Reinforcement Learning

Prof. Christian Bauckhage

lecture 02

inference and decision making with game trees

in this lecture, we look at how game trees allow for *inference* about games and for *optimal decision making* when playing *zero-sum games*

$\Leftrightarrow$ we have a look at the idea of *backward induction* and study the *minmax algorithm* and some of its variants (minmax with α-β *pruning* and *depth limited search*)

along the way, we will encounter numerous concepts which will reoccur again and again throughout this course

outline

recap / introduction

for games with discrete state spaces, we have

game tree

$\Leftrightarrow$ tree whose nodes represent game states and whose edges represent moves

complete game tree

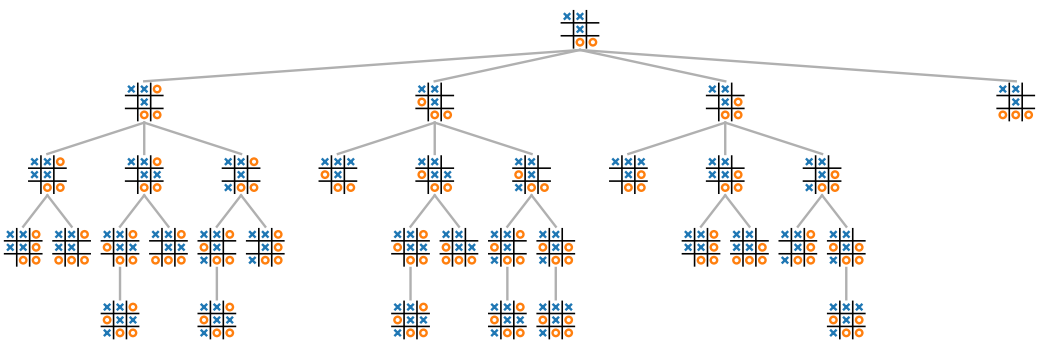
$\Leftrightarrow$ tree whose root node represents the *initial game state* and whose leaf nodes represent all *terminal game states*, i.e. all possible outcomes of a game

note

as game trees go, the ***tic tac toe*** game tree is tiny

and yet it is way too big to reasonably fit on a slide

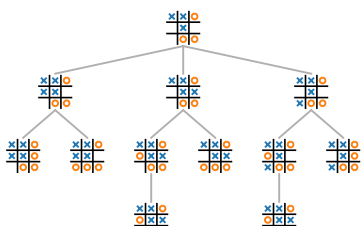
$\Rightarrow$ we typically only show incomplete ***tic tac toe*** trees



note

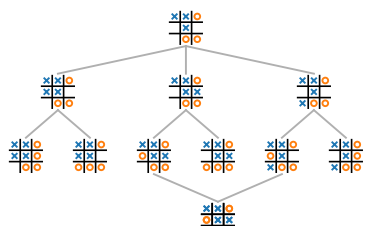
we often have

game tree $\Leftrightarrow$ “blown up” *partial order* of game states



game tree

vs



partially ordered set

partially ordered set (poset)

$\Leftrightarrow$ a pair $(\mathcal{P}, \sqsubseteq)$ where

$\mathcal{P}$ is a set

$\sqsubseteq$ is a binary relation such that for any $a, b, c \in \mathcal{P}$

$a \sqsubseteq a$ reflexivity

$a \sqsubseteq b \wedge b \sqsubseteq a \Rightarrow a = b$ anti-symmetry

$a \sqsubseteq b \wedge b \sqsubseteq c \Rightarrow a \sqsubseteq c$ transitivity

note

the relation $\sqsubseteq$ is *not* necessarily
defined for every $(a, b) \in \mathcal{P} \times \mathcal{P}$

we therefore additionally define

$$a \sqsubseteq b \vee b \sqsubseteq a \;\Leftrightarrow\; a \text{ and } b \text{ are comparable}$$

$$\neg(a \sqsubseteq b \vee b \sqsubseteq a) \Leftrightarrow\; a \text{ and } b \text{ are incomparable}$$

example

given the *blockworld* on the right, we have

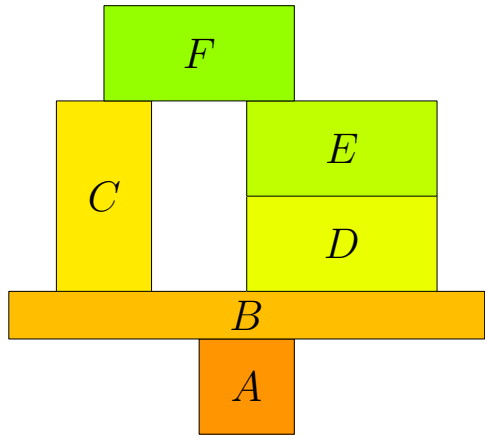
$$\mathcal{P} = \{A, B, C, D, E, F\}$$

with, for instance, these relations

$$A \sqsubseteq C$$

$$A \sqsubseteq D$$

$$C \not\sqsubseteq D$$



disclaimer

to be able to fully appreciate some of the material studied below,
we also recall / introduce the following concepts

(maximum or deepest) depth D of a tree

$\Leftrightarrow$ the maximum depth of a leaf node in a tree

minimum or shallowest depth d of a tree

$\Leftrightarrow$ the shallowest depth of a leaf node in a tree

(average) branching factor b of a tree

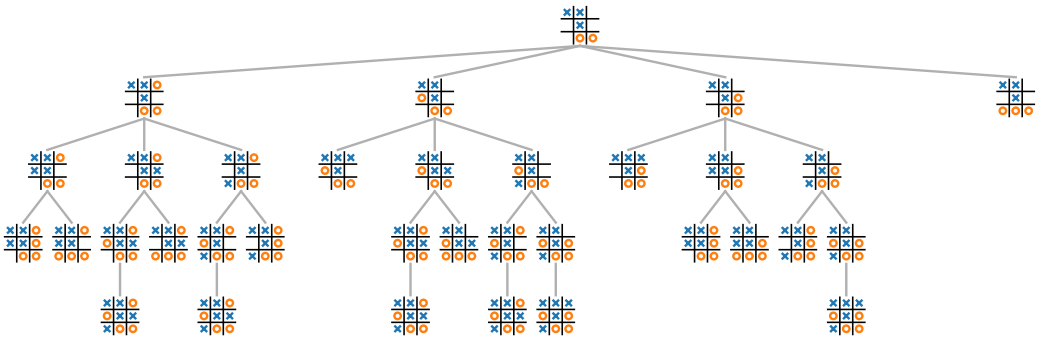
$\Leftrightarrow$ the (average) number of children of a node in a tree

examples

$D = 4$

$d = 1$

$b \approx 1.94$



for the game tree of ***chess*** , the average branching factor has been *estimated* to be

$$b \approx 35$$

$\Leftrightarrow$ on average, a ***chess*** player can choose between 35 legal moves in any game state

for the game tree of ***chess*** , the average branching factor has been *estimated* to be

$$b \approx 35$$

$\Leftrightarrow$ on average, a ***chess*** player can choose between 35 legal moves in any game state

for the game tree of ***go*** , the average branching factor has been *estimated* to be

$$b \approx 250$$

for the game tree of ***connect 4*** , it is *known* that the average branching factor is

$$b = 4$$

note

the question of whether or not to include leafs when computing b causes confusion among novices, indeed ...

