Game Search

CSCE 420 – Spring 2023

read: Ch. 5

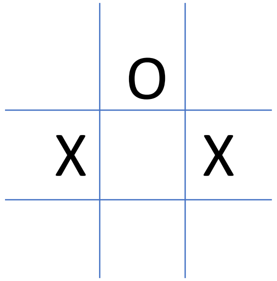
Game Search

- games are useful to study for AI because they represent adversarial environments
  - the world state is not controlled solely by the agent
  - the world state can change because of actions by other agents (players)
  - different agents might have different objectives
  - this can lead to *competitive* behavior, or *cooperative* behavior
- there are many different kinds of games
  - simultaneous vs. sequential vs. iterated
  - single-player, two-player, multi-player
  - stochastic games with an element of chance
  - complete vs. incomplete information (*partially observable*)
  - also applies to economics: pricing of goods, auctions, contract negotiations...
- Of course, DeepBlue and AlphaGo are widely-recognized successes in AI, representing achievement of intelligent behaviour

Sequential Games

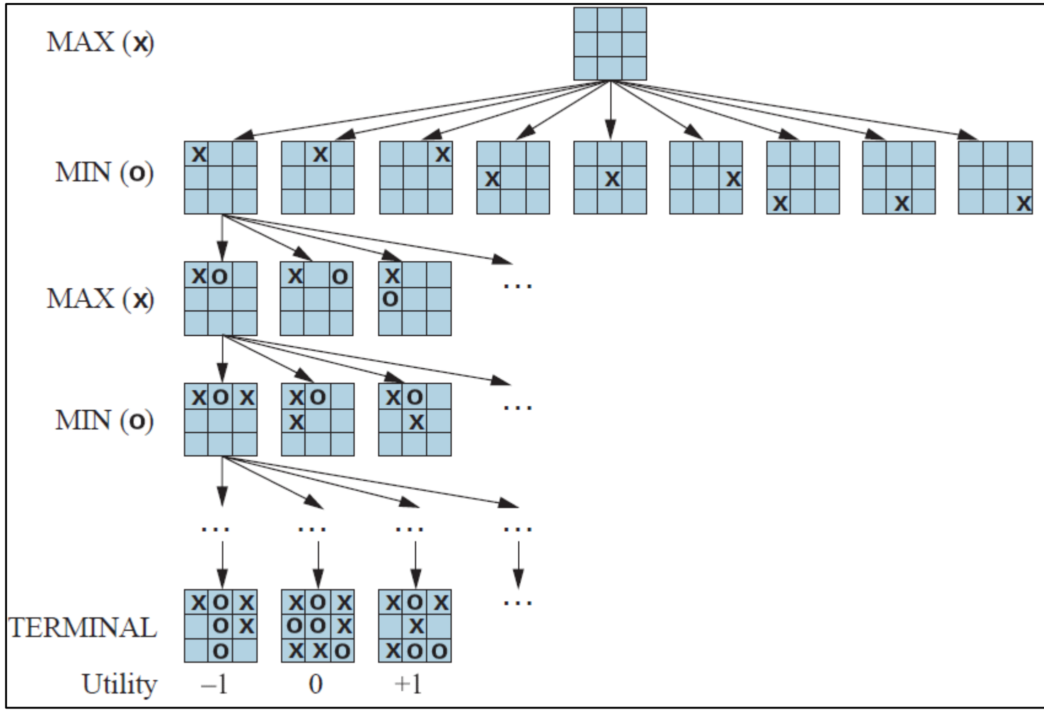
- multiple steps – players take turns
- each player has a utility function
  - +1 for win; -1 for lose; 0 for draw (tic-tac-toe); 0 for non-terminal states
  - money (poker)
  - rewards for achieving goals - cost of actions or resources used
- simplest form: 2-player, 0-sum games
  - $\Sigma_i u_i(s) = 0$ or $u_1(s) = -u_2(s)$
- examples: tic-tac-toe, checkers, chess...

Minimax Search



- in a 2-player, 0-sum game like tic-tac-toe, how can we decide what move to make?
- method 1: write a bunch of rules that encode a *strategy*
- method 2: use systematic search
  - use *look-ahead* for each possible action to imagine what opponent response might be
  - key idea: we can anticipate what move the opponent will make, because their utility is assumed to be the opposite of ours
  - thus the opponent will change the game in the way that is best for them, which is worst for us
  - *recursion*: of course, to simulate the opponent's reasoning, they will have to consider our response to their response, and so on...

Minimax Search



- recall that $u_i(s)=0$ for non-terminal states
- label alternating levels in search tree as max nodes and min nodes
- define *minimax* value for each state s as follows:

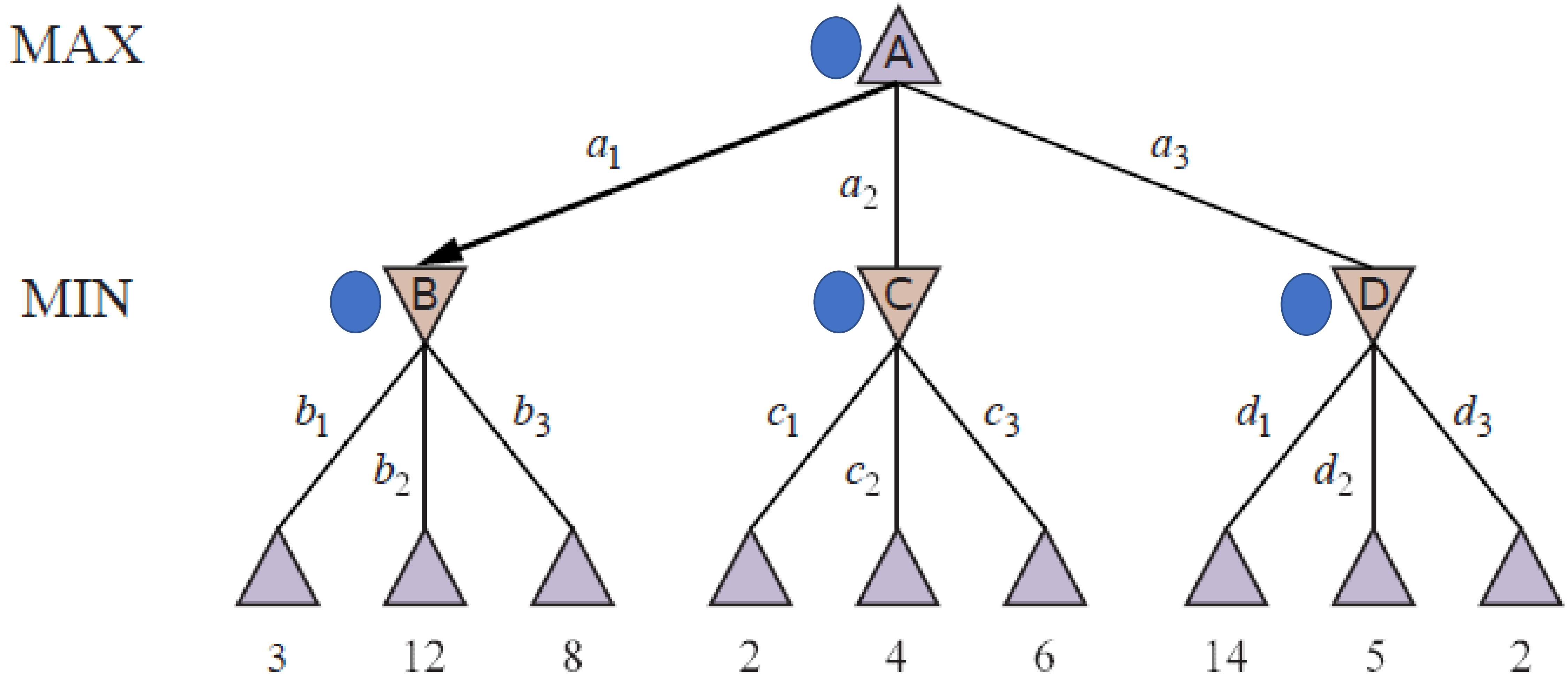
$$\text{minimax}(s) = \begin{cases} u_i(s) \text{ if s is a terminal state} \\ \max \{ \text{minimax}(s') \text{ for } s' \in \text{succ}(s) \} \text{ if s is a max node} \\ \min \{ \text{minimax}(s') \text{ for } s' \in \text{succ}(s) \} \text{ if s is a min node} \end{cases}$$

