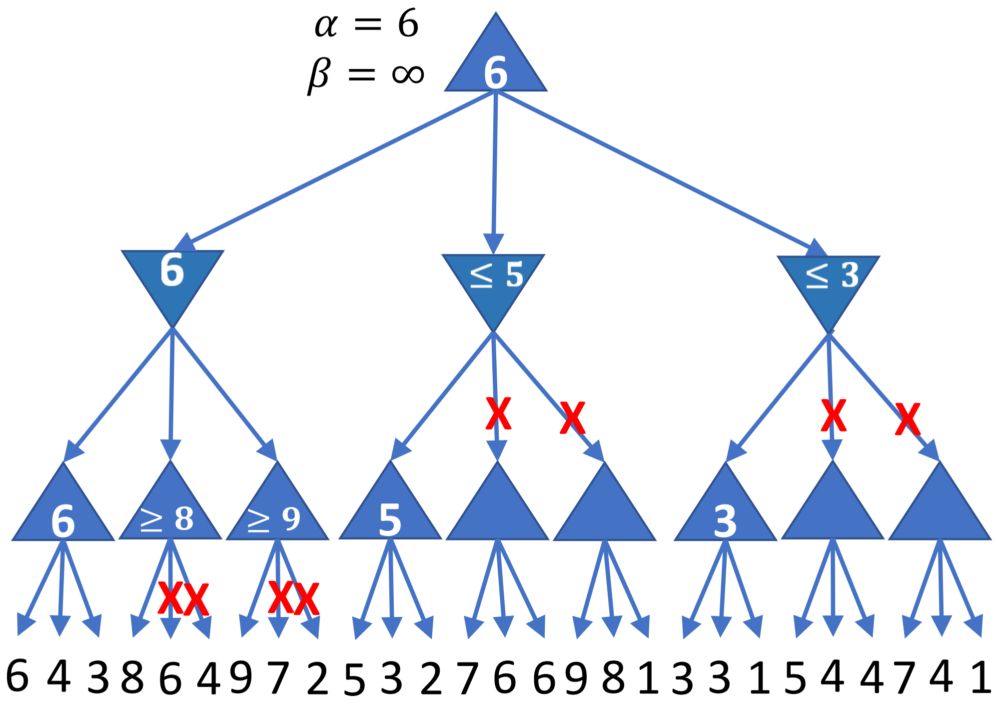
- For each child $s' \in C(s)$:
  - If you realize that $U(s') < \alpha(s)$ then prune all remaining children of s: MAX will never let us reach this node.
  - Otherwise, if $U(s') < \beta(s)$, then set $\beta(s) = U(s')$. MAX might still choose s (because $U(s') \geq \alpha(s)$), then MIN can choose s'.



Optimum node ordering

Imagine you had an oracle, who could tell you which node to evaluate first. Which one should you evaluate first?

- Children of MAX nodes: evaluate the highest-value child first.
- Children of MIN nodes: evaluate the lowest-value child first.



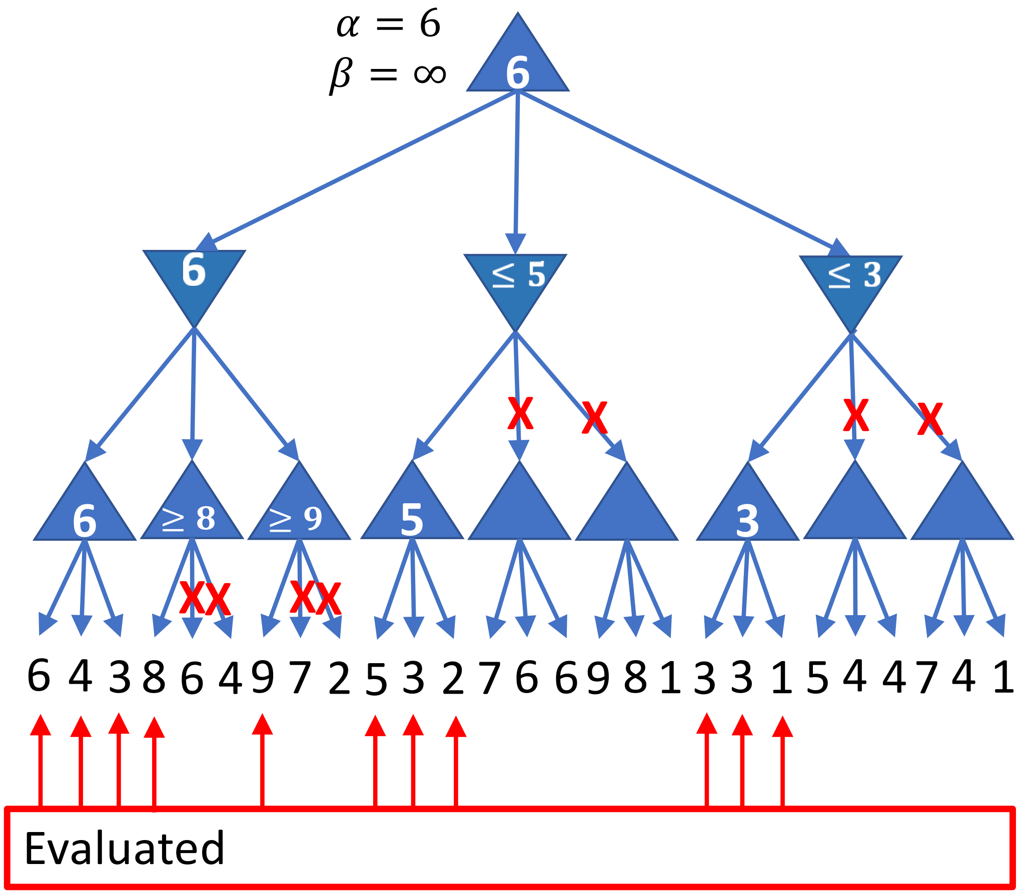
Complexity of alpha-beta

If nodes are optimally ordered, then for each node s, we evaluate

- The b children of its first child.
- The first child of each of its other $b-1$ children.