a typical tree has lot's of leaf nodes

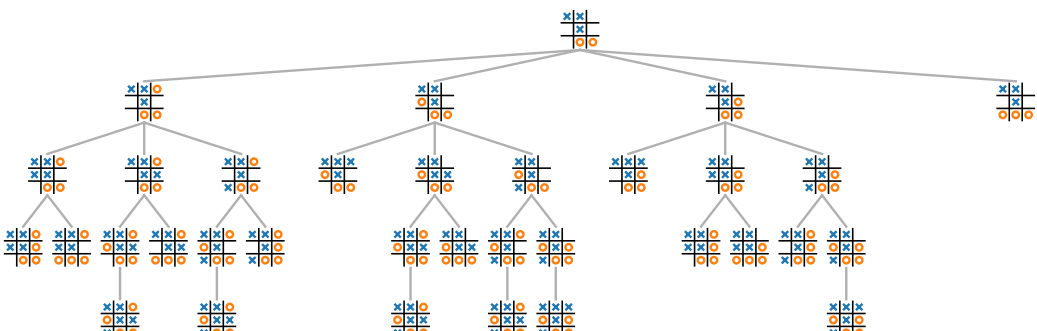
leaf nodes do not have *successors*, their *out degree* is 0

$\Leftrightarrow$ inclusion or exclusion of leafs crucially impacts the value of b

the *common convention* is to exclude leaf nodes when computing b

working with game trees

let us look at how we might compute ***tic tac toe*** game trees
(we will “of course” proceed recursively)

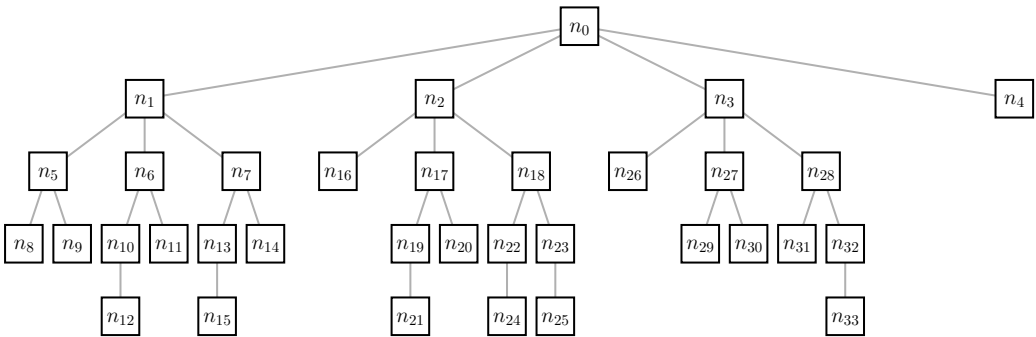


disclaimer

when we code / plot game trees, we write their nodes as $n_0, n_1, \ldots$

each game tree node n is uniquely associated with a game state s

to associate node n with state s, we will use a hash map $\texttt{n2s}[n] = s$



data structures (global for convenience)

```
node2state = {}   # maps tree nodes (numbers) to game states
node2succs = {}   # maps tree nodes to successors (captures state space structure)

def new_node_id():
    return 0 if not node2state else max(node2state.keys()) + 1
```

initialization and call of recursive procedure

```
B = ... # 3x3 array
p = ... # -1 or +1

root = new_node_id()

# register root state
node2state[root] = (B, p)

# populate dictionaries
build_ttt_tree(root)
```

```
def build_ttt_tree(node):
    (B,p) = node2state[node]                # board and player who is to move
    succs = []                              # initial list of successors of s

    if not is_terminal(s):
        for (r,c) in zip(*np.where(B==0)):
            B_ = np.copy(B); B_[r,c] = p    # board of successor state
            p_ = -p                         # player of successor state

            succ = new_node_id()            # get id of successor node

            node2state[succ] = (B_, p_)     # register successor state

            succs.append(succ)              # extend list of successors

    node2succ[node] = succs                 # register successors of node

    for succ in succs:
        build_ttt_tree(succ)
```

assignment

familiarize yourself with the *Python* graph processing library *networkx*

(this will come in very handy all throughout this course and its exercises)

disclaimer

our `Python` / `numpy` code is naturally very language specific

yet, henceforth, **we will often focus on high-level concepts**

function *Succ*(s)
    return $\{s' \mid s' = f(s,a),\ a \in \mathcal{A}(s)\}$

function *BuildTree*$(tree,\ s)$
    AddNode$(tree,\ s)$

    for s' in *Succ*(s)
        BuildTree$(tree,\ s')$

note

tic tac toe is a simple example of a deterministic, turn-based, two-player game of perfect information

it is also a canonical example of a *zero-sum game*

zero-sum game

$\Leftrightarrow$ a game in which a player's gain (or loss) of *utility* is exactly balanced by the loss(es) (or gain(s)) of the other player(s)

$\Leftrightarrow$ **when added, gains and losses of players sum to zero**

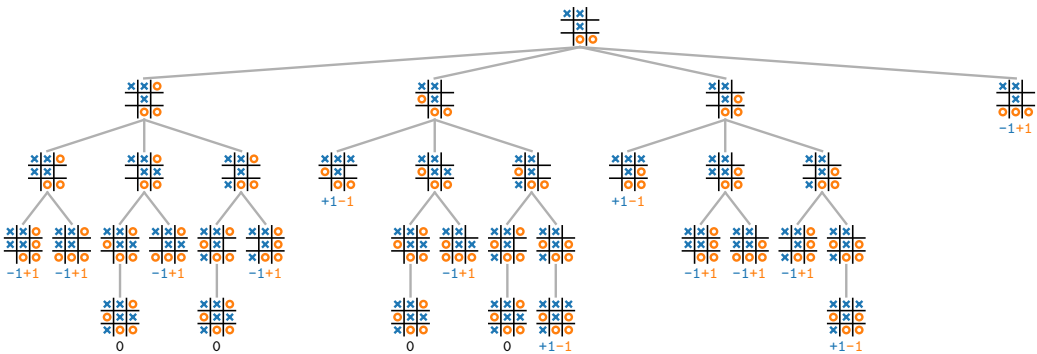
utility / payoff function

$\Leftrightarrow$ assigns a numerical value to the terminal states of a game

in a two player game, commonly chosen utilities or payoffs are $+1$, -1, and 0 for win, loss, or draw, respectively

example

reasonable payoffs for ***tic tac toe***



goal state

$\Leftrightarrow$ a terminal state of high utility (usually a winning state)

$\Leftrightarrow$ a leaf node of a game tree that has a high utility

backward induction

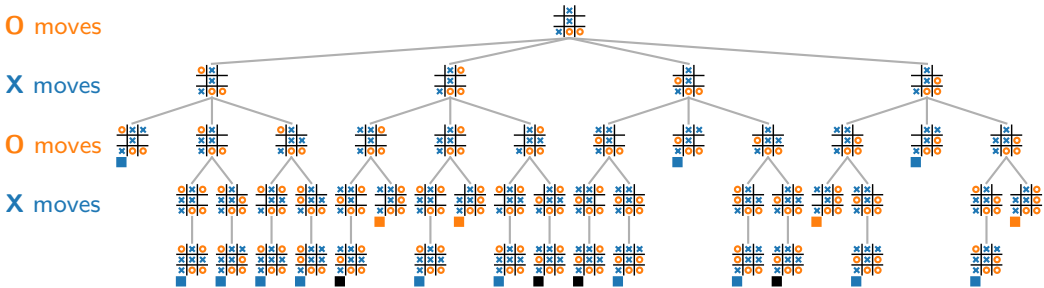
note

given a complete game tree and payoffs for terminals, it is possible to solve a game

$\Leftrightarrow$ assuming *optimal play* for both players, the outcome of the game can be determined

our following example will of course only consider an incomplete game tree . . .

