Artificial Intelligence

5. Adversarial Search

What To Do When Your "Solution" is Somebody Else's Failure

Jörg Hoffmann, Daniel Fiser, Daniel Höller, Sophia Saller

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Summer Term 2022

Agenda

The Problem

→ "Adversarial search" = Game playing against an opponent.

Why AI Game Playing?

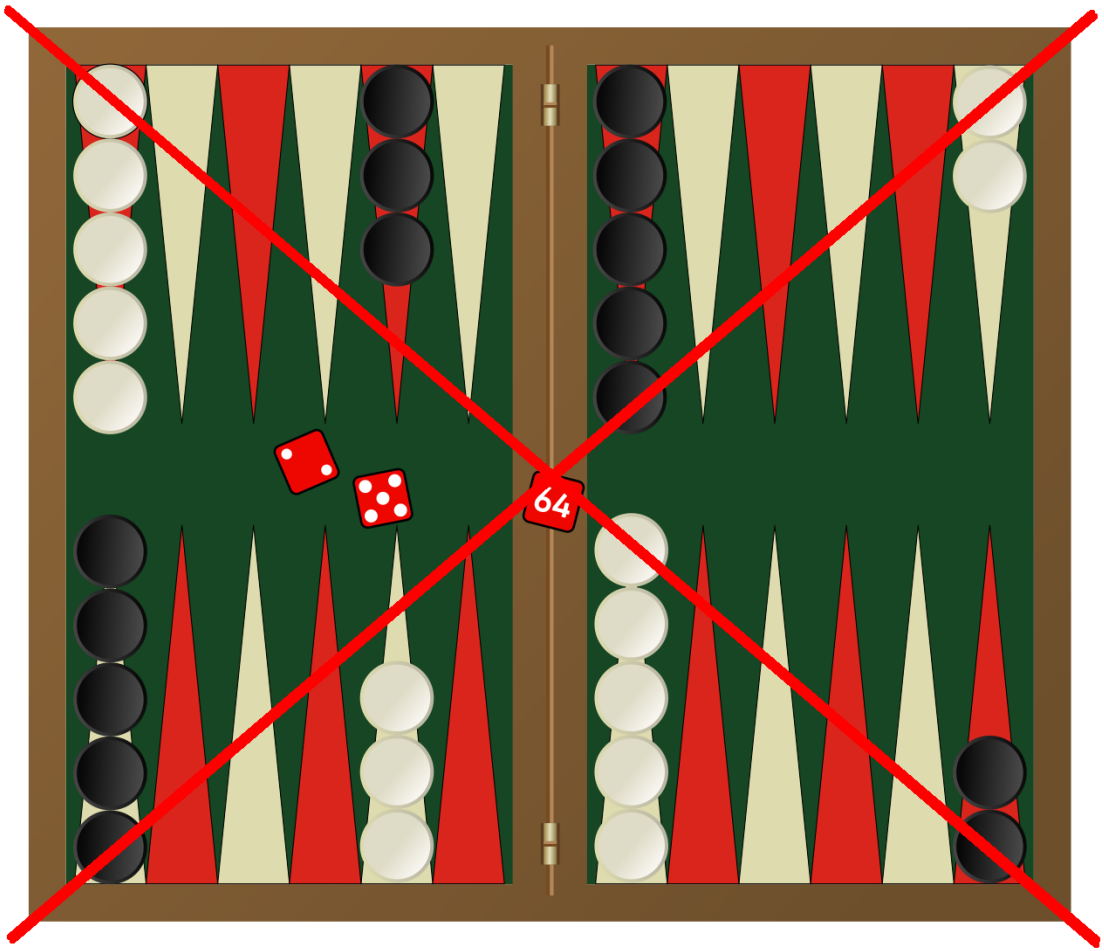
Many good reasons:

- Playing a game well clearly requires a form of "intelligence".
- Games capture a pure form of competition between opponents.
- Games are abstract and precisely defined, thus very easy to formalize.

→ Game playing is one of the oldest sub-areas of AI (ca. 1950).

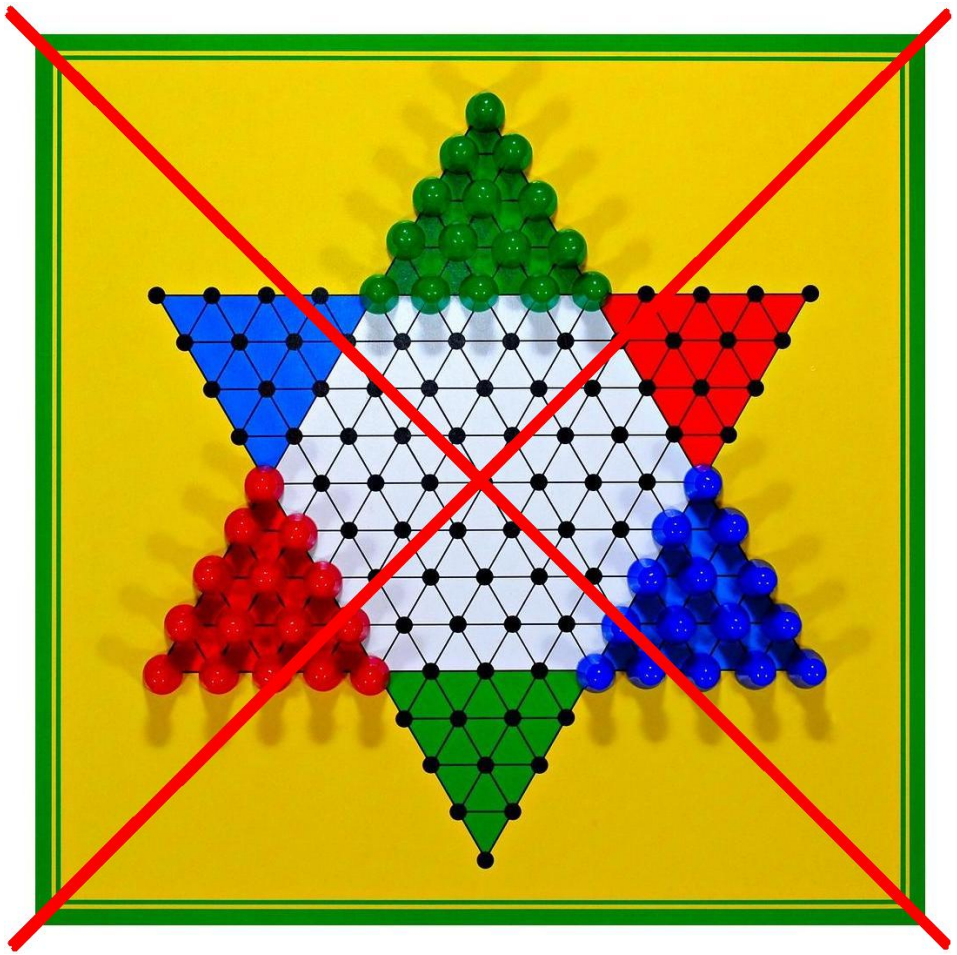
→ The dream of a machine that plays Chess is, indeed, *much* older than AI! (von Kempelen's "Schachtürke" (1769), Torres y Quevedo's "El Ajedrecista" (1912))

Which Games?



→ No chance element.

Which Games?



→ Exactly two players.

Which Games?

→ Game state fully observable.

Which Games?

→ Player utilities are diametrically opposed.
(Else: game theory, equilibria, auctions, . . .)

These Games!

Restrictions:

- The game state is fully observable.
- The outcome of each move is deterministic.
- Game states discrete, finite number of possible moves and game states.
- There are no infinite runs of the game: a terminal state is always reached after a finite number of steps.
- Two-player zero-sum game: two players, terminal states have utility with utility(player1) $=$ $-$utility(player2).
- Our formulation (equivalent): single utility function u, players Max vs. Min trying to maximize vs. minimize u.
- Turn-taking: Players move alternatingly. Max begins.

An Example Game

- Game states: Positions of figures.
- Moves: Given by rules.
- Players: White (Max), Black (Min).
- Terminal states: Checkmate.
- Utility function: $+100$ if Black is checkmated, 0 if stalemate, -100 if White is checkmated.

(A Brief Note On) Formalization

Definition (Game State Space). *A game state space is a 6-tuple* $\Theta = (S, A, T, I, S^T, u)$ *where:*

- S, A, T, I: *States, actions, deterministic transition relation, initial state. As in classical search problems, except:*
  - S *is the disjoint union of* S^{Max}, S^{Min}, *and* S^T.
  - A *is the disjoint union of* A^{Max} *and* A^{Min}.
  - *For* $a \in A^{Max}$, *if* $s \xrightarrow{a} s'$ *then* $s \in S^{Max}$ *and* $s' \in S^{Min} \cup S^T$.
  - *For* $a \in A^{Min}$, *if* $s \xrightarrow{a} s'$ *then* $s \in S^{Min}$ *and* $s' \in S^{Max} \cup S^T$.
- S^T *is the set of terminal states.*
- $u : S^T \mapsto \mathbb{R}$ *is the utility function.*

Commonly used terminology: state=position, terminal state=end state, action=move.

(A round of the game – one move Max, one move Min – is often referred to as a "move", and individual actions as "half-moves". We do *NOT* do that here.)

Why Games are Hard to Solve

Why Games are hard to solve, part 1: $\rightarrow$ What is a "solution" here?

Definition (Policy). *Let Θ be a game state space, and let $X \in \{Max, Min\}$. A policy for X is a function $p^X : S^X \mapsto A^X$ so that a is applicable to s whenever $p^X(s) = a$.*

- We don't know how the opponent will react, and need to prepare for all possibilities.
- A policy is optimal if it yields the best possible utility for X assuming perfect opponent play (not formalized here).
- In (almost) all games, computing a policy is infeasible. Instead, compute the next move "on demand", given the current game state.

Why Games are hard to solve, part 2:

- Number of reachable states: in Chess 10^{40}; in Go 10^{100}.
- Chess: branching factor ca. 35, hence 1000 per move/counter-move lookahead. Go: 200, hence 40000.

Our Agenda for This Chapter

- **Minimax Search:** How to compute an optimal policy?

  → Minimax is the canonical (and easiest to understand) algorithm for *solving* games, i.e., computing an optimal policy.

- **Alpha-Beta Search:** How to prune unnecessary parts of the tree?

  → An essential improvement over Minimax.

- **Evaluation Functions:** How to evaluate a game position?

  → Heuristic functions for games, and how to obtain them.

- **AlphaGo/Zero:** How does it work?

  → Overview of the AlphaGo/Zero systems architecture.

Questionnaire

Question!

When was the first game-playing computer built?

(A): 1941 (B): 1950

(C): 1958 (D): 1965

→ In 1941, a small box beat humans at Nim (take away objects from heaps, player taking the last object looses).

Question!

Does the video game industry attempt to make the computer opponents as intelligent as possible?

(A): Yes (B): No

→ In some cases, yes (I guess). In general, no. For example, in Ego-Shooter games, if your computer opponents did the best they can, you'd be shot immediately and always.

"Minimax"?

$\rightarrow$ We want to compute an optimal move for player "Max". In other words: "We are Max, and our opponent is Min."

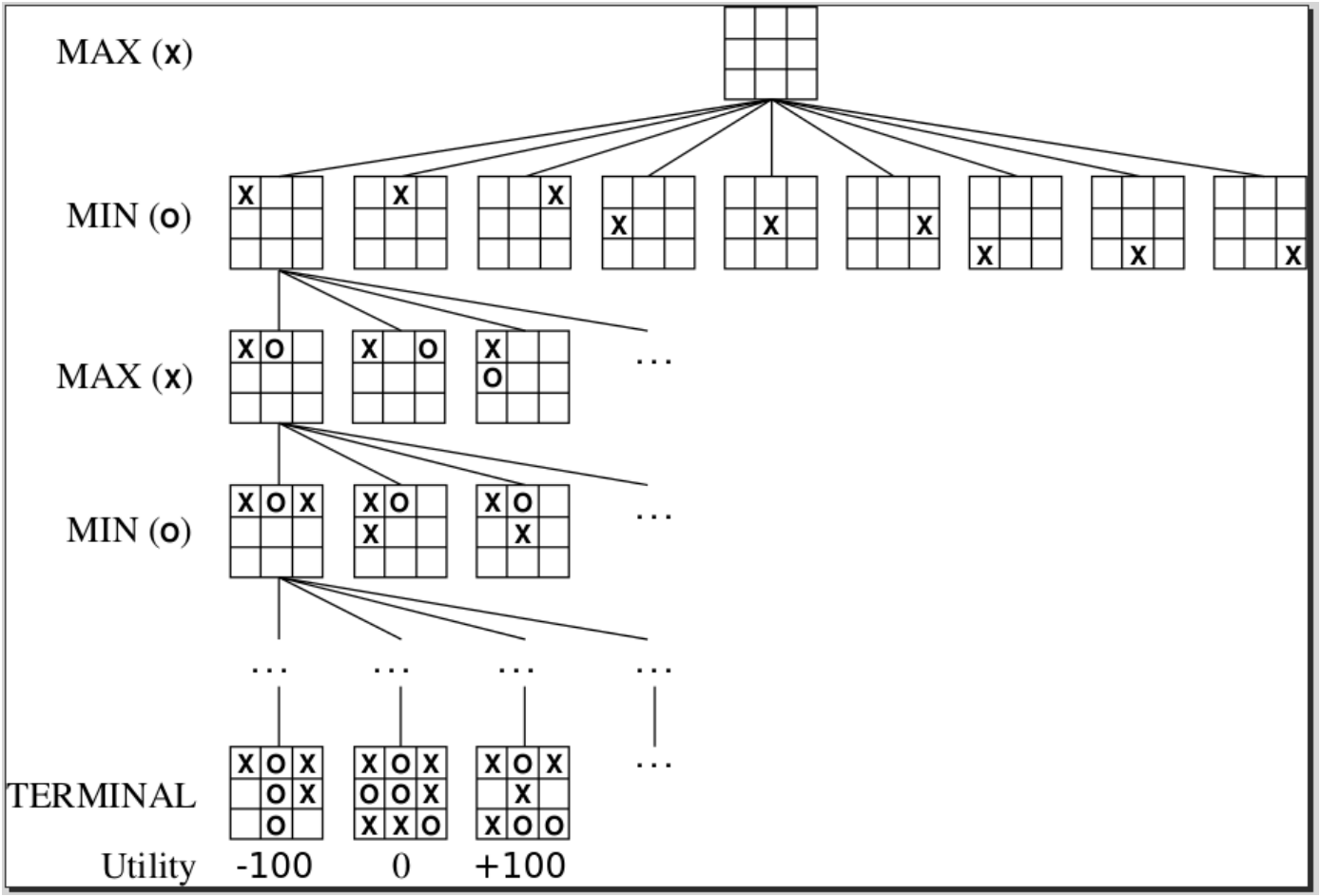
Remember:

- Max attempts to *maximize* the utility $u(s)$ of the terminal state that will be reached during play.
- Min attempts to *minimize* $u(s)$.

So what?

- The computation alternates between minimization and maximization $\Longrightarrow$ hence "Minimax".

Example Tic-Tac-Toe



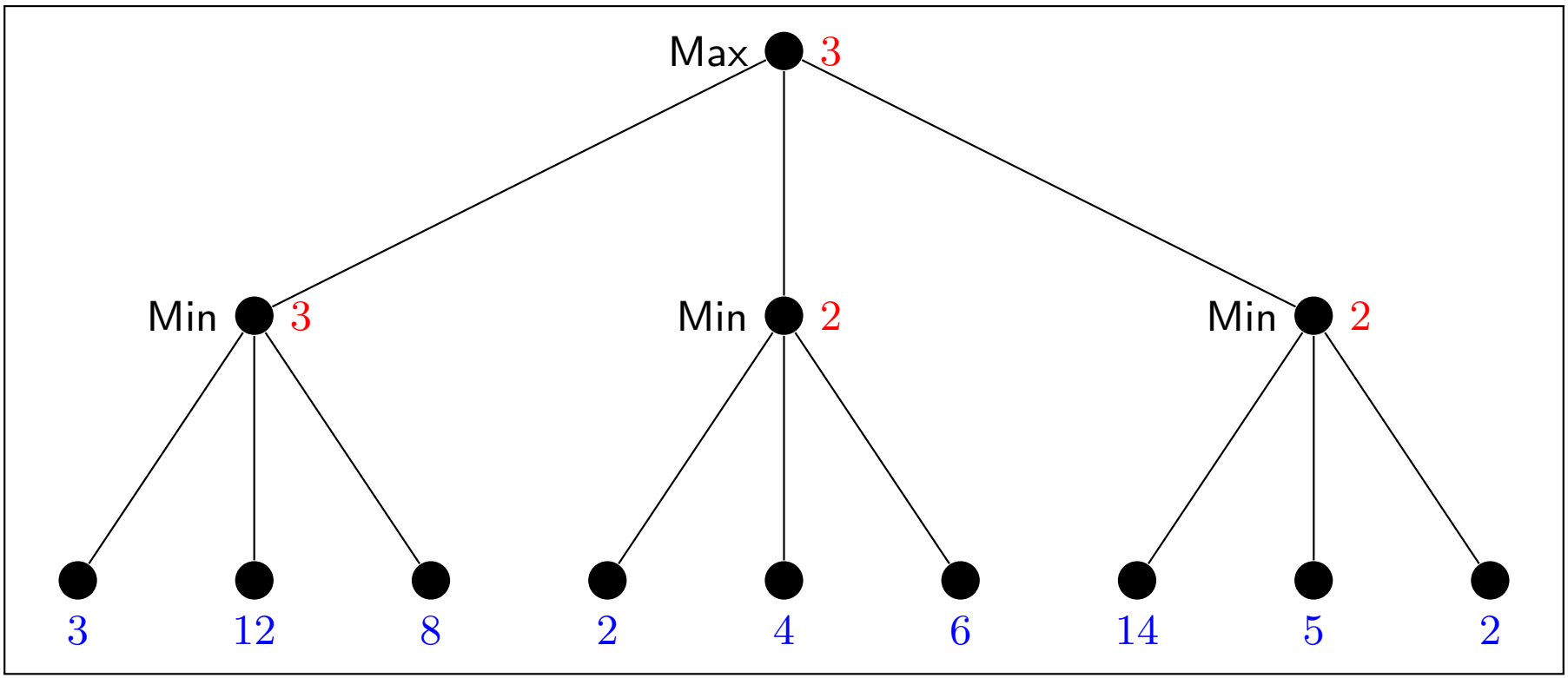
- Game tree, current player marked on the left.
- Last row: terminal positions with their utility.

Minimax: Outline

We max, we min, we max, we min . . .

1. Depth-first search in game tree, with Max in the root.
2. Apply utility function to terminal positions.
3. Bottom-up for each inner node n in the tree, compute the utility $u(n)$ of n as follows:
   - If it's Max's turn: Set $u(n)$ to the maximum of the utilities of n's successor nodes.
   - If it's Min's turn: Set $u(n)$ to the minimum of the utilities of n's successor nodes.
4. Selecting a move for Max at the root: Choose one move that leads to a successor node with maximal utility.

Minimax: Example



- Blue numbers: Utility function u applied to terminal positions.
- Red numbers: Utilities of inner nodes, as computed by Minimax.