Total complexity: $2b-1 = O\{b\}$ per ***two*** levels.

- With d levels, total complexity = $(2b-1)^{d/2} = O\{b^{d/2}\}$.

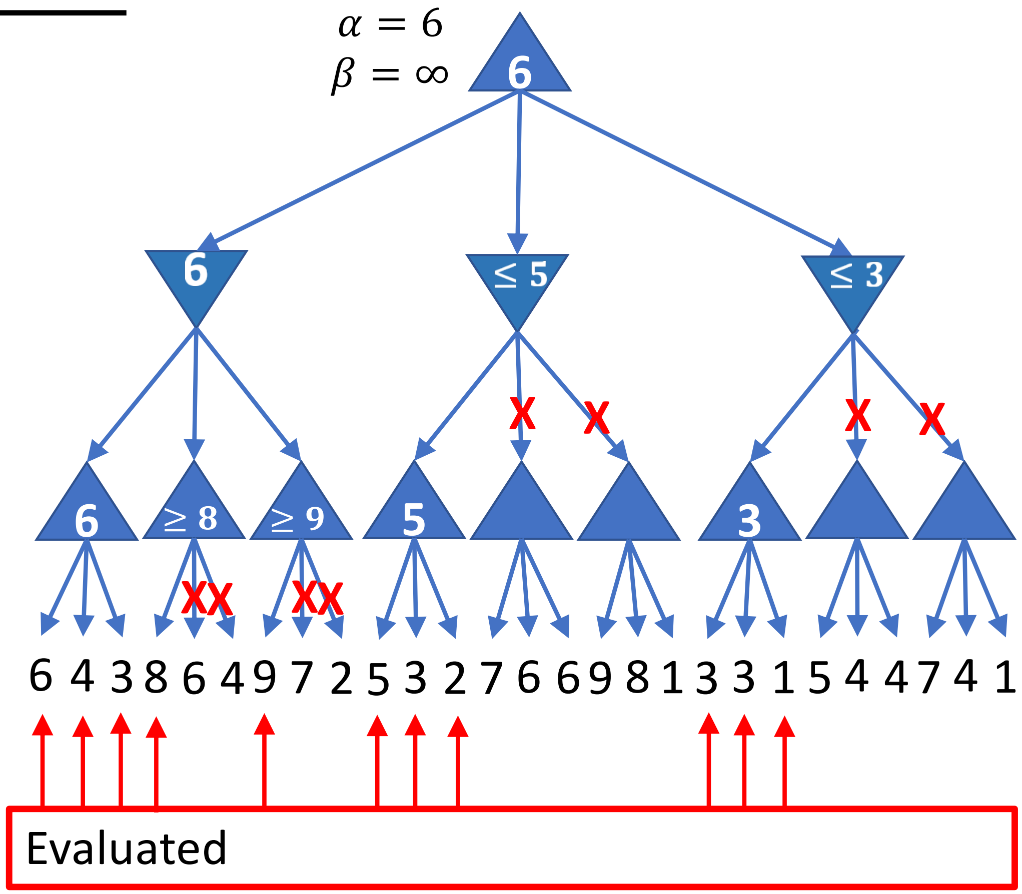


Optimal node ordering???!!!

How on Earth can we decide which child to evaluate first?

- "Children of MAX nodes: evaluate the highest-value child first."

But if we knew which one had the highest value, we wouldn't need to search the tree! We would already know the optimal move!



Outline

- Review: minimax and alpha-beta
- Move ordering: policy network
- Evaluation function: value network
- Training the value network
  - Exact training: endgames
  - Stochastic training: Monte Carlo tree search
- Case study: alphago

Optimal node ordering???!!!

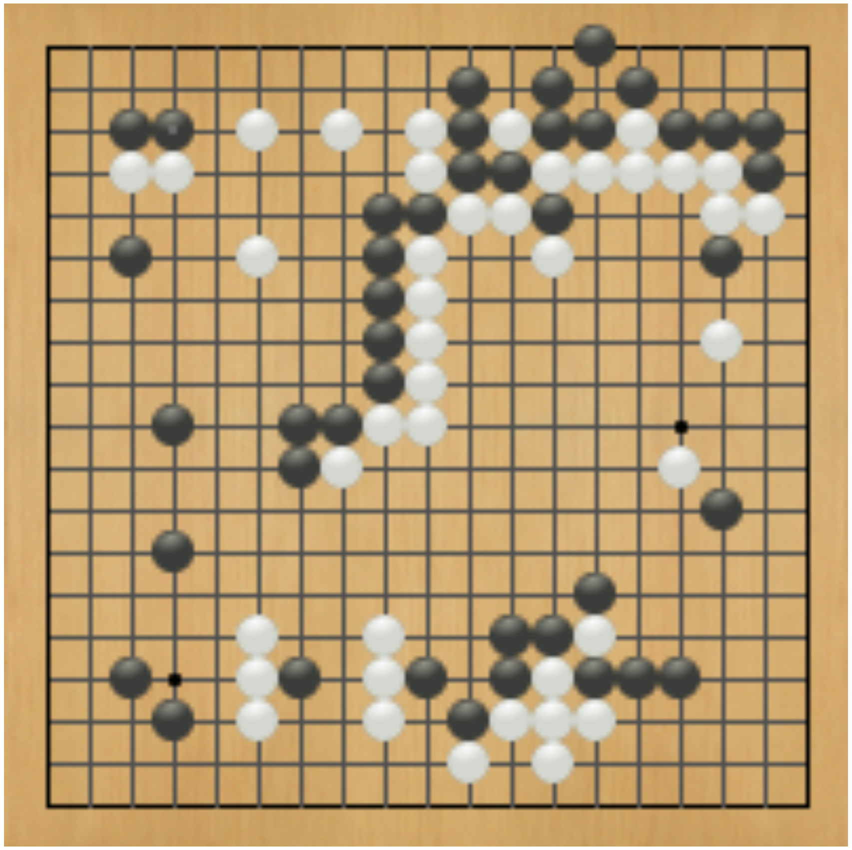
- If we knew which child had the highest value, we wouldn't need to search the tree! We would already know the optimal move!
- Solution: train a policy network, $\pi(s,a)$