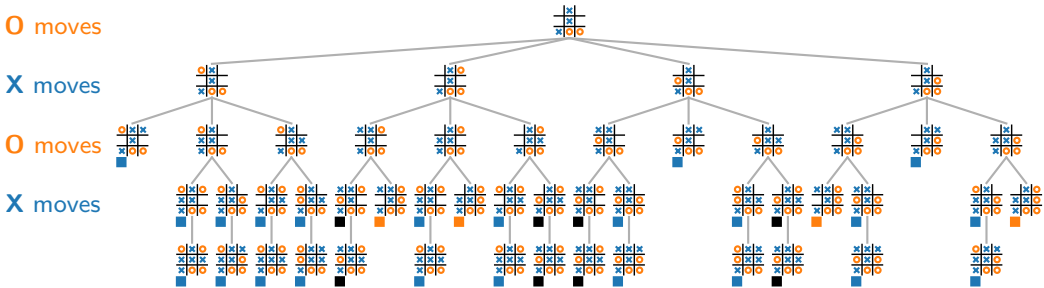
backward induction



1) “color” the terminal states of the game tree such that

each **X** win is marked ■, each **O** win is marked ■, and each draw is marked ■

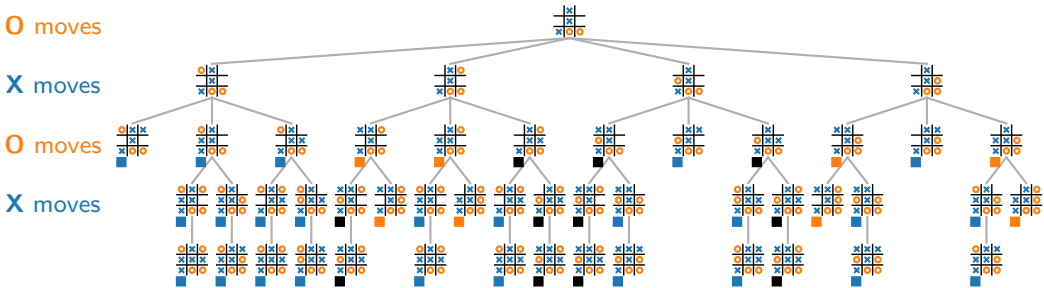
backward induction



2) iterate over the tree levels l from bottom to top

   if state s in level l is not yet colored and represents a state where player p is to move, set s to the color of a / the child that maximizes the payoff for p

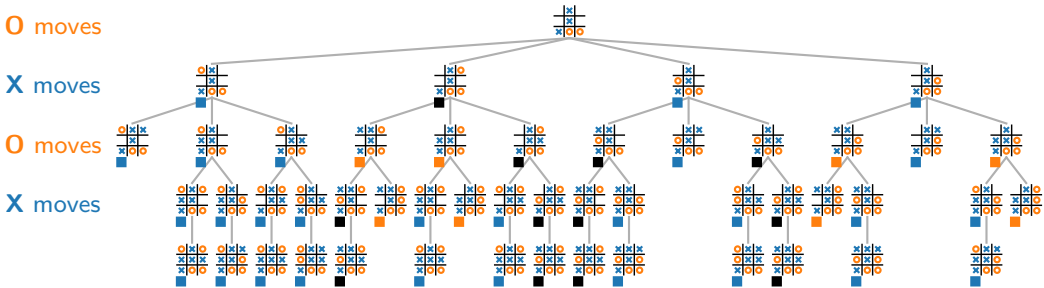
backward induction



2) iterate over the tree levels l from bottom to top

   if state s in level l is not yet colored and represents a state where player p is to move, set s to the color of a / the child that maximizes the payoff for p

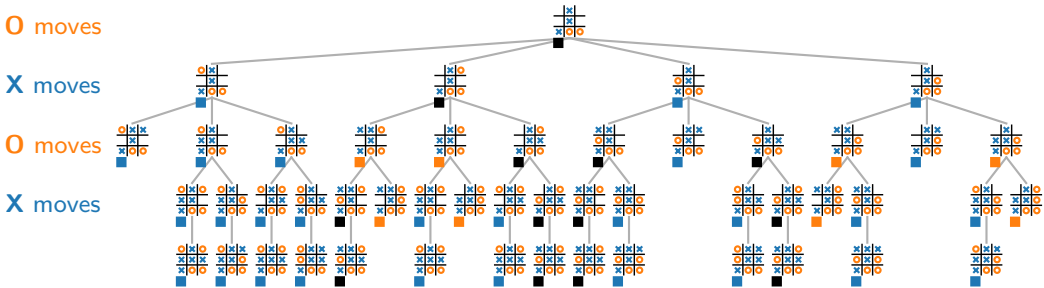
backward induction



2) iterate over the tree levels l from bottom to top

   if state s in level l is not yet colored and represents a state where player p is to move, set s to the color of a / the child that maximizes the payoff for p

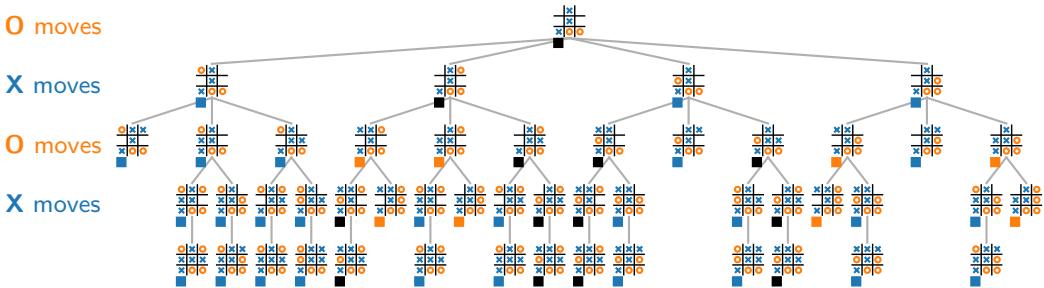
backward induction



2) iterate over the tree levels l from bottom to top

   if state s in level l is not yet colored and represents a state where player p is to move, set s to the color of a / the child that maximizes the payoff for p

backward induction



3) upon termination, the color of the root state determines the nature of the game

remarks

with “optimal play”, ***tic tac toe*** will always end in a draw

this was the central motif of the 1983 movie *Wargames*

for ***checkers*** , it was also shown (after 18 years of processing on hundreds of top computers back then) that “optimal play” will lead to a draw

J. Schaeffer et al., *Checkers is Solved*, Science, 317(5844), **2007**

in ***connect 4*** , on the other hand, the opening player always wins

questions

bottom-up is nice . . .

but what about top-down ?

how to realize “optimal play” ?

answer

. . .

