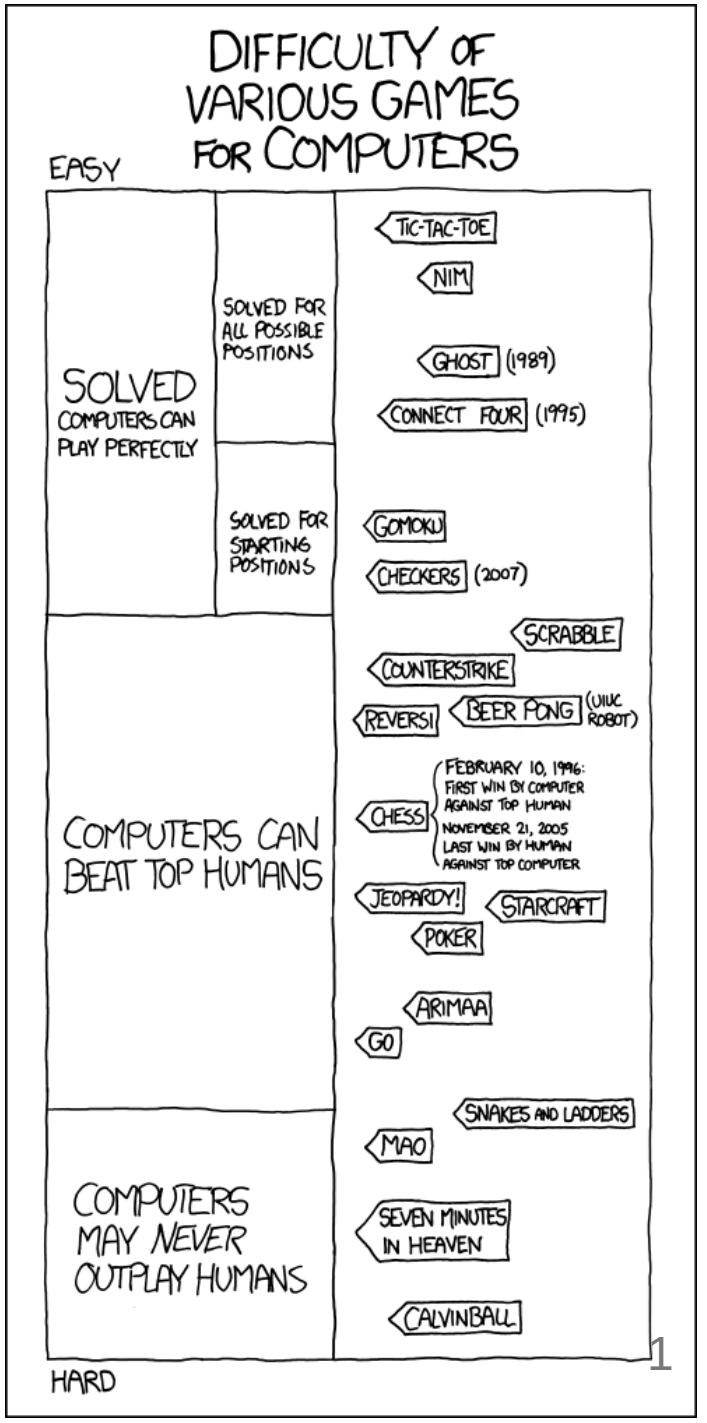
Lecture 5
Adversarial Search and Games

Gleb Sizov
Norwegian University of Science and Technology



Multi-agent, competitive environments

Board games: chess, go, poker

- Easy to represent state
- Small number of actions
- Precise rules

Physical games: croquet, hockey

- Complex descriptions
- Large number of actions
- Imprecise rules

Approaches in multi-agent environments

1. Agents as aggregates - economy

- Large number of agents, no model of individual agents

2. Opponents are part of the environment, i.e. like rain

- Nondeterministic, not adversarial

3. **Model agents explicitly - game-tree seach**

- Minimax search, pruning, heuristic evaluation function

Two-player zero-sum games

Deterministic, two-player, turn-taking, perfect information, **zero-sum**,
e.g. chess, go

- S_0 - initial state
- $ToMove(s)$ - player whose turn it is to move
- $Actions(s)$ - set of legal moves
- $Result(s, a)$ - **transition model**, result state from taking action
- $IsTerminal(s)$ - game over
- $Utility(s, p)$ - objective function, value for player p, e.g. 1 - win, 0 - lose, 0.5 - draw

Game tree

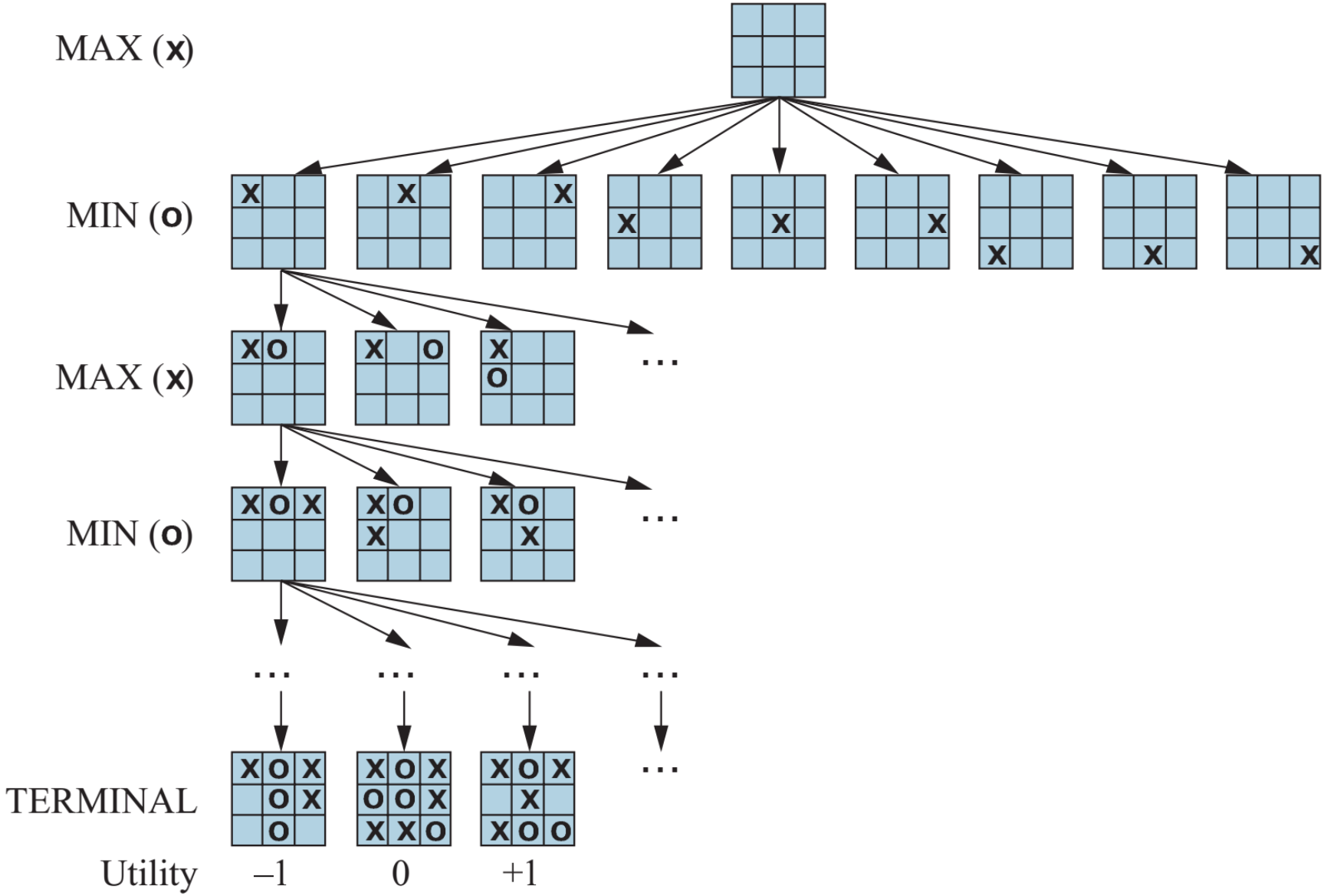
Tic-tac-toe:

363 880 nodes

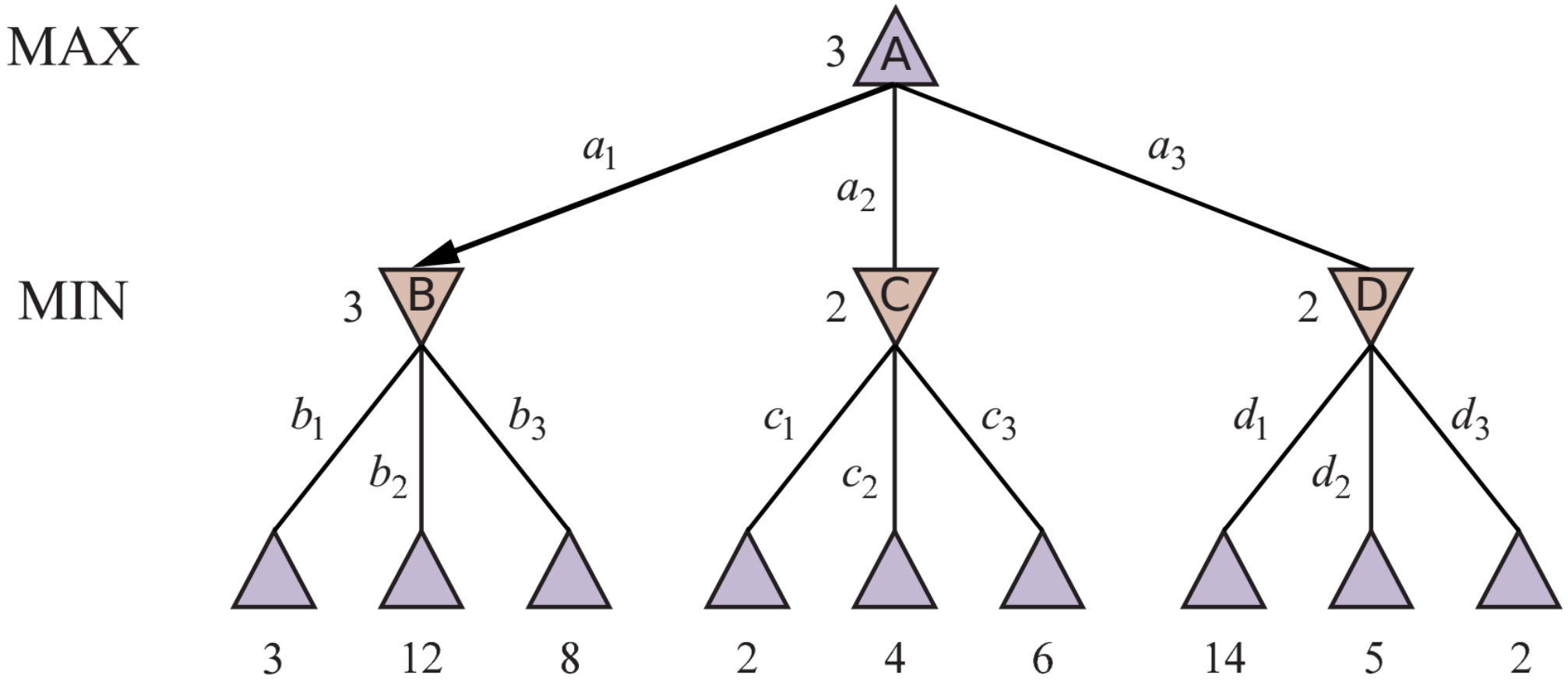
Chess:

10^{40} nodes

Theoretical construct



Minimax search

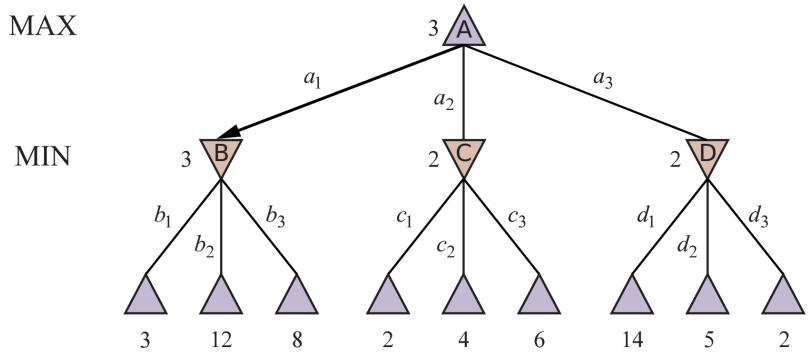


Minimax value

Conditional plan - similar to AND-OR search

Goal - "guaranteed" win

$$\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s, \textsc{max}) & \text{if } \textsc{Is-Terminal}(s) \\ \max_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s, a)) & \text{if } \textsc{To-Move}(s) = \textsc{max} \\ \min_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s, a)) & \text{if } \textsc{To-Move}(s) = \textsc{min} \end{cases}$$

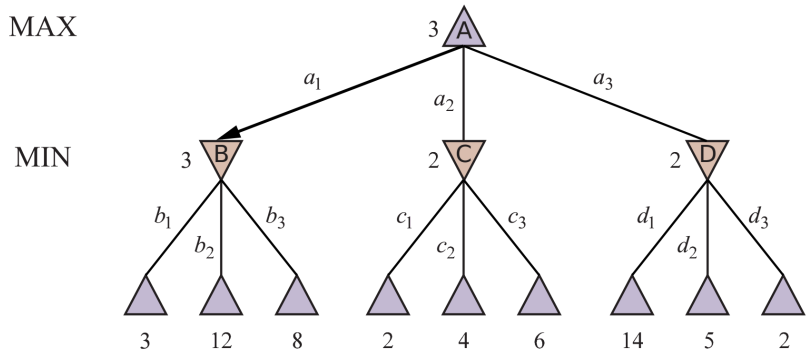


What if min doesn not play optimaly?

When it is better **not to play** the minimax optimal move?

Minimax search algorithm

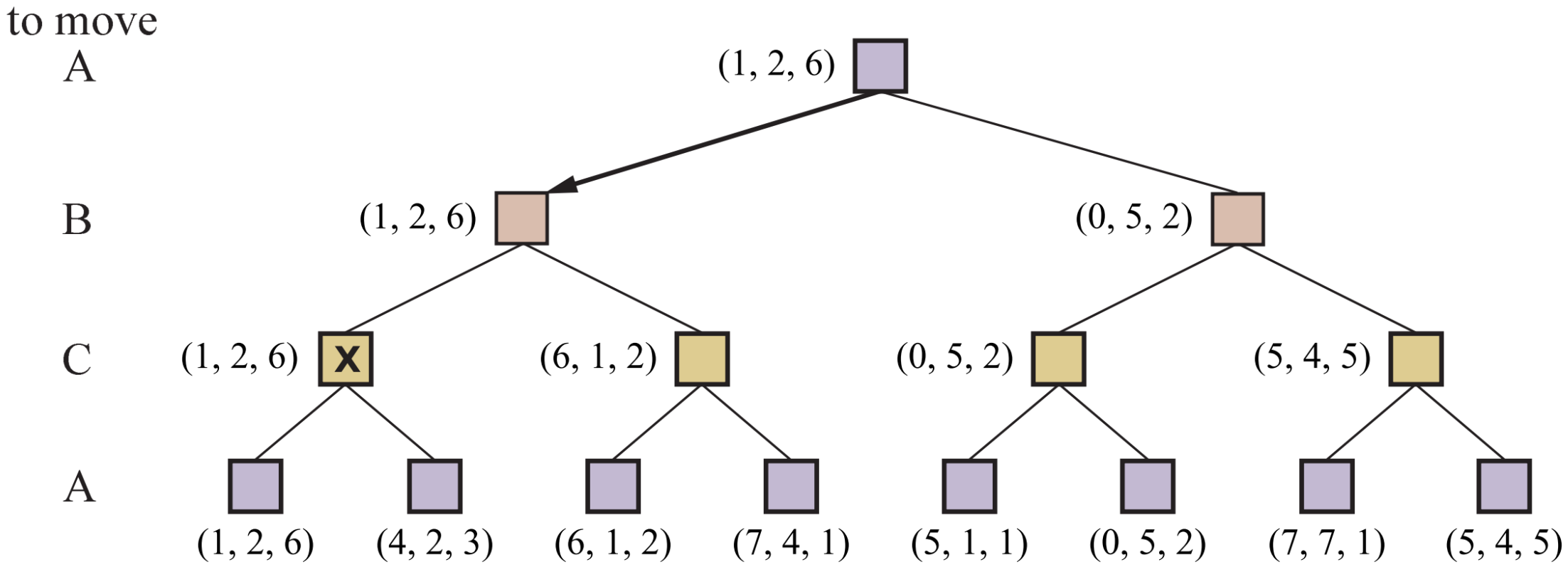
```
function MINIMAX-SEARCH(game, state) returns an action
    player ← game.TO-MOVE(state)
    value, move ← MAX-VALUE(game, state)
    return move

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
```



Minimax search complexity

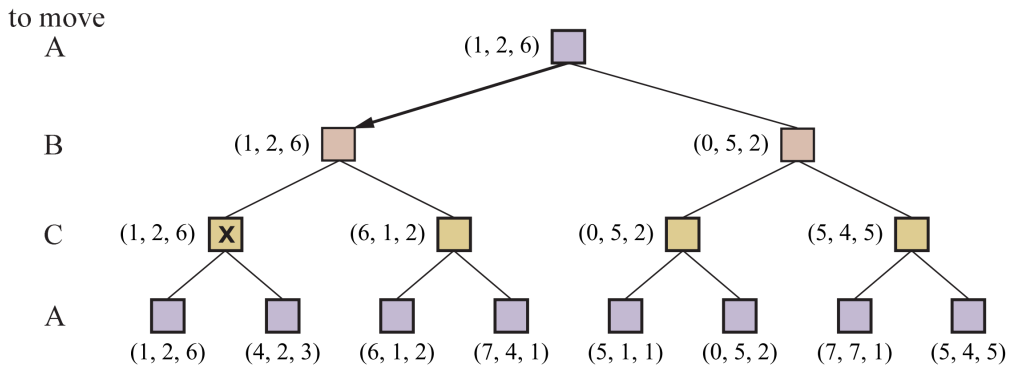
- Depth-first, recursive
- Time complexity: $O(b^m)$ - exponential!
- Space complexity: $O(bm)$ or $O(m)$

Multiplayer game tree



Alliances in multiplayer games

- Alliances - attack the strongest player
- Collaboration from selfish behaviour
- Breaking alliance - trust issues
- Collaboration