Policy networks for two-player games

For example, the game of Go:

- s (state) is a vector of $19 \times 19 = 361$ positions, each of which is $1 =$black (MAX), $-1 =$white (MIN), or $0 =$empty.
- a (action) is the next move = position on the board to place the next stone.
- Neural net estimates $\pi_{MAX}(s, a)$ and $\pi_{MIN}(s, a)$, probability that action a is the best move for MAX/MIN,

$$\pi_{MAX}(s, a) = \frac{e^{f_{MAX}(s,a)}}{\sum_{a'} e^{f_{MAX}(s,a')}}$$

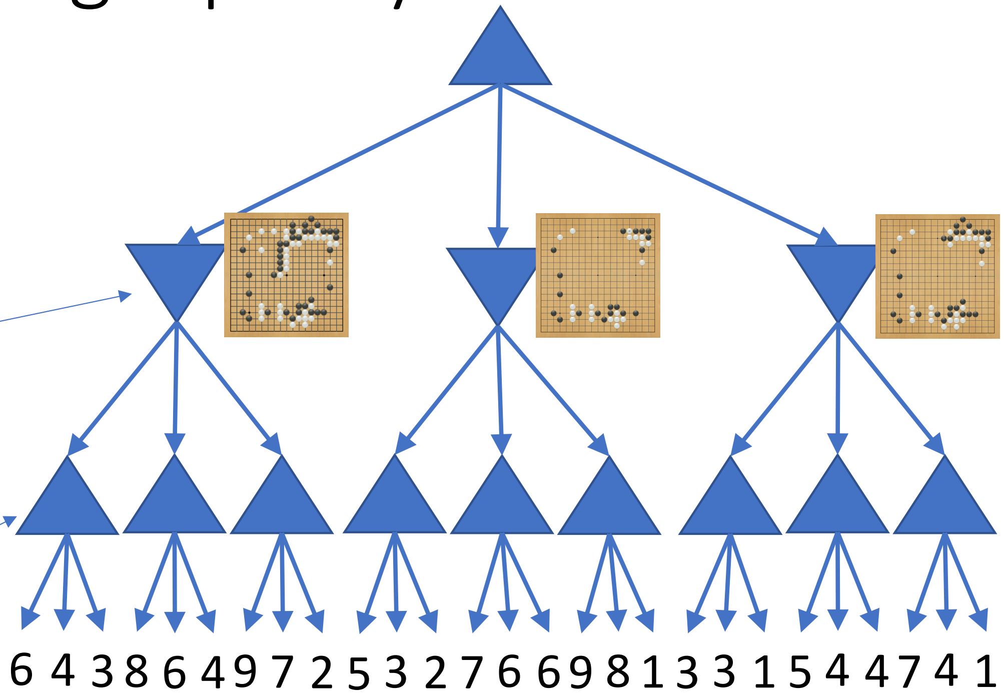


Snapshot of a gnugo game,
http://www.gnu.org/software/gnugo/

Optimal node ordering using a policy network

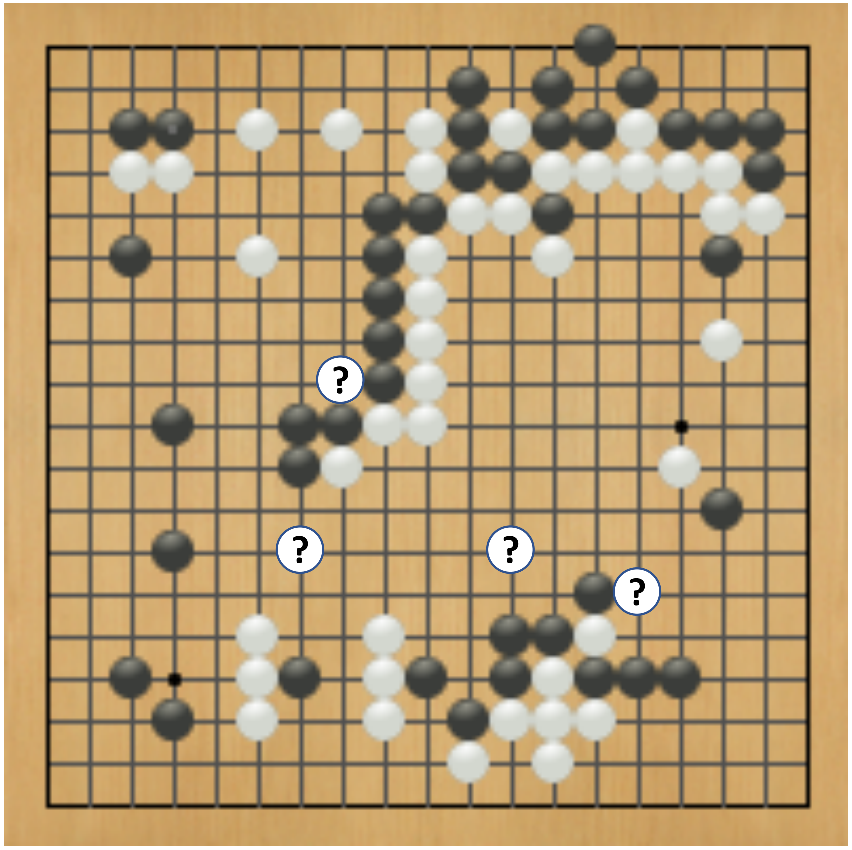
How on Earth can we decide which child to evaluate first?