the minmax algorithm

disclaimers

in what follows, the player who is to move will be called MAX,
the opponent will be called MIN

in what follows, the terms *game state* and *(game tree) node*
will be used interchangeably

in what follows, an arbitrary game state will just be called s

in what follows, the terms *game tree* and *search tree* will be
used interchangeably, because

> looking for an optimal move is to *search* the game tree
> for a path from the current node n to a goal node

how *intelligent agents* play a game

in state s, MAX will try to move towards a *goal state*

in the subsequent state s', MIN will try do the same

how *intelligent agents* play a game

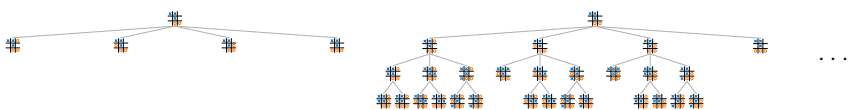
in state s, MAX will try to move towards a *goal state*

in the subsequent state s', MIN will try do the same

$\Rightarrow$ MAX has to find an *optimal strategy* which specifies

how to move in s
how to move in states resulting from possible responses
how to move in states resulting from possible responses to possible responses

$\vdots$



strategy

$\Leftrightarrow$ a *plan* of how to achieve an overall goal (usually under uncertain, unpredictable or adversarial conditions)

optimal strategy

$\Leftrightarrow$ a strategy that is *at least as good* as any other strategy when playing against an infallible opponent

strategy is a game-theory term the *policies* we talk about in RL are essentially the same thing

question

how to find an optimal strategy ?

answer

it is “easy” to come up with an algorithm :-)

it is usually infeasible to apply this algorithm :-(

by introducing *heuristics*, the algorithm becomes applicable :-|

MAX's reasoning

I should move in a "direction" that maximizes my payoff

however, if I do so, *MIN* will try to counter my move and move in a "direction" that minimizes my payoff

$\Rightarrow$ I should try to

maximize my minimum gain

or at least try to

minimize my maximum loss

minmax algorithm

$\Leftrightarrow$ a *decision rule* for how to play against an infallible opponent

$\Leftrightarrow$ try to *minimize maximum loss* or to *maximize minimum gain*

minmax algorithm

assumption

players can compute the utilities

$$Util(t)$$

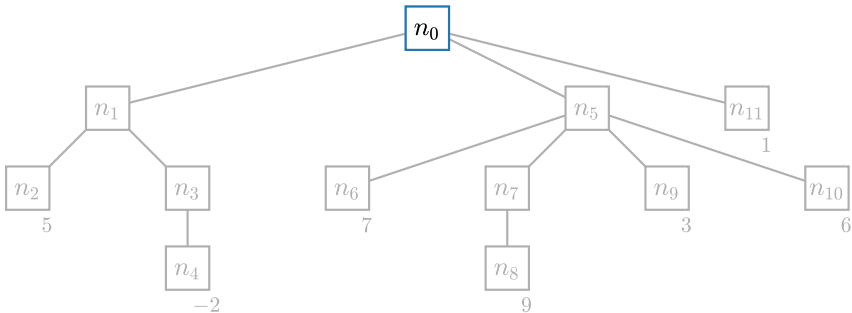
for every terminal game state t

basic idea

to decide for a move in state s, recursively compute a minmax value mmv for every state that can result from s and then move to *a* or *the* state s' for which $mmv(s')$ is largest

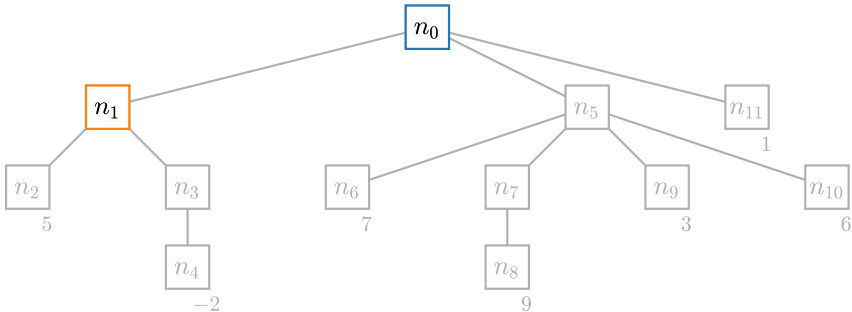
minmax algorithm

$$mmv(s) = \begin{cases} Util(s) & \text{if } s \text{ is terminal} \\ \max_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } MAX \text{ state} \\ \min_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } MIN \text{ state} \end{cases}$$



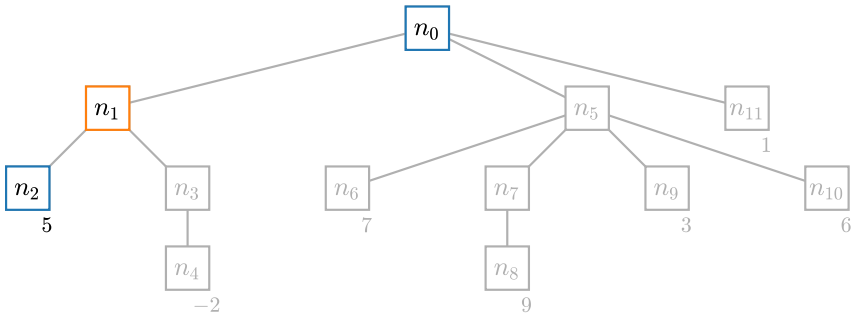
minmax algorithm

$$mmv(s) = \begin{cases} Util(s) & \text{if } s \text{ is terminal} \\ \max\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } MAX \text{ state} \\ \min\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } MIN \text{ state} \end{cases}$$



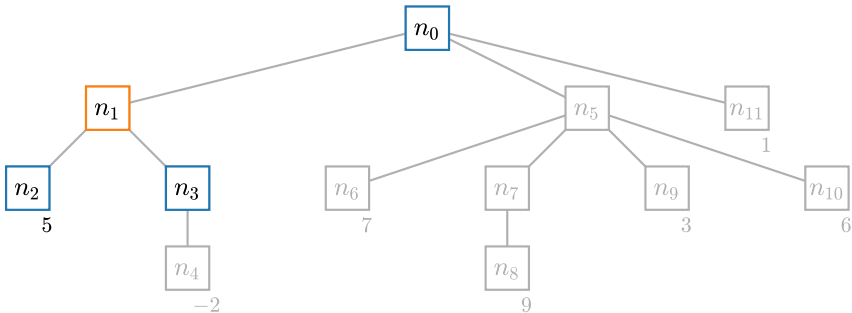
minmax algorithm

$$mmv(s) = \begin{cases} Util(s) & \text{if } s \text{ is terminal} \\ \max\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } MAX \text{ state} \\ \min\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } MIN \text{ state} \end{cases}$$