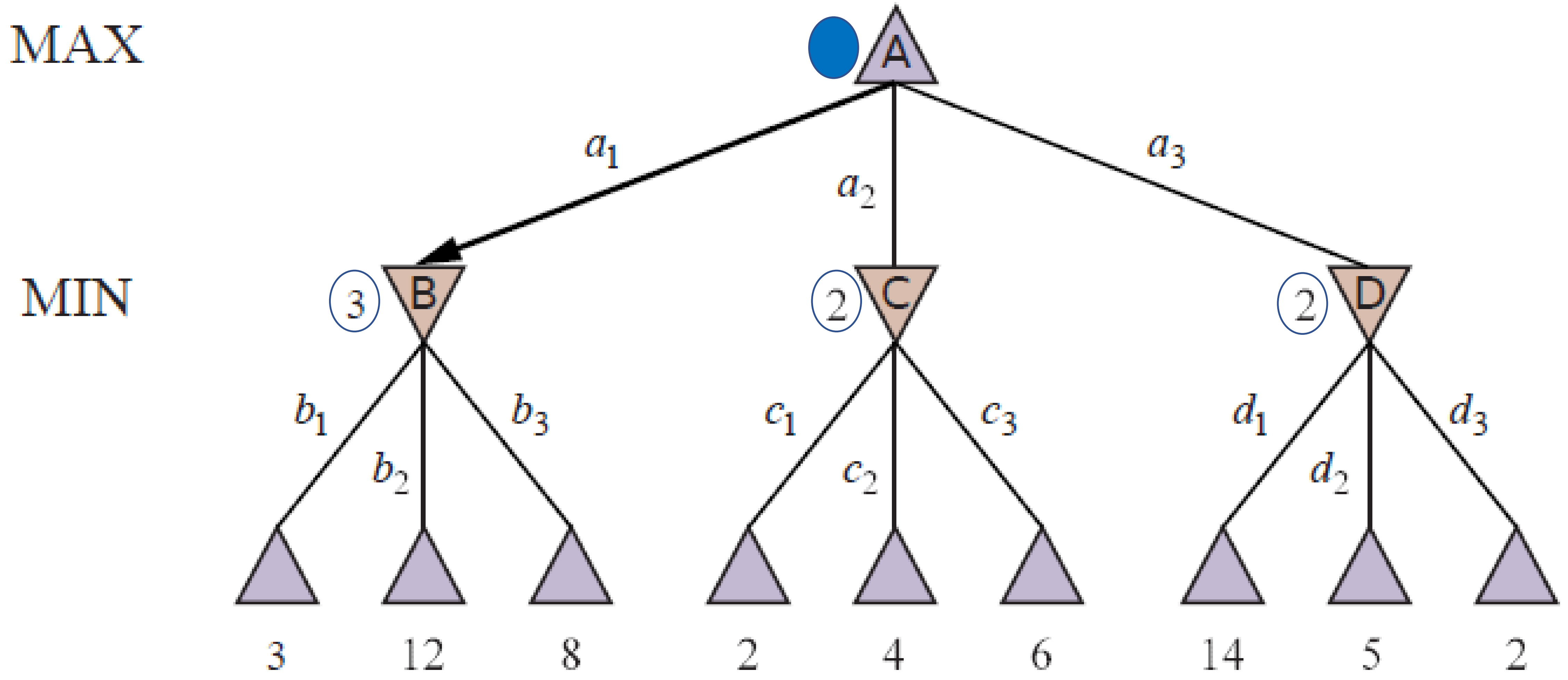
- decision at root node: argmax { minimax(s') for $s' \in$ succ(s) }
  - i.e. choose the action that leads to the successor with highest score, which has the highest expected payoff

Minimax Search

```
function MINIMAX-SEARCH(game, state) returns an action
    player ← game.TO-MOVE(state)
    value, move ← MAX-VALUE(game, state)
    return move

function MAX-VALUE(game, state) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v ← −∞
    for each a in game.ACTIONS(state) do
        v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
        if v2 > v then
            v, move ← v2, a
    return v, move

function MIN-VALUE(game, state) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v ← +∞
    for each a in game.ACTIONS(state) do
        v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
        if v2 < v then
            v, move ← v2, a
    return v, move
```

double-recursion: each function calls the other


MAX
MIN
A
B
C
D
a_1
a_2
a_3
b_1
b_2
b_3
c_1
c_2
c_3
d_1
d_2
d_3
3
12
8
2
4
6
14
5
2

