Game Search

CSCE 420 – Fall 2023

read: Ch. 5

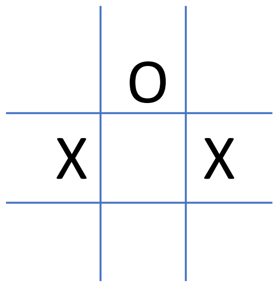
Game Search

- 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

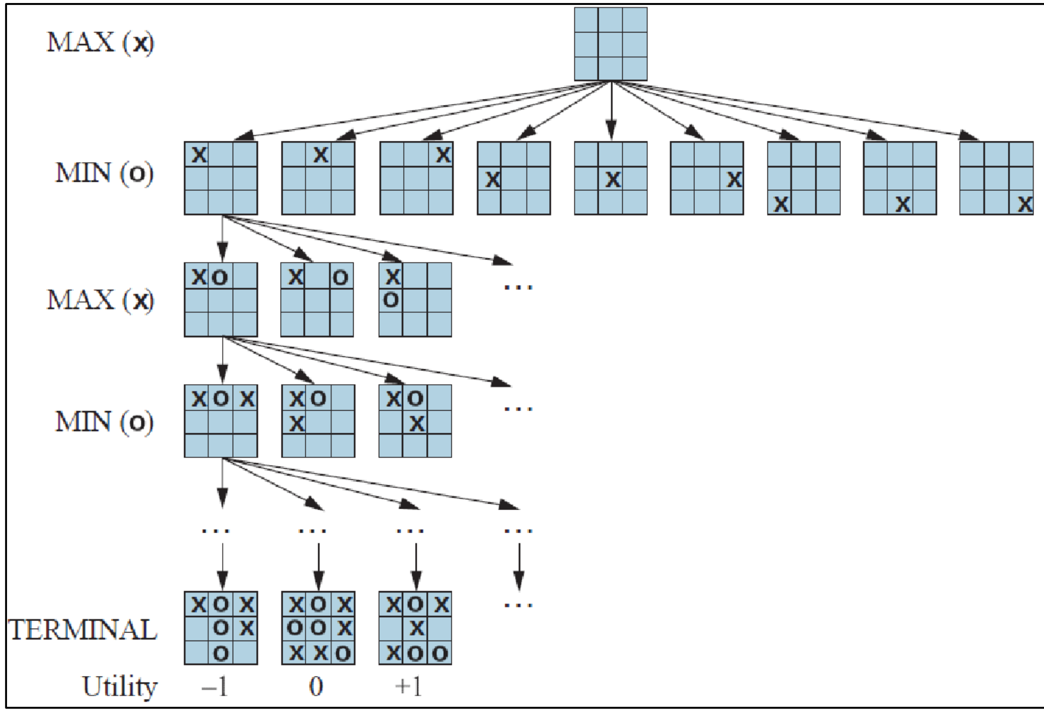
- multiple steps – players take turns
- each player has a utility function
  - $u_i(s)$ (where *i* is the player, and *s* is a game state)
  - +1 for win; -1 for lose; 0 for draw (tic-tac-toe); 0 for non-terminal states
  - money (poker)
  - rewards for achieving goals - cost of actions or resources 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 <u>max nodes</u> and <u>min nodes</u>
- 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'∈ succ(s) }
  - i.e. choose the action that leads to the successor with highest score, which has the highest expected payoff

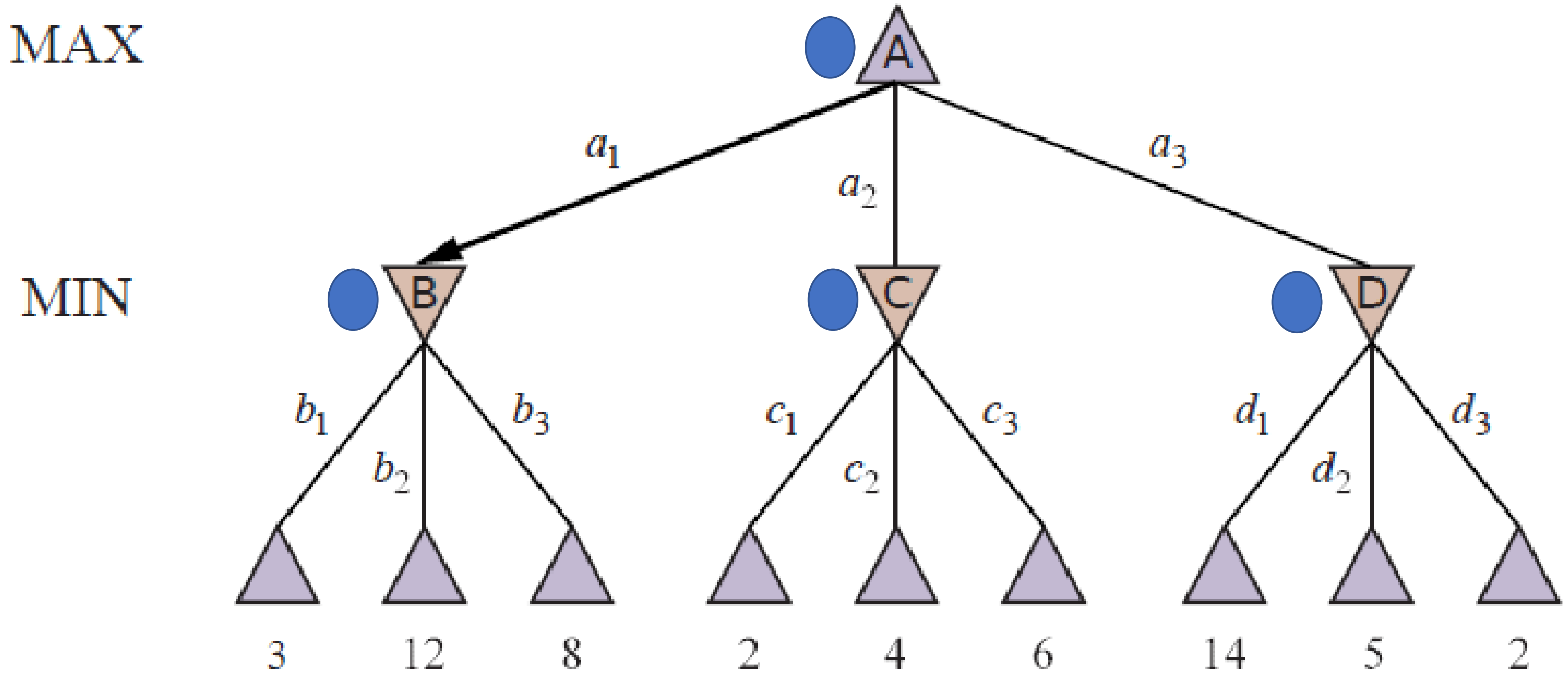
Minimax Search

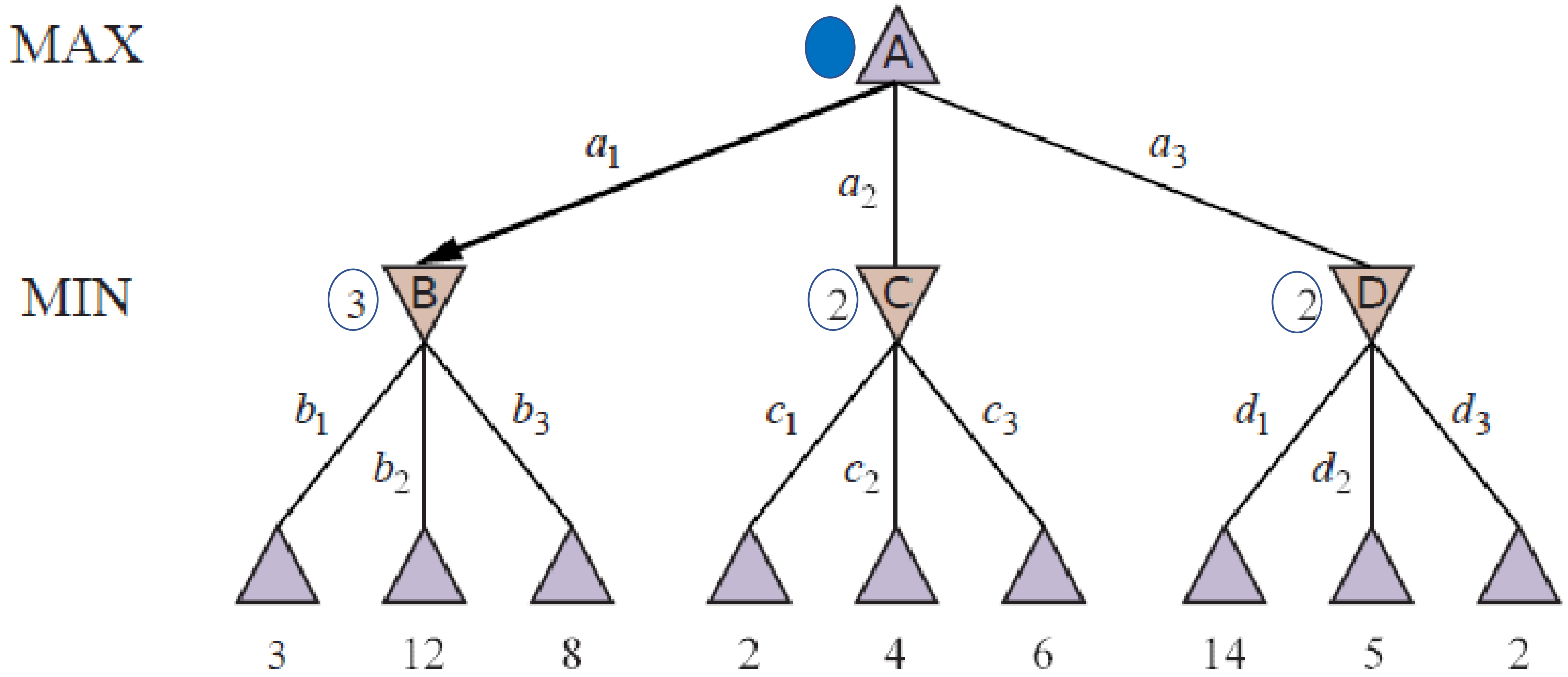
```
function MINIMAX-SEARCH(game, state) returns an action
   player ← game.TO-MOVE(state)
