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 16, 2021

Notes

Games, man vs. algorithm

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

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

Notes

Please note, the hyperlinks at the main slides are not active in the slides with notes. Hyperlinks within the notes should be active, though.

Games, man vs. algorithm

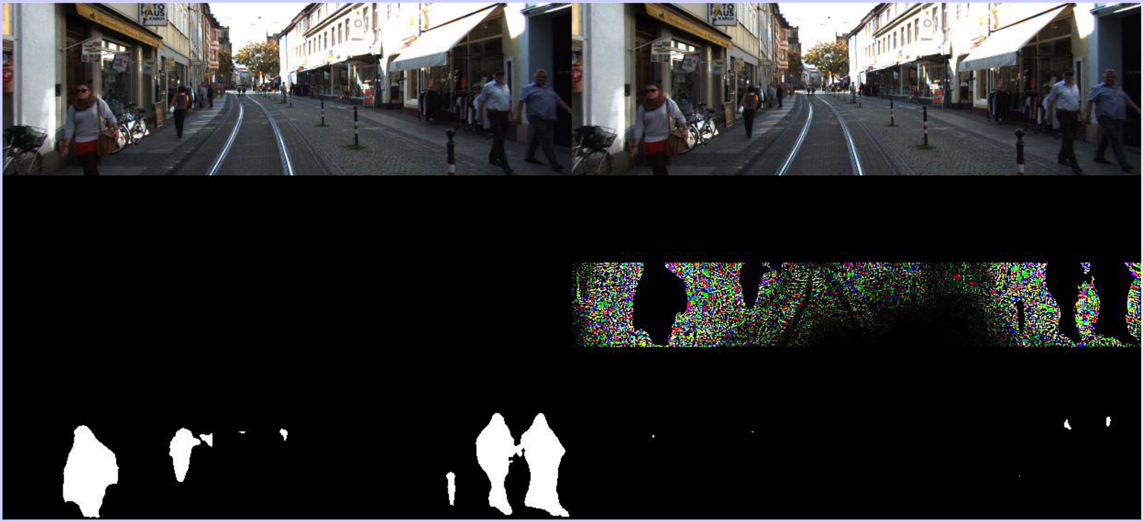
- Deep Blue
- Alpha Go
- Deep Stack
- Why Games, actually?

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

Notes

Please note, the hyperlinks at the main slides are not active in the slides with notes. Hyperlinks within the notes should be active, though.

More: Adversarial Learning



Video: Adversing visual segmentation

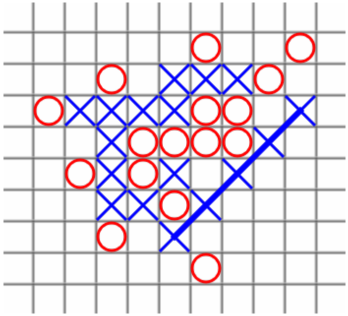
Vision for Robotics and Autonomous Systems, http://cyber.felk.cvut.cz/vras, video at YT: https://youtu.be/KvdZmtVguOo

Notes

- Fooling Tesla autopilot by adversarial attack:

Elements of the game

- s_0: The initial state
- $\textsc{Player}(s)$. Which player has to move in s.
- $\textsc{Actions}(s)$. What are the legal moves?
- $\textsc{Result}(s, a)$. Transition, result of a move.
- $\textsc{Terminal-Test}(s)$. Game over?
- $\textsc{Terminal-Utility}(s, p)$. What is the prize? Examples for some games ...



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

Notes

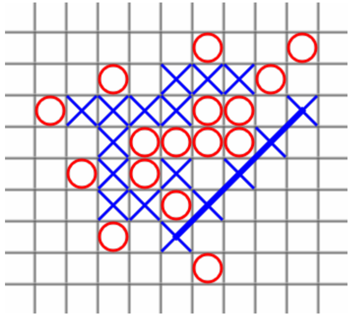
Defining a game as a kind of search problem:
Considering the notation, we are making slight transition from [1] to [2].

- Players: $P = \{1, 2, \dots, N\}$ (often just $N = 2$)
- Transition functions: $S \times A \to S$.
- Terminal utilities: $S \times P \to R$. (R - as a Reward)

What are we loking for? A strategy/policy $S \to A$

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 the prize? Examples for some games ...



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

Notes

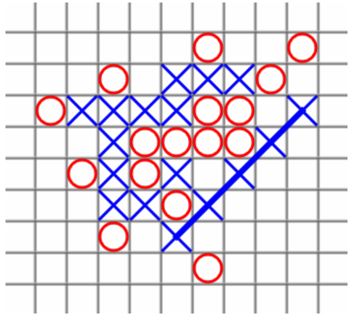
Defining a game as a kind of search problem:
Considering the notation, we are making slight transition from [1] to [2].

- Players: $P = \{1, 2, \ldots, N\}$ (often just $N = 2$)
- Transition functions: $S \times A \rightarrow S$.
- Terminal utilities: $S \times P \rightarrow R$. (R - as a Reward)

What are we loking for? A strategy/policy $S \rightarrow A$

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 the prize? Examples for some games ...



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

Notes

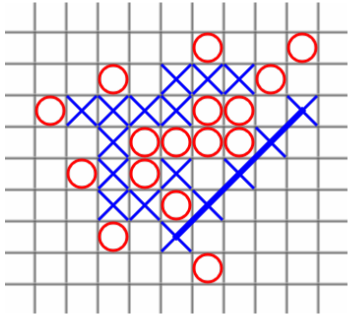
Defining a game as a kind of search problem:
Considering the notation, we are making slight transition from [1] to [2].

- Players: $P = \{1, 2, \dots, N\}$ (often just $N = 2$)
- Transition functions: $S \times A \rightarrow S$.
- Terminal utilities: $S \times P \rightarrow R$. (R - as a Reward)

What are we loking for? A strategy/policy $S \rightarrow A$

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 the prize? Examples for some games ...



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

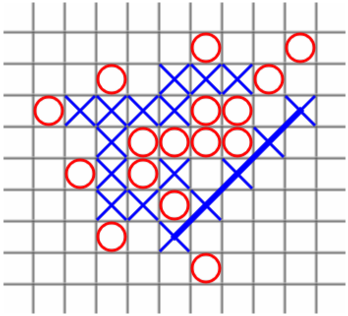
Notes

Defining a game as a kind of search problem:
Considering the notation, we are making slight transition from [1] to [2].

- Players: $P = \{1, 2, \ldots, N\}$ (often just $N = 2$)
- Transition functions: $S \times A \rightarrow S$.
- Terminal utilities: $S \times P \rightarrow R$. (R - as a Reward)

What are we loking for? A strategy/policy $S \rightarrow A$

Elements of the game



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

- 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 the prize? Examples for some games ...

Notes

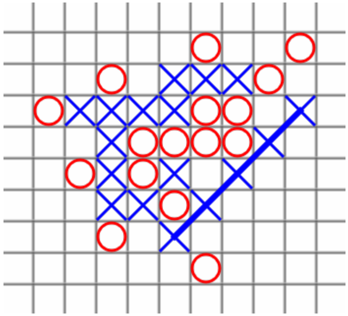
Defining a game as a kind of search problem:
Considering the notation, we are making slight transition from [1] to [2].

- Players: $P = \{1, 2, \ldots, N\}$ (often just $N = 2$)
- Transition functions: $S \times A \to S$.
- Terminal utilities: $S \times P \to R$. (R - as a Reward)

What are we loking for? A strategy/policy $S \to A$

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 the prize? Examples for some games ...



https://commons.wikimedia.org/wiki/File:

Tic-tac-toe_5.png

Notes

Defining a game as a kind of search problem:
Considering the notation, we are making slight transition from [1] to [2].

- Players: $P = \{1, 2, \ldots, N\}$ (often just $N = 2$)
- Transition functions: $S \times A \rightarrow S$.
- Terminal utilities: $S \times P \rightarrow R$. (R - as a Reward)

What are we loking for? A strategy/policy $S \rightarrow A$

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, ...

Notes

Most common games—such as chess—have these properties:

- two-player
- turn-taking
- deterministic with perfect information (a.k.a. deterministic, fully observable environments)

In some games, there is imperfect information (evironment is not fully observable). E.g., poker – no access to what cards opponents hold.

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, ...

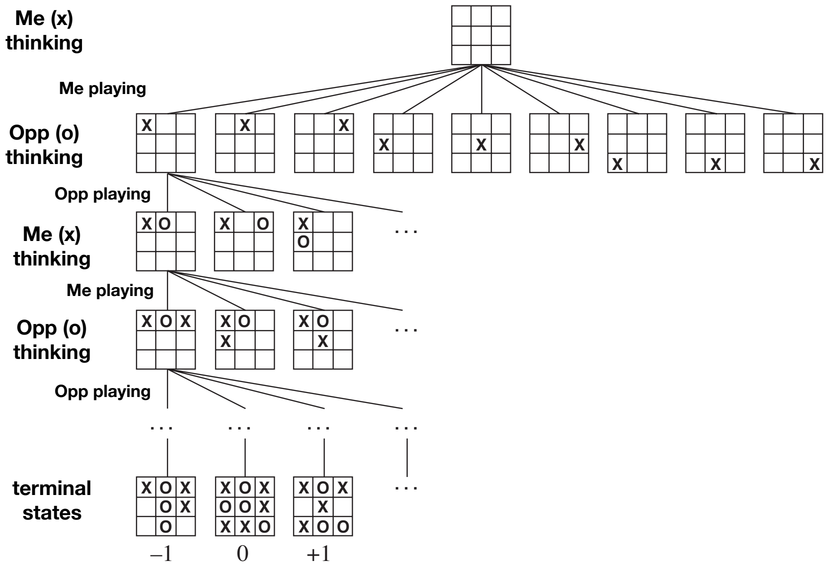
Notes

Most common games—such as chess—have these properties:

- two-player
- turn-taking
- deterministic with perfect information (a.k.a. deterministic, fully observable environments)

In some games, there is imperfect information (evironment is not fully observable). E.g., poker – no access to what cards opponents hold.

Game Tree(s)



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

Notes

Init state, ACTIONS function, and RESULT function defines game tree.

Note: *game tree* as opposed to *search tree*. *Game tree* are all possible evolutions of the game.
(With standard search, we similarly had *state space graph* vs. *search tree*.)

Note: Tic-tac-toe actually is literally zero-sum (at least in our slides, winner: 1, loser: -1, draw: both 0). Unlike chess (sum is 1)... Conceptually, it is the same.

State Value $V(s)$

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

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

Notes

Think about the State Value. It is a theoretical construct, definition. Depending on the problem, there may be various computational algorithms.

In a game, what State Values are known? Usually, only terminal states.

Think, for a moment, you are the only player. You can control every step. How would you compute the $V(s)$ for a given state s?

State Value $V(s)$

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

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

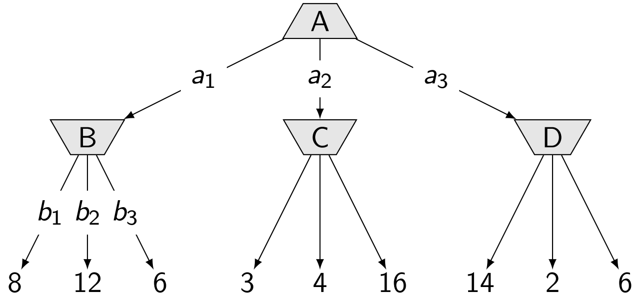
Notes

Think about the State Value. It is a theoretical construct, definition. Depending on the problem, there may be various computational algorithms.

In a game, what State Values are known? Usually, only terminal states.

Think, for a moment, you are the only player. You can control every step. How would you compute the $V(s)$ for a given state 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: $V(A) = 6$

B: $V(A) = 3$

C: $V(A) = 2$

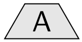
D: $V(A) = 16$

Notes

The correct answer is A: $V(A) = 6$.