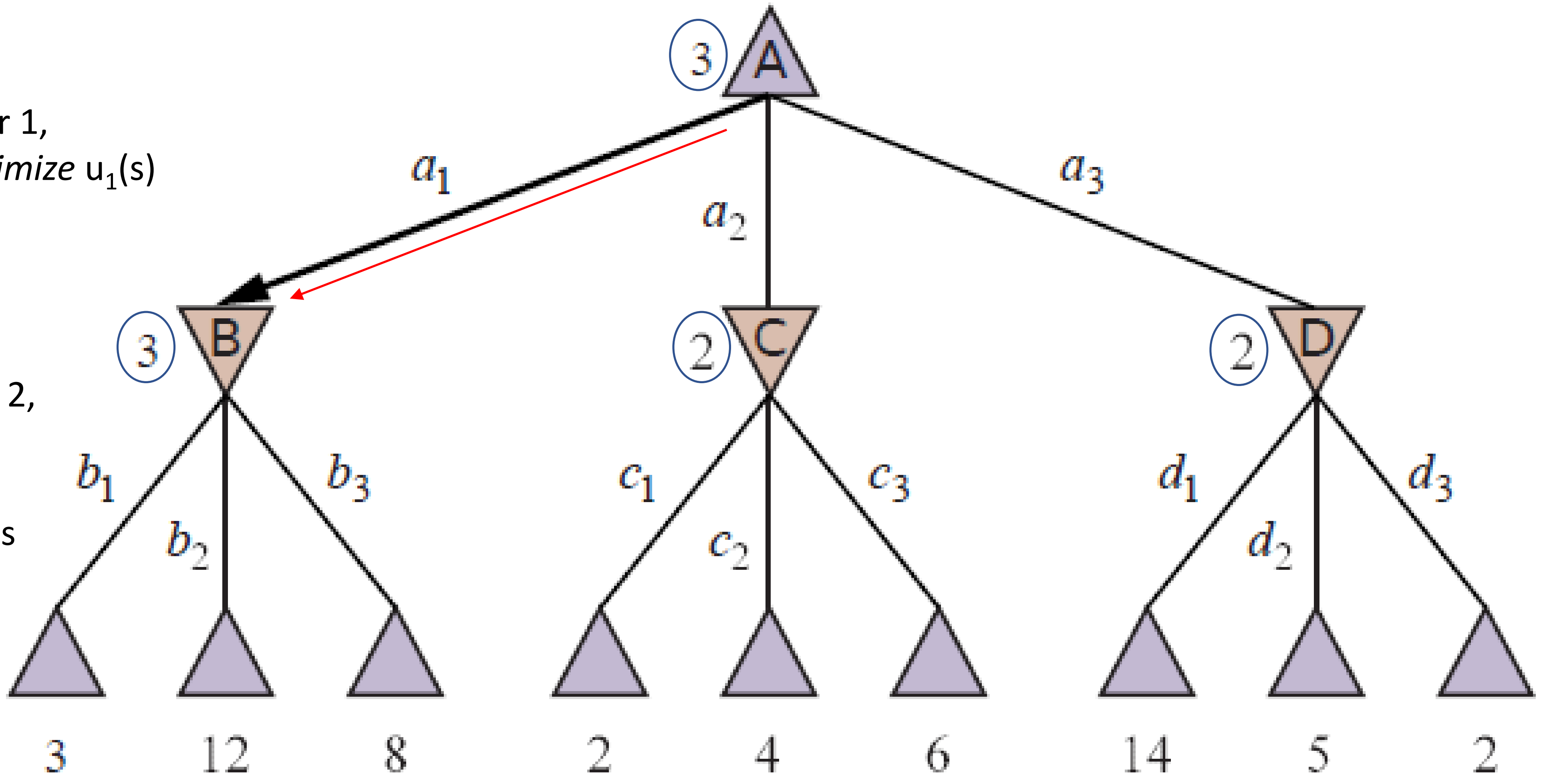

MAX
MIN
A
B
C
D
3
2
2
a1
a2
a3
b1
b2
b3
c1
c2
c3
d1
d2
d3
3
12
8
2
4
6
14
5
2

MAX

representing player 1,
who wants to *maximize* $u_1(s)$

MIN

representing player 2,
who wants to
maximize $u_2(s)$,
which is the same as
minimizing $u_1(s)$



Minimax Search

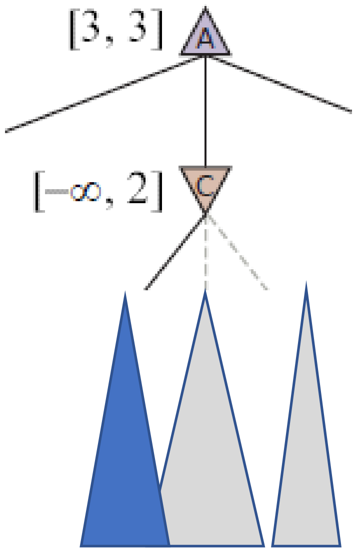
- note: this only determines next move (by player 1)
- then player 2 chooses an action
- then we have to recompute the game tree from that state to decide the next move
- minimax does not determine the entire sequence of play; you cannot force the choices of the other player
- we *assume* the opponent will make optimal choices (for them)
- what happens if they make a sub-optimal move (e.g. a mistake)?

Complexity of Game Search

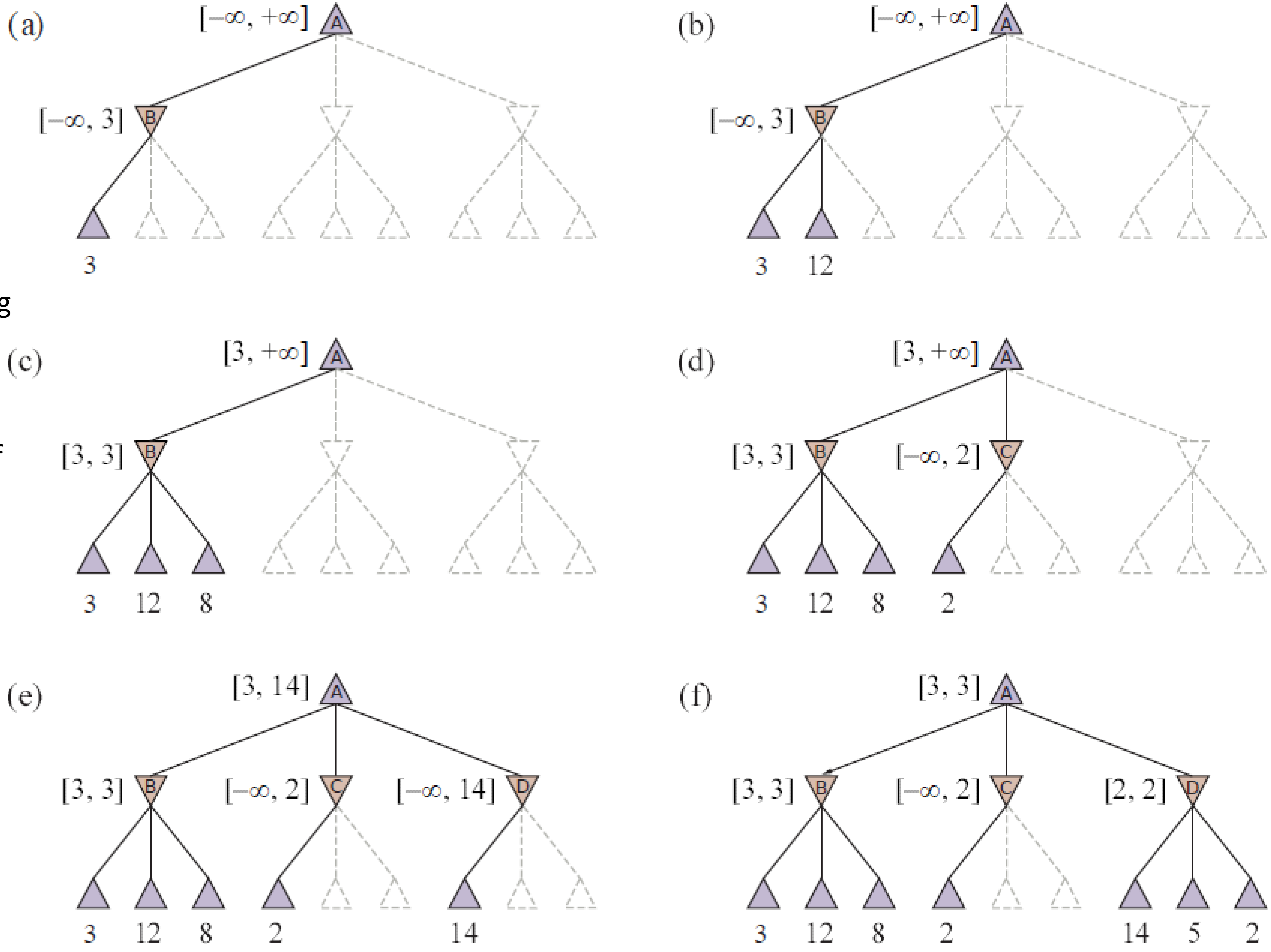
- the problem with applying Minimax to most games is that the search space is too large
  - estimates for chess: avg game=70 moves, avg branching factor=35, state space = $\sim 35^{70} = \sim 10^{108}$
  - so we can't search all the way to leaves (end-games) where utility is defined to propagate the minimax values back up
- solution 1: use intelligent *pruning* to reduce the search space
  - sometimes we can infer parts of the space that do not need to be searched