   value, move ← MAX-VALUE(game, state)
   return move
```

```
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

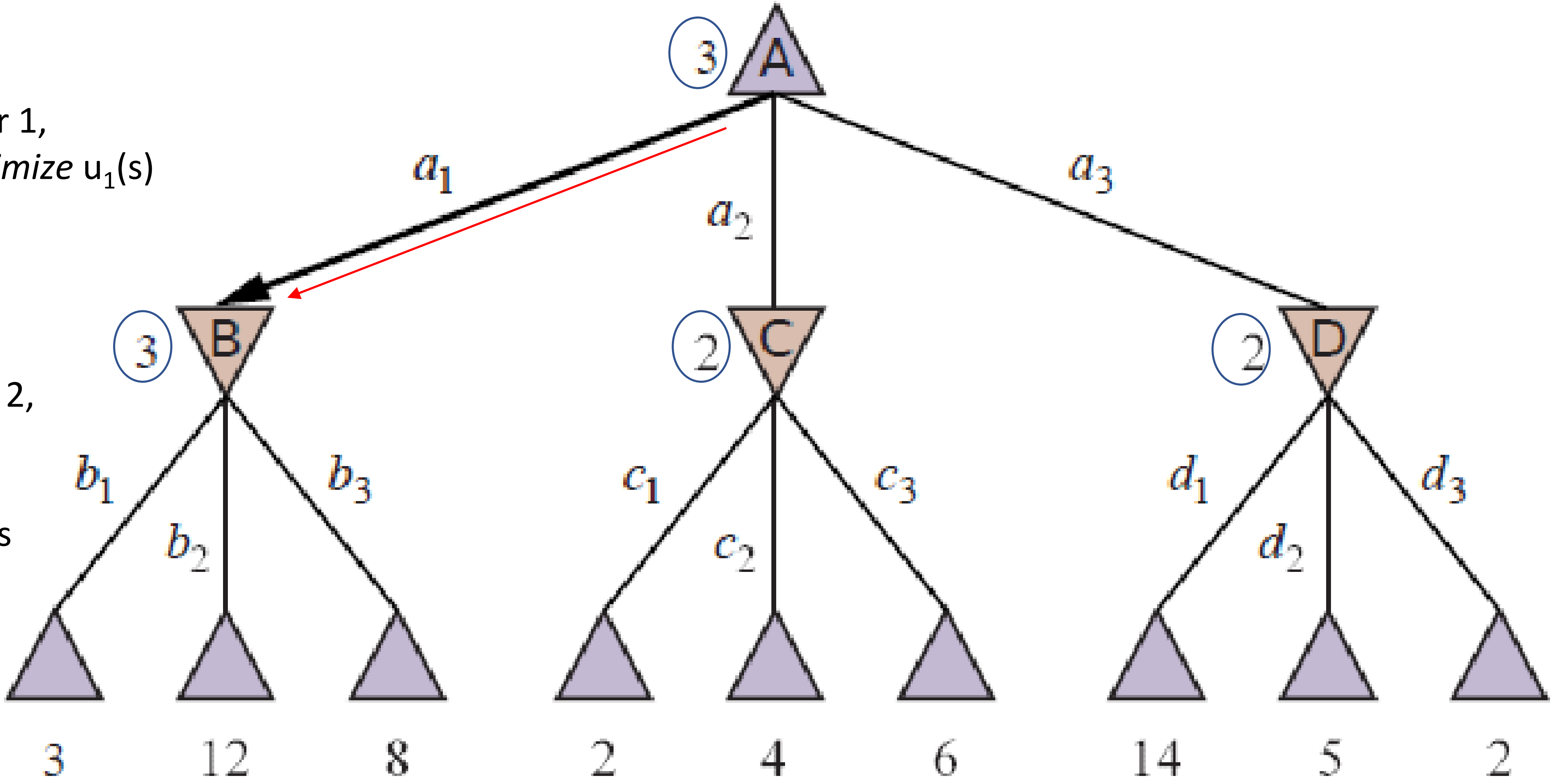

MAX
MIN
A
B
C
D
3
2
2
a_1
a_2
a_3
b_1
b_2
b_3
c_1
c_2
c_3
d_1
d_2
d_3
3
12
8
2
4
6
14
5
2

MAX

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

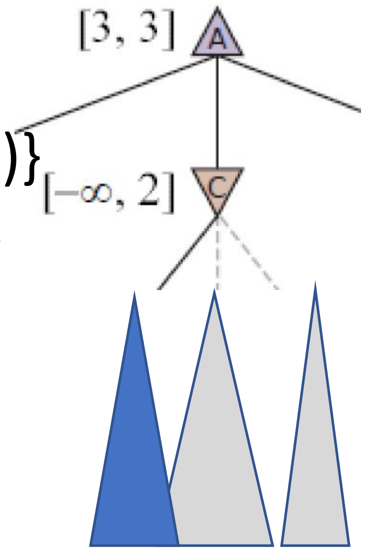
- note: this only determines next move (by player 1)
- then player 2 chooses an action
- then we have to recompute the game tree from that state to decide the next move
- minimax does not determine the entire sequence of play; you cannot force the choices of the other player
- we *assume* the opponent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

Complexity of Game Search

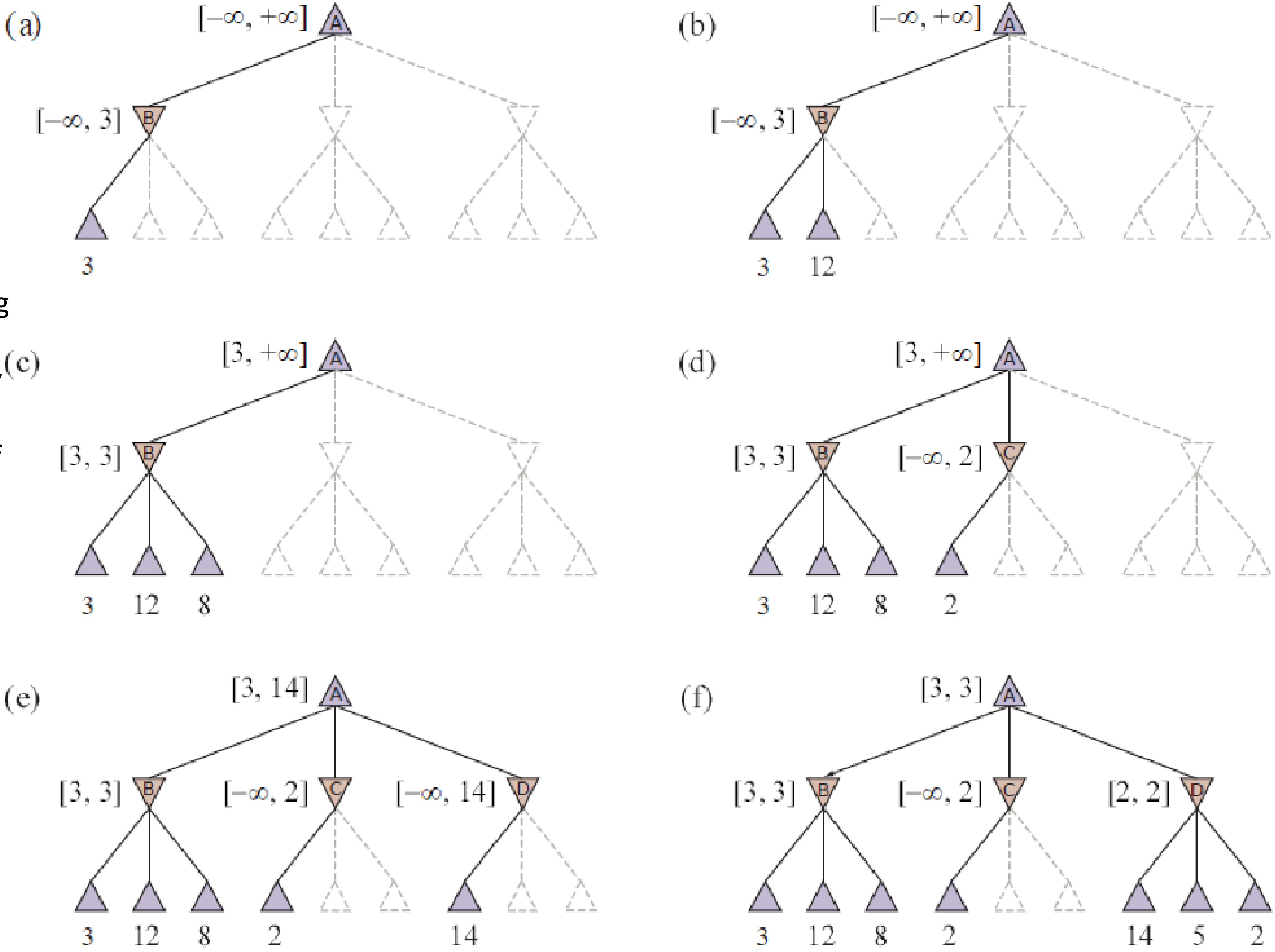
- the problem with applying Minimax to most games is that the search space is too large
  - estimates for chess: avg game=70 moves, avg branching factor=35, state space = $\sim 35^{70} = \sim 10^{108}$
