EECS 391
Intro to AI

Evaluation Functions, Stochastic Games, Other Approaches

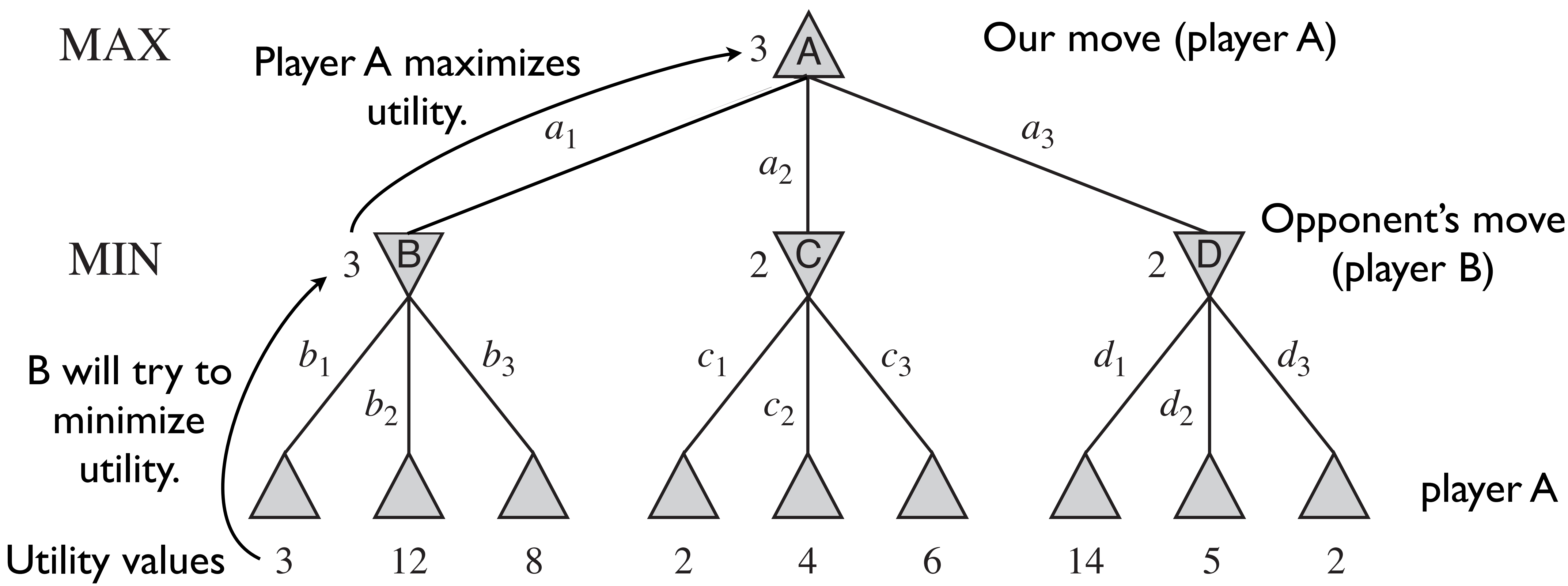
L7 Thu Sep 21, 2017

Imagine how great universities could be without all those human teachers

- *Quartz* series on “The Vanishing University”, a four-part series exploring the tech-driven future of higher education in America.
- *Jill Watson is the best damn teaching assistant you could ever want.*
- *“I got comments like ‘Mind blown’ and ‘I want to nominate Jill for outstanding TA award’,” says Ashok Goel, a computer-science professor at Georgia Tech.*
- Ashok Goel is also Jill Watson’s *creator.*
- Jill Watson is a chat bot.

Minimax: perfect play for deterministic 2-player games

- Idea: Chose move to position (or state) with highest minimax value.
  Best achievable payoff against optimal player
- minimax-val(n) =
  - Utility(n) if terminal state
  - max s ∈ result(n) if n is max node
  - min s ∈ result(n) if n is min node



Minimax algorithm

arg max returns the best action

transition model: state resulting from move a in state state

function MINIMAX-DECISION($state$) **returns** *an action*
  return $\arg\max_{a \in \text{ACTIONS}(s)}$ MIN-VALUE(RESULT($state$, a))

Available actions from state s

---

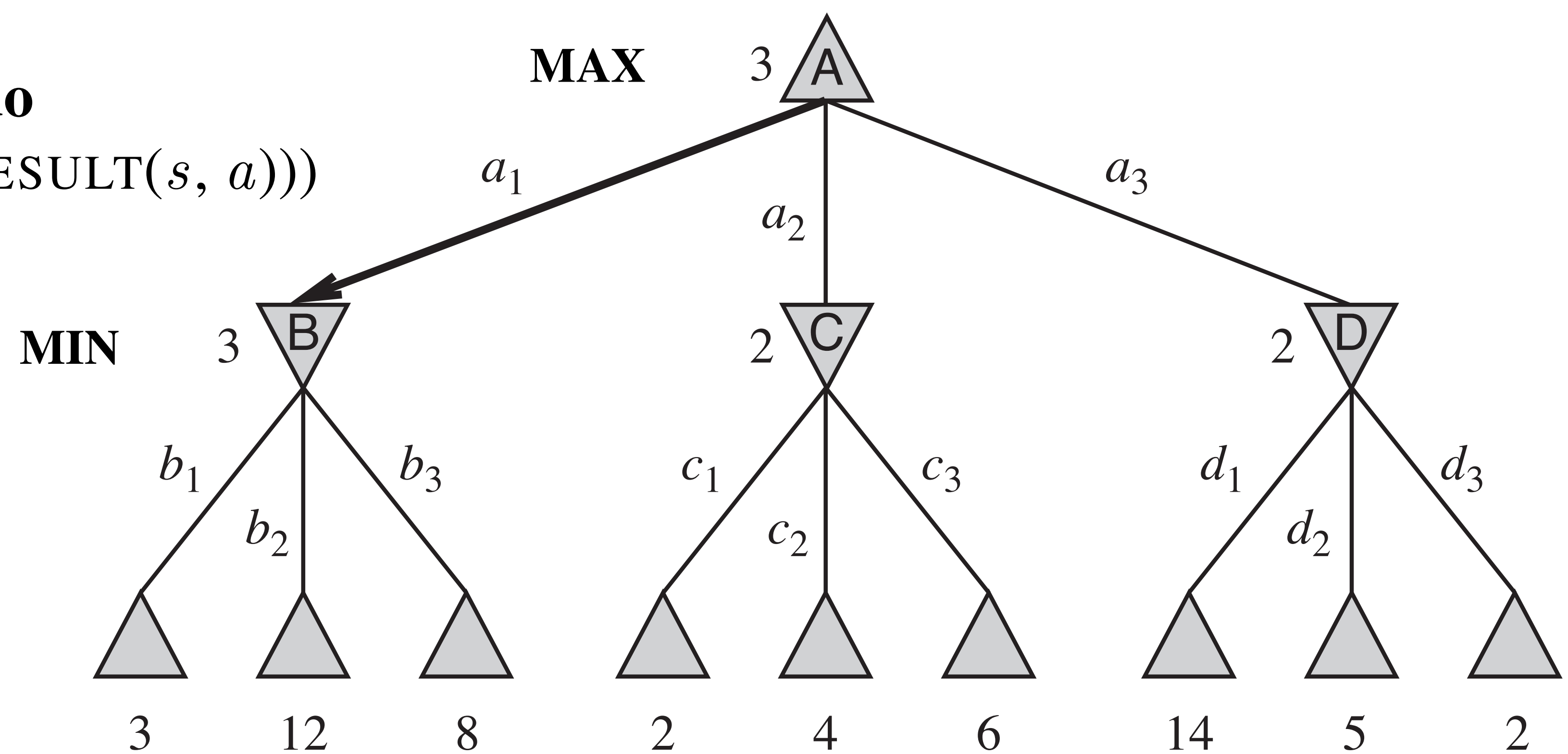
function MAX-VALUE($state$) **returns** *a utility value*
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

Utility function: numerical value of state (defined only for terminal states)

---

function MIN-VALUE($state$) **returns** *a utility value*
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow \infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s, a)))
  return v

Is the game over ?



Minimax algorithm

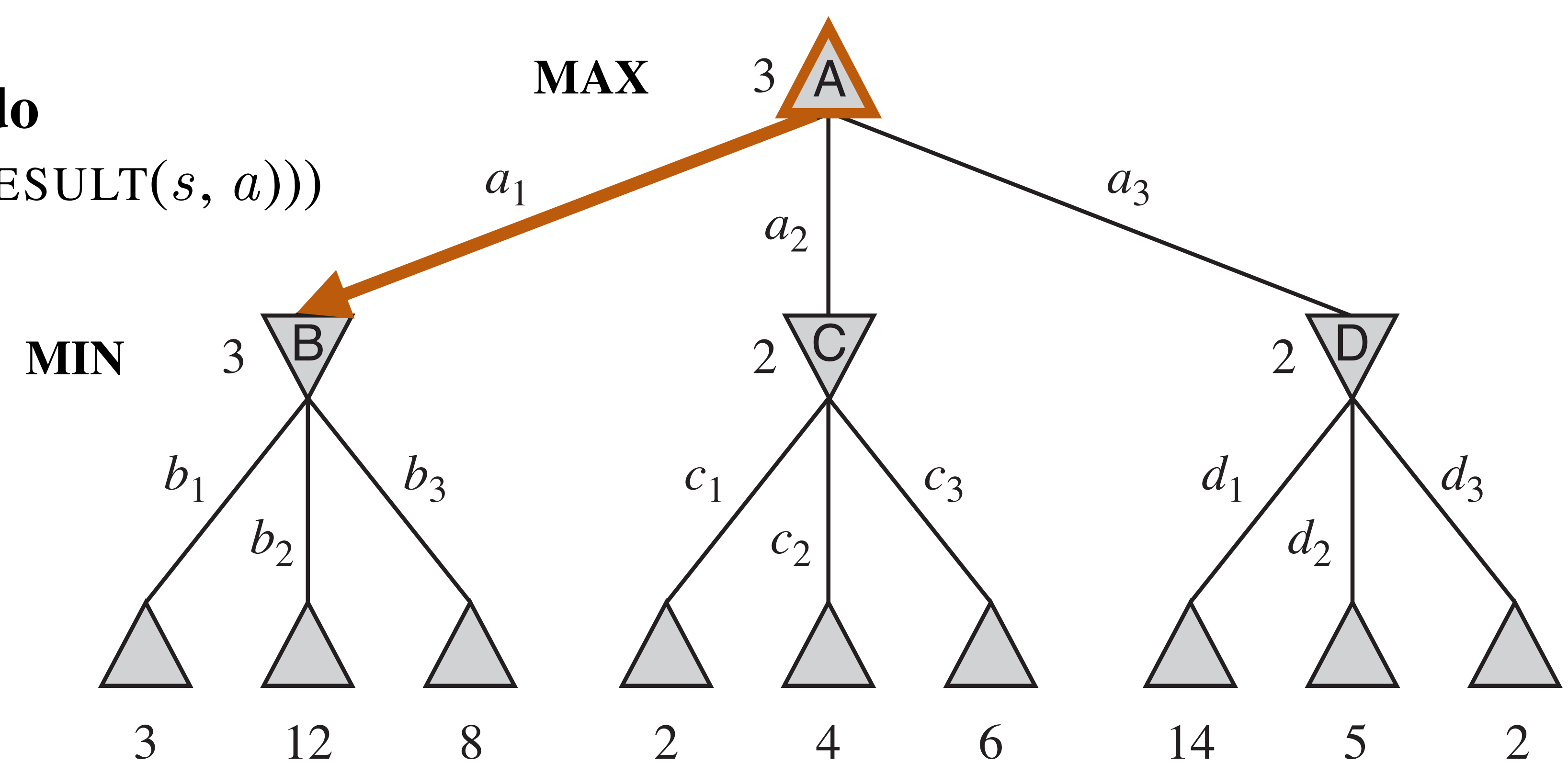
function MINIMAX-DECISION(*state*) **returns** *an action*
  return $\arg\max_{a \in \text{ACTIONS}(s)}$ MIN-VALUE(RESULT **(A, a₁)**)

---

function MAX-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

---

function MIN-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow \infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s, a)))
  return v



Minimax algorithm

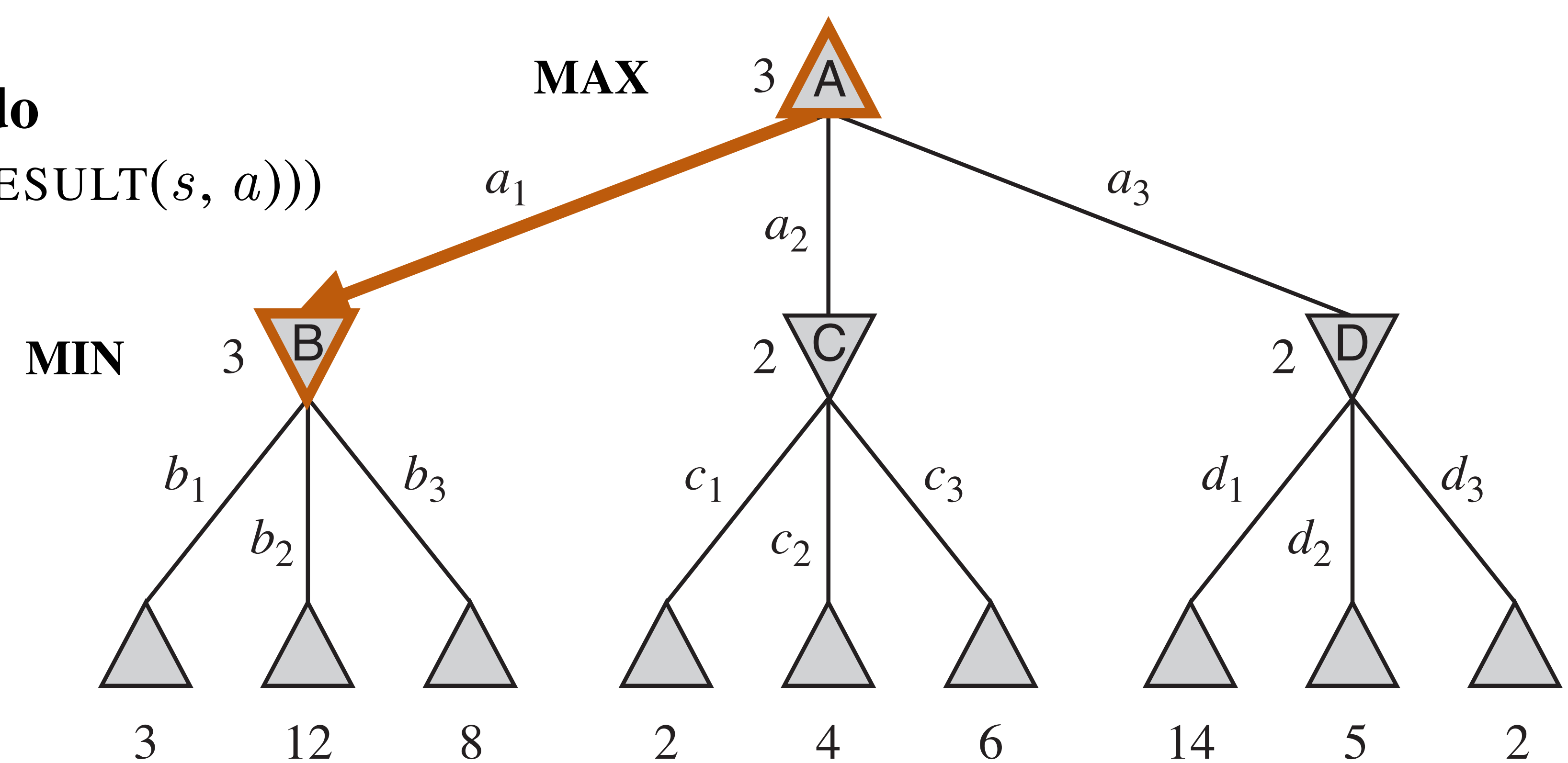
function MINIMAX-DECISION(*state*) **returns** *an action*
  return $\arg\max_{a \in \text{ACTIONS}(s)}$ MIN-VALUE(RESULT **(A, a₁)**)

---

function MAX-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

---

function MIN-VALUE(**B**) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow \infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s, a)))
  return v



Minimax algorithm

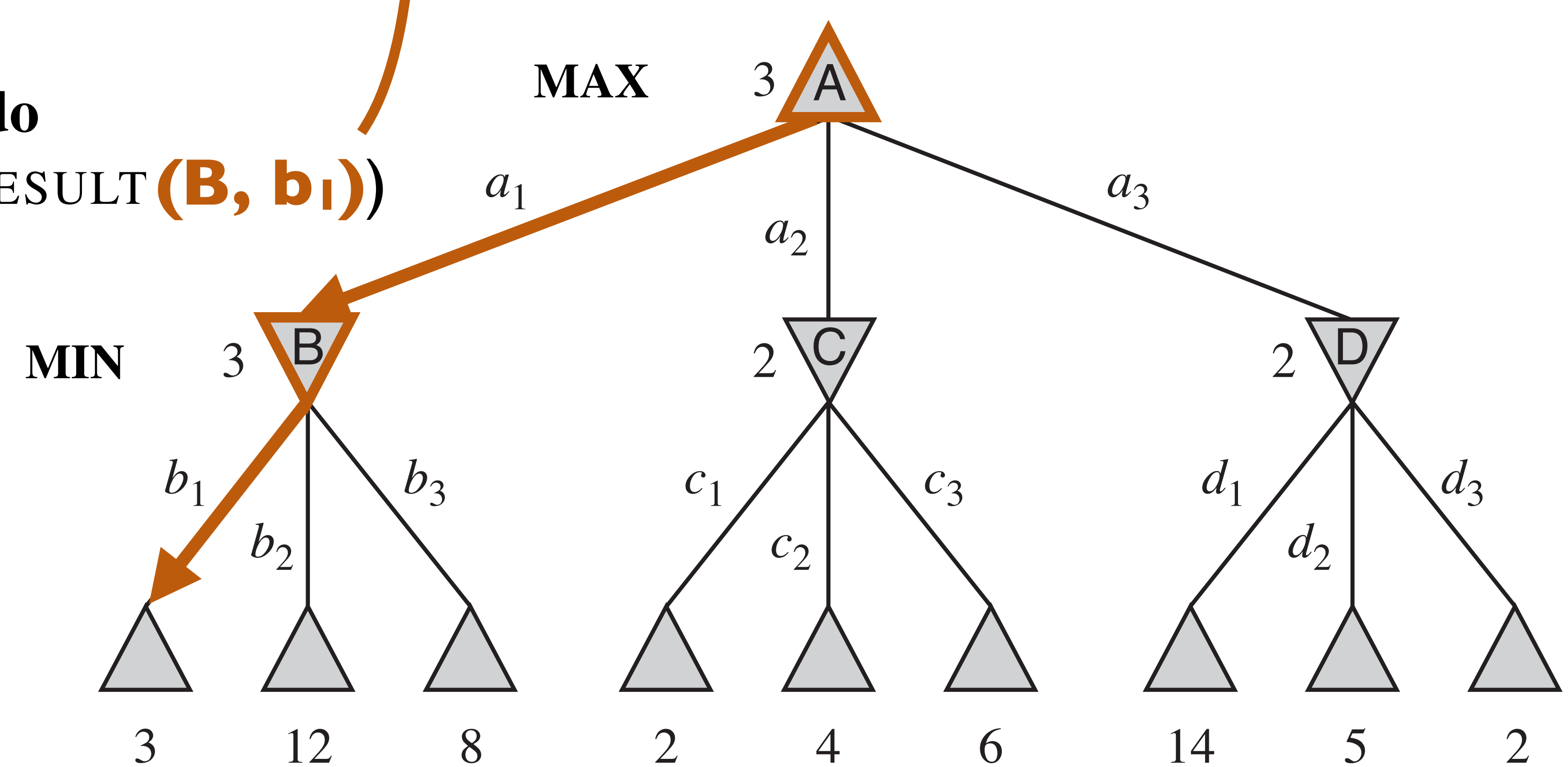
function MINIMAX-DECISION(*state*) **returns** *an action*
  return $\arg\max_{a \in \text{ACTIONS}(s)}$ MIN-VALUE(RESULT **(A, a1)**)

---

function MAX-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

---

function MIN-VALUE(**B**) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow \infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT **(B, b1)**)
  return v

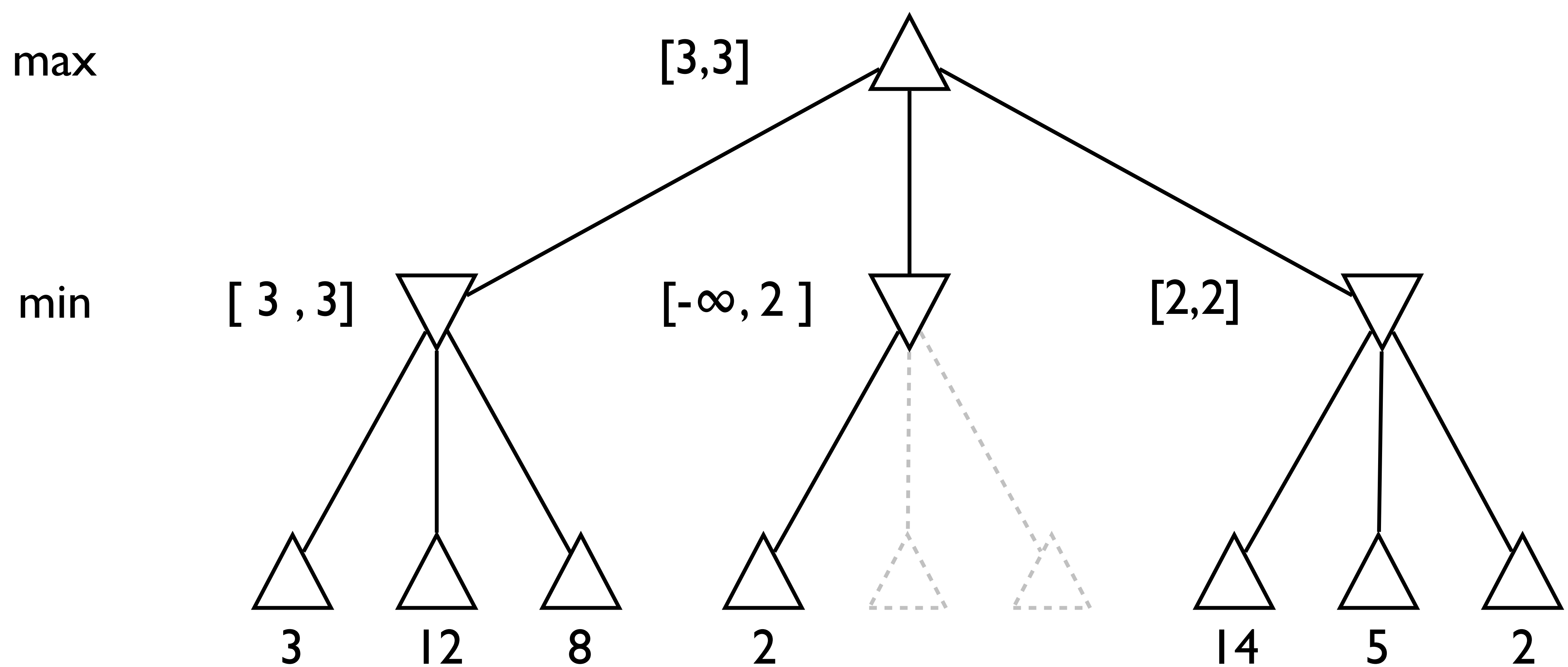


Alpha-beta pruning

- Idea: prune branches that can't influence final decision
- alpha-beta pruning keeps track of the range of possible values:

  α = max's best choice (highest val.) so far at any point along path
  β = min's best choice (lowest val.) so far at any point along path