Minimax: Pseudo-Code

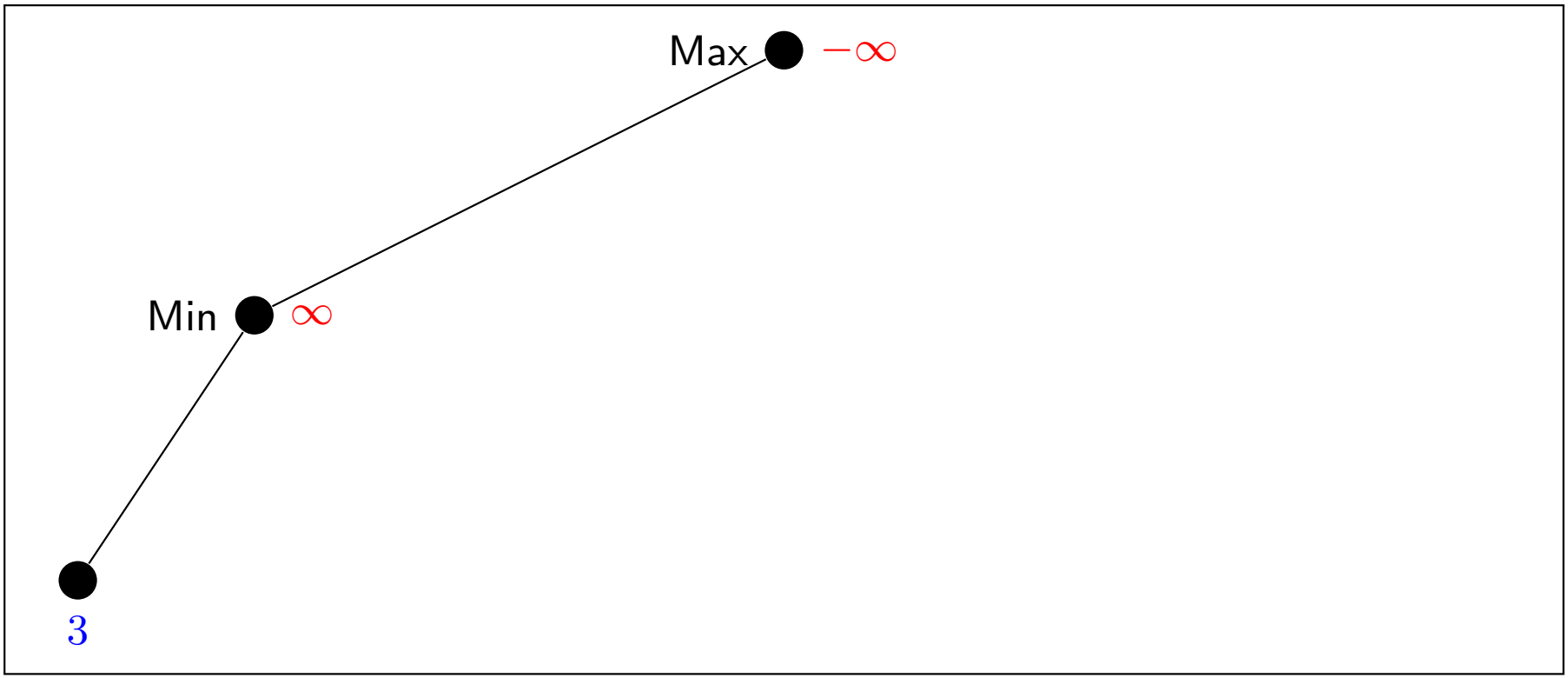
Input: State $s \in S^{Max}$, in which Max is to move.

function Minimax-Decision(s) **returns** an action
  $v \leftarrow$ *Max-Value*(s)
  return an *action* $a \in$ *Actions(*s*)* yielding value v

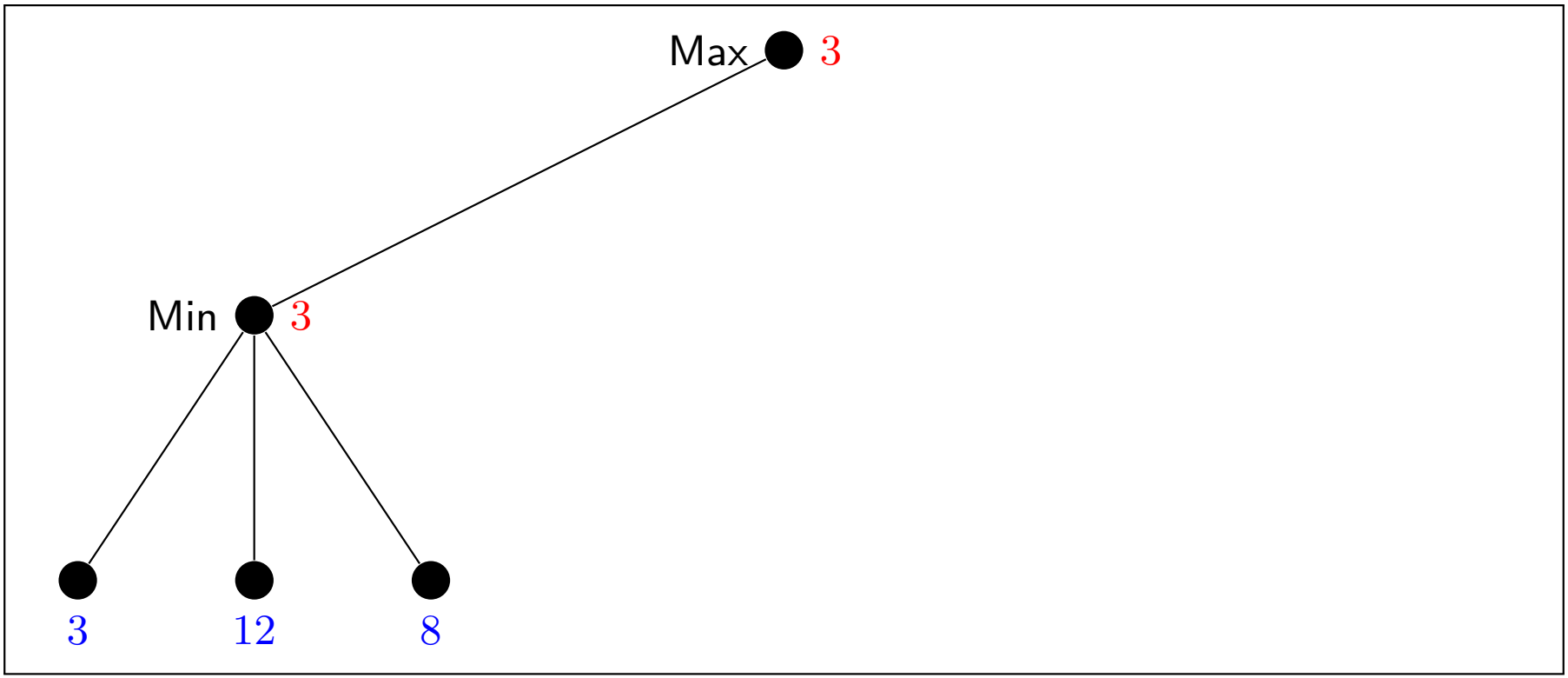
function Max-Value(s) **returns** a utility value
  if *Terminal-Test*(s) **then return** $u(s)$
  $v \leftarrow -\infty$
  for each $a \in$ *Actions(*s*)* **do**
    $v \leftarrow \max(v,$*Min-Value(ChildState*$(s,a)))$
  return v

function Min-Value(s) **returns** a utility value
  if *Terminal-Test*(s) **then return** $u(s)$
  $v \leftarrow +\infty$
  for each $a \in$ *Actions(*s*)* **do**
    $v \leftarrow \min(v,$*Max-Value(ChildState*$(s,a)))$
  return v