  - so we can’t search all the way to leaves (end-games) where utility is defined to propagate the 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)
  - α is the best possible value for any max node above so far (initially $-\infty$)
  - β is the best possible value for any min node above so far (initially $+\infty$)
- as we process children, update these params
  - at max nodes, update α: $\alpha=\max\{\alpha, \text{minimax}(s')\}$ for each $s' \in \text{children}(s)$}
  - at min nodes, update β: $\beta=\min\{\beta,\text{minimax}(s')\}$ for each $s' \in \text{children}(s)$}
- pruning condition:
  - at min nodes: when $v<\alpha$ (i.e. best choice of parent max node)
  - at max nodes: when $v>\beta$ (i.e. best choice of parent min node)
  - equivalently: when interval of v at node no longer overlaps interval of parent



(this example is for a simplified version of the alpha-beta pruning algorithm where we initialize minimax value v to the range [-∞,∞] at every node (instead of passing α and β 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*, $-\infty$, $+\infty$)
  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*
  $v \leftarrow -\infty$
  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*
      $\alpha \leftarrow$ MAX(α, v)
    if $v \geq \beta$ **then return** *v*, *move*
  return *v*, *move*

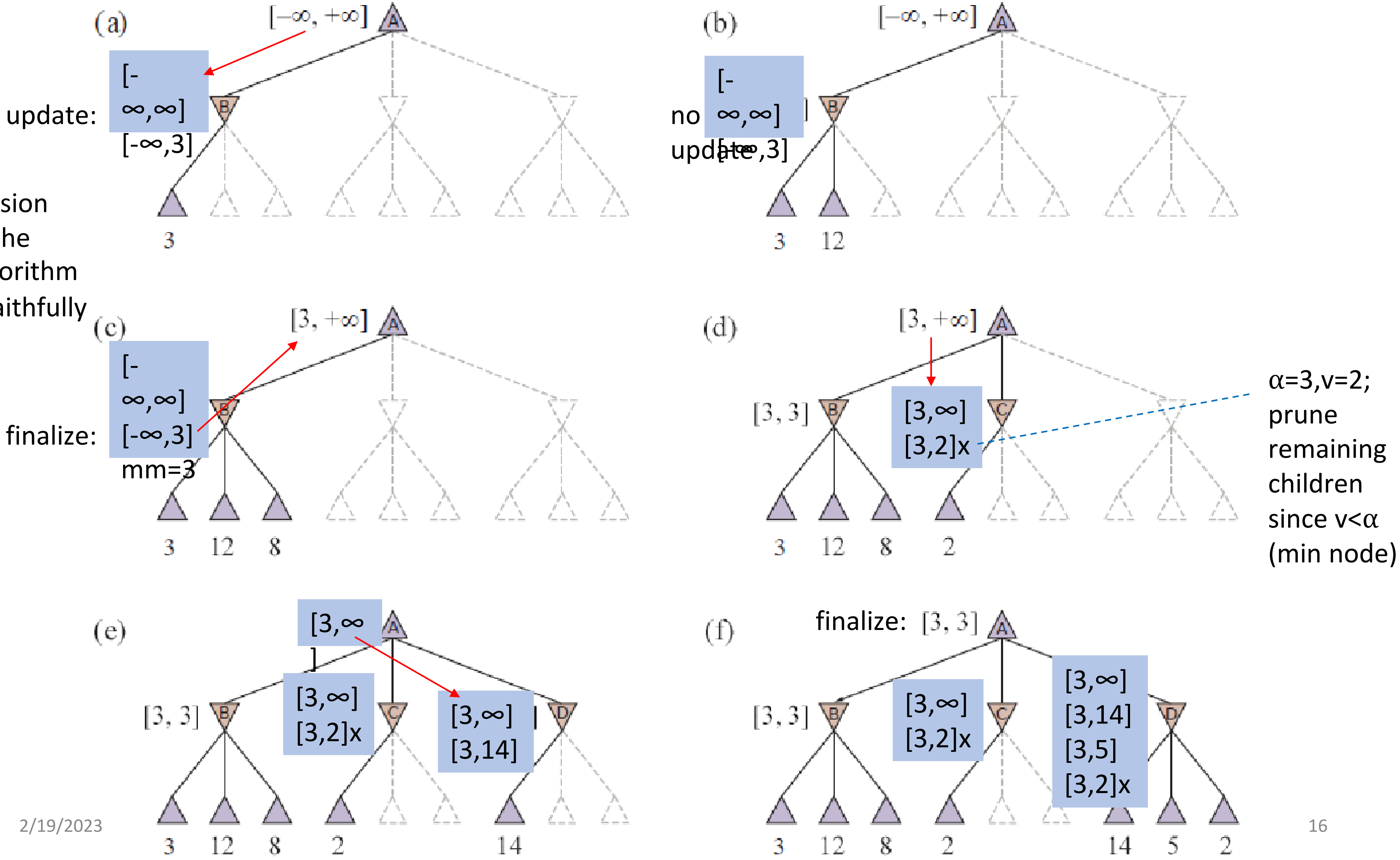
max nodes update α

prune if score becomes greater than upper-bound of parent's interval, since parent would never choose this branch

function MIN-VALUE(*game*, *state*, α, β) **returns** a (*utility*, *move*) pair
  if *game*.IS-TERMINAL(*state*) **then return** *game*.UTILITY(*state*, *player*), *null*
  $v \leftarrow +\infty$
  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*
      $\beta \leftarrow$ MIN(β, v)
    if $v \leq \alpha$ **then return** *v*, *move*
  return *v*, *move*

min nodes update β