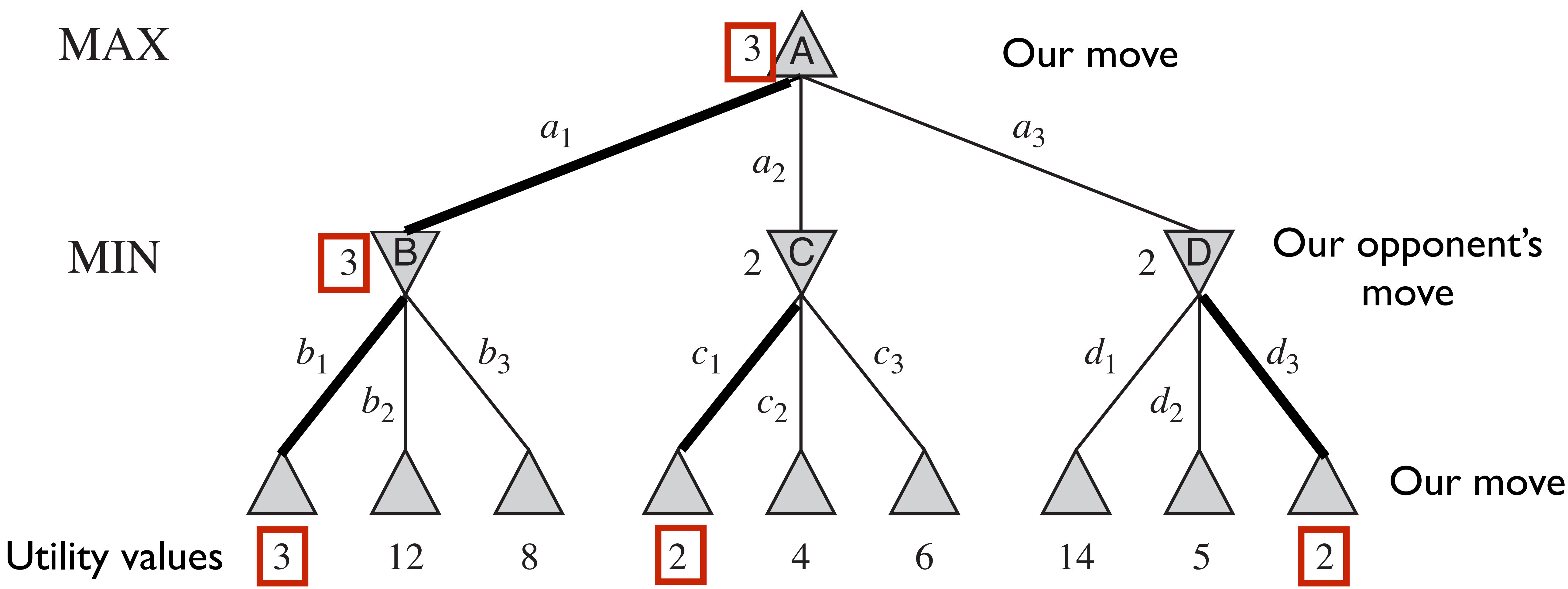
- alpha-beta doesn't search ("*prunes*"), branches that don't affect current estimates



Minimax: perfect play for deterministic 2-player games

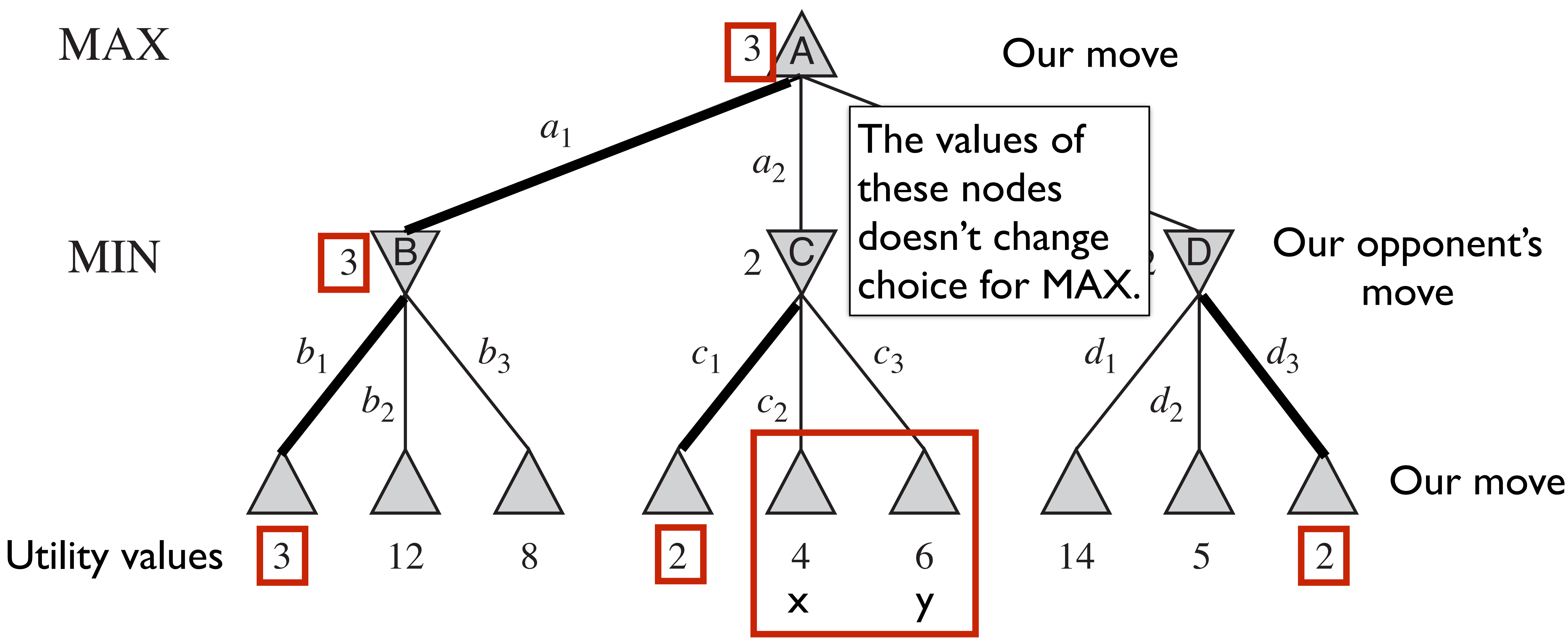
- minimax(n) =
  - Utility(n) if terminal state
  - max s ∈ result(n) if n is max node
  - min s ∈ result(n) if n is min node
- minimax(A) = max(min(3,12, 8), min(2, 4, 6), min(14, 5, 2))

Utilities are computed recursively from terminal states.



Minimax: perfect play for deterministic 2-player games

- minimax(n) =
  - Utility(n) if terminal state
  - max s ∈ result(n) if n is max node
  - min s ∈ result(n) if n is min node
- minimax(A) = max(min(3,12, 8), min(2, x, y), min(14, 5, 2))

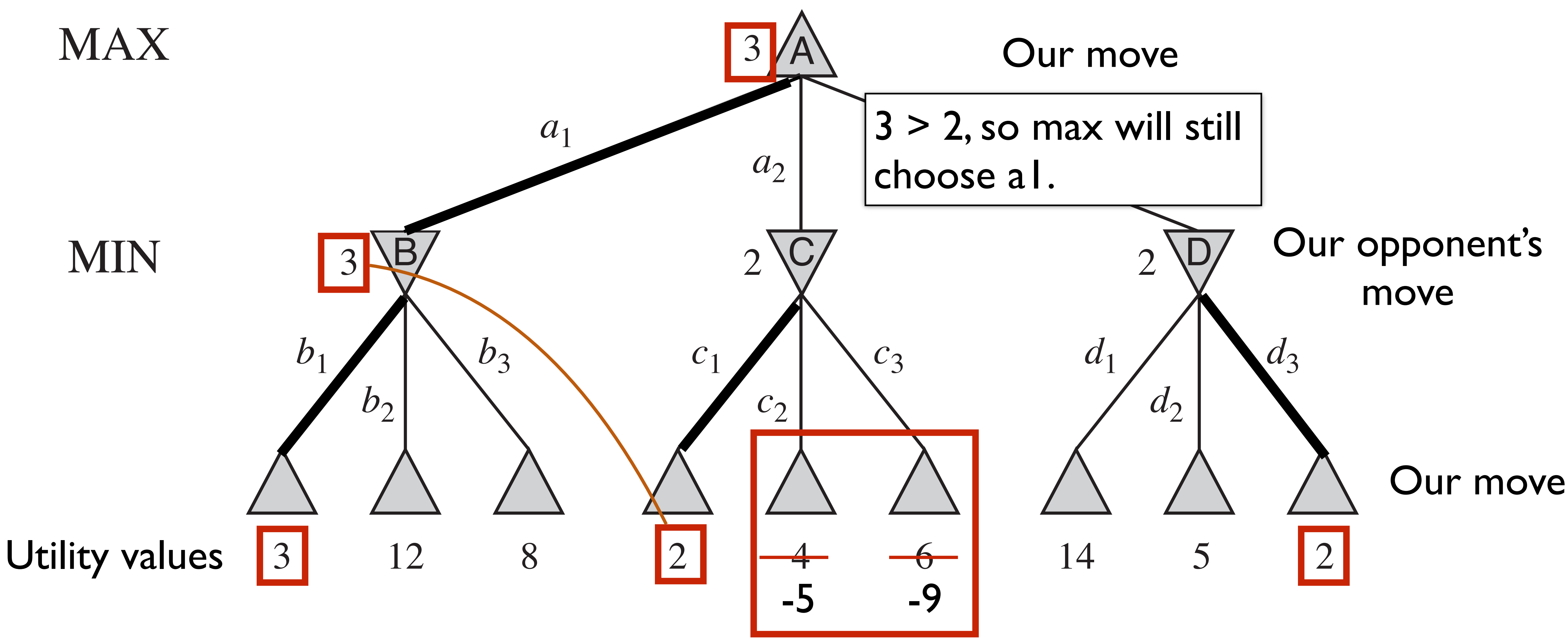


Minimax: perfect play for deterministic 2-player games

- minimax(n) =
  - Utility(n) — if terminal state
  - max $s \in$ result(n) — if n is max node
  - min $s \in$ result(n) — if n is min node
- minimax(A) = max(min(3,12, 8), min(2, -5,-9), min(14, 5, 2))

  = -9

  but max(3, -9, 2) = 3 $\Rightarrow$ a1

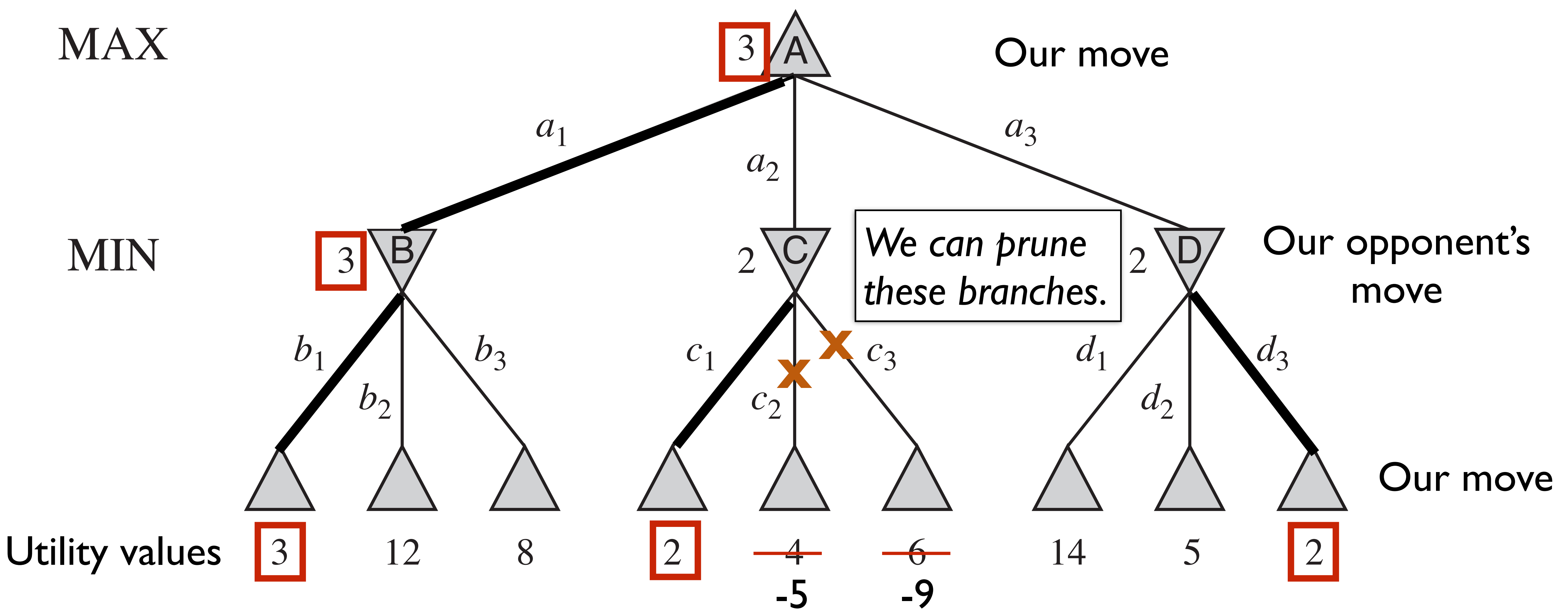


Minimax: perfect play for deterministic 2-player games

- minimax(n) =
  - Utility(n) if terminal state
  - max s ∈ result(n) if n is max node
  - min s ∈ result(n) if n is min node
- minimax(A) = max(min(3,12, 8), min(2, -5,-9), min(14, 5, 2))

= -9

but max(3, -9, 2) = 3 ⇒ a1



Alpha-Beta Search Algorithm

Alpha-beta pruning is a modification of the standard mini-max algorithm.

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the *action* in ACTIONS(*state*) with value v

---

function MAX-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

It cuts off the search, when the result can't change.

function MIN-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Alpha-Beta Search Algorithm

Alpha-beta pruning is a modification of the standard mini-max algorithm.

function ALPHA-BETA-SEARCH($state$) **returns** an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the $action$ in ACTIONS($state$) with value v

---

function MAX-VALUE($state, \alpha, \beta$) **returns** a $utility$ $value$
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

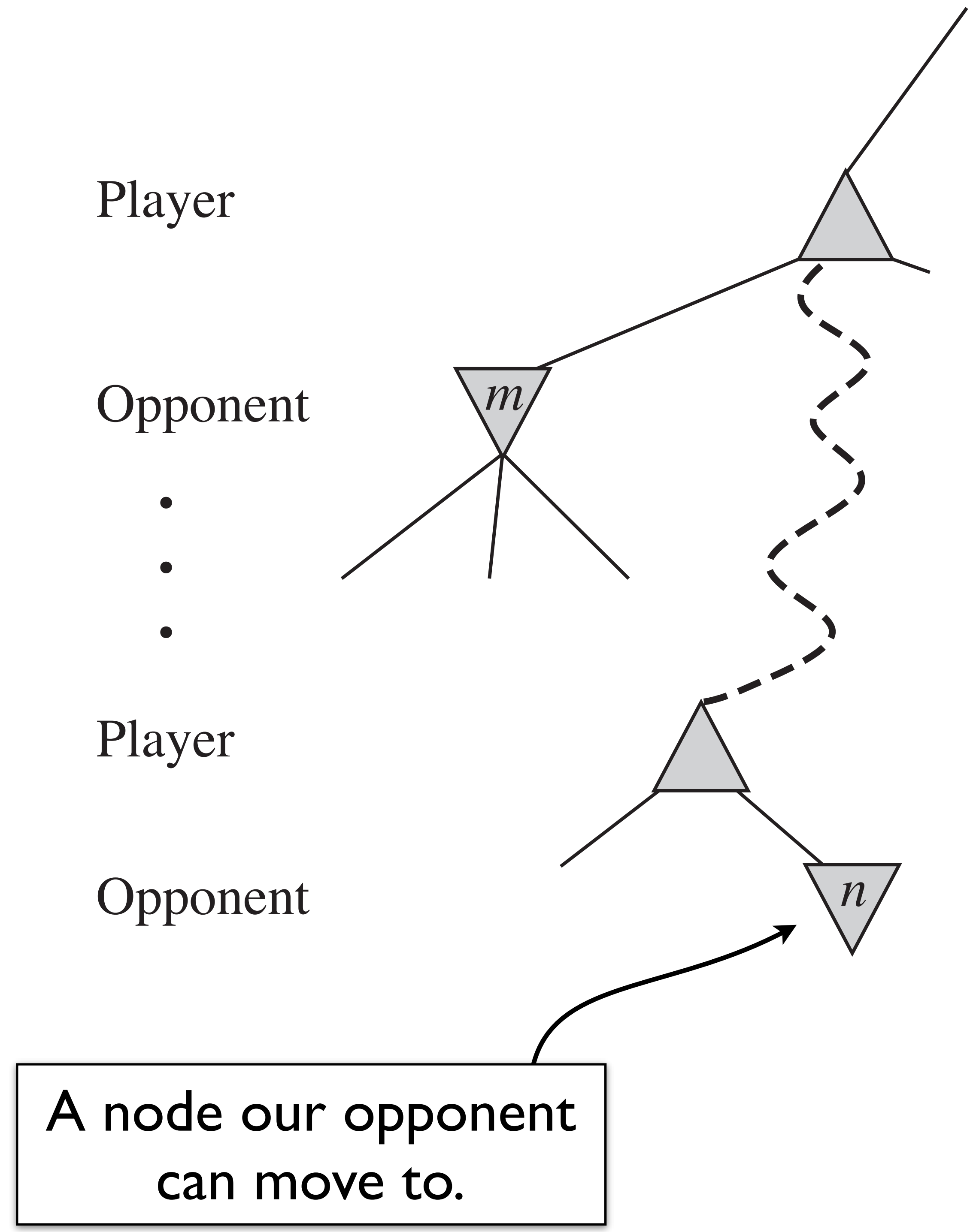
---

function MIN-VALUE($state, \alpha, \beta$) **returns** a $utility$ $value$
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Otherwise, it updates the best value found so far.

More properties of alpha-beta pruning

- applies to tree of any depth
- can prune entire sub-trees, not just leaves
- Idea:
  - If node n has greater value than node m, then
  - *node n would never be reached*
  - because our Opponent (the min player) would have to choose node m to be optimal
- That means the entire branch can be pruned once we find such a node.

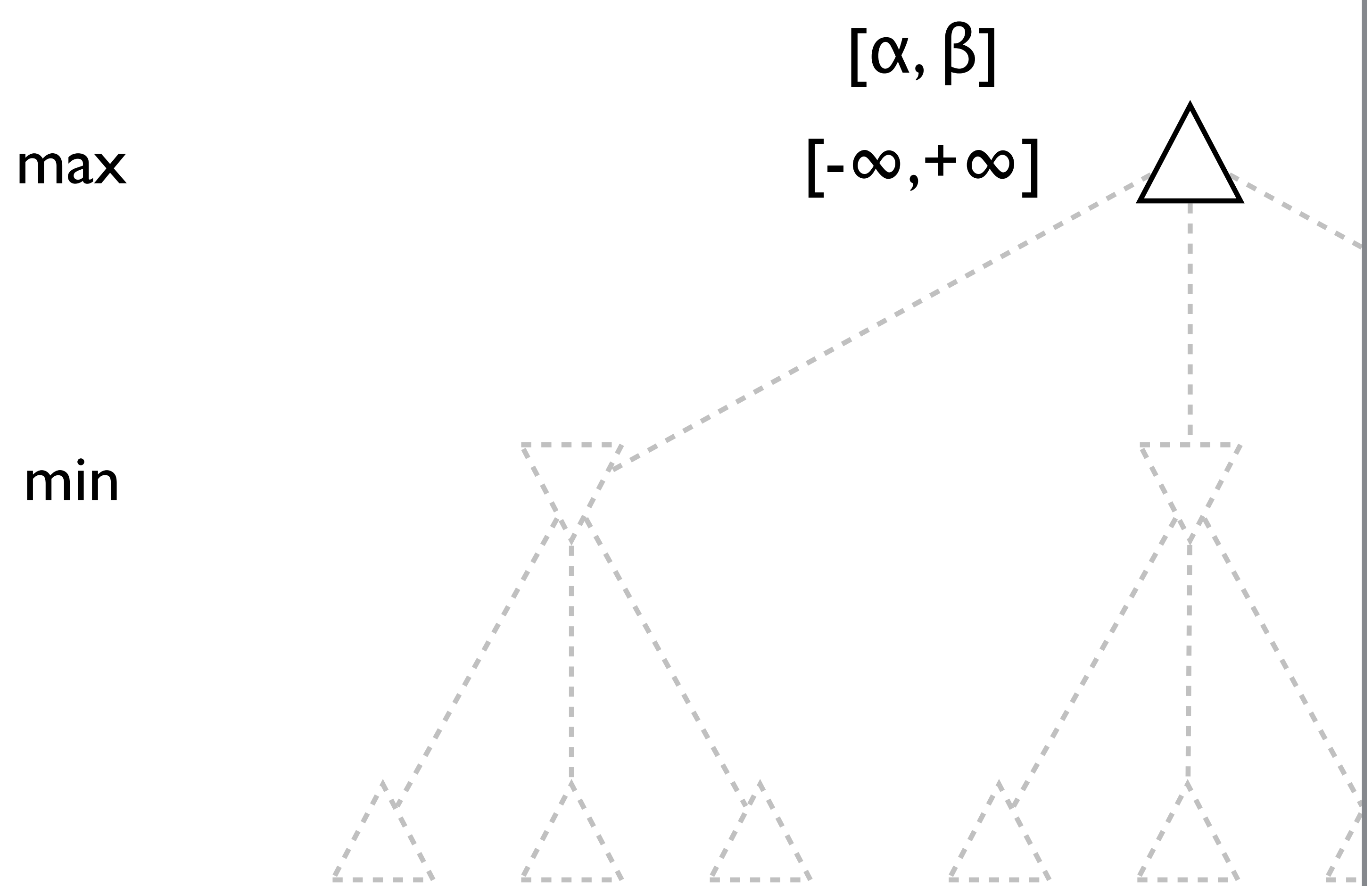


Alpha-beta pruning

- Idea: prune branches that can't influence final decision
- alpha-beta pruning keeps track of the range of possible values:

  α = max's best choice (highest val.) so far at any point along path
  β = min's best choice (lowest val.) so far at any point along path

- alpha-beta doesn't search: it "*prunes*" branches that don't affect current estimates



function ALPHA-BETA-SEARCH($state$) **returns** an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the $action$ in ACTIONS($state$) with value v

---

function MAX-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

---

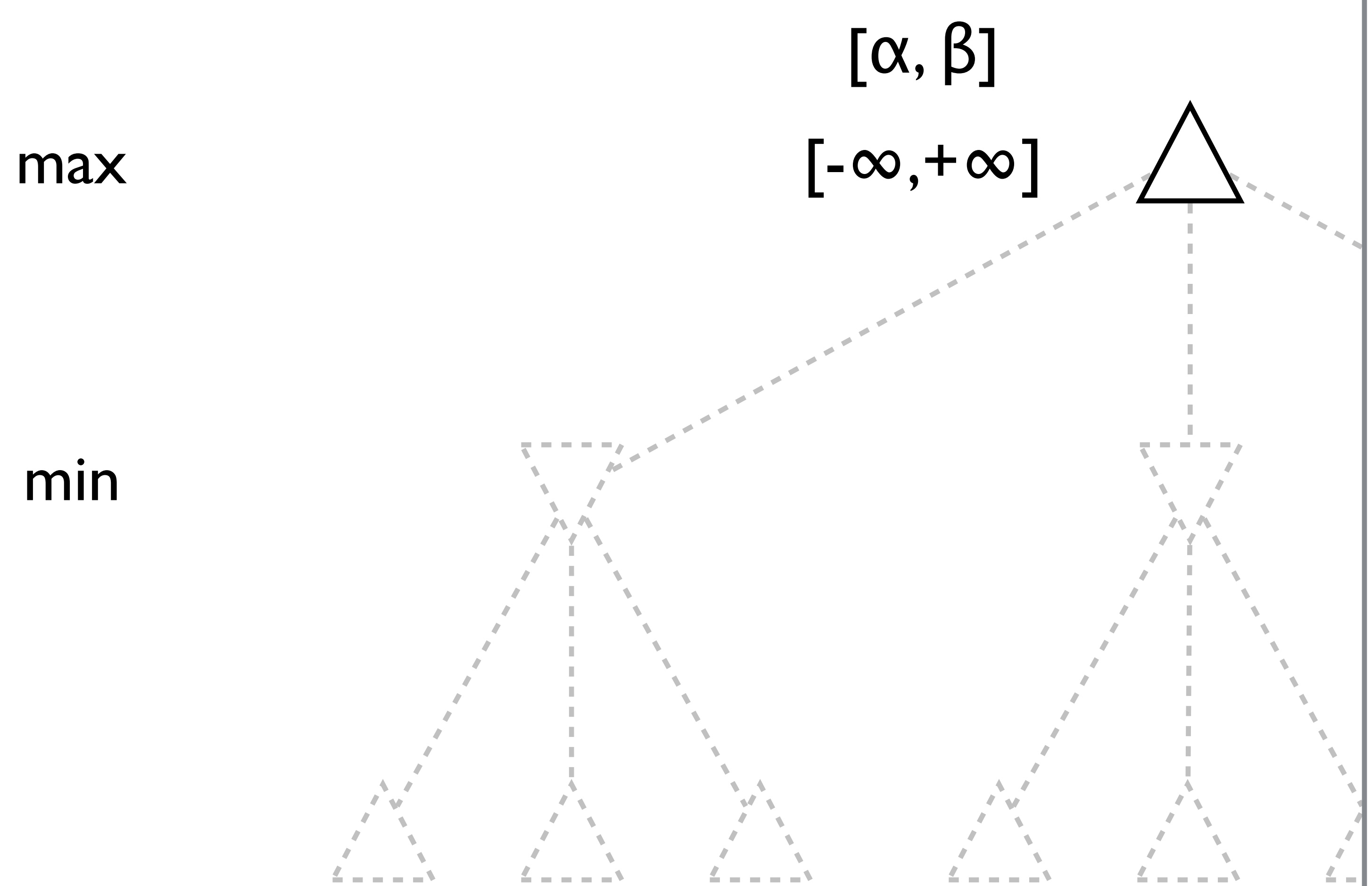
function MIN-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Alpha-beta pruning

