Adversarial Search

Tomáš Svoboda and Matěj Hoffmann

Vision for Robots and Autonomous Systems, Center for Machine Perception
Department of Cybernetics
Faculty of Electrical Engineering, Czech Technical University in Prague

March 17, 2020

Games, man vs. algorithm

- Deep Blue
- Alpha Go
- Deep Stack
- Why Games, actually?

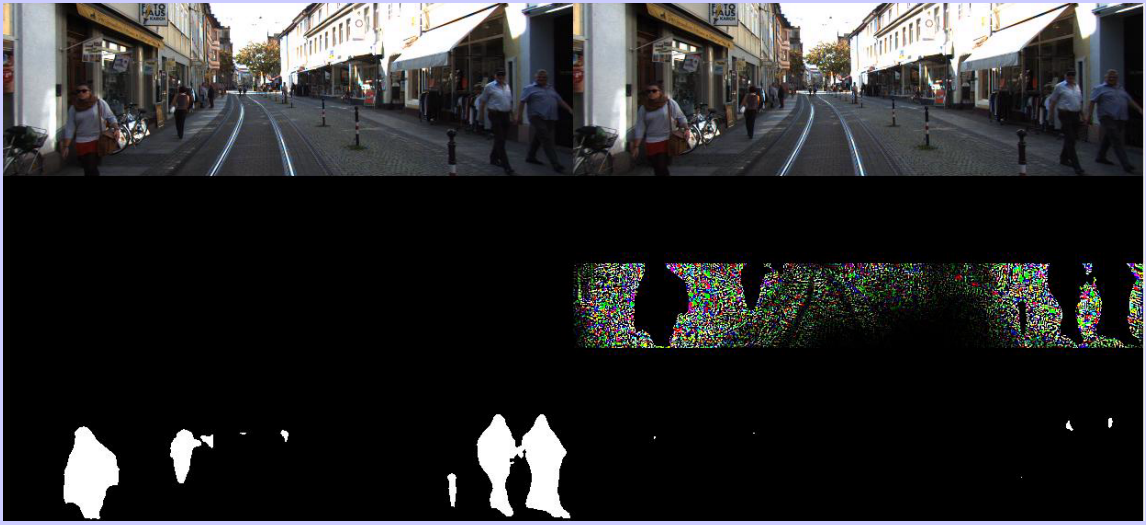
Games are interesting for AI *because* they are hard (to solve).

Games, man vs. algorithm

- Deep Blue
- Alpha Go
- Deep Stack
- Why Games, actually?

Games are interesting for AI *because* they are hard (to solve).

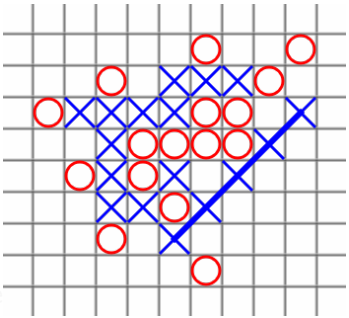
More: Adversarial Learning



Video: Adversing visual segmentation

Elements of the game

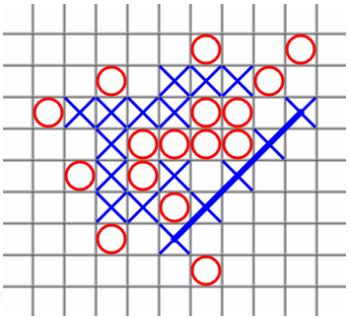
- s_0: The initial state
- PLAYER(s): Which player has to move in s.
- ACTIONS(s): What are the legal moves?
- RESULT(s, a): Transition, result of a move.
- TERMINAL-TEST(s): Game over?
- TERMINAL-UTILITY(s, p): What is prize? Examples for some games ...



https://commons.wikimedia.org/wiki/File:Tic-tac-toe_5.png

Elements of the game

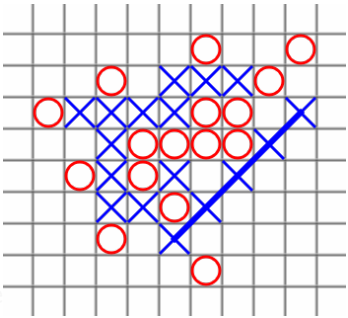
- s_0: The initial state
- PLAYER(s). Which player has to move in s.
- ACTIONS(s). What are the legal moves?
- RESULT(s, a). Transition, result of a move.
- TERMINAL-TEST(s). Game over?
- TERMINAL-UTILITY(s, p). What is prize? Examples for some games ...



https://commons.wikimedia.org/wiki/File:Tic-tac-toe_5.png

Elements of the game

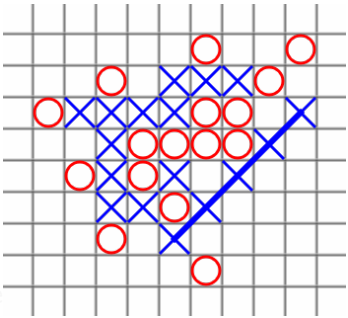
- s_0: The initial state
- PLAYER(s). Which player has to move in s.
- ACTIONS(s). What are the legal moves?
- RESULT(s, a). Transition, result of a move.
- TERMINAL-TEST(s). Game over?
- TERMINAL-UTILITY(s, p). What is prize? Examples for some games ...



https://commons.wikimedia.org/wiki/File:Tic-tac-toe_5.png

Elements of the game

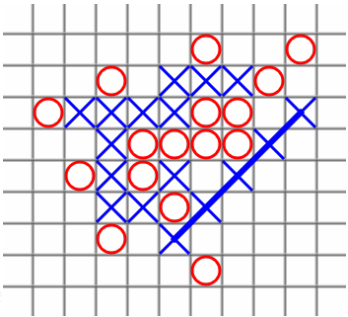
- s_0: The initial state
- PLAYER(s). Which player has to move in s.
- ACTIONS(s). What are the legal moves?
- RESULT(s, a). Transition, result of a move.
- TERMINAL-TEST(s). Game over?
- TERMINAL-UTILITY(s, p). What is prize? Examples for some games ...



https://commons.wikimedia.org/wiki/File:Tic-tac-toe_5.png

Elements of the game

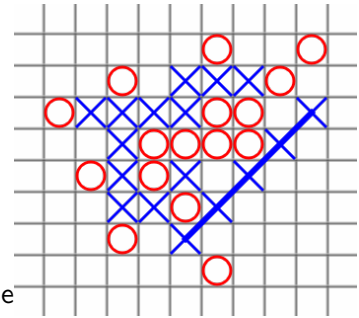
- s_0: The initial state
- PLAYER(s). Which player has to move in s.
- ACTIONS(s). What are the legal moves?
- RESULT(s, a). Transition, result of a move.
- TERMINAL-TEST(s). Game over?
- TERMINAL-UTILITY(s, p). What is prize? Examples for some games ...



https://commons.wikimedia.org/wiki/File:Tic-tac-toe_5.png

Elements of the game

- s_0: The initial state
- PLAYER(s). Which player has to move in s.
- ACTIONS(s). What are the legal moves?
- RESULT(s, a). Transition, result of a move.
- TERMINAL-TEST(s). Game over?
- TERMINAL-UTILITY(s, p). What is prize? Examples for some games ...



https://commons.wikimedia.org/wiki/File:Tic-tac-toe_5.png

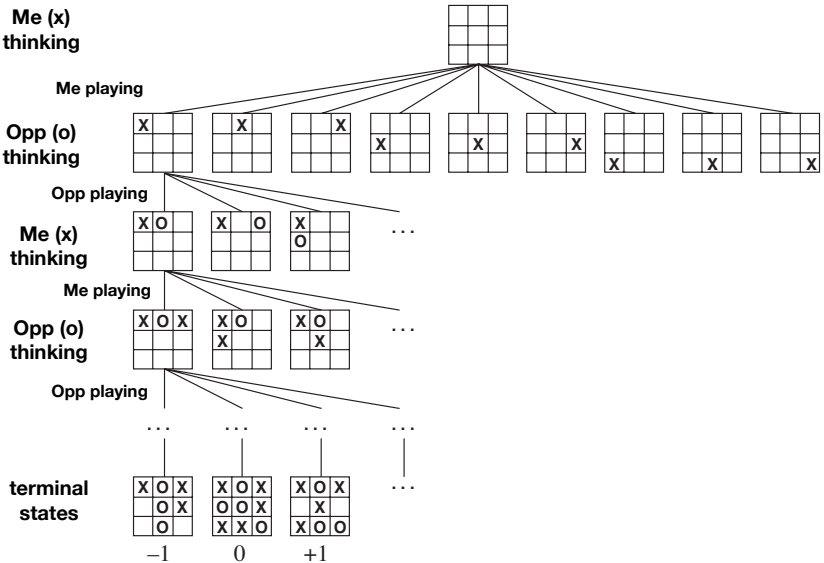
Terminal utilitity: Zero–Sum and General games

- Zero-sum: players have opposite utilities (values)
- Zero-sum: playing against opponent
- General game: independent utilities
- General game: cooperations, competition, ...

Terminal utilitity: Zero–Sum and General games

- Zero-sum: players have opposite utilities (values)
- Zero-sum: playing against opponent
- General game: independent utilities
- General game: cooperations, competition, ...

Game Tree(s)



TERMINAL-UTILITY$(s, \mathbf{x})$

State Value $V(s)$

$V(s)$ – value V of a state s : The best utility achievable from this state.

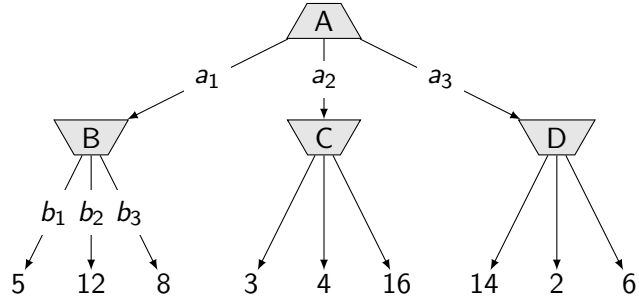
$$V(s) = \max_{s' \in \text{children}(s)} V(s')$$

State Value $V(s)$

$V(s)$ – value V of a state s : The best utility achievable from this state.

$$V(s) = \max_{s' \in \text{children}(s)} V(s')$$

What is the Value of the root $V(A)$?