this version traces the α/β algorithm more faithfully

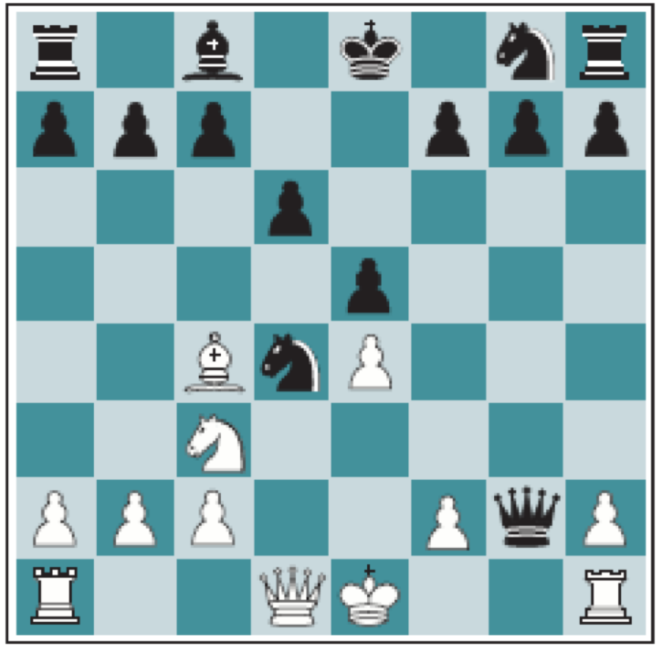


Complexity of Game Search

- 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 <u>probability of winning</u> or <u>expected payoff</u> 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

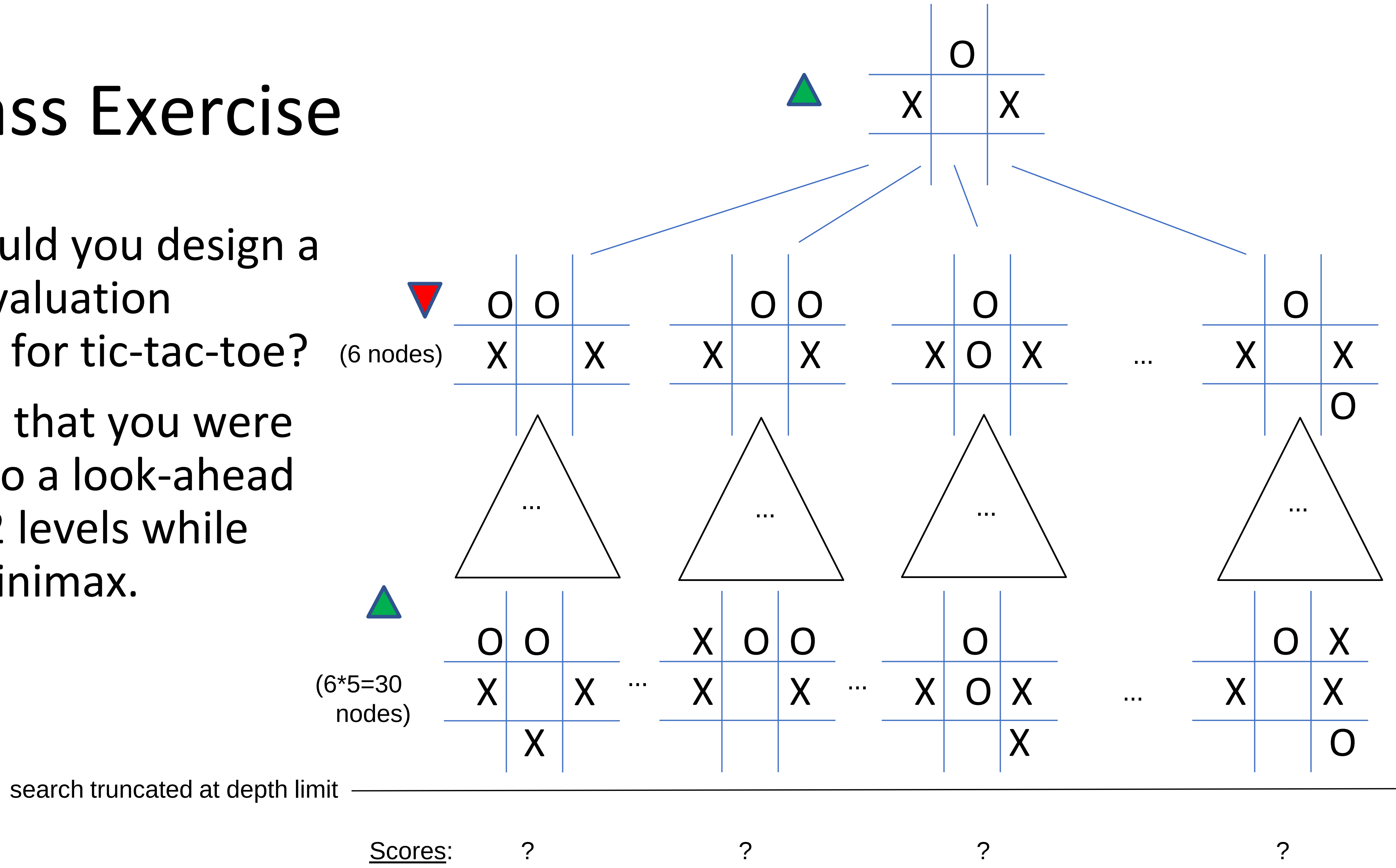
- a board evaluation function must guess the value (probable outcome) of each state
- they are typically based on *features*
- examples from chess:
  - piece differential (#PlayerPieces - #OpponentPieces)
  - material value (pawn=1, knight/bishop=3, rook=5, queen=9)
  - center control
  - # of pieces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



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

In-class Exercise

- How would you design a board evaluation function for tic-tac-toe?
- Suppose that you were limited to a look-ahead of only 2 levels while doing minimax.



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?