- Idea: prune branches that can't influence final decision
- alpha-beta pruning keeps track of the range of possible values:

  α = max's best choice (highest val.) so far at any point along path
  β = min's best choice (lowest val.) so far at any point along path

- alpha-beta doesn't search: it "*prunes*" branches that don't affect current estimates



function ALPHA-BETA-SEARCH($state$) **returns** an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the $action$ in ACTIONS($state$) with value v

---

function MAX-VALUE($state, \alpha, \beta$) **returns** a $utility$ $value$
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

---

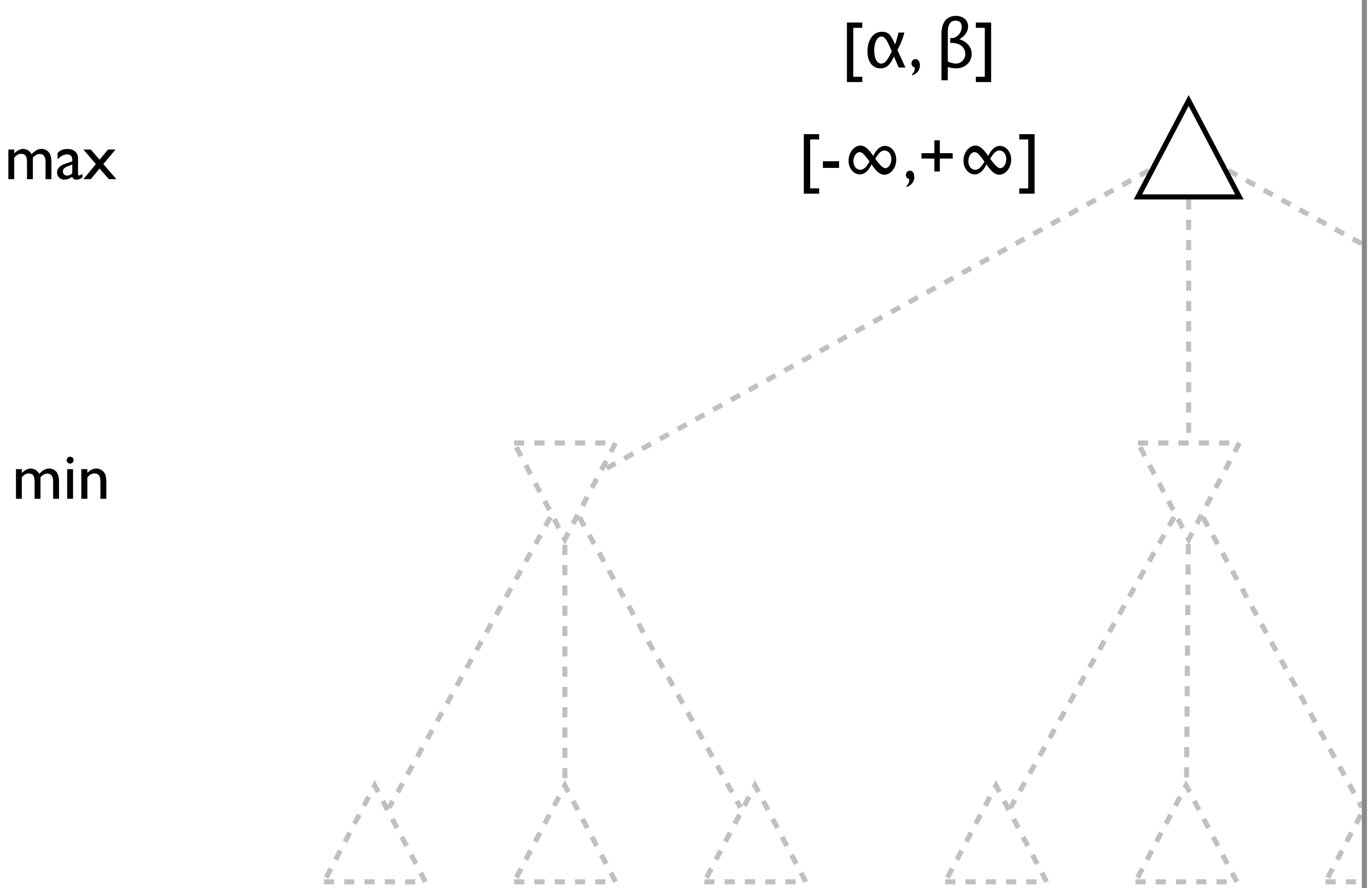
function MIN-VALUE($state, \alpha, \beta$) **returns** a $utility$ $value$
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Alpha-beta pruning

- Idea: prune branches that can't influence final decision
- alpha-beta pruning keeps track of the range of possible values:

  α = max's best choice (highest val.) so far at any point along path
  β = min's best choice (lowest val.) so far at any point along path

- alpha-beta doesn't search: it "*prunes*" branches that don't affect current estimates



function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the *action* in ACTIONS(*state*) with value v

function MAX-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

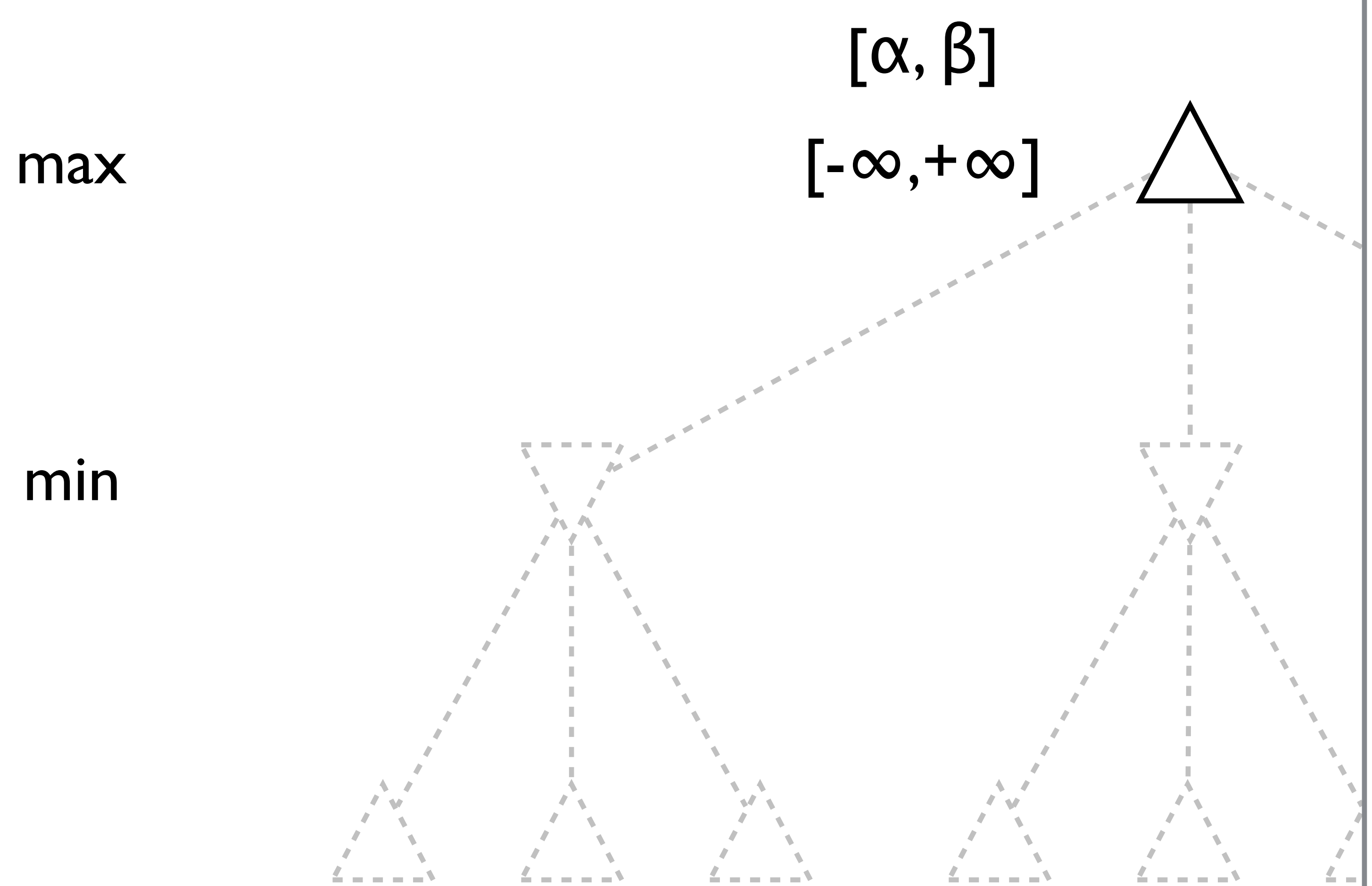
function MIN-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Alpha-beta pruning

- Idea: prune branches that can't influence final decision
- alpha-beta pruning keeps track of the range of possible values:

  α = max's best choice (highest val.) so far at any point along path
  β = min's best choice (lowest val.) so far at any point along path

- alpha-beta doesn't search: it "*prunes*" branches that don't affect current estimates



function ALPHA-BETA-SEARCH($state$) **returns** an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the $action$ in ACTIONS($state$) with value v

function MAX-VALUE($state, \alpha, \beta$) **returns** a $utility$ $value$
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

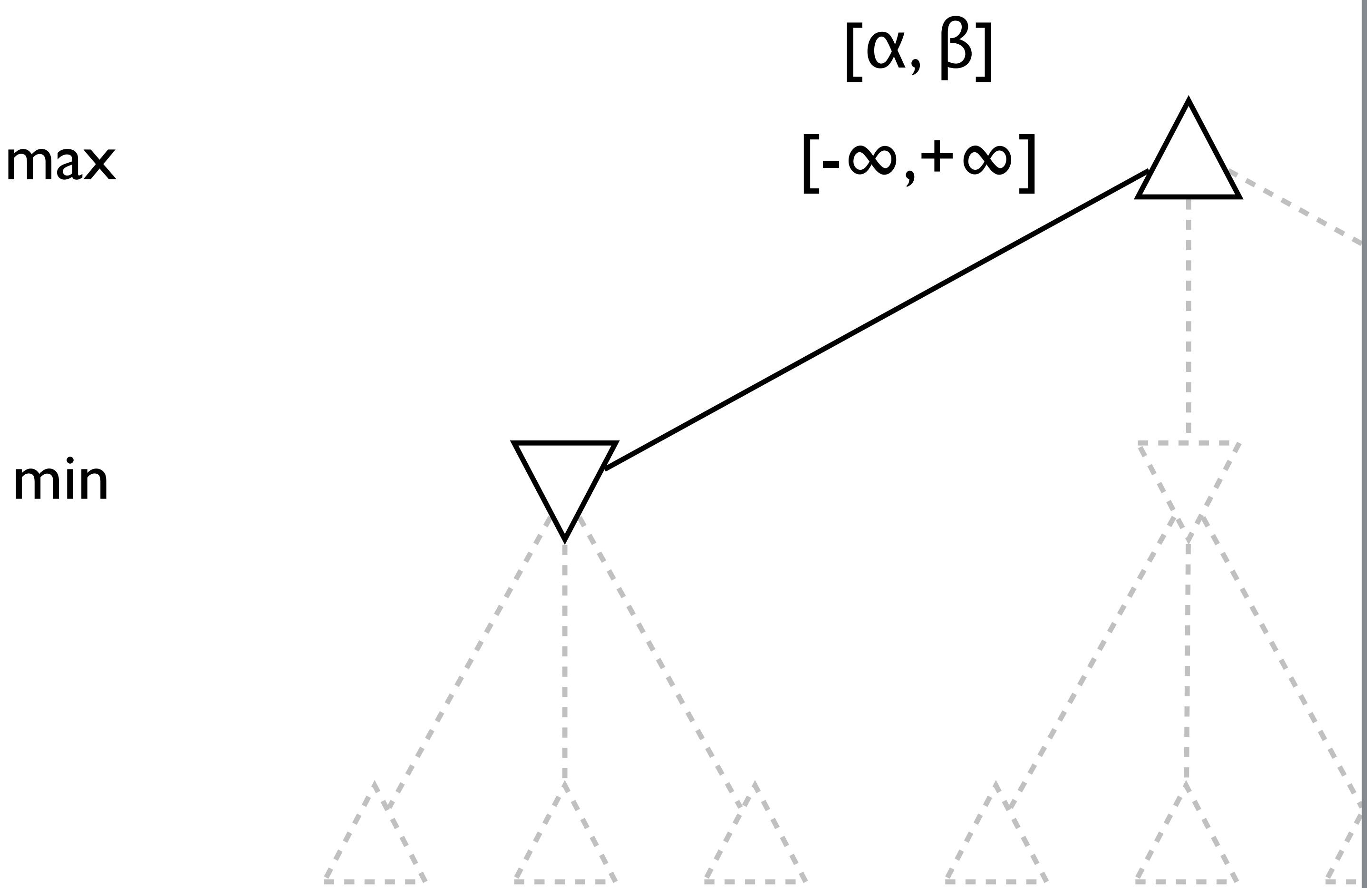
function MIN-VALUE($state, \alpha, \beta$) **returns** a $utility$ $value$
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Alpha-beta pruning

- Idea: prune branches that can't influence final decision
- alpha-beta pruning keeps track of the range of possible values:

  α = max's best choice (highest val.) so far at any point along path
  β = min's best choice (lowest val.) so far at any point along path

- alpha-beta doesn't search: it "*prunes*" branches that don't affect current estimates



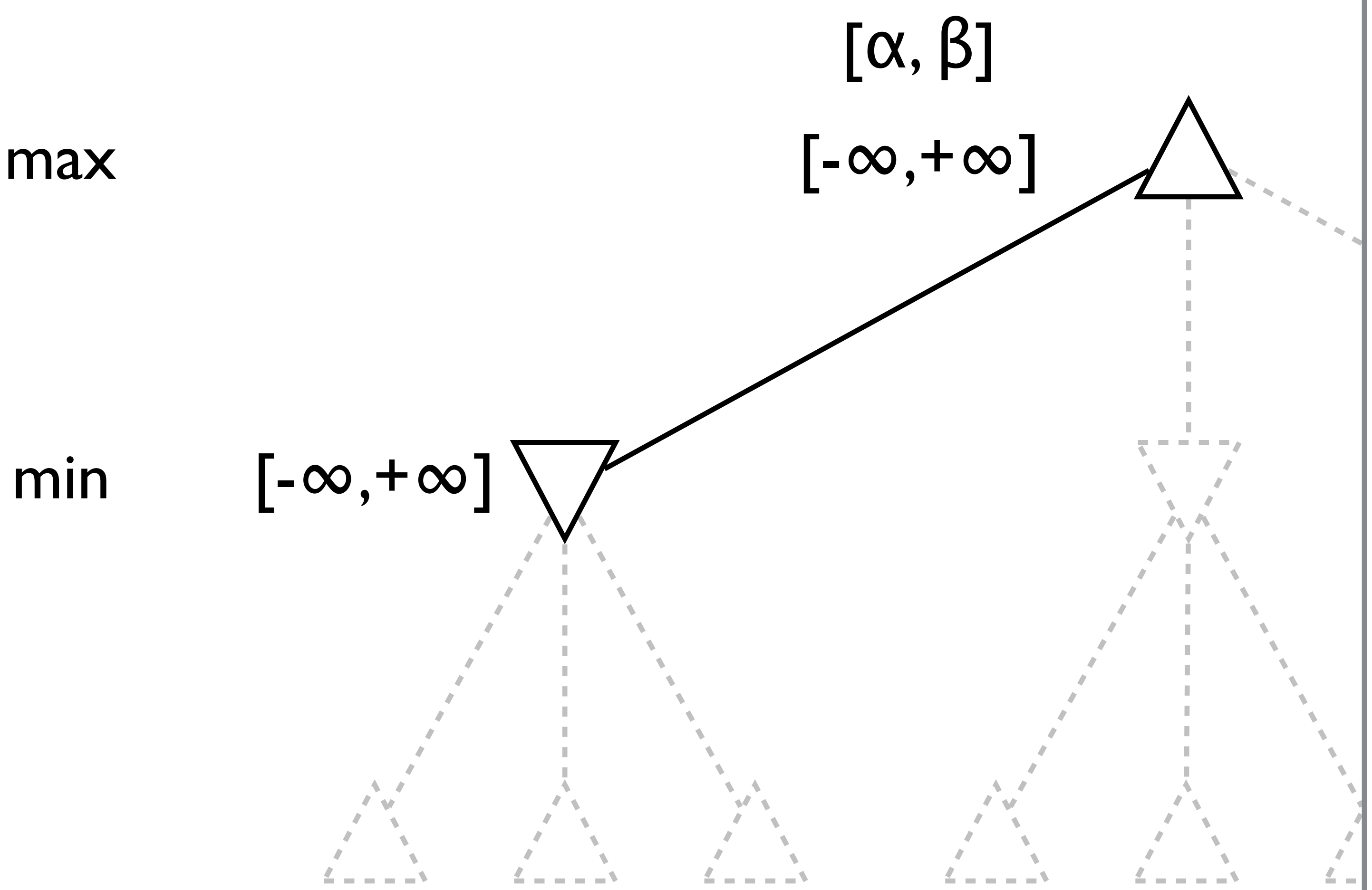

```
function ALPHA-BETA-SEARCH(state) returns an action
   v ← MAX-VALUE(state, −∞, +∞)
   return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for each a in ACTIONS(state) do
      v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
      if v ≥ β then return v
      α ← MAX(α, v)
   return v

function MIN-VALUE(state, α, β) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← +∞
   for each a in ACTIONS(state) do
      v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
      if v ≤ α then return v
      β ← MIN(β, v)
   return v
```

Alpha-beta pruning

- Idea: prune branches that can't influence final decision
- alpha-beta pruning keeps track of the range of possible values:

  α = max's best choice (highest val.) so far at any point along path
  β = min's best choice (lowest val.) so far at any point along path

- alpha-beta doesn't search: it "*prunes*" branches that don't affect current estimates



function ALPHA-BETA-SEARCH($state$) **returns** an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the $action$ in ACTIONS($state$) with value v

---

function MAX-VALUE($state, \alpha, \beta$) **returns** $a\ utility\ value$
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

---

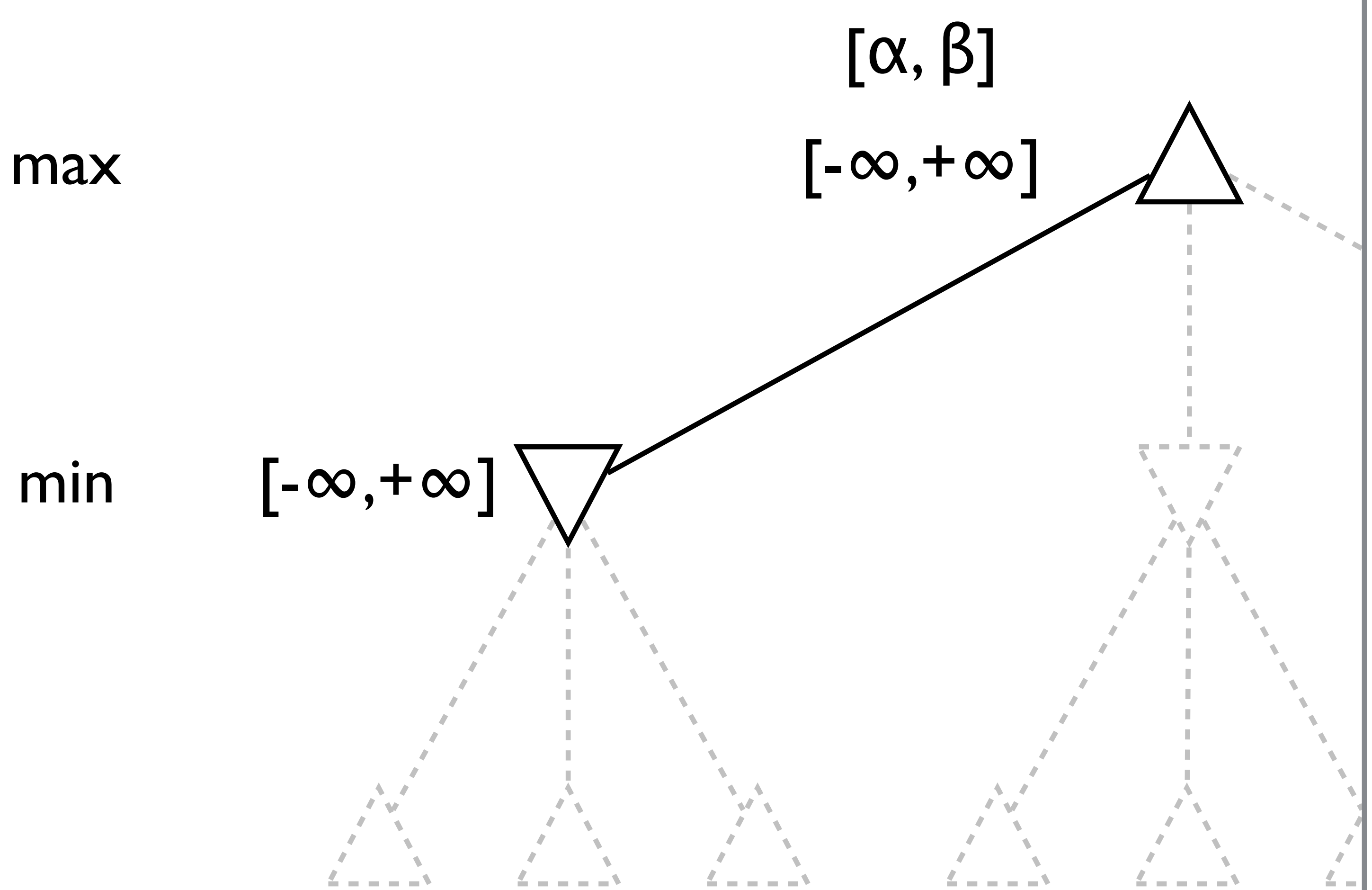
function MIN-VALUE($state, \alpha, \beta$) **returns** $a\ utility\ value$
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Alpha-beta pruning

- Idea: prune branches that can't influence final decision
- alpha-beta pruning keeps track of the range of possible values:

  α = max's best choice (highest val.) so far at any point along path
  β = min's best choice (lowest val.) so far at any point along path

- alpha-beta doesn't search: it "*prunes*" branches that don't affect current estimates



function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the *action* in ACTIONS(*state*) with value v

function MAX-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

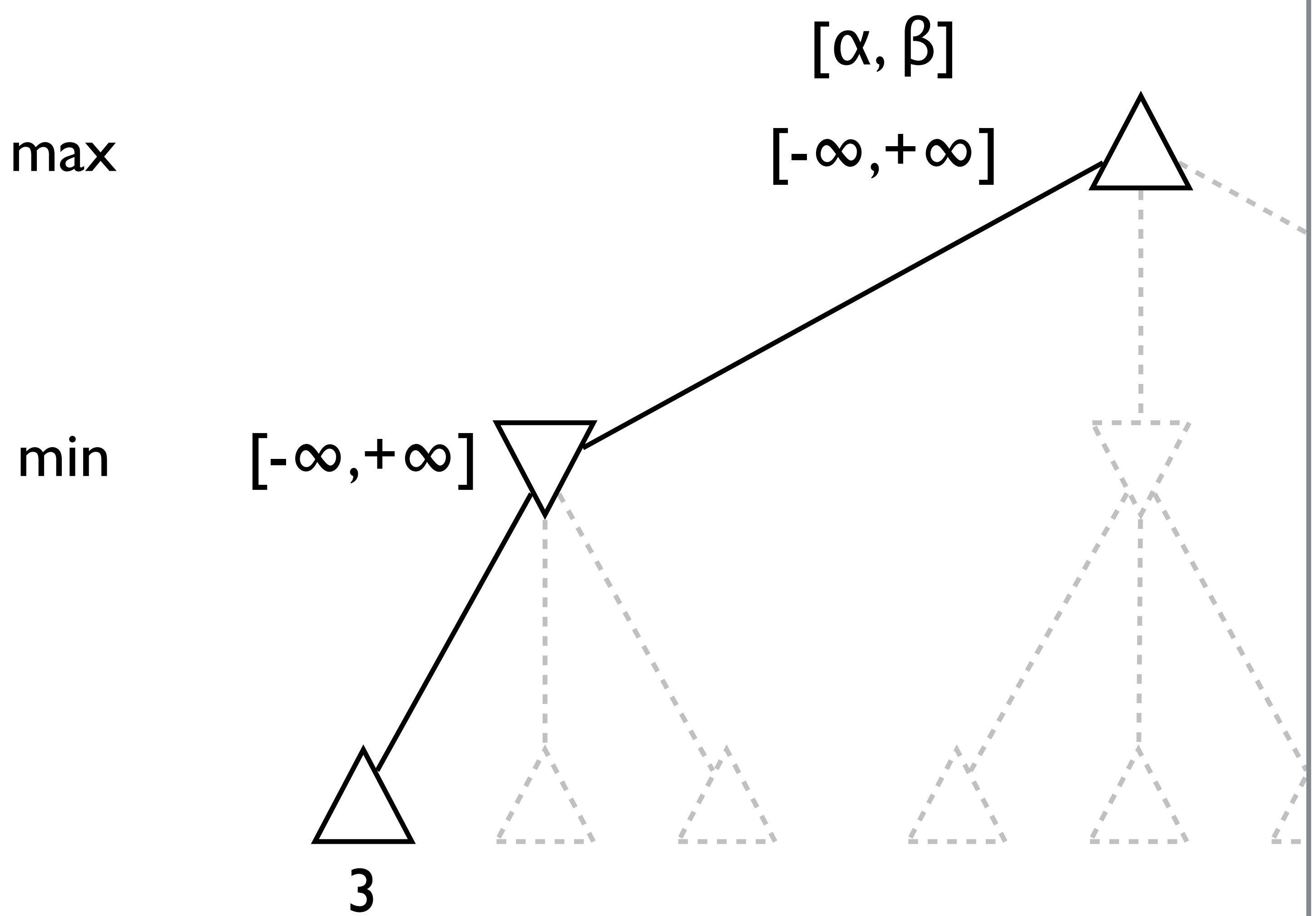
function MIN-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Alpha-beta pruning