- Children of MIN nodes: child with highest value of $\pi_{MIN}(s,a)$ (=probability that this node will be evaluated to have the highest value).
- Children of MAX nodes: child with highest value of $\pi_{MAX}(s,a)$ (= probability that this node will be evaluated to have the lowest value).



Hidden advantage: reduce the branching factor

- Policy network can be used to order the moves, as on previous slide.
- Policy network can also be used to reduce the branching factor, from $b = 361$ (the complete branching factor in Go) to $b \approx 4$ or 5. Just choose the 4 or 5 moves with the highest $\pi(s, a)$.
- Russell & Norvig call this "heuristic minimax." It's not guaranteed to work, but it usually works.



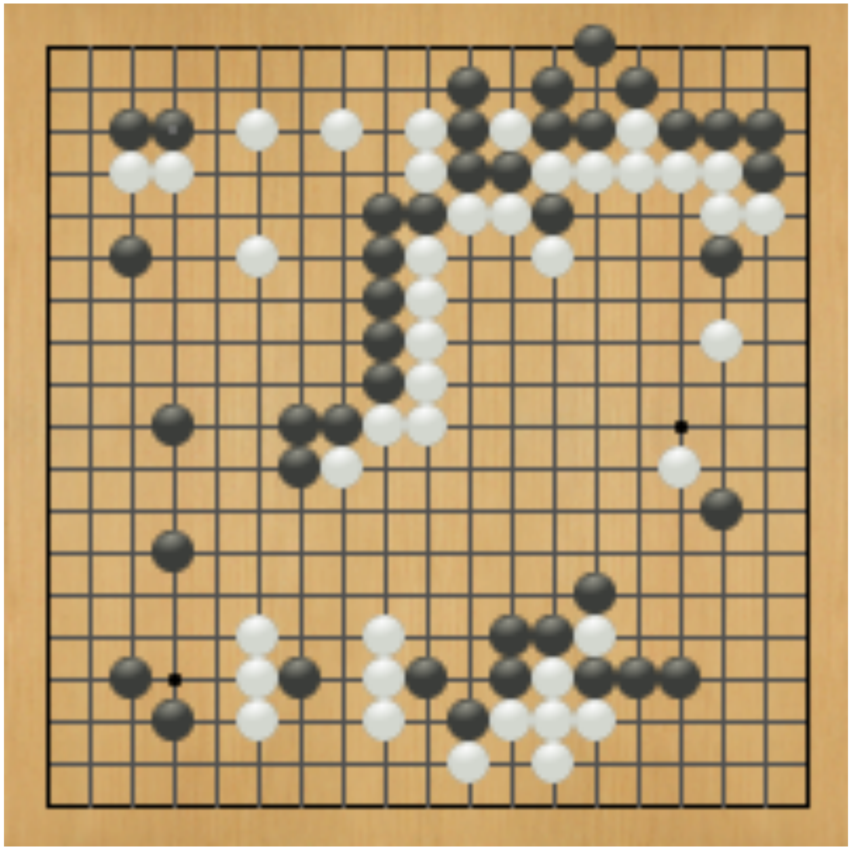
Snapshot of a gnugo game,
http://www.gnu.org/software/gnugo/

Training the policy network

- But how can we train $\pi(s, a)$?
- Answer: Actor-Critic reinforcement learning!

$$U^*(s) = \sum_a \pi_{MAX}(s, a) Q^*(s, a)$$

- Train $Q^*(s, a)$ using deep Q-learning (play the game many times, gain reward each time you win)
- Train $\pi_{MAX}(s, a)$ to maximize $U^*(s)$
- Train $\pi_{MIN}(s, a)$ to minimize $U^*(s)$.



Snapshot of a gnugo game, http://www.gnu.org/software/gnugo/

Outline

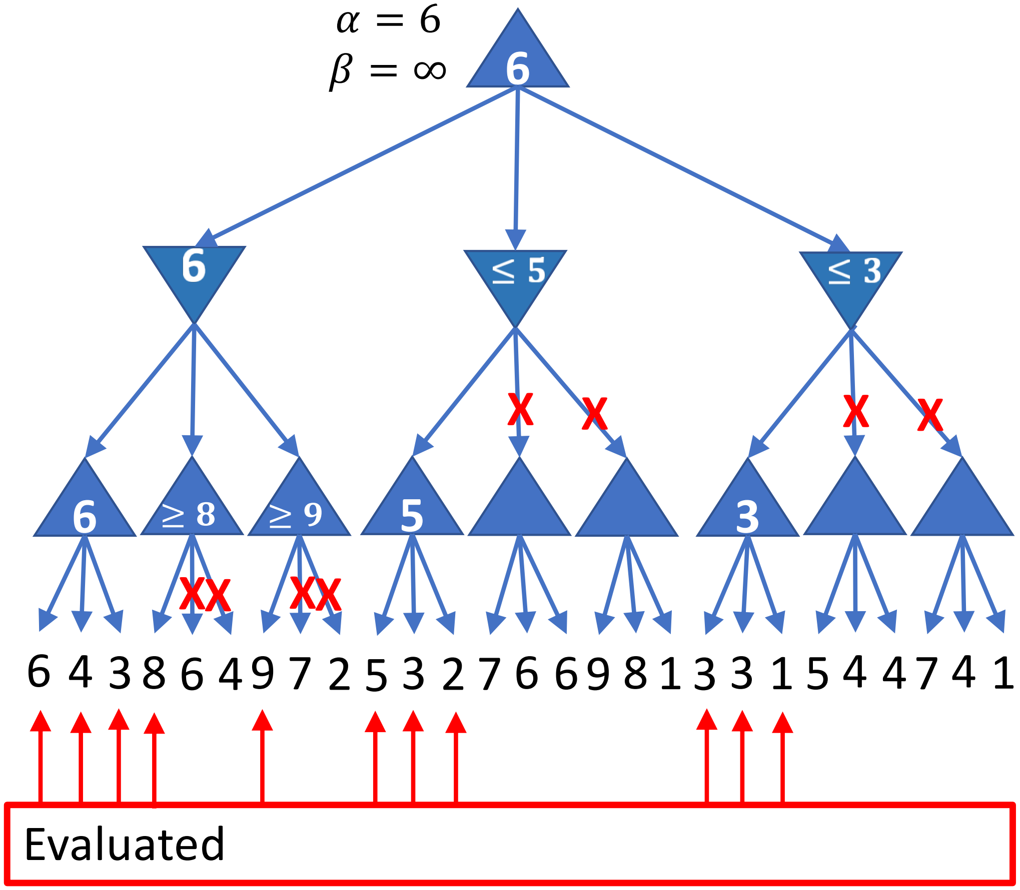
- Review: minimax and alpha-beta
- Move ordering: policy network
- Evaluation function: value network
- Training the value network
  - Exact training: endgames
  - Stochastic training: Monte Carlo tree search
- Case study: alphago