Important is that we need to evaluate from the bottom and then go up.

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



$$a_1 = \operatorname*{arg\,max}_{a \in \textsc{Actions}(\text{state}=A)} \textsc{Result}(\text{state} = A, a)$$

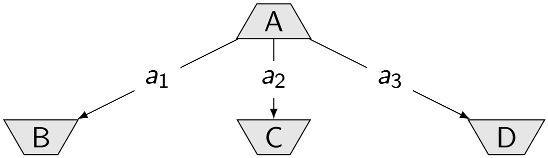
Notes

One *move* consists of two *plies* (half-moves).

I'm the player that starts (state A) and want to decide what to play; actions/plies a_1, a_2, a_3 are the options. B, C, D are the possible outcomes of my moves (plies). Now the opponent is about to play. The numbers in terminal states denote *my* profit/utility.

Node evaluation: *minimax* in action.

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



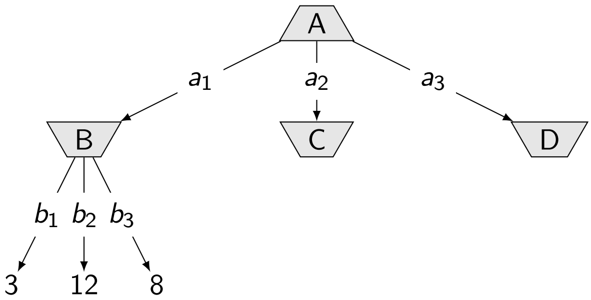
$$a_1 = \operatorname*{arg\,max}_{a \in \textsc{Actions}(\text{state}=A)} \textsc{Result}(\text{state} = A, a)$$

Notes

One *move* consists of two *plies* (half-moves).
I'm the player that starts (state A) and want to decide what to play; actions/plies a_1, a_2, a_3 are the options. B, C, D are the possible outcomes of my moves (plies). Now the opponent is about to play. The numbers in terminal states denote *my* profit/utility.

Node evaluation: *minimax* in action.

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



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

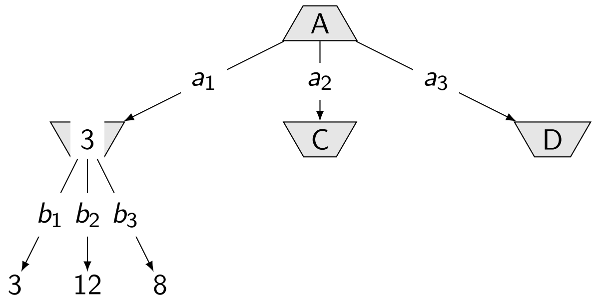
Notes

One *move* consists of two *plies* (half-moves).

I'm the player that starts (state A) and want to decide what to play; actions/plies a_1, a_2, a_3 are the options. B, C, D are the possible outcomes of my moves (plies). Now the opponent is about to play. The numbers in terminal states denote *my* profit/utility.

Node evaluation: *minimax* in action.

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



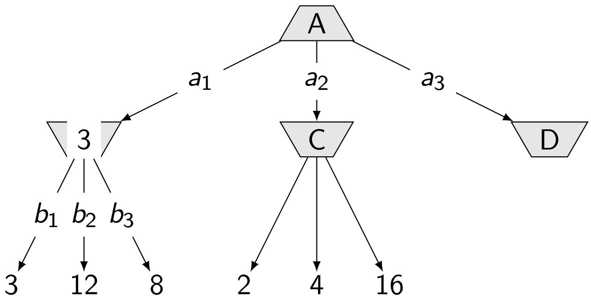
$$a_1 = \underset{a \in \text{ACTIONS}(\text{state}=A)}{\arg\max} \text{RESULT}(\text{state} = A, a)$$

Notes

One *move* consists of two *plies* (half-moves).
I'm the player that starts (state A) and want to decide what to play; actions/plies a_1, a_2, a_3 are the options. B, C, D are the possible outcomes of my moves (plies). Now the opponent is about to play. The numbers in terminal states denote *my* profit/utility.

Node evaluation: *minimax* in action.

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



$$a_1 = \operatorname*{arg\,max}_{a \in \text{ACTIONS}(\text{state}=A)} \text{RESULT}(\text{state} = A, a)$$

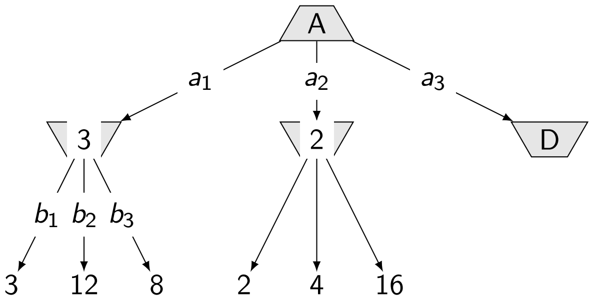
Notes

One *move* consists of two *plies* (half-moves).

I'm the player that starts (state A) and want to decide what to play; actions/plies a_1, a_2, a_3 are the options. B, C, D are the possible outcomes of my moves (plies). Now the opponent is about to play. The numbers in terminal states denote *my* profit/utility.

Node evaluation: *minimax* in action.

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



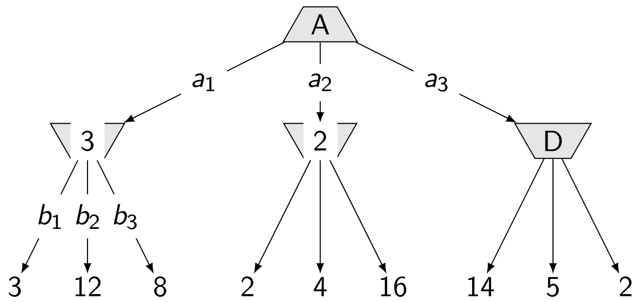
$$a_1 = \arg\max_{a \in \textsc{Actions}(\text{state}=A)} \textsc{Result}(\text{state} = A, a)$$

Notes

One *move* consists of two *plies* (half-moves).
I'm the player that starts (state A) and want to decide what to play; actions/plies a_1, a_2, a_3 are the options. B, C, D are the possible outcomes of my moves (plies). Now the opponent is about to play. The numbers in terminal states denote *my* profit/utility.

Node evaluation: *minimax* in action.

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



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

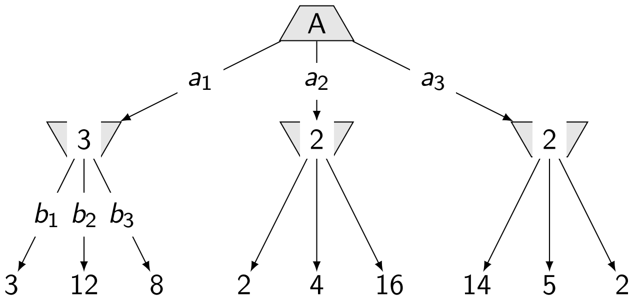
Notes

One *move* consists of two *plies* (half-moves).

I'm the player that starts (state A) and want to decide what to play; actions/plies a_1, a_2, a_3 are the options. B, C, D are the possible outcomes of my moves (plies). Now the opponent is about to play. The numbers in terminal states denote *my* profit/utility.

Node evaluation: *minimax* in action.

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



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

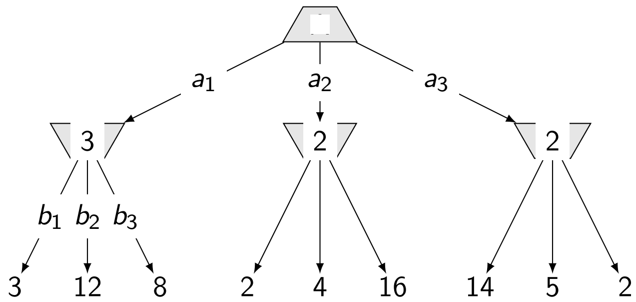
Notes

One *move* consists of two *plies* (half-moves).

I'm the player that starts (state A) and want to decide what to play; actions/plies a_1, a_2, a_3 are the options. B, C, D are the possible outcomes of my moves (plies). Now the opponent is about to play. The numbers in terminal states denote *my* profit/utility.

Node evaluation: *minimax* in action.

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



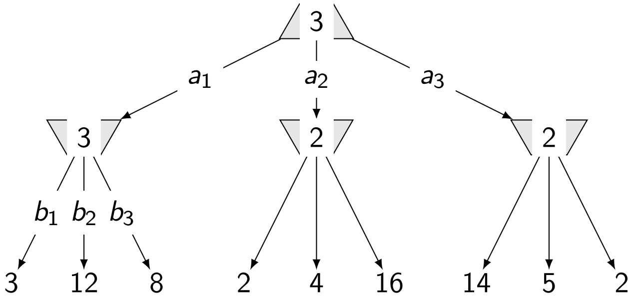
$$a_1 = \underset{a \in \textsc{Actions}(\text{state}=A)}{\arg\max} \textsc{Result}(\text{state} = A, a)$$

Notes

One *move* consists of two *plies* (half-moves).
I'm the player that starts (state A) and want to decide what to play; actions/plies a_1, a_2, a_3 are the options. B, C, D are the possible outcomes of my moves (plies). Now the opponent is about to play. The numbers in terminal states denote *my* profit/utility.

Node evaluation: *minimax* in action.

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



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

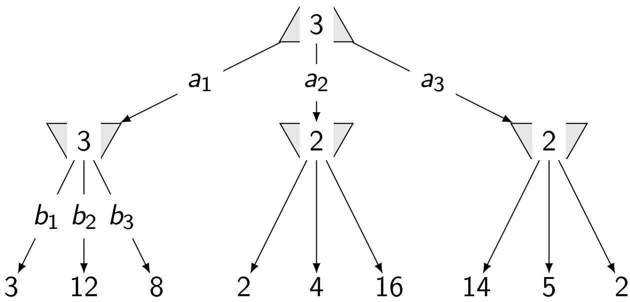
Notes

One *move* consists of two *plies* (half-moves).

I'm the player that starts (state A) and want to decide what to play; actions/plies a_1, a_2, a_3 are the options. B, C, D are the possible outcomes of my moves (plies). Now the opponent is about to play. The numbers in terminal states denote *my* profit/utility.

Node evaluation: *minimax* in action.

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



$$a_1 = \operatorname*{arg\,max}_{a \in \text{ACTIONS}(\text{state}=A)} \text{RESULT}(\text{state} = A, a)$$

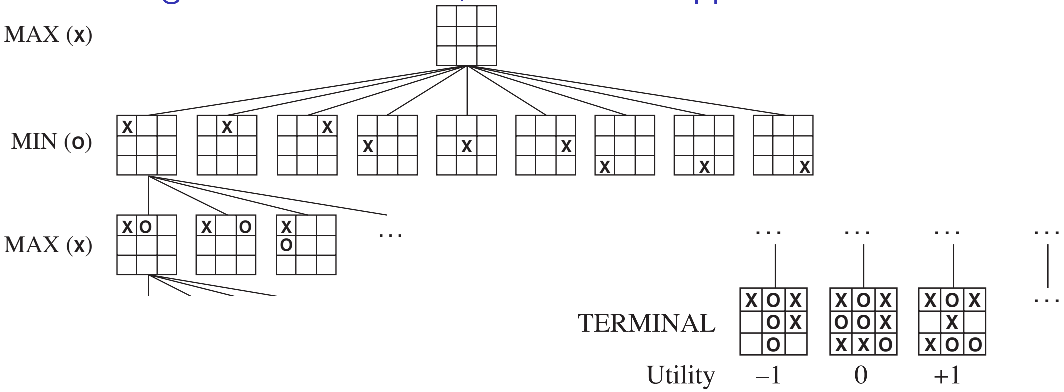
Notes

One *move* consists of two *plies* (half-moves).

I'm the player that starts (state A) and want to decide what to play; actions/plies a_1, a_2, a_3 are the options. B, C, D are the possible outcomes of my moves (plies). Now the opponent is about to play. The numbers in terminal states denote *my* profit/utility.