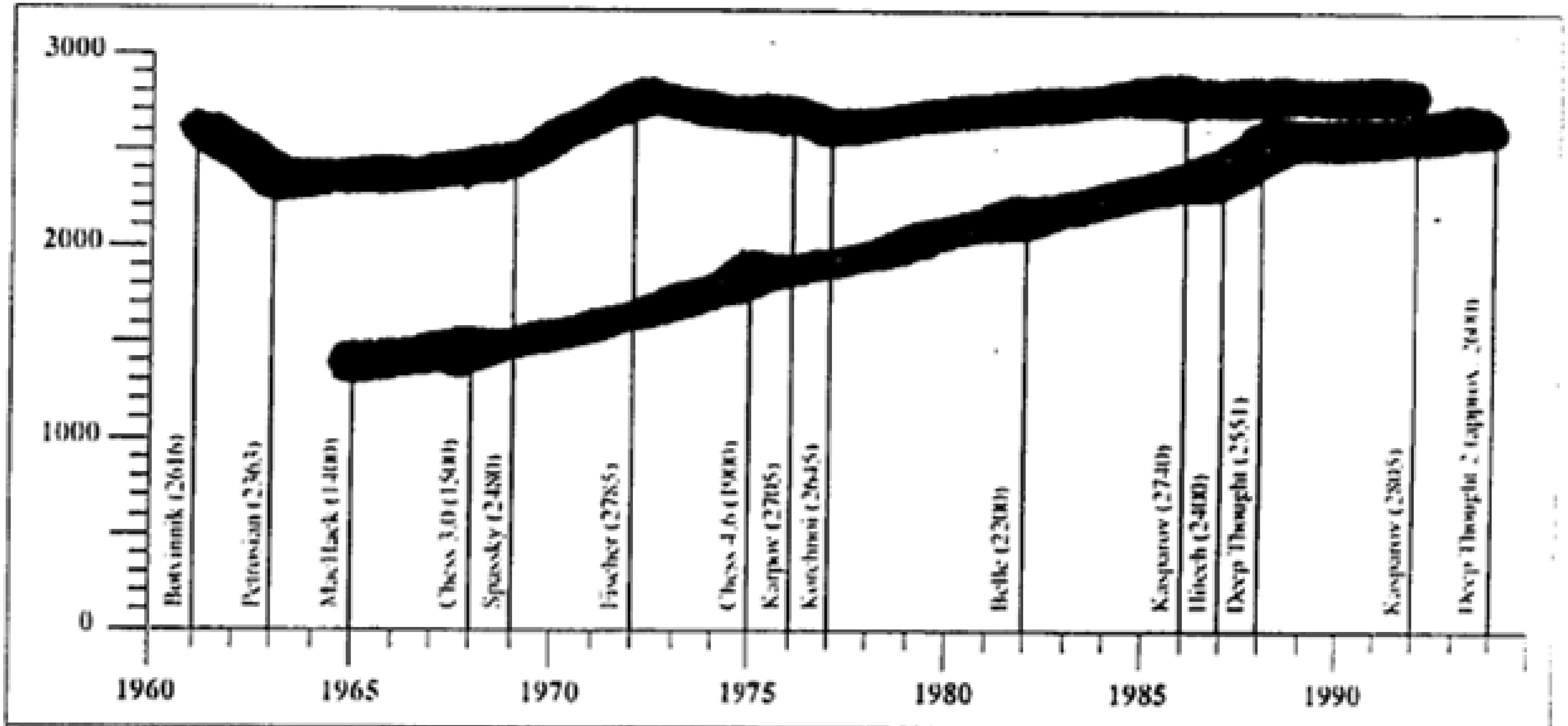
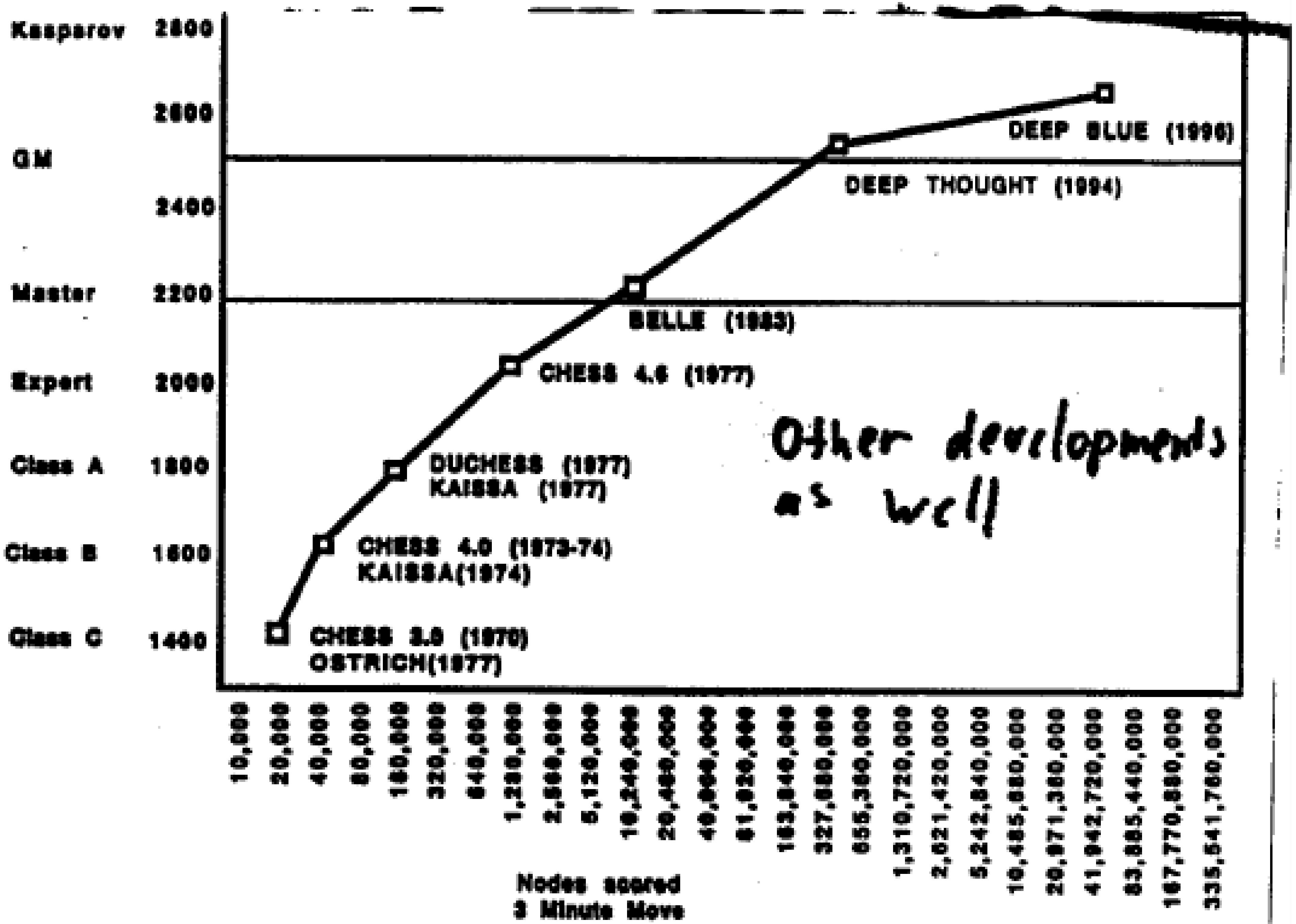


Figure 5.12 Ratings of human and machine chess champions.