Complexity of alpha-beta

If nodes are optimally ordered, then, with d levels, total complexity = $(2b-1)^{d/2} = O\{b^{d/2}\}$.

…but wait…

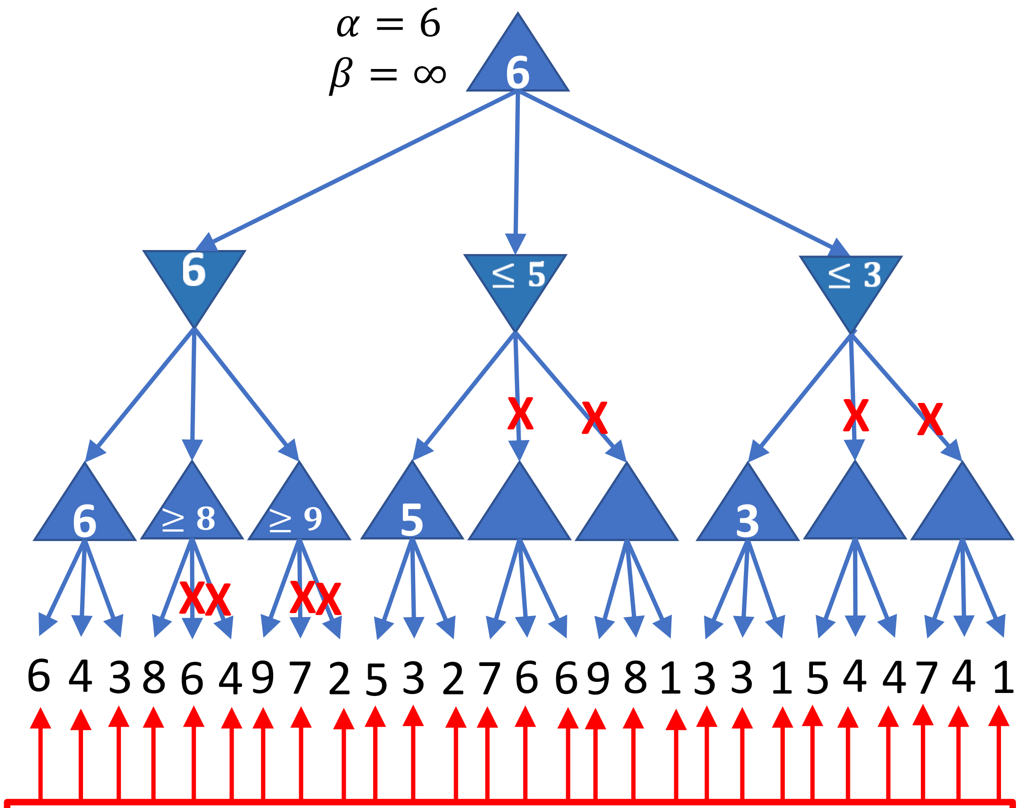
A game of Go has up to 361 moves, each of which takes any of the available 361 points. $O\{361^{361/2}\}$ is very large…



Limited-horizon game search

Instead of searching to the end of the game, we choose a depth (d) that’s within our computational resources.

Then, at depth d, call the value network $U^*(s)$ to estimate the probability that MAX wins from that position.



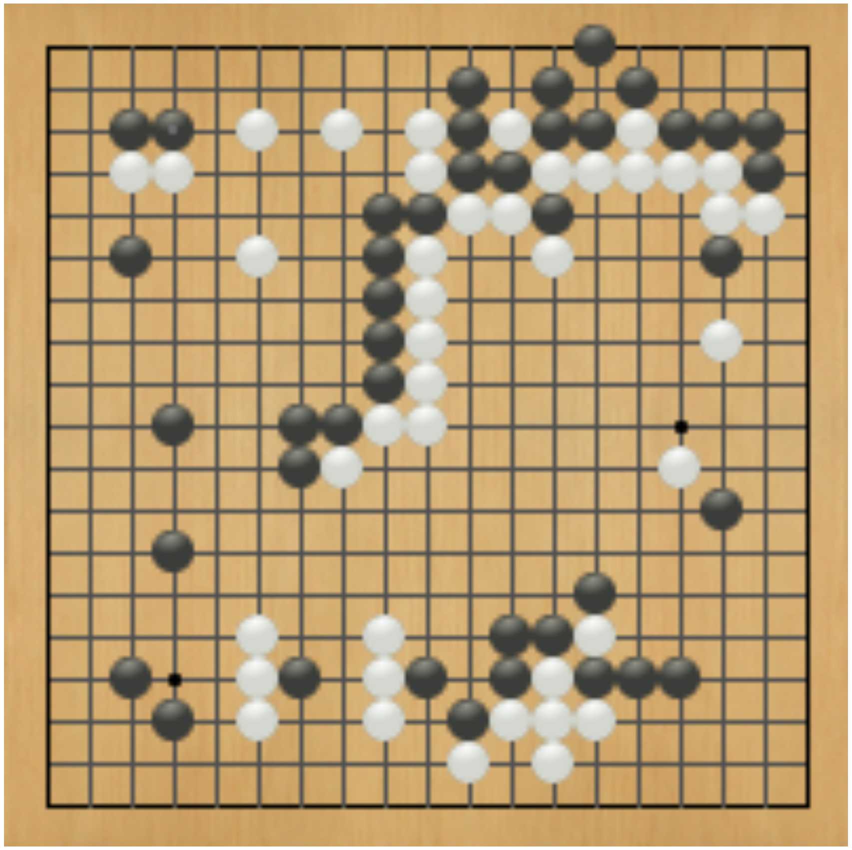
These are not the end of the game! These are actually the outputs of the value network, $U^*(s)$, at these game positions.