Node evaluation: *minimax* in action.

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



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

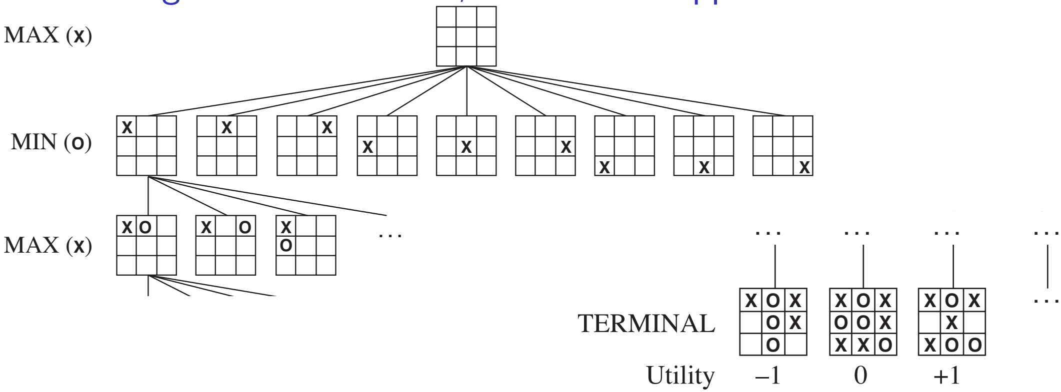
Notes

Max step: I want to maximize my outcome.

Min step: Opponent wants to maximize his outcome which is equivalent to minimizing my outcome.

UTILITY of a state is here the same as VALUE of a state

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

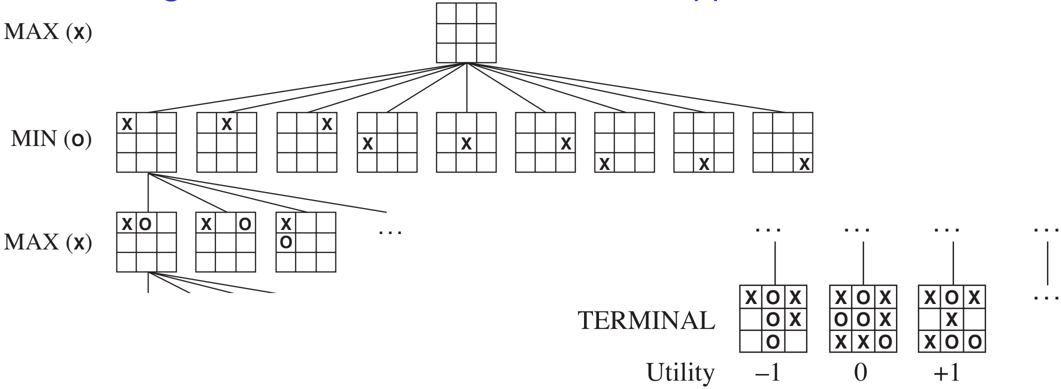


$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if} \quad \text{TERMINAL-TEST}(s) \\ \max_{a \in \text{ACTIONS}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if} \quad \text{PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{ACTIONS}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if} \quad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

Notes

Max step: I want to maximize my outcome.
Min step: Opponent wants to maximize his outcome which is equivalent to minimizing my outcome.
UTILITY of a state is here the same as VALUE of a state

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

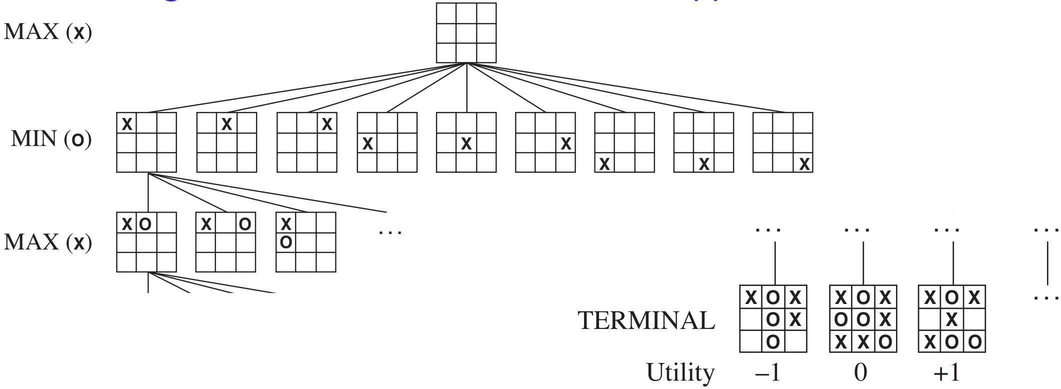


$$\textsc{minimax}(s) = \begin{cases} \textsc{utility}(s) & \text{if } \textsc{terminal-test}(s) \\ \max_{a \in \textsc{actions}(s)} \textsc{minimax}(\textsc{result}(s,a)) & \text{if } \textsc{player}(s) = \textsc{max} \\ \min_{a \in \textsc{actions}(s)} \textsc{minimax}(\textsc{result}(s,a)) & \text{if } \textsc{player}(s) = \textsc{min} \end{cases}$$

Notes

Max step: I want to maximize my outcome.
Min step: Opponent wants to maximize his outcome which is equivalent to minimizing my outcome.
UTILITY of a state is here the same as VALUE of a state

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

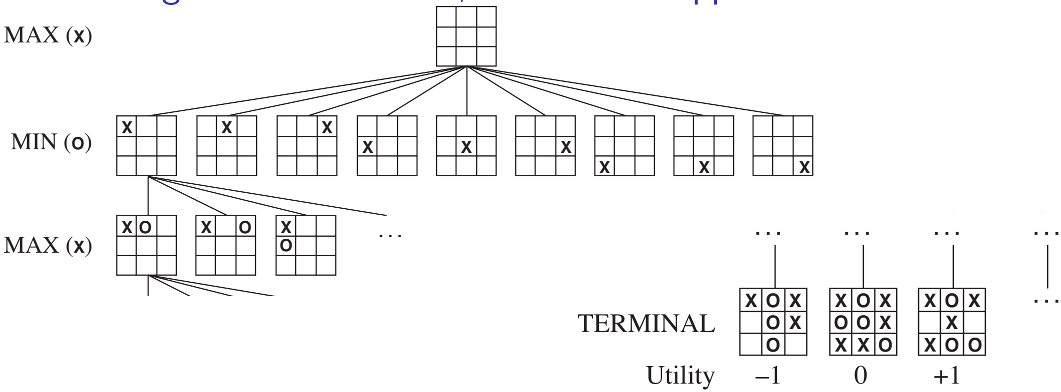


$$
\textsc{minimax}(s) = \begin{cases} \textsc{utility}(s) & \text{if } \textsc{terminal-test}(s) \\ \max_{a \in \textsc{actions}(s)} \textsc{minimax}(\textsc{result}(s,a)) & \text{if } \textsc{player}(s) = \textsc{max} \\ \min_{a \in \textsc{actions}(s)} \textsc{minimax}(\textsc{result}(s,a)) & \text{if } \textsc{player}(s) = \textsc{min} \end{cases}
$$

Notes

Max step: I want to maximize my outcome.
Min step: Opponent wants to maximize his outcome which is equivalent to minimizing my outcome.
UTILITY of a state is here the same as VALUE of a state

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



$$\textsc{minimax}(s) = \begin{cases} \textsc{utility}(s) & \text{if } \textsc{terminal-test}(s) \\ \max_{a \in \textsc{actions}(s)} \textsc{minimax}(\textsc{result}(s,a)) & \text{if } \textsc{player}(s) = \textsc{max} \\ \min_{a \in \textsc{actions}(s)} \textsc{minimax}(\textsc{result}(s,a)) & \text{if } \textsc{player}(s) = \textsc{min} \end{cases}$$

Notes

Max step: I want to maximize my outcome.

Min step: Opponent wants to maximize his outcome which is equivalent to minimizing my outcome.

UTILITY of a state is here the same as VALUE of a state

Minimax algorithm

function MINIMAX(state) **returns** an action
  return $\operatorname{argmax}_{a \in \text{Actions}(s)}$ 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

Notes

Minimax algorithm

function MINIMAX(state) **returns** an action
  return $\operatorname{argmax}_{a \in \text{Actions}(s)}$ 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

Notes

Minimax algorithm

function MINIMAX(state) **returns** an action
    return $\underset{a \in \text{Actions}(s)}{\text{argmax}}$ 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

Notes

Minimax algorithm

```
function MINIMAX(state) returns an action
    return argmax_{a∈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 ← ∞
    for all ACTIONS(state) do
        v ← 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 ← −∞
    for all ACTIONS(state) do
        v ← max(v, MIN-VALUE(RESULT(state,a)))
    end for
end function
```


Notes

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

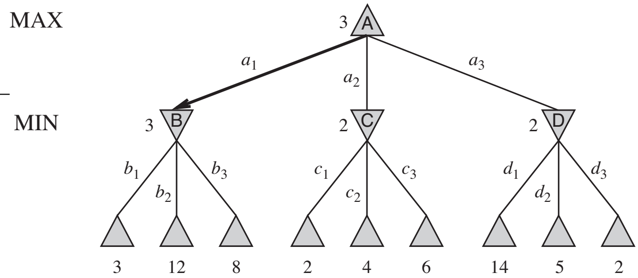
function MINIMAX(s) **returns** a
  $\underset{a \in \text{Actions}(s)}{\text{argmax}}$ MINVAL(RES(s, a))
end function

---

function MINVAL(s) **returns** v
  if TERMINAL(s) **then** UTIL(s)
  end if
  $v \leftarrow \infty$
  for all ACTIONS(s) **do**
    $v \leftarrow$ **min**(v, MAXVAL(RES(s, a)))
  end for
end function

---

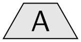
function MAXVAL(s) **returns** v
  if TERMINAL(s) **then** UTIL(s)
  end if
  $v \leftarrow -\infty$
  for all ACTIONS(s) **do**
    $v \leftarrow$ **max**(v, MINVAL(RES(s, a)))
  end for
end function



Notes

Before going to the animation on the next slide, try to follow the algorithm by a pencil and paper.

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?

Notes

Efficiency/complexity:

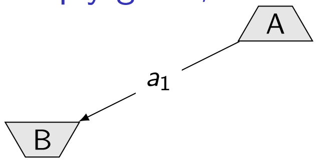
- Exhaustive DFS
- Time $O(b^m)$
- Space $O(bm)$

Chess $b \approx 35, m \approx 100 \ldots$

Note on implementation: Natural implementation of this? Recursion.... Similar to DFS, but there you could circumvent it by using stack for the frontier. Here you have to really dive deep using recursive calls.

- We cannot go(dive) to the end
- Can we save something?

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?

Notes

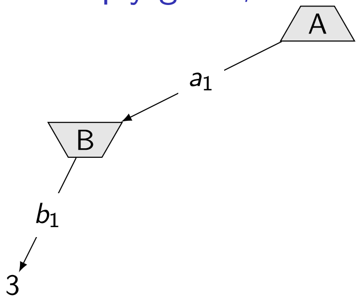
Efficiency/complexity:

- Exhaustive DFS
- Time $O(b^m)$
- Space $O(bm)$

Chess $b \approx 35, m \approx 100$ . . .
Note on implementation: Natural implementation of this? Recursion.... Similar to DFS, but there you could circumvent it by using stack for the frontier. Here you have to really dive deep using recursive calls.

- We cannot go(dive) to the end
- Can we save something?

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?