Alpha-beta pruning: Intro

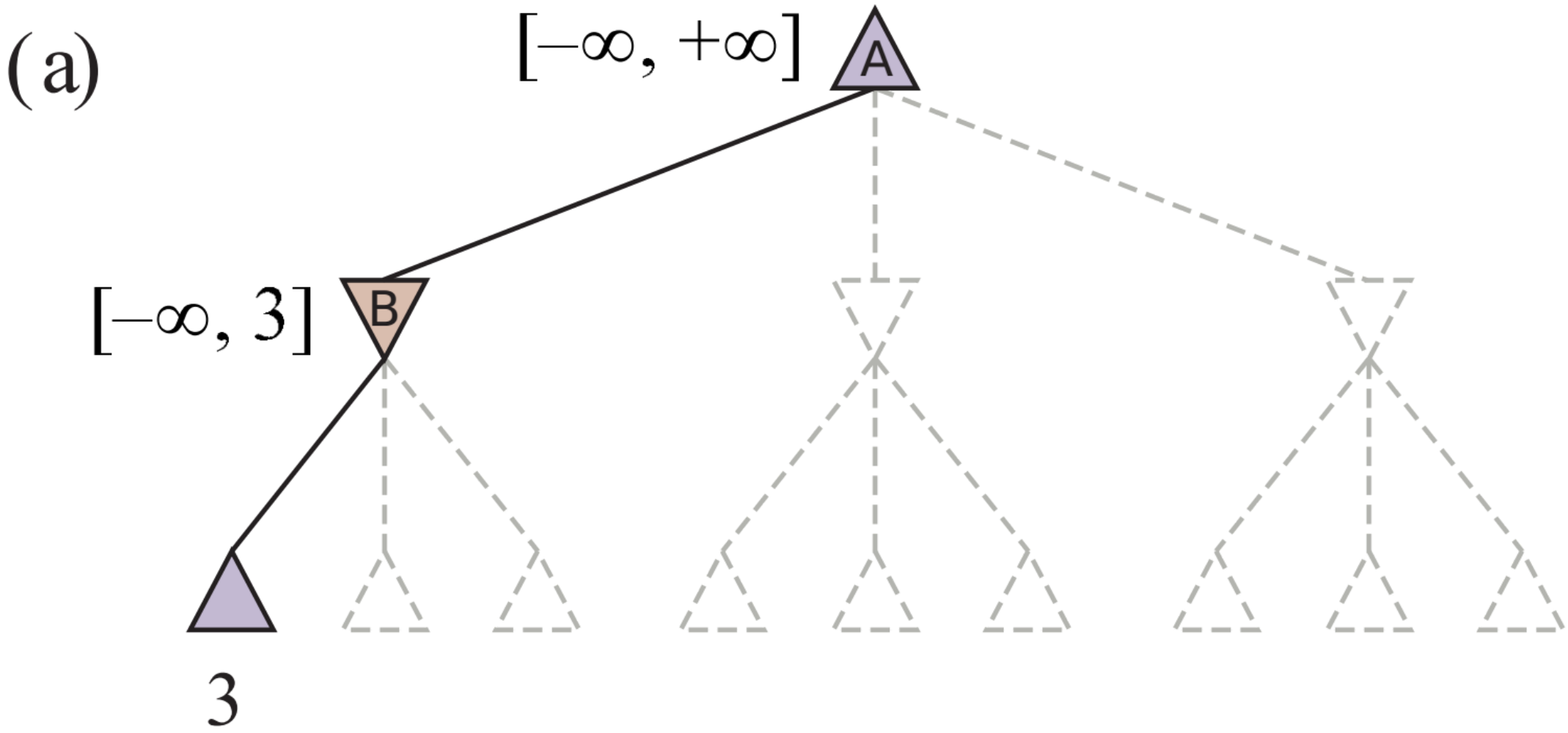
Problem:

$O(b^d)$ - exponential number of states wrt to tree depth

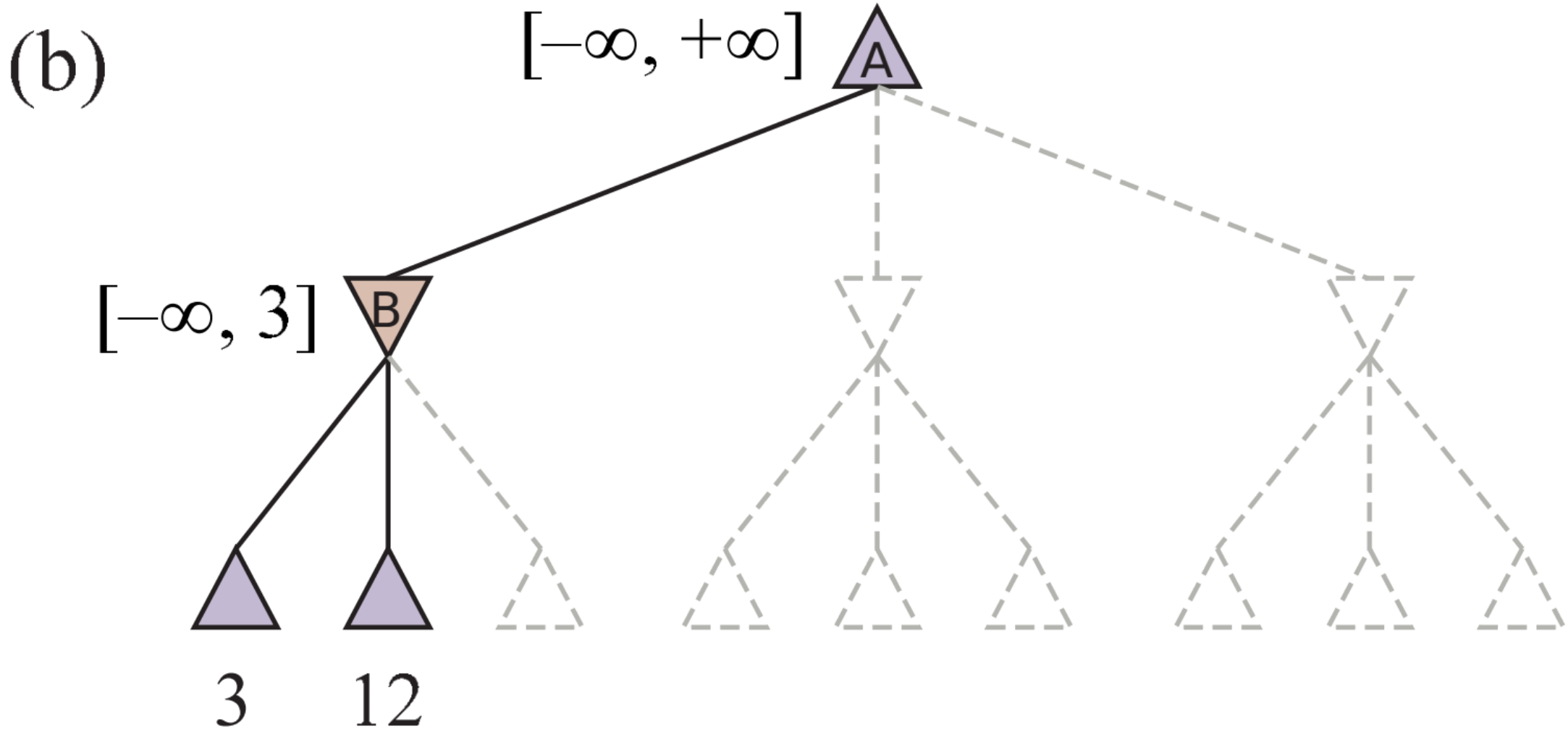
Solution:

Prune (remove) parts of the tree that make no difference in the outcome for **minimax** decisions.