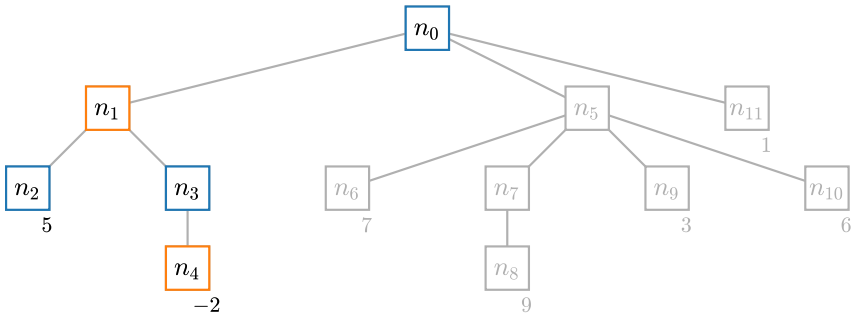
minmax algorithm

$$mmv(s) = \begin{cases} Util(s) & \text{if } s \text{ is terminal} \\ \max\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } MAX \text{ state} \\ \min\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } MIN \text{ state} \end{cases}$$



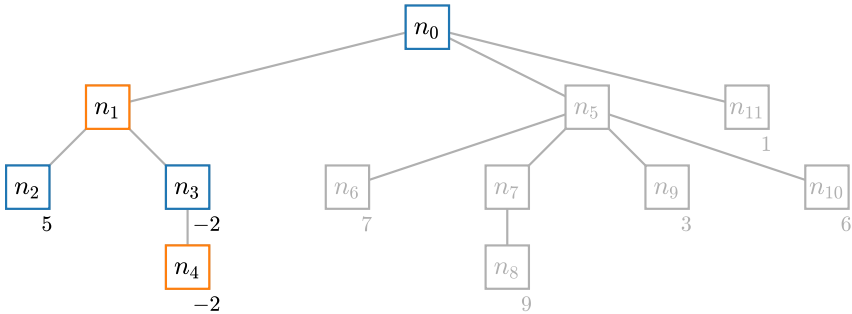
minmax algorithm

$$mmv(s) = \begin{cases} Util(s) & \text{if } s \text{ is terminal} \\ \max\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } \textit{MAX} \text{ state} \\ \min\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } \textit{MIN} \text{ state} \end{cases}$$



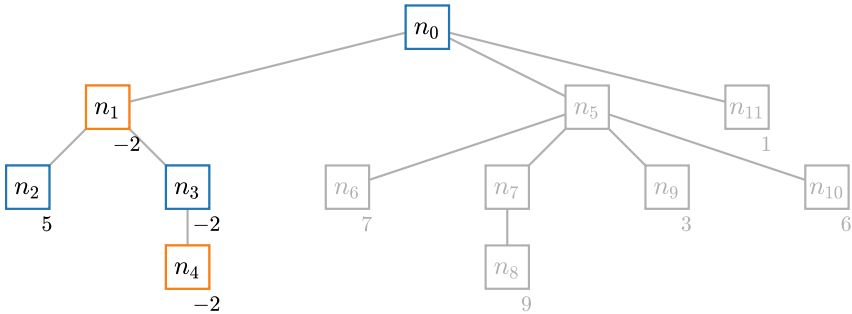
minmax algorithm

$$mmv(s) = \begin{cases} Util(s) & \text{if } s \text{ is terminal} \\ \max\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } \textit{MAX} \text{ state} \\ \min\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } \textit{MIN} \text{ state} \end{cases}$$



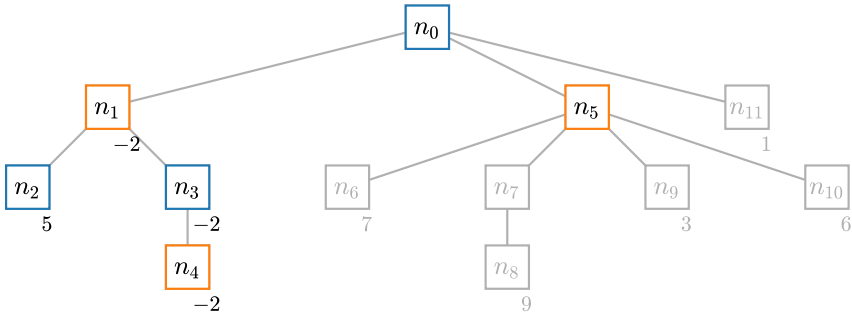
minmax algorithm

$$mmv(s) = \begin{cases} Util(s) & \text{if } s \text{ is terminal} \\ \max\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } \textit{MAX} \text{ state} \\ \min\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } \textit{MIN} \text{ state} \end{cases}$$



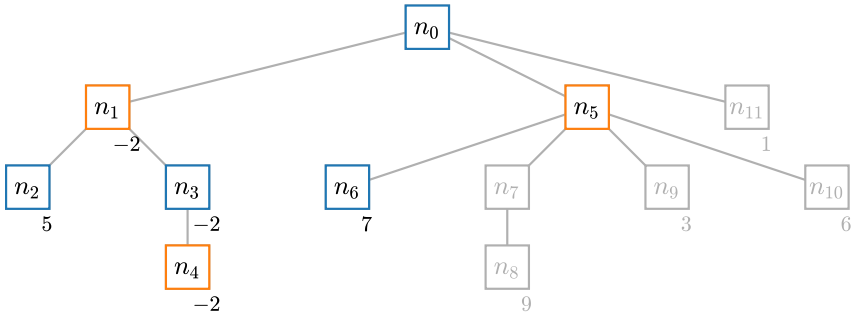
minmax algorithm

$$mmv(s) = \begin{cases} Util(s) & \text{if } s \text{ is terminal} \\ \max\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } MAX \text{ state} \\ \min\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } MIN \text{ state} \end{cases}$$



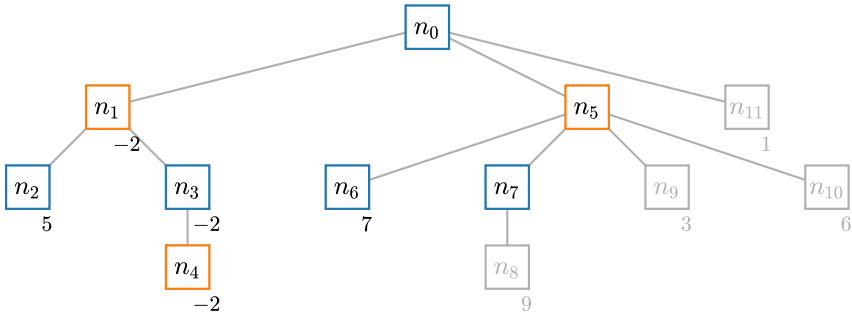
minmax algorithm

$$mmv(s) = \begin{cases} Util(s) & \text{if } s \text{ is terminal} \\ \max\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } MAX \text{ state} \\ \min\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } MIN \text{ state} \end{cases}$$



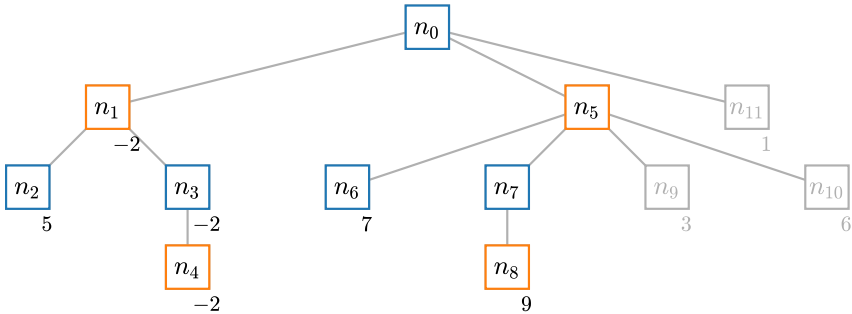
minmax algorithm

$$mmv(s) = \begin{cases} Util(s) & \text{if } s \text{ is terminal} \\ \max_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } MAX \text{ state} \\ \min_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } MIN \text{ state} \end{cases}$$