Other developments as well

Connect4

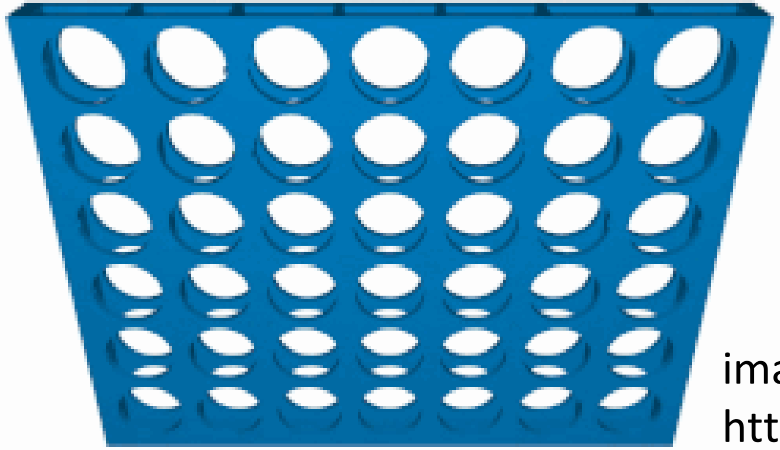
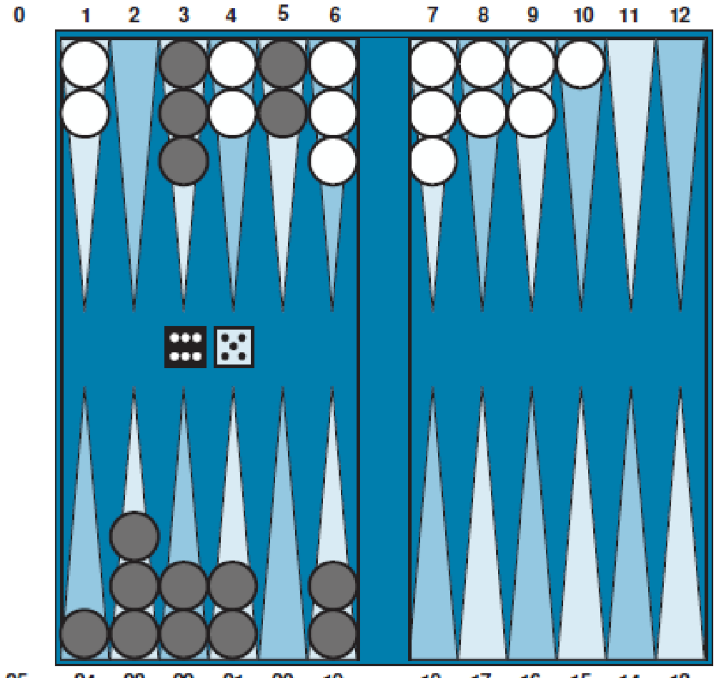


image obtained from
https://en.wikipedia.org/wiki/Connect_Four

- 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)

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

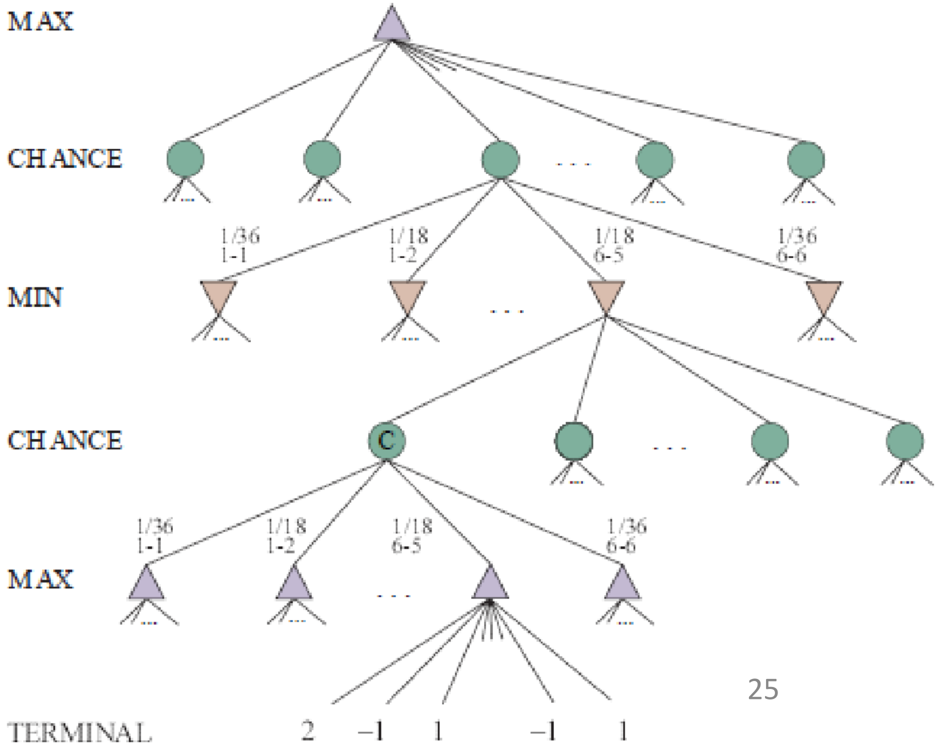
a famous backgammon program called TDgammon (by Gary Tesauro) used Reinforcement Learning



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"

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



Monte Carlo Tree Search (MCTS) (Sec 5.4)

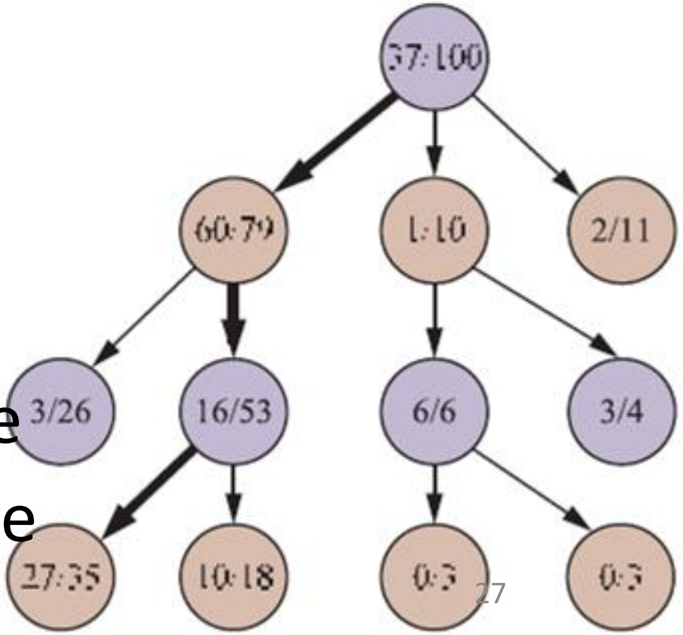
- instead of exhaustively exploring search tree, sample random paths (“rollouts”) all the way to terminal states (end-games with defined utility)
- the value of a state is taken as the statistical *average* outcome of trajectories passing through it (“back-propagate” outcomes)
- also keep track of *n*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

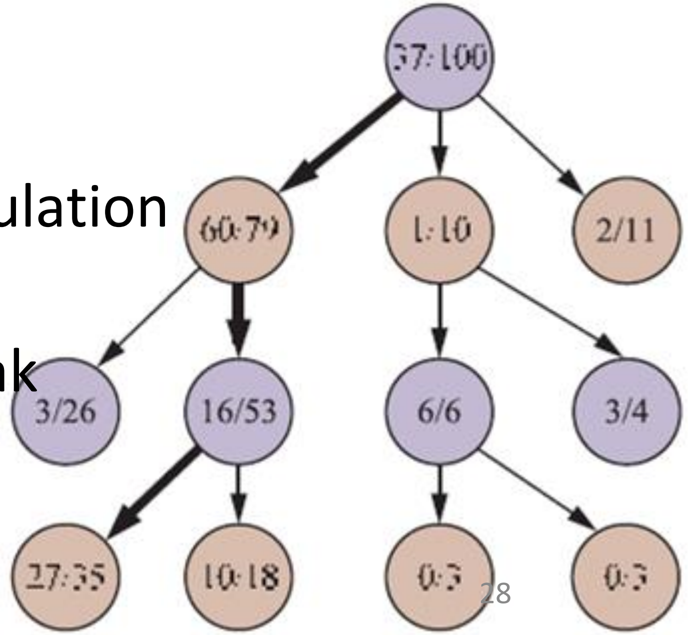
- think of MCTS as an alternative to manually creating a board evaluation function