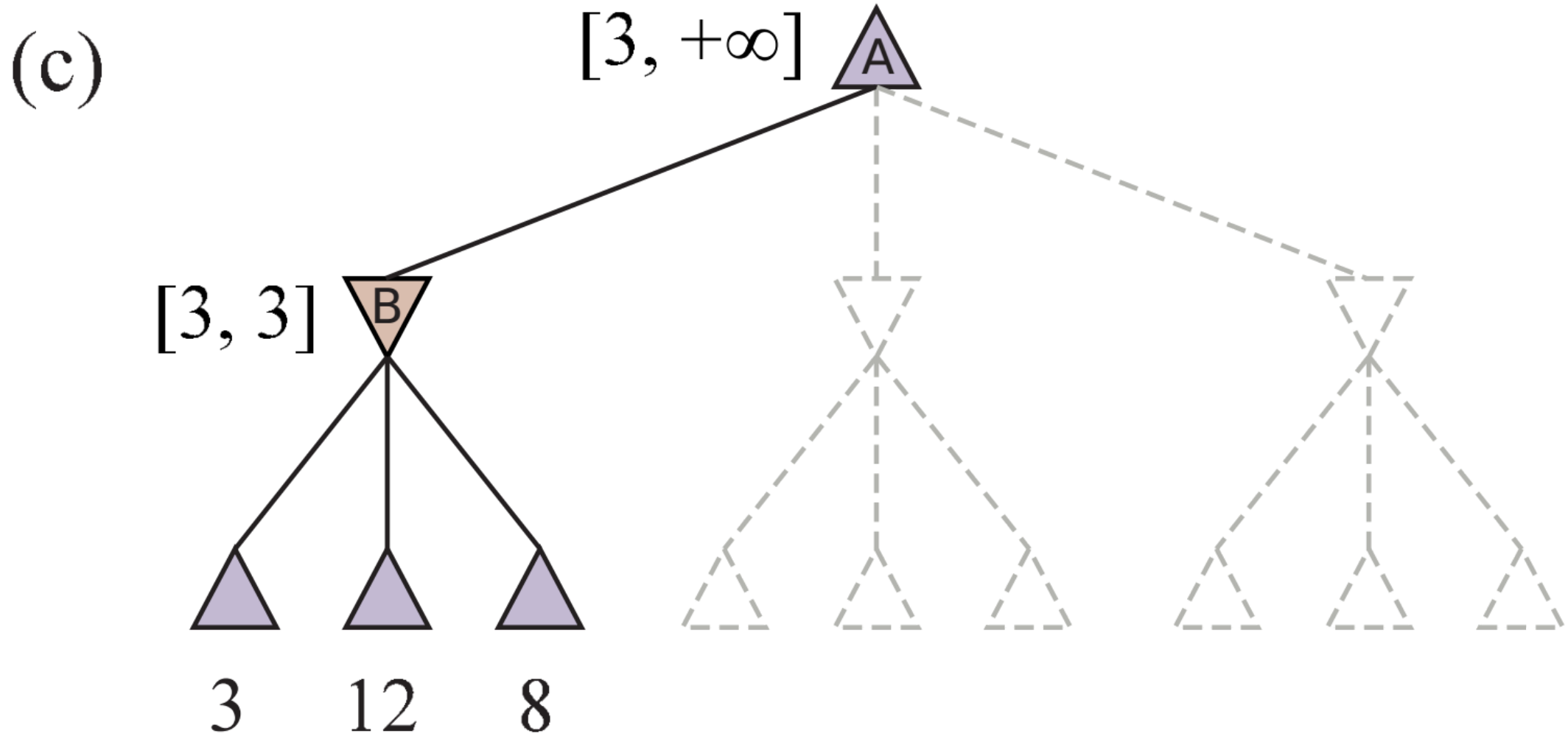
Alpha-beta pruning: Example - Stage 1



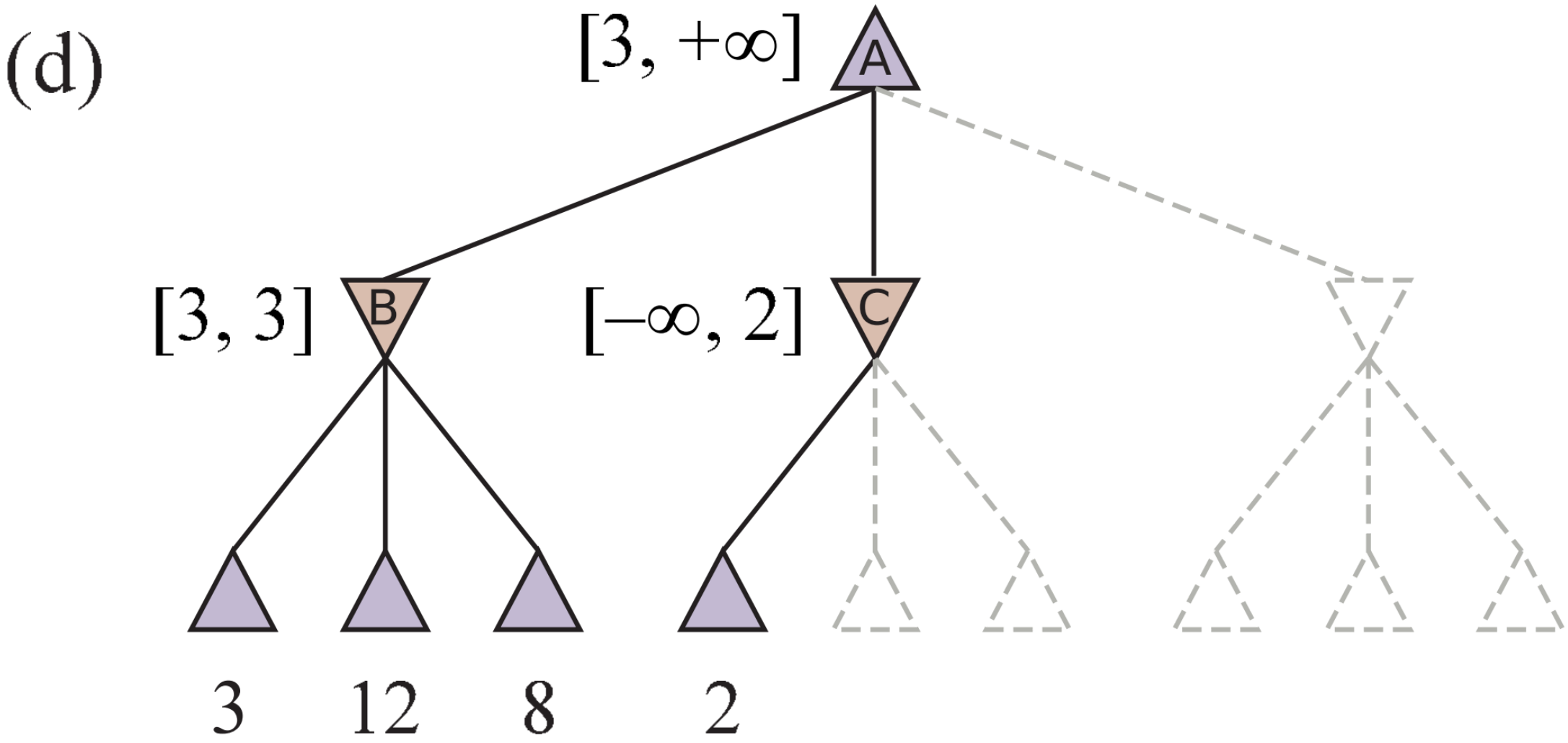
Alpha-beta pruning: Example - Stage 2



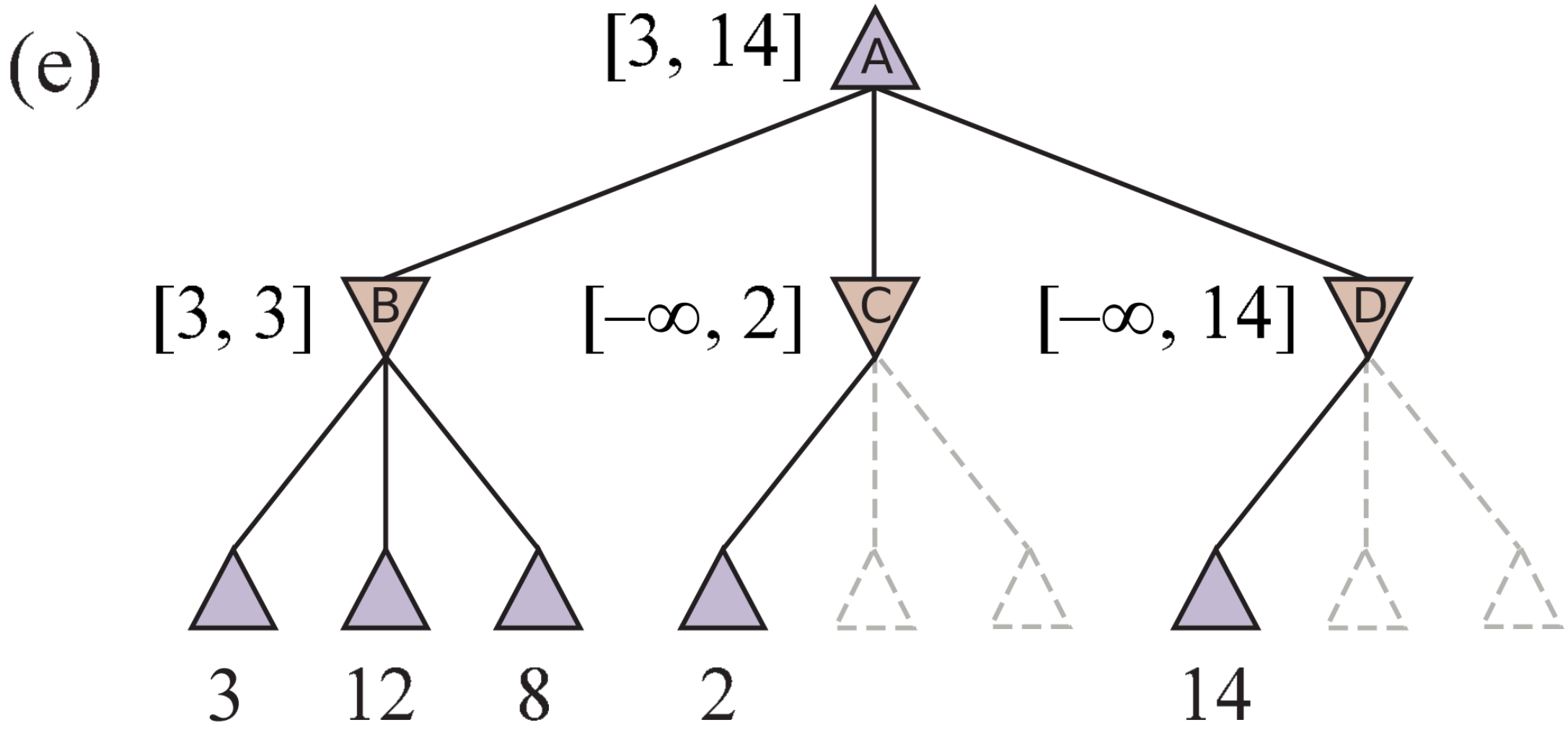
Alpha-beta pruning: Example - Stage 3



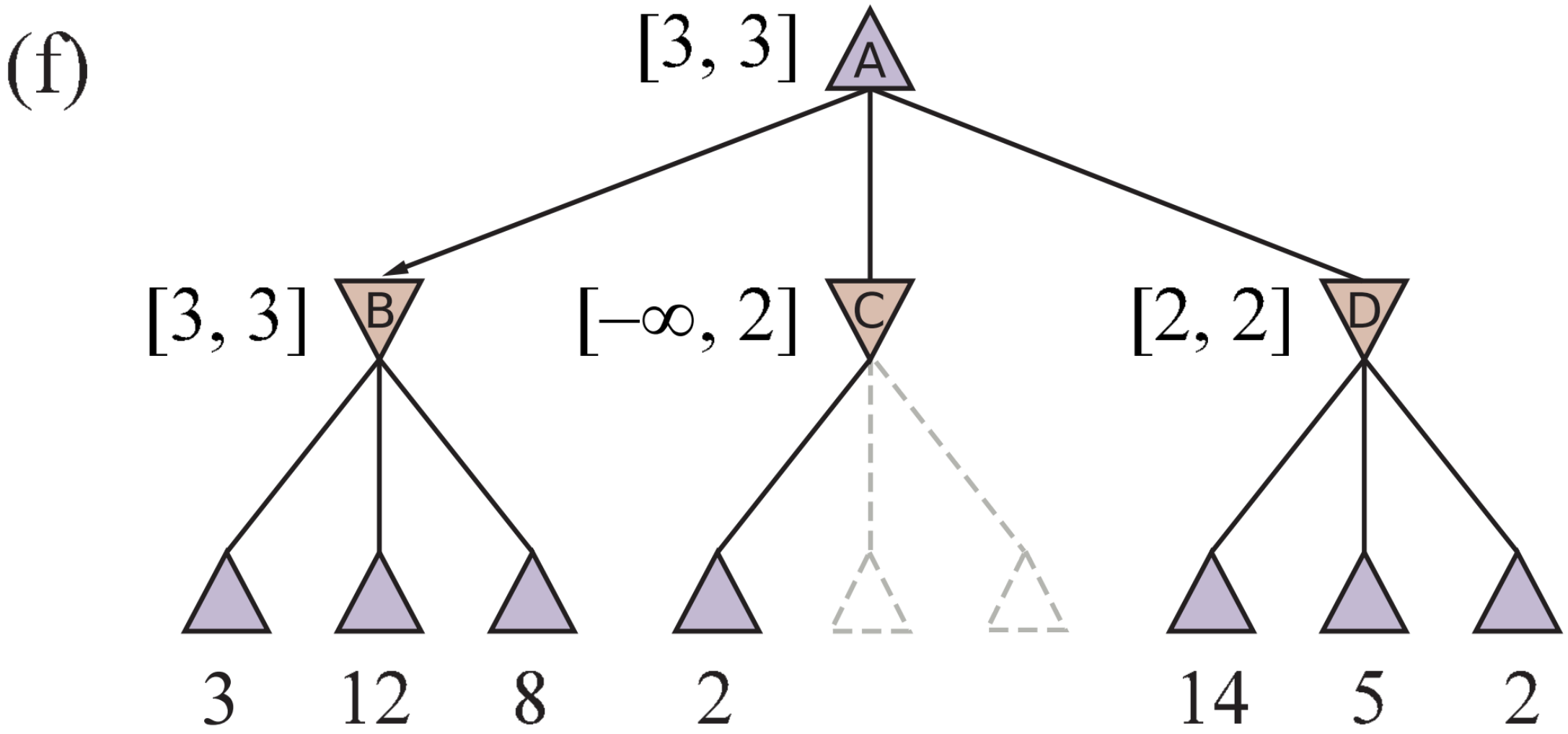
Alpha-beta pruning: Example - Stage 4



Alpha-beta pruning: Example - Stage 5

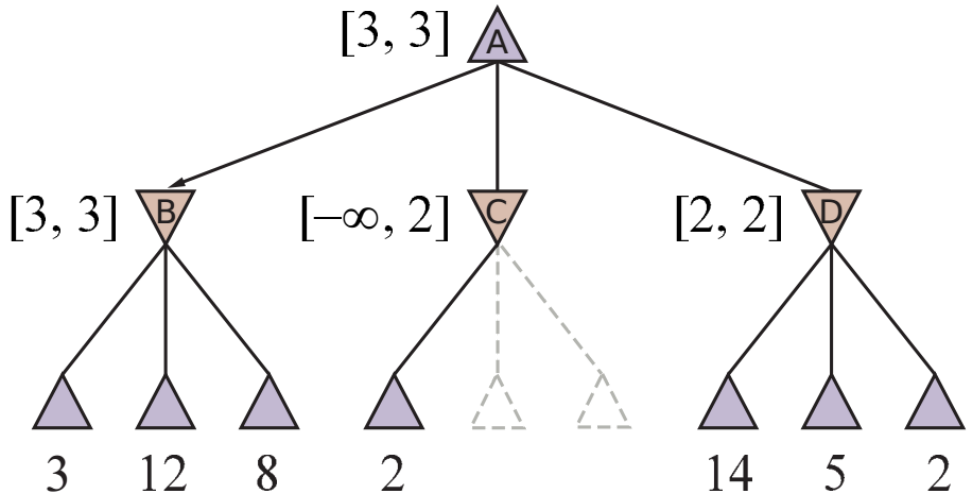


Alpha-beta pruning: Example - Stage 6