$V(s)$ – value V of a state s : The best utility achievable from this state.

A, B, C, D - states of the game. I begin, values represent values of terminal states, more is better for me - think about the (my) money prize. Assume (strictly) rational players.

A: $V(A) = 5$

B: $V(A) = 3$

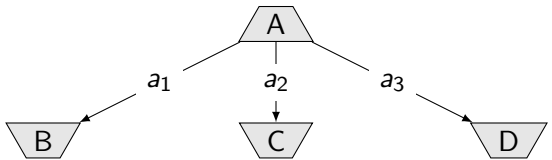
C: $V(A) = 2$

D: $V(A) = 16$

Two-ply game: **max** for me, **min** for the opponent.

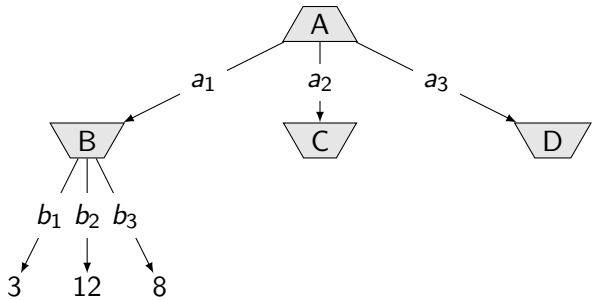
$$a_1 = \arg\max_{a \in \text{ACTIONS}(A)} \text{RESULT}(A, a)$$

Two-ply game: **max** for me, **min** for the opponent.



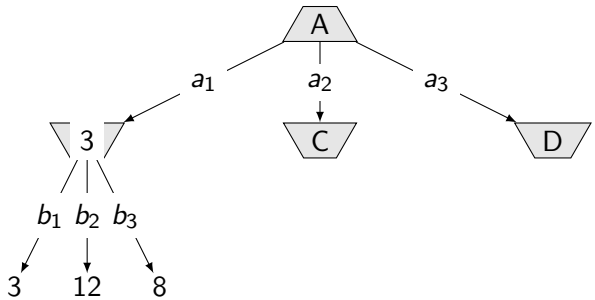
$$a_1 = \underset{a \in \text{ACTIONS}(A)}{\arg\max} \ \text{RESULT}(A, a)$$

Two-ply game: **max** for me, **min** for the opponent.



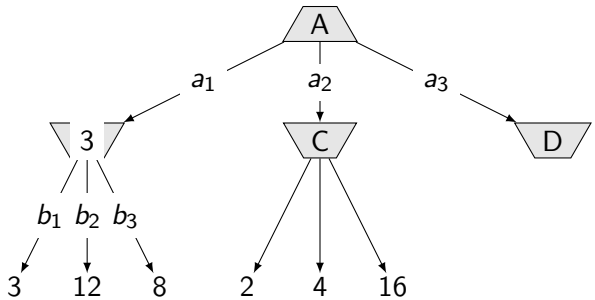
$$a_1 = \underset{a \in \text{ACTIONS}(A)}{\arg\max} \ \text{RESULT}(A, a)$$

Two-ply game: **max** for me, **min** for the opponent.



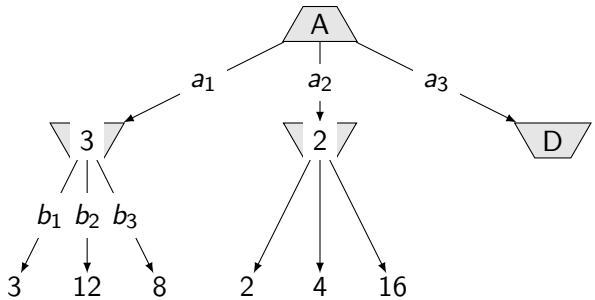
$$a_1 = \underset{a \in \text{ACTIONS}(A)}{\arg\max} \ \text{RESULT}(A, a)$$

Two-ply game: **max** for me, **min** for the opponent.



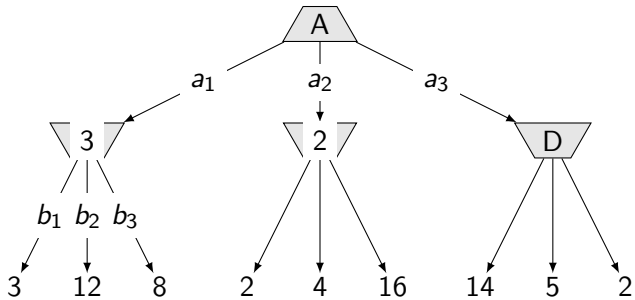
$$a_1 = \underset{a \in \text{ACTIONS}(A)}{\arg\max} \text{ RESULT}(A, a)$$

Two-ply game: **max** for me, **min** for the opponent.



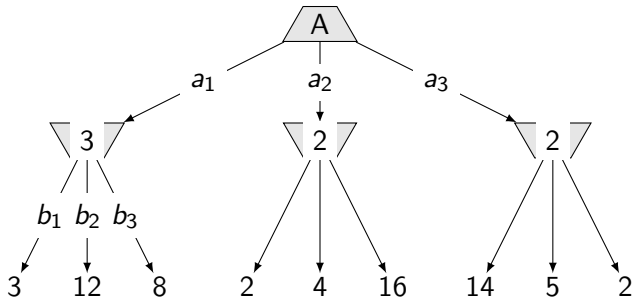
$$a_1 = \underset{a \in \text{ACTIONS}(A)}{\arg\max} \ \text{RESULT}(A, a)$$

Two-ply game: **max** for me, **min** for the opponent.



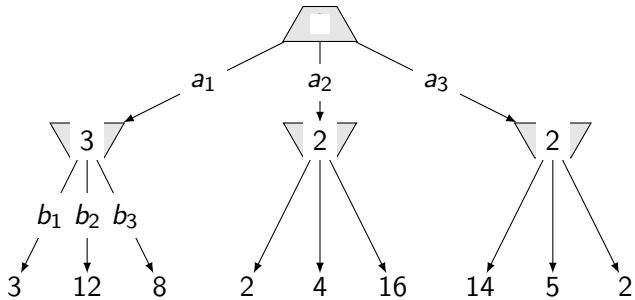
$$a_1 = \underset{a \in \text{ACTIONS}(A)}{\arg\max} \text{RESULT}(A, a)$$

Two-ply game: **max** for me, **min** for the opponent.



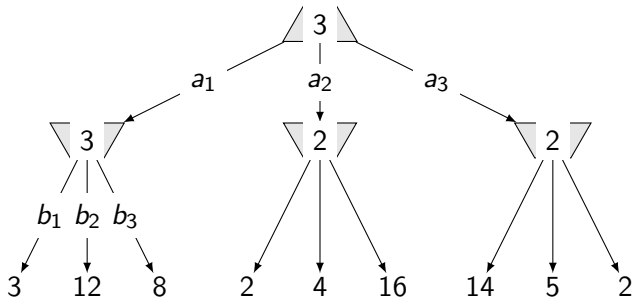
$$a_1 = \underset{a \in \text{ACTIONS}(A)}{\arg\max} \text{ RESULT}(A, a)$$

Two-ply game: **max** for me, **min** for the opponent.



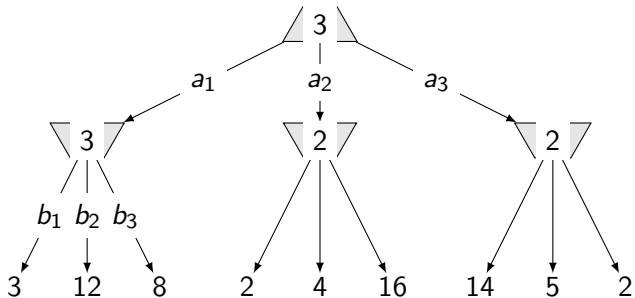
$$a_1 = \arg\max_{a \in \text{ACTIONS}(A)} \text{RESULT}(A, a)$$

Two-ply game: **max** for me, **min** for the opponent.



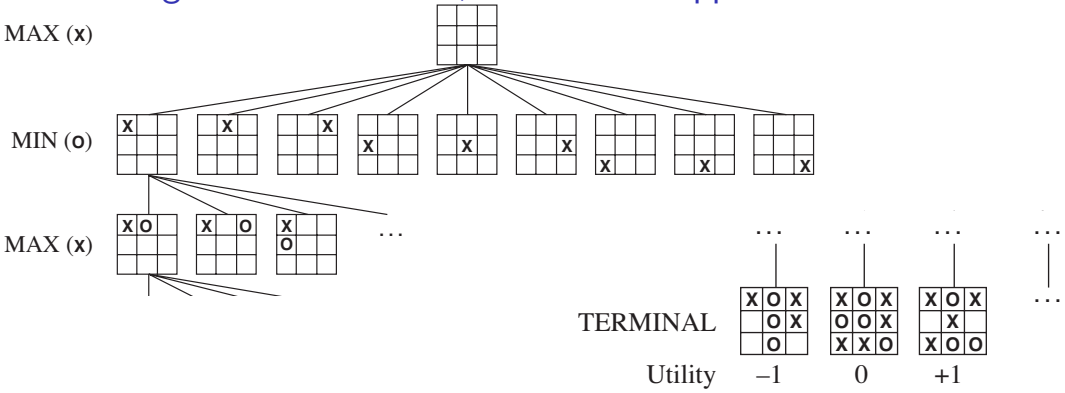
$$a_1 = \underset{a \in \text{ACTIONS}(A)}{\arg\max} \text{RESULT}(A, a)$$

Two-ply game: **max** for me, **min** for the opponent.