Training the value network

- But how can we train $U^*(s)$?
- Answer: Actor-Critic reinforcement learning!

$$U^*(s) = \sum_a \pi_{MAX}(s,a) Q^*(s,a)$$

- Train $Q^*(s,a)$ using deep Q-learning (play the game many times, gain reward each time you win)
- Train $\pi_{MAX}(s,a)$ to maximize $U^*(s)$
- Train $\pi_{MIN}(s,a)$ to minimize $U^*(s)$.



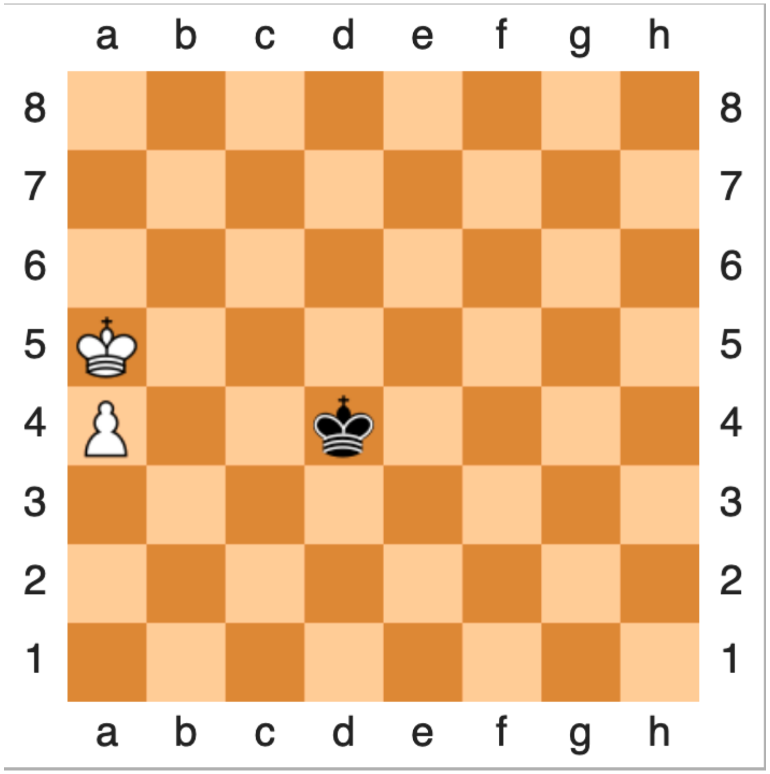
Snapshot of a gnugo game, http://www.gnu.org/software/gnugo/

Outline

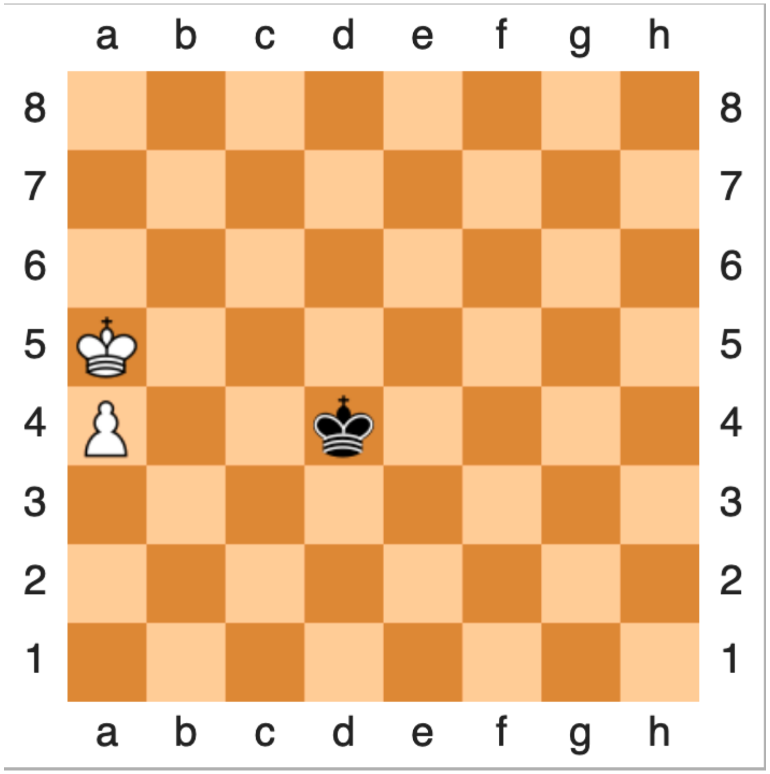
- Review: minimax and alpha-beta
- Move ordering: policy network
- Evaluation function: value network
- Training the value network
  - Exact training: endgames
  - Stochastic training: Monte Carlo tree search
- Case study: alphago

Endgames

- $U^*(s)$ can be exact when the game is near its end. This situation is called "endgame."
- For example, in chess, if there are only three pieces left, then there are just under $64^3 = 2^{18}$ possible board positions. With two bytes to encode the value of each, that's half a megabyte.
- Thus we can bypass the neural net, in favor of a lookup table.