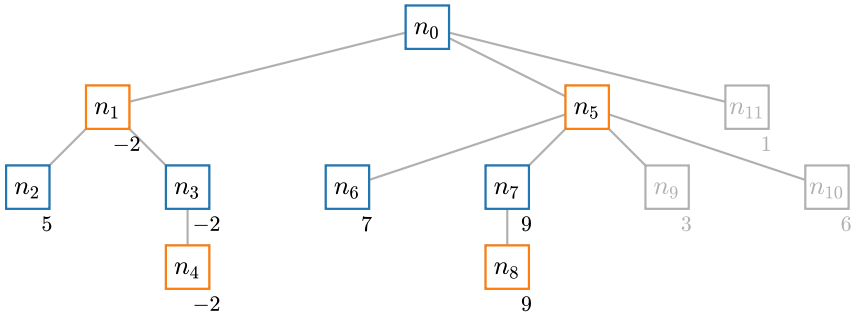
minmax algorithm

$$mmv(s) = \begin{cases} Util(s) & \text{if } s \text{ is terminal} \\ \max_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } \textit{MAX} \text{ state} \\ \min_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } \textit{MIN} \text{ state} \end{cases}$$



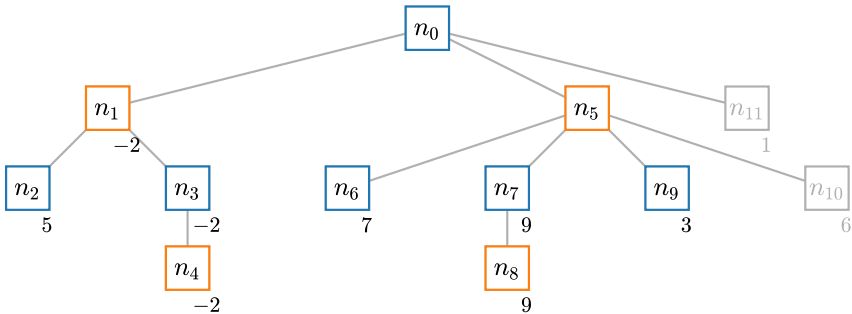
minmax algorithm

$$mmv(s) = \begin{cases} Util(s) & \text{if } s \text{ is terminal} \\ \max\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } \textit{MAX} \text{ state} \\ \min\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } \textit{MIN} \text{ state} \end{cases}$$



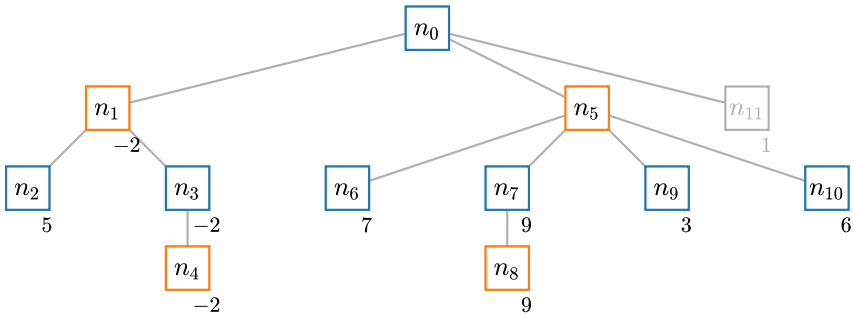
minmax algorithm

$$mmv(s) = \begin{cases} Util(s) & \text{if } s \text{ is terminal} \\ \max_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } MAX \text{ state} \\ \min_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } MIN \text{ state} \end{cases}$$



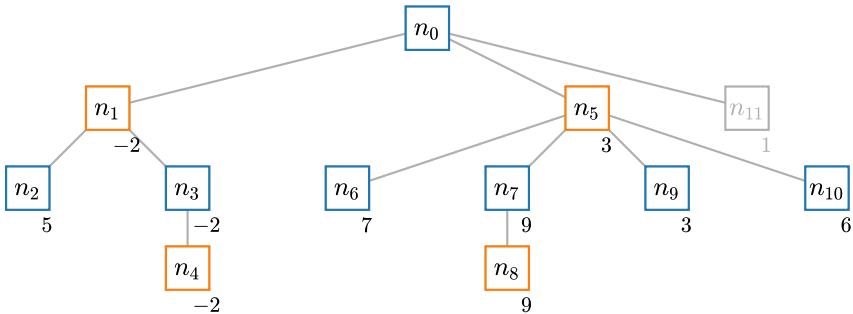
minmax algorithm

$$mmv(s) = \begin{cases} Util(s) & \text{if } s \text{ is terminal} \\ \max\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } MAX \text{ state} \\ \min\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } MIN \text{ state} \end{cases}$$



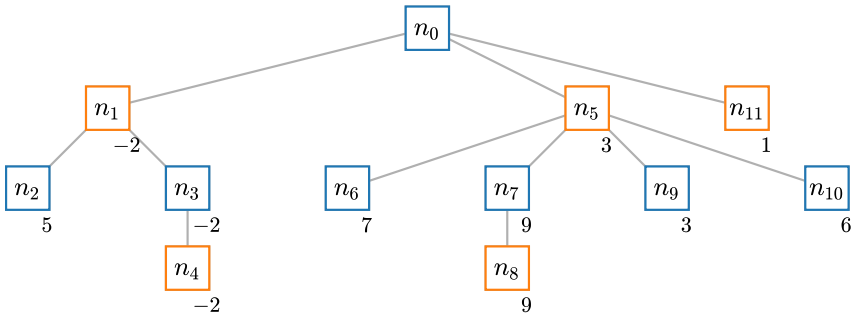
minmax algorithm

$$mmv(s) = \begin{cases} Util(s) & \text{if } s \text{ is terminal} \\ \max\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } MAX \text{ state} \\ \min\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } MIN \text{ state} \end{cases}$$



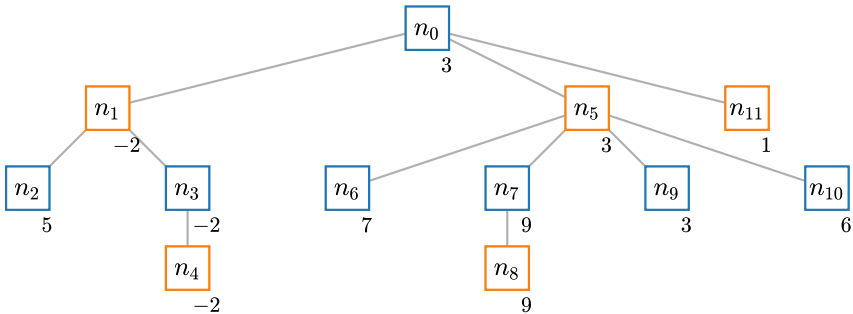
minmax algorithm

$$mmv(s) = \begin{cases} Util(s) & \text{if } s \text{ is terminal} \\ \max\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } MAX \text{ state} \\ \min\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } MIN \text{ state} \end{cases}$$



minmax algorithm

$$mmv(s) = \begin{cases} Util(s) & \text{if } s \text{ is terminal} \\ \max\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } MAX \text{ state} \\ \min\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } MIN \text{ state} \end{cases}$$



question

what if MIN does not play optimally ?