Notes

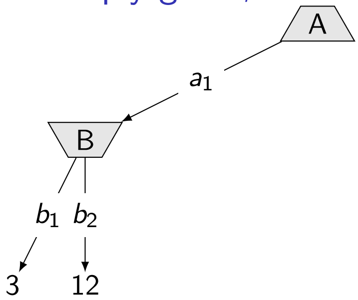
Efficiency/complexity:

- Exhaustive DFS
- Time $O(b^m)$
- Space $O(bm)$

Chess $b \approx 35, m \approx 100$ . . .
Note on implementation: Natural implementation of this? Recursion.... Similar to DFS, but there you could circumvent it by using stack for the frontier. Here you have to really dive deep using recursive calls.

- We cannot go(dive) to the end
- Can we save something?

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?

Notes

Efficiency/complexity:

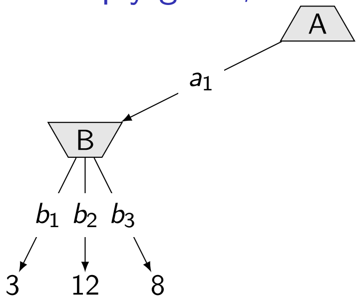
- Exhaustive DFS
- Time $O(b^m)$
- Space $O(bm)$

Chess $b \approx 35, m \approx 100$ ...

Note on implementation: Natural implementation of this? Recursion.... Similar to DFS, but there you could circumvent it by using stack for the frontier. Here you have to really dive deep using recursive calls.

- We cannot go(dive) to the end
- Can we save something?

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?

Notes

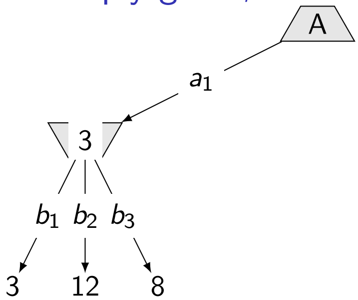
Efficiency/complexity:

- Exhaustive DFS
- Time $O(b^m)$
- Space $O(bm)$

Chess $b \approx 35, m \approx 100 \ldots$
Note on implementation: Natural implementation of this? Recursion.... Similar to DFS, but there you could circumvent it by using stack for the frontier. Here you have to really dive deep using recursive calls.

- We cannot go(dive) to the end
- Can we save something?

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?

Notes

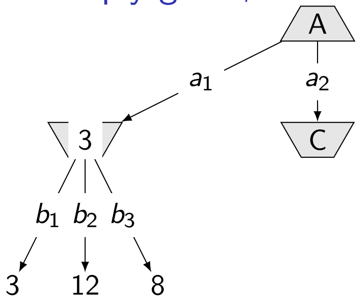
Efficiency/complexity:

- Exhaustive DFS
- Time $O(b^m)$
- Space $O(bm)$

Chess $b \approx 35, m \approx 100 \ldots$
Note on implementation: Natural implementation of this? Recursion.... Similar to DFS, but there you could circumvent it by using stack for the frontier. Here you have to really dive deep using recursive calls.

- We cannot go(dive) to the end
- Can we save something?

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?

Notes

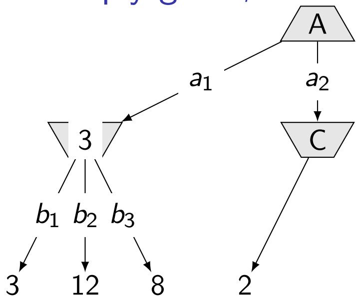
Efficiency/complexity:

- Exhaustive DFS
- Time $O(b^m)$
- Space $O(bm)$

Chess $b \approx 35, m \approx 100 \ldots$
Note on implementation: Natural implementation of this? Recursion.... Similar to DFS, but there you could circumvent it by using stack for the frontier. Here you have to really dive deep using recursive calls.

- We cannot go(dive) to the end
- Can we save something?

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?

Notes

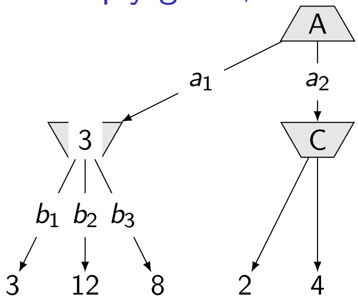
Efficiency/complexity:

- Exhaustive DFS
- Time $O(b^m)$
- Space $O(bm)$

Chess $b \approx 35, m \approx 100 \ldots$
Note on implementation: Natural implementation of this? Recursion.... Similar to DFS, but there you could circumvent it by using stack for the frontier. Here you have to really dive deep using recursive calls.

- We cannot go(dive) to the end
- Can we save something?

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?

Notes

Efficiency/complexity:

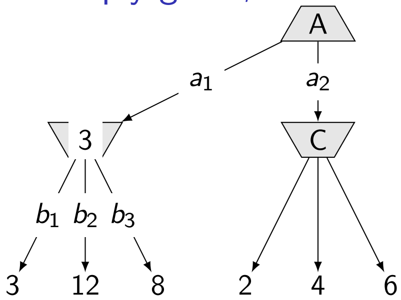
- Exhaustive DFS
- Time $O(b^m)$
- Space $O(bm)$

Chess $b \approx 35, m \approx 100 \ldots$

Note on implementation: Natural implementation of this? Recursion.... Similar to DFS, but there you could circumvent it by using stack for the frontier. Here you have to really dive deep using recursive calls.

- We cannot go(dive) to the end
- Can we save something?

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?

Notes

Efficiency/complexity:

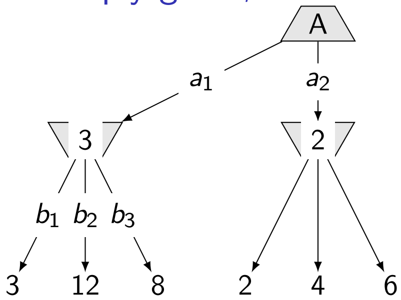
- Exhaustive DFS
- Time $O(b^m)$
- Space $O(bm)$

Chess $b \approx 35, m \approx 100$ ...

Note on implementation: Natural implementation of this? Recursion.... Similar to DFS, but there you could circumvent it by using stack for the frontier. Here you have to really dive deep using recursive calls.

- We cannot go(dive) to the end
- Can we save something?

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?

Notes

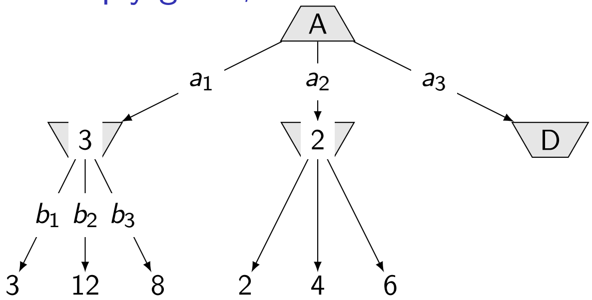
Efficiency/complexity:

- Exhaustive DFS
- Time $O(b^m)$
- Space $O(bm)$

Chess $b \approx 35, m \approx 100$ ...
Note on implementation: Natural implementation of this? Recursion.... Similar to DFS, but there you could circumvent it by using stack for the frontier. Here you have to really dive deep using recursive calls.

- We cannot go(dive) to the end
- Can we save something?

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?

Notes

Efficiency/complexity:

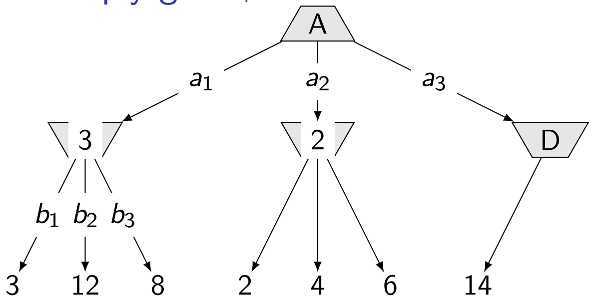
- Exhaustive DFS
- Time $O(b^m)$
- Space $O(bm)$

Chess $b \approx 35, m \approx 100 \ldots$

Note on implementation: Natural implementation of this? Recursion.... Similar to DFS, but there you could circumvent it by using stack for the frontier. Here you have to really dive deep using recursive calls.

- We cannot go(dive) to the end
- Can we save something?

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?

Notes

Efficiency/complexity:

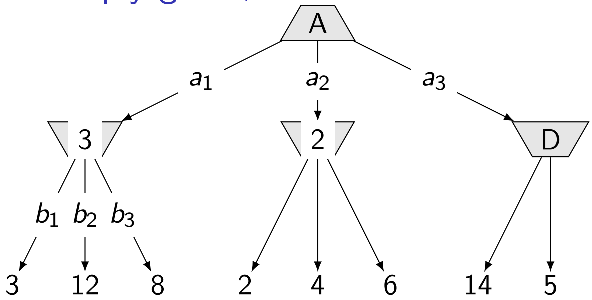
- Exhaustive DFS
- Time $O(b^m)$
- Space $O(bm)$

Chess $b \approx 35, m \approx 100 \ldots$

Note on implementation: Natural implementation of this? Recursion.... Similar to DFS, but there you could circumvent it by using stack for the frontier. Here you have to really dive deep using recursive calls.

- We cannot go(dive) to the end
- Can we save something?

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?

Notes

Efficiency/complexity:

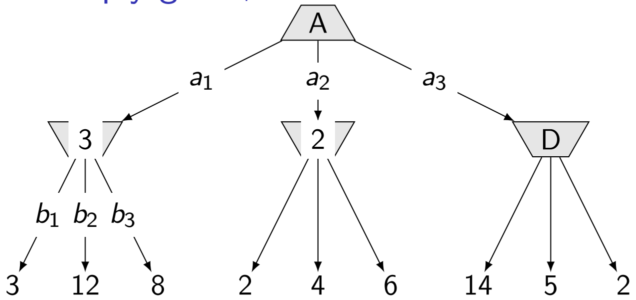
- Exhaustive DFS
- Time $O(b^m)$
- Space $O(bm)$

Chess $b \approx 35, m \approx 100 \ldots$

Note on implementation: Natural implementation of this? Recursion.... Similar to DFS, but there you could circumvent it by using stack for the frontier. Here you have to really dive deep using recursive calls.

- We cannot go(dive) to the end
- Can we save something?

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?

Notes

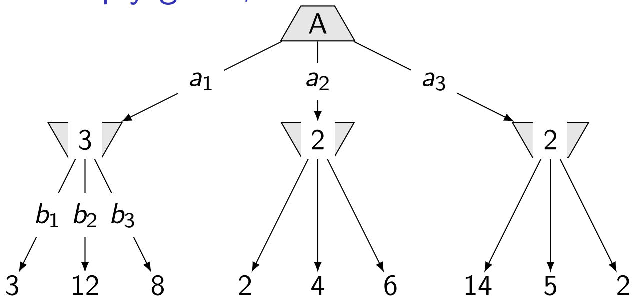
Efficiency/complexity:

- Exhaustive DFS
- Time $O(b^m)$
- Space $O(bm)$

Chess $b \approx 35, m \approx 100 \ldots$
Note on implementation: Natural implementation of this? Recursion.... Similar to DFS, but there you could circumvent it by using stack for the frontier. Here you have to really dive deep using recursive calls.

- We cannot go(dive) to the end
- Can we save something?

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?

Notes

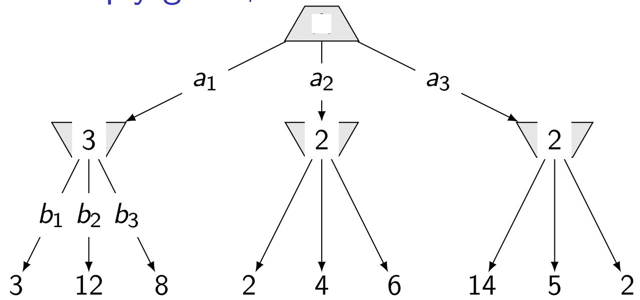
Efficiency/complexity:

- Exhaustive DFS
- Time $O(b^m)$
- Space $O(bm)$

Chess $b \approx 35, m \approx 100 \ldots$
Note on implementation: Natural implementation of this? Recursion.... Similar to DFS, but there you could circumvent it by using stack for the frontier. Here you have to really dive deep using recursive calls.