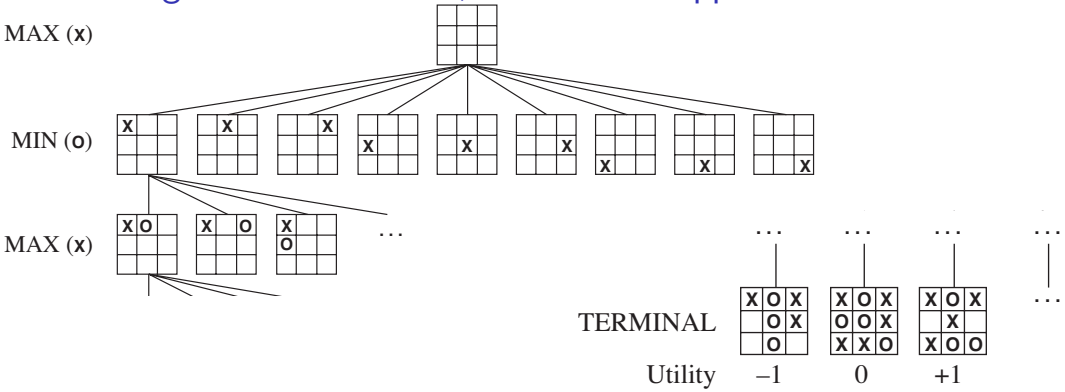
$$a_1 = \underset{a \in \textsc{Actions}(A)}{\arg\max} \textsc{Result}(A, a)$$

Zero-Sum game: **max** for me, **min** for the opponent.



$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in \text{ACTIONS}(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{ACTIONS}(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

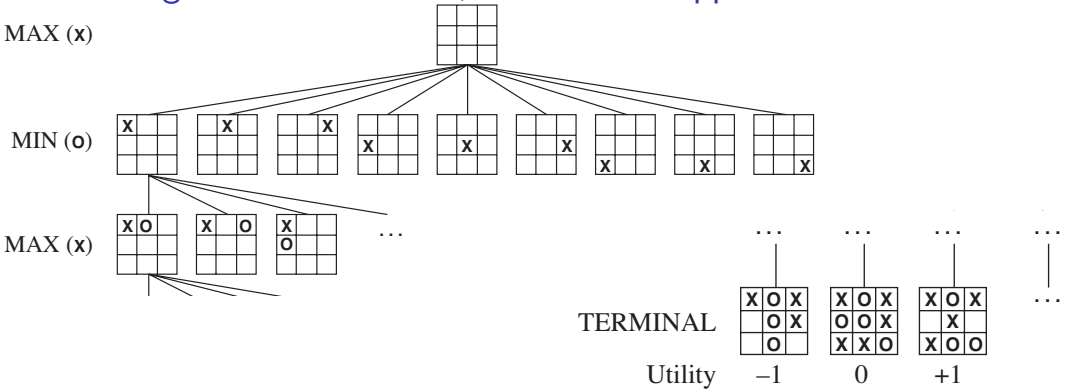
Zero-Sum game: **max** for me, **min** for the opponent.



$$\text{MINIMAX}(s) =$$

$$\begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in \text{ACTIONS}(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{ACTIONS}(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

Zero-Sum game: **max** for me, **min** for the opponent.



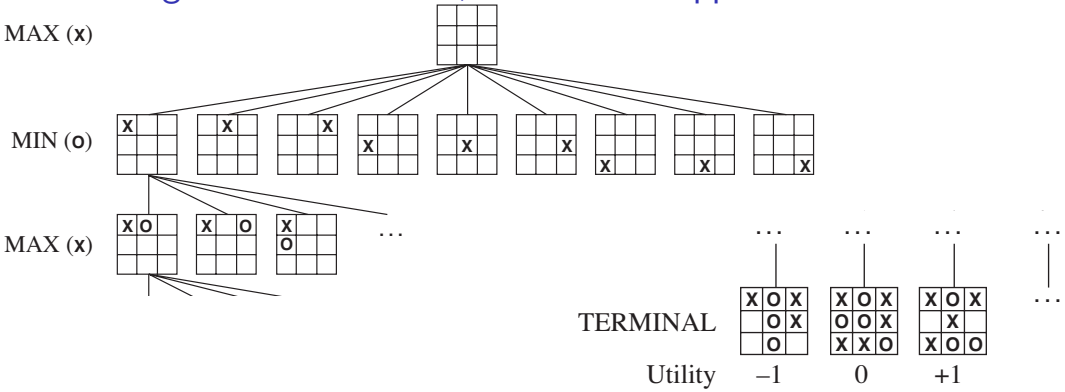
$$\text{MINIMAX}(s) =$$

$$\text{UTILITY}(s) \quad \text{if} \quad \text{TERMINAL-TEST}(s)$$

$$\max_{a \in \text{ACTIONS}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) \quad \text{if} \quad \text{PLAYER}(s) = \text{MAX}$$

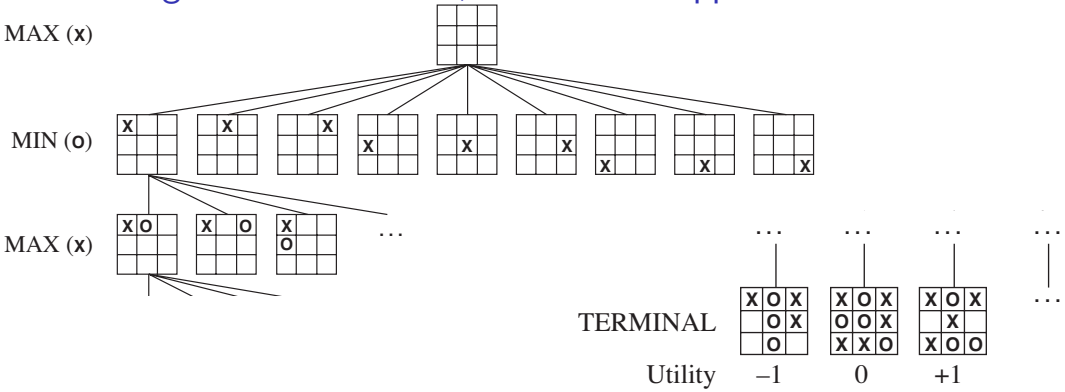
$$\min_{a \in \text{ACTIONS}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) \quad \text{if} \quad \text{PLAYER}(s) = \text{MIN}$$

Zero-Sum game: **max** for me, **min** for the opponent.



$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if } \text{TERMINAL-TEST}(s) \\ \max_{a \in \text{ACTIONS}(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if } \text{PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{ACTIONS}(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if } \text{PLAYER}(s) = \text{MIN} \end{cases}$$

Zero-Sum game: **max** for me, **min** for the opponent.



$$\begin{aligned}
&\text{MINIMAX}(s) = \\
&\quad \begin{cases}
\text{UTILITY}(s) & \text{if } \text{TERMINAL-TEST}(s) \\
\max_{a \in \text{ACTIONS}(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if } \text{PLAYER}(s) = \text{MAX} \\
\min_{a \in \text{ACTIONS}(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if } \text{PLAYER}(s) = \text{MIN}
\end{cases}
\end{aligned}$$

Minimax algorithm

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

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

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

Minimax algorithm

function MINIMAX(state) **returns** an action
  return $\underset{a \in \text{Actions}(s)}{\text{arg}\mathbf{max}}$ MIN-VALUE(RESULT(state, a))
end function

---

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

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

Minimax algorithm

function MINIMAX(state) **returns** an action
  return $\underset{a \in \text{Actions}(s)}{\text{arg}\mathbf{max}}$ MIN-VALUE(RESULT(state, a))
end function

---

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

---

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

Minimax algorithm

function MINIMAX(state) **returns** an action
  return $\underset{a \in \text{Actions}(s)}{\text{arg}\mathbf{max}}$ MIN-VALUE(RESULT(state, a))
end function

---

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

---

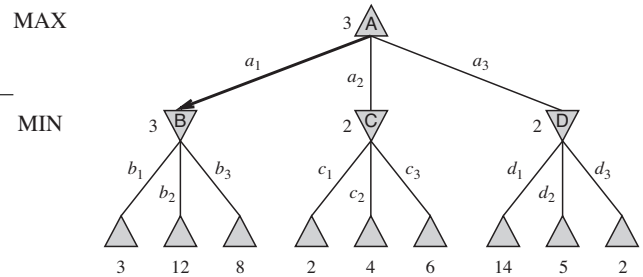
function MAX-VALUE(state) **returns** a utility value v
  if TERMINAL-TEST(state) **then return** UTILITY(state)
  end if
  $v \leftarrow -\infty$
  for all ACTIONS(state) **do**
    $v \leftarrow$ **max**(v, MIN-VALUE(RESULT(state,a)))
  end for
end function

A two ply game, down to terminal and back again . . .

```
function MINIMAX(s) returns a
    argmax_{a∈Actions(s)} MINVAL(RES(s, a))
end function
```

```
function MINVAL(s) returns v
    if TERMINAL(s) then UTIL(s)
    end if
    v ← ∞
    for all ACTIONS(s) do
        v ← min(v, MAXVAL(RES(s, a)))
    end for
end function
```

```
function MAXVAL(s) returns v
    if TERMINAL(s) then UTIL(s)
    end if
    v ← −∞
    for all ACTIONS(s) do
        v ← max(v, MINVAL(RES(s, a)))
    end for
end function
```



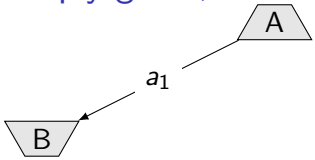
A two ply game, recursive run

Is it like DFS or BFS?

What is the complexity? How many nodes to visit?

Can we do better? How?

A two ply game, recursive run

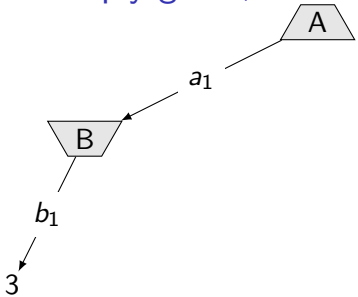


Is it like DFS or BFS?

What is the complexity? How many nodes to visit?

Can we do better? How?