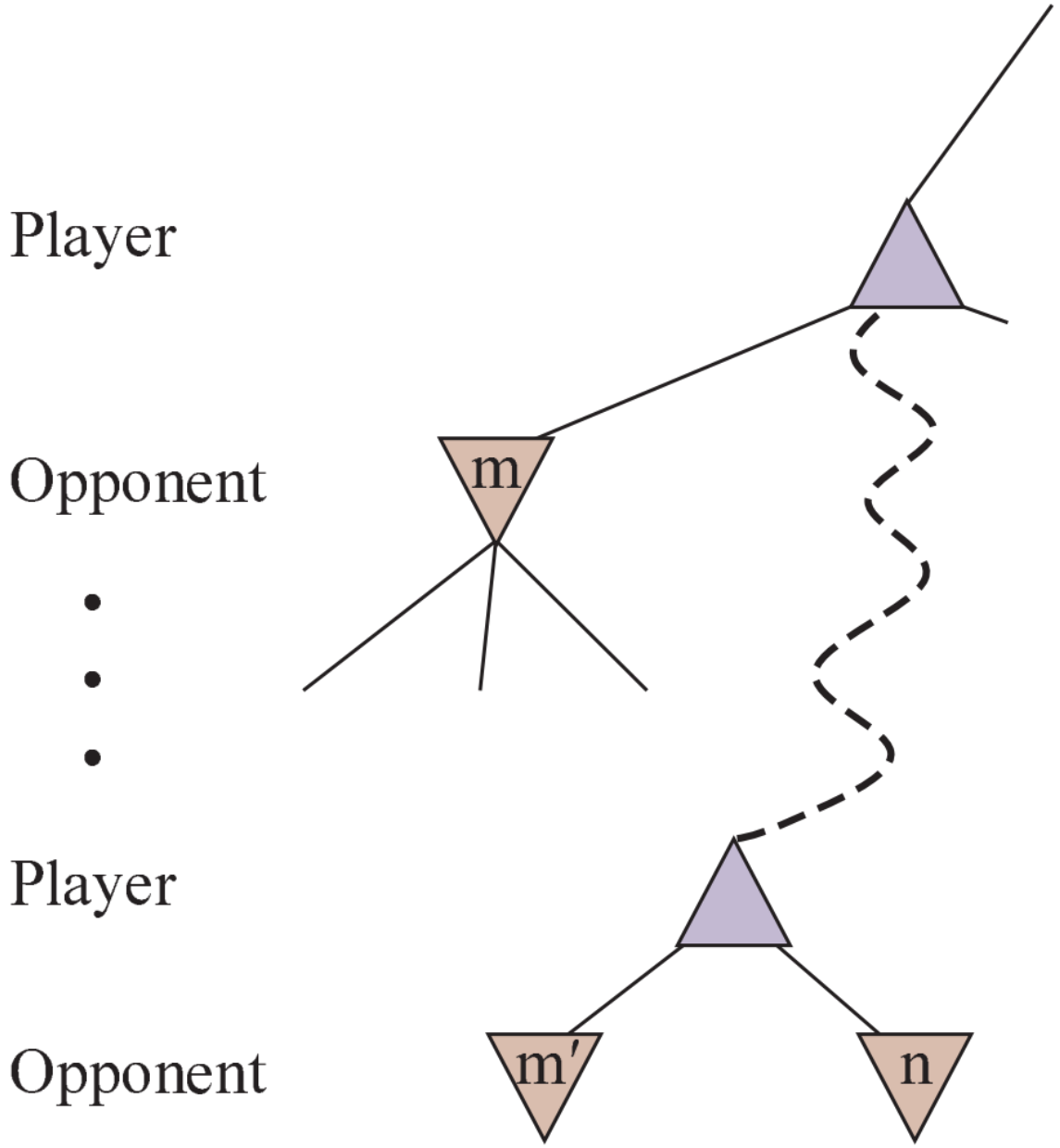
Alpha-beta pruning: Minimax formula

$$\begin{aligned}\textsc{Minimax}(root) &= \max(\min(3,12,8), \min(2,x,y), \min(14,5,2)) \\ &= \max(3, \min(2,x,y), 2) \\ &= \max(3,z,2) \qquad \text{where } z = \min(2,x,y) \leq 2 \\ &= 3.\end{aligned}$$

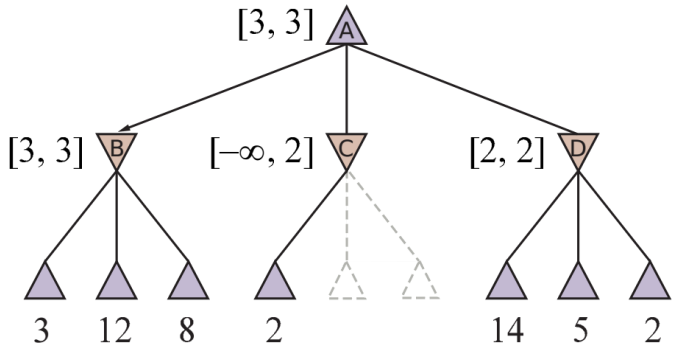


Alpha-beta pruning: General case

If m or m' is better than n for Player, we will never get to n in play.