α/β-pruning

- at each node, keep track of 2 additional values α, β (along with minimax value)
- these represent the lower- and upper-bound on what minimax(s) could eventually be
- initially, set α, $\beta = [-\infty,+\infty]$
- as we process children, update these
  - at max nodes, update α: $\alpha=\max\{\alpha$, minimax(ch)$\}$ for each ch$\in$succ(s)$\}$
  - at min nodes, update β: $\beta=\min\{\beta$,minimax(ch)$\}$ for each ch$\in$succ(s)$\}$
- pruning condition: when interval of node an parent no longer overlap



(this example is for a simplified version of the alpha-beta pruning algorithm where we initialize alpha and beta to the range [-∞,∞] at every node (instead of passing them in as parameters), and the pruning condition is evaluated by checking the overlap between the range of each node and it's parent)

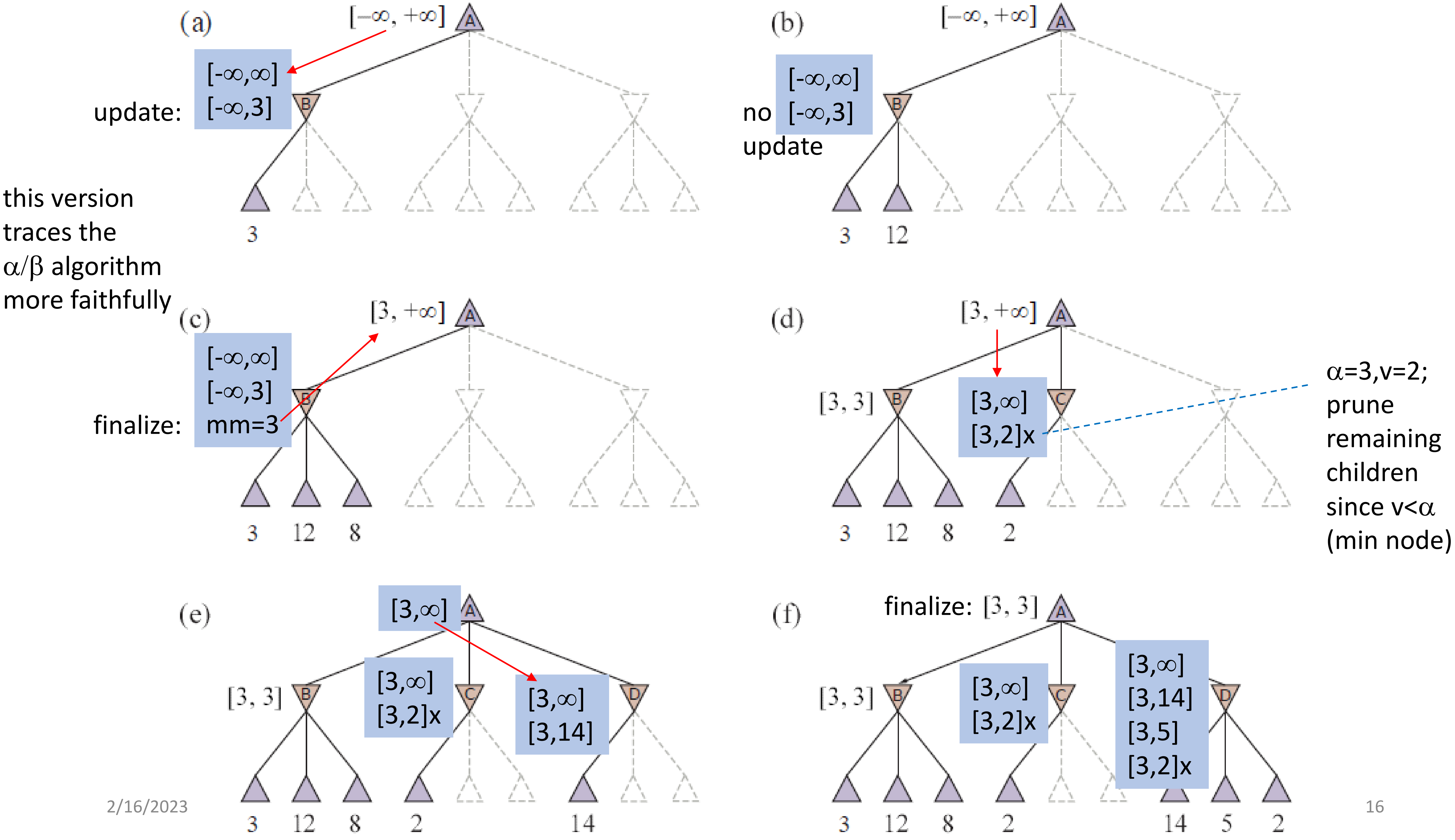


```
function ALPHA-BETA-SEARCH(game, state) returns an action
   player ← game.TO-MOVE(state)
   value, move ← MAX-VALUE(game, state, −∞, +∞)
   return move

function MAX-VALUE(game, state, α, β) returns a (utility, move) pair
   if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
   v ← −∞
   for each a in game.ACTIONS(state) do
      v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), α, β)
      if v2 > v then
         v, move ← v2, a
         α ← MAX(α, v)
      if v ≥ β then return v, move
   return v, move

function MIN-VALUE(game, state, α, β) returns a (utility, move) pair
   if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
   v ← +∞
   for each a in game.ACTIONS(state) do
      v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), α, β)
      if v2 < v then
         v, move ← v2, a
         β ← MIN(β, v)
      if v ≤ α then return v, move
   return v, move
```

max nodes update α → (boxed: if $v2 > v$ then ... $\alpha \leftarrow \text{MAX}(\alpha, v)$)

prune if score becomes greater than upper-bound of parent's interval, since parent would never choose this branch (→ if $v \geq \beta$ then return v, *move*)

min nodes update β → (boxed: if $v2 < v$ then ... $\beta \leftarrow \text{MIN}(\beta, v)$)