A two ply game, recursive run

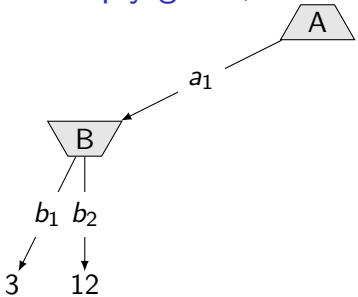


Is it like DFS or BFS?

What is the complexity? How many nodes to visit?

Can we do better? How?

A two ply game, recursive run

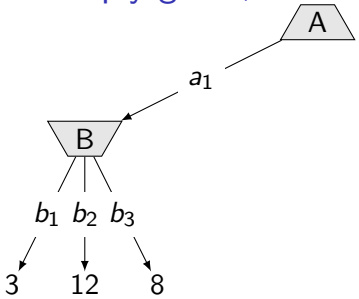


Is it like DFS or BFS?

What is the complexity? How many nodes to visit?

Can we do better? How?

A two ply game, recursive run

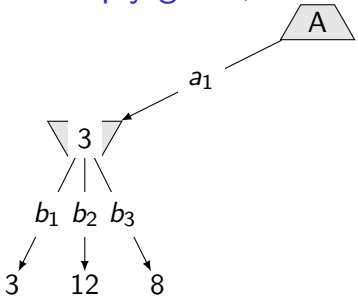


Is it like DFS or BFS?

What is the complexity? How many nodes to visit?

Can we do better? How?

A two ply game, recursive run

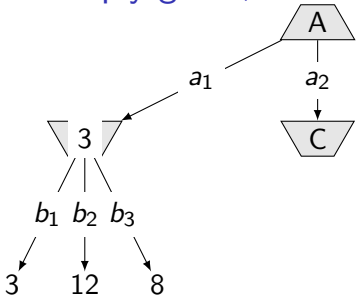


Is it like DFS or BFS?

What is the complexity? How many nodes to visit?

Can we do better? How?

A two ply game, recursive run

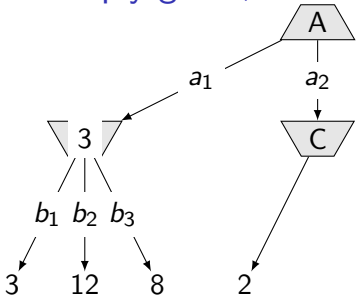


Is it like DFS or BFS?

What is the complexity? How many nodes to visit?

Can we do better? How?

A two ply game, recursive run

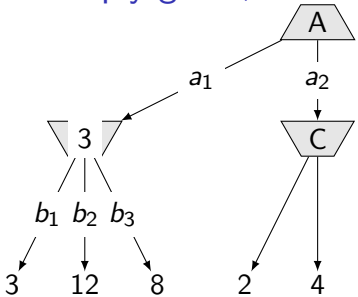


Is it like DFS or BFS?

What is the complexity? How many nodes to visit?

Can we do better? How?

A two ply game, recursive run

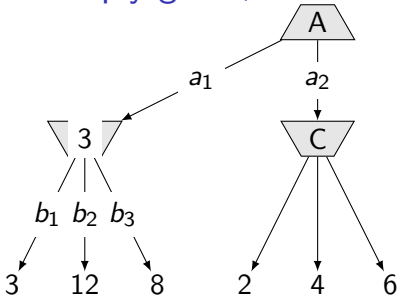


Is it like DFS or BFS?

What is the complexity? How many nodes to visit?

Can we do better? How?

A two ply game, recursive run

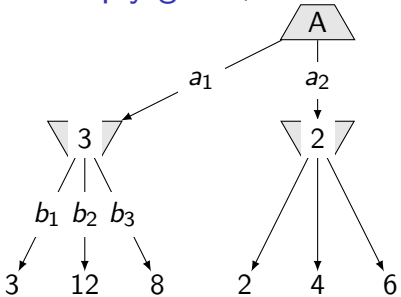


Is it like DFS or BFS?

What is the complexity? How many nodes to visit?

Can we do better? How?

A two ply game, recursive run

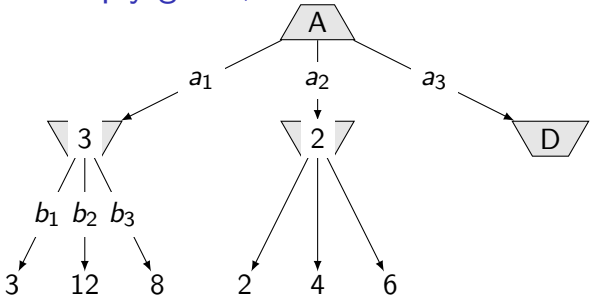


Is it like DFS or BFS?

What is the complexity? How many nodes to visit?

Can we do better? How?

A two ply game, recursive run

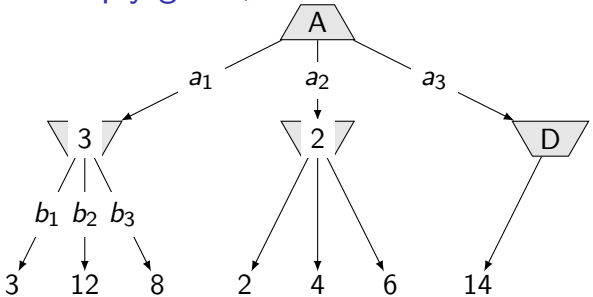


Is it like DFS or BFS?

What is the complexity? How many nodes to visit?

Can we do better? How?

A two ply game, recursive run

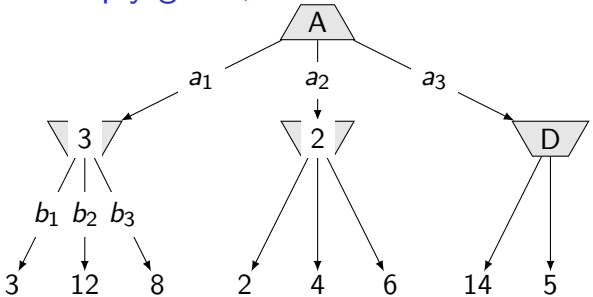


Is it like DFS or BFS?

What is the complexity? How many nodes to visit?

Can we do better? How?

A two ply game, recursive run

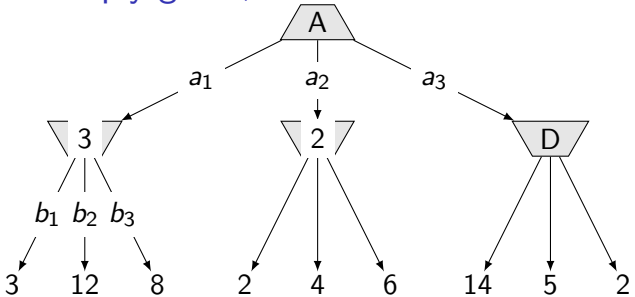


Is it like DFS or BFS?

What is the complexity? How many nodes to visit?

Can we do better? How?

A two ply game, recursive run

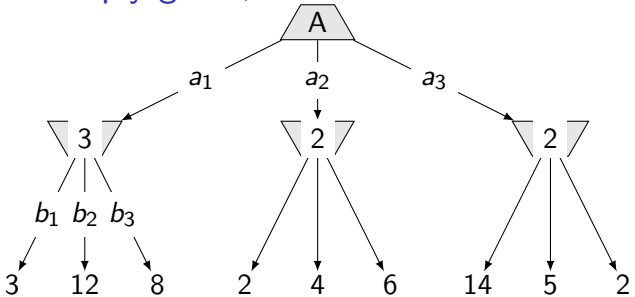


Is it like DFS or BFS?

What is the complexity? How many nodes to visit?

Can we do better? How?

A two ply game, recursive run

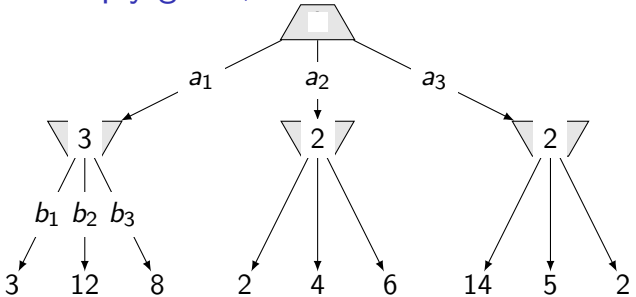


Is it like DFS or BFS?

What is the complexity? How many nodes to visit?

Can we do better? How?

A two ply game, recursive run

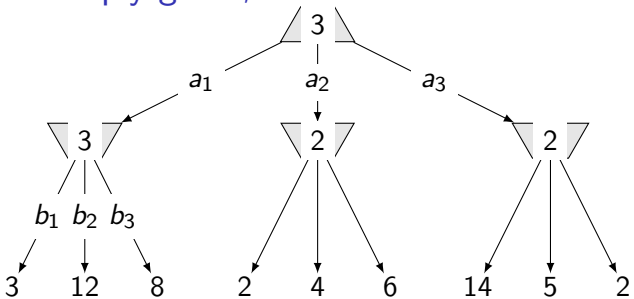


Is it like DFS or BFS?

What is the complexity? How many nodes to visit?

Can we do better? How?

A two ply game, recursive run

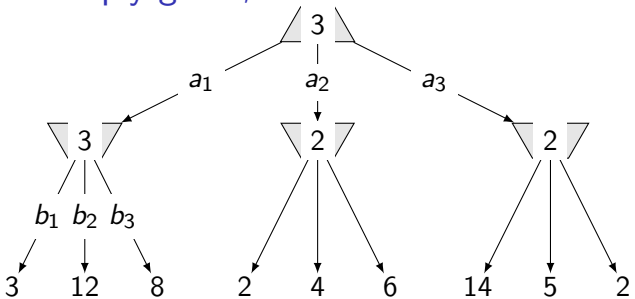


Is it like DFS or BFS?

What is the complexity? How many nodes to visit?

Can we do better? How?