answer

even better for MAX

question

what if MIN does not play optimally ?

answer

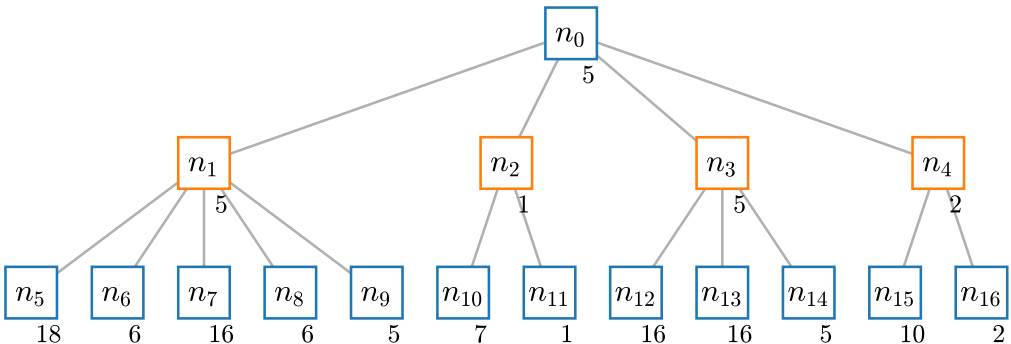
even better for MAX, because

$$\begin{aligned} mmv(s) &= \max \left\{ mmv(s') \;\middle|\; s' \in Succ(s) \right\} \\ &= \max \left\{ \min \left\{ mmv(s'') \;\middle|\; s'' \in Succ(s') \right\} \;\middle|\; s' \in Succ(s) \right\} \\ &\leqslant \max \left\{ \text{rnd} \left\{ mmv(s'') \;\middle|\; s'' \in Succ(s') \right\} \;\middle|\; s' \in Succ(s) \right\} \end{aligned}$$

riddle me this

where should *MAX* move to? to state n2s$[n_1]$ or to state n2s$[n_3]$?

does it make a difference, if *MAX* moves to n2s$[n_1]$ or to n2s$[n_3]$?



note

the minmax algorithm is a depth-first search algorithm

the number of states to be examined is *exponential* in the number of possible moves

$\Leftrightarrow$ the worst case effort is $O(b^D)$

$\Rightarrow$ **it is usually *infeasible* to compute the (true) minmax value of every state encountered throughout a game**

for ***tic tac toe*** it is feasible but already painful

“obvious” solutions

recall that search tree nodes often reoccur multiple times

also, we can often compute *hash values* for game states

$\Rightarrow$ create a lookup table (aka a *transposition table*) of previously seen states

$\Rightarrow$ if minmax encounters a node that has already been expanded in another branch of the game tree, do not expand it again but read its mmv from the lookup table

“less obvious” solutions

using pruning techniques, large parts of the game tree do not have to be searched

often, it is possible to compute the correct minmax decision *without* exploring every game state (node in the game tree)

often, the worst case effort exponent can be halved $O\left(b^{\frac{D}{2}}\right)$

minmax with α-β pruning

pruning (in the context of AI)

$\Leftrightarrow$ eliminating possibilities from consideration without having to examine them

minmax with α-β pruning (w/o detailed explanation)

for each node, register $[\alpha, \beta]$ where

$\alpha =$ highest *mmv* found so far for *MAX*

$\beta =$ lowest *mmv* found so far for *MIN*

this typically involves these alternating functions

```
function MaxVal(s, α, β)
    if isTerminalState(s)
        return Util(s)
    for s' in Succ(s)
        α ← max(α, MinVal(s', α, β))
        if α ⩾ β
            return α
    return α
```

```
function MinVal(s, α, β)
    if isTerminalState(s)
        return Util(s)
    for s' in Succ(s)
        β ← min(β, MaxVal(s', α, β))
        if β ⩽ α
            return β
    return β
```

minmax with α-β pruning (w/o detailed explanation)

change minmax from

$$mmv(s) = \begin{cases} Util(s) & \text{if } s \text{ is terminal} \\ \max\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } MAX \text{ state} \\ \min\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } MIN \text{ state} \end{cases}$$

to this call of $MaxVal$

$$mmv(s) = MaxVal(s, -\infty, +\infty)$$

“one can show” that, if successors are expanded in a random order, the number of nodes to be examined is $O\left(b^{\frac{3}{4}D}\right)$

“one can show” that, *if successors are ranked according to utility*, the number of nodes to be examined is $O\left(b^{\frac{1}{2}D}\right)$

observe that $b^{\frac{D}{2}} = \left(\sqrt{b}\right)^D$

$\Rightarrow$ the effective branching factor reduces from b to $\sqrt{b}$

observe that $D = 2 \cdot \frac{D}{2}$

$\Rightarrow$ in the same amount of time, minmax with α-β pruning can look up to twice as far ahead as ordinary minmax

note

even with pruning techniques such as α-β pruning, minmax has to *search* all the way down to the leafs of the search tree

while this is feasible for games such as ***tic tac toe*** or ***connect 4*** , it is usually infeasible for games with larger branching factors

further “less obvious” solutions

in his 1950 paper, Shannon proposed to cut off searches after a few levels, i.e. to perform depth limited searches

to estimate utilities of non-terminal nodes, he proposed to apply heuristics or evaluation functions

minmax with depth limitation

heuristic / evaluation function

$\Leftrightarrow$ provides an estimate of the utility of a game state which is *not* a terminal state

cut-off function

$\Leftrightarrow$ decides whether to stop search tree expansion at state s

minmax with depth limitation

change minmax from

$$mmv(s) = \begin{cases} Util(s) & \text{if } s \text{ is terminal} \\ \max\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } MAX \text{ state} \\ \min\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } MIN \text{ state} \end{cases}$$

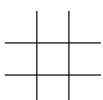
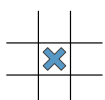
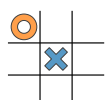
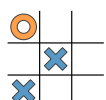
to depth limited minmax

$$mmv(s) = \begin{cases} Eval(s) & \text{if } CutOff(s) \text{ yields } \texttt{true} \\ \max\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } MAX \text{ state} \\ \min\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } MIN \text{ state} \end{cases}$$

example

a simple evaluation function for ***tic tac toe***

$$Eval(\boldsymbol{B}, p) = (\text{number of lines where } p \text{ can win}) - (\text{number of lines where } o \text{ can win})$$

s				
$Eval(\boldsymbol{B},\mathbf{X})$	$8-8=0$	$8-4=+4$	$5-4=+1$	$5-3=+2$
$Eval(\boldsymbol{B},\mathbf{O})$	$8-8=0$	$4-8=-4$	$4-5=-1$	$3-5=-2$

question

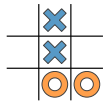
is this evaluation function any good ?