- Idea: prune branches that can't influence final decision
- alpha-beta pruning keeps track of the range of possible values:

  α = max's best choice (highest val.) so far at any point along path
  β = min's best choice (lowest val.) so far at any point along path

- alpha-beta doesn't search: it "*prunes*" branches that don't affect current estimates



function ALPHA-BETA-SEARCH($state$) **returns** an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the $action$ in ACTIONS($state$) with value v

function MAX-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

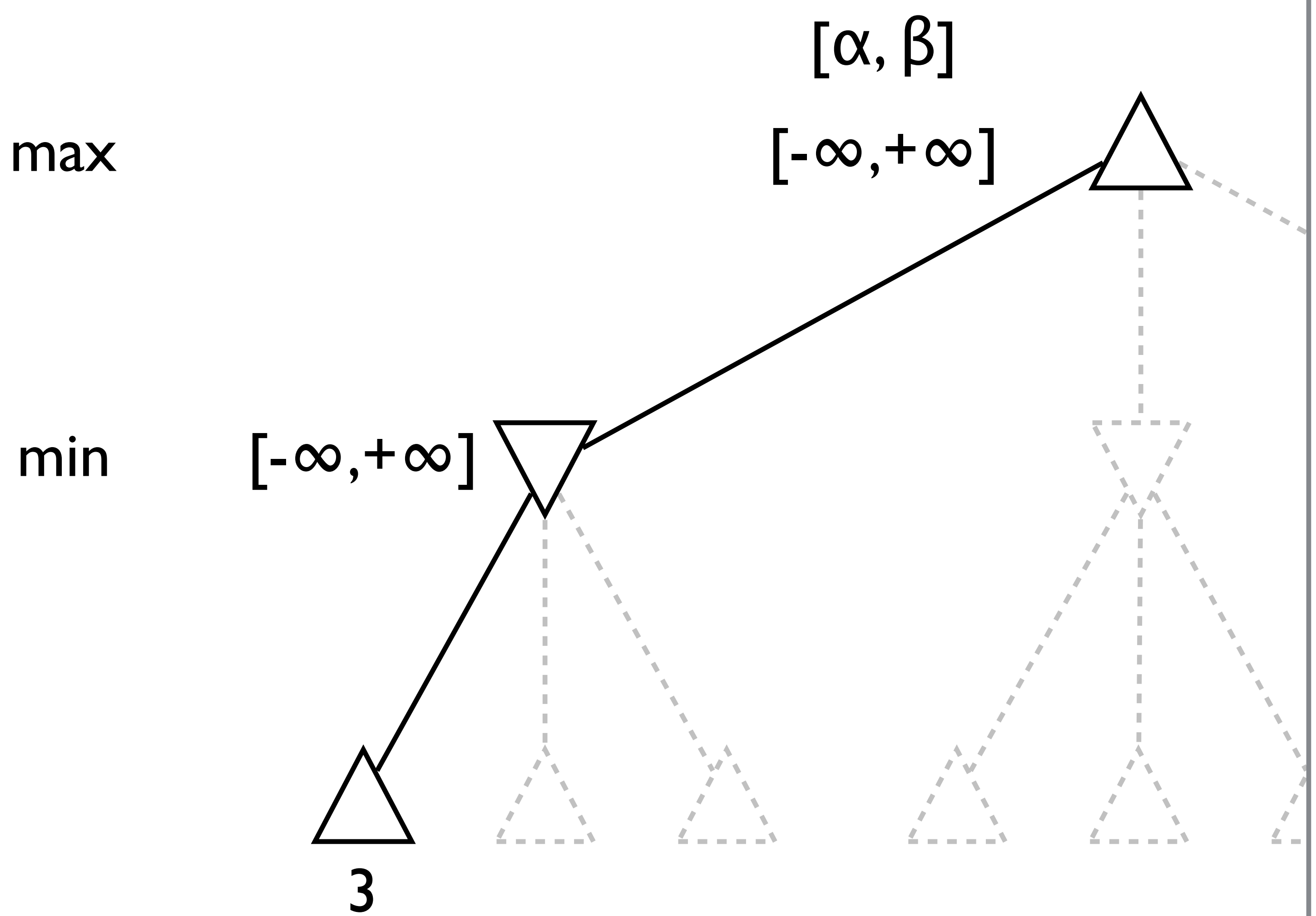
function MIN-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Alpha-beta pruning

- Idea: prune branches that can't influence final decision
- alpha-beta pruning keeps track of the range of possible values:

  α = max's best choice (highest val.) so far at any point along path
  β = min's best choice (lowest val.) so far at any point along path

- alpha-beta doesn't search: it "*prunes*" branches that don't affect current estimates



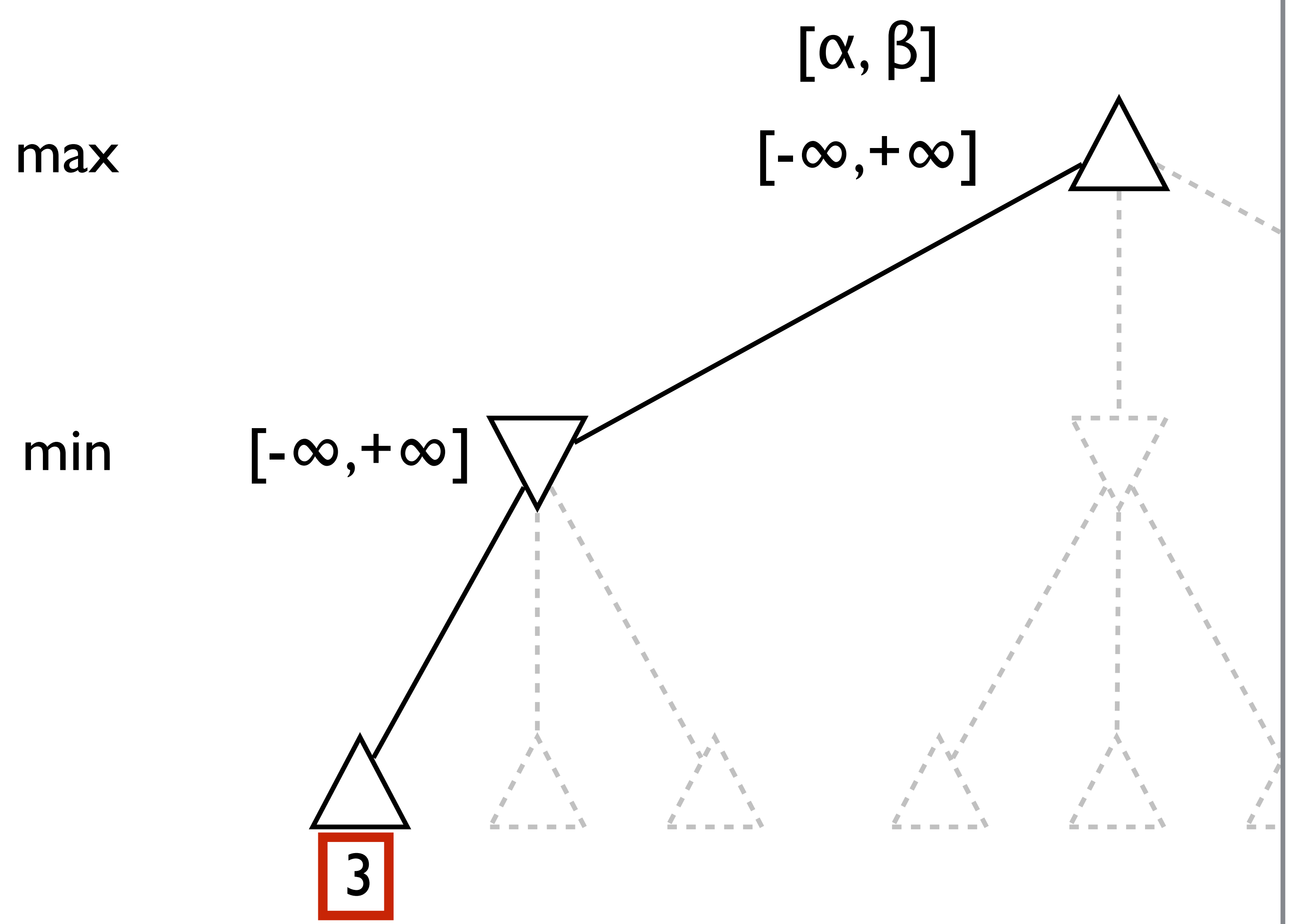

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Alpha-beta pruning

- Idea: prune branches that can't influence final decision
- alpha-beta pruning keeps track of the range of possible values:

  α = max's best choice (highest val.) so far at any point along path
  β = min's best choice (lowest val.) so far at any point along path

- alpha-beta doesn't search: it "*prunes*" branches that don't affect current estimates



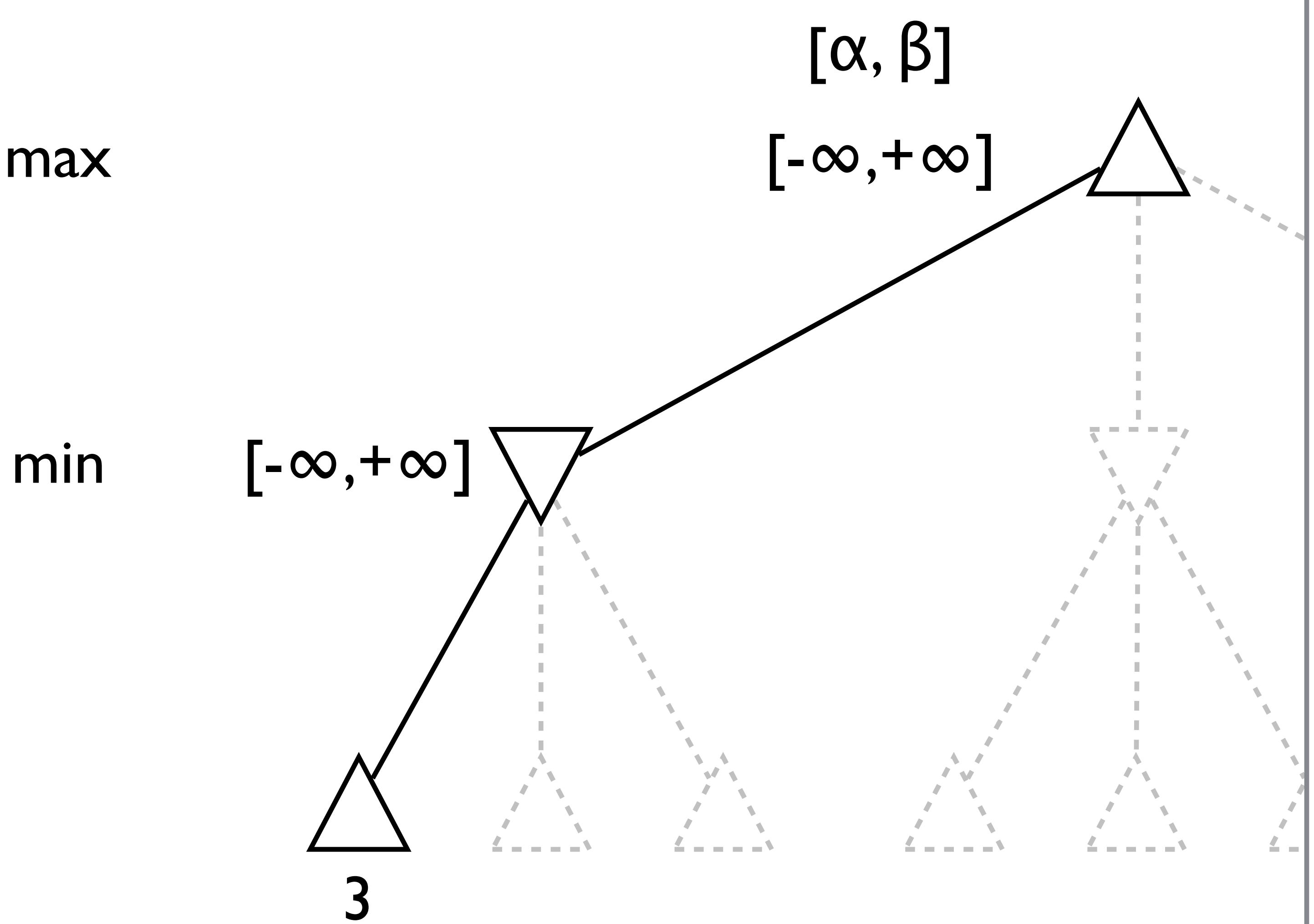

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Alpha-beta pruning

- Idea: prune branches that can't influence final decision
- alpha-beta pruning keeps track of the range of possible values:

  α = max's best choice (highest val.) so far at any point along path
  β = min's best choice (lowest val.) so far at any point along path

- alpha-beta doesn't search: it "*prunes*" branches that don't affect current estimates



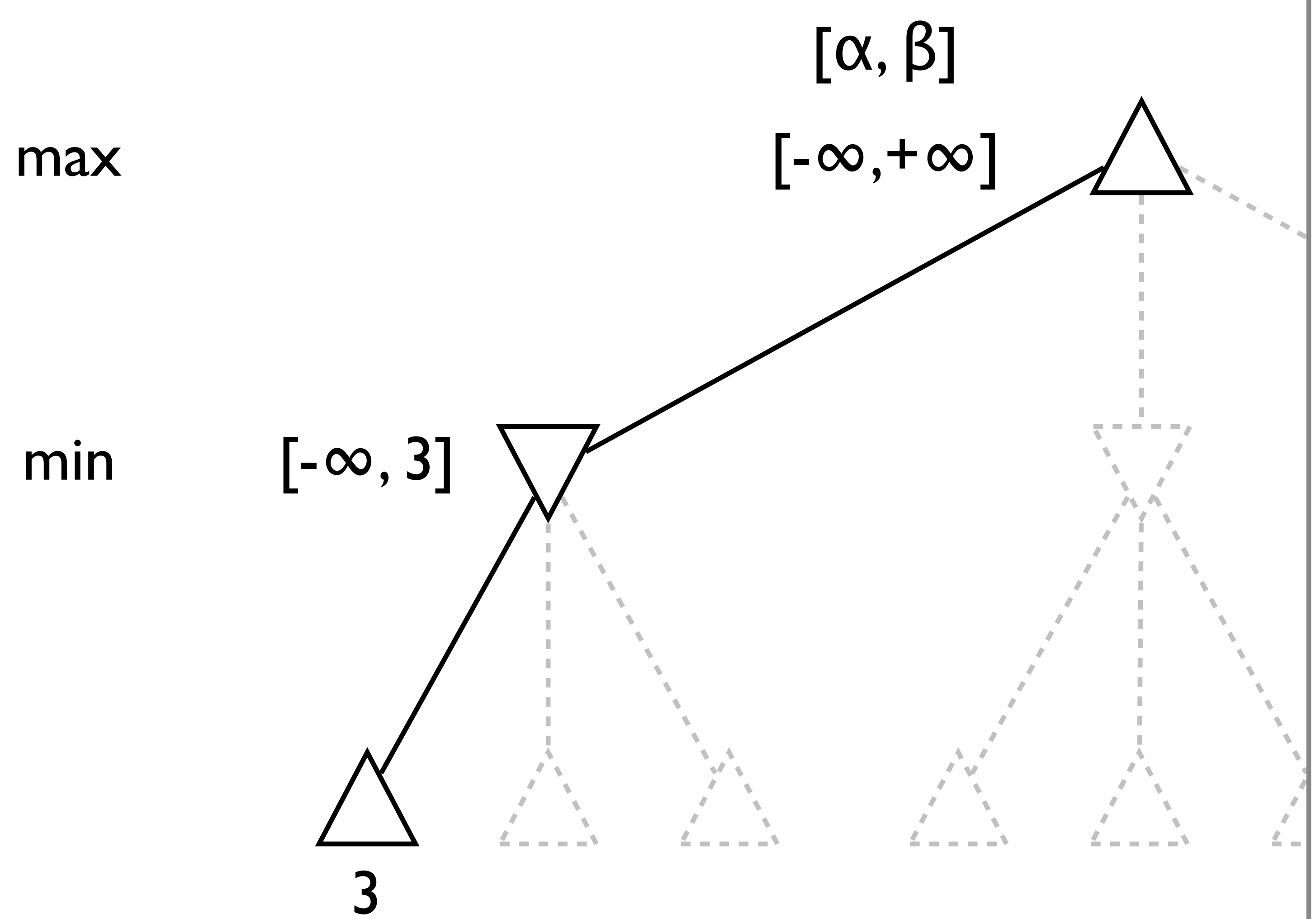

```
function ALPHA-BETA-SEARCH(state) returns an action
   v ← MAX-VALUE(state, −∞, +∞)
   return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for each a in ACTIONS(state) do
      v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
      if v ≥ β then return v
      α ← MAX(α, v)
   return v

function MIN-VALUE(state, α, β) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← +∞
   for each a in ACTIONS(state) do
      v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
      if v ≤ α then return v
      β ← MIN(β, v)
   return v
```

Alpha-beta pruning

- Idea: prune branches that can't influence final decision
- alpha-beta pruning keeps track of the range of possible values:

  α = max's best choice (highest val.) so far at any point along path
  β = min's best choice (lowest val.) so far at any point along path

- alpha-beta doesn't search: it "*prunes*" branches that don't affect current estimates



function ALPHA-BETA-SEARCH($state$) **returns** an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the $action$ in ACTIONS($state$) with value v

---

function MAX-VALUE($state, \alpha, \beta$) **returns** a $utility$ $value$
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

---

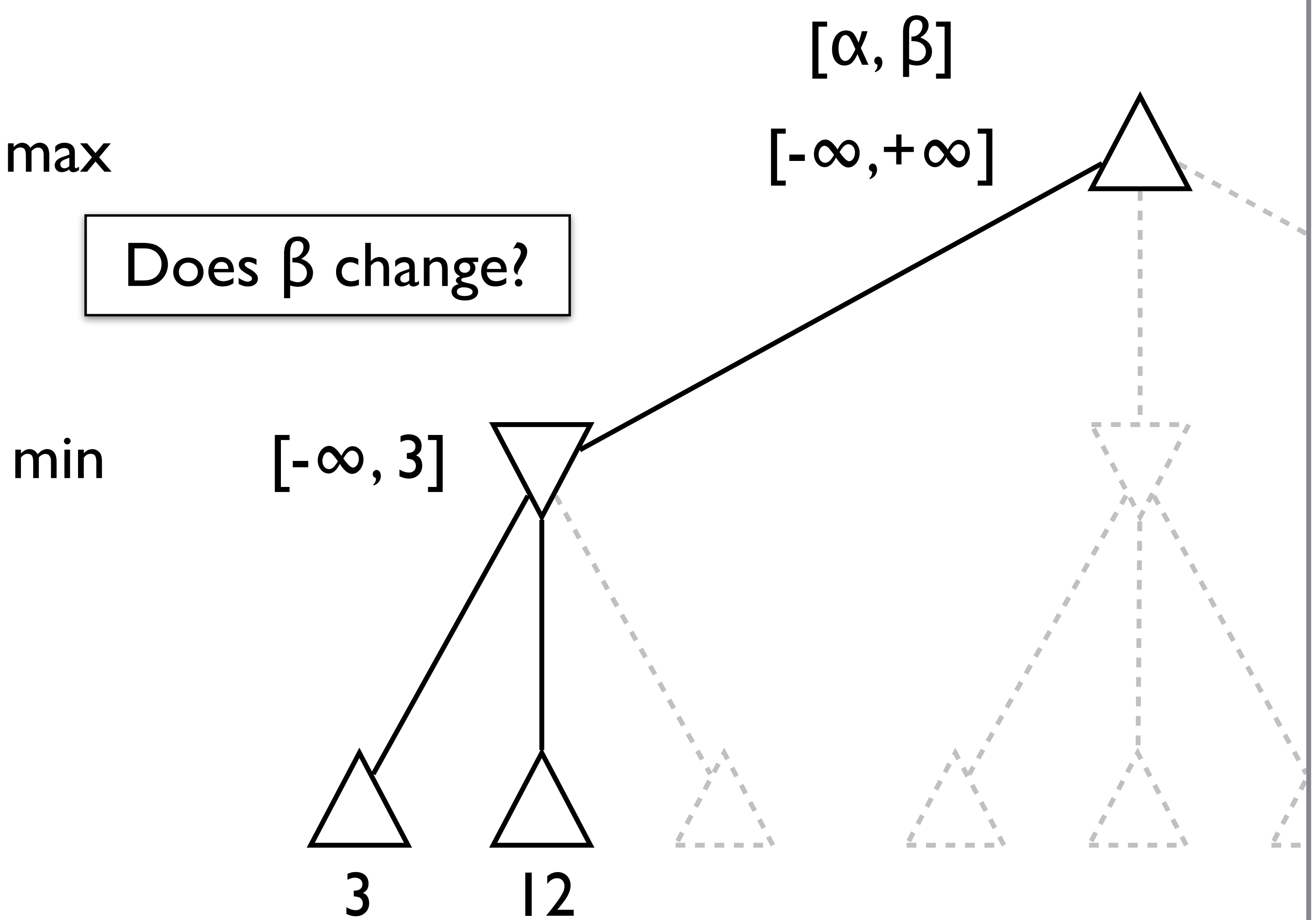
function MIN-VALUE($state, \alpha, \beta$) **returns** a $utility$ $value$
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Alpha-beta pruning

- Idea: prune branches that can't influence final decision
- alpha-beta pruning keeps track of the range of possible values:

  α = max's best choice (highest val.) so far at any point along path
  β = min's best choice (lowest val.) so far at any point along path

- alpha-beta doesn't search: it "*prunes*" branches that don't affect current estimates