A two ply game, recursive run

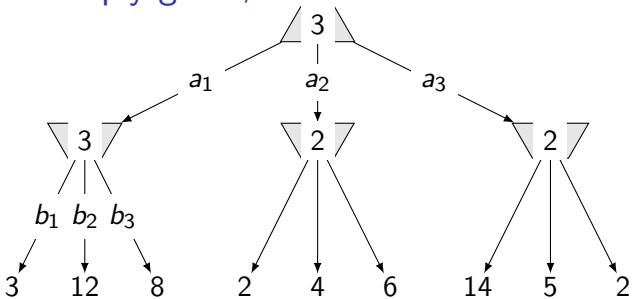


Is it like DFS or BFS?

What is the complexity? How many nodes to visit?

Can we do better? How?

A two ply game, recursive run

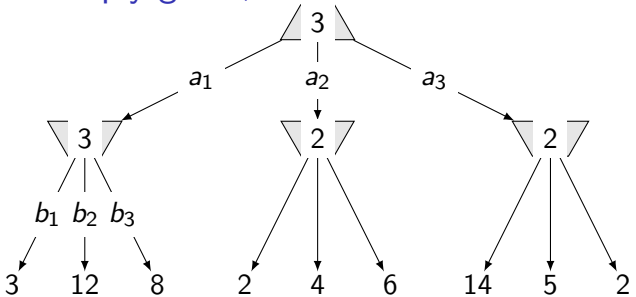


Is it like DFS or BFS?

What is the complexity? How many nodes to visit?

Can we do better? How?

A two ply game, recursive run



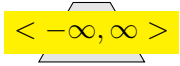
Is it like DFS or BFS?

What is the complexity? How many nodes to visit?

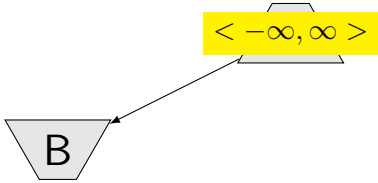
Can we do better? How?