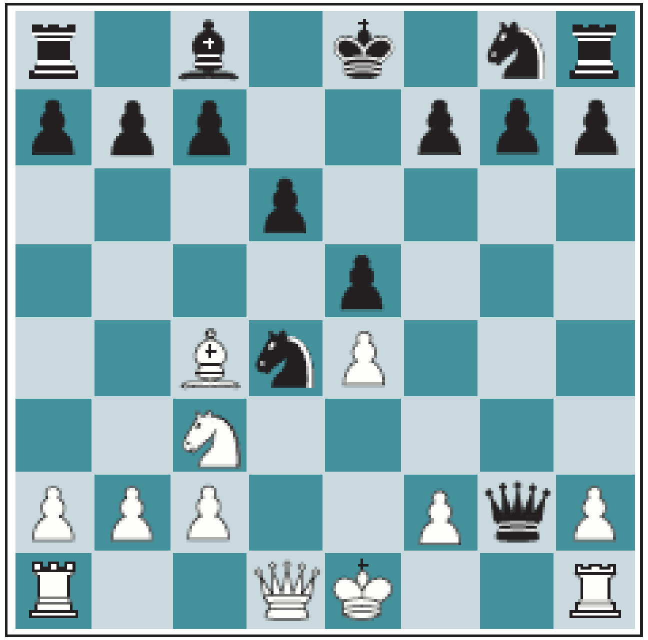

this version traces the α/β algorithm more faithfully
(a)
[−∞, +∞]
A
update:
[-∞,∞]
[-∞,3]
B
3
(b)
[−∞, +∞]
A
no update
[-∞,∞]
[-∞,3]
B
3
12
(c)
[3, +∞]
A
finalize:
[-∞,∞]
[-∞,3]
mm=3
B
3
12
8
(d)
[3, +∞]
A
[3, 3]
B
[3,∞]
[3,2]x
C
α=3,v=2; prune remaining children since v<α (min node)
3
12
8
2
(e)
[3,∞]
A
[3, 3]
B
[3,∞]
[3,2]x
C
[3,∞]
[3,14]
D
3
12
8
2
14
(f)
finalize: [3, 3]
A
[3, 3]
B
[3,∞]
[3,2]x
C
[3,∞]
[3,14]
[3,5]
[3,2]x
D
3
12
8
2
14
5
2

Complexity of Game Search

- solution 2: use a depth-limit while searching a game tree
  - need a *board-evaluation function* to assign scores to internal nodes (or non-terminal states, or non-end-games)
  - the value estimates the probability of winning or expected payoff from each state (heuristically)
  - the computer can then perform Minimax (possibly with α/β-pruning) down to a fixed level, apply the board evaluation function, and propagate values upward
  - choose depth limit based on time available (and CPU speed)
    - expressed as number of “ply” (moves, or levels)
    - 2-6 ply (a few sec): rudimentary chess performance (amateur skill level)
    - 6-10 ply (a few min): much better moves due to deeper search/look-ahead

Board Evaluation Functions

- a board evaluation function must guess the value (probable outcome) of each state
- they are typically based on *features*
- examples from chess:
  - piece differential (#PlayerPieces - #OpponentPieces)
  - material value (pawn=1, knight/bishop=3, rook=5, queen=9)
  - center control
  - # of pieces threatened or constrained
  - patterns or special arrangements of pieces

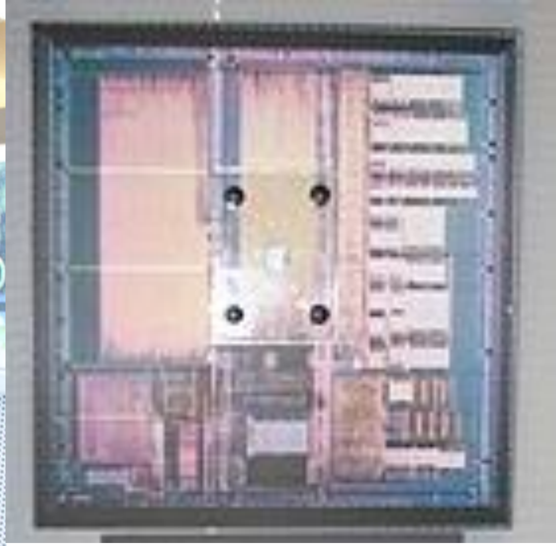


$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

Board Evaluation Functions

- problems with using board evaluation functions
  - non-quiescence
    - board evaluation function should only be applied to quiescent states, where the value has stopped changing (i.e. “converged”)
    - if there have been large changes in value, extend the search to allow it to quiesce
    - rather than enforcing a strict depth limit, can be non-uniform
    - use a dynamic IS-CUTOFF(s) test
  - horizon effect
    - sometimes, enough dodging moves can be made to forestall a bad outcome so it occurs just beyond the depth limit (like moving a bishop back and forth to delay capture, or repeatedly checking the opponent’s king)
    - delaying the inevitable – it might change our decision if we knew this
    - hard to detect and mitigate

DeepBlue



(images from Wikipedia)

- developed by IBM
- achieved grandmaster rating in 1990's
- defeated Gary Kasparov in 1997
- a supercomputer with **custom ASICs** for very fast α/**β-Minimax search**
  - 30-node IBM RS/6000 SP computer; 120 MHz and 1GB per proc.
  - 16 "chess chips" on each node, for generating moves and computing a board evaluation function
  - explored ~100 million moves/s, down to 10-12 ply (though non-uniform)
- included an **end-game database** (for example, once there are only 5 pieces left, lookup optimal moves in a pre-computed table)
- What did we learn about Intelligence?