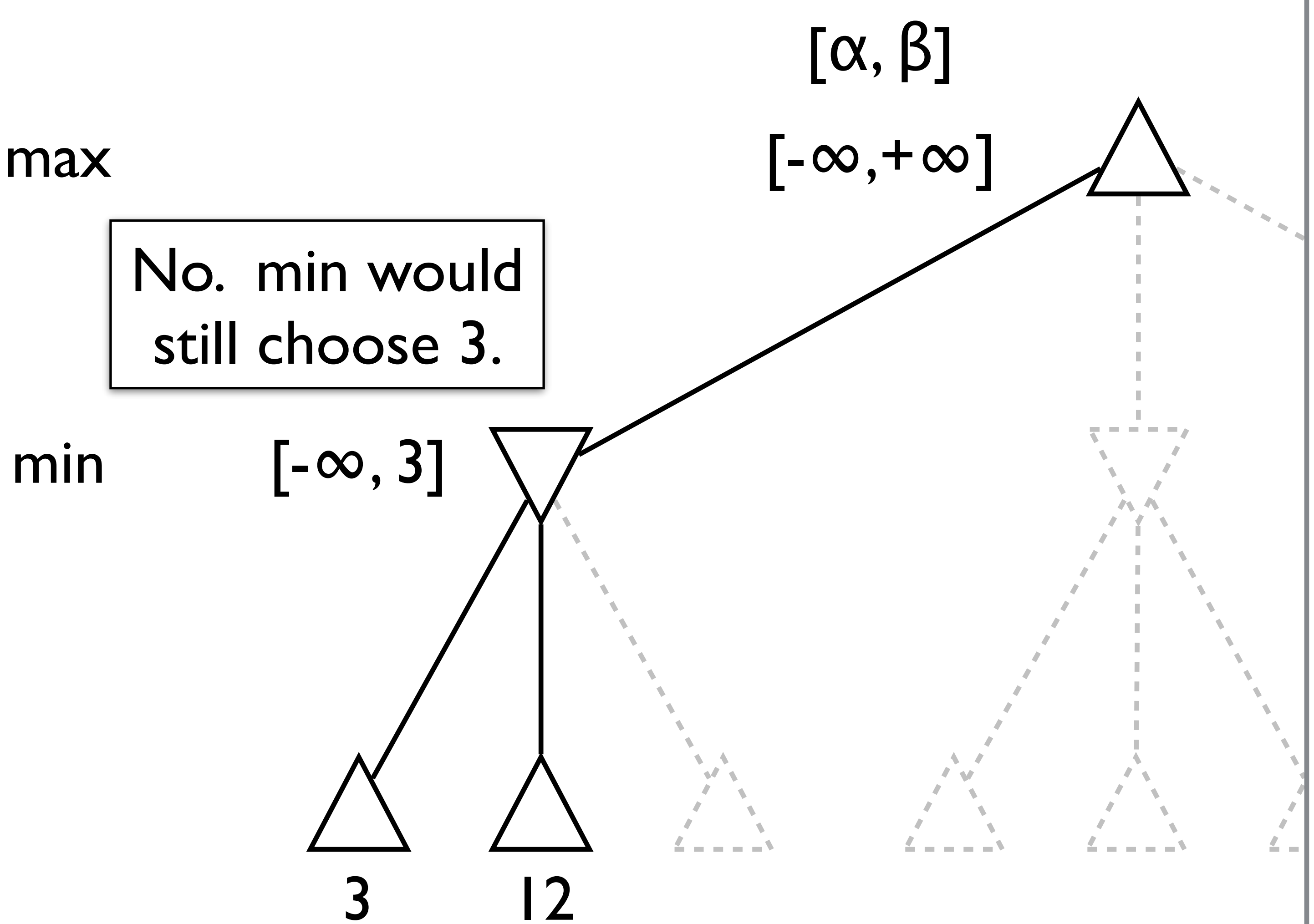

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Alpha-beta pruning

- Idea: prune branches that can't influence final decision
- alpha-beta pruning keeps track of the range of possible values:

  α = max's best choice (highest val.) so far at any point along path
  β = min's best choice (lowest val.) so far at any point along path

- alpha-beta doesn't search: it "*prunes*" branches that don't affect current estimates



function ALPHA-BETA-SEARCH($state$) **returns** an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the $action$ in ACTIONS($state$) with value v

---

function MAX-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

---

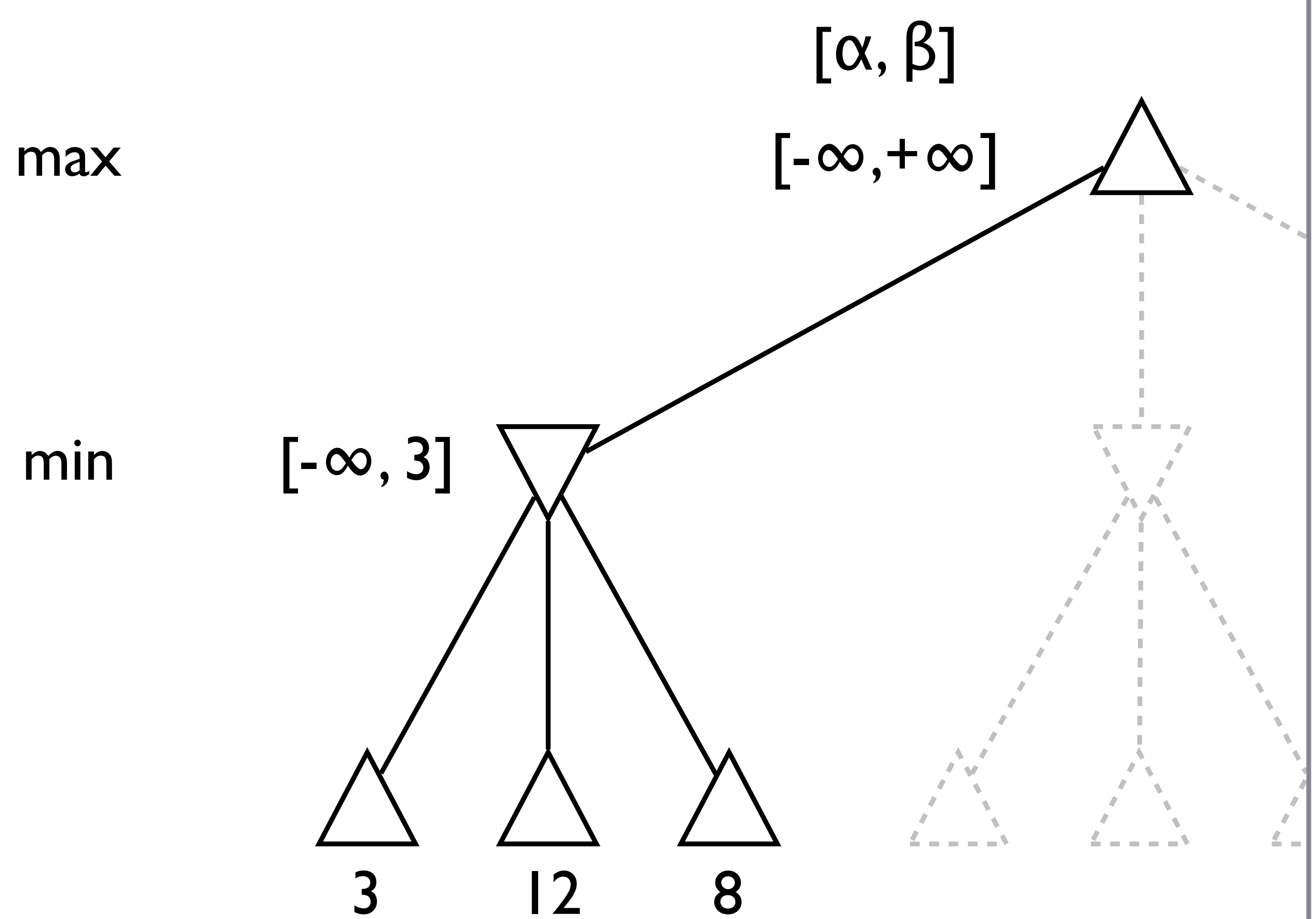
function MIN-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Alpha-beta pruning

- Idea: prune branches that can't influence final decision
- alpha-beta pruning keeps track of the range of possible values:

  α = max's best choice (highest val.) so far at any point along path
  β = min's best choice (lowest val.) so far at any point along path

- alpha-beta doesn't search: it "*prunes*" branches that don't affect current estimates



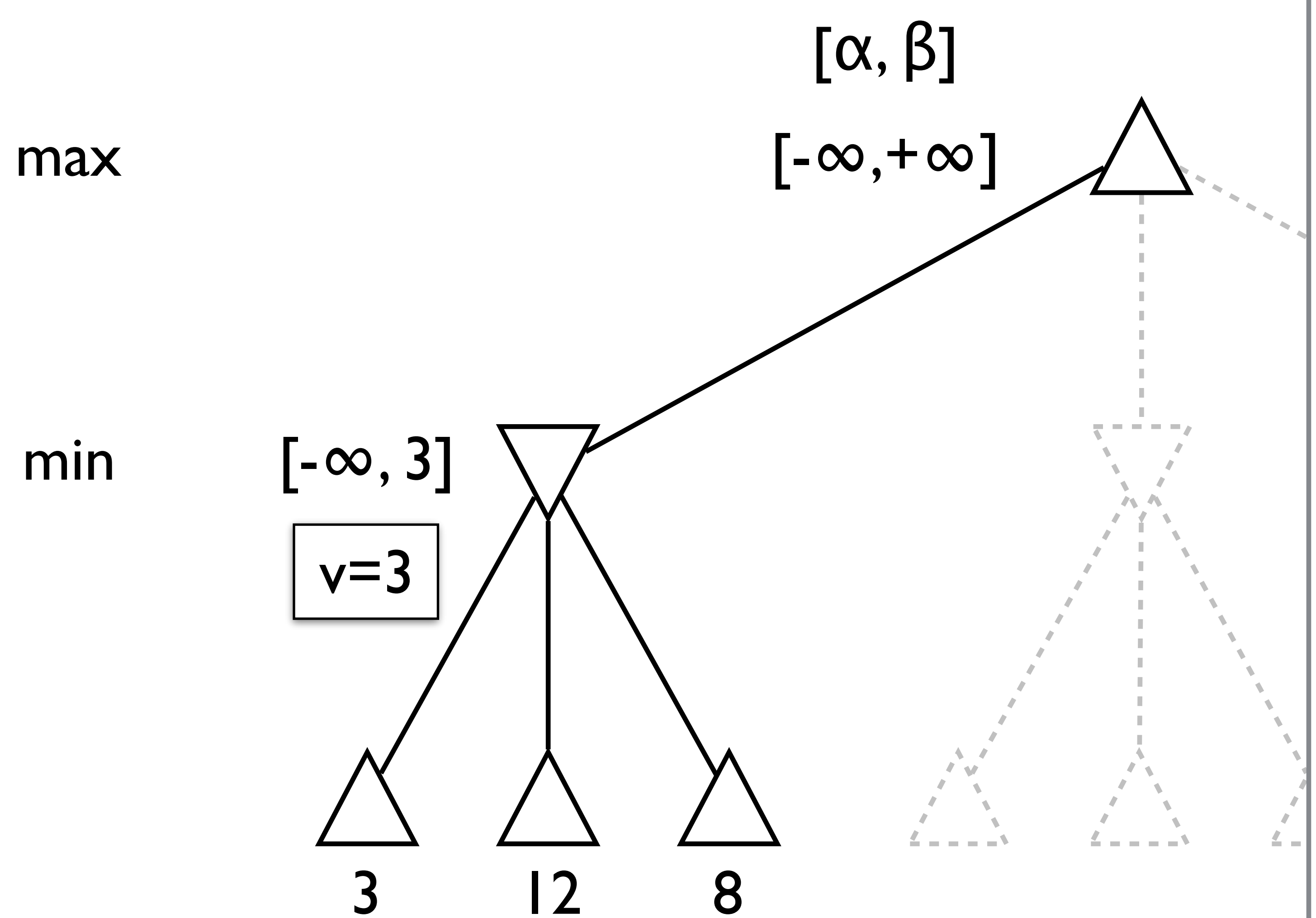

```
function ALPHA-BETA-SEARCH(state) returns an action
   v ← MAX-VALUE(state, −∞, +∞)
   return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for each a in ACTIONS(state) do
      v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
      if v ≥ β then return v
      α ← MAX(α, v)
   return v

function MIN-VALUE(state, α, β) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← +∞
   for each a in ACTIONS(state) do
      v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
      if v ≤ α then return v
      β ← MIN(β, v)
   return v
```

Alpha-beta pruning

- Idea: prune branches that can't influence final decision
- alpha-beta pruning keeps track of the range of possible values:

  α = max's best choice (highest val.) so far at any point along path
  β = min's best choice (lowest val.) so far at any point along path

- alpha-beta doesn't search: it "*prunes*" branches that don't affect current estimates



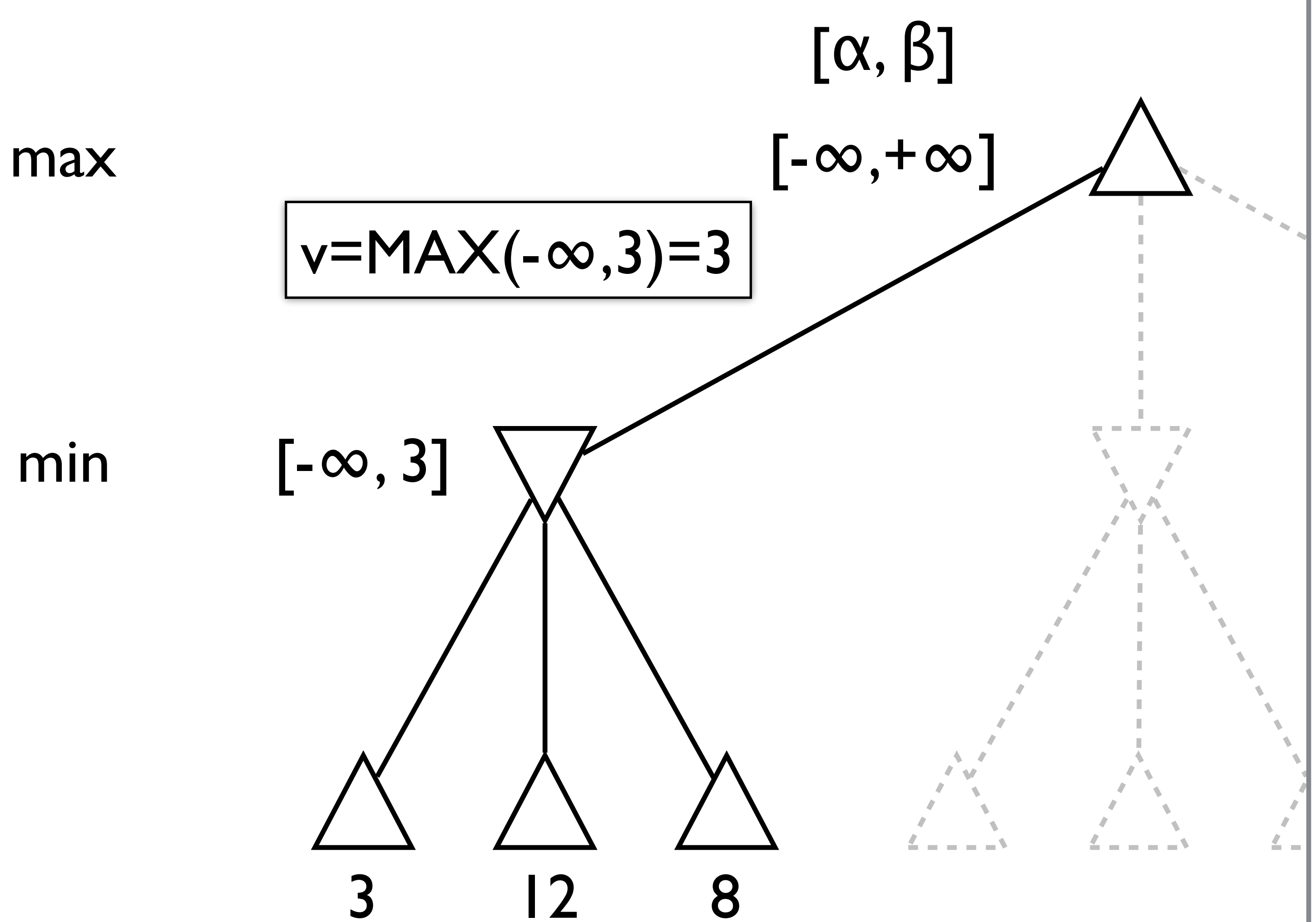

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Alpha-beta pruning

- Idea: prune branches that can't influence final decision
- alpha-beta pruning keeps track of the range of possible values:

  α = max's best choice (highest val.) so far at any point along path
  β = min's best choice (lowest val.) so far at any point along path

- alpha-beta doesn't search: it "*prunes*" branches that don't affect current estimates



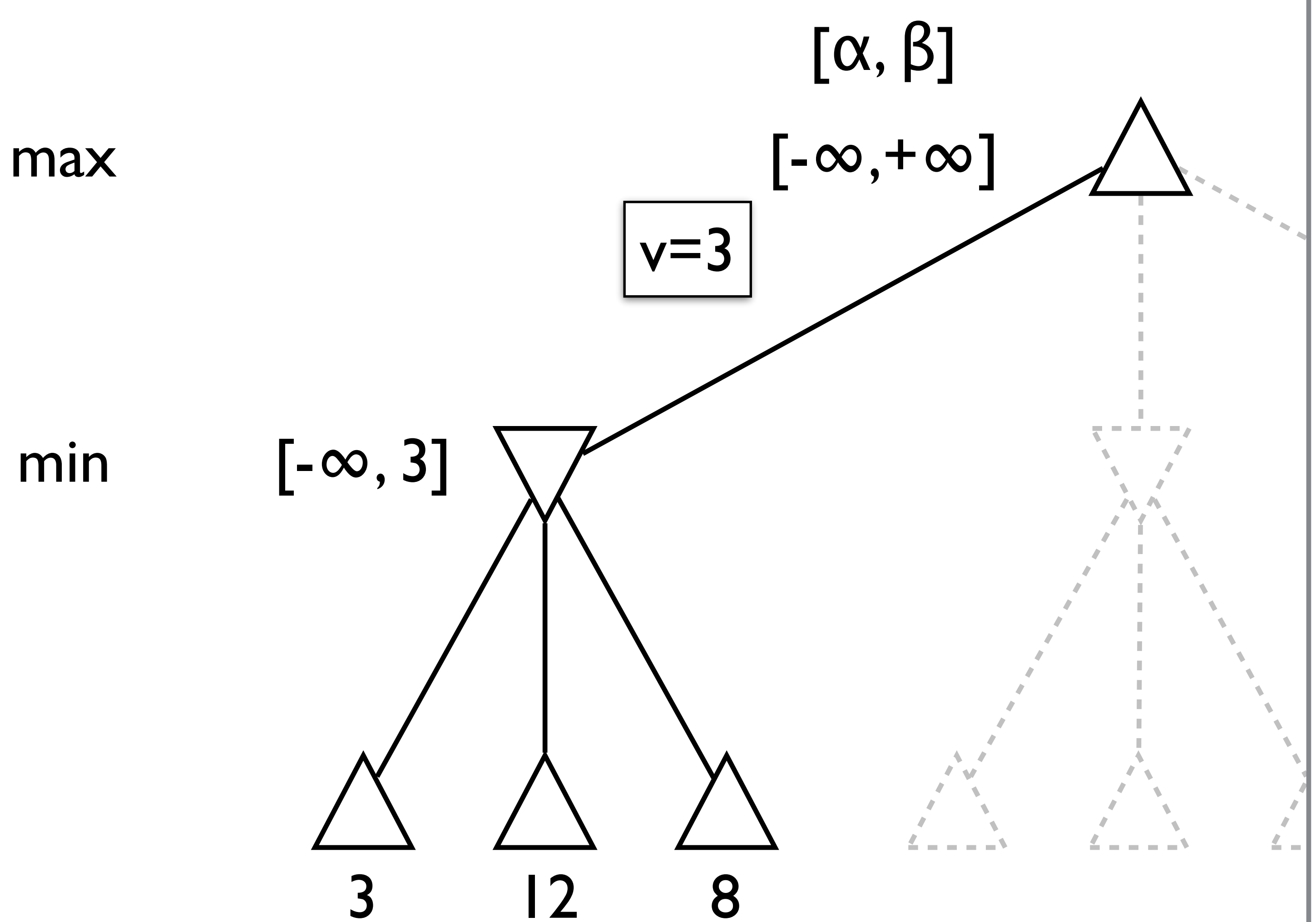

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Alpha-beta pruning

- Idea: prune branches that can't influence final decision
- alpha-beta pruning keeps track of the range of possible values:

  α = max's best choice (highest val.) so far at any point along path
  β = min's best choice (lowest val.) so far at any point along path

- alpha-beta doesn't search: it "*prunes*" branches that don't affect current estimates



function ALPHA-BETA-SEARCH($state$) **returns** an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the $action$ in ACTIONS($state$) with value v

---

function MAX-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

---

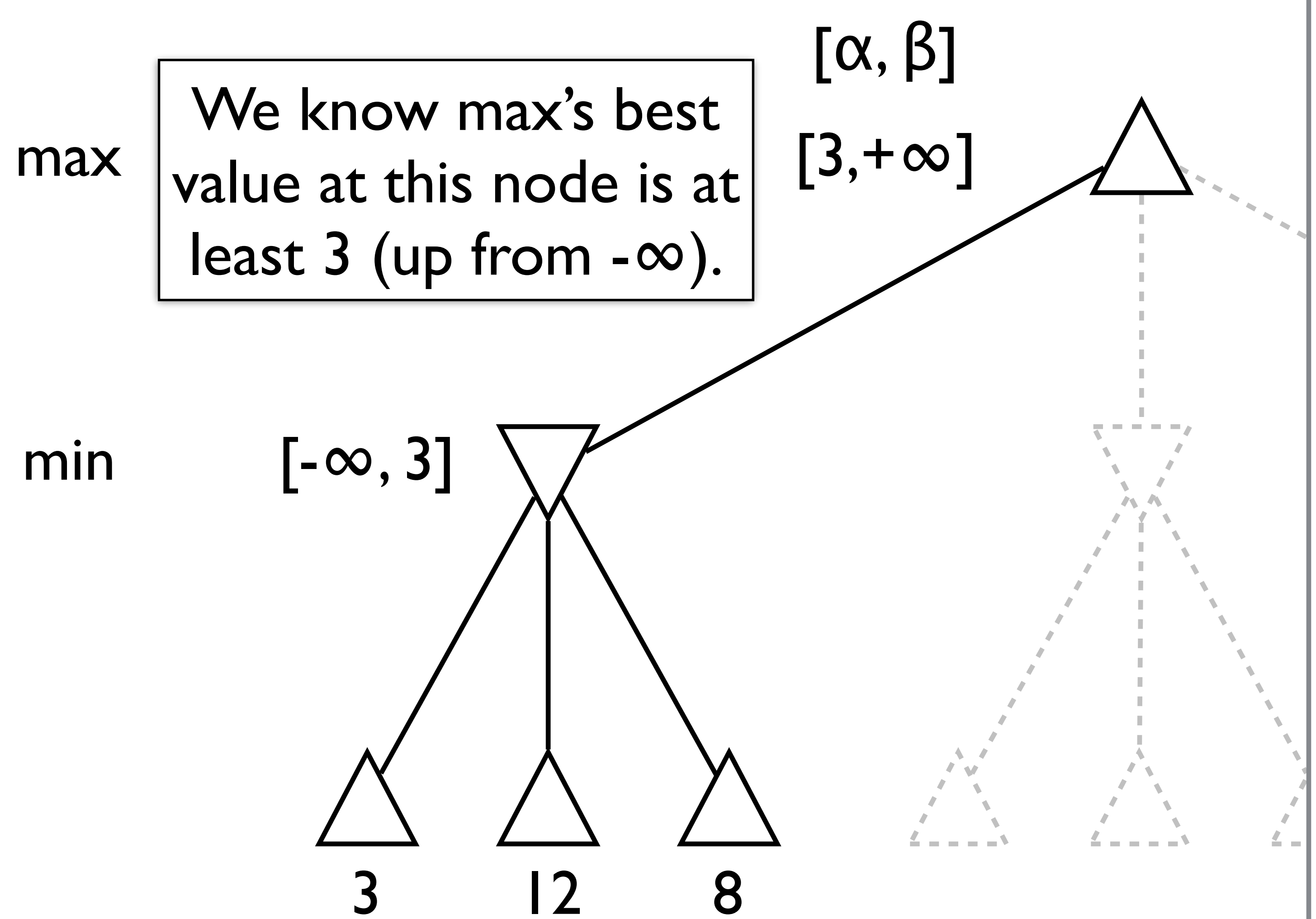
function MIN-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Alpha-beta pruning

- Idea: prune branches that can't influence final decision
- alpha-beta pruning keeps track of the range of possible values:

  α = max's best choice (highest val.) so far at any point along path
  β = min's best choice (lowest val.) so far at any point along path

- alpha-beta doesn't search: it "*prunes*" branches that don't affect current estimates