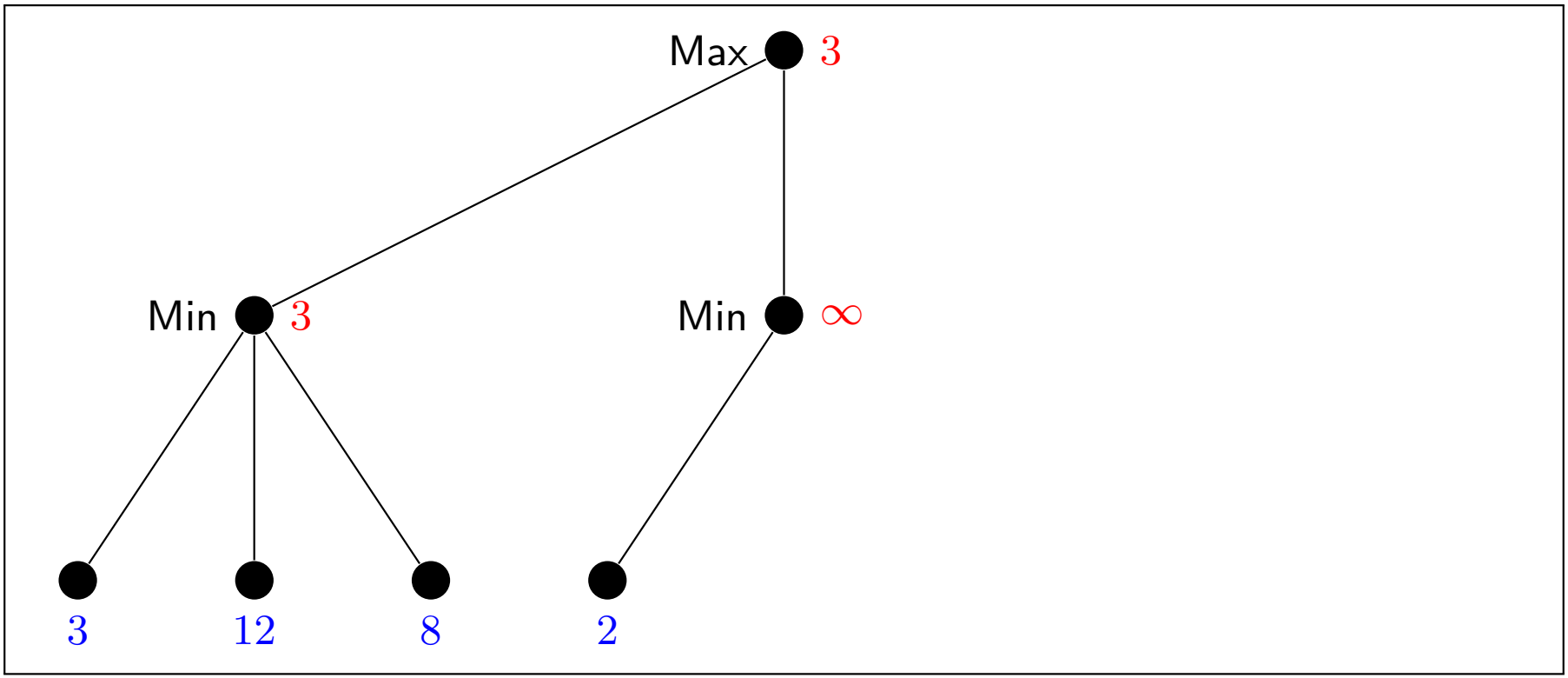
Minimax: Example, Now in Detail



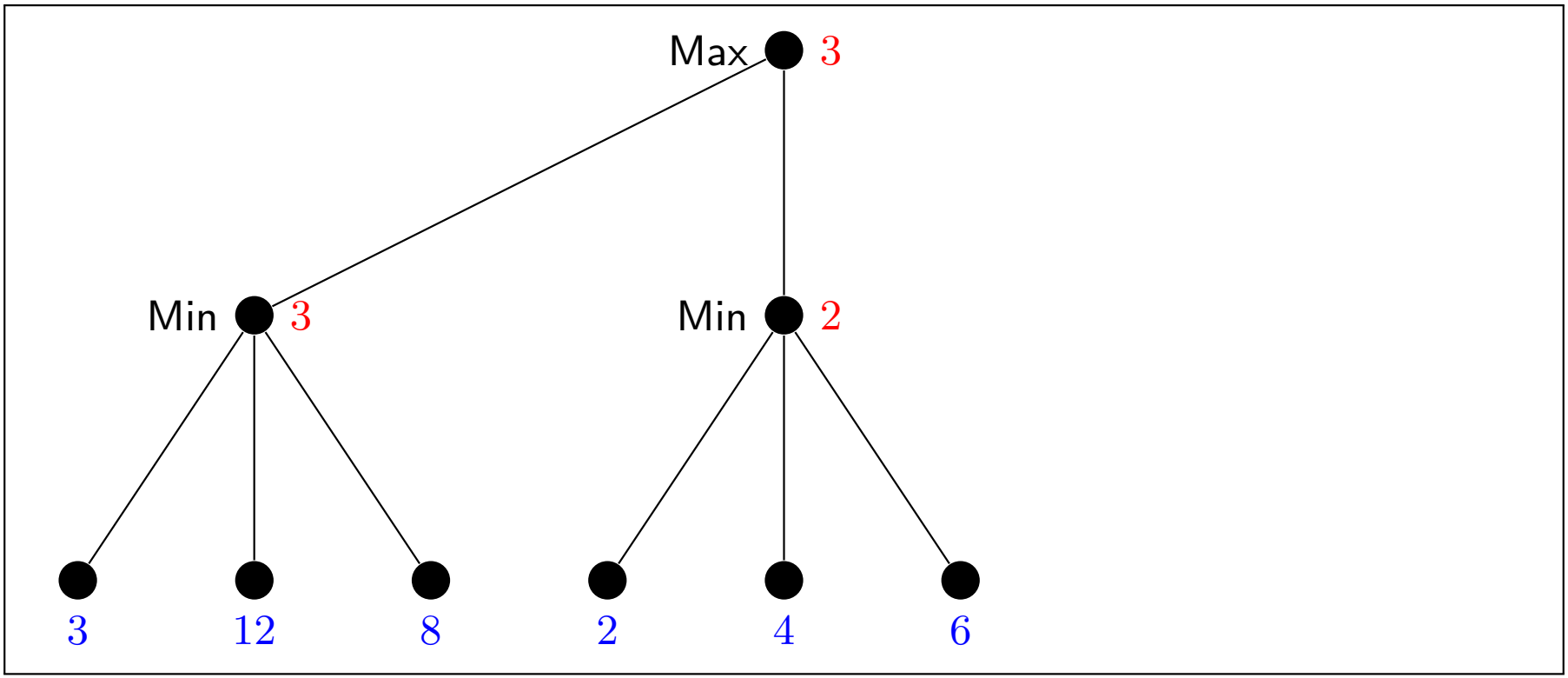
Minimax: Example, Now in Detail



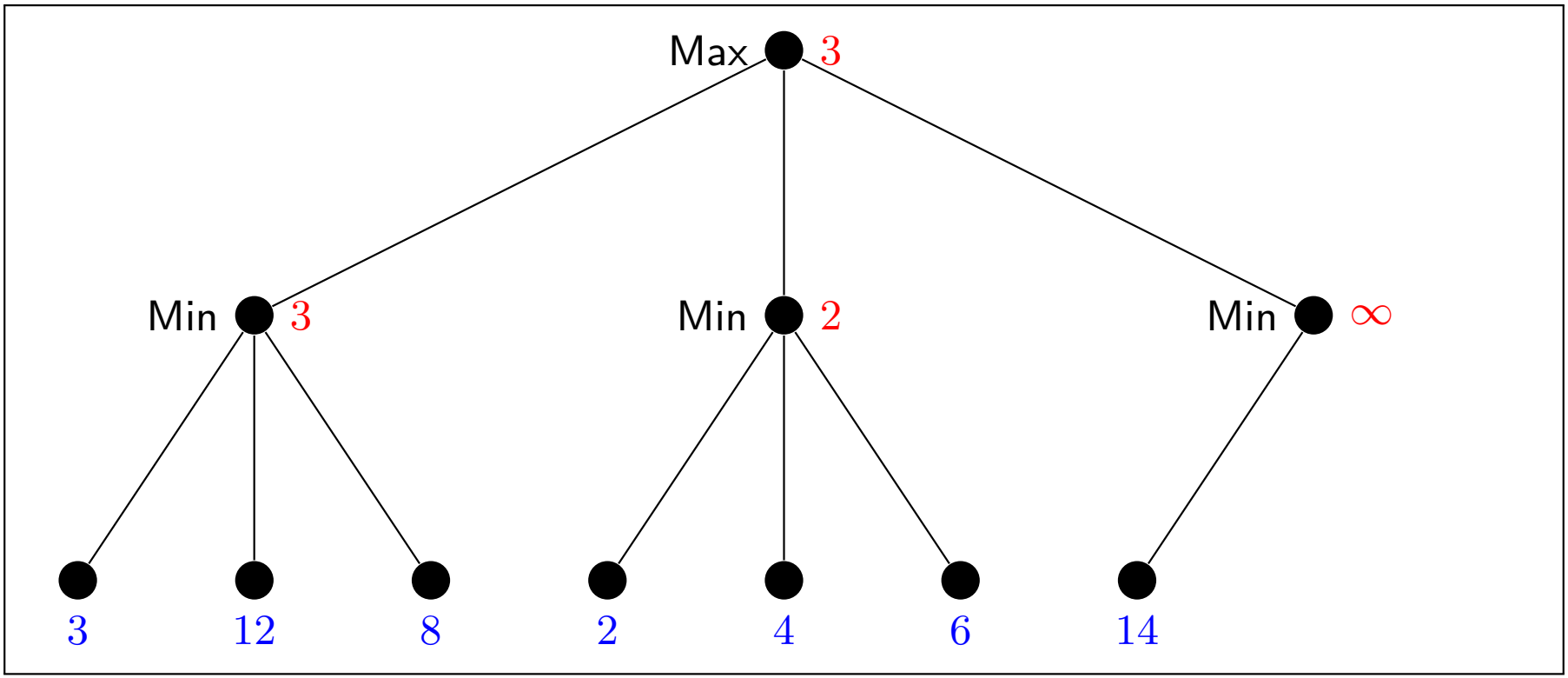
Minimax: Example, Now in Detail



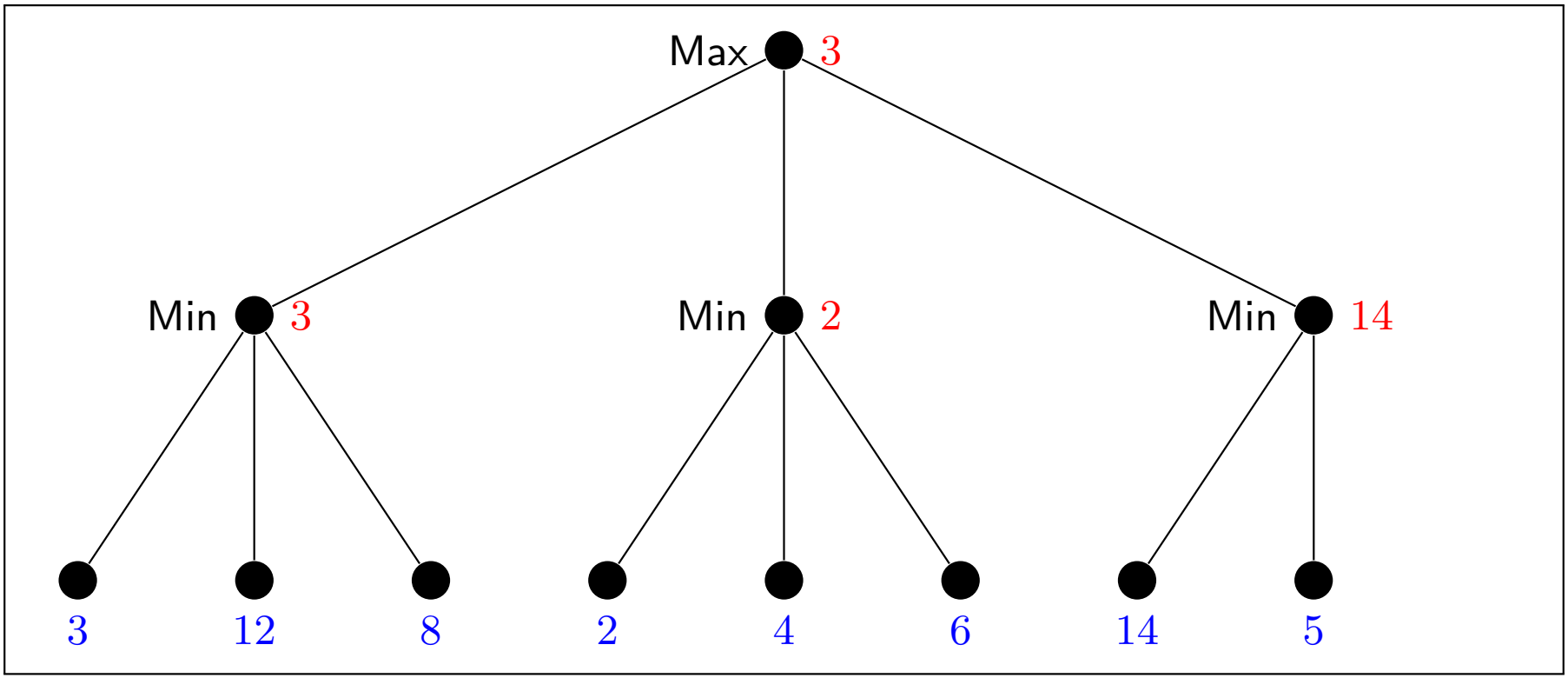
Minimax: Example, Now in Detail



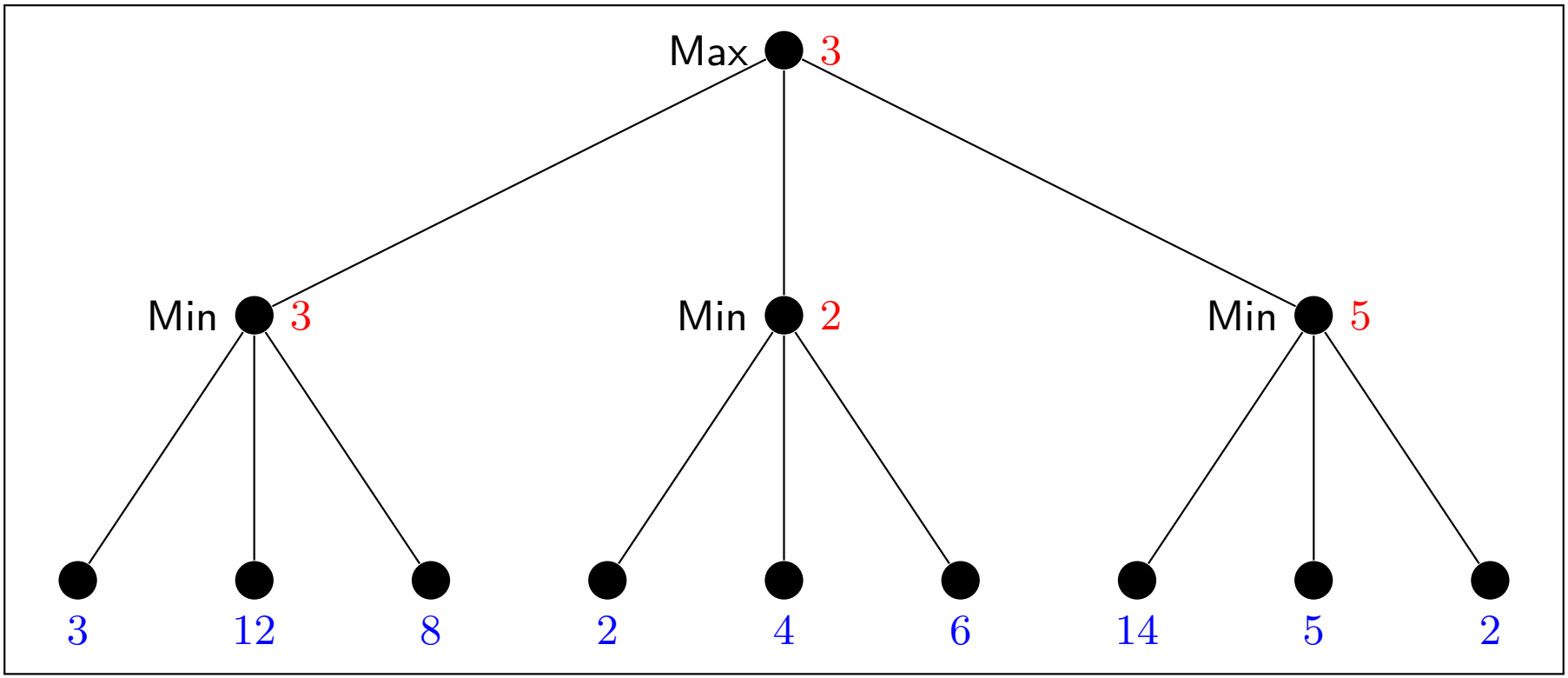
Minimax: Example, Now in Detail



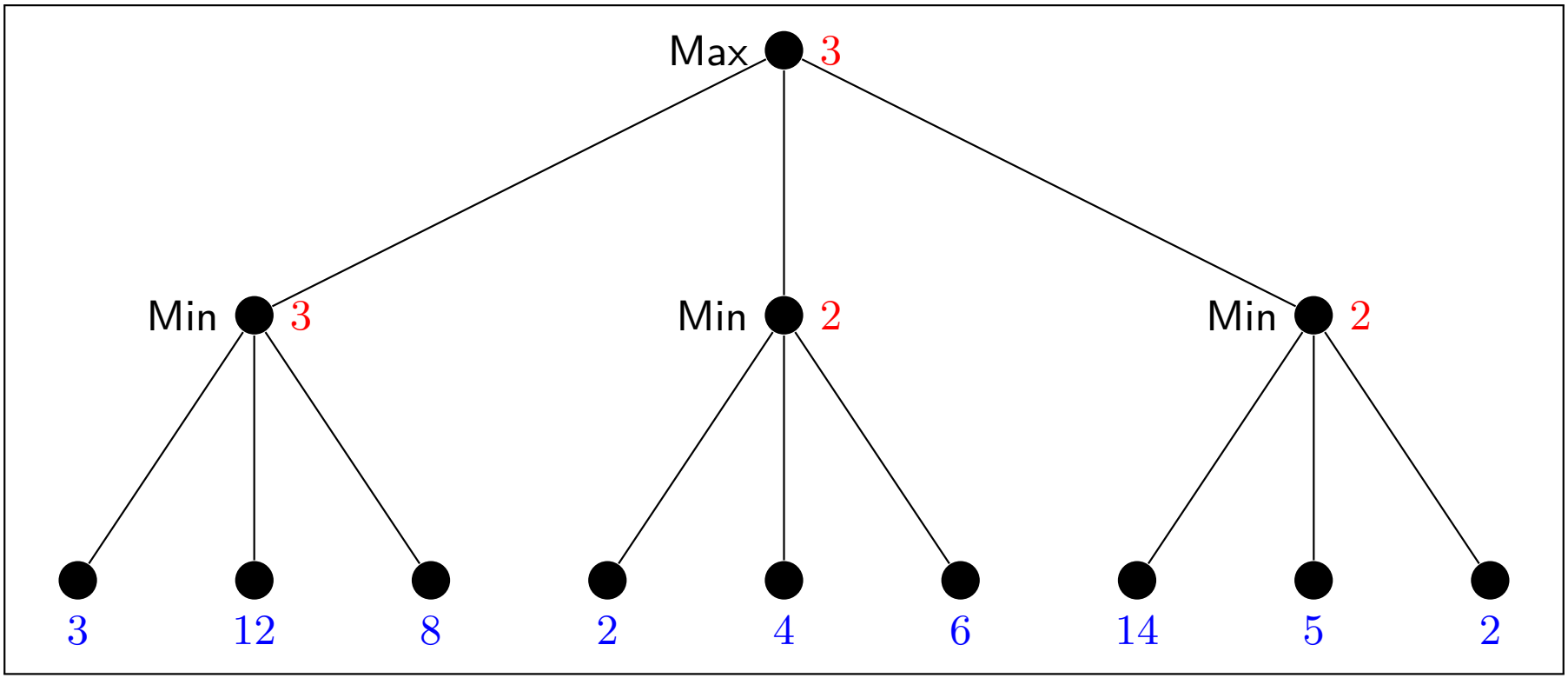
Minimax: Example, Now in Detail



Minimax: Example, Now in Detail



Minimax: Example, Now in Detail



→ So which action for Max is returned? Leftmost branch. Note: The maximal possible pay-off is higher for the rightmost branch, but assuming perfect play of Min, it's better to go left. (Going right would be "relying on your opponent to do something stupid".)

Minimax, Pro and Contra

Pro:

- Returns an optimal action, assuming perfect opponent play.
- Extremely simple.

Contra:

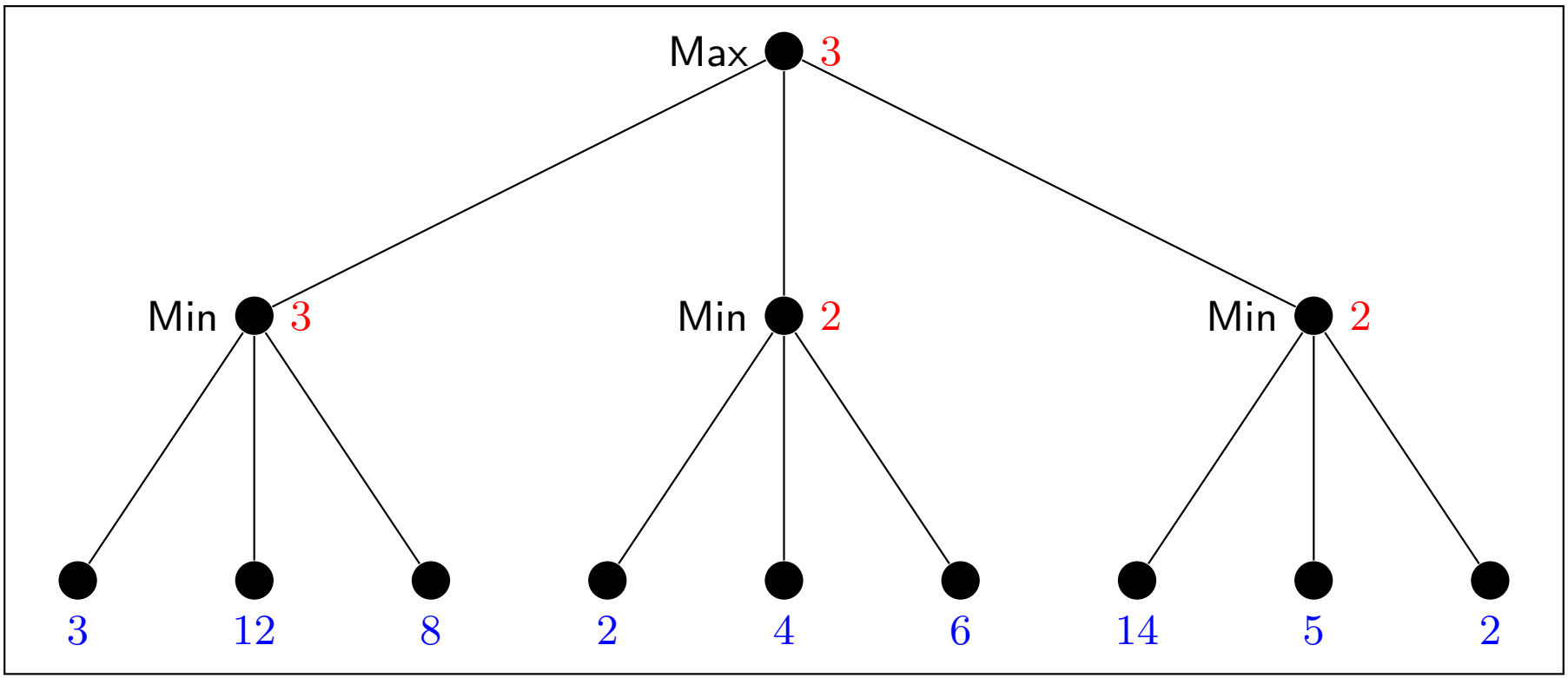
- Completely infeasible (search tree way too large).

Remedies:

- Limit search depth, apply evaluation function at cut-off states.
- Sparse search (MCTS) instead of exhaustive search.
- Alpha-beta pruning reduces search yet preserves optimality.

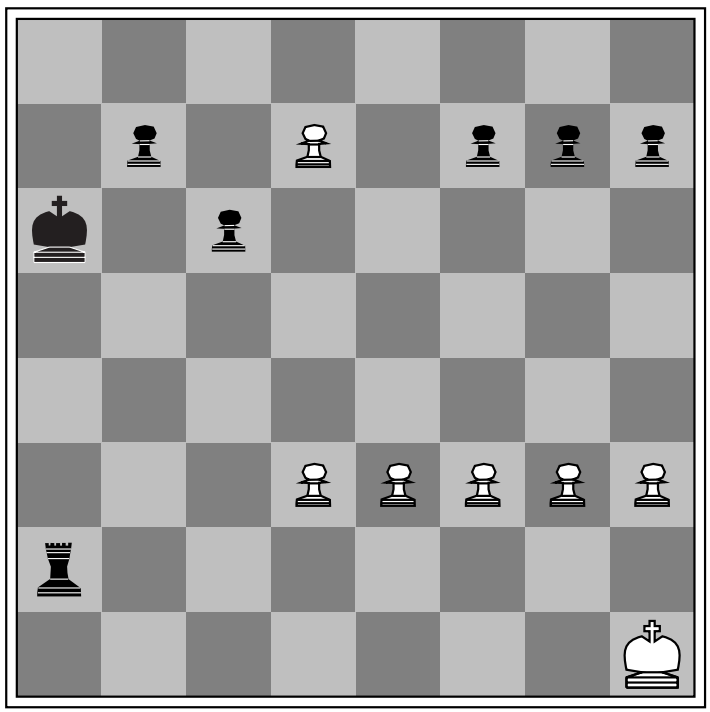
Minimax With Depth Limit: Example

Notation: blue: evaluation function value on cut-off states; red: non-cut-off state value as computed by Minimax with depth limit 2.



$\rightarrow$ Search pretends that states at depth limit d (number of actions i.e. half-moves) are terminal; requires evaluation function to estimate their values (see later).

Questionnaire



Black to move

Question!

Who's gonna win here?

(A): White (B): Black

- White wins (Pawn cannot be prevented from becoming a queen.)
- Black has a large advantage in material. If cut-off is here, then the evaluation function will say "-100, black wins".
- The loss for black is beyond our horizon unless we search extremely deeply: Black can hold off the end by repeatedly giving check to White's king.

$\rightarrow$ In other words: Minimax is not robust to inaccurate cut-off evaluations.

MCTS Single-Player Case (for Reference)

Algorithm sketch: (T: search tree prefix built so far)

while time not up **do**
  select actions within T up to a state s' and $s' \xrightarrow{a'} s''$ s.t. $s'' \notin T$
    with bias to maximize reward
  rollout from s'' until terminal state t
  add s'' to T
  update, from s'' up to root, #expansions and average rewards
return an a for s with maximal average $reward(a)$
When executing a, keep the part of T below a

Notes:

- Search sparse, information from deep samples.
- Action selection: within T. Maximize reward: only "bias" (not exclusive!), to ensure convergence to optimal choices.
- Different strategies/guidance for exploration vs. exploitation, e.g. UCT [Kocsis and Szepesvári (2006)].

MCTS Two-Player Case (for Reference)

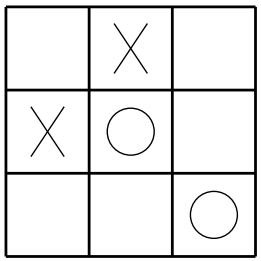
Two-Player Zero-Sum MCTS:

while time not up **do**
  select actions within T up to a state s' and $s' \xrightarrow{a'} s''$ s.t. $s'' \notin T$,
    with bias to maximize (minimize) reward in Max (Min) nodes
  rollout from s'' until terminal state t
  add s'' to T
  update, from a' up to root, #expansions and average rewards with $u(t)$
return an a for s with maximal average $reward(a)$
When executing a, keep the part of T below a

Notes:

- With suitable selection bias, action decisions in tree converge to optimal.
  $\Rightarrow$ Rewards converge to Minimax values.
- Sparse deep search = "focus on most relevant moves".
  $\Rightarrow$ Horizon problem not as critical. (May fall prey to "traps" though [Ramanujan *et al.* (2010)].)

Questionnaire



- Tic Tac Toe.
- Max = x, Min = o.
- Max wins: $u = 100$; Min wins: $u = -100$; stalemate: $u = 0$.

Question!

What's the Minimax value for the state shown above? (Note: Max to move)

(A): 100 (B): -100

$\rightarrow$ 100: Max moves; choosing the top left corner, it's a certain win for Max.

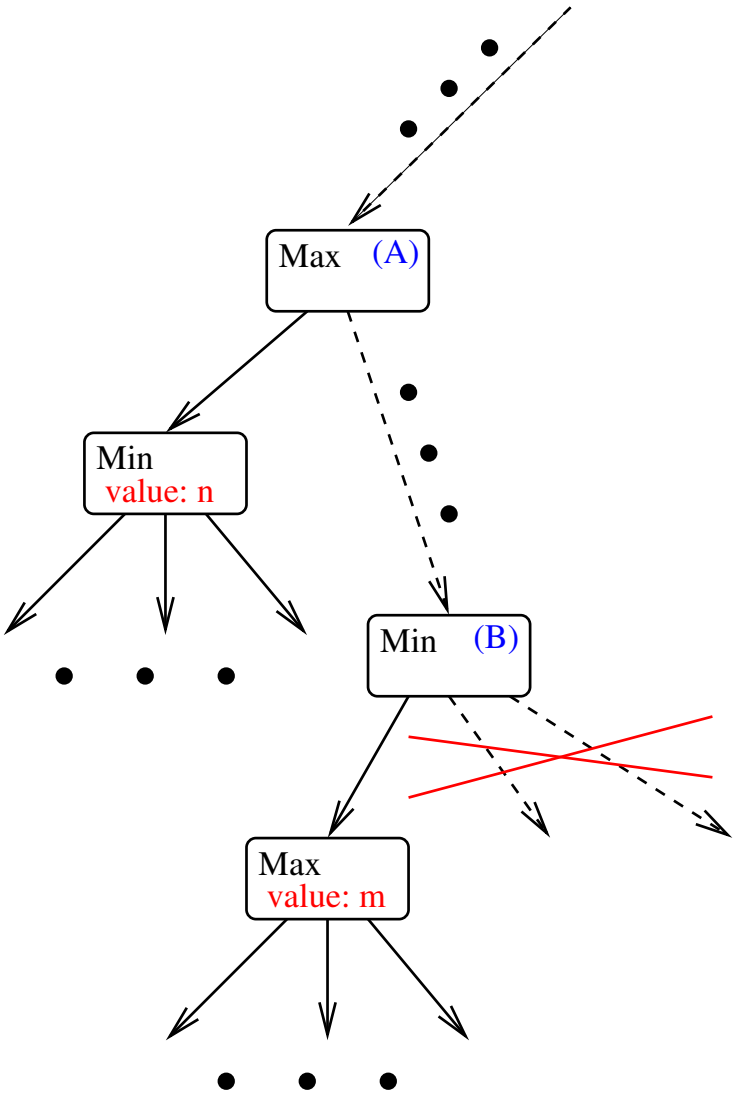
Question!

What's the Minimax value for the initial game state?

(A): 100 (B): -100

$\rightarrow$ The correct value (and thus the value computed by Minimax) is 0: Given perfect play, Tic Tac Toe always results in a stalemate. (Seen "War Games", anybody?)