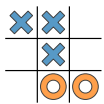
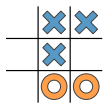
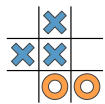
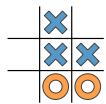
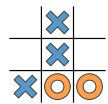
answer

let's see ...

example

for player **X**, evaluate all successors of the state with board configuration is



s					
$Eval(\boldsymbol{B},\mathbf{X})$	2	2	2	2	3

remarks

general criteria for an evaluation function

- $Eval(s)$ should be “easy” to compute
- $Eval(s)$ should order terminal states as $Util(s)$ would do
- $Eval(s)$ should correlate with the *chance of winning* if s is non-terminal

if searches stop at non-terminals, minmax will be uncertain about the true $mmv(s)$

traditionally, evaluation functions are conceived / designed by human engineers

evaluations provide numerical features

$$f_1^s, f_2^s, \ldots, f_k^s$$

of a state s

common / canonical examples include

- number of winning lines for ***tic tac toe***
- *material values* of pieces on a ***chess*** board
- ⋮

numerical features of s can be gathered in a feature vector

$$\boldsymbol{f}^s = \left[f_1^s \;\; f_2^s \;\; \cdots \; f_k^s \right]^\mathsf{T} \in \mathbb{R}^k$$

feature vectors can be *clustered* into game state categories

experience ($\Leftrightarrow$ statistics) can be used to estimate how likely *category* $C(s)$ leads to a win $(+1)$, a loss (-1), or a draw (0) later on ...

note

among others, this allows for defining evaluation functions like

$$Eval(s) = p_w(C(s)) \cdot (+1) \; + \; p_l(C(s)) \cdot (-1) \; + \; p_d(C(s)) \cdot 0$$

where

$$p_w + p_l + p_d = 1$$

function $Eval(s)$ defined above computes an expected value

$\Leftrightarrow$ **we can work with expected values to evaluate (game) states**

this is foreshadowing !!!

note

evaluation functions can also be used to *rank* game states

recall that rankings allow for more efficient α-β pruning

they also allow for realizing best-first search algorithms

question

what about games with randomness (e.g. ***backgammon***) ?

answer

introduce a *chance node* after every MAX and MIN node

minmax under uncertainty

chance node

$\Leftrightarrow$ indicates outcome x and probability $p(x)$ of a random event X

minmax under uncertainty

change minmax from

$$mmv(s) = \begin{cases} Util(s) & \text{if } s \text{ is terminal} \\ \max\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } MAX \text{ state} \\ \min\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is } MIN \text{ state} \end{cases}$$

to expected minmax

$$emmv(s) = \begin{cases} Util(s) & \text{if } s \text{ is terminal} \\ \max\limits_{s' \in Succ(s)} emmv(s') & \text{if } s \text{ is } MAX \text{ state} \\ \min\limits_{s' \in Succ(s)} emmv(s') & \text{if } s \text{ is } MIN \text{ state} \\ \sum\limits_{s' \in Succ(s)} p(s') \cdot emmv(s') & \text{if } s \text{ is chance state} \end{cases}$$

history and state of the art

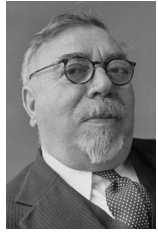
the history of minmax

J. von Neumann, *Zur Theorie der Gesellschaftsspiele*, Mathem. Annalen, 100(1), **1928**

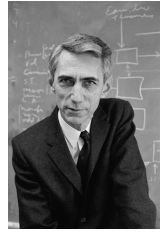
N. Wiener, *Cybernetics or Control and Communication in the Animal and the Machine*, MIT Press, **1948**

C. Shannon, *Programming a Computer for Playing Chess*, Philosophical Magazine, Ser.7, 41(314), **1950**

J. von Neumann
(∗1903, †1957)



N. Wiener
(∗1894, †1964)



C. Shannon
(∗1916, †2001)

the history of α-β pruning

discovered several times

McCarthy **1955**, Newell and Simon **1958**, Knuth and Moore **1975**, ...

conclusively settled by Pearl

J. Pearl, *The Solution for the Branching Factor of the Alpha–beta Pruning Algorithm and its Optimality*, Comm. of the ACM 25(8), **1982**

J. Pearl ($*$1936)

the old state of the art (pre 2016)

programs that play *chess*, *checkers*, etc. rely on

- minmax with α-β pruning or other cutoff techniques
- sophisticated and highly tuned evaluation functions
- large transposition tables and databases of opening- and endgame-moves

Deep Blue stats

- on average evaluated 126 000 000 nodes per second
- generated about 30 000 000 positions per move and thus typically computed a look-ahead of depth 14
- considered 8 000 features for heuristic node evaluation
- accessed a data base of 700 000 grandmaster games

the new state of the art (post 2016)

AlphaGO (2016) characteristics

combines tree search and machine learning

$\Leftrightarrow$ uses *Monte Carlo tree search* guided by a *value-* and a *policy-network*

the latter are *deep neural nets* trained with a vast number of examples from human gameplay (for bootstrap) and selfplay (for refinement)

AlphaGO Zero (2017) characteristics

works just as AlpheGO but creates training data from scratch (using extensive selfplay)

AlphaZero (2017) characteristics

can play any two player turn based game and outperforms all previous chess programs

note

AlphaGO, AlphaGO Zero, and AlphaZero show superhuman performance

they all require “insane” amounts of compute power (at least during training)

note

AlphaGO, AlphaGO Zero, and AlphaZero show superhuman performance

they all require “insane” amounts of compute power (at least during training)

consider this

> GPU clusters have an energy expenditure of about 10 000 000 watts

this is insane, because

> the average human brain consumes (at max) 500 kilocalories per day
>
> $\Leftrightarrow$ the average human brain has an energy expenditure of only 24 watts

for details, see slide 80

summary

we now know that

there exists an algorithm for *optimal decision making* (in (2-player) zero-sum games)

the *minmax algorithm* is a *tree search algorithm* that asks a lot of “what if” questions

action selection based on *minmax values* provides players with an optimal strategy

the computation of minmax values is based on the notion of *utility values* or *payoffs* of terminal states or leaf nodes of a game tree

the minmax algorithm *propagates* utility values from leafs to higher level tree nodes

alas . . .

the computational complexity of the plain vanilla minmax algorithm usually prevents its use in practice (even for more modern versions such as *negmax* or *negascout* which we ignored)

$\Rightarrow$ from the 1950s to the 1990s, people have developed many (heuristic) modifications

notable modifications involve

- state evaluation functions
- the incorporation of elements of chance
- the computation of numerical features of states
- the computation of expected values of the utilities of states

these ideas will reoccur later on, because ...

note

AI		RL
strategy	$\sim$	policy
utility	$\sim$	reward
expected utility	$\sim$	expected return
evaluation function	$\sim$	value function
chance nodes	$\sim$	Markov processes

some biology and physics

a human body needs up to $2\,500\,000$ calories per day; as the brain uses 20% of the body's energy, it consumes up to $500\,000$ calories per day

a calorie is a unit of *energy* ; the SI unit for energy is joule, and we have

$$1\,\text{cal} = 4.184\,\text{J}$$

a watt is an SI unit of *power* (energy consumed per time), and we have

$$1\,\text{W} = 1\,\text{J/s}$$

$\Rightarrow$ we get this estimate for the energy expenditure of a hungry human brain

$$\frac{500\,000\,\text{cal}}{1\,\text{day}} = \frac{2\,092\,000\,\text{J}}{86\,400\,\text{s}} \approx 24\,\text{W}$$