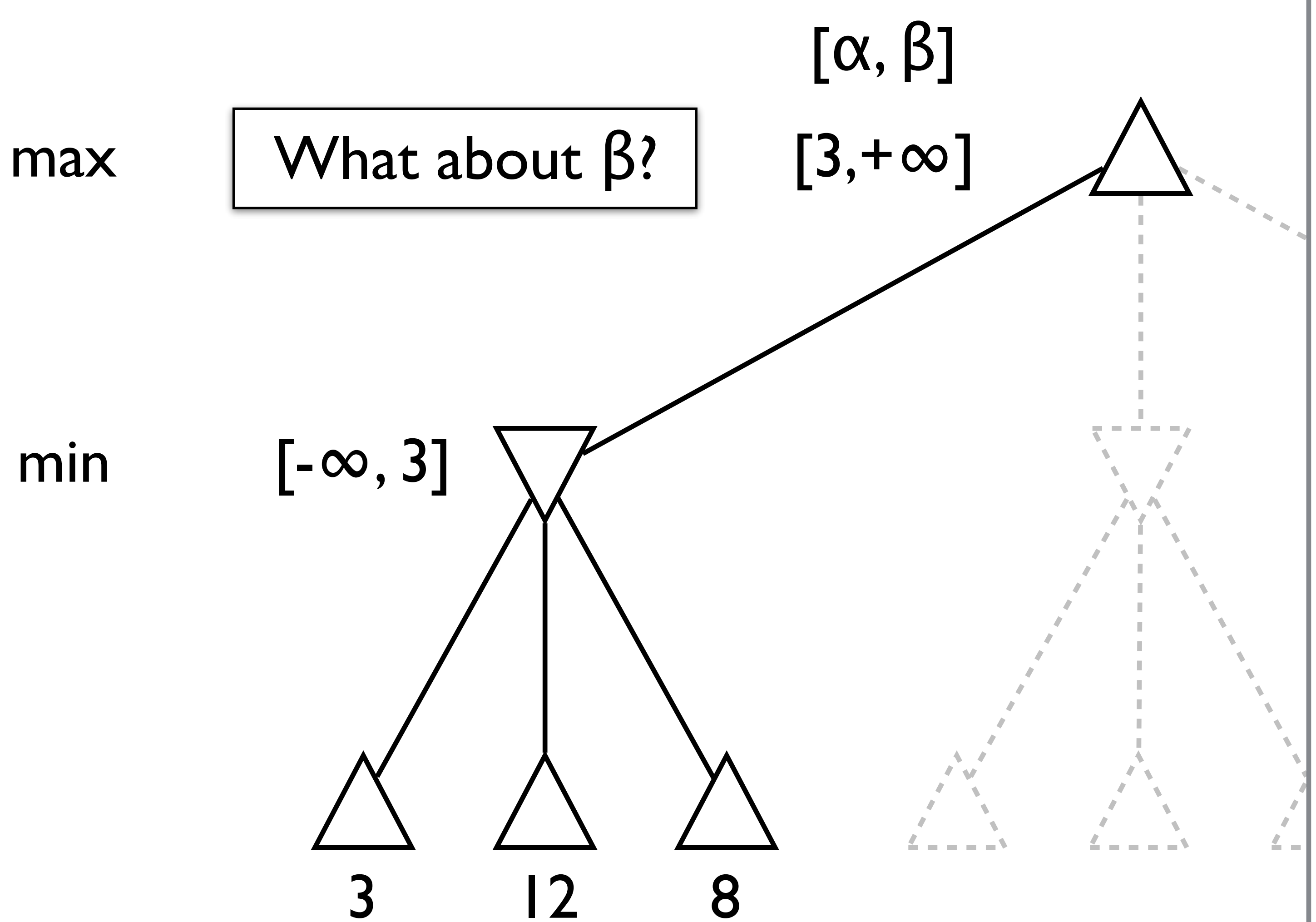

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Alpha-beta pruning

- Idea: prune branches that can't influence final decision
- alpha-beta pruning keeps track of the range of possible values:

  α = max's best choice (highest val.) so far at any point along path
  β = min's best choice (lowest val.) so far at any point along path

- alpha-beta doesn't search: it "*prunes*" branches that don't affect current estimates



function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the *action* in ACTIONS(*state*) with value v

---

function MAX-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

---

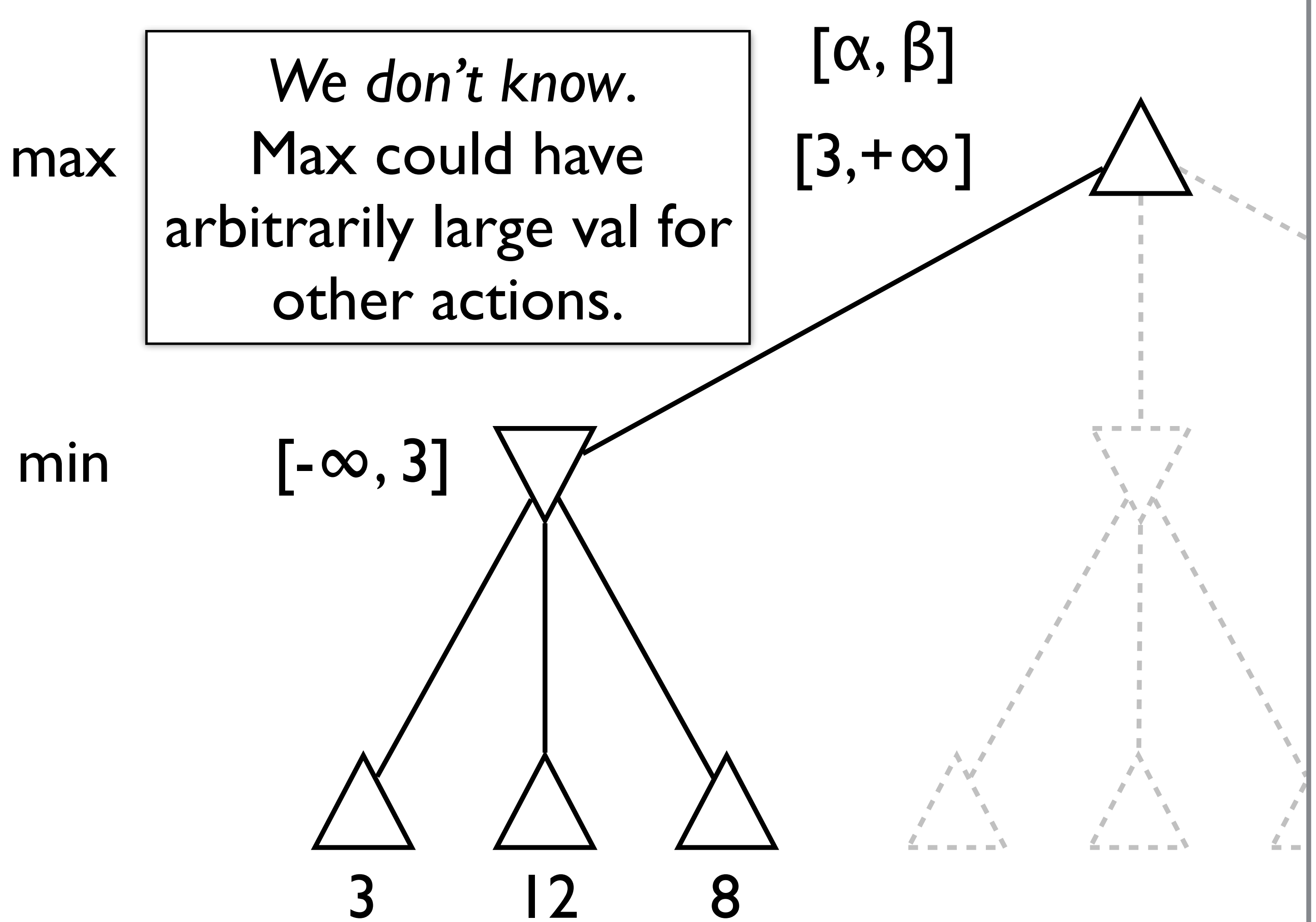
function MIN-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Alpha-beta pruning

- Idea: prune branches that can't influence final decision
- alpha-beta pruning keeps track of the range of possible values:

  α = max's best choice (highest val.) so far at any point along path
  β = min's best choice (lowest val.) so far at any point along path

- alpha-beta doesn't search: it "*prunes*" branches that don't affect current estimates



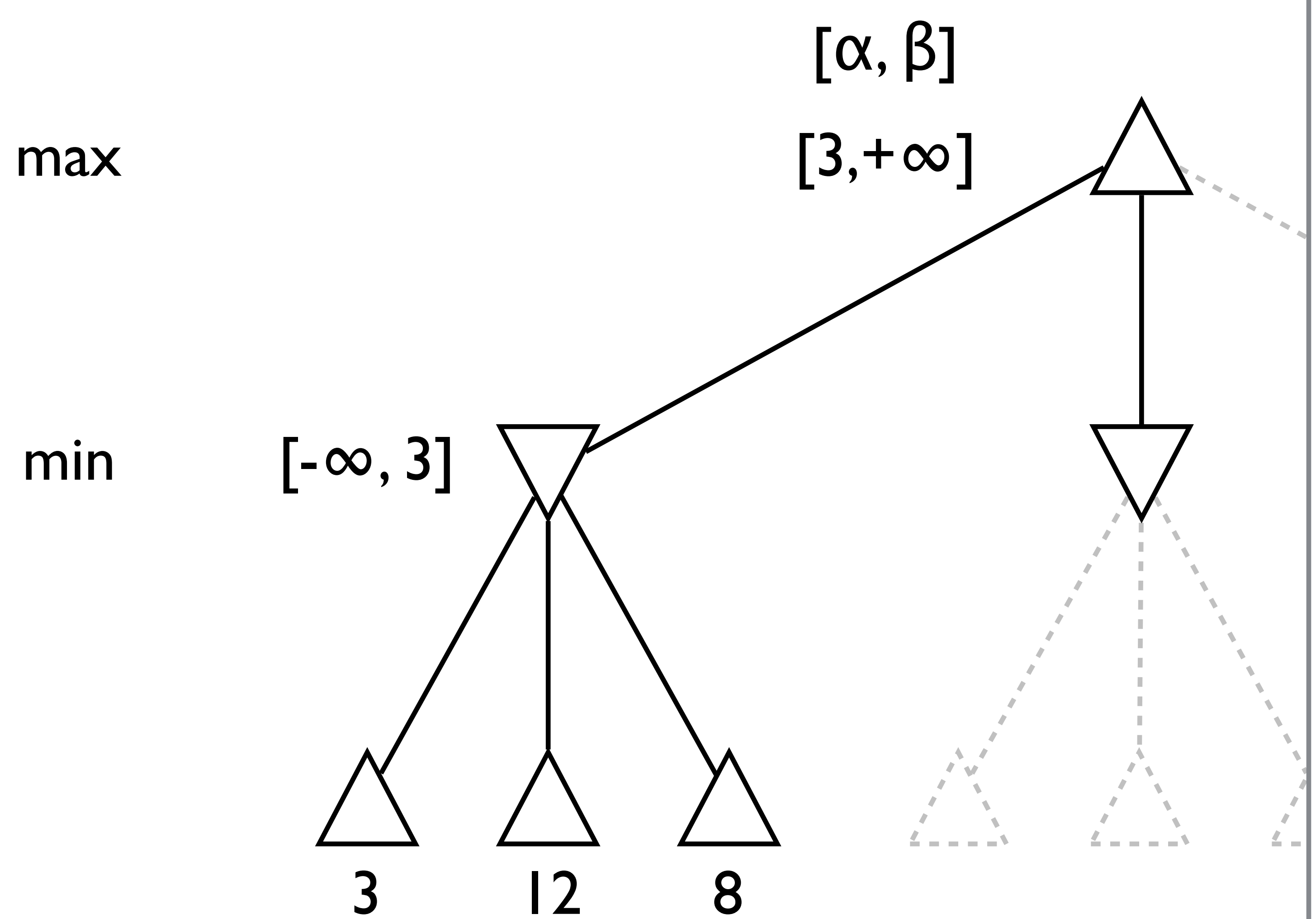

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Alpha-beta pruning

- Idea: prune branches that can't influence final decision
- alpha-beta pruning keeps track of the range of possible values:

  α = max's best choice (highest val.) so far at any point along path
  β = min's best choice (lowest val.) so far at any point along path

- alpha-beta doesn't search: it "*prunes*" branches that don't affect current estimates



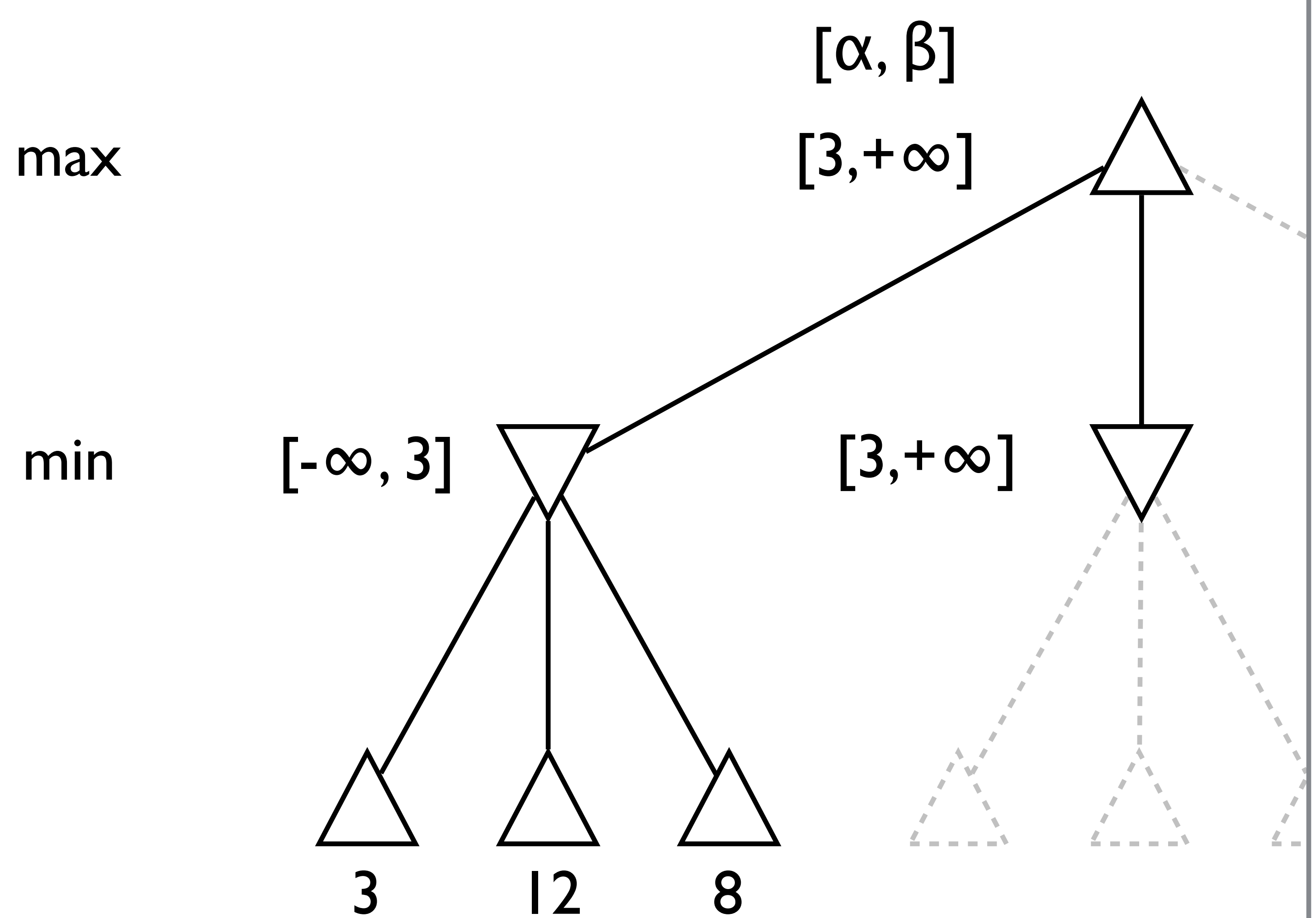

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
     v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
     if v ≥ β then return v
     α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
     v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
     if v ≤ α then return v
     β ← MIN(β, v)
  return v
```

Alpha-beta pruning

- Idea: prune branches that can't influence final decision
- alpha-beta pruning keeps track of the range of possible values:

  α = max's best choice (highest val.) so far at any point along path
  β = min's best choice (lowest val.) so far at any point along path

- alpha-beta doesn't search: it "*prunes*" branches that don't affect current estimates



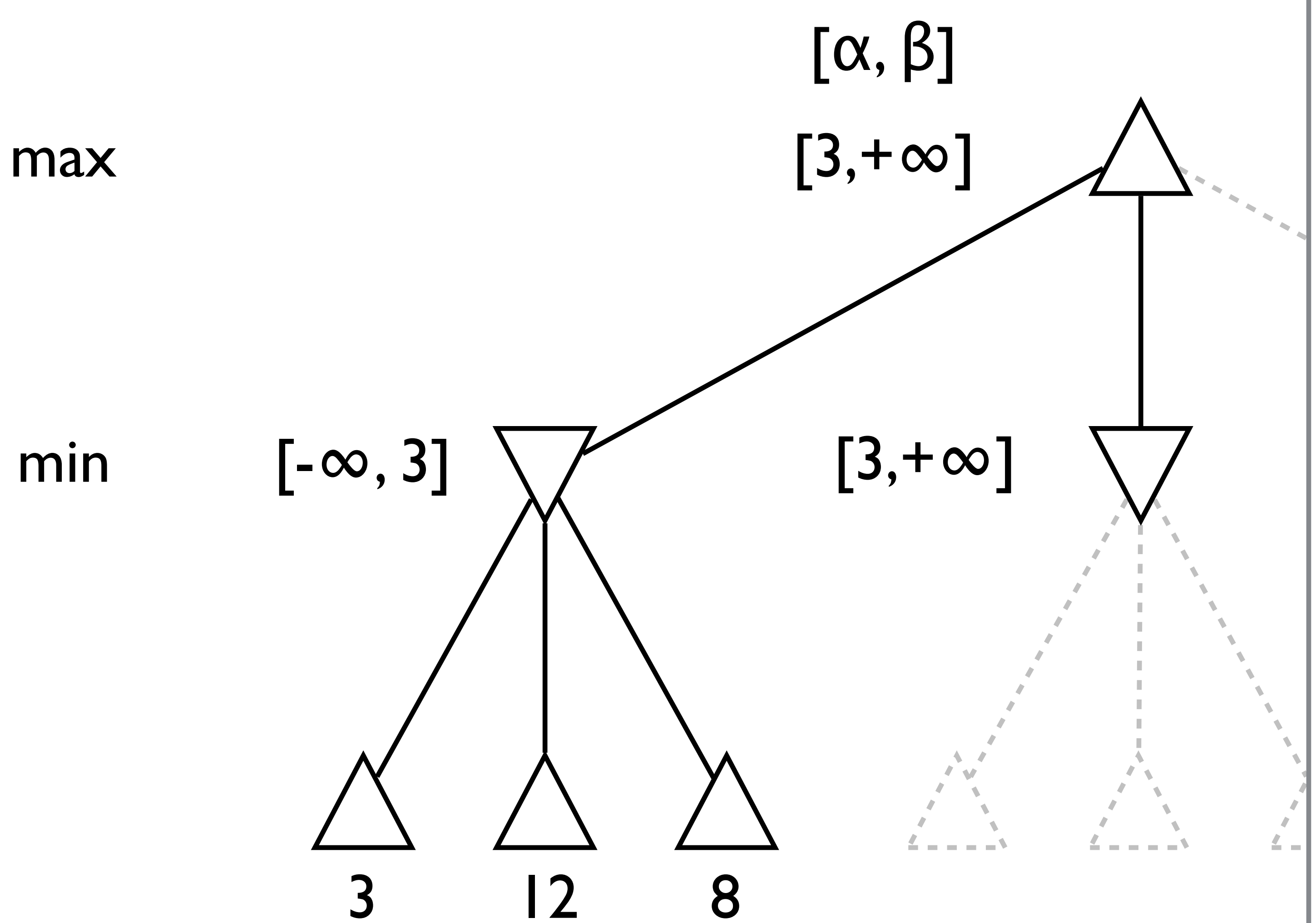

```
function ALPHA-BETA-SEARCH(state) returns an action
   v ← MAX-VALUE(state, −∞, +∞)
   return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for each a in ACTIONS(state) do
      v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
      if v ≥ β then return v
      α ← MAX(α, v)
   return v

function MIN-VALUE(state, α, β) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← +∞
   for each a in ACTIONS(state) do
      v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
      if v ≤ α then return v
      β ← MIN(β, v)
   return v
```

Alpha-beta pruning

- Idea: prune branches that can't influence final decision
- alpha-beta pruning keeps track of the range of possible values:

  α = max's best choice (highest val.) so far at any point along path
  β = min's best choice (lowest val.) so far at any point along path

- alpha-beta doesn't search: it "*prunes*" branches that don't affect current estimates



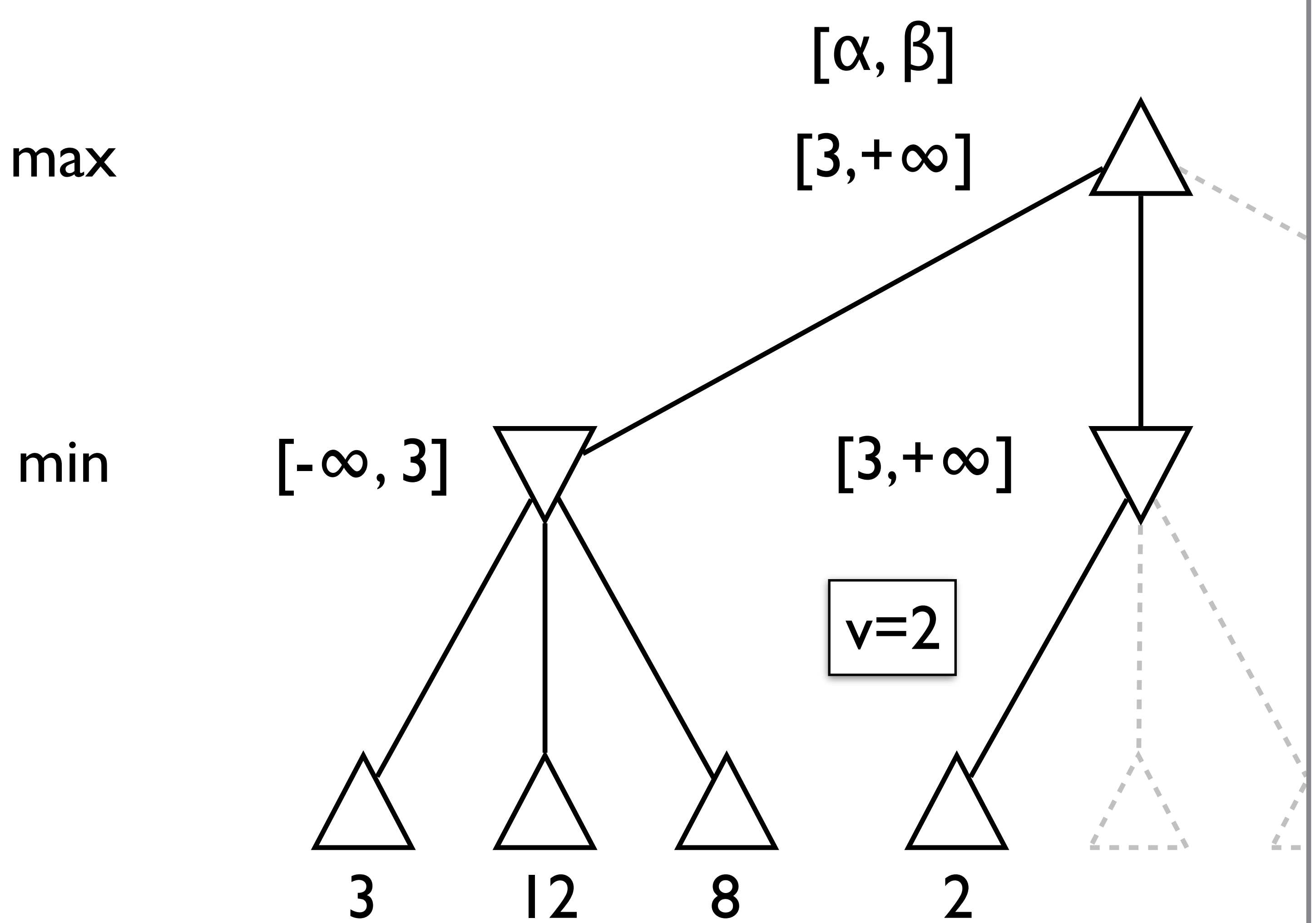