How to create an endgame table



- Of the 2^{18} possible board positions, find all the terminal states (white checkmate, black checkmate, or draw).
- Iterate minimax *backward* from the set of terminal states until you know the result for each of the 2^{18} board positions.
- Computation is limited not by the search depth, but by the limited number of board positions in the table.

Outline

- Review: minimax and alpha-beta
- Move ordering: policy network
- Evaluation function: value network
- Training the value network
  - Exact training: endgames
  - Stochastic training: Monte Carlo tree search
- Case study: alphago

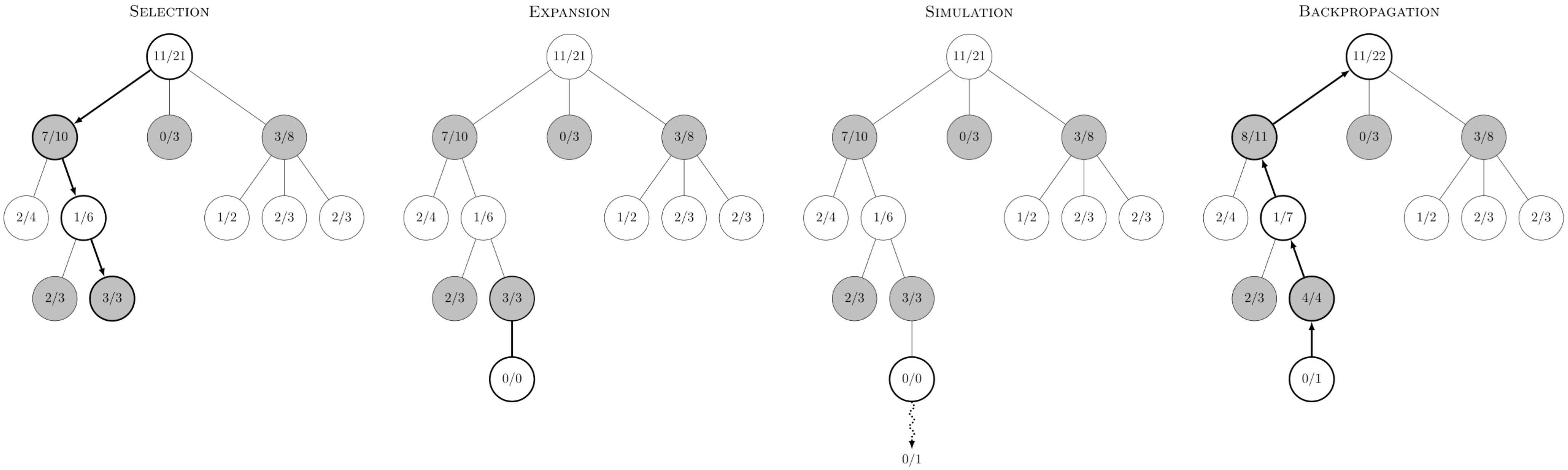
Monte Carlo Tree Search

Suppose s is too complicated for an endgame search. We still need to estimate its value and policy. How?

- **Selection:** Run minimax forward a few steps, then use **value network** to estimate values of the nodes at the end of the tree. Select one of those nodes (call it s).
- **Expansion**: Minimax one step further using action a.
- **Simulation**: Play a random game, starting with node (s, a). At each step, choose a move at random from the current **policy network**.
- **Backpropagate**: Set $Q_{local}(s, a)$ equal to the average win frequency of the random games starting from (s, a).

After your training dataset gets large enough, re-train $Q^*(s, a)$ with $Q_{local}(s, a)$ as its target.

Monte Carlo Tree Search



Steps in Monte Carlo Tree Search.
By Rmoss92 - Own work, CC BY-SA 4.0,
https://commons.wikimedia.org/w/index.php?curid=88889583

Exploration vs. Exploitation

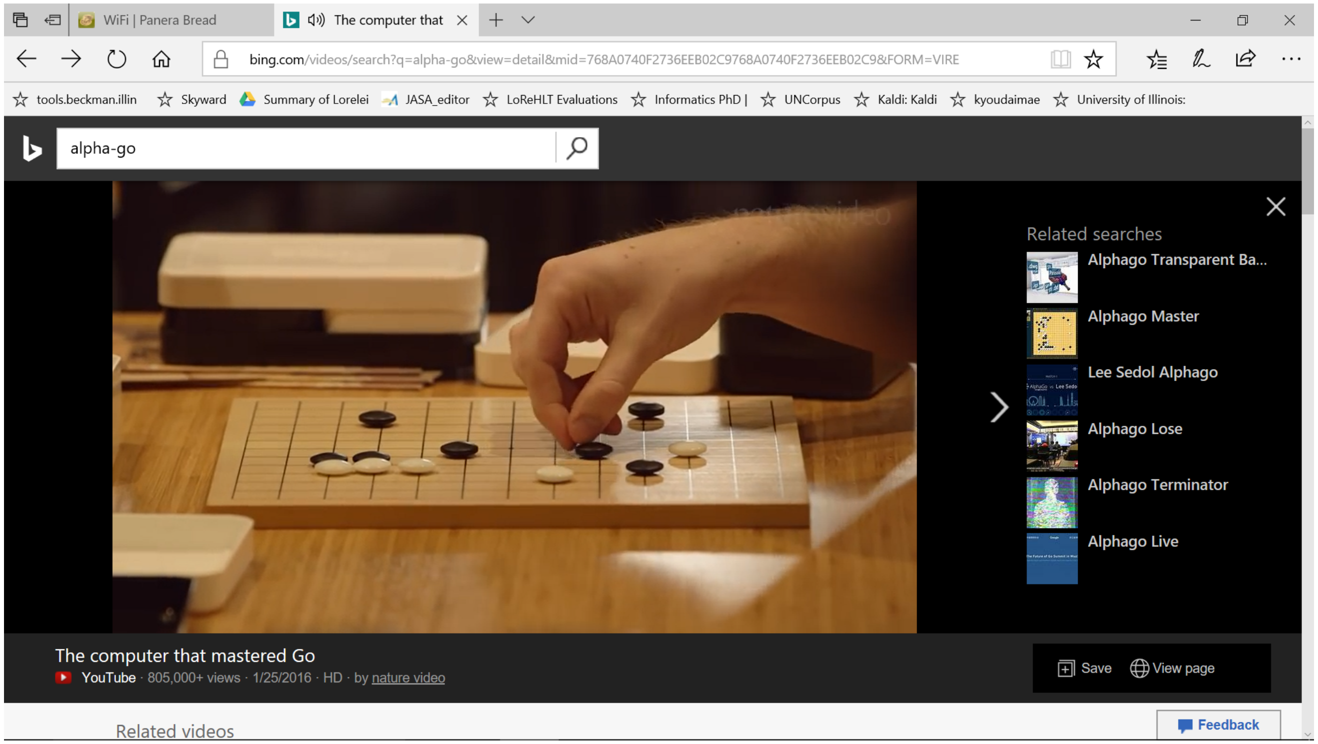
- In order to gain information about the win probability of node (s,a), you need to put some randomness into the game.
- Exploration strategies from reinforcement learning, like epsilon-greedy, work well.
- AlphaGo used this strategy:
  - From a large database of human-vs-human games, train the initial “supervised learning” policy network, $\pi_{SL}(s, a)$.
  - From the same database, train another policy network that’s the same, but with too few trainable parameters, hence less accurate.   Call this the “rollout network,” $\pi_{Rollout}(s, a)$.
  - Use $\pi_{Rollout}(s, a)$ to play games – its low accuracy adds randomness -- use its results to improve $\pi_{SL}(s, a)$.