The alpha–beta search



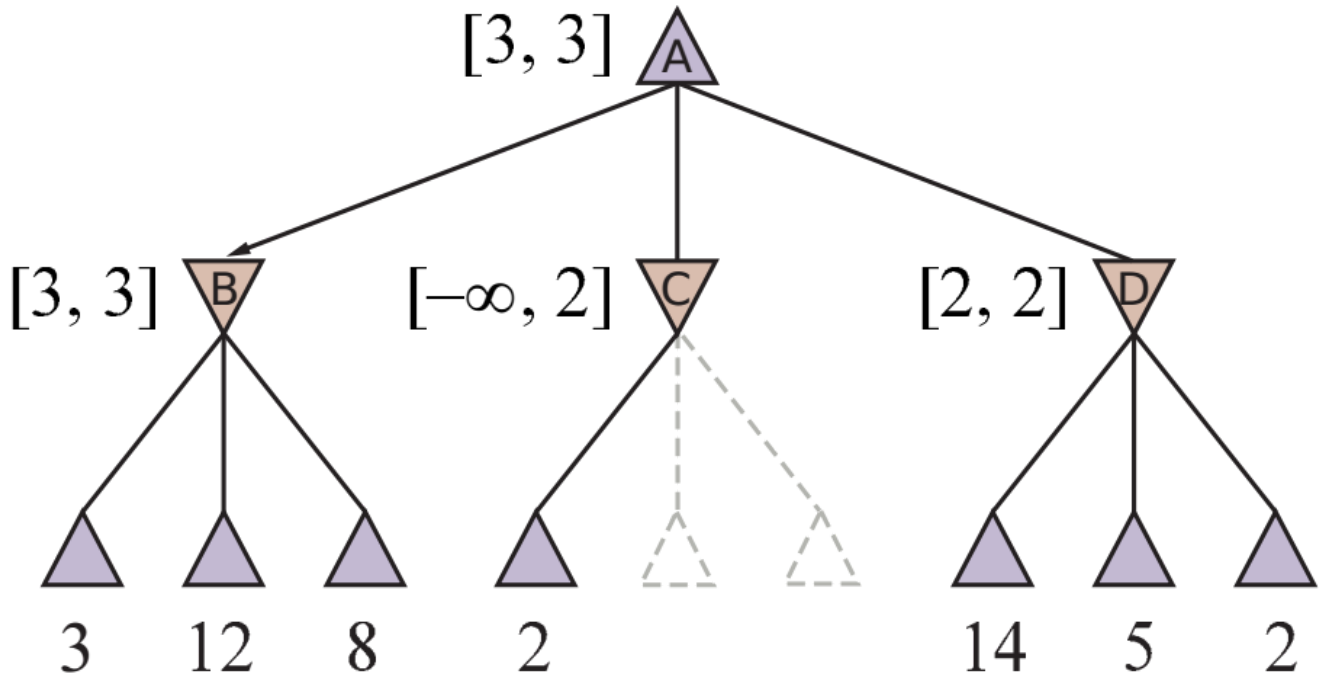
function ALPHA-BETA-SEARCH(*game*, *state*) **returns** an action
  player ← *game*.TO-MOVE(*state*)
  value, *move* ← MAX-VALUE(*game*, *state*, $-\infty$, $+\infty$)
  return *move*

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*

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*

Alpha-beta pruning: Effect of move ordering

Effectiveness of alpha–beta pruning is highly dependent on the order in which the states are examined.



Alpha-beta pruning: Complexity

Minimax with pruning: $O(b^m)$ nodes

Alpha-beta search (perfect ordering): $O(b^{m/2})$ nodes

Alpha-beta search (random ordering): $O(b^{3m/4})$ nodes

With perfect ordering

Can explore twice as deep.

Branching factor: b $\rightarrow \sqrt{b}$

For *chess*, branching factor: 36 $\rightarrow \sqrt{6}$

Alpha-beta pruning: Move-ordering schemes

Trying first the moves that were found to be best in the past (**killer moves**)

1. From previous move, e.g. same threats
2. From previous stage of iterative deepening
3. From **transposition table** - cache of state heuristic values

Type A vs Type B strategy

> Claude Shannon (1950), Programming a Computer for Playing Chess

Type A strategy - wide but shallow - considers all moves to a certain depth, then uses a heuristic function to estimate their utility.

Type B strategy - deep but narrow - ignores moves that look bad, and follows promising lines as deep as possible.

Historically, Type A for *chess* (b = 35), Type B for *Go* (b = 250)
Now, Type B for *chess* as well.

Heuristic minimax value

$$\text{H-MINIMAX}(s, d) = \begin{cases} \text{EVAL}(s, \text{MAX}) & \text{if IS-CUTOFF}(s, d) \\ \max_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,\ a), d+1) & \text{if TO-MOVE}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,\ a), d+1) & \text{if TO-MOVE}(s) = \text{MIN}. \end{cases}$$

$Eval(s, MAX)$ - heuristic **evaluation function**

$IsCutoff(s, d)$ - **cutoff test**

Evaluation function characteristics

1. $Utility(loss, p) \leq Eval(s, p) \leq Utility(win, p)$
2. Strong correlation with chances of winning
3. Fast

Evaluation function based on features and categories

1. Define state features
   e.g. number of black/white pawns, queens, etc.
2. Define *categories* or *equivalence classes* of states with same feature values
   e.g. two-pawns vs one-pawn endgames.
3. Estimate of proportion of statew with wind/losses/draws,
   e.g. 82% win (utility +1), 2% loss (utility 0), 16% to draw (utility 1/2)

Expected value = (0.82 × +1) + (0.02 × 0) + (0.16 × 1/2) = 0.90

Problem: Too many categories to estimate

Evaluation function based on combining feature values

1. Assign **material value** for each piece
   e.g. pawn is worth 1, a knight is 3, rook is 5
2. Combine material values
   e.g. weighted linear function

$$\mathrm{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)$$

Problem: Contributions of features are independant

Solution: Nonlinear combinations of features.

Q: Where weights come from?

A: Use machine learning.

Cutting off search

In alpha-beta search replace:

$$\textbf{if } game.\text{IS-TERMINAL}(state) \textbf{ then return } game.\text{UTILITY}(state, player), null$$

with:

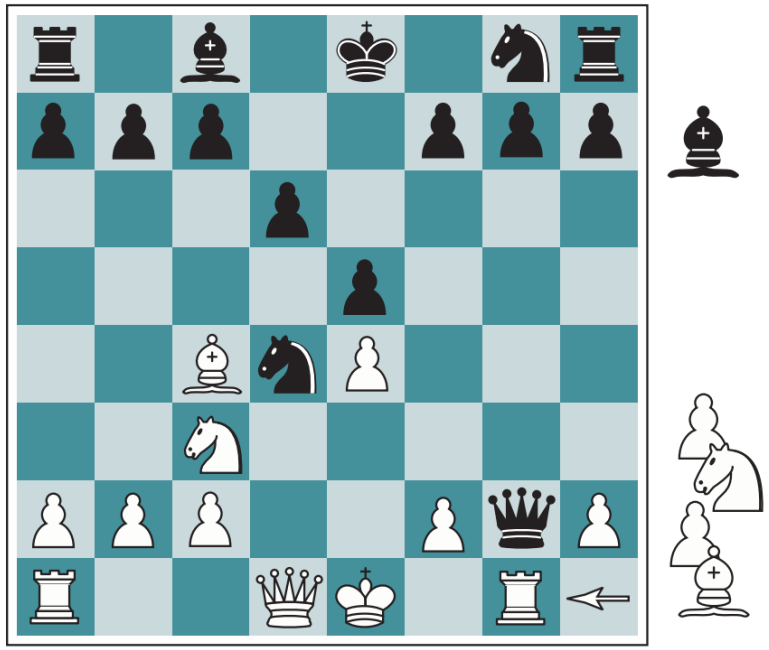
$$\textbf{if } game.\text{IS-CUTOFF}(state, depth) \textbf{ then return } game.\text{EVAL}(state, player), null$$

Cutoff depth:

- Fixed
- Iterative deepening - use previous iteration for ordering.

Cutting off search: Quiescent positions

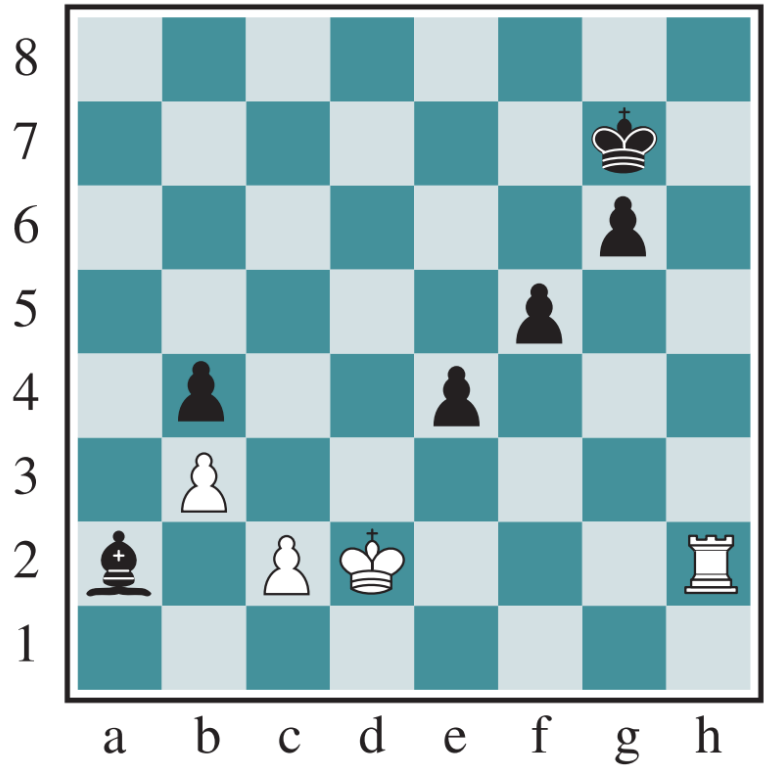
Problem: Missing damage in the pending move.