Alpha Pruning: Idea



Say $n > m$.

$\rightarrow$ By choosing to move left in Max node (A), Max already can get utility at least n in this part of the game.

Say that below a different move at (A), in Min node (B) Min can force Max to get value $m < n$.

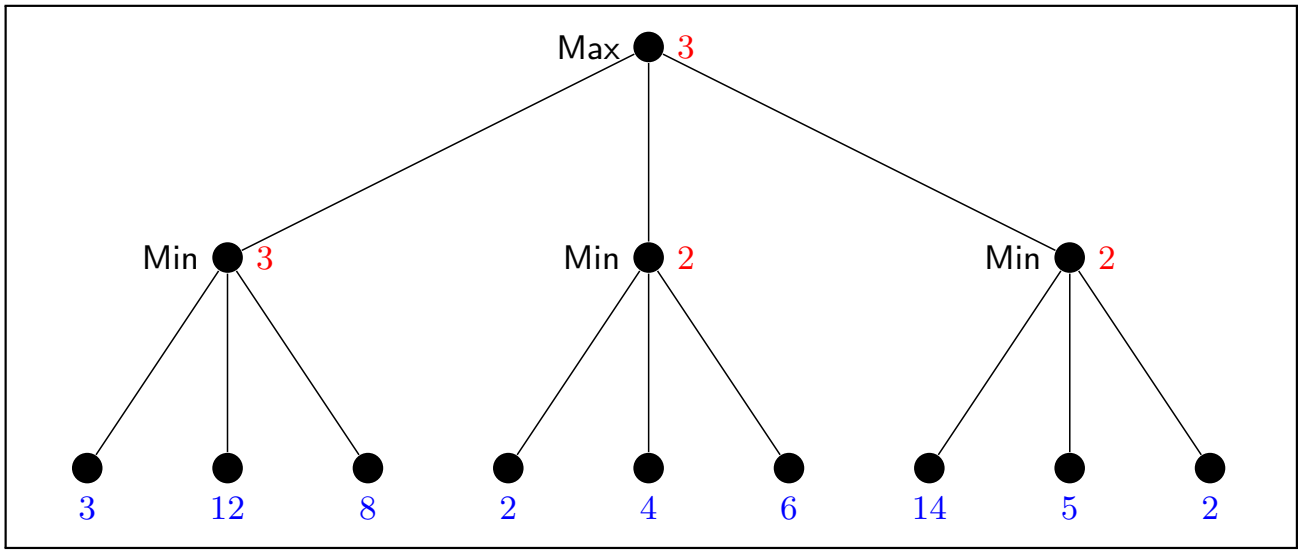
Then we already know that (B) will not be reached during the game, given the policy we currently compute for Max (Max can prevent the game from reaching (B), e.g. by moving left at (A)).

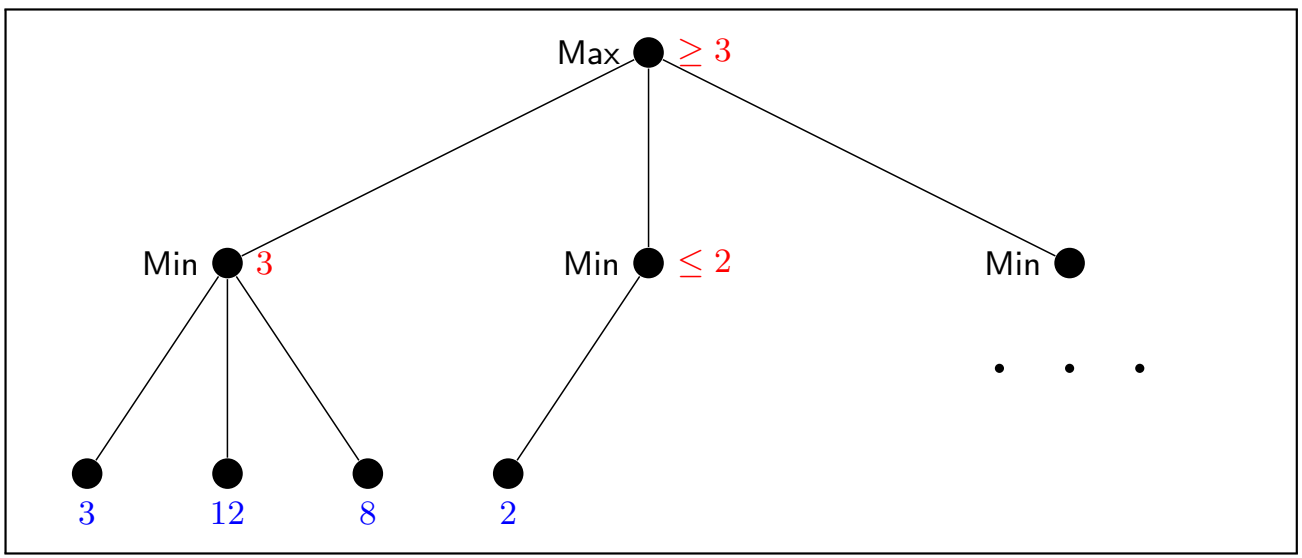
Hence we can spare ourselves the effort of searching the other children of (B).

Alpha Pruning: The Idea in Our Example

Question:

Can we save some work here?



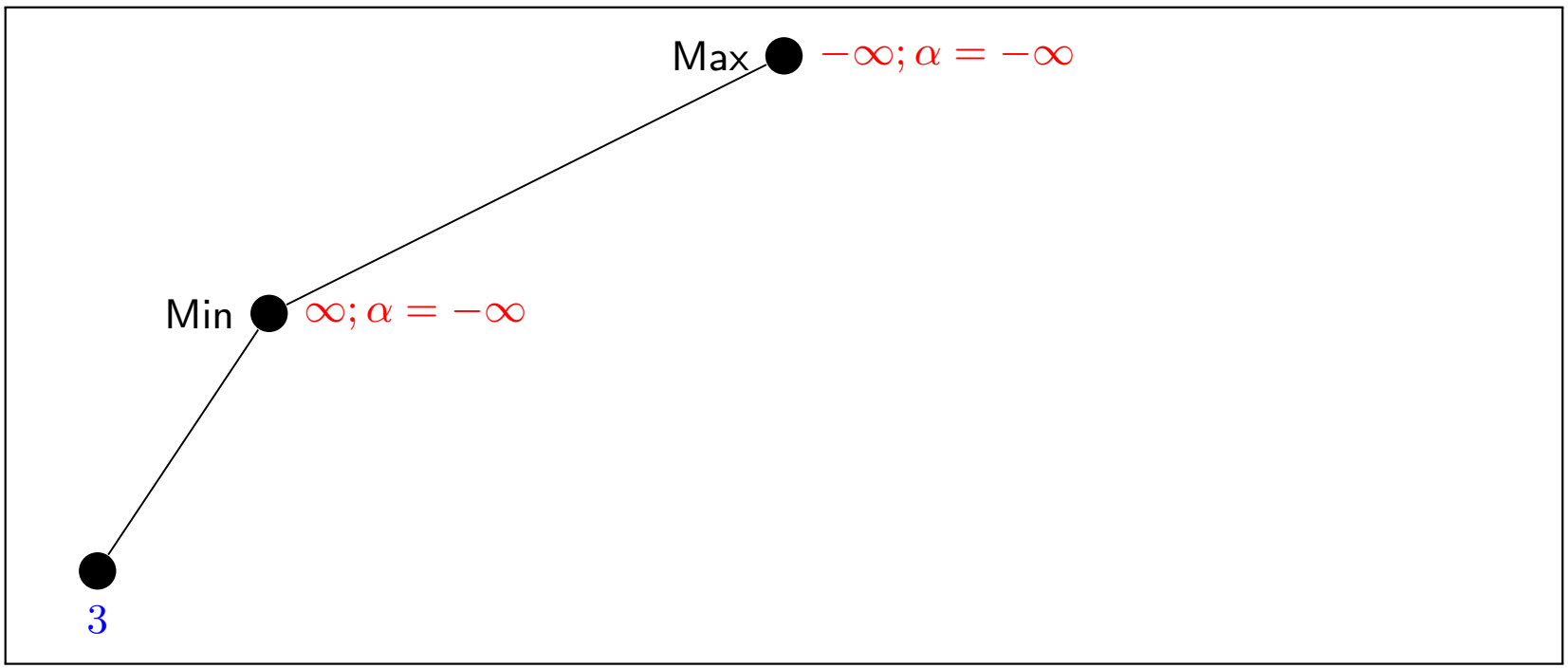


Answer: Yes!

→ We already know at this point that the middle action won't be taken by Max.

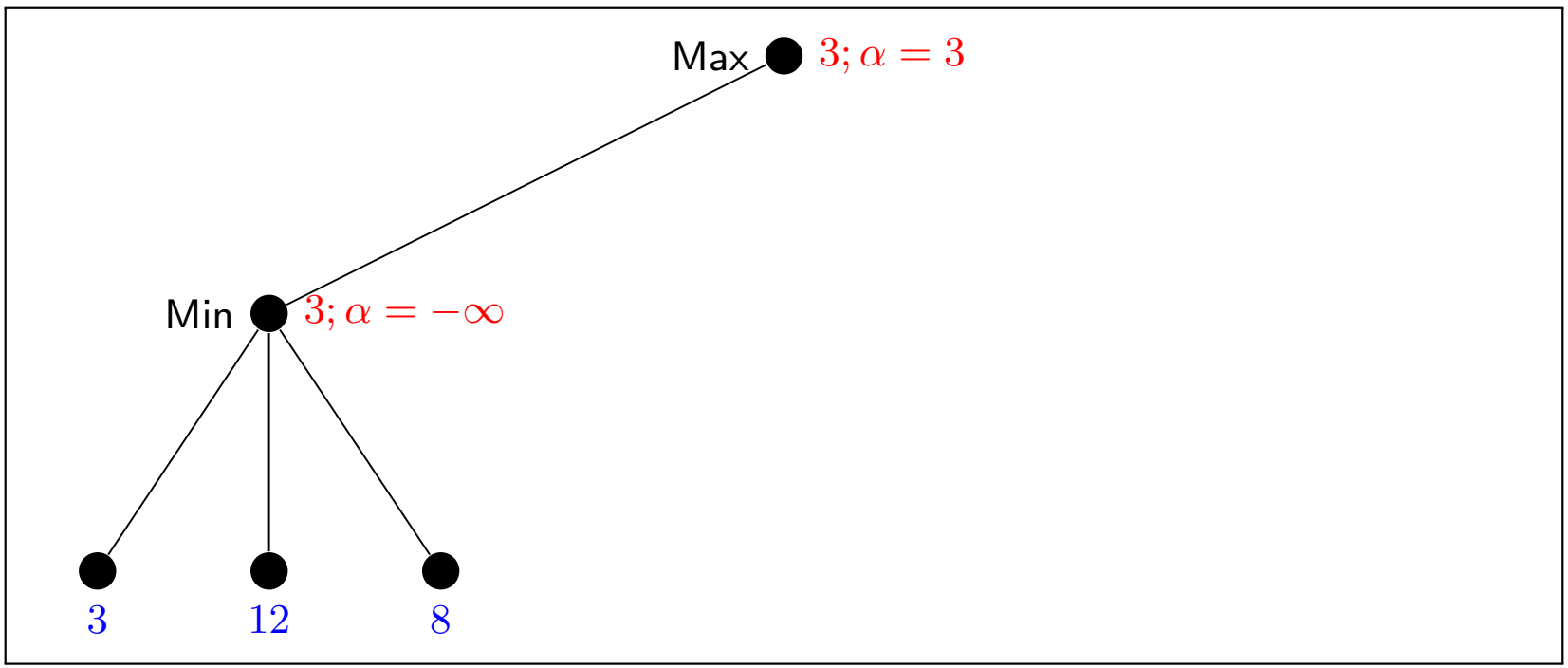
Alpha Pruning

What is α? For each search node n, the highest Max-node utility that search has found already on its path to n.



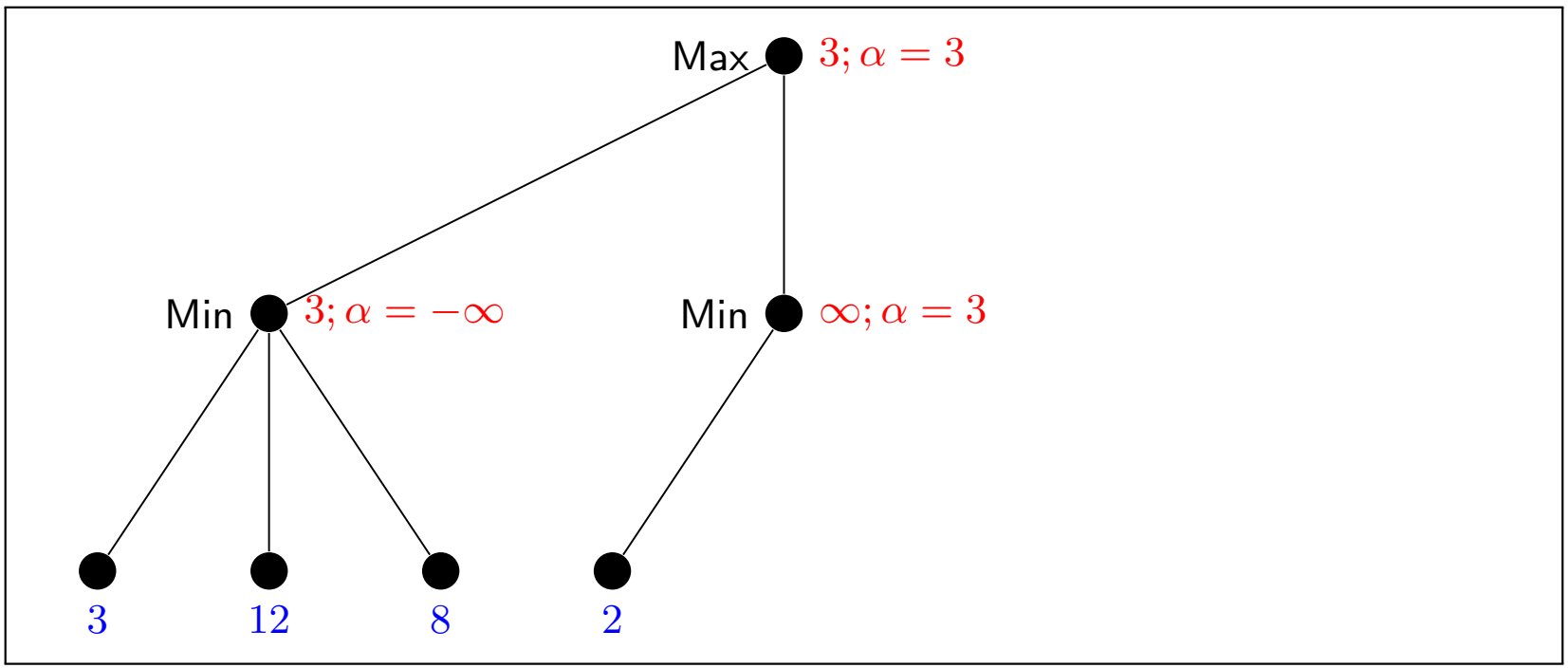
Alpha Pruning

What is α? For each search node n, the highest Max-node utility that search has found already on its path to n.



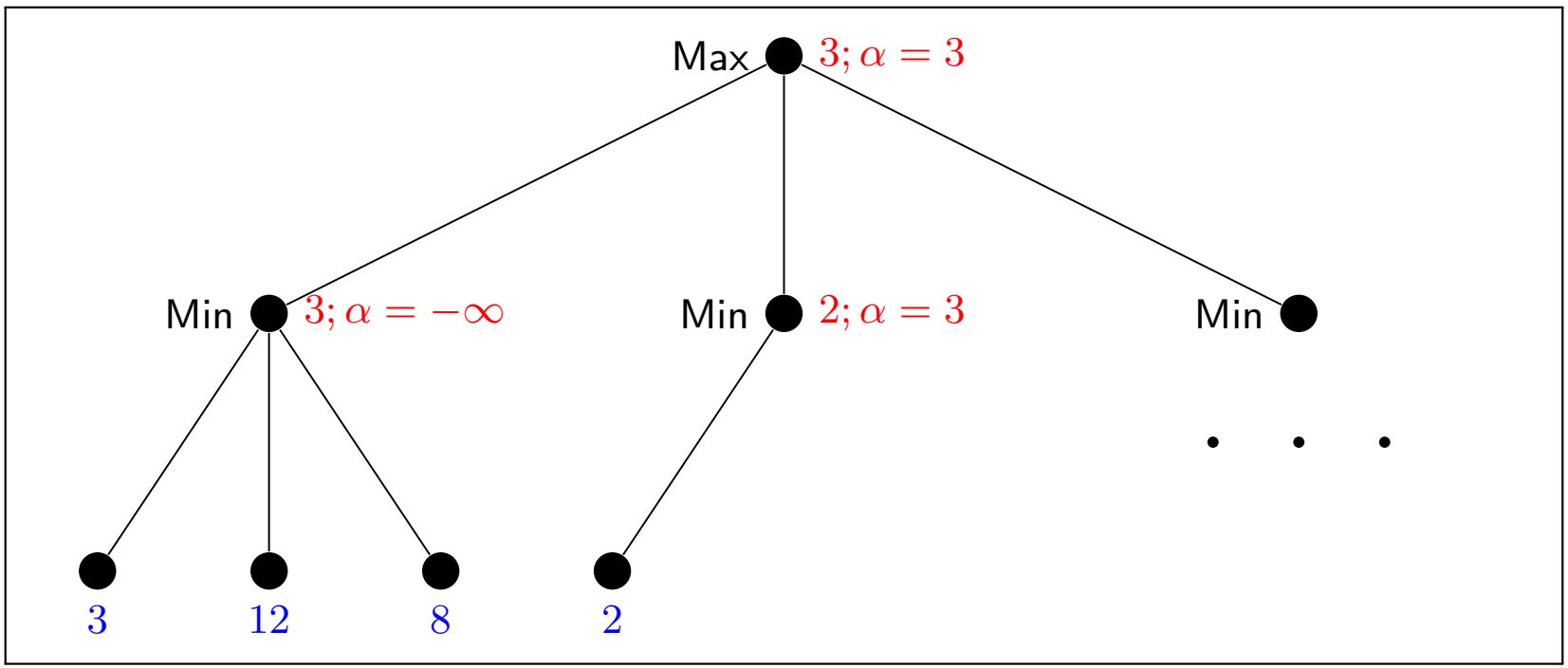
Alpha Pruning

What is α? For each search node n, the highest Max-node utility that search has found already on its path to n.



Alpha Pruning

What is α? For each search node n, the highest Max-node utility that search has found already on its path to n.



How to use α? In a Min node n, if one of the children already has utility $\leq \alpha$, then stop considering n. (Pruning out its remaining successors.)

Alpha-Beta Pruning

Reminder:

- **What is α:** For each search node n, the highest Max-node utility that search has found already on its path to n.
- **How to use α:** In a Min node n, if one of the successors already has utility $\leq \alpha$, then stop considering n. (Pruning out its remaining successors.)

We can use a dual method for Min:

- **What is β:** For each search node n, the lowest Min-node utility that search has found already on its path to n.
- **How to use β:** In a Max node n, if one of the successors already has utility $\geq \beta$, then stop considering n. (Pruning out its remaining successors.)

. . . and of course we can use both together.