- We cannot go(dive) to the end
- Can we save something?

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?

Notes

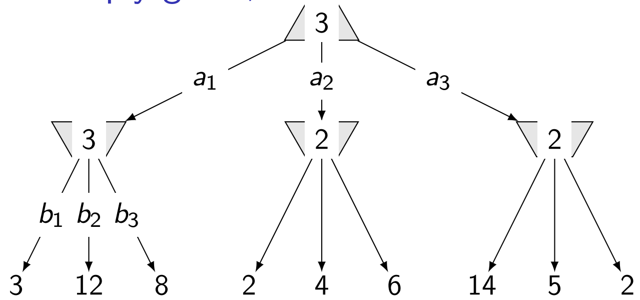
Efficiency/complexity:

- Exhaustive DFS
- Time $O(b^m)$
- Space $O(bm)$

Chess $b \approx 35, m \approx 100$ ...
Note on implementation: Natural implementation of this? Recursion.... Similar to DFS, but there you could circumvent it by using stack for the frontier. Here you have to really dive deep using recursive calls.

- We cannot go(dive) to the end
- Can we save something?

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?

Notes

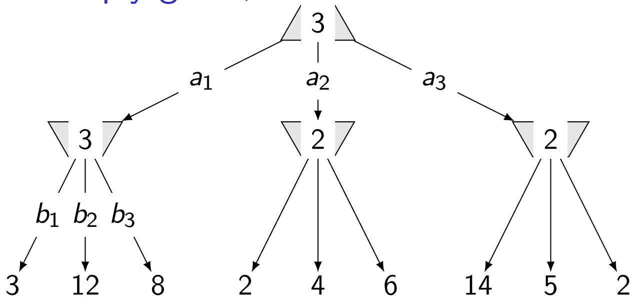
Efficiency/complexity:

- Exhaustive DFS
- Time $O(b^m)$
- Space $O(bm)$

Chess $b \approx 35, m \approx 100 \ldots$
Note on implementation: Natural implementation of this? Recursion.... Similar to DFS, but there you could circumvent it by using stack for the frontier. Here you have to really dive deep using recursive calls.

- We cannot go(dive) to the end
- Can we save something?

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?

Notes

Efficiency/complexity:

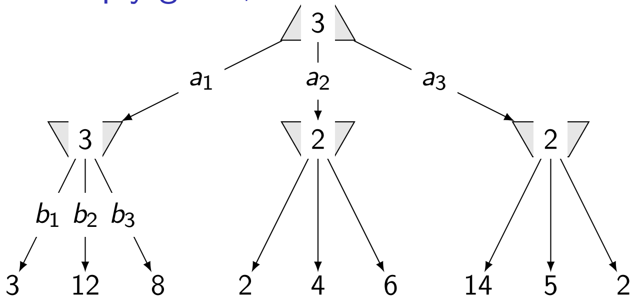
- Exhaustive DFS
- Time $O(b^m)$
- Space $O(bm)$

Chess $b \approx 35, m \approx 100 \ldots$

Note on implementation: Natural implementation of this? Recursion.... Similar to DFS, but there you could circumvent it by using stack for the frontier. Here you have to really dive deep using recursive calls.

- We cannot go(dive) to the end
- Can we save something?

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?

Notes

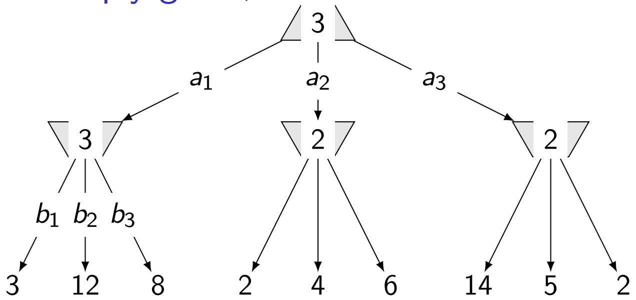
Efficiency/complexity:

- Exhaustive DFS
- Time $O(b^m)$
- Space $O(bm)$

Chess $b \approx 35, m \approx 100$ ...
Note on implementation: Natural implementation of this? Recursion.... Similar to DFS, but there you could circumvent it by using stack for the frontier. Here you have to really dive deep using recursive calls.

- We cannot go(dive) to the end
- Can we save something?

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?

Notes

Efficiency/complexity:

- Exhaustive DFS
- Time $O(b^m)$
- Space $O(bm)$

Chess $b \approx 35, m \approx 100 \ldots$
Note on implementation: Natural implementation of this? Recursion.... Similar to DFS, but there you could circumvent it by using stack for the frontier. Here you have to really dive deep using recursive calls.

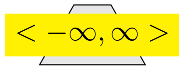
- We cannot go(dive) to the end
- Can we save something?

Nodes (sub-trees) worth visiting

Notes

Constraining the possible node values as search progresses...

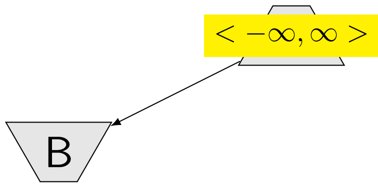
Nodes (sub-trees) worth visiting



Notes

Constraining the possible node values as search progresses...

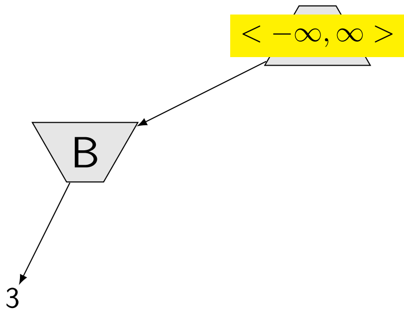
Nodes (sub-trees) worth visiting



Notes

Constraining the possible node values as search progresses...

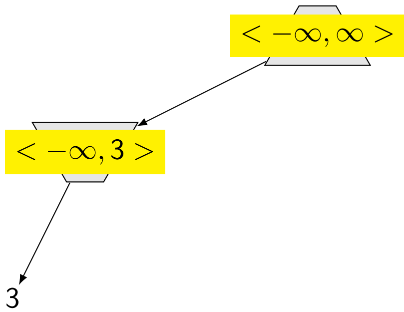
Nodes (sub-trees) worth visiting



Notes

Constraining the possible node values as search progresses...

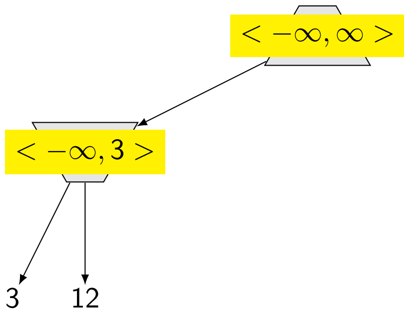
Nodes (sub-trees) worth visiting



Notes

Constraining the possible node values as search progresses...

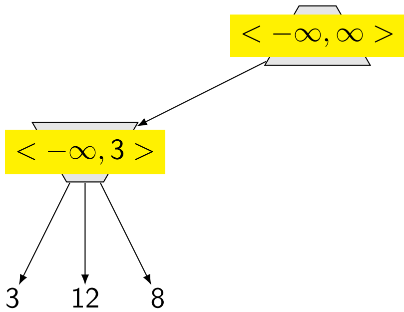
Nodes (sub-trees) worth visiting



Notes

Constraining the possible node values as search progresses...

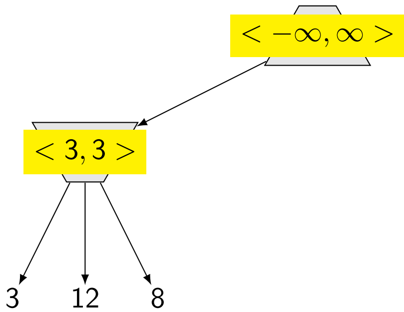
Nodes (sub-trees) worth visiting



Notes

Constraining the possible node values as search progresses...

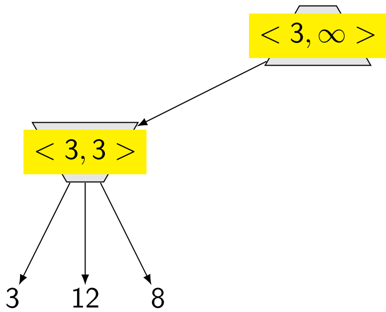
Nodes (sub-trees) worth visiting



Notes

Constraining the possible node values as search progresses...

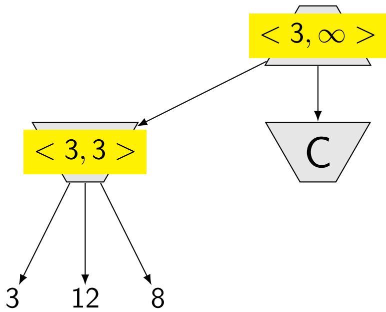
Nodes (sub-trees) worth visiting



Notes

Constraining the possible node values as search progresses...

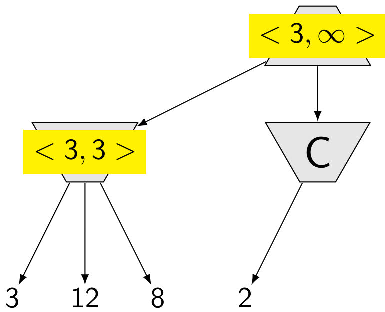
Nodes (sub-trees) worth visiting



Notes

Constraining the possible node values as search progresses...

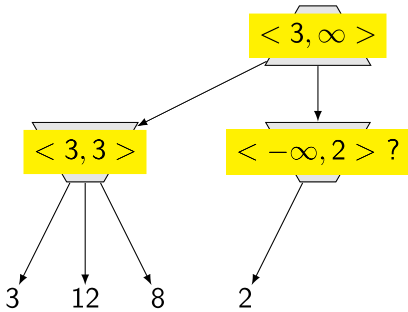
Nodes (sub-trees) worth visiting



Notes

Constraining the possible node values as search progresses...

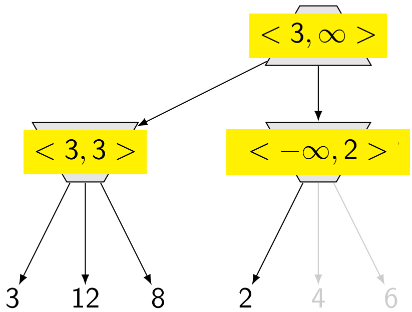
Nodes (sub-trees) worth visiting



Notes

Constraining the possible node values as search progresses...

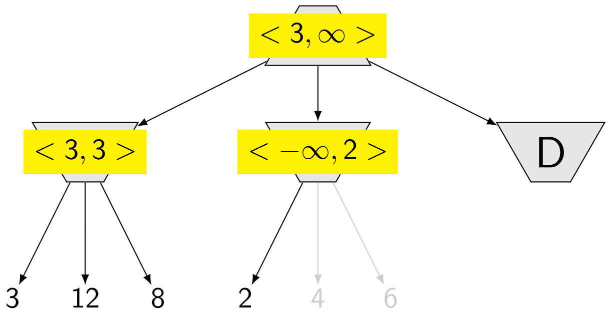
Nodes (sub-trees) worth visiting



Notes

Constraining the possible node values as search progresses...

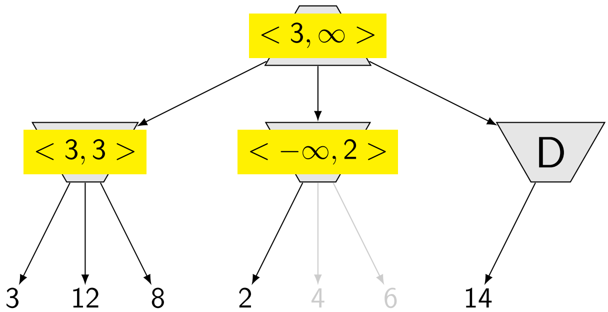
Nodes (sub-trees) worth visiting



Notes

Constraining the possible node values as search progresses...