```
function ALPHA-BETA-SEARCH(state) returns an action
   v ← MAX-VALUE(state, −∞, +∞)
   return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for each a in ACTIONS(state) do
      v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
      if v ≥ β then return v
      α ← MAX(α, v)
   return v

function MIN-VALUE(state, α, β) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← +∞
   for each a in ACTIONS(state) do
      v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
      if v ≤ α then return v
      β ← MIN(β, v)
   return v
```

Alpha-beta pruning

- Idea: prune branches that can't influence final decision
- alpha-beta pruning keeps track of the range of possible values:

  α = max's best choice (highest val.) so far at any point along path
  β = min's best choice (lowest val.) so far at any point along path

- alpha-beta doesn't search: it "*prunes*" branches that don't affect current estimates



function ALPHA-BETA-SEARCH($state$) **returns** an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the $action$ in ACTIONS($state$) with value v

function MAX-VALUE($state, \alpha, \beta$) **returns** a $utility$ $value$
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

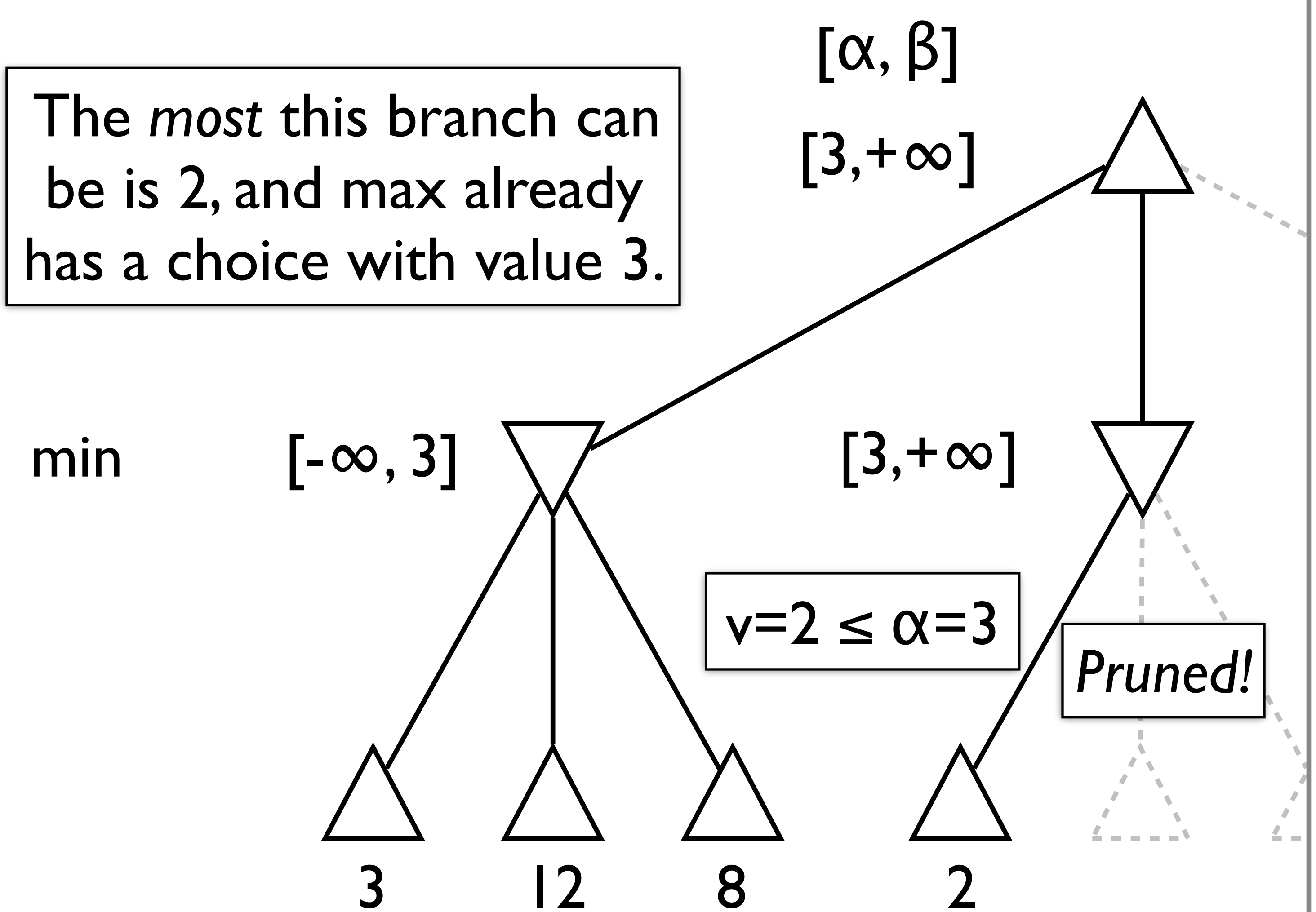
function MIN-VALUE($state, \alpha, \beta$) **returns** a $utility$ $value$
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Alpha-beta pruning

- Idea: prune branches that can't influence final decision
- alpha-beta pruning keeps track of the range of possible values:

  α = max's best choice (highest val.) so far at any point along path
  β = min's best choice (lowest val.) so far at any point along path

- alpha-beta doesn't search: it "*prunes*" branches that don't affect current estimates



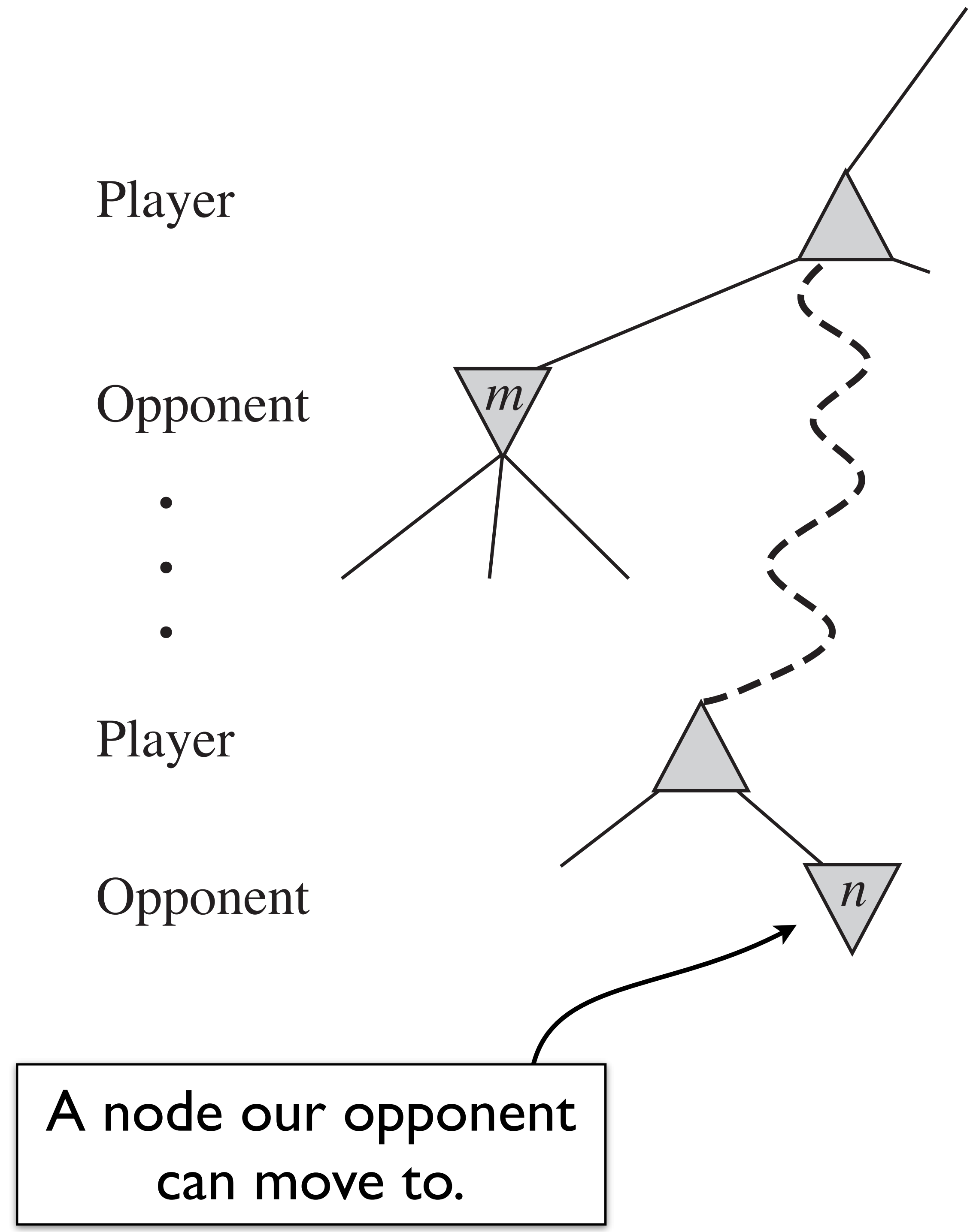
function ALPHA-BETA-SEARCH($state$) **returns** an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the $action$ in ACTIONS($state$) with value v

function MAX-VALUE($state, \alpha, \beta$) **returns** a $utility$ $value$
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

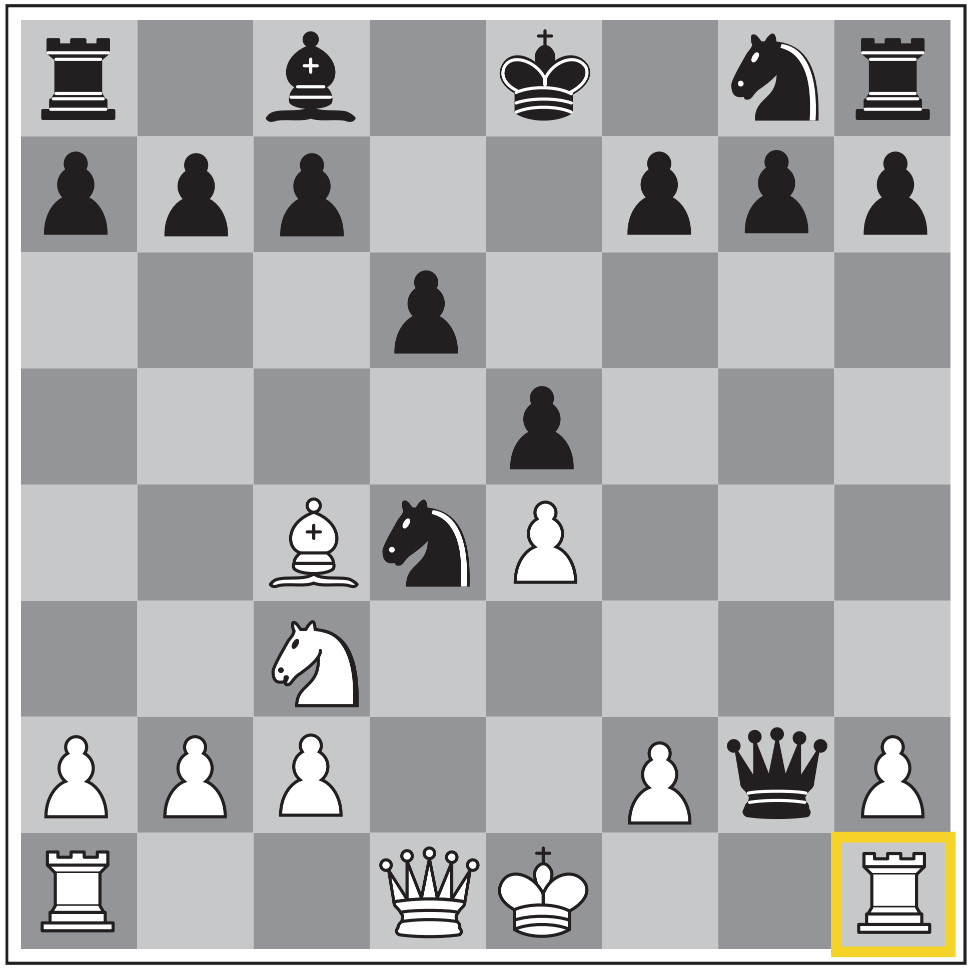
function MIN-VALUE($state, \alpha, \beta$) **returns** a $utility$ $value$
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

More properties of alpha-beta pruning

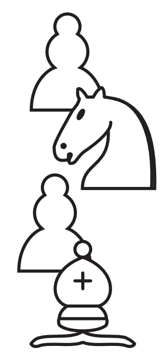
- can prune entire sub-trees, not just leaves
- Can result in a huge savings, e.g. $O(b^{m/2})$ vs $O(b^m)$
- search trees twice as deep
- But chess is still $O(35^{50})$
- *Problem*: game trees are far too *deep*.
- *Idea*: estimate *expected* utility based on game position evaluation function.



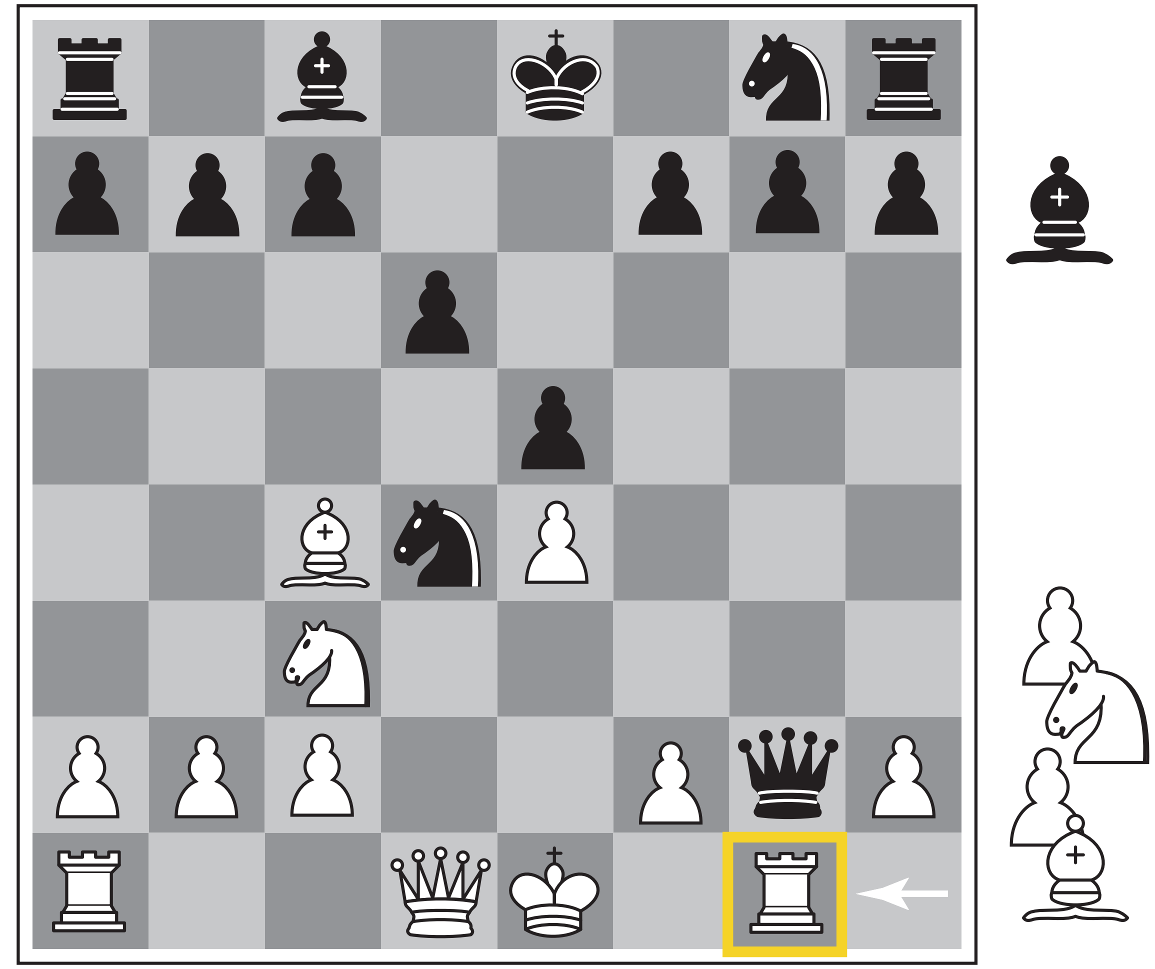
evaluation function example



Black has
advantage of
knight and
two pawns



white to move, but black has advantage (2x1 + 3 = 5)



white to move, can capture queen, and take the advantage

For chess, typically linear weighted sum of features

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

e.g., $w_1 = 9$ with
$f_1(s)$ = (number of white queens) – (number of black queens), etc.

pawn = 1
knight or bishop = 3
rook = 5
queen = 9
king = ∞

Minimax for heuristic evaluation functions

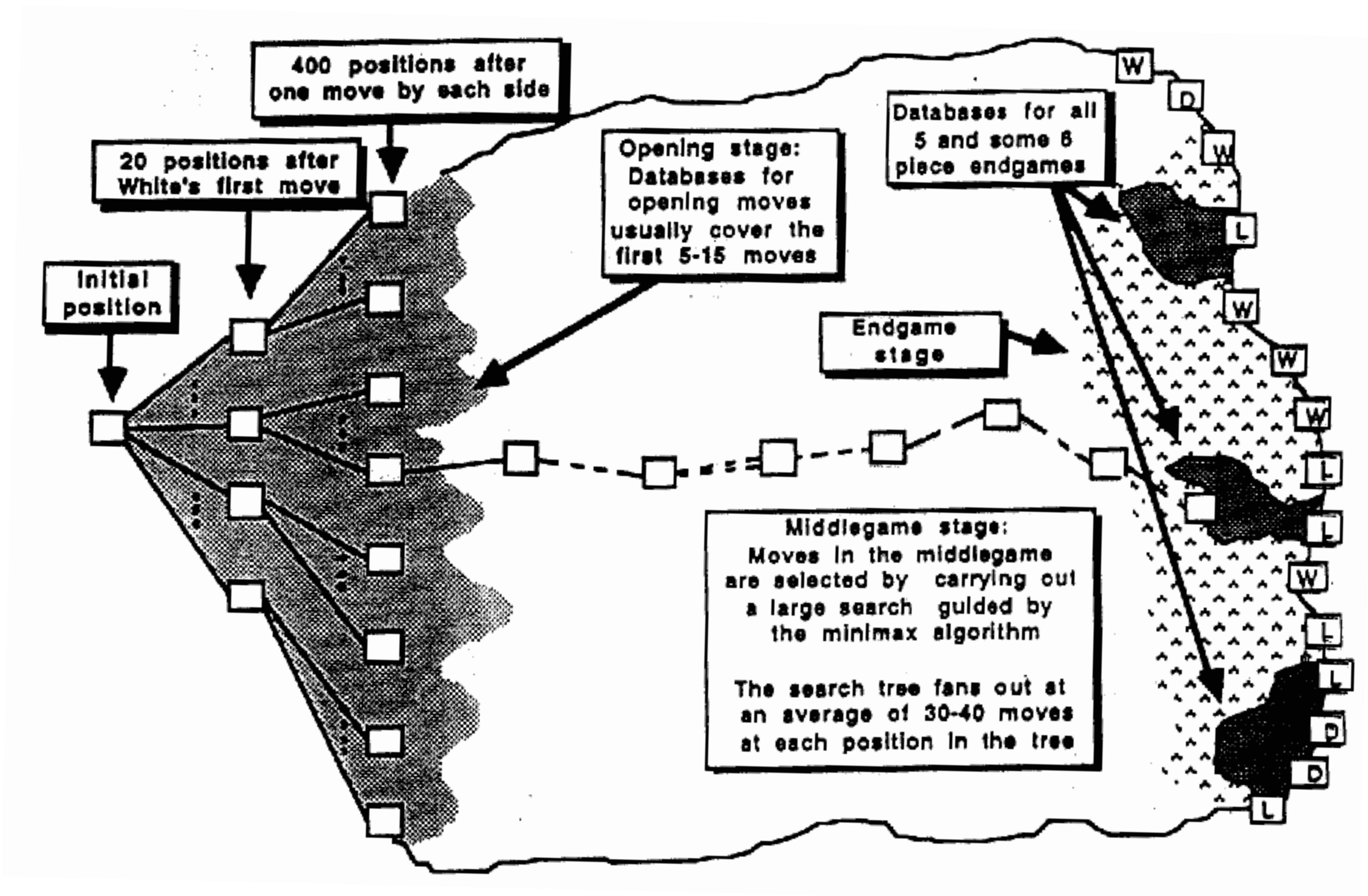
- Can't search very deep in complex games, need to cutoff search early
- Instead of using Utility of Terminal State, use heuristics at maximum depth.
- H-Minimax(s,d) =
  - Eval(s) if cutoff-test(s,d)
  - max a in Actions(s) H-Minimax(Result(s,a), d+1) if Player(s) = Max
  - min a in Actions(s) H-Minimax(Result(s,a), d+1) if Player(s) = Min

A brief history of computer game playing

- computer considers moves (Babbage, 1846)
- algorithm for perfect game play (Zermelo, 1912; Von Neumann, 1944)
- finite horizon, approximate evaluation (Zuse, 1945; Wiener, 1948)
- first chess program (Turning, 1951)
- machine learning to improve evaluation accuracy (Samuel, 52-57)
- pruning to allow deeper search (McCarthy, 1956)

Shannon, Turing	minimax search with scoring function	1950
Kotok/McCarthy program, & ITEP program	alpha-beta	1966
Mac Kack	transposition tables	1967
Chess 3.0 - 4.9	iterative-deepening depth-first search	1975
Belle	special purpose circuitry	1978
Cray Blitz	parallel search	1983
Hitech	parallel evaluation	1985
Deep Blue	parallel search and special-purpose circuitry	1987
many others since		

Computer Chess

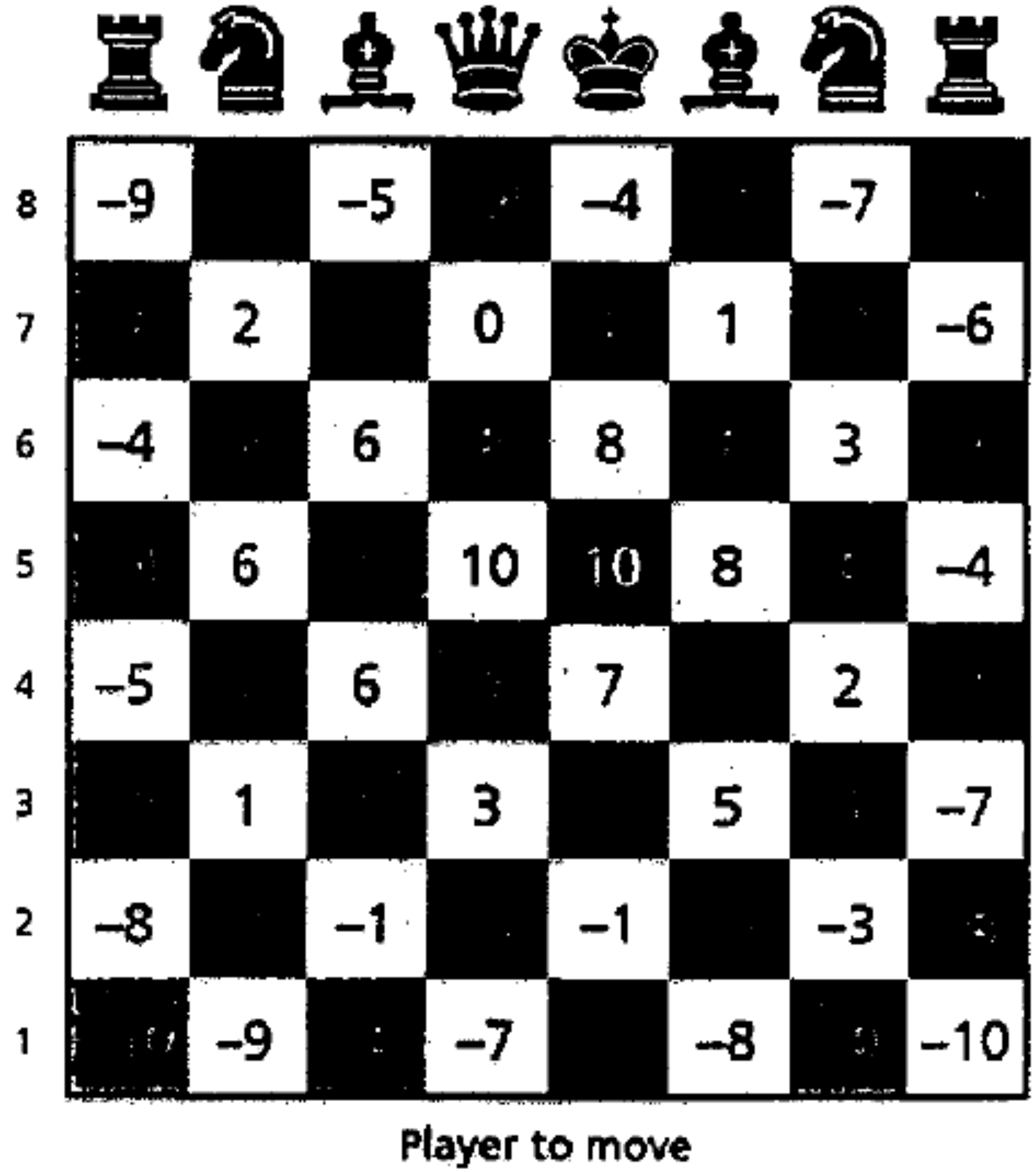


Aspects of sophisticated evaluation functions

- Difference between player and opponent of:
  - # pieces
  - mobility
  - king position
  - bishop pair
  - rook pair
  - control of center (piecewise)
  - etc

Deep Blue

- Deep Blue evaluation function
  - ~6,000 different features
  - weighting depends on board configuration (downloaded int computer after each move)