Alpha-Beta Search: Pseudo-Code

function Alpha-Beta-Search(s) **returns** an action
  $v \leftarrow$ *Max-Value*$(s, -\infty, +\infty)$
  return an *action* $a \in$ *Actions(s)* yielding value v

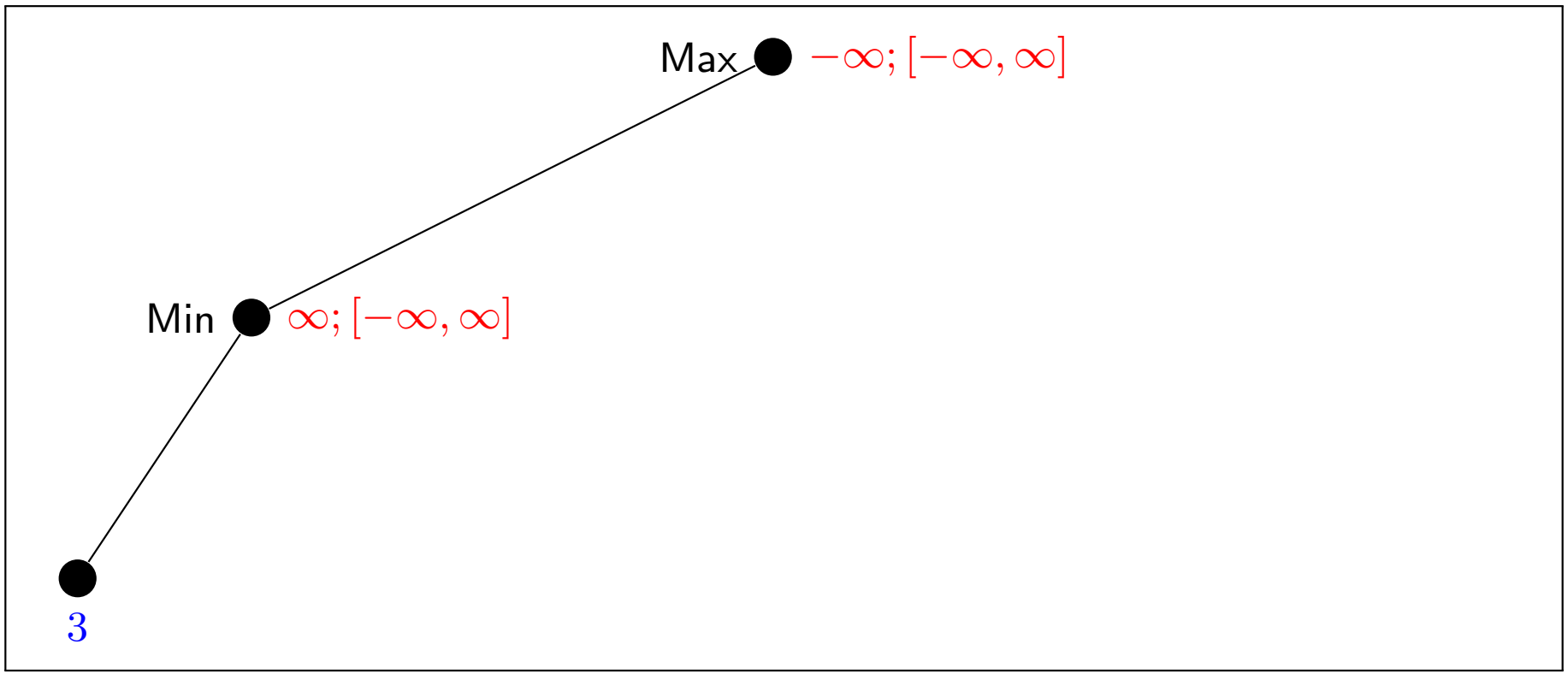
function Max-Value(s, α, β) **returns** a utility value
  if *Terminal-Test*(s) **then return** $u(s)$
  $v \leftarrow -\infty$; $\alpha' \leftarrow \alpha$
  for each $a \in$ *Actions(s)* **do**
    $v \leftarrow \max(v,$*Min-Value*$($*ChildState*$(s, a), \alpha', \beta))$
    $\alpha' \leftarrow \max(\alpha', v)$
    if $v \geq \beta$ **then return** v /* Here: $v \geq \beta \Leftrightarrow \alpha \geq \beta$ */
  return v

function Min-Value(s, α, β) **returns** a utility value
  if *Terminal-Test*(s) **then return** $u(s)$
  $v \leftarrow +\infty$; $\beta' \leftarrow \beta$
  for each $a \in$ *Actions(s)* **do**
    $v \leftarrow \min(v,$*Max-Value*$($*ChildState*$(s, a), \alpha, \beta'))$
    $\beta' \leftarrow \min(\beta', v)$
    if $v \leq \alpha$ **then return** v /* Here: $v \leq \alpha \Leftrightarrow \alpha \geq \beta$ */
  return v

$=$ Minimax (slide 18) $+$ α/β book-keeping and pruning.