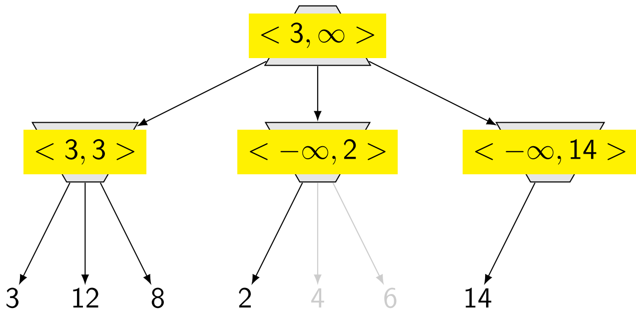
Nodes (sub-trees) worth visiting



Notes

Constraining the possible node values as search progresses...

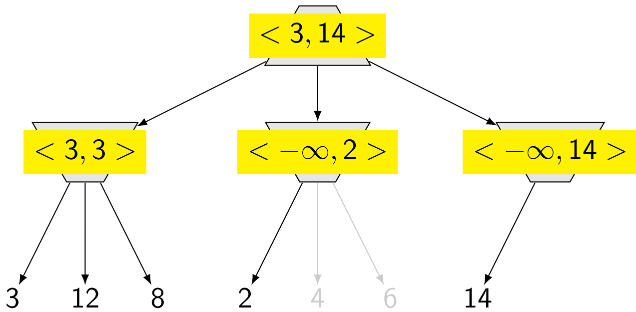
Nodes (sub-trees) worth visiting



Notes

Constraining the possible node values as search progresses...

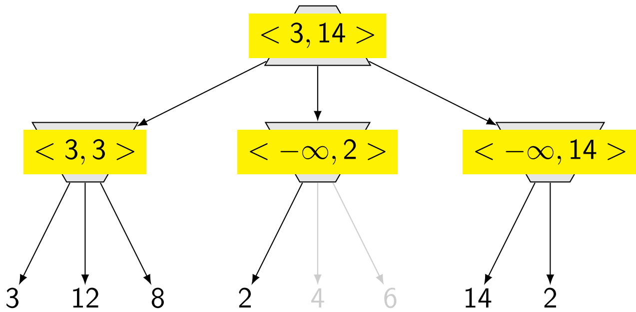
Nodes (sub-trees) worth visiting



Notes

Constraining the possible node values as search progresses...

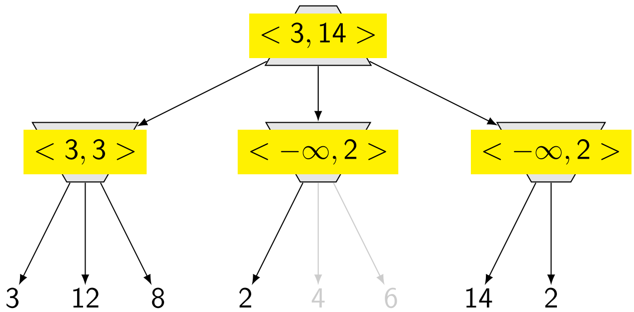
Nodes (sub-trees) worth visiting



Notes

Constraining the possible node values as search progresses...

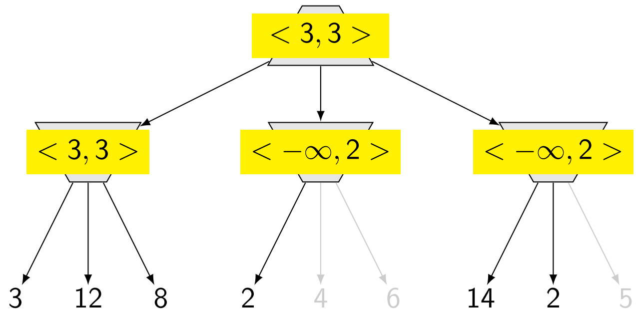
Nodes (sub-trees) worth visiting



Notes

Constraining the possible node values as search progresses...

Nodes (sub-trees) worth visiting

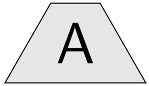


Notes

Constraining the possible node values as search progresses...

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

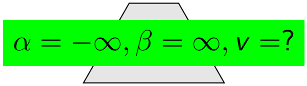
In MIN-VAL: $v \leftarrow 2$
$v \leq \alpha$ then: return v!

Notes

Functions scope: MAX-VALUE MIN-VALUE. The terminal nodes are served/answered within the MAX-VALUE function.
Once a node (subtree) is exhausted (fully expanded), values are propagated towards the root.
In MAX nodes α is changing and β is stopping, in MIN nodes β is changing and α is stopping.

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

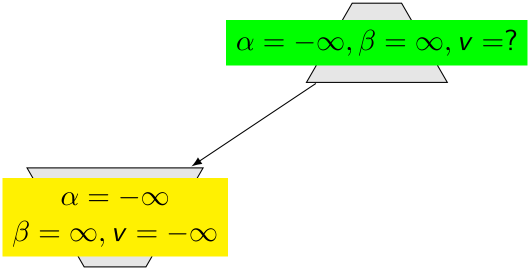
In MIN-VAL: $v \leftarrow 2$
$v \leq \alpha$ then: return v!

Notes

Functions scope: MAX-VALUE MIN-VALUE. The terminal nodes are served/answered within the MAX-VALUE function.
Once a node (subtree) is exhausted (fully expanded), values are propagated towards the root.
In MAX nodes α is changing and β is stopping, in MIN nodes β is changing and α is stopping.

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

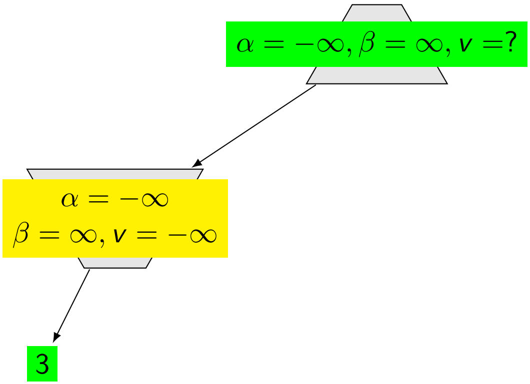
In MIN-VAL: $v \leftarrow 2$
$v \leq \alpha$ then: return v!

Notes

Functions scope: MAX-VALUE MIN-VALUE. The terminal nodes are served/answered within the MAX-VALUE function.
Once a node (subtree) is exhausted (fully expanded), values are propagated towards the root.
In MAX nodes α is changing and β is stopping, in MIN nodes β is changing and α is stopping.

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!

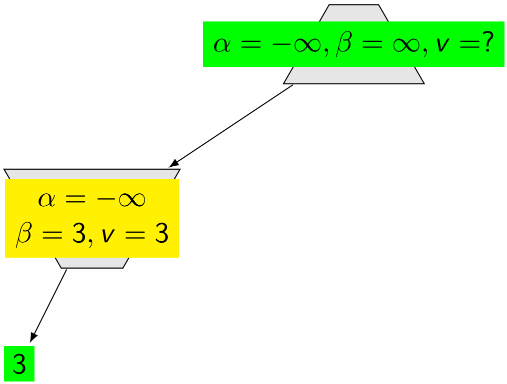
Notes

Functions scope: MAX-VALUE MIN-VALUE. The terminal nodes are served/answered within the MAX-VALUE function.
Once a node (subtree) is exhausted (fully expanded), values are propagated towards the root.
In MAX nodes α is changing and β is stopping, in MIN nodes β is changing and α is stopping.

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!

Notes

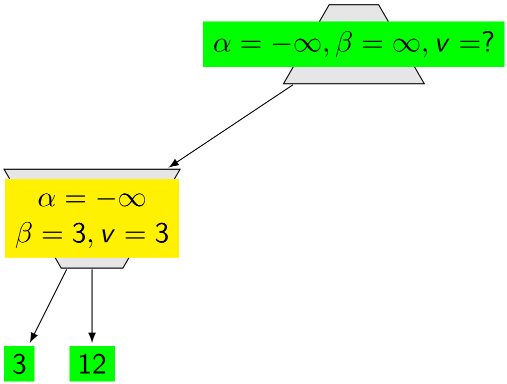
Functions scope: MAX-VALUE MIN-VALUE. The terminal nodes are served/answered within the MAX-VALUE function.

Once a node (subtree) is exhausted (fully expanded), values are propagated towards the root.

In MAX nodes α is changing and β is stopping, in MIN nodes β is changing and α is stopping.

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!

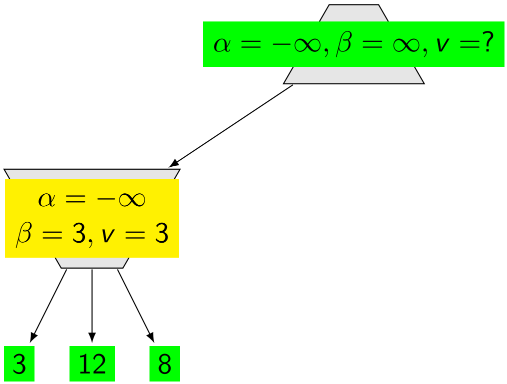
Notes

Functions scope: MAX-VALUE MIN-VALUE. The terminal nodes are served/answered within the MAX-VALUE function.
Once a node (subtree) is exhausted (fully expanded), values are propagated towards the root.
In MAX nodes α is changing and β is stopping, in MIN nodes β is changing and α is stopping.

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!

Notes

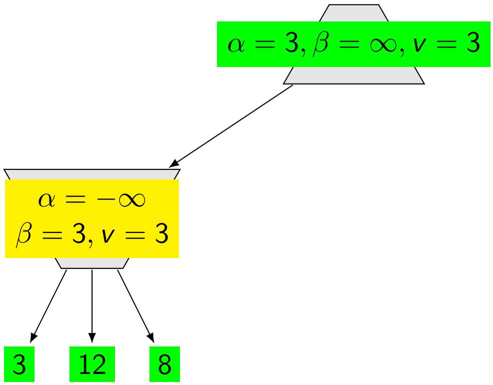
Functions scope: MAX-VALUE MIN-VALUE. The terminal nodes are served/answered within the MAX-VALUE function.

Once a node (subtree) is exhausted (fully expanded), values are propagated towards the root.

In MAX nodes α is changing and β is stopping, in MIN nodes β is changing and α is stopping.

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

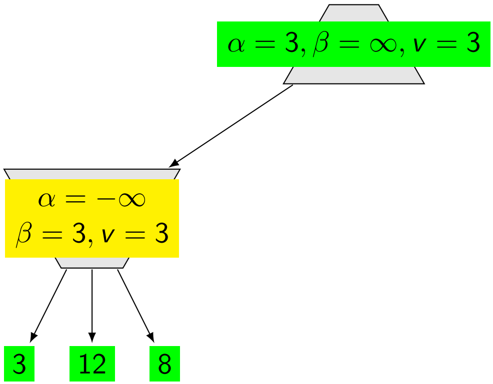
In MIN-VAL: $v \leftarrow 2$
$v \leq \alpha$ then: return v!

Notes

Functions scope: MAX-VALUE MIN-VALUE. The terminal nodes are served/answered within the MAX-VALUE function.
Once a node (subtree) is exhausted (fully expanded), values are propagated towards the root.
In MAX nodes α is changing and β is stopping, in MIN nodes β is changing and α is stopping.

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

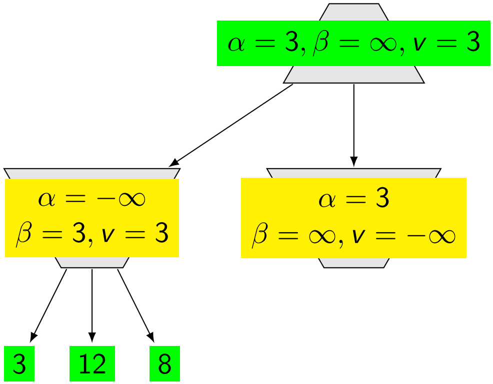
In MIN-VAL: $v \leftarrow 2$
$v \leq \alpha$ then: return v!