Nodes (sub-trees) worth visiting

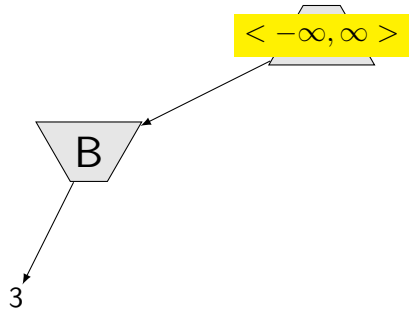
Nodes (sub-trees) worth visiting



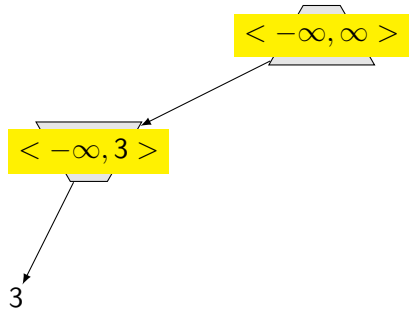
Nodes (sub-trees) worth visiting



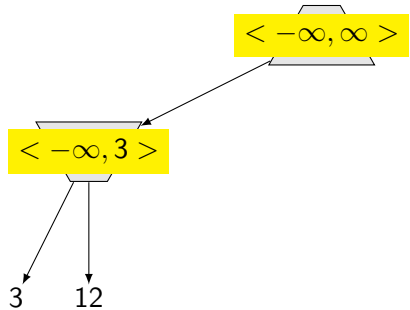
Nodes (sub-trees) worth visiting



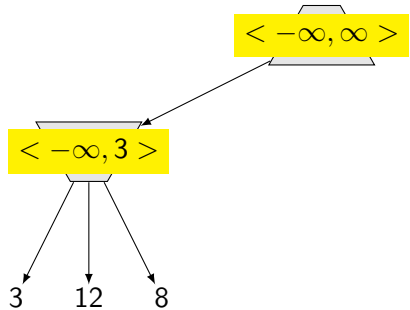
Nodes (sub-trees) worth visiting



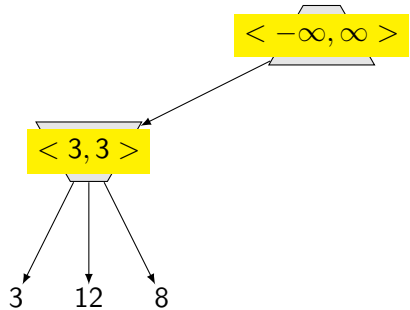
Nodes (sub-trees) worth visiting



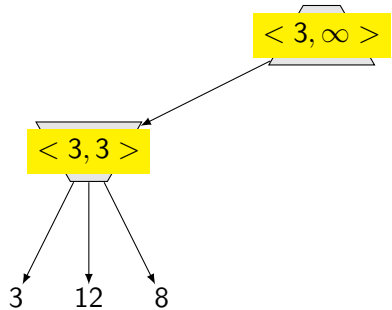
Nodes (sub-trees) worth visiting



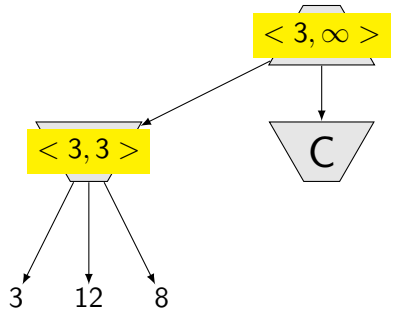
Nodes (sub-trees) worth visiting



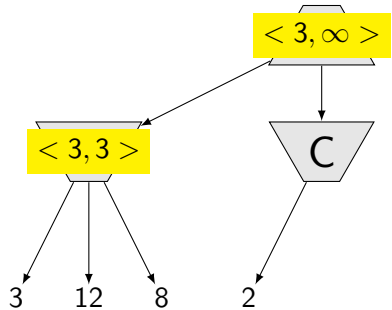
Nodes (sub-trees) worth visiting



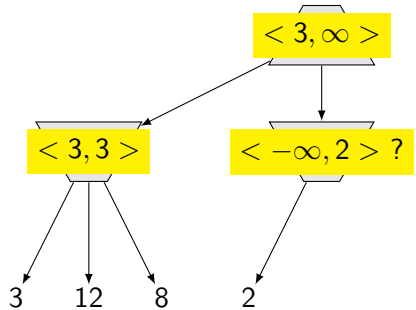
Nodes (sub-trees) worth visiting



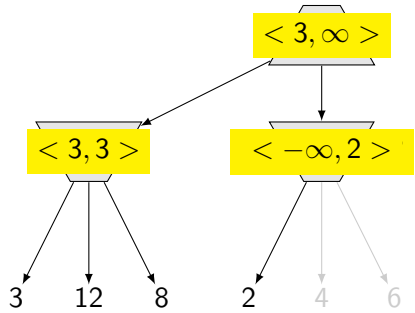
Nodes (sub-trees) worth visiting



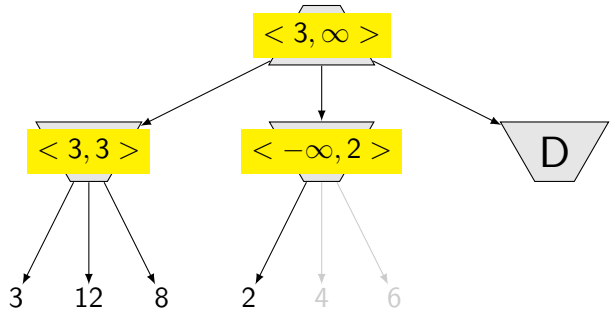
Nodes (sub-trees) worth visiting



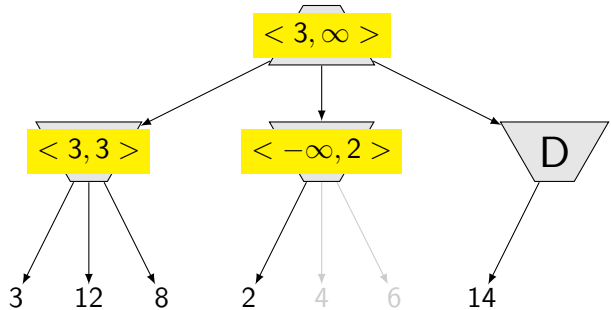
Nodes (sub-trees) worth visiting



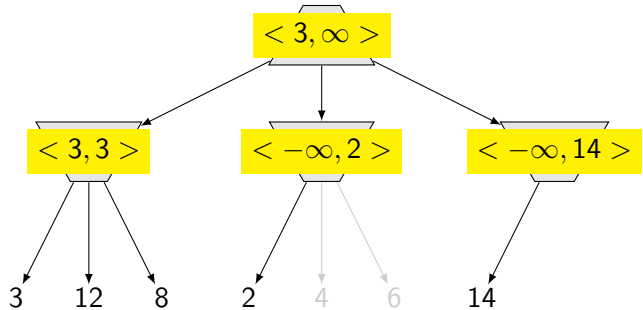
Nodes (sub-trees) worth visiting



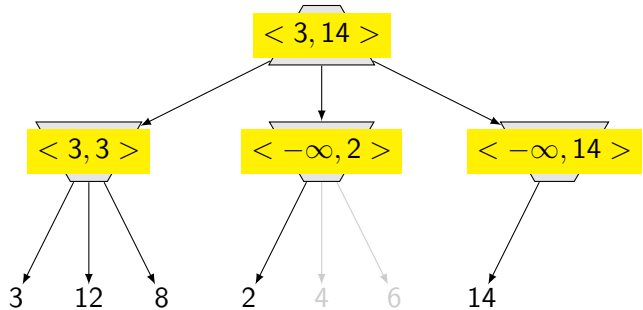
Nodes (sub-trees) worth visiting



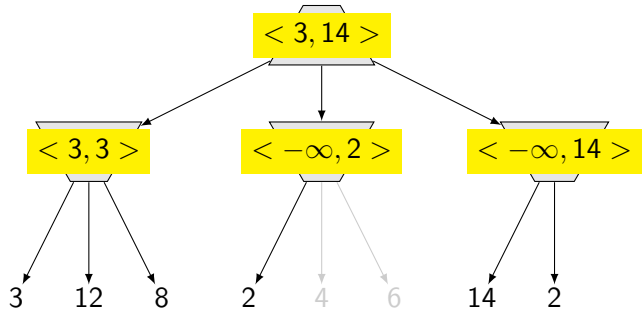
Nodes (sub-trees) worth visiting



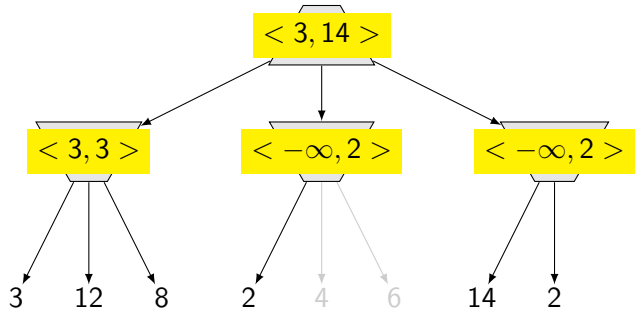
Nodes (sub-trees) worth visiting



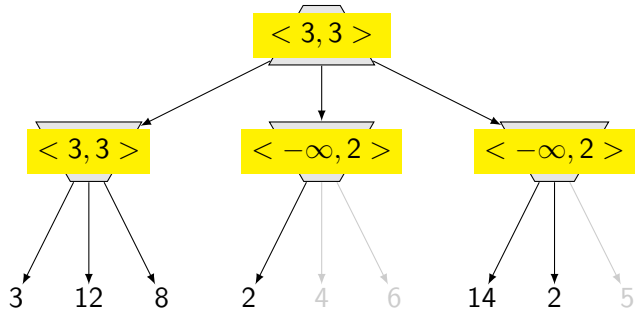
Nodes (sub-trees) worth visiting



Nodes (sub-trees) worth visiting

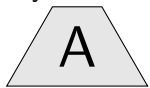


Nodes (sub-trees) worth visiting



α-β pruning

α highest (best) value choice found so far for any choice along MAX
β lowest (best) value choice found so far for any choice along MIN

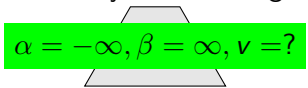


v value of the state

In MIN-VAL: $v \leftarrow 2$
$v \leq \alpha$ then: return v!

α-β pruning

α highest (best) value choice found so far for any choice along MAX
β lowest (best) value choice found so far for any choice along MIN

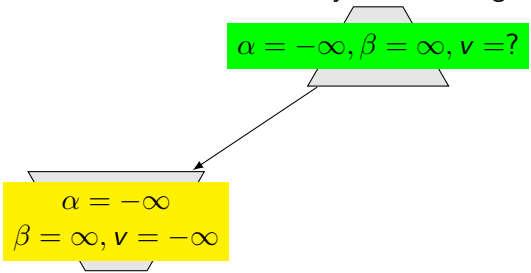


v value of the state

In MIN-VAL: $v \leftarrow 2$
$v \leq \alpha$ then: return v!

α-β pruning

α highest (best) value choice found so far for any choice along MAX
β lowest (best) value choice found so far for any choice along MIN

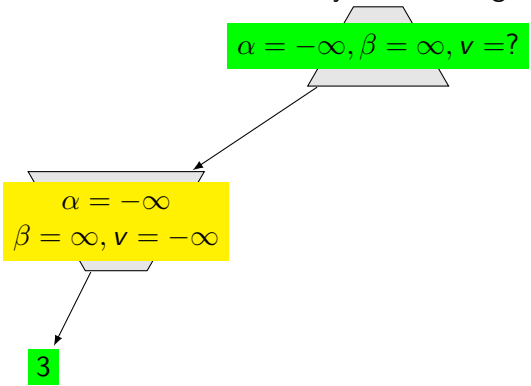


v value of the state

In MIN-VAL: $v \leftarrow 2$
$v \leq \alpha$ then: return v!

α-β pruning

α highest (best) value choice found so far for any choice along MAX
β lowest (best) value choice found so far for any choice along MIN

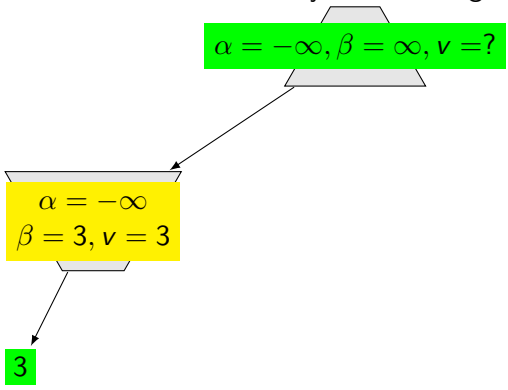


v value of the state

In MIN-VAL: $v \leftarrow 2$
$v \leq \alpha$ then: return v!

α-β pruning

α highest (best) value choice found so far for any choice along MAX
β lowest (best) value choice found so far for any choice along MIN

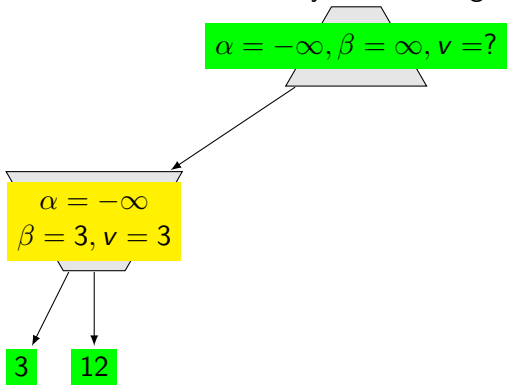


v value of the state

In MIN-VAL: $v \leftarrow 2$
$v \leq \alpha$ then: return v!

α-β pruning

α highest (best) value choice found so far for any choice along MAX
β lowest (best) value choice found so far for any choice along MIN

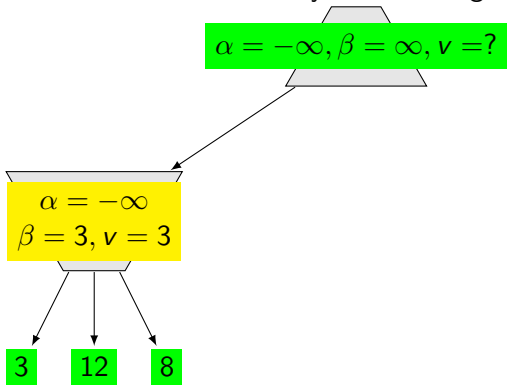


v value of the state

In MIN-VAL: $v \leftarrow 2$
$v \leq \alpha$ then: return v!

α-β pruning

α highest (best) value choice found so far for any choice along MAX
β lowest (best) value choice found so far for any choice along MIN

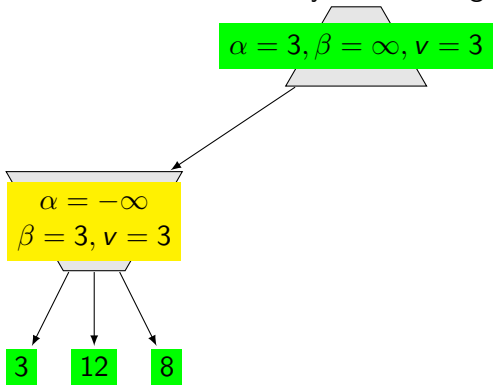


v value of the state

In MIN-VAL: $v \leftarrow 2$
$v \leq \alpha$ then: return v!

α-β pruning

α highest (best) value choice found so far for any choice along MAX
β lowest (best) value choice found so far for any choice along MIN

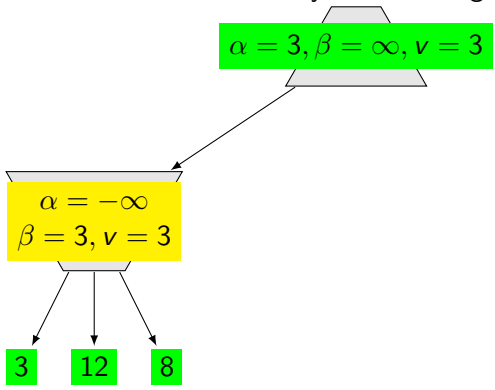


v value of the state

In MIN-VAL: $v \leftarrow 2$
$v \leq \alpha$ then: return v!

α-β pruning

α highest (best) value choice found so far for any choice along MAX
β lowest (best) value choice found so far for any choice along MIN

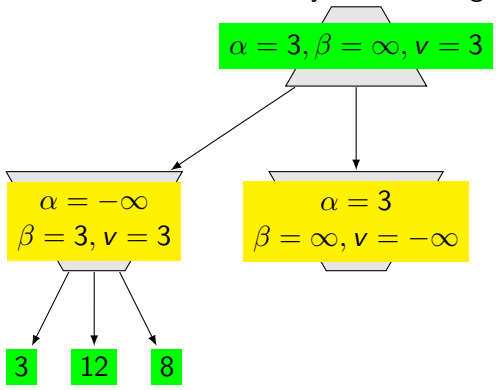


v value of the state

In MIN-VAL: $v \leftarrow 2$
$v \leq \alpha$ then: return v!

α-β pruning

α highest (best) value choice found so far for any choice along MAX
β lowest (best) value choice found so far for any choice along MIN

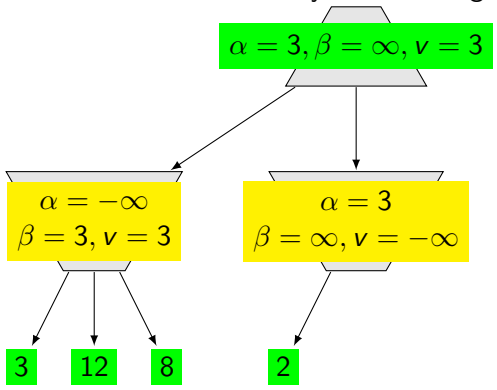


v value of the state

In MIN-VAL: $v \leftarrow 2$
$v \leq \alpha$ then: return v!