- estimate quality of each state (prob of winning) by simulating random game trajectories (playouts)
- at each node, keep track of how many times it led to a win; more trajectories provide higher confidence
- can use these values to select children in minimax search

- select a node (game state) whose value is uncertain
- run simulation: play game to see outcome from that state
- back-propagation: update nodes along path with outcome



Monte Carlo Tree Search (MCTS)

- selection policy – which states could use more sampling?
  - 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
  - there are many choices about how to make moves during simulation
  - just making subsequent random moves is not realistic
  - it helps to define an initial strategy to play against, even if weak



Monte Carlo Tree Search (MCTS)

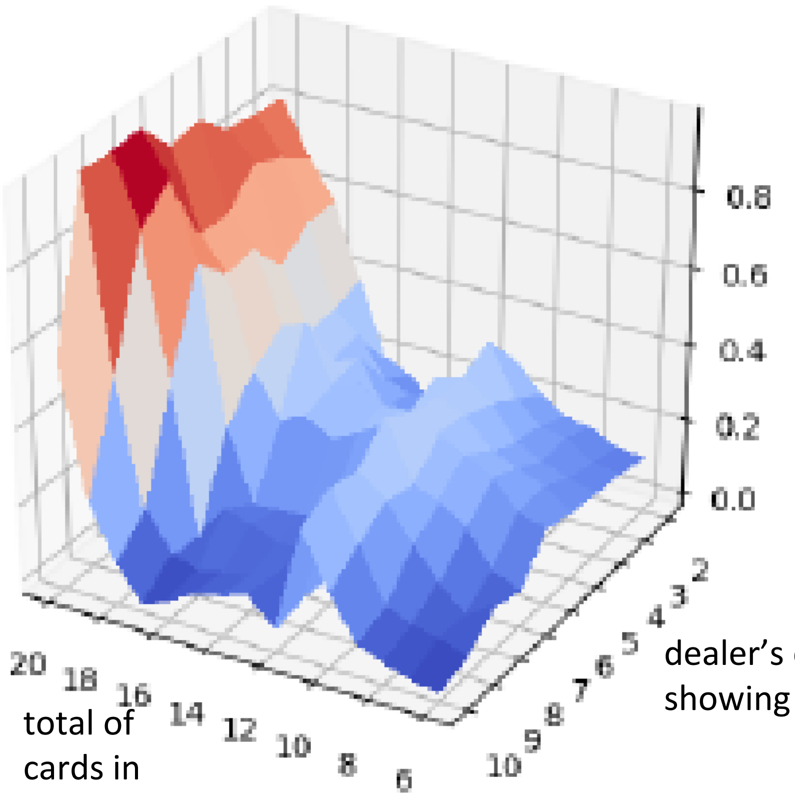
- using MCTS to learn strategy for Blackjack
  - simulate >10,000 random games to learn policy

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

dealer's card showing

total of cards in hand

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)