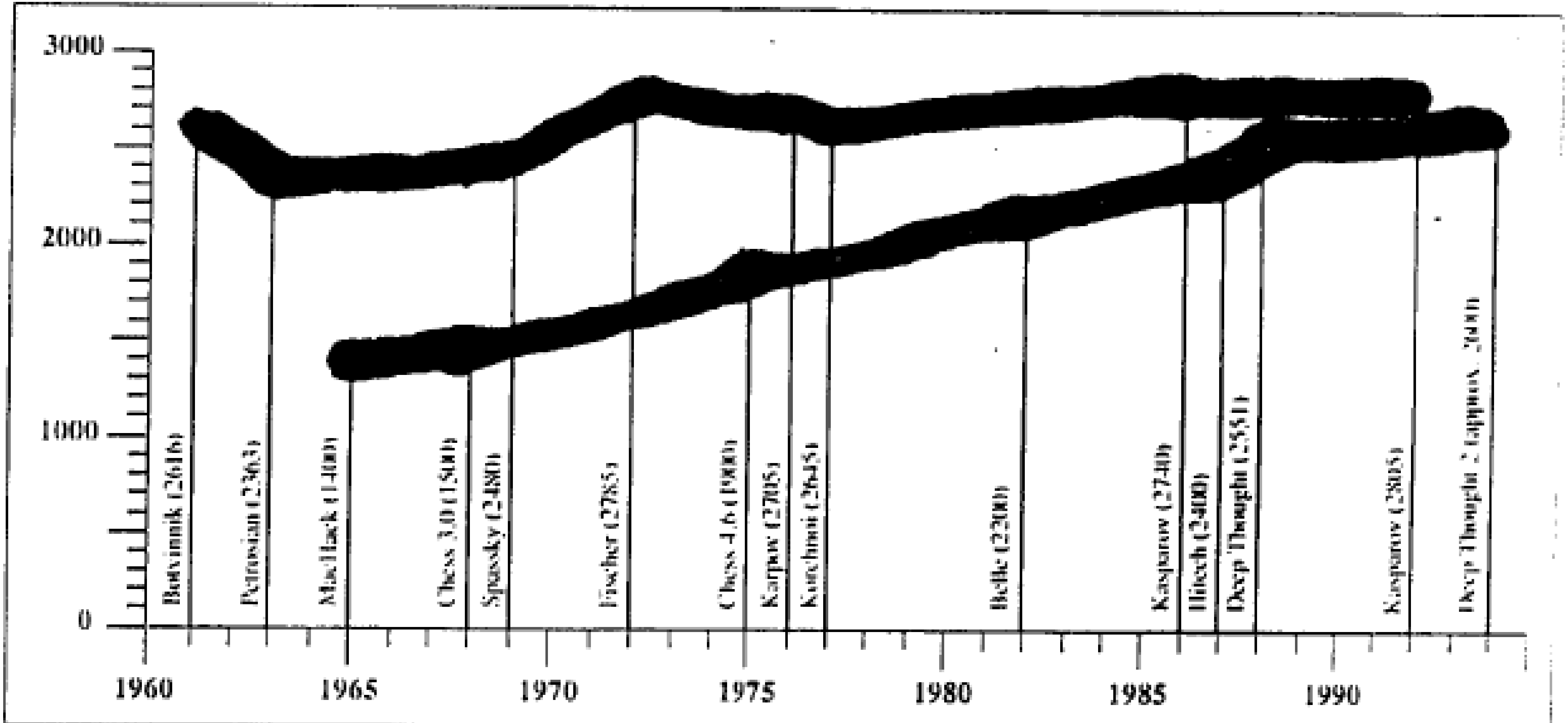
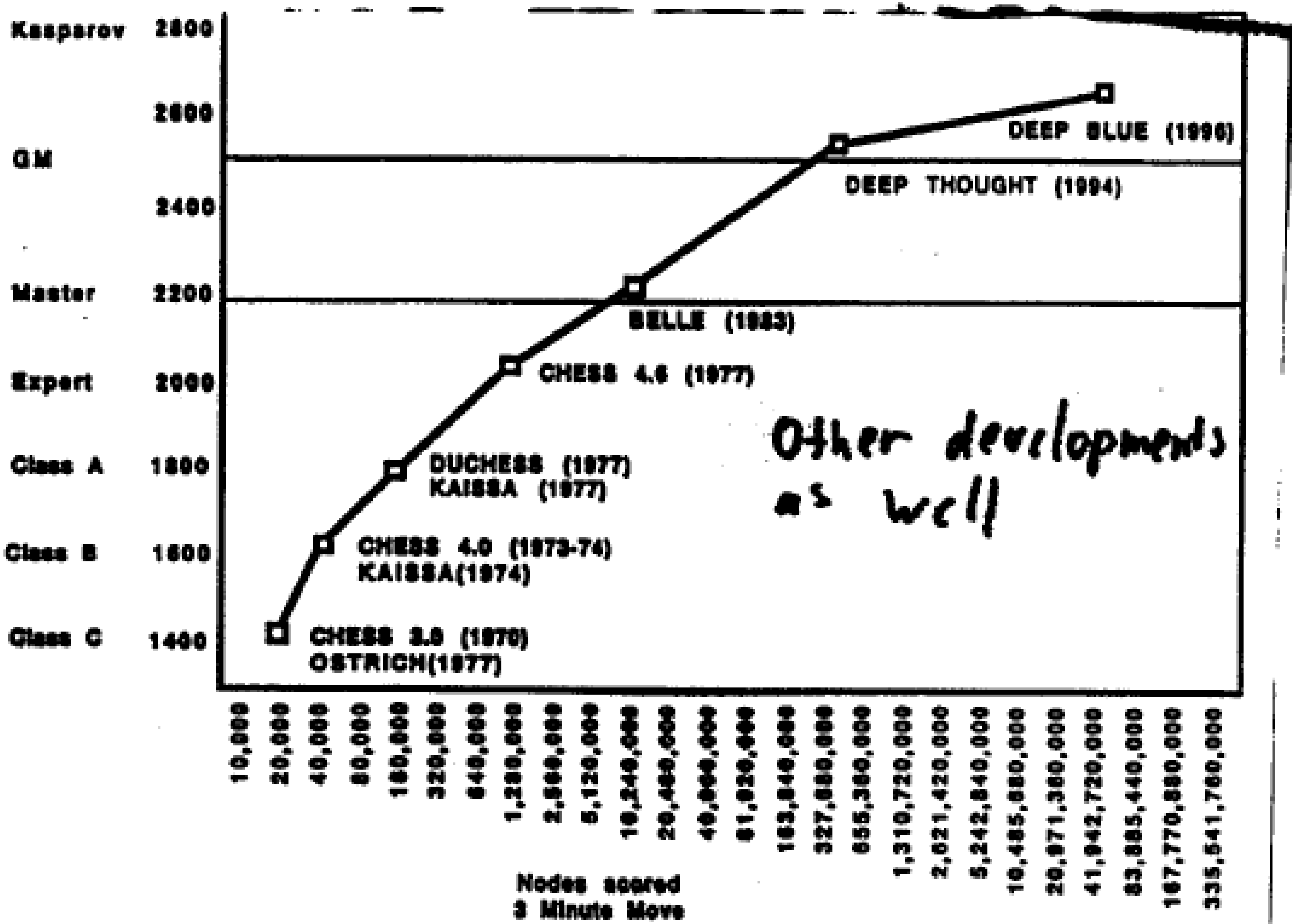


Figure 5.12 Ratings of human and machine chess champions.