Outline

- Review: minimax and alpha-beta
- Move ordering: policy network
- Evaluation function: value network
- Training the value network
  - Exact training: endgames
  - Stochastic training: Monte Carlo tree search
- Case study: Alpha-Go

Alpha-Go Video by Nature Magazine

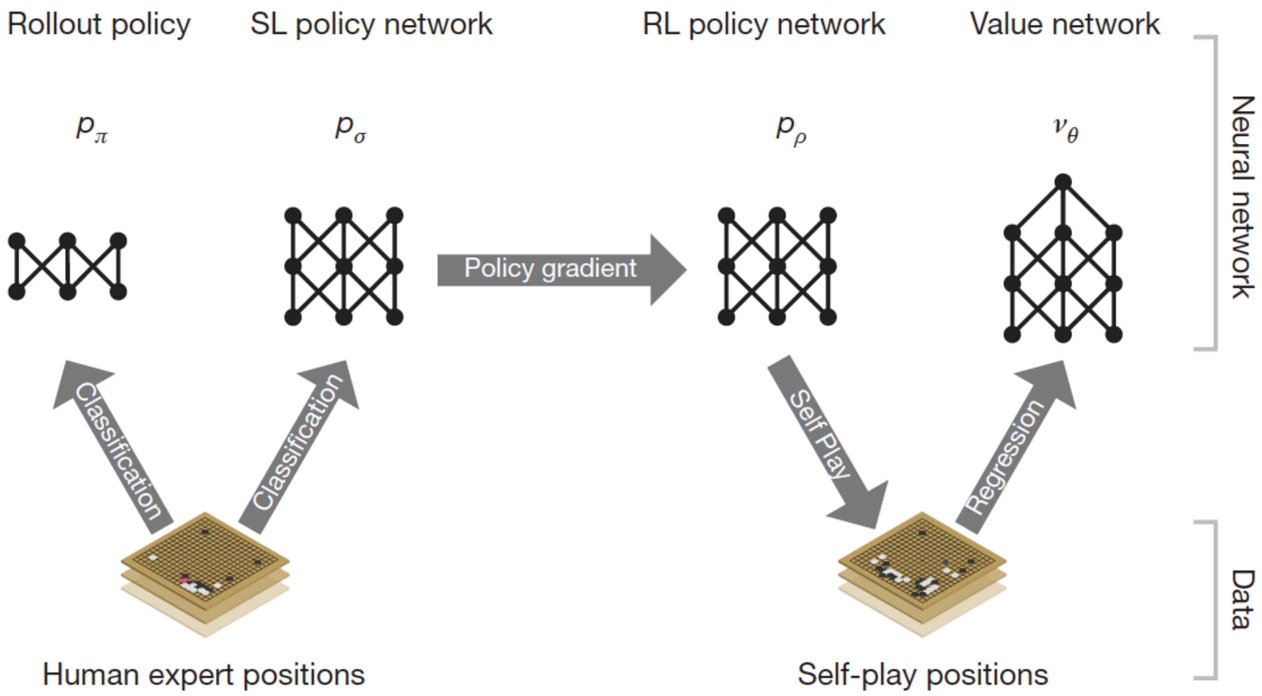
(8 minutes, 2016)



The computer that mastered Go

YouTube · 805,000+ views · 1/25/2016 · HD · by nature video

AlphaGo



D. Silver et al., [Mastering the Game of Go with Deep Neural Networks and Tree Search](), Nature 529, January 2016

Conclusions

- Review: minimax and alpha-beta
  - Complexity: $(2b-1)^{d/2} = O\{b^{d/2}\}$ with depth d and branching factor b, if the children of each node are ordered just right (MAX: largest first, MIN: smallest first)
- Move ordering: policy network
  - Can be used to order the children, with no loss of accuracy; Can also limit the set of moves evaluated, with some loss of accuracy
- Evaluation function: value network
  - Estimates the value of each board position in limited-horizon search
- Exact value: endgames
  - Minimax search backward from a set of known terminal positions
- Stochastic training: Monte Carlo tree search
  - Choose a policy that includes exploration vs. exploitation, play games at random, use the data to estimate win frequency