Notes

Functions scope: MAX-VALUE MIN-VALUE. The terminal nodes are served/answered within the MAX-VALUE function.
Once a node (subtree) is exhausted (fully expanded), values are propagated towards the root.
In MAX nodes α is changing and β is stopping, in MIN nodes β is changing and α is stopping.

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

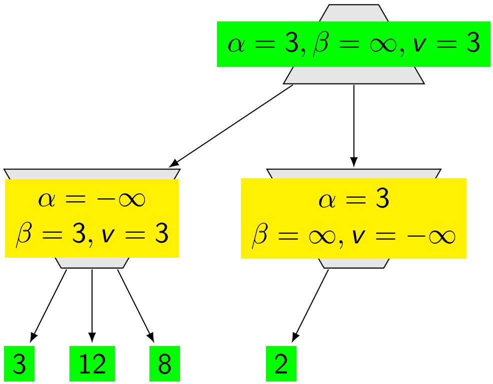
In MIN-VAL: $v \leftarrow 2$
$v \leq \alpha$ then: return v!

Notes

Functions scope: MAX-VALUE MIN-VALUE. The terminal nodes are served/answered within the MAX-VALUE function.
Once a node (subtree) is exhausted (fully expanded), values are propagated towards the root.
In MAX nodes α is changing and β is stopping, in MIN nodes β is changing and α is stopping.

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

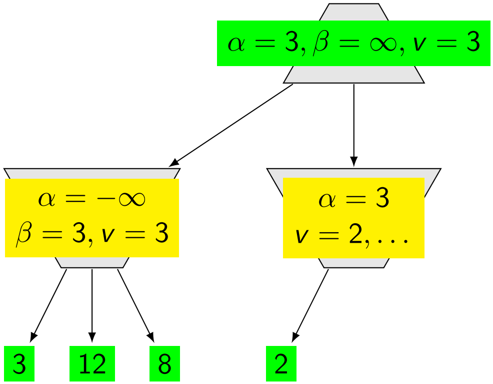
In MIN-VAL: $v \leftarrow 2$
$v \leq \alpha$ then: return v!

Notes

Functions scope: MAX-VALUE MIN-VALUE. The terminal nodes are served/answered within the MAX-VALUE function.
Once a node (subtree) is exhausted (fully expanded), values are propagated towards the root.
In MAX nodes α is changing and β is stopping, in MIN nodes β is changing and α is stopping.

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: $v \leftarrow 2$

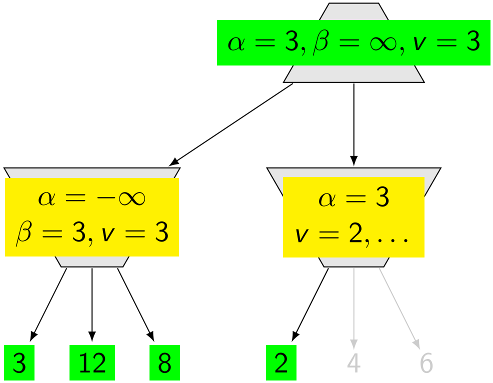
$v \leq \alpha$ then: return v!

Notes

Functions scope: MAX-VALUE MIN-VALUE. The terminal nodes are served/answered within the MAX-VALUE function.
Once a node (subtree) is exhausted (fully expanded), values are propagated towards the root.
In MAX nodes α is changing and β is stopping, in MIN nodes β is changing and α is stopping.

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

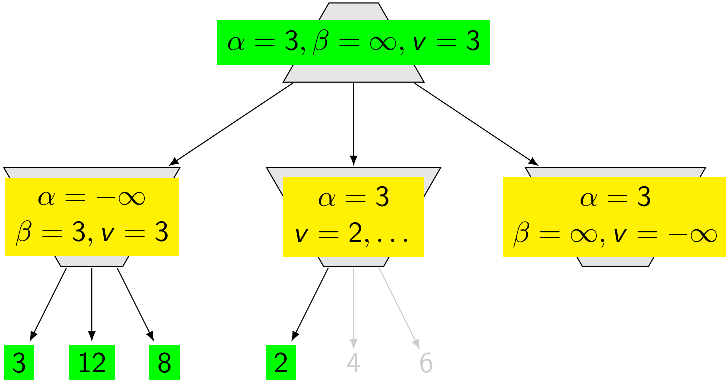
In MIN-VAL: $v \leftarrow 2$
$v \leq \alpha$ then: return v!

Notes

Functions scope: MAX-VALUE MIN-VALUE. The terminal nodes are served/answered within the MAX-VALUE function.
Once a node (subtree) is exhausted (fully expanded), values are propagated towards the root.
In MAX nodes α is changing and β is stopping, in MIN nodes β is changing and α is stopping.

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

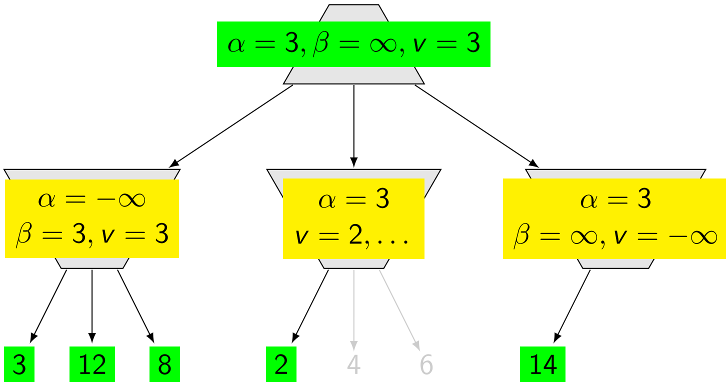
In MIN-VAL: $v \leftarrow 2$
$v \leq \alpha$ then: return v!

Notes

Functions scope: MAX-VALUE MIN-VALUE. The terminal nodes are served/answered within the MAX-VALUE function.
Once a node (subtree) is exhausted (fully expanded), values are propagated towards the root.
In MAX nodes α is changing and β is stopping, in MIN nodes β is changing and α is stopping.

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

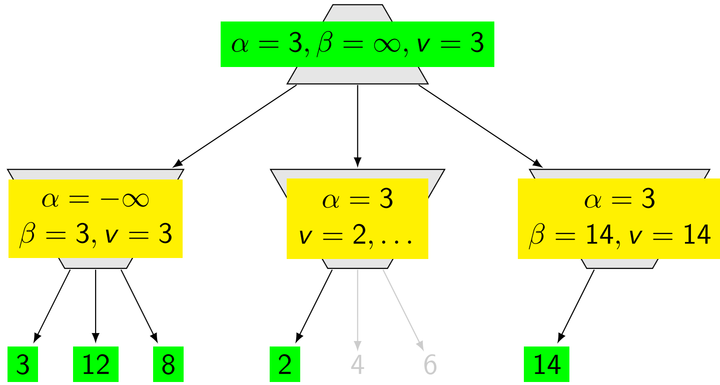
In MIN-VAL: $v \leftarrow 2$
$v \leq \alpha$ then: return v!

Notes

Functions scope: MAX-VALUE MIN-VALUE. The terminal nodes are served/answered within the MAX-VALUE function.
Once a node (subtree) is exhausted (fully expanded), values are propagated towards the root.
In MAX nodes α is changing and β is stopping, in MIN nodes β is changing and α is stopping.

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

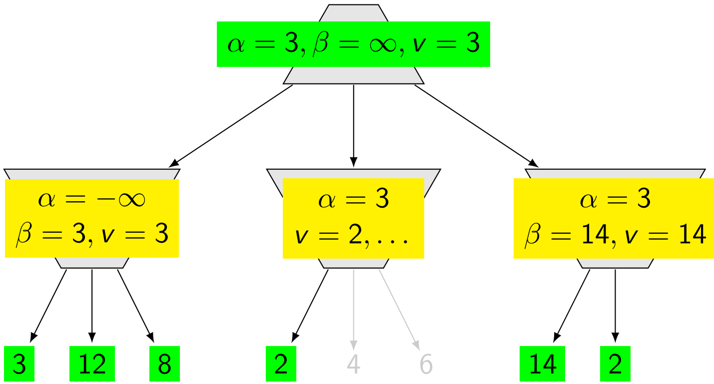
In MIN-VAL: $v \leftarrow 2$
$v \leq \alpha$ then: return v!

Notes

Functions scope: MAX-VALUE MIN-VALUE. The terminal nodes are served/answered within the MAX-VALUE function.
Once a node (subtree) is exhausted (fully expanded), values are propagated towards the root.
In MAX nodes α is changing and β is stopping, in MIN nodes β is changing and α is stopping.

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!

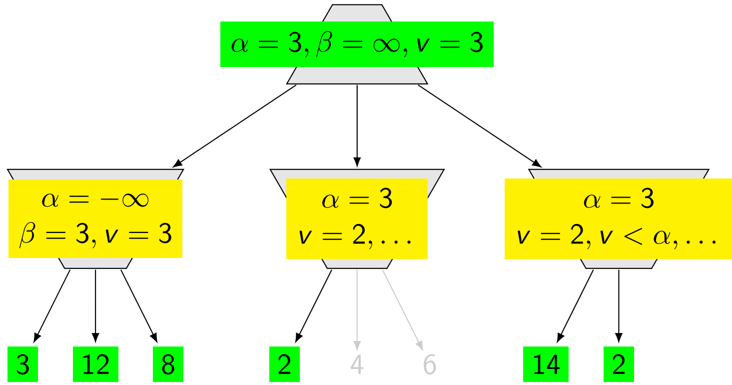
Notes

Functions scope: MAX-VALUE MIN-VALUE. The terminal nodes are served/answered within the MAX-VALUE function.
Once a node (subtree) is exhausted (fully expanded), values are propagated towards the root.
In MAX nodes α is changing and β is stopping, in MIN nodes β is changing and α is stopping.

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: $v \leftarrow 2$

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

Notes

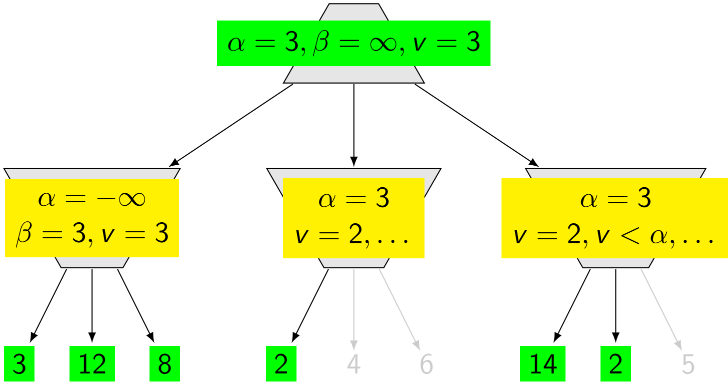
Functions scope: MAX-VALUE MIN-VALUE. The terminal nodes are served/answered within the MAX-VALUE function.

Once a node (subtree) is exhausted (fully expanded), values are propagated towards the root.

In MAX nodes α is changing and β is stopping, in MIN nodes β is changing and α is stopping.

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!