Kasparov 2800
2600
GM
2400
Master 2200
Expert 2000
Class A 1800
Class B 1600
Class C 1400
DEEP BLUE (1996)
DEEP THOUGHT (1994)
BELLE (1983)
CHESS 4.6 (1977)
DUCHESS (1977)
KAISSA (1977)
CHESS 4.0 (1973-74)
KAISSA(1974)
CHESS 3.0 (1970)
OSTRICH(1977)
Other developments as well
10,000
20,000
40,000
80,000
160,000
320,000
640,000
1,280,000
2,560,000
5,120,000
10,240,000
20,480,000
40,960,000
81,920,000
163,840,000
327,680,000
655,360,000
1,310,720,000
2,621,420,000
5,242,840,000
10,485,680,000
20,971,360,000
41,942,720,000
83,885,440,000
167,770,880,000
335,541,760,000
Nodes scored
3 Minute Move

Connect4

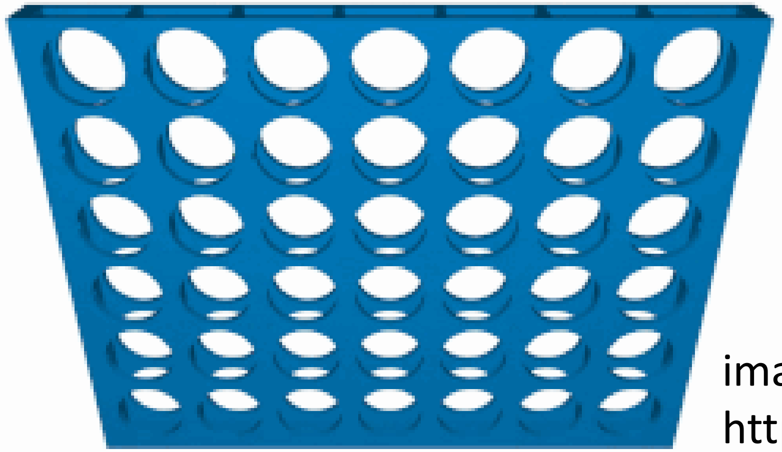
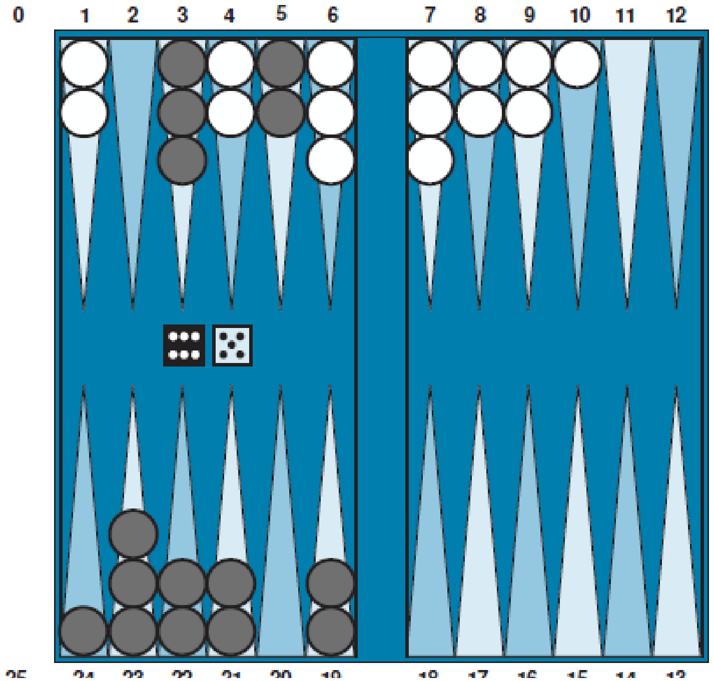


image obtained from
https://en.wikipedia.org/wiki/Connect_Four

- pieces are dropped in vertical columns; 4-in-a-row wins the game
  - here is an online app you can play around with: https://www.cbc.ca/kids/games/all/connect-4
- Challenge: <u>Can you come up with a board evaluation function for playing Connect4</u>?
  - it would not be hard to implement this on the command line (similar to tic-tac-toe)
  - the State Space is **much** larger, so you would have to use a depth cutoff in the Minimax search and apply a board evaluation function to incomplete states
  - (try pausing the animation above and estimating the value of the state)

```
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . x . . . .
. . x o o . .
-------------
```

a famous backgammon program called TDgammon
(by Gary Tesauro) used Reinforcement Learning

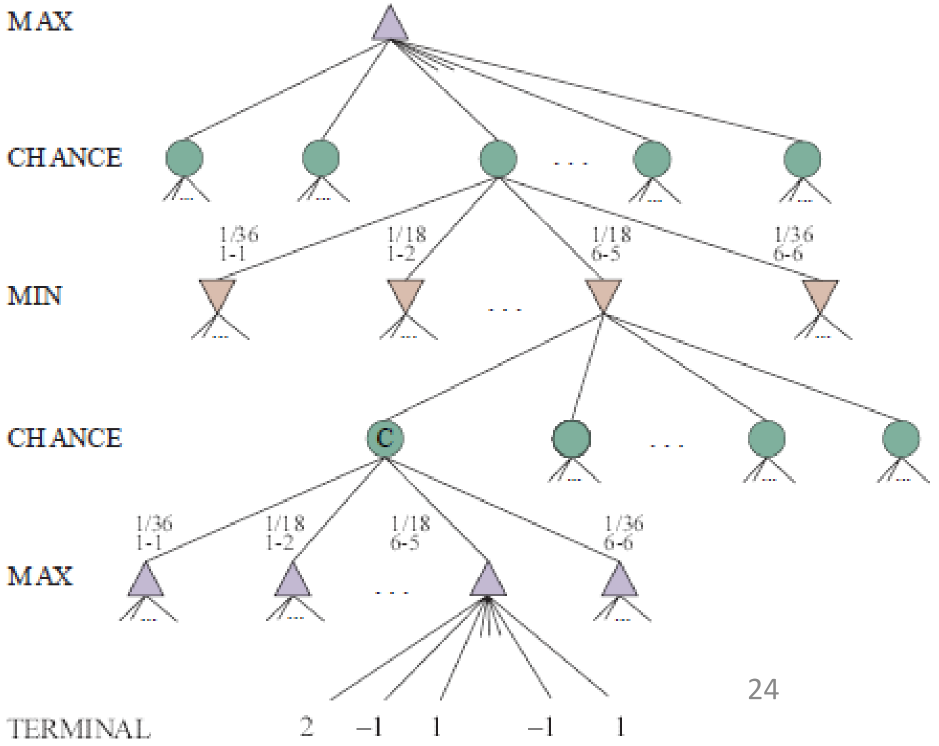


Expectiminimax

- stochastic games – games with an element of chance (e.g. dice, cards...)
  - examples: backgammon, yahtze...
- can we apply minimax search?
  - yes, if we interleave min and max nodes with a level of *chance nodes*
  - at chance nodes, the score is the weighted sum over the children, weighted by probability, i.e. "expected outcome"

Expectiminimax(s) =

$$
\begin{cases}
u_1(s) \text{ if is a terminal node} \\
\max\{\textit{Expectiminimax}(s') \mid s' \in succ(s)\} \text{ if max node} \\
\min\{\textit{Expectiminimax}(s') \mid s' \in succ(s)\} \text{ if min node} \\
\Sigma_{s' \in succ(s)} P(s') \cdot \textit{Expectiminimax}(s') \text{ if chance node}
\end{cases}
$$



Monte Carlo Tree Search (MCTS) (Sec 5.4)