α-β pruning

α highest (best) value choice found so far for any choice along MAX
β lowest (best) value choice found so far for any choice along MIN

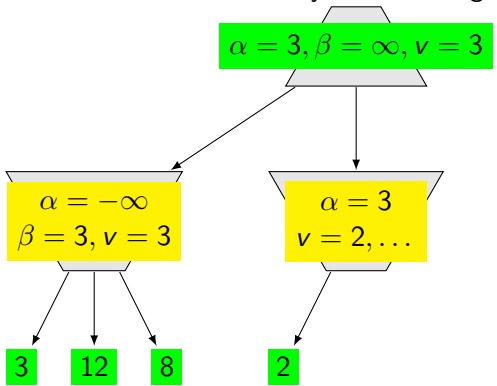


v value of the state

In MIN-VAL: $v \leftarrow 2$
$v \leq \alpha$ then: return v!

α-β pruning

α highest (best) value choice found so far for any choice along MAX
β lowest (best) value choice found so far for any choice along MIN



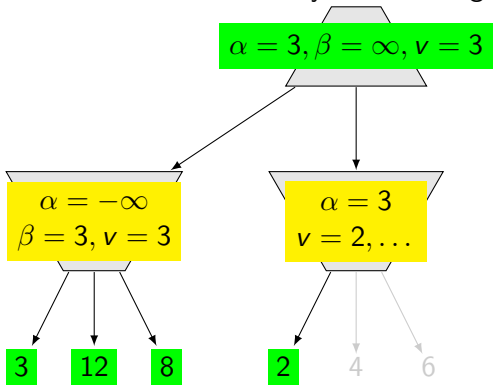
v value of the state

In MIN-VAL: $v \leftarrow 2$

$v \leq \alpha$ then: return v!

α-β pruning

α highest (best) value choice found so far for any choice along MAX
β lowest (best) value choice found so far for any choice along MIN

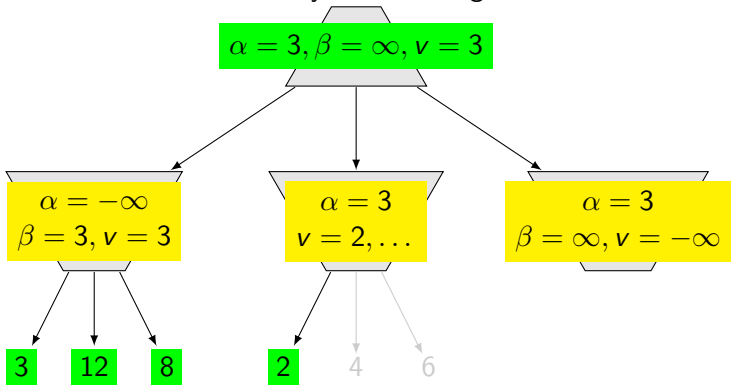


v value of the state

In MIN-VAL: $v \leftarrow 2$
$v \leq \alpha$ then: return v!

α-β pruning

α highest (best) value choice found so far for any choice along MAX
β lowest (best) value choice found so far for any choice along MIN

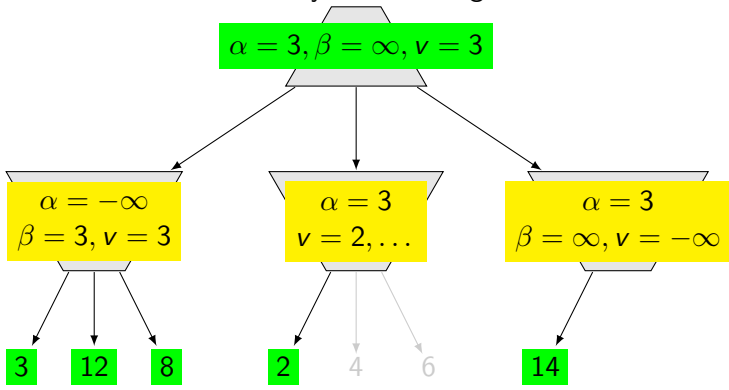


v value of the state

In MIN-VAL: $v \leftarrow 2$
$v \leq \alpha$ then: return v!

α-β pruning

α highest (best) value choice found so far for any choice along MAX
β lowest (best) value choice found so far for any choice along MIN

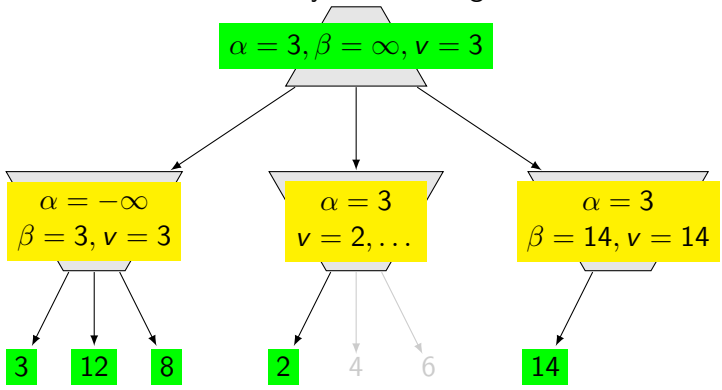


v value of the state

In MIN-VAL: $v \leftarrow 2$
$v \leq \alpha$ then: return v!

α-β pruning

α highest (best) value choice found so far for any choice along MAX
β lowest (best) value choice found so far for any choice along MIN

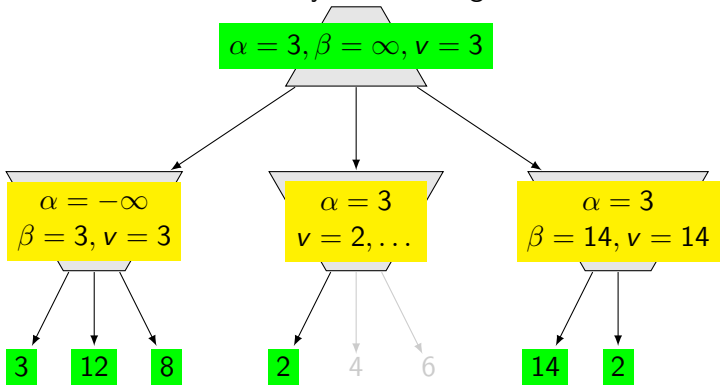


v value of the state

In MIN-VAL: $v \leftarrow 2$
$v \leq \alpha$ then: return v!

α-β pruning

α highest (best) value choice found so far for any choice along MAX
β lowest (best) value choice found so far for any choice along MIN

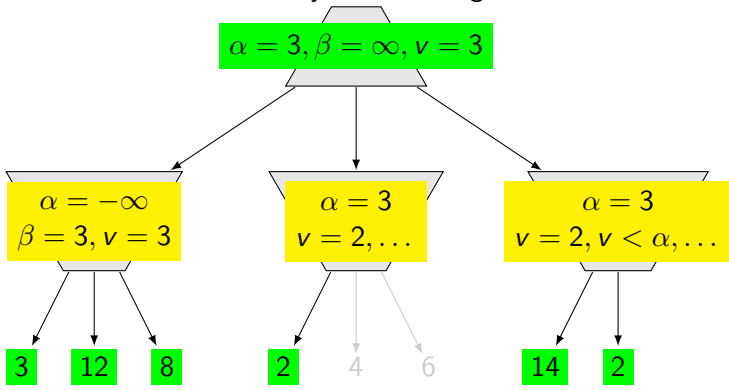


v value of the state

In MIN-VAL: $v \leftarrow 2$
$v \leq \alpha$ then: return v!

α-β pruning

α highest (best) value choice found so far for any choice along MAX
β lowest (best) value choice found so far for any choice along MIN

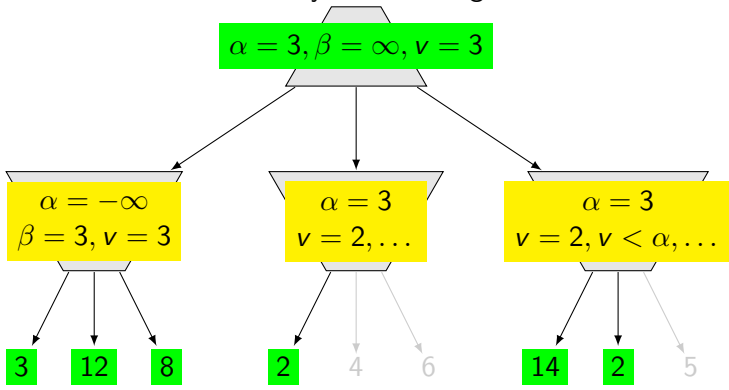


v value of the state

In MIN-VAL: $v \leftarrow 2$
$v \leq \alpha$ then: return v!

α-β pruning

α highest (best) value choice found so far for any choice along MAX
β lowest (best) value choice found so far for any choice along MIN



v value of the state

In MIN-VAL: $v \leftarrow 2$
$v \leq \alpha$ then: return v!

α-β prunnig – How much can we save?

original: Time: $O(b^m)$

- how to consider next actions/moves (in what order)?
- perfect ordering?

function ALPHA-BETA-SEARCH(state) **returns** an action
  $v \leftarrow$ MAX-VALUE(state, $\alpha = -\infty$, $\beta = \infty$)
  return action corresponding to v
end function

---

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

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

function ALPHA-BETA-SEARCH(state) **returns** an action
  $v \leftarrow$ MAX-VALUE(state, $\alpha = -\infty$, $\beta = \infty$)
  return action corresponding to v
end function

---

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

---

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

```
function ALPHA-BETA-SEARCH(state) returns an action
    v ← MAX-VALUE(state, α = −∞, β = ∞)
    return action corresponding to v
end function
```

```
function MAX-VALUE(state,α, β) returns a utility value v
    if TERMINAL-TEST(state) return UTILITY(state)
    v ← −∞
    for all ACTIONS(state) do
        v ← max(v, MIN-VALUE(RESULT(state,a),α, β))
        if v ≥ β return v
        α ← max(α, v)
    end for
end function
```

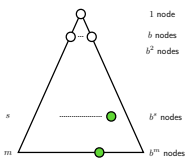
```
function MIN-VALUE(state, α, β) returns a utility value v
    if TERMINAL-TEST(state) return UTILITY(state)
    v ← ∞
    for all ACTIONS(state) do
        v ← min(v, MAX-VALUE(RESULT(state,a),α, β))
        if v ≤ α return v
        β ← min(β, v)
    end for
```

Recall: Iterative deepening DFS (ID-DFS)

- Start with `maxdepth = 1`
- Perform DFS with limited depth. Report success or failure.
- If failure, forget everything, increase `maxdepth` and repeat DFS

The "wasting" of resources is not too bad. Recall:

- Most nodes are at the deepest levels.
- Asymptotic complexity unchanged.