- Weights are based on:
  - database of ~900 grand maste games
  - tries to adjust weights to match play of grandmasters
- Also adjusted manually by Grand Master Joel Benjamin.



value of knight's position in Deep Blue

Deep Blue’s search

- ~200 million moves / sec
- 3.6 x 10^{10} movies in 3 minutes
  - 7 plies of uniform depth minimax search
  - 10-14 plies of uniform depth alpha-beta search
- software searches first
- specialized hardware searches last 5 ply

Deep Blue's hardware

- 32-node RS6000 SP multicomputer
- Each node
  - 1 IBM Power2 Super Chip
  - 16 chess chips
    - move generation
    - evaluation
    - some end-game heuristics and small end-game databases
- 32 GByte opening & endgame database

Role of computing power

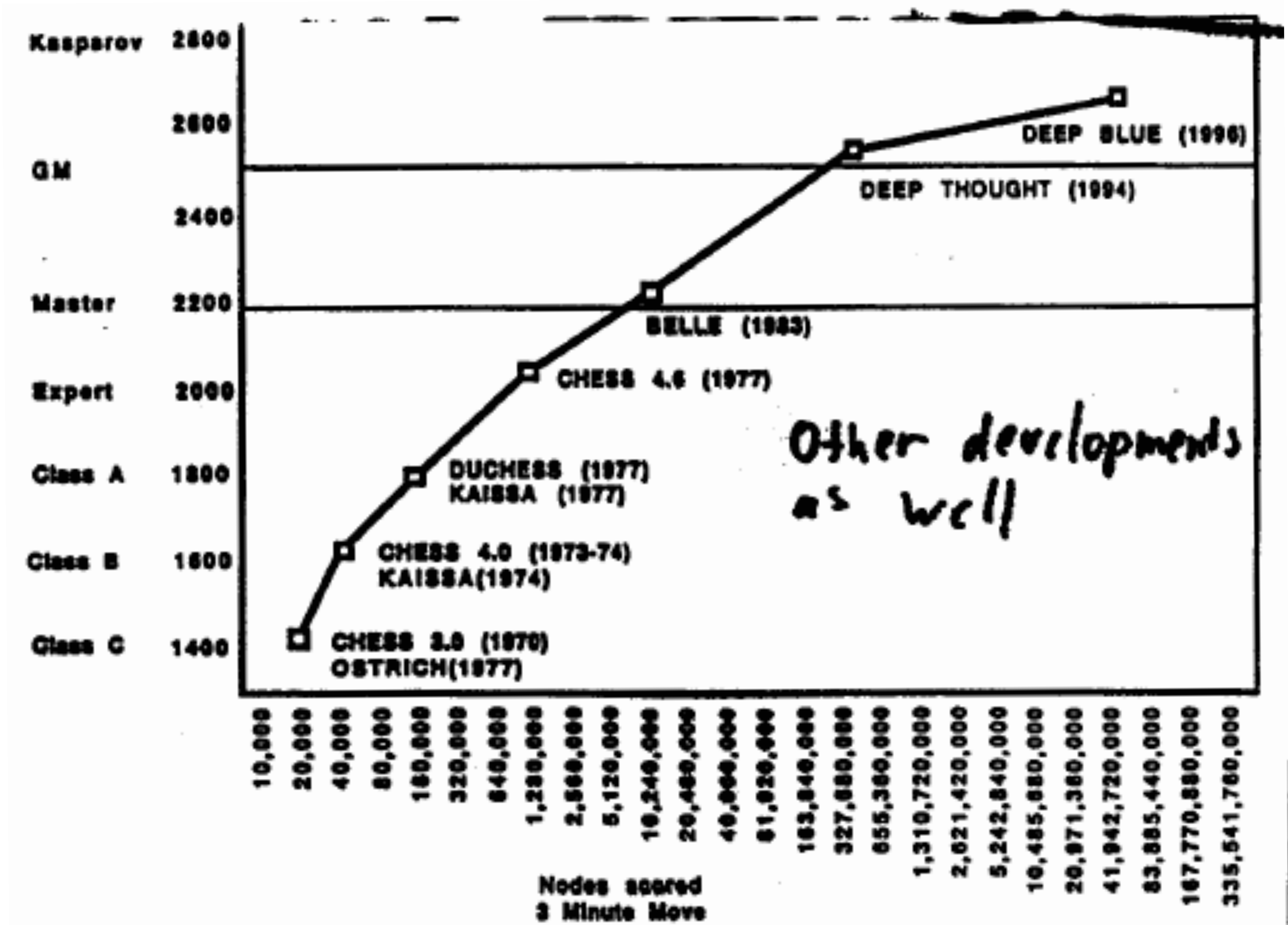


Figure 6.23. Relationship between the level of play by chess programs and the size of the tree searched during a three minute move.

Deep Blue defeats Kasparov, 1997

Deep Fritz defeats Kramnik, 2006*

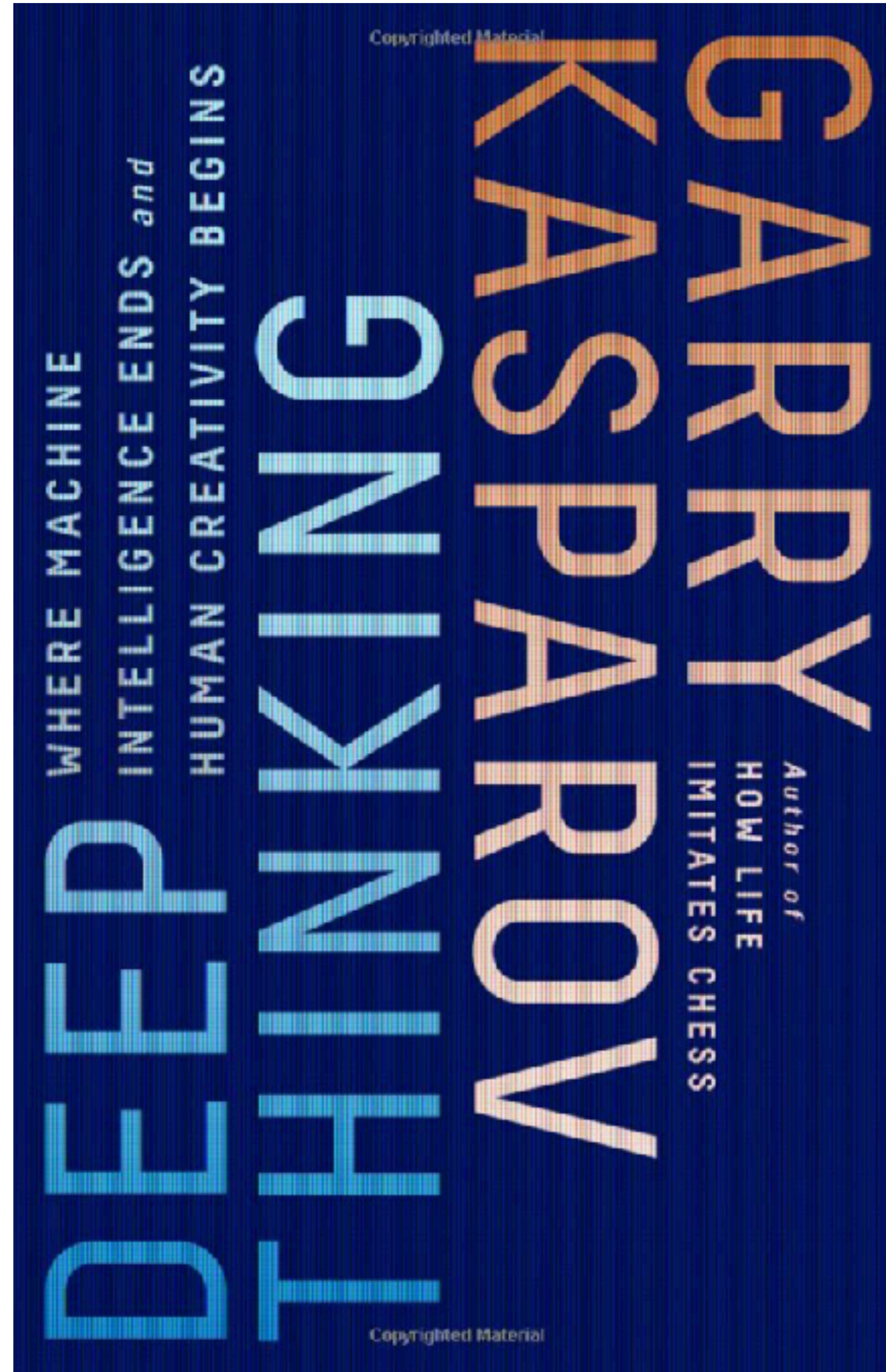
*but Kramnik overlooked a mate-in-one

Kasparov and Deep Blue

http://www.youtube.com/watch?v=NJarxpYyoFI

Kasparov *Business Insider* interview (May 24, 2017)

- Kasparov had just published *Deep Thinking*
  - reflections on AI over the 20 years after Deep Blue
- *Is AI a result or a process?*
  - *result:* Yes. Deep Blue is obviously intelligent because it plays chess at a grand master level
  - *process:* No. It's not human-like intelligence; just a fast "alarm clock."
- On making chess moves:
  - 1% calculation
  - 99% understanding of the game: patterns, previous experience
  - For a machine it's the exact opposite.



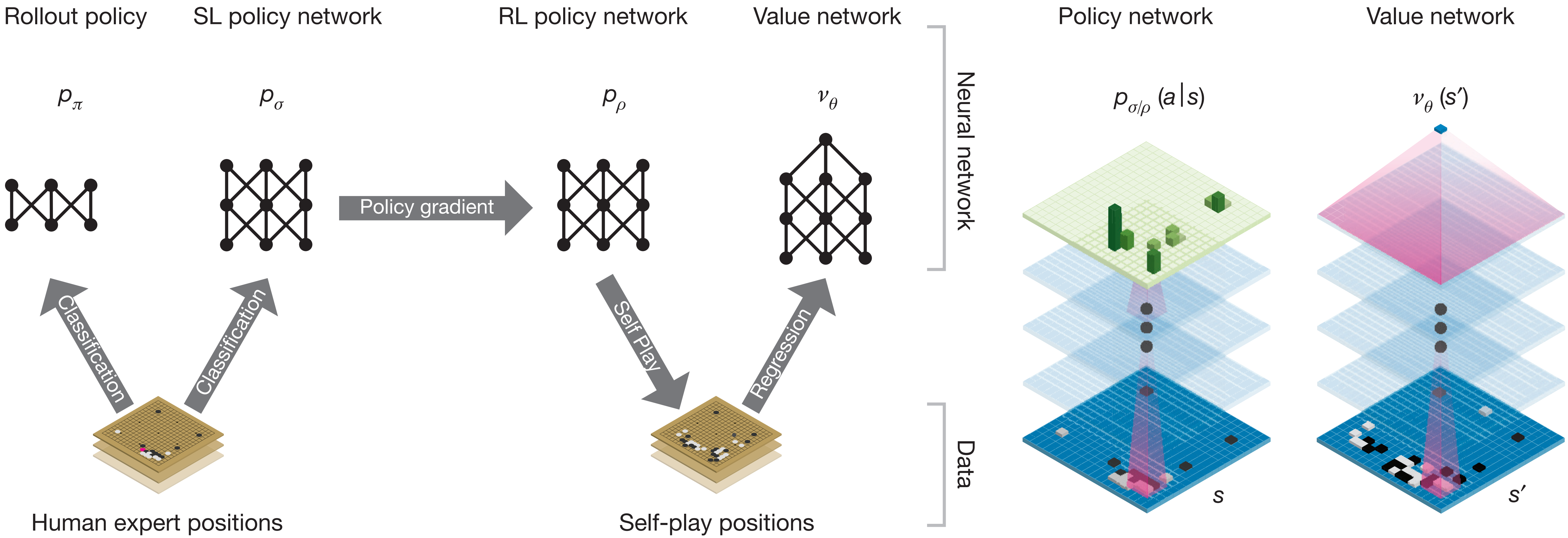
Mastering the game of Go with deep neural networks and tree search

David Silver[1]*, Aja Huang[1]*, Chris J. Maddison[1], Arthur Guez[1], Laurent Sifre[1], George van den Driessche[1], Julian Schrittwieser[1], Ioannis Antonoglou[1], Veda Panneershelvam[1], Marc Lanctot[1], Sander Dieleman[1], Dominik Grewe[1], John Nham[2], Nal Kalchbrenner[1], Ilya Sutskever[2], Timothy Lillicrap[1], Madeleine Leach[1], Koray Kavukcuoglu[1], Thore Graepel[1] & Demis Hassabis[1]

- Alpha Go: *Nature* 28 Jan 2016
- Combined many machine learning techniques (some of which we will cover in future lectures)
- By the end of the course, you should be able to understand the basic approach
- Beat 18-time world champion Lee Sedol in March 2016
- Won 4/5 games.

Alpha Go

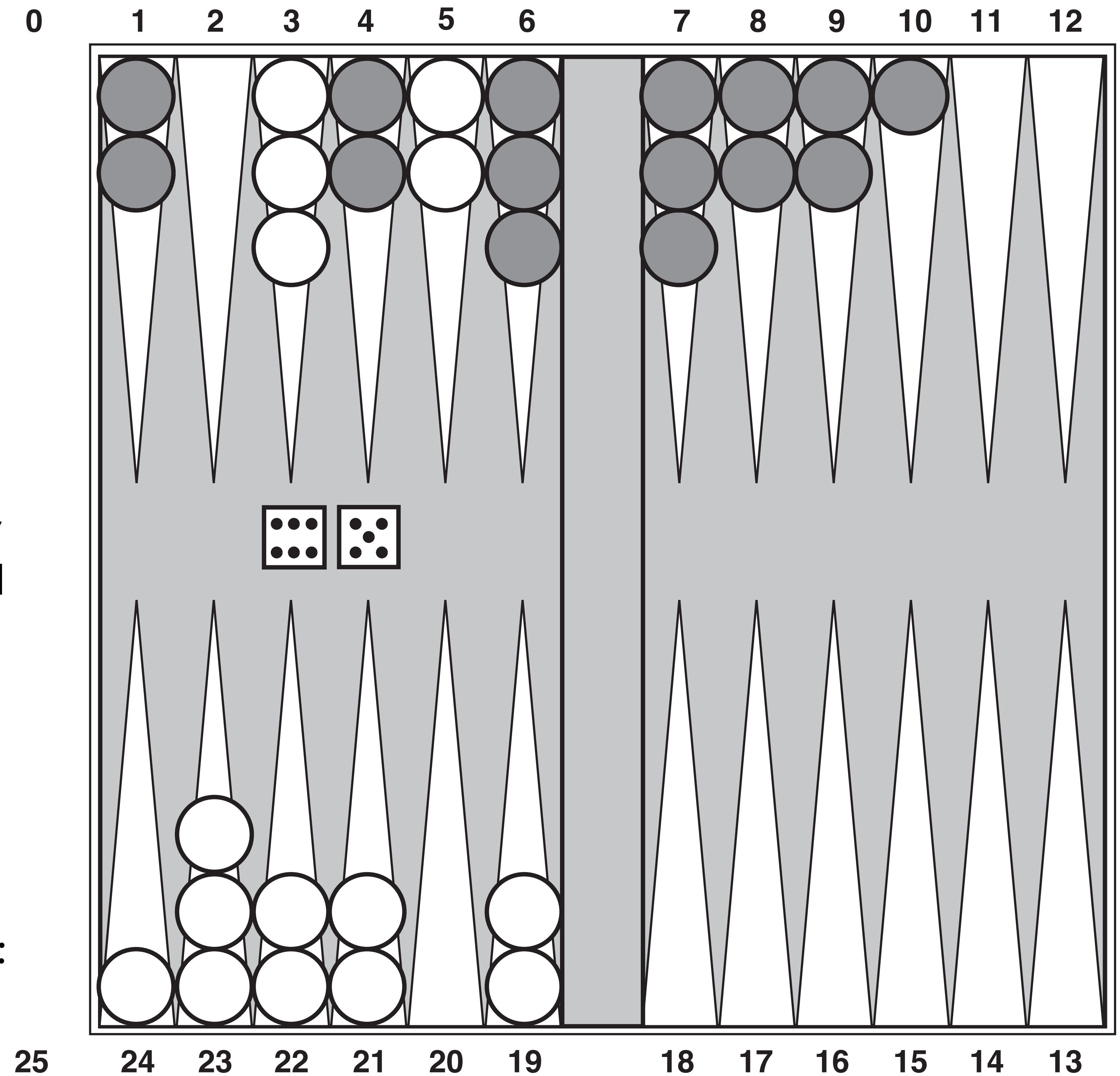
- Key ideas:
  - *learn* what to do instead of using classical search-based techniques
  - use *value* (neural) networks to evaluate board positions
  - use *policy* networks to select moves
- No look-ahead search, but
  - played at the level of state-of-the-art Monte Carlo tree-search algorithms
  - achieved 99.8% winning rate against other Go programs



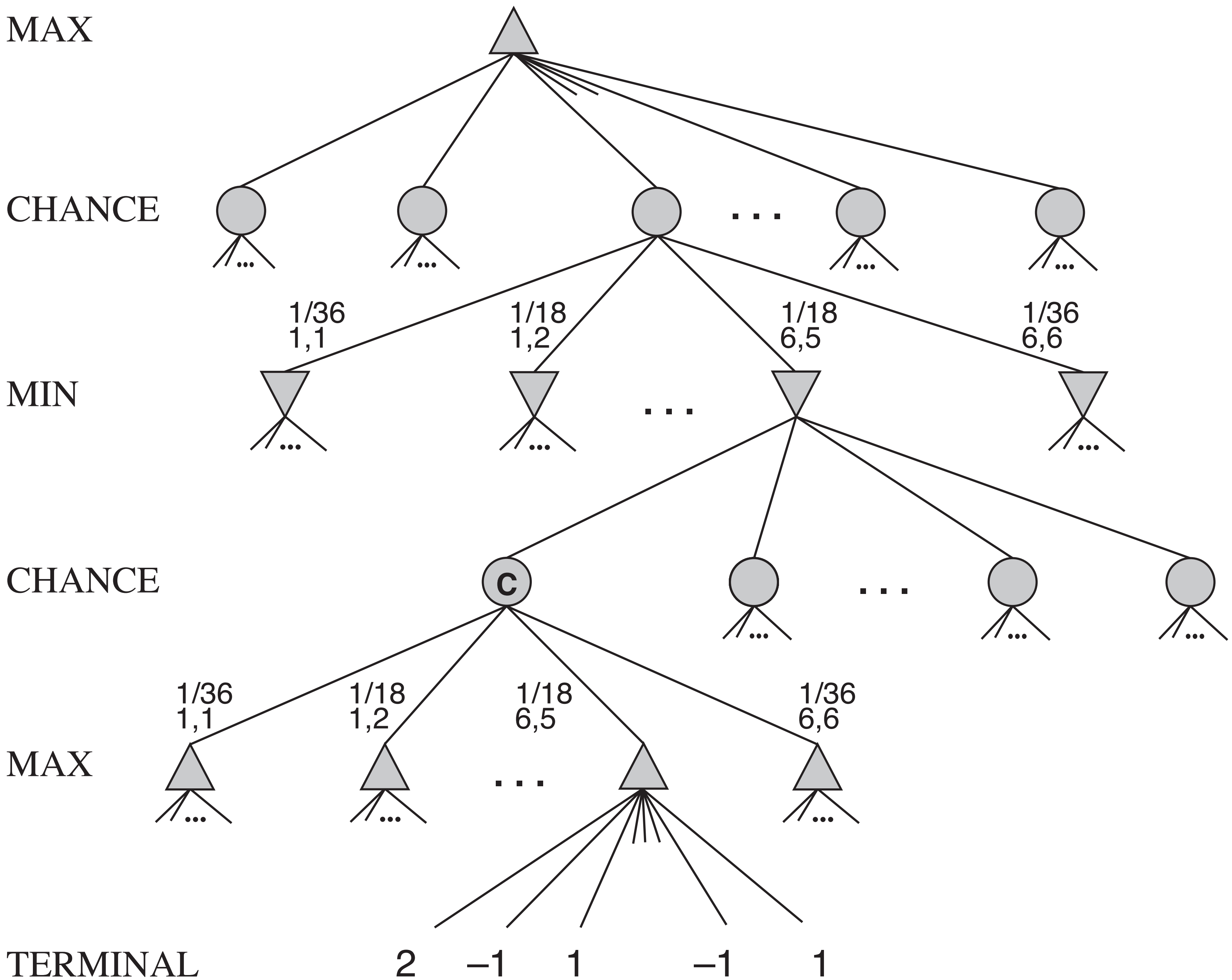
- Use supervised learning to predict moves of Human experts
- The policy network generates p(a|s):
  - probability distribution over legal moves given current board state
- Use reinforcement learning to improve SL policy network
  - generate huge numbers of games for learning using self-play
- Train a value network to predict game outcome (probability of winning)

Stochastic games

- In backgammon, object is to move all pieces off the board.
- White moves clockwise, black counter-clockwise
- Die rolls determine possible moves
- A piece can move to any position, unless occupied by multiple opponents.
- If there is only one, that piece is captured and sent back to start.
- For a roll of 6 & 5, White has 4 legal moves: (5-10,5-11) (5-11,19-24) (5-10,10-16), (5-11,11-16)

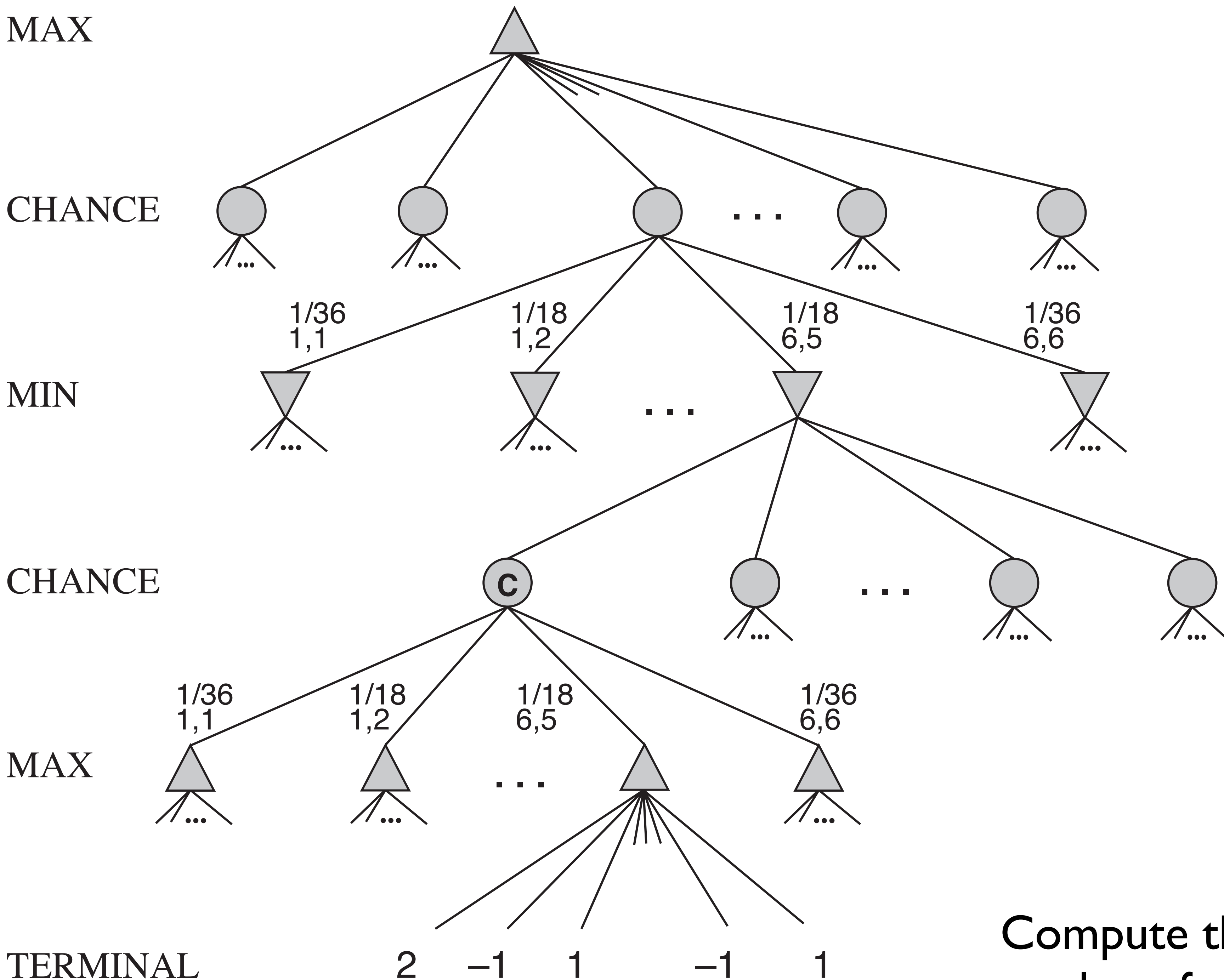


Stochastic game tree for backgammon position



How should we modify the evaluation function to accommodate chance?

Stochastic game tree for backgammon position



Compute the expected value of successors.

$$\mathrm{eval}(s) = \sum_{s' \in \mathrm{succ}(s)} p(s')\,\mathrm{minimax}(s')$$

Properties of stochastic games

- die rolls: 21 possible rolls with two dies ⇒ *much* larger branching factor
- backgammon
  - ≈ 20 legal moves
  - considering opponent responses effective branching factor ≈ 400
- traditional searches are completely ineffective
- α-β pruning isn't very helpful
- Evaluation function becomes much more important
- TD-gammon (1992):
  - depth 2 search
  - learned a very good evaluation function:
    a neural network, learned with temporal different learning
  - knowledge free: achieved intermediate-level play
  - when combined with expert-designed features: world champion level
  - stopped improving after 1.5M games