- instead of exhaustively exploring search tree, sample random paths (“rollouts”) all the way to terminal states (end-games with defined utility)
- the value of a state is taken as the statistical *average* outcome of trajectories passing through it (“back-propagate” outcomes)
- also keep track of *n* (# trial trajectories passing through each node) and variance (σ^2) at each state to assess certainty

function MONTE-CARLO-TREE-SEARCH(*state*) **returns** *an action*
  tree ← NODE(*state*)
  while IS-TIME-REMAINING() **do**
    leaf ← SELECT(*tree*)
    child ← EXPAND(*leaf*)
    result ← SIMULATE(*child*)
    BACK-PROPAGATE(*result*, *child*)
  return the move in ACTIONS(*state*) whose node has highest number of playouts

Monte Carlo Tree Search (MCTS)

- there are many choices about how to make move during simulation
- selection policy – which states to start simulation from?
  - expansion vs. exploration
  - is it better to refine value estimate at good nodes, or increase certainty of bad nodes?
  - allow occasional sub-optimal choices for the sake of seeing how they turn out
- playout policy
  - just making subsequent random moves is not realistic
  - it helps to define an "approximate strategy" to simulate reasonable moves by both players

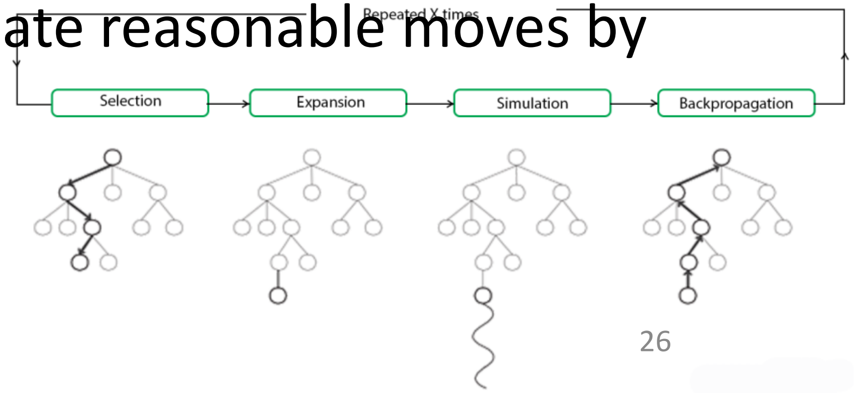


image from https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/

Monte Carlo Tree Search (MCTS)

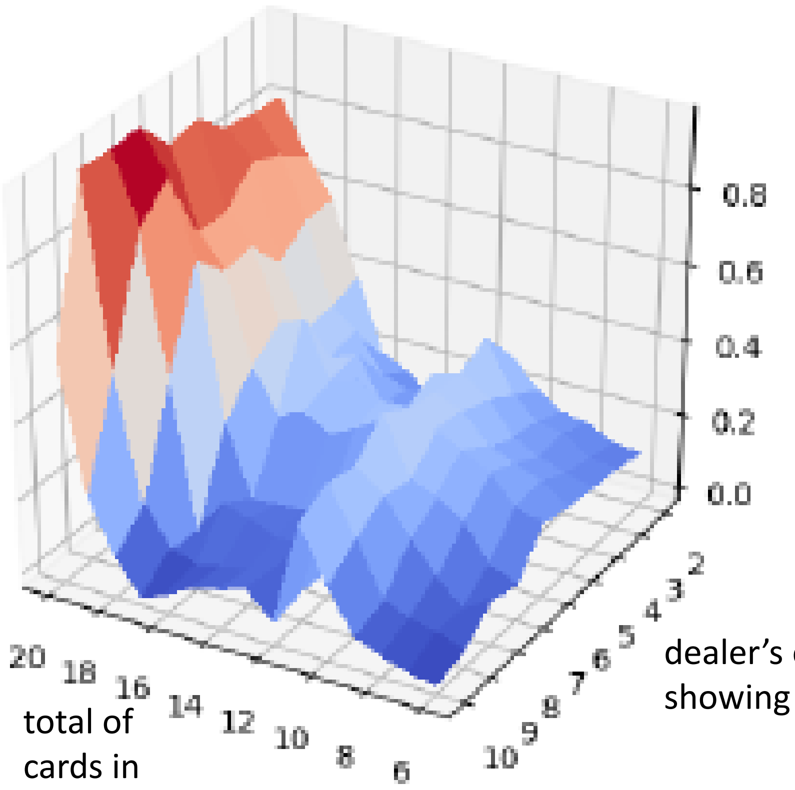
- using MCTS to learn strategy for Blackjack
  - simulate >10,000 random games to learn policy

dealer's card showing

total of cards in hand

	2	3	4	5	6	7	8	9	10	A
17+	ST	ST	ST	ST	ST	ST	ST	ST	ST	ST
16	ST	ST	ST	ST	ST	H	H	H	H	H
15	ST	ST	ST	ST	ST	H	H	H	H	H
14	ST	ST	ST	ST	ST	H	H	H	H	H
13	ST	ST	ST	ST	ST	H	H	H	H	H
12	H	H	ST	ST	ST	H	H	H	H	H
11	D	D	D	D	D	D	D	D	D	H
10	D	D	D	D	D	D	D	D	H	H
9	H	D	D	D	H	H	H	H	H	H
5-8	H	H	H	H	H	H	H	H	H	H

H=hit
ST=stand
D=double-down



expected prob. of winning for Hit, based on mean of rollouts

total of cards in hand

dealer's card showing

AlphaGO

image from
https://en.wikipedia.org/wiki/Go_(game)

- GO is played with b/w stones on a 19x19 board
  - search space much larger than chess (bran. fact. starts at 361)
- from Google DeepMind, 2017
- after decades of attempts by other AI programs, AlphaGO finally beat the human GO world champion
- learns from *self-play* (bootstrapping), >100,000 games
- trains a *deep neural network* (14 conv. layers) to represent a value function (reinforcement learning, MCTS)
- reached grandmaster rating after 21 days (176 GPUs)

Games with Imperfect Information

- in games like tic-tac-toe and chess, all information about state is available to both players (i.e. on the board)
- in some games, some information might be private to individuals
- examples: card games like hearts, bridge, poker...
- (note: we are talking about the cards dealt and in the hands of others
  - we are not talking about stochasticity of which cards will be drawn next)
- the optimal move often depends on information we don't have access to
- representative of "partially observable" environments

images from https://en.wikipedia.org/wiki/Contract_bridge

Games with Imperfect Information

- simple view: payoff of actions is averaged over all possible opponent hands (probability distribution)
- in some cases, we can infer what opponents hold by their actions
- there are sophisticated AI methods (POMDPs) for estimating and reasoning over “belief states”
- interesting effect: in some cases, there is value in taking actions with a cost, primarily for gaining information
  - such as a real-estate developer paying for geological survey, because it will help them better decide how to develop a property and estimate it’s potential value as a hotel vs. mall vs. warehouse vs. drilling site
  - in partially observable environments, there is value in information
- GIB - famous bridge-playing program (c.a. 1999) - uses Monte Carlo
  - bidding phase is still challenging – communication relying on social conventions