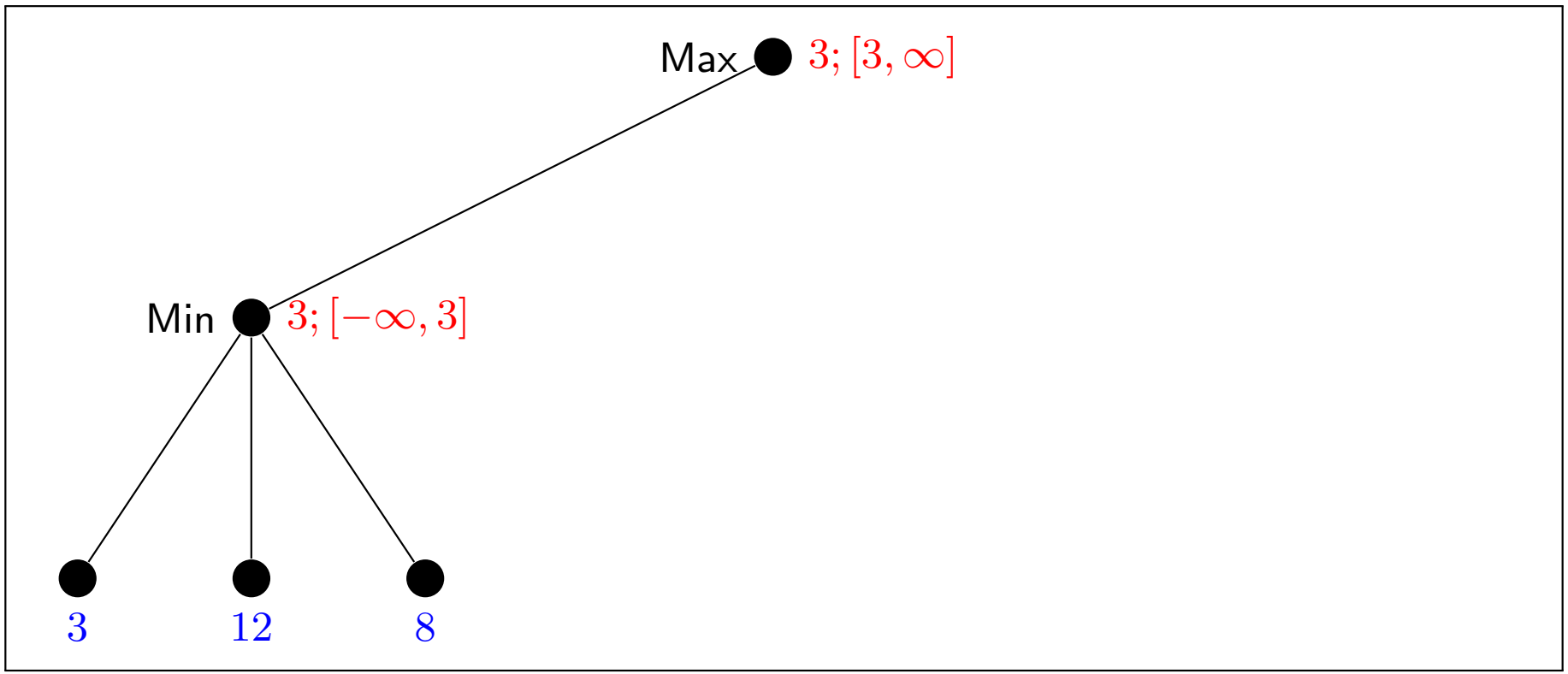
Alpha-Beta Search: Example

Notation: $v; [\alpha, \beta]$



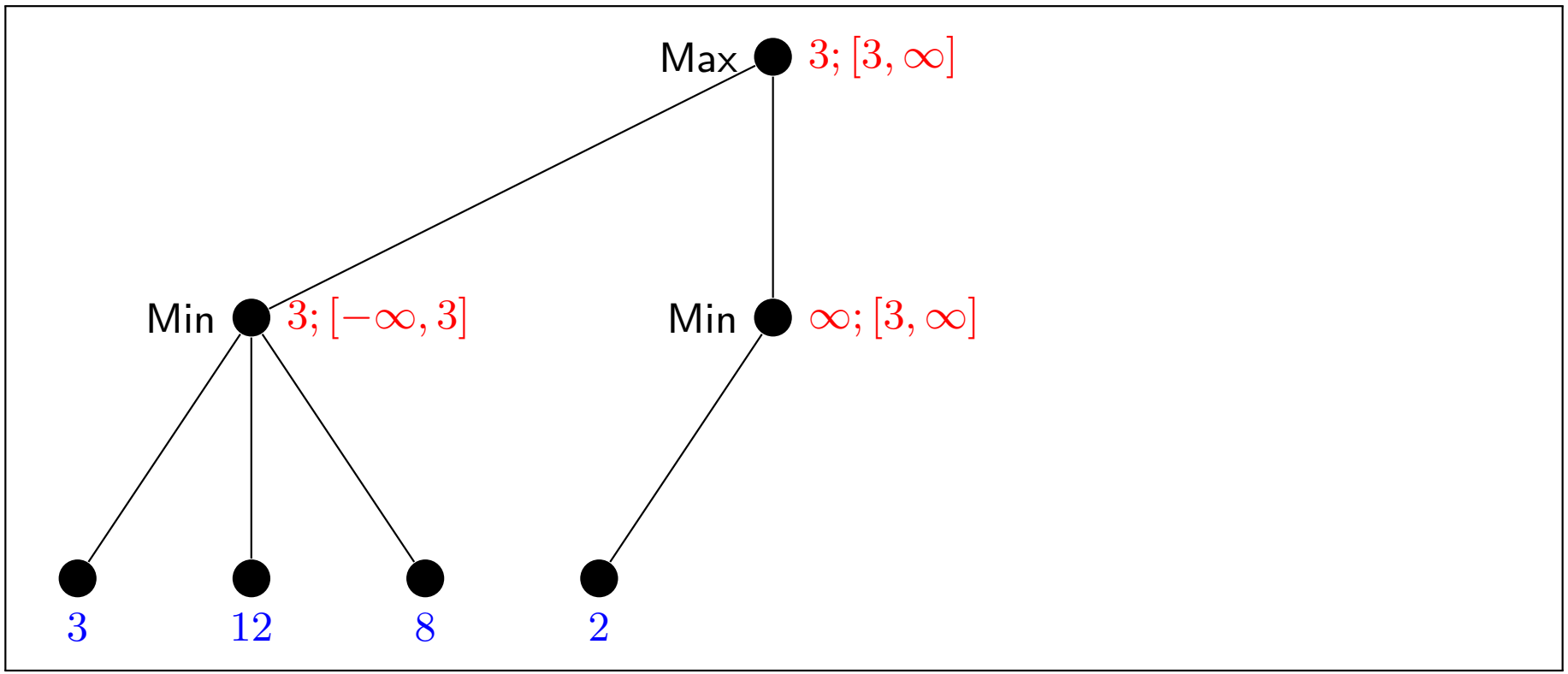
Alpha-Beta Search: Example

Notation: $v; [\alpha, \beta]$



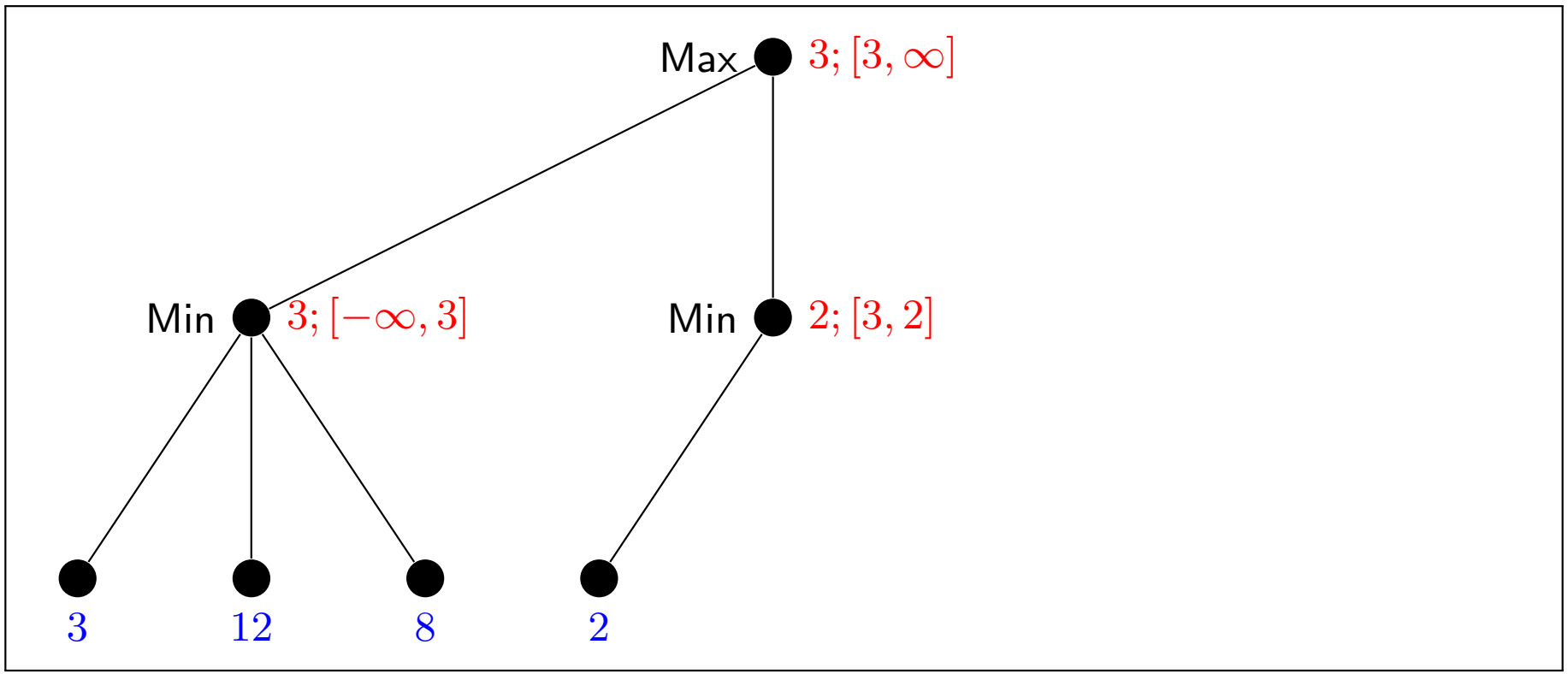
Alpha-Beta Search: Example

Notation: $v; [\alpha, \beta]$



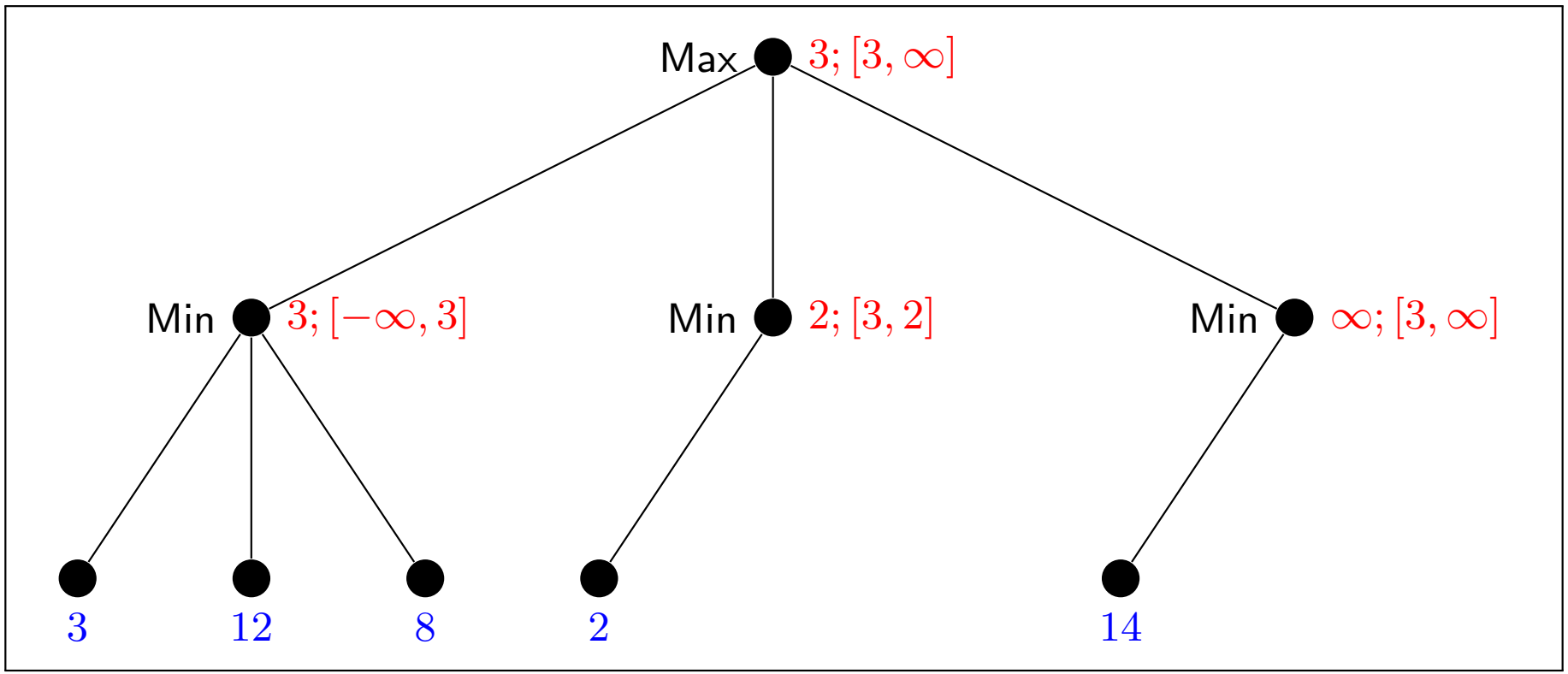
Alpha-Beta Search: Example

Notation: $v; [\alpha, \beta]$



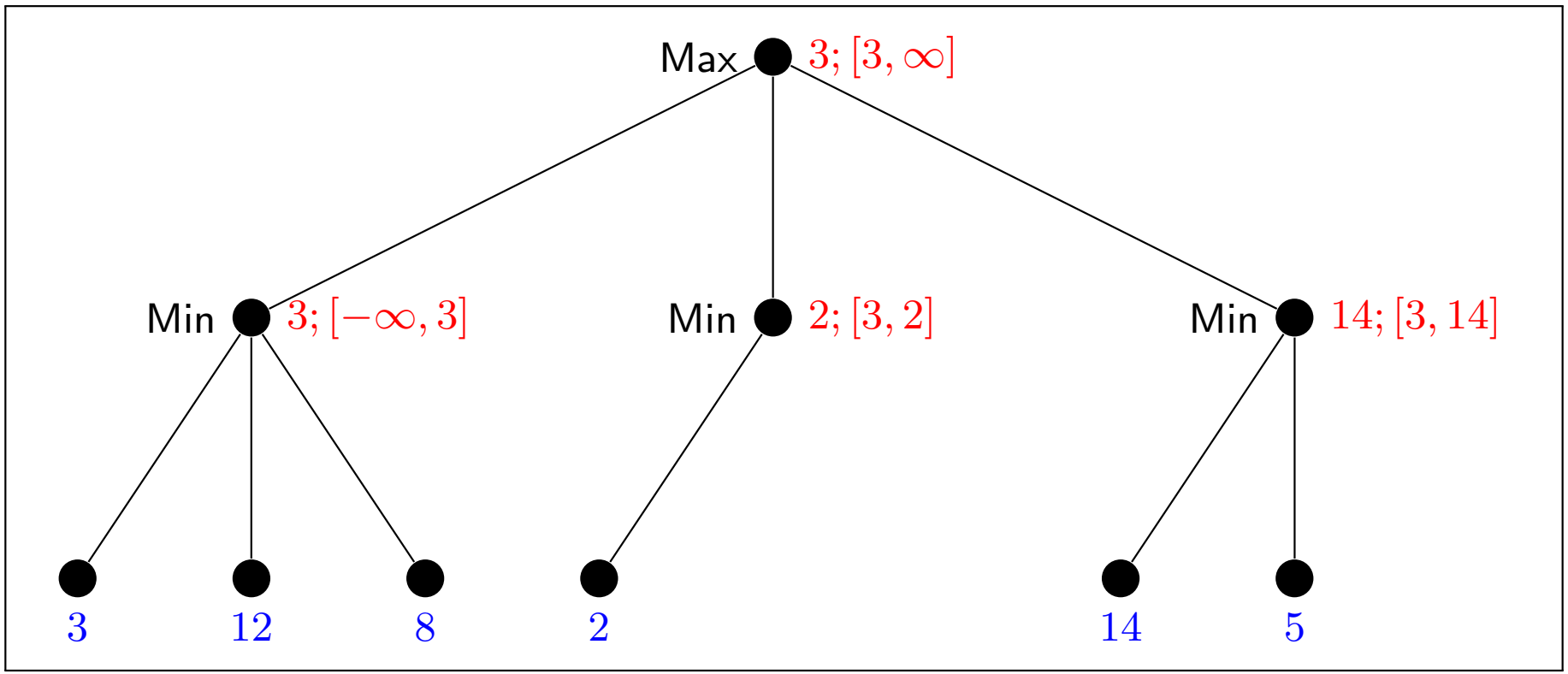
Alpha-Beta Search: Example

Notation: $v; [\alpha, \beta]$



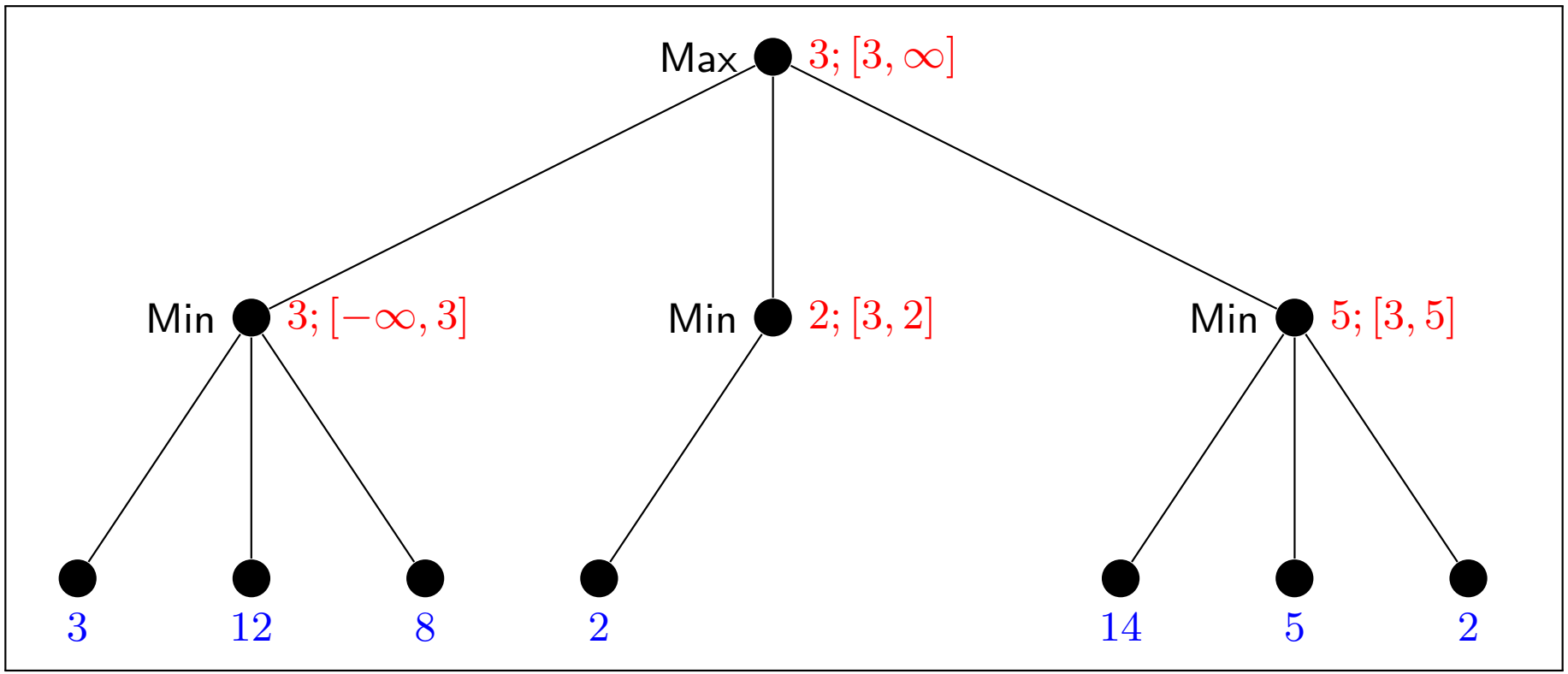
Alpha-Beta Search: Example

Notation: $v; [\alpha, \beta]$



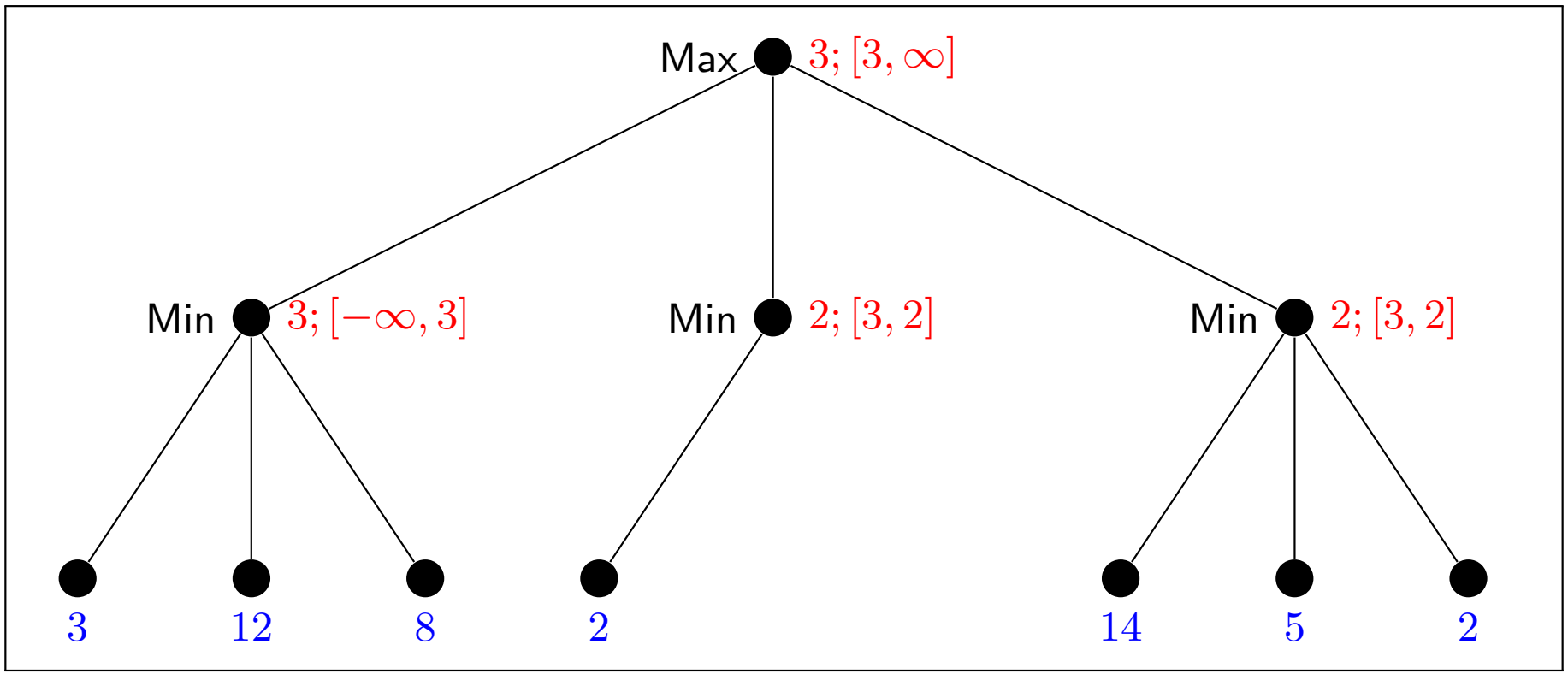
Alpha-Beta Search: Example

Notation: $v; [\alpha, \beta]$



Alpha-Beta Search: Example

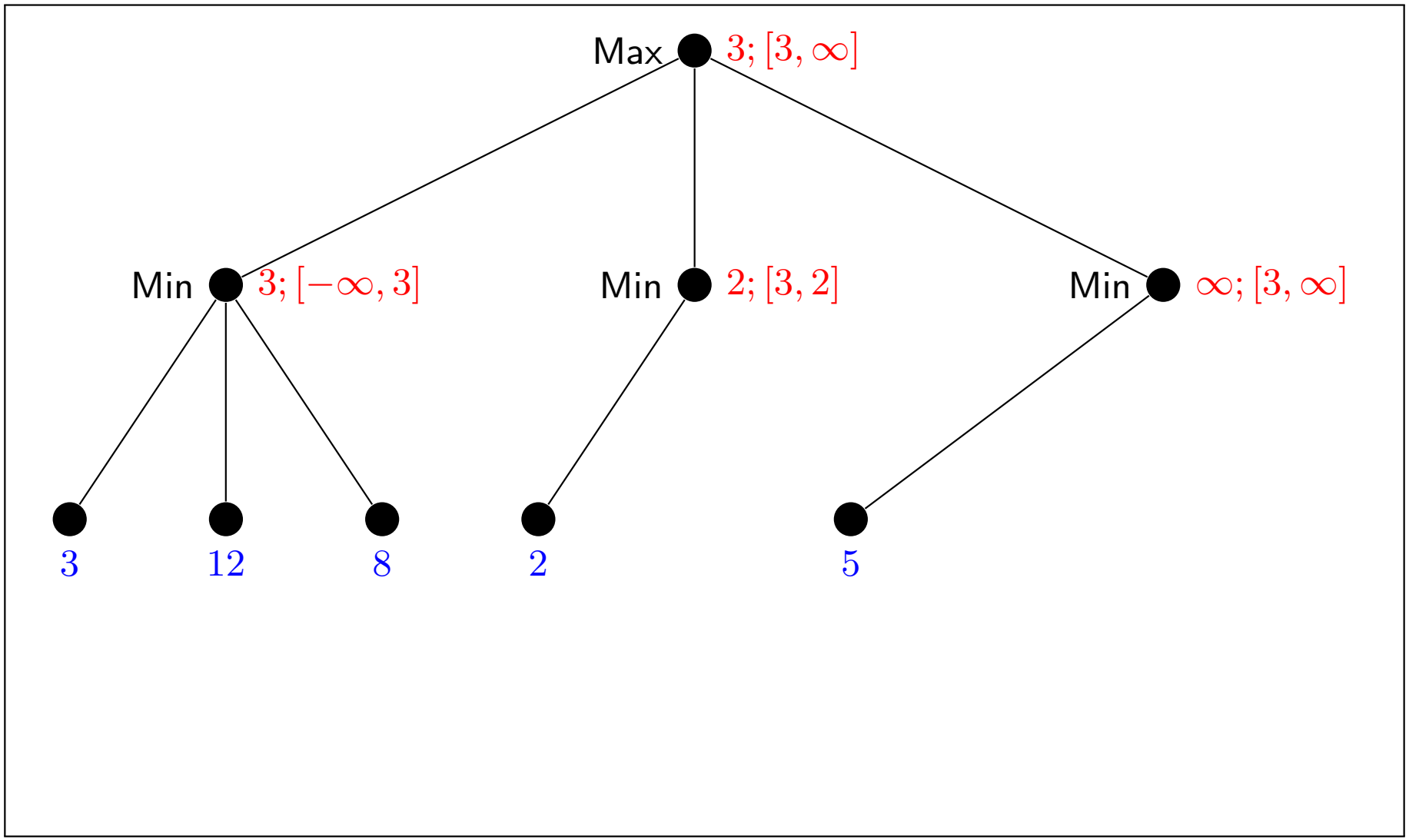
Notation: $v; [\alpha, \beta]$



→ Note: We could have saved work by choosing the opposite order for the successors of the rightmost Min node. Choosing the best moves (for each of Max and Min) first yields more pruning!

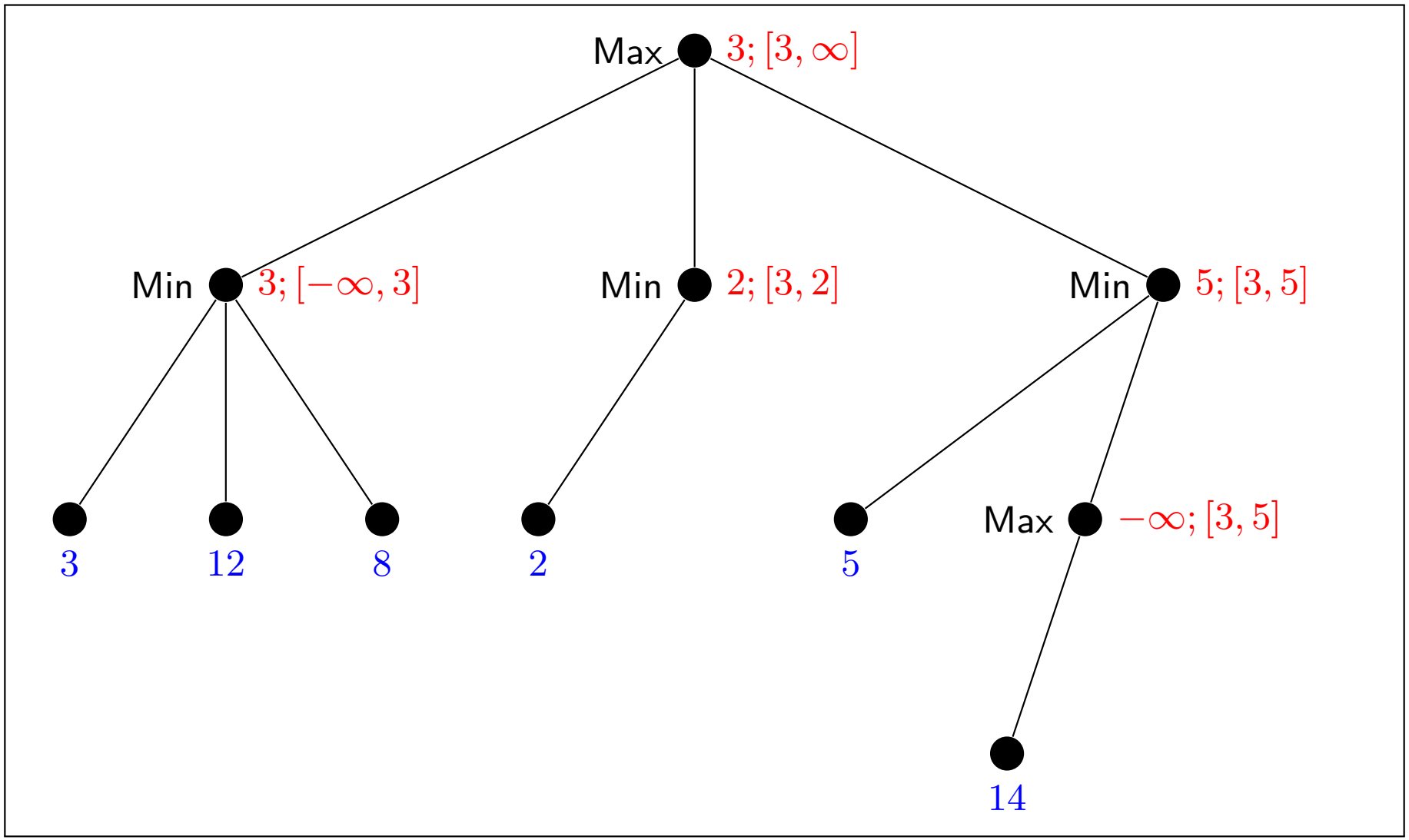
Alpha-Beta Search: Modified Example

Showing off some actual β pruning:



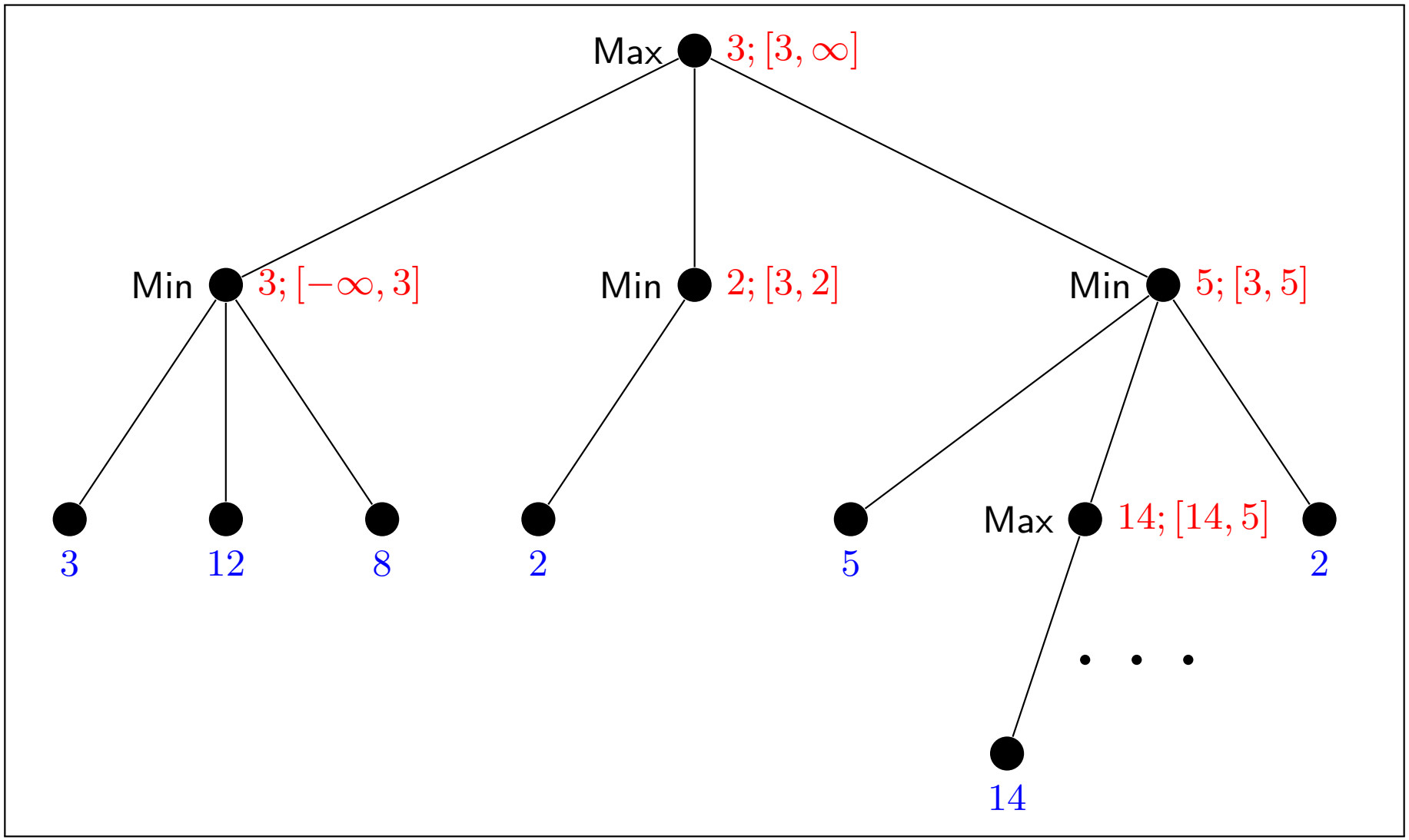
Alpha-Beta Search: Modified Example

Showing off some actual β pruning:



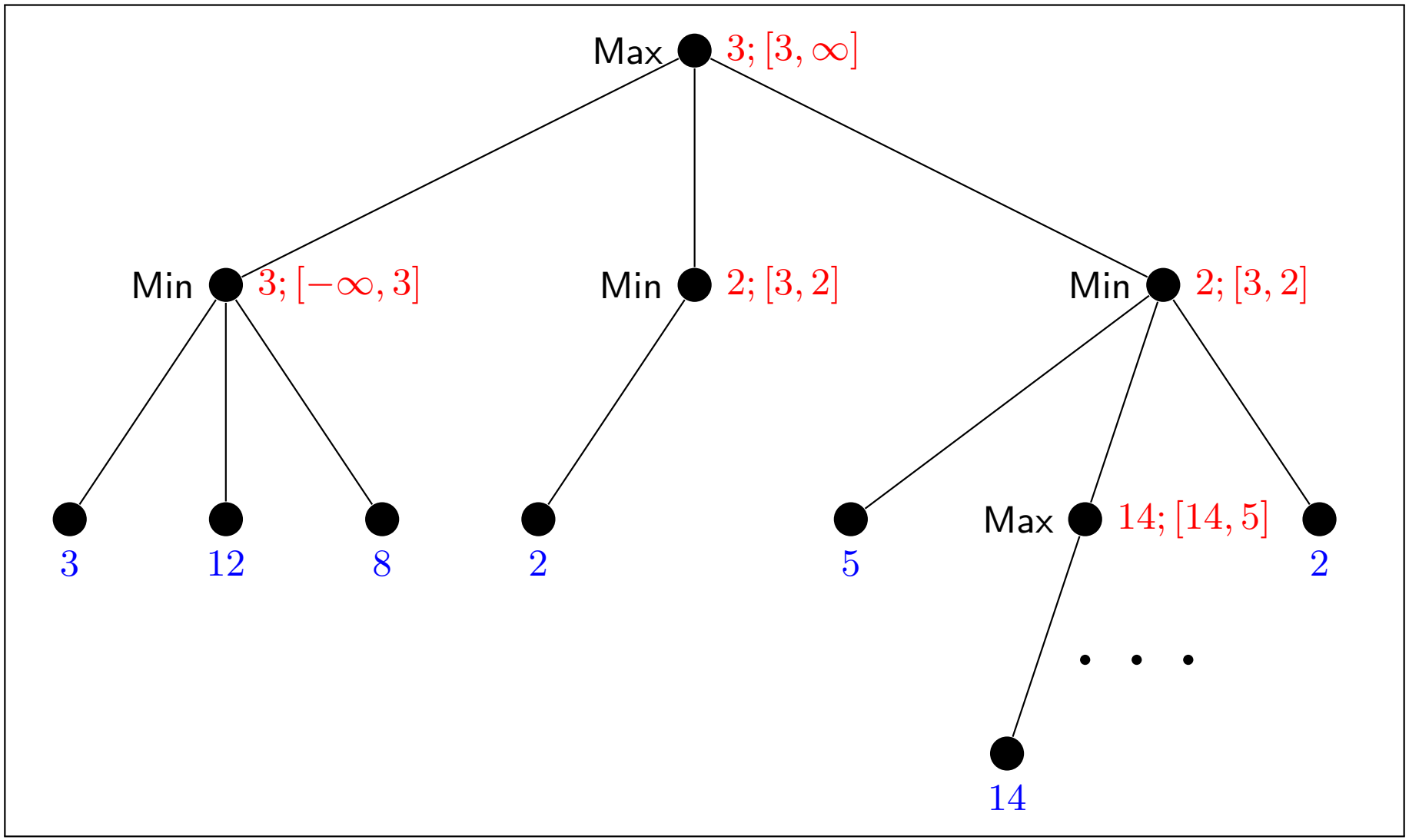
Alpha-Beta Search: Modified Example

Showing off some actual β pruning:



Alpha-Beta Search: Modified Example

Showing off some actual β pruning:



How Much Pruning Do We Get?

→ Choosing best moves first yields most pruning in alpha-beta search.

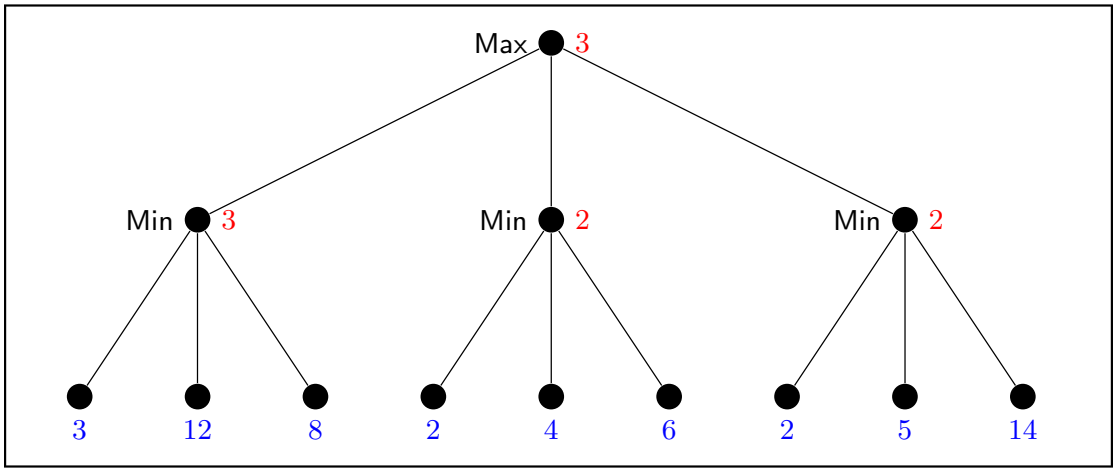
With branching factor b and depth limit d:

- Minimax: b^d nodes.
- Best case: Best moves first $\Rightarrow b^{d/2}$ nodes! Double the lookahead!
- Practice: Often possible to get close to best case.

Example Chess:

- Move ordering: Try captures first, then threats, then forward moves, then backward moves.
- Double lookahead: E.g. with time for 10^9 nodes, Minimax 3 rounds (white move, black move), alpha-beta 6 rounds.

Questionnaire



Question!

With left-to-right node ordering, how many nodes does alpha-beta prune here?

(A): 0 (B): 2

(C): 4 (D): 6

$\rightarrow$ (C): Same example as before, except that we changed the ordering of the right-branch leaves to have the best Min move first. Thus the $f = 5$ and $f = 14$ right-branch leaves will now be pruned. As before, the $f = 4$ and $f = 6$ middle-branch leaves will be pruned, yielding a total of 4 pruned nodes.

Evaluation Functions

Definition: *Given a game with states S, a (heuristic) evaluation function is a function $h : S \mapsto \mathbb{R}$.*

- h estimates the expected utility of s. (In particular, we can use $h := u$ on terminal states)
- In Minimax: Impose depth limit, use h at (non-terminal) cut-off states.
- In MCTS: Use h as part of the state-value estimates. (e.g. AlphaGo: leaf state value estimate is linear combination of h and rollouts)

How to obtain h?

- Relaxed game: Possible in principle, too costly in practice.
- Encode human expert knowledge.
- Learn from data.

Position Evaluation in Chess

- Material: Pawn (Bauer) 1, Knight (Springer) 3, Bishop (Läufer) 3, Rook (Turm) 5, Queen (Dame) 9.
  $\rightarrow$ Rule of thumb:
  3 points advantage $\Longrightarrow$ safe win.
- Mobility: How many fields do you control?
- King safety, Pawn structure, ...

Linear Feature-Based Evaluation Functions

Functions taking the form:

$$h(s) := w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s)$$

f_i are features, w_i are weights.

How to obtain such functions?

- Features f_i designed by human experts.
- Weights w_i set by experts, or learned automatically (see later).

Discussion: Pro/Con

- Very fast. (Unless there are many features or computing their value is very expensive; incremental computation helps)
- Very simplistic. For example, assumes that features are independent. (But, e.g., value of Rook depends on Pawn structure)
- Human knowledge crucial in design of features.

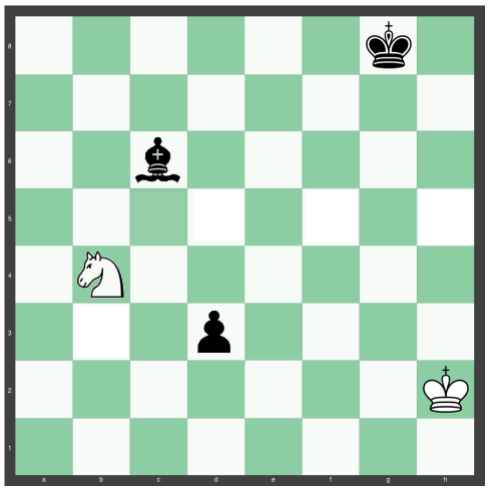
Feature-Based Evaluation in Chess

- Material: Pawn (Bauer) 1, Knight (Springer) 3, Bishop (Läufer) 3, Rook (Turm) 5, Queen (Dame) 9.
  $\rightarrow$ Rule of thumb:
  3 points advantage $\implies$ safe win.
- Mobility: How many fields do you control?
- King safety, Pawn structure, ...

$\rightarrow h(s) = \Delta pawn(s) + 3 * \Delta knight(s) + 3 * \Delta bishop(s) +$
$5 * \Delta rook(s) + 9 * \Delta queen(s)$ (Δ: #White$-$#Black)

$+w_k kingsafety(s) + w_p pawnstructure(s)$?

Questionnaire



- White to move.
- $h(s) = \Delta pawn(s) + 3 * \Delta knight(s) + 3 * \Delta bishop(s) + 5 * \Delta rook(s) + 9 * \Delta queen(s)$.

  (Δ: #White$-$#Black)

Question!

Say Minimax with depth limit d uses h at cut-off states. Which move does it choose in this state with a depth limit of $d = 1$ half-moves (i.e. considering only the first action)? For which values of d does it choose the best action?

$\rightarrow$ With $d = 1$, Minimax chooses to capture the black bishop due to the superior material advantage.

$\rightarrow$ The best action is to capture the black pawn, as this is the only way to prevent it from turning into a queen. To see this, we need $d \geq 4$.

Supervised Learning of Evaluation Functions (for Reference)

Human expert annotates states with evaluation-function value
$\Rightarrow$ standard supervised learning problem.

- Set of annotated states, i.e., state/value pairs (s, v).
- Learn ML model that predicts output v from input s.

Possible ML methods: arbitrary

- Classic approach: learn weights in feature-based evaluation function.
- Recent breakthrough successes: neural networks!

Policies & Supervised Learning (for Reference)

Definition: *By p, we denote a (combined) policy for both players. A probabilistic policy returns a probability distribution over actions instead.*

- An optimal policy captures perfect play from both players.
- (Probabilistic) policies can be used as search guidance in MCTS: action selection in tree, action selection in rollouts.

Supervised learning of policies:

Human expert annotates states with preferred **moves**
$\Rightarrow$ standard supervised classification problem.

- Way more natural for humans; side effect of expert game play.
- e.g. KGS Go Server: 30 million positions with expert moves, used for training in AlphaGo.

Learning from Self-Play (for Reference)

Self-play for reinforcement learning:

Repeat: play a game using the current h and/or p; at each step (s_t, a_t) along the game trace, reinforce the game outcome in h and/or p.

- Evaluation function learning: update weights in h to reduce the error $h(s_t) - \mathit{game\text{-}outcome\text{-}value}$.
- Probabilistic policy learning: update weights in p to increase (game won)/decrease (game lost) the likelihood of choosing a_t in s_t.

Self-play to generate data for supervised learning:

Fix policy p. Repeat: play game using p; annotate the states in each game trace with the game outcome value.

- Use this data for supervised learning of evaluation function.
- Might sound strange, but actually successful: used in AlphaGo.

The AlphaGo/Zero System Series

AlphaGo, March 2016:

https://www.youtube.com/watch?v=WXuK6gekU1Y

- AlphaGo beats Lee Sedol (winner of 18 world titles).
- MCTS with guidance information from neural networks (NN), trained by expert data and self-play.

AlphaGo Zero, early 2017:

https://deepmind.com/blog/alphago-zero-learning-scratch/

- AlphaGo Zero beats AlphaGo using NN trained without expert data.
- Attention: Training time in "days" requires Google computing power! Intensely computation-expensive but massively parallelizable.

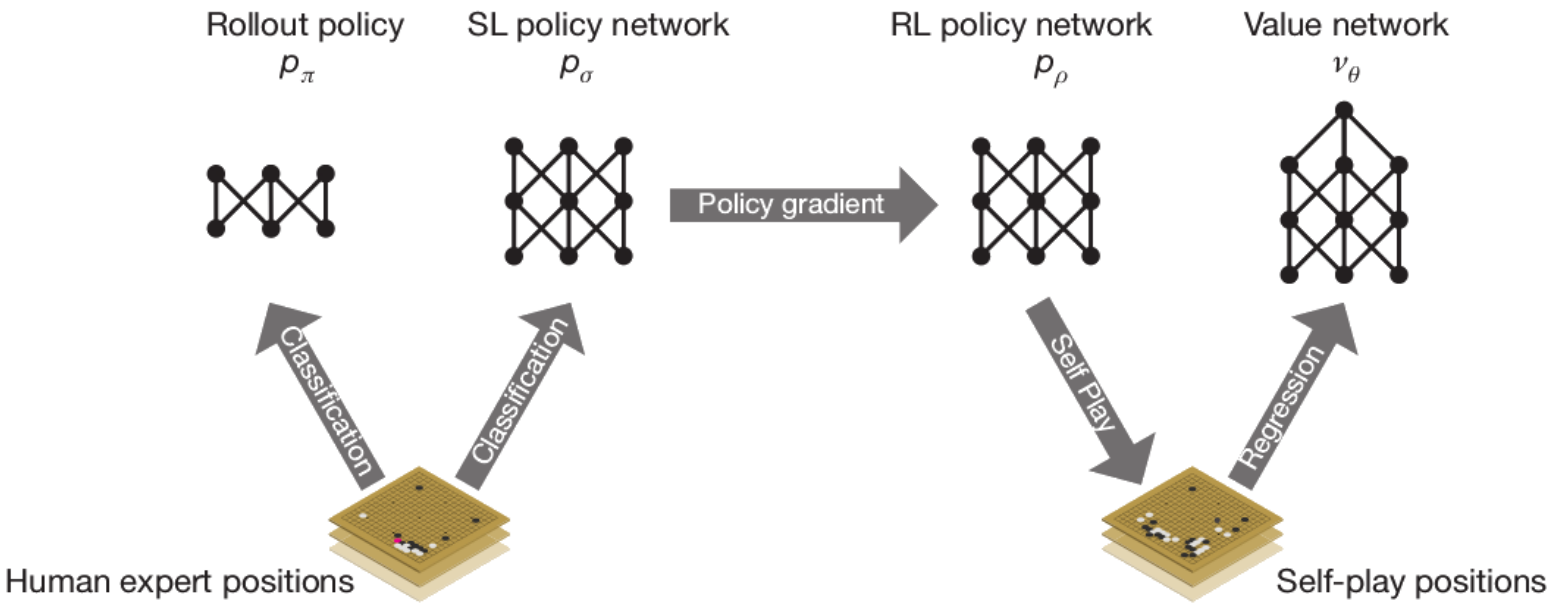
AlphaZero, late 2017:

https://deepmind.com/blog/alphazero-shedding-new-light-grand-games-chess-shogi-and-go/

- AlphaZero beats world-class computer players in Go, Chess, and Shogi.

Learning in AlphaGo

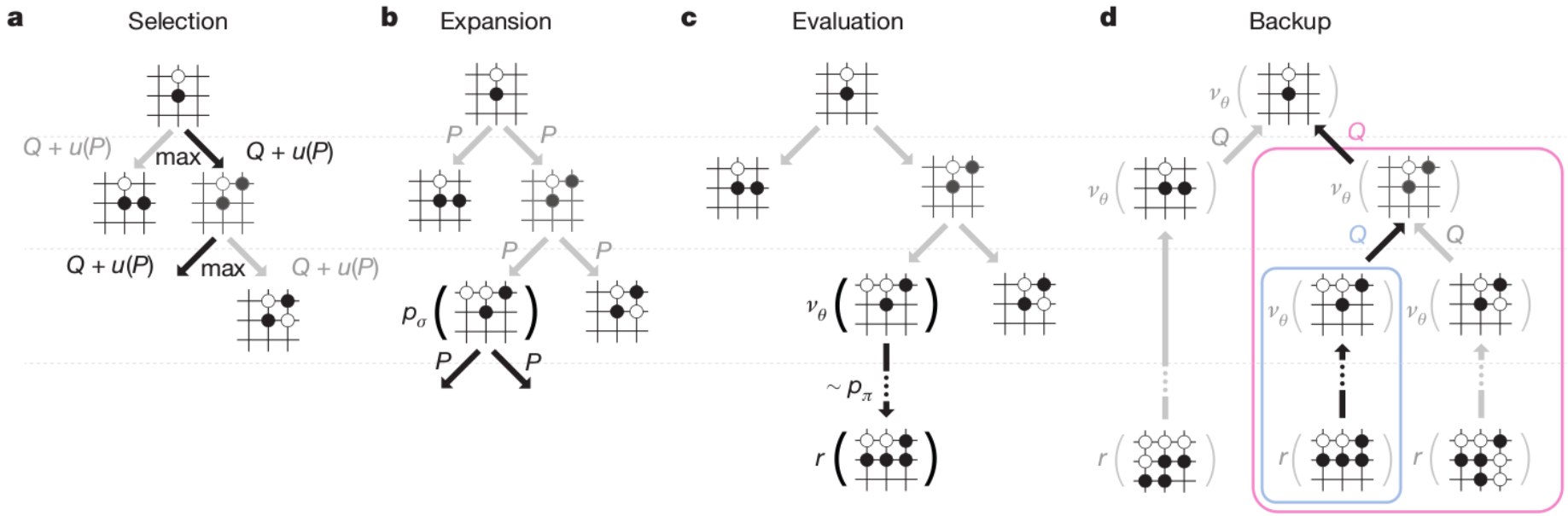
Illustration: (taken from [Silver *et al.* (2016)])



- SL policy network p_σ: Supervised learning from human expert data (cf. slide 43).
- Rollout policy p_π: Simple but fast version of p_σ (linear feature based function for each action, cf. slide 39).
- RL policy network p_ρ: Start with p_σ, improve by reinforcement learning from self-play (cf. slide 44).
- Value network v_θ: Supervised learning, training data generated by self-play using p_σ (cf. slide 44).