Bonus for α-β pruning: previous "shallower" iterations can be reused for node ordering.

Imperfect but real-time decisions: iterative deepening

$$\text{H-MINIMAX}(s, d) =$$

$$\begin{cases} \text{EVAL}(s) & \text{if } \text{CUTOFF-TEST}(s, d) \\ \max_{a \in \text{ACTIONS}(s)} \text{H-MINIMAX}(\text{RESULT}(s, a), d+1) & \text{if } \text{PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{ACTIONS}(s)} \text{H-MINIMAX}(\text{RESULT}(s, a, d+1)) & \text{if } \text{PLAYER}(s) = \text{MIN} \end{cases}$$

Imperfect but real-time decisions: iterative deepening

$$\text{H-MINIMAX}(s, d) =$$
$$\text{EVAL}(s) \quad \text{if} \quad \text{CUTOFF-TEST}(s, d)$$
$$\max_{a \in \text{ACTIONS}(s)} \text{H-MINIMAX}(\text{RESULT}(s, a), d + 1) \quad \text{if} \quad \text{PLAYER}(s) = \text{MAX}$$
$$\min_{a \in \text{ACTIONS}(s)} \text{H-MINIMAX}(\text{RESULT}(s, a, d + 1)) \quad \text{if} \quad \text{PLAYER}(s) = \text{MIN}$$

Imperfect but real-time decisions: iterative deepening

$$\text{H-MINIMAX}(s, d) = \begin{cases} \text{EVAL}(s) & \text{if } \text{CUTOFF-TEST}(s, d) \\ \max_{a \in \text{ACTIONS}(s)} \text{H-MINIMAX}(\text{RESULT}(s, a), d+1) & \text{if } \text{PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{ACTIONS}(s)} \text{H-MINIMAX}(\text{RESULT}(s, a), d+1) & \text{if } \text{PLAYER}(s) = \text{MIN} \end{cases}$$

Imperfect but real-time decisions: iterative deepening

$$
\begin{aligned}
&\text{H-MINIMAX}(s, d) = \\
&\qquad \text{EVAL}(s) \quad \text{if} \quad \text{CUTOFF-TEST}(s, d) \\
&\max_{a \in \text{ACTIONS}(s)} \text{H-MINIMAX}(\text{RESULT}(s, a), d + 1) \quad \text{if} \quad \text{PLAYER}(s) = \text{MAX} \\
&\min_{a \in \text{ACTIONS}(s)} \text{H-MINIMAX}(\text{RESULT}(s, a, d + 1)) \quad \text{if} \quad \text{PLAYER}(s) = \text{MIN}
\end{aligned}
$$

Cutting off search and evaluation functions

Replace
if TERMINAL-TEST(s) **then return** TERMINAL-UTILITY(s)
with:
if CUTOFF-TEST(s,d) **then return** EVAL(s)

Historical note: cutting search off earlier and use of heuristic evaluation functions proposed by Claude Shannon in *Programming a Computer for Playing Chess* (1950).

EVAL(s) – Evaluation functions

(estimate of) State value for non-terminal states

We need an easy-to-compute function correlated with "chance of winning". For chess:

- $f_1(s)$ Material value for pieces—1 for pawn, 3 for knight/bishop, 5 for rook, 10 for queen. (minus opponent's pieces)
- $f_2(s)$ Finetuning: 2 bishops are worth 6.5; knights are worth more in closed positions...
- Other features worth evaluating: controlling the center of the board, good pawn structure (no double pawns), king safety...
- $f_i(s) = \cdots$ We can create many, how to combine them?

$$\text{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots w_n f_n(s)$$

How to find/compute proper weights?

How to find/create $f_i(s)$?

EVAL(s) – Evaluation functions

(estimate of) State value for non-terminal states

We need an easy-to-compute function correlated with "chance of winning". For chess:

- $f_1(s)$ Material value for pieces—1 for pawn, 3 for knight/bishop, 5 for rook, 10 for queen. (minus opponent's pieces)
- $f_2(s)$ Finetuning: 2 bishops are worth 6.5; knights are worth more in closed positions...
- Other features worth evaluating: controlling the center of the board, good pawn structure (no double pawns), king safety...
- $f_i(s) = \cdots$ We can create many, how to combine them?

$$\textsc{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots w_n f_n(s)$$

How to find/compute proper weights?

How to find/create $f_i(s)$?

EVAL(s) – Evaluation functions

(estimate of) State value for non-terminal states

We need an easy-to-compute function correlated with "chance of winning". For chess:

- $f_1(s)$ Material value for pieces—1 for pawn, 3 for knight/bishop, 5 for rook, 10 for queen. (minus opponent's pieces)
- $f_2(s)$ Finetuning: 2 bishops are worth 6.5; knights are worth more in closed positions...
- Other features worth evaluating: controlling the center of the board, good pawn structure (no double pawns), king safety...
- $f_i(s) = \cdots$ We can create many, how to combine them?

$$\textsc{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots w_n f_n(s)$$

How to find/compute proper weights?

How to find/create $f_i(s)$?

EVAL(s) – Evaluation functions

(estimate of) State value for non-terminal states
We need an easy-to-compute function correlated with "chance of winning". For chess:

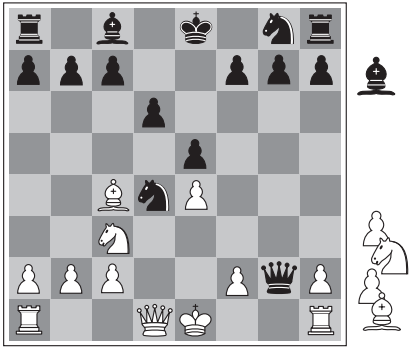
- $f_1(s)$ Material value for pieces—1 for pawn, 3 for knight/bishop, 5 for rook, 10 for queen. (minus opponent's pieces)
- $f_2(s)$ Finetuning: 2 bishops are worth 6.5; knights are worth more in closed positions...
- Other features worth evaluating: controlling the center of the board, good pawn structure (no double pawns), king safety...
- $f_i(s) = \cdots$ We can create many, how to combine them?

$$\textsc{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots w_n f_n(s)$$

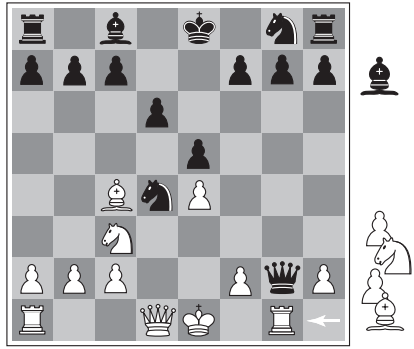
How to find/compute proper weights?
How to find/create $f_i(s)$?

EVAL(s) – Problems

What if something important happens just after the cut – in the next ply?



(a) White to move



(b) White to move

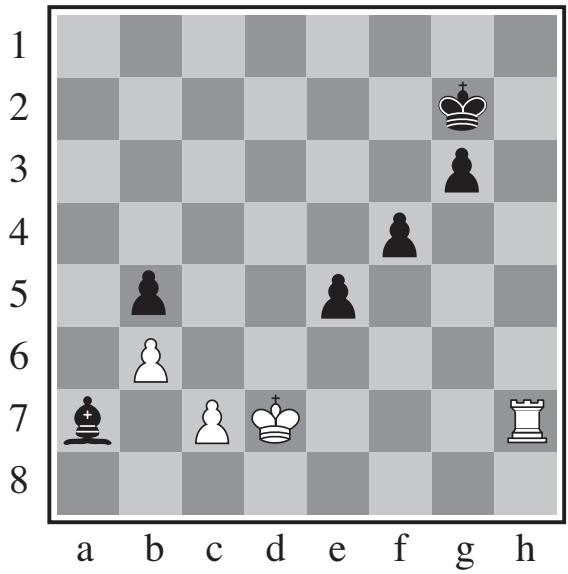
Additional improvements:

- "Killer moves"—capturing opponent's pieces, check etc.—should be considered first.
- *Quiescence search* – EVAL function should be applied only once things calm down. During capturing of pieces, depth should be locally increased.

Horizon effect

Pushing unavoidable loss deeper in tree by a delaying tactics. We know it is useless but does the machine?

See the situation on right. Black is on move, her bishop is surely doomed. However, the inevitable loss can be postponed by moving her pawns and checking the white king. Depending on the searchable depth this may put the loss over the horizon and moving pawns may look promising.



Computer play vs. grandmaster play

- Computers are better since 1997 (Deep Blue defeating Garry Kasparov).
- The way they play is still very different: “dumb”, relying on “brute force”.
- Grandmasters do not excel in being able to compute very deep—many moves ahead.
  - They play based on experience: super-effective pruning and evaluation functions.
  - They consider only 2 to 3 moves in most positions (branching factor).

References

Many images, including the chess plates are from Chapter 5, “Adversarial search” in [1].

[1] Stuart Russell and Peter Norvig.
Artificial Intelligence: A Modern Approach.
Prentice Hall, 3rd edition, 2010.
http://aima.cs.berkeley.edu/.

[2] Richard S. Sutton and Andrew G. Barto.
Reinforcement Learning; an Introduction.
MIT Press, 2nd edition, 2018.
http://www.incompleteideas.net/book/the-book-2nd.html.