Notes

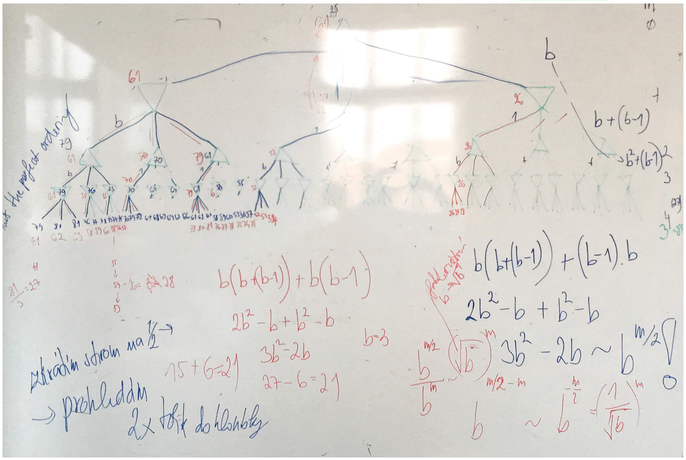
Functions scope: MAX-VALUE MIN-VALUE. The terminal nodes are served/answered within the MAX-VALUE function.
Once a node (subtree) is exhausted (fully expanded), values are propagated towards the root.
In MAX nodes α is changing and β is stopping, in MIN nodes β is changing and α is stopping.

α-β prunnig – How much can we save?

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

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

Notes

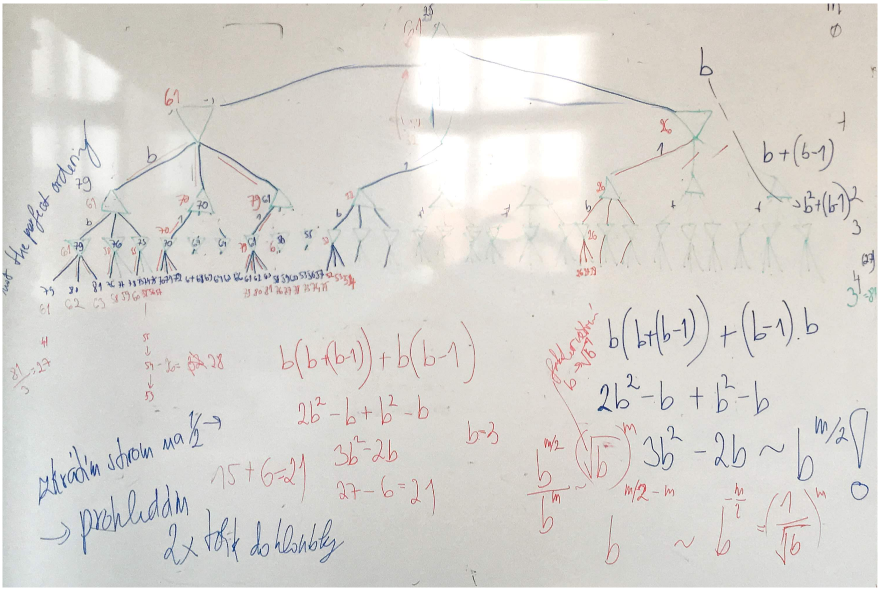


It is clear that ordering of child nodes matters. It is depth-first search. Picking useless action first may be a huge waste of time—a complete subtree beneath the current node will be explored.

Draw a tree of α-β search in case of perferct ordering. Effective branching factor becomes $\sqrt{b}$ instead of b which effectively doubles the depth that can be searched: Time: $O(b^{m/2})$

α-β saving, sketch . . .

Notes



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
end function

Notes

Take the tree from the previous slide and try to go step-by-step, watch α, β and v

```
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
end function
```


Notes

Take the tree from the previous slide and try to go step-by-step, watch α, β and v

```
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
end function
```


Notes

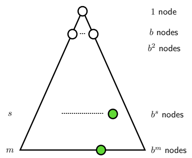
Take the tree from the previous slide and try to go step-by-step, watch α, β and v

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.

Notes

α-β pruning is good. Still, in chess, for example, there is no way we can compute till the end.
Time is limited. We need to respond within a certain amount of time.
Possible solution: iterative deepening search. If I can’t complete the computation for the current depth, I can use the previous shallower one that finished (also called *anytime algorithm*).

Imperfect but real-time decisions: iterative deepening

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

Notes

Even with perfect ordering, α-β pruning is $O(b^{m/2})$. It doubles the depth we can search. Often, we still cannot go the very bottom of the search tree.

One problem left: can't compute till the end and need to cut off. Need for **Evaluation function**.

Imperfect but real-time decisions: iterative deepening

$$\textsc{h-minimax}(s,d) =$$

$$\begin{cases} \textsc{eval}(s) & \text{if } \textsc{cutoff-test}(s,d) \\ \max_{a \in \textsc{actions}(s)} \textsc{h-minimax}(\textsc{result}(s,a), d+1) & \text{if } \textsc{player}(s) = \textsc{max} \\ \min_{a \in \textsc{actions}(s)} \textsc{h-minimax}(\textsc{result}(s,a,d+1)) & \text{if } \textsc{player}(s) = \textsc{min} \end{cases}$$

Notes

Even with perfect ordering, α-β pruning is $O(b^{m/2})$. It doubles the depth we can search. Often, we still cannot go the very bottom of the search tree.

One problem left: can't compute till the end and need to cut off. Need for **Evaluation function**.

Imperfect but real-time decisions: iterative deepening

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

Notes

Even with perfect ordering, α-β pruning is $O(b^{m/2})$. It doubles the depth we can search. Often, we still cannot go the very bottom of the search tree.

One problem left: can't compute till the end and need to cut off. Need for **Evaluation function**.

Imperfect but real-time decisions: iterative deepening

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

Notes

Even with perfect ordering, α-β pruning is $O(b^{m/2})$. It doubles the depth we can search. Often, we still cannot go the very bottom of the search tree.

One problem left: can't compute till the end and need to cut off. Need for **Evaluation function**.

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).

Notes

Cutting depends on *d* only, why we need *s* as the input parameter?

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 $f_i(s)$?

Notes

For many problems it is not so easy to find/construct a proper function. We may try more functions and combine them conveniently.

$$f_1(s) = \text{number of white pawns} - \text{number of black pawns}$$

Weighted sum:

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

How to tune weights w_i?

- Look (read) into (abundant) chess literature.
- Ask experts.
- Machine analysis of historical records - machine learning .
- We will talk about learning linear classifiers, weights, later in this course.
- New: have the computer play against itself and learn everything himself. See *AlphaZero* (2017) - learned to play chess, Go, and shogi like this, achieving superhuman level of play within 24 hours.

If we do not know the individual functions, is there a way for creating them? Deep Convolution Nets! Yeah!

How to get training data for supervised learning? More later.

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 $f_i(s)$?

Notes

For many problems it is not so easy to find/construct a proper function. We may try more functions and combine them conveniently.

$$f_1(s) = \text{number of white pawns} - \text{number of black pawns}$$

Weighted sum:

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

How to tune weights w_i?

- Look (read) into (abundant) chess literature.
- Ask experts.
- Machine analysis of historical records - machine learning .
- We will talk about learning linear classifiers, weights, later in this course.
- New: have the computer play against itself and learn everything himself. See *AlphaZero* (2017) - learned to play chess, Go, and shogi like this, achieving superhuman level of play within 24 hours.

If we do not know the individual functions, is there a way for creating them? Deep Convolution Nets! Yeah!

How to get training data for supervised learning? More later.

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 $f_i(s)$?

Notes

For many problems it is not so easy to find/construct a proper function. We may try more functions and combine them conveniently.

$$f_1(s) = \text{number of white pawns} - \text{number of black pawns}$$

Weighted sum:

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

How to tune weights w_i?

- Look (read) into (abundant) chess literature.
- Ask experts.
- Machine analysis of historical records - machine learning .
- We will talk about learning linear classifiers, weights, later in this course.
- New: have the computer play against itself and learn everything himself. See *AlphaZero* (2017) - learned to play chess, Go, and shogi like this, achieving superhuman level of play within 24 hours.

If we do not know the individual functions, is there a way for creating them? Deep Convolution Nets! Yeah!

How to get training data for supervised learning? More later.

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 $f_i(s)$?

Notes

For many problems it is not so easy to find/construct a proper function. We may try more functions and combine them conveniently.

$$f_1(s) = \text{number of white pawns} - \text{number of black pawns}$$

Weighted sum:

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

How to tune weights w_i?

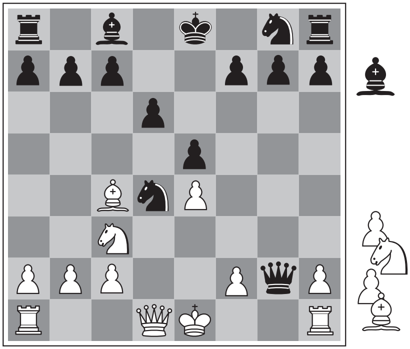
- Look (read) into (abundant) chess literature.
- Ask experts.
- Machine analysis of historical records - machine learning .
- We will talk about learning linear classifiers, weights, later in this course.
- New: have the computer play against itself and learn everything himself. See *AlphaZero* (2017) - learned to play chess, Go, and shogi like this, achieving superhuman level of play within 24 hours.

If we do not know the individual functions, is there a way for creating them? Deep Convolution Nets! Yeah!

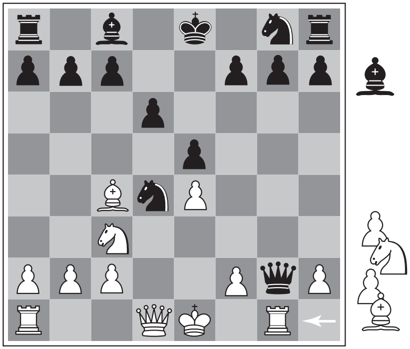
How to get training data for supervised learning? More later.

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:

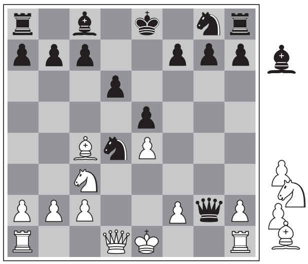
- "Killer moves"—capturing opponent's pieces, check etc.—should be considered first.
- *Quiescence search* – EVAL function should be applied only once things calm down. During capturing of pieces, depth should be locally increased.

Notes

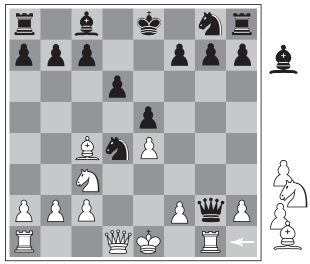
Cutting search at a wrong moment – important moves/changes are beyond horizon. Think abou the two situations – states s_a, s_b on the right. They are almost indentical. The only difference is the position of white rook, see bottom right corner. Very likely:

$$\text{EVAL}(s_a) \approx \text{EVAL}(s_b)$$

for many possible EVAL functions.



(a) White to move



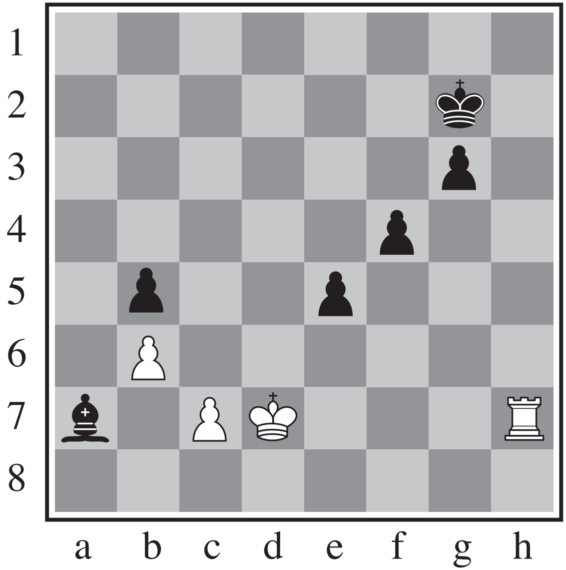
(b) White to move

A good heuristics – which moves to be considered first – may help a lot. Remember perfect ordering from α-β pruning?

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.



Notes

The horizon effect is difficult to mitigate. Singular extension may help. It is a move that is clearly better than others at this position. Once discovered in the search tree, remember it and use whenever appropriate.

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”.
  - Deep Blue examined 200M positions per second.
  - In some cases, depth of search was 40 ply.
- 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).

Notes

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.

Notes