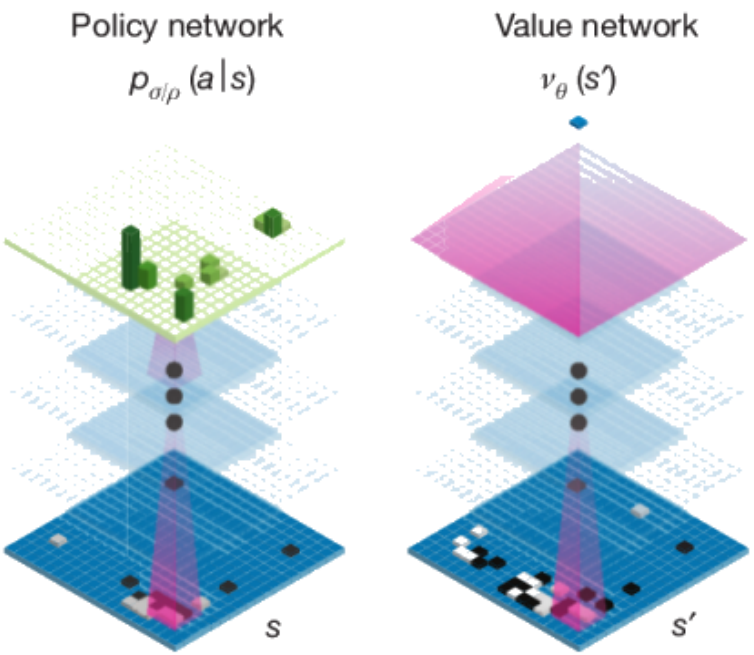
MCTS in AlphaGo

Illustration: (taken from [Silver *et al.* (2016)])



- SL policy network p_σ: Action choice bias (along with average value Q) within the tree ("P", gets smaller to "$u(P)$" with number of visits).
- Rollout policy p_π: Action choice in rollouts.
- Value network v_θ: Used to evaluate leaf states s, in weighted linear sum with the value returned by a rollout on s.
- RL policy network p_ρ: Not used here (used only to learn v_θ).

Neural Networks in AlphaGo



Neural network architecture. The input to the policy network is a $19 \times 19 \times 48$ image stack consisting of 48 feature planes. The first hidden layer zero pads the input into a 23×23 image, then convolves k filters of kernel size 5×5 with stride 1 with the input image and applies a rectifier nonlinearity. Each of the subsequent hidden layers 2 to 12 zero pads the respective previous hidden layer into a 21×21 image, then convolves k filters of kernel size 3×3 with stride 1, again followed by a rectifier nonlinearity. The final layer convolves 1 filter of kernel size 1×1 with stride 1, with a different bias for each position, and applies a softmax function. The match version of AlphaGo used $k = 192$ filters;

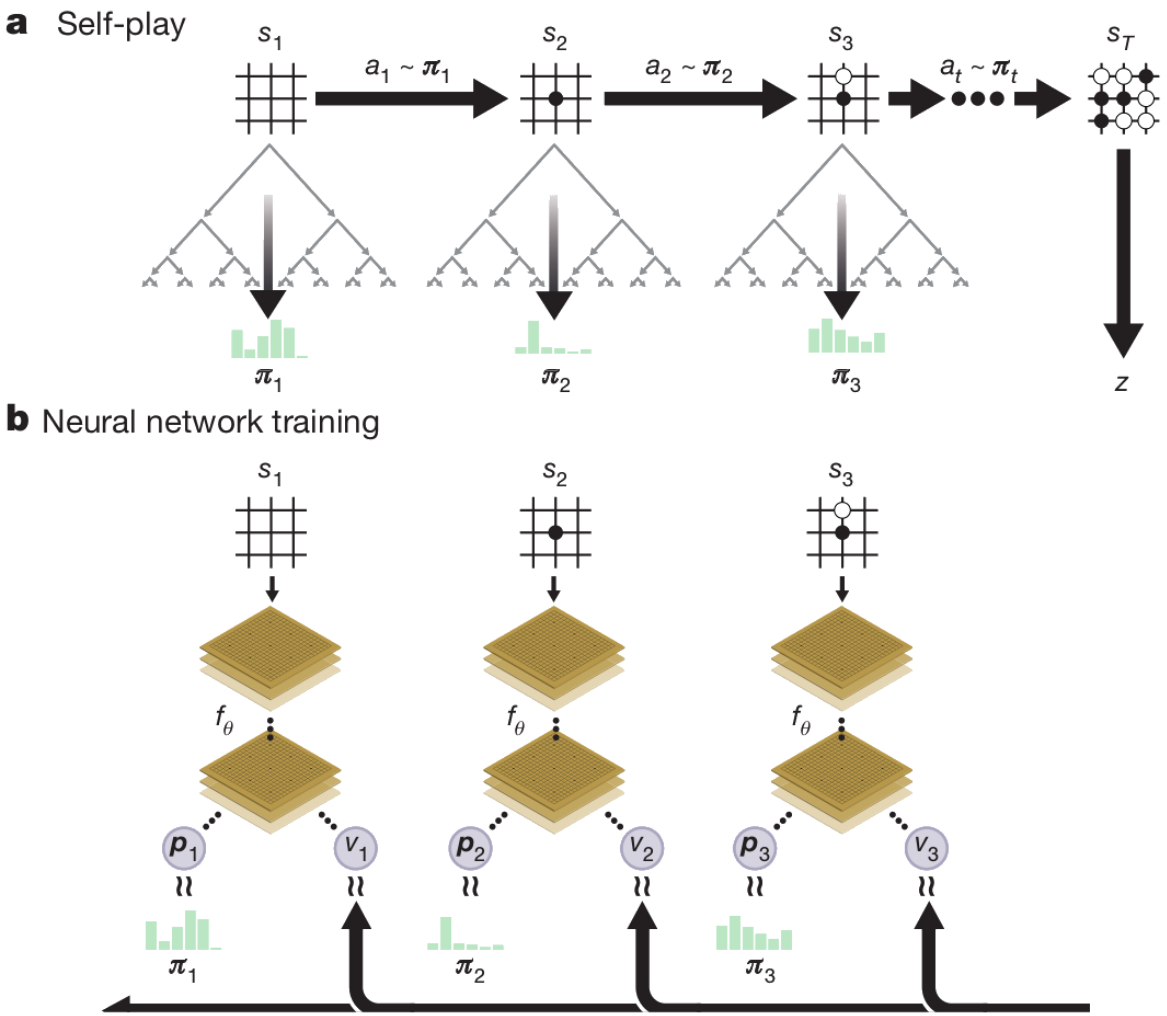
The value network similarly uses many convolutional layers with parameters θ, but outputs a scalar value $v_\theta(s')$ that predicts the expected outcome in position s'.

(Illustration and text taken from [Silver *et al.* (2016)])

- Architecture from image classification: convolutional NN.
- "Image" = game board, multiple feature planes encoding game rules (stone liberties etc.) visually.
- Size "small" compared to recent results in image classification: Work done in 2014–2016, leveraging NN architecture of that time. This changes next ...

Learning in AlphaGo Zero

Illustration: (taken from [Silver *et al.* (2017)])



Self-play:

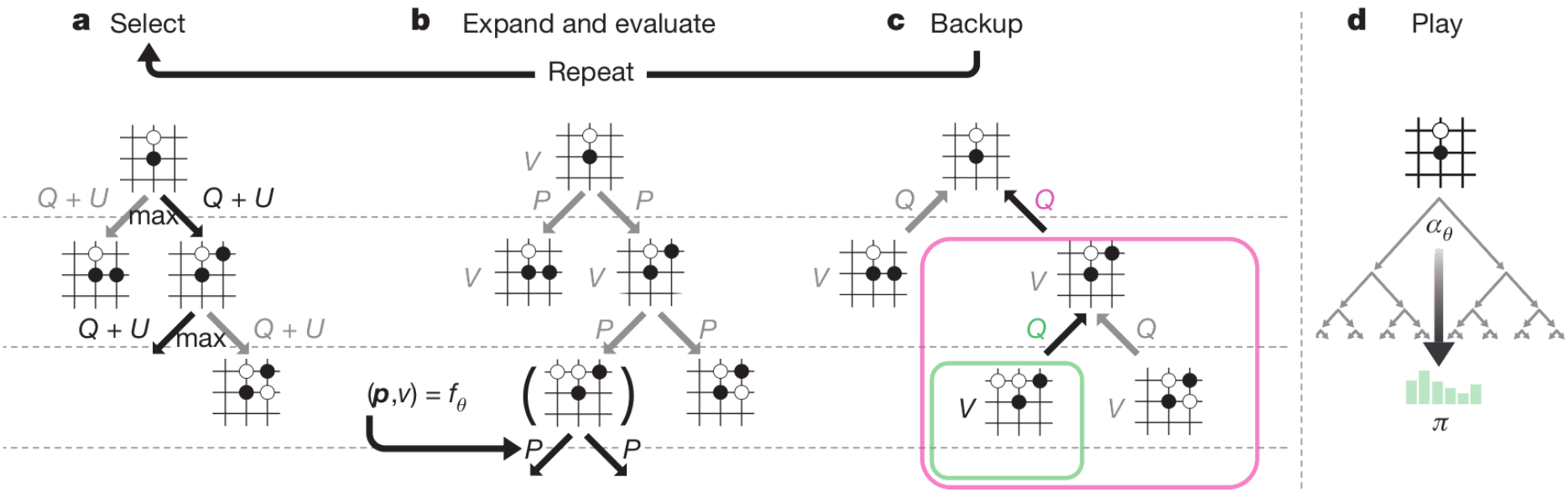
- Game: MCTS guided by current neural network.
- Learning: update weights to reduce error of v, and to move p closer to action π_i chosen by MCTS.
- MCTS controls exploitation vs. exploration trade-off for reinforcement learning.

Single neural network f_θ:

- Output (p, v): move probabilities p, value v. $\rightarrow$ Probabilistic policy *and* evaluation function.
- Residual blocks [He *et al.* (2016)], much improved performance.

MCTS in AlphaGo Zero

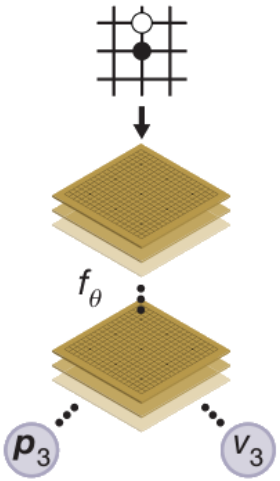
Illustration: (taken from [Silver *et al.* (2017)])



- Basically as in AlphaGo.
- Except: No rollouts! Leaf-state evaluation = NN output v.

  $\rightarrow$ Monte-Carlo tree search without "Monte-Carlo" :-) ... like a heuristic search with MCTS-style node-expansion strategy.

Neural Network (Singular!) in AlphaGo Zero



The input features s_t are processed by a residual tower that consists of a single convolutional block followed by either 19 or 39 residual blocks[4].

The convolutional block applies the following modules:
(1) A convolution of 256 filters of kernel size 3×3 with stride 1
(2) Batch normalization[18]
(3) A rectifier nonlinearity

Each residual block applies the following modules sequentially to its input:
(1) A convolution of 256 filters of kernel size 3×3 with stride 1
(2) Batch normalization
(3) A rectifier nonlinearity
(4) A convolution of 256 filters of kernel size 3×3 with stride 1
(5) Batch normalization
(6) A skip connection that adds the input to the block
(7) A rectifier nonlinearity

The output of the residual tower is passed into two separate 'heads' for computing the policy and value. The policy head applies the following modules:
(1) A convolution of 2 filters of kernel size 1×1 with stride 1
(2) Batch normalization
(3) A rectifier nonlinearity
(4) A fully connected linear layer that outputs a vector of size $19^2 + 1 = 362$, corresponding to logit probabilities for all intersections and the pass move

The value head applies the following modules:
(1) A convolution of 1 filter of kernel size 1×1 with stride 1
(2) Batch normalization
(3) A rectifier nonlinearity
(4) A fully connected linear layer to a hidden layer of size 256
(5) A rectifier nonlinearity
(6) A fully connected linear layer to a scalar
(7) A tanh nonlinearity outputting a scalar in the range $[-1, 1]$

The overall network depth, in the 20- or 40-block network, is 39 or 79 parameterized layers, respectively, for the residual tower, plus an additional 2 layers for the policy head and 3 layers for the value head.

(Illustration and text taken from [Silver *et al.* (2016)])

- Architecture from more recent image classification works: now including residual blocks! $\rightarrow$ Enables much deeper network.
- Evaluation function and policy are just different "heads" to the same network.
- 19 vs. 39 residual blocks: 19 in an initial system version; 39 in the final version.

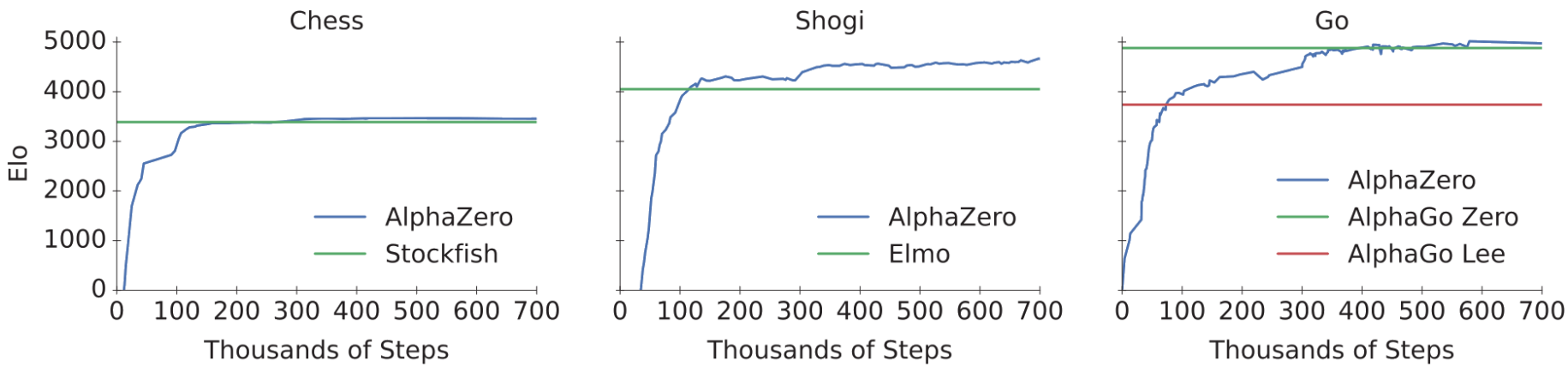
$\rightarrow$ Keys to success: 1. Integration of reinforcement learning with MCTS. 2. Leveraging recent NN architectures from image classification, in particular residual blocks.

Changes in AlphaZero

AlphaGo Zero with relatively small changes . . .

- No symmetry handling (applies only to Go).
- Various smallish details/parameters in configurations.

. . . generalizes very well to chess and shogi:



Across AlphaGo/Zero: NN Input/Output Representation

AlphaZero:

- Input: "$N \times N \times (MT + L)$ image stack ... each set of planes represents the board position at a time-step ...".
- Output: "move in chess ... $8 \times 8 \times 73$ stack of planes ... first 56 planes represent possible queen moves for any piece ...".

$\rightarrow$ Image-like representation of both, game state and moves. Crucial for success of NN methods originating in image classification.

AlphaGo Zero and AlphaGo:

- Similar.
- Just easier for Go than for chess and shogi.

AlphaGo/Zero: Conclusion

Amazing progress!

→ "Search & Learn NN" seems a great recipe for AI action decisions.

Rapidly growing research trend! (Including my own research group)

Limitations: Beyond (board) games?

1. How well does this generalize to problems with no image-like structure? With incomplete information? Multiple agents? Where random play does not produce interesting data?
2. How to find "the right" hyperparameters (NN architecture etc)? Especially without large teams of full-time researchers and "5000 first-generation tensor processing units (TPUs)"?
3. In many problems, generating training data is not easy ("mild" example: autonomous driving; extreme example: NASA).

Beyond Board Games: Progress on Limitation 1

StarCraft: [Vinyals *et al.* (2019)]

- Game map ($\approx$ board) *plus* other lists/sets data (unit attributes etc).
  $\rightarrow$ "Scatter connections" in NN architecture.
- Incomplete information, need move history to judge game state.
  $\rightarrow$ LSTM NN architecture (as in natural language processing).
- Multiple collaborative and competitive agents.
  $\rightarrow$ Multi-agent self-play learning (each agent separate NN).
- Random play does not produce interesting strategies (hence self-play reinforcement learning insufficient on its own).
  $\rightarrow$ Human knowledge (supervised learning and more).
- Generalization beyond StarCraft unclear at this point.

Beyond Board Games: Progress on Limitation 1

Dota 2: [Berner *et al.* (2019)]

- Incomplete information, need move history to judge game state.
  → LSTM NN architecture.
- Multiple collaborative and competitive agents.
  → Multi-agent self-play learning.
- Random play slow to produce interesting strategies.
  → Reward shaping.
- LOTS of game-specific human engineering in system.
  → Observation&action space, learning process design, hyperparameters, reward shaping, etc.
- MASSIVE hardware & training process.
  → Thousands of GPUs, 10 months.
- Generalization beyond Dota unclear at this point.

Summary

- Games (2-player turn-taking zero-sum discrete and finite games) can be understood as a simple extension of classical search problems.
- Each player tries to reach a terminal state with the best possible utility (maximal vs. minimal).
- Minimax searches the game depth-first, max'ing and min'ing at the respective turns of each player. It yields perfect play, but takes time $O(b^d)$ where b is the branching factor and d the search depth.
- Except in trivial games (Tic-Tac-Toe), Minimax needs a depth limit and apply an evaluation function to estimate the value of the cut-off states.
- Alpha-beta search remembers the best values achieved for each player elsewhere in the tree already, and prunes out sub-trees that won't be reached in the game.
- Monte-Carlo tree search (MCTS) samples game branches, and averages the findings. AlphaGo/Zero uses neural networks to learn evaluation functions and approximate policies in MCTS.

Reading

- *Chapter 5: Adversarial Search*, Sections 5.1 – 5.4 [Russell and Norvig (2010)].

  Content: Section 5.1 corresponds to my "Introduction", Section 5.2 corresponds to my "Minimax Search", Section 5.3 corresponds to my "Alpha-Beta Search". I have tried to add some additional clarifying illustrations. RN gives many complementary explanations, nice as additional background reading.

  Section 5.4 corresponds to my "Evaluation Functions", but discusses additional aspects relating to narrowing the search and look-up from opening/termination databases. Nice as additional background reading.

  I suppose a discussion of MCTS and AlphaGo will be added to the next edition ...

References I

Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Christopher Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. *CoRR*, abs/1912.06680, 2019.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pages 770–778. IEEE Computer Society, 2016.

Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *Proceedings of the 17th European Conference on Machine Learning (ECML 2006)*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer-Verlag, 2006.

References II

Raghuram Ramanujan, Ashish Sabharwal, and Bart Selman. On adversarial search spaces and sampling-based planning. In Ronen I. Brafman, Hector Geffner, Jörg Hoffmann, and Henry A. Kautz, editors, *Proceedings of the 20th International Conference on Automated Planning and Scheduling (ICAPS'10)*, pages 242–245. AAAI Press, 2010.

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (Third Edition)*. Prentice-Hall, Englewood Cliffs, NJ, 2010.

David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–359, 2017.

References III

Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander Sasha Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom Le Paine, Çaglar Gülçehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy P. Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in starcraft II using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.