Solution: Only cutoff in **quiescent** positions - no pending damage.

Cutting off search: Horizon effect

Problem: Uavoidable damange that can be delayed.



Solution: **Singular extensions** - take moves that are "clearly better".

Forward pruning

> Remove moves that appear to be poor moves, but might possibly be good ones

Approaches:

- Beam search - consider n best moves.
- ProbCut - combines alpha-beta search with statistics from prior experiences.
- Late move reduction - uses move ordering to reduce the depth for later nodes

Endgame lookup tables

Generate a table for endgames with seven or few pieces.
Table size: 400 trillion position

Retrograde analysis - start from from final positions (check mate, steal mate) and play backwards to mark all nodes in the tree as win, loose or draw.

Bring it all together: Chess example

Compute: 60 million nodes/min

1. Combine **minimax** with **evaluation function**, **cuttoff** and **quiescence search**
   35^5 = 50 million, depth = 5
   Average human player, depth = 6, 8
2. Add **alpha-beta prunning**, **transposition table**
   Expert level, depth = 14
3. Add **8 GPUs**, **better evaluation function** + **lookup table**
   Sockfish, depth = 30
   Beats any human player

Weaknesses of heuristic alpha-beta tree search

Example: Go

1. High branching factor = 361, can only reach depth = 4, 5
2. Hard to define goode evaluation function:
    i. Material value is not a strong indicator
    ii. Most positions are influx until the endgame.

Monte Carlo tree search: Main idea

Estimate value of a *state* as the average utility over a a number of simulations of complete games starting from this *state*.

Simulation = **playout** = rollout

For win/loss games: average utility = win percentage

Monte Carlo tree search: Playout move selection

How to choose playout moves?

1. Randomly? - for some simple games it works, mostly not
2. Use **playout policy** that tends to select good moves
    i. Use game-specific heurisitc, e.g. "capture moves" for Chess
    ii. Learn good moves with machine learning, e.g. neural network for Go

Pure Monte Carlo search

What positions do we start the playouts?

How many playouts for each position?

N simulations for all possible moves from the current state.

Problem: Need large N for optimal play - high computational cost.

Monte Carlo tree search: Selection policy

Focus computational resources on the imporatant parts of the game tree.

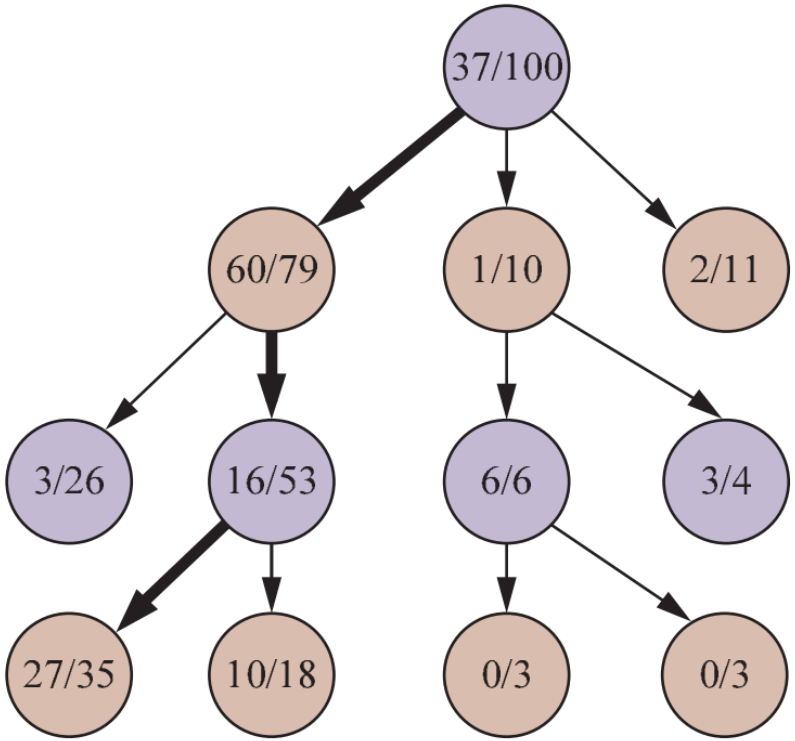
Balance exploration vs exploitation:

- **Exploration** - states that have had few playouts
- **Exploitation** - states that have done well in the past playouts

Monte Carlo tree search steps

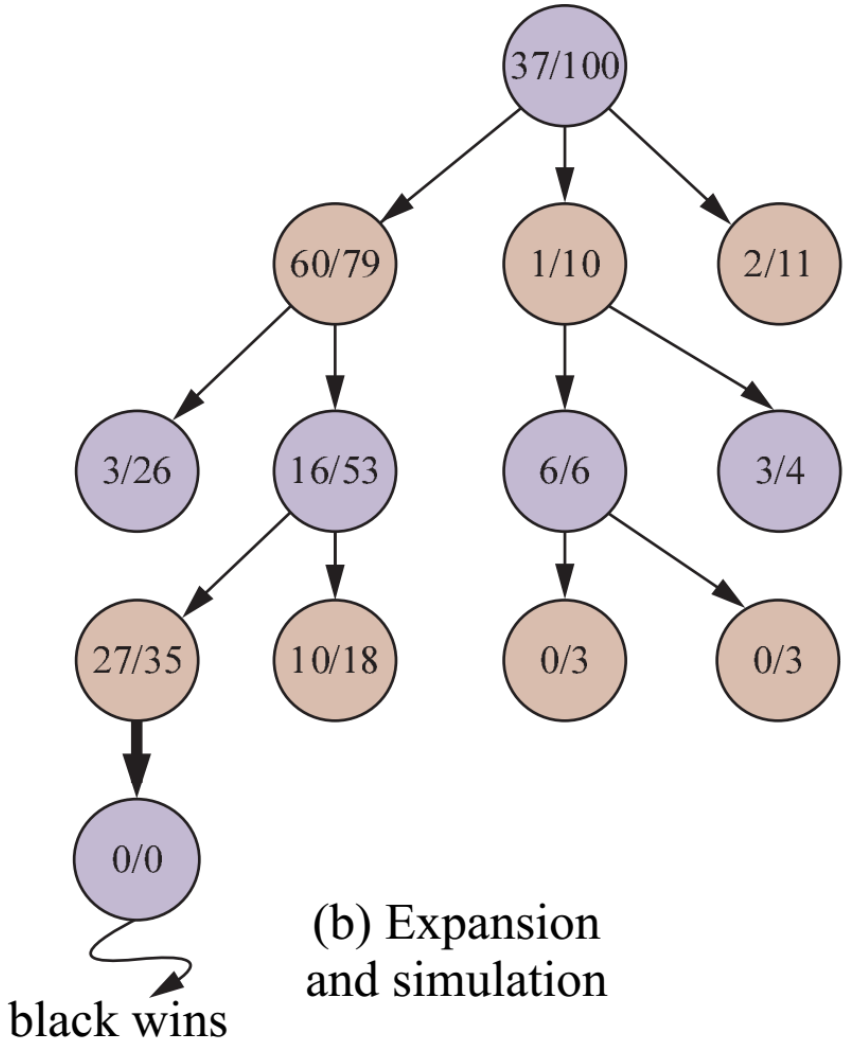
1. Selection
2. Expansion
3. Simulation
4. Backpropagation

Monte Carlo tree search: Selection step



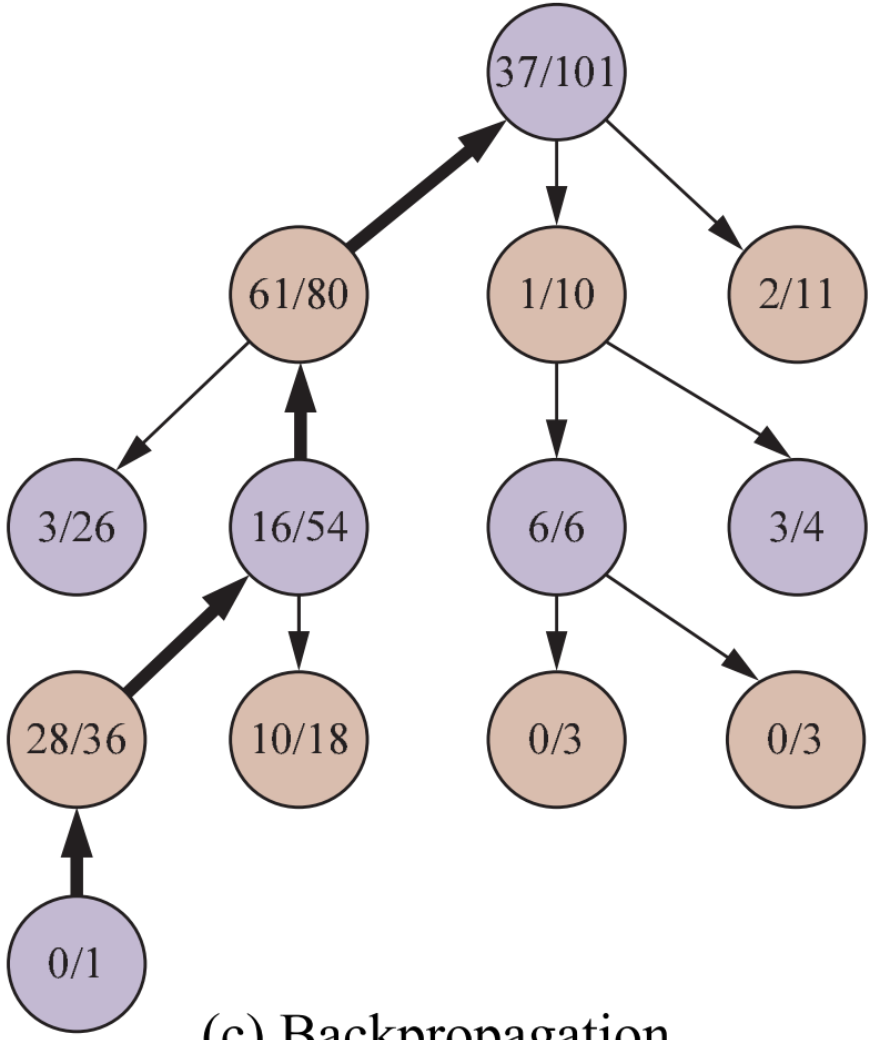
(a) Selection

Monte Carlo tree search: Expansion and simulation steps



(b) Expansion and simulation

Monte Carlo tree search: Backpropagation



(c) Backpropagation

Monte Carlo tree search algorithm

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

Selection policy: Upper confidence bounds applied to trees - UCT

Upper confidence bound - **UCB1**:

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(\text{PARENT}(n))}{N(n)}}$$

Exploitation term

Exploration term

- $U(n)$ - total utility of playouts through node n
- $N(n)$ - number of playouts through node n
- $Parent(n)$ - parent node of n
- C - constant to balance exporation and exploitation

Monte Carlo tree search pros and cons

Pros:

- Better for high branching factor than alpha-beta search
- Less sensistive to evaluation function errors than alpha-beta search
- Can be applied to new games without evaluation function
- Easily parallelized

Cons:

- Fails for games where a single move can often change the course of the game.

Variations:

- Monte Carlo tree search + evaluation function

Stochastic games

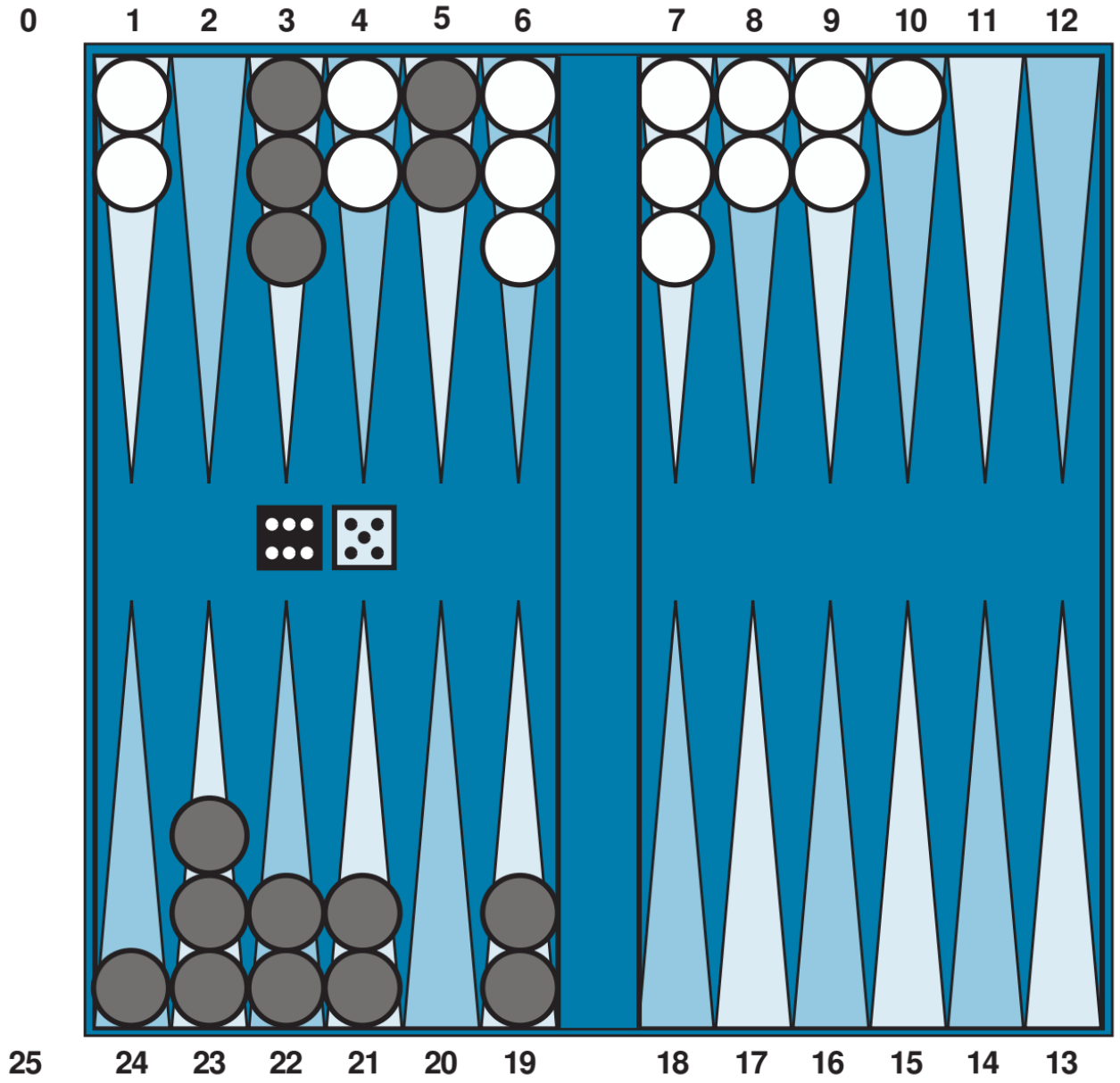
Include random element, e.g. dice

Example: Backgammon

Black rolled 6 - 5

Legal moves:

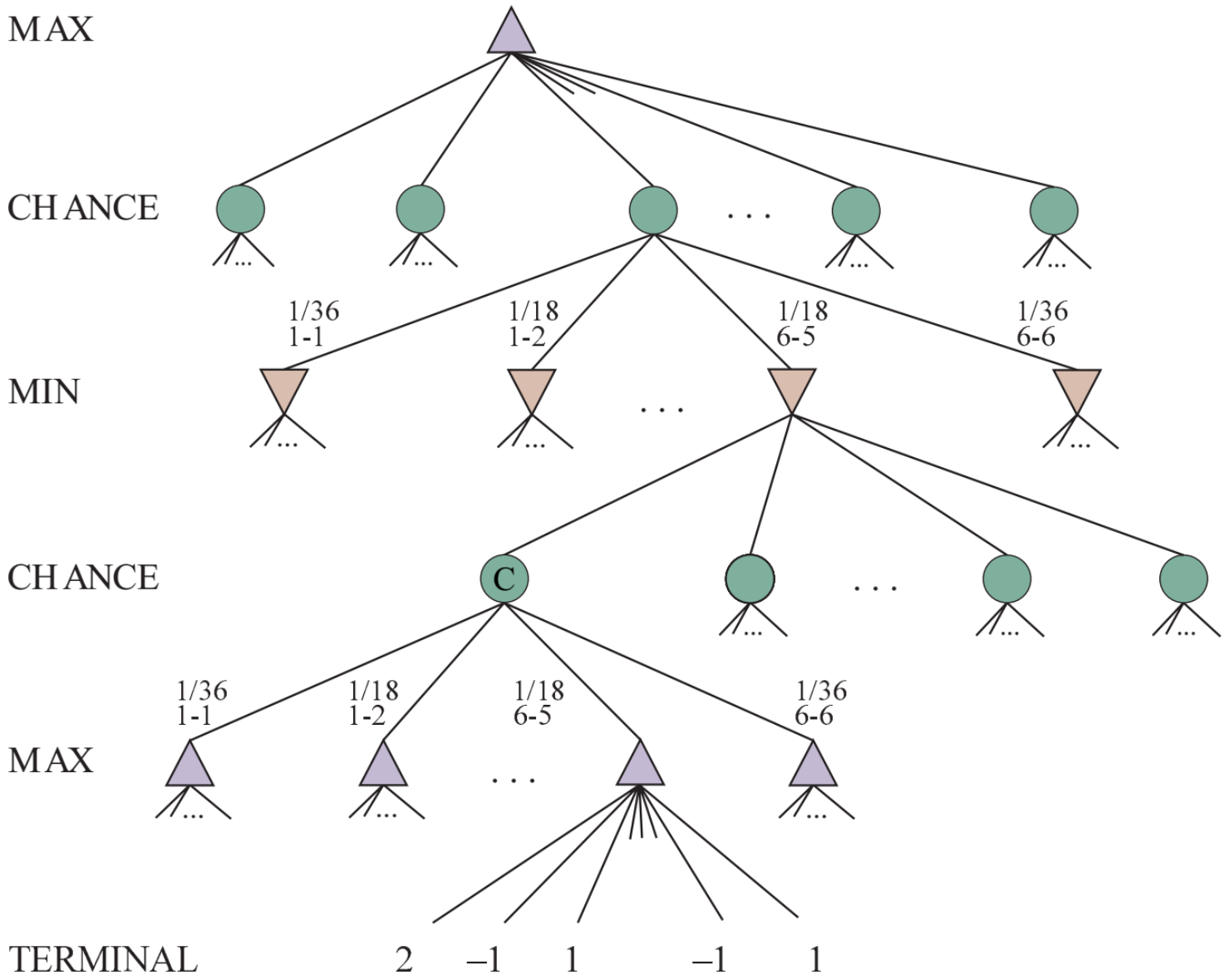
1. (5 − 11, 5 - 10)
2. (5 − 11, 19 - 24)
3. (5 − 10, 10 - 16)
4. (5 - 11, 11 - 16)



Stochastic game tree

Nodes:

- Max
- Min
- Chance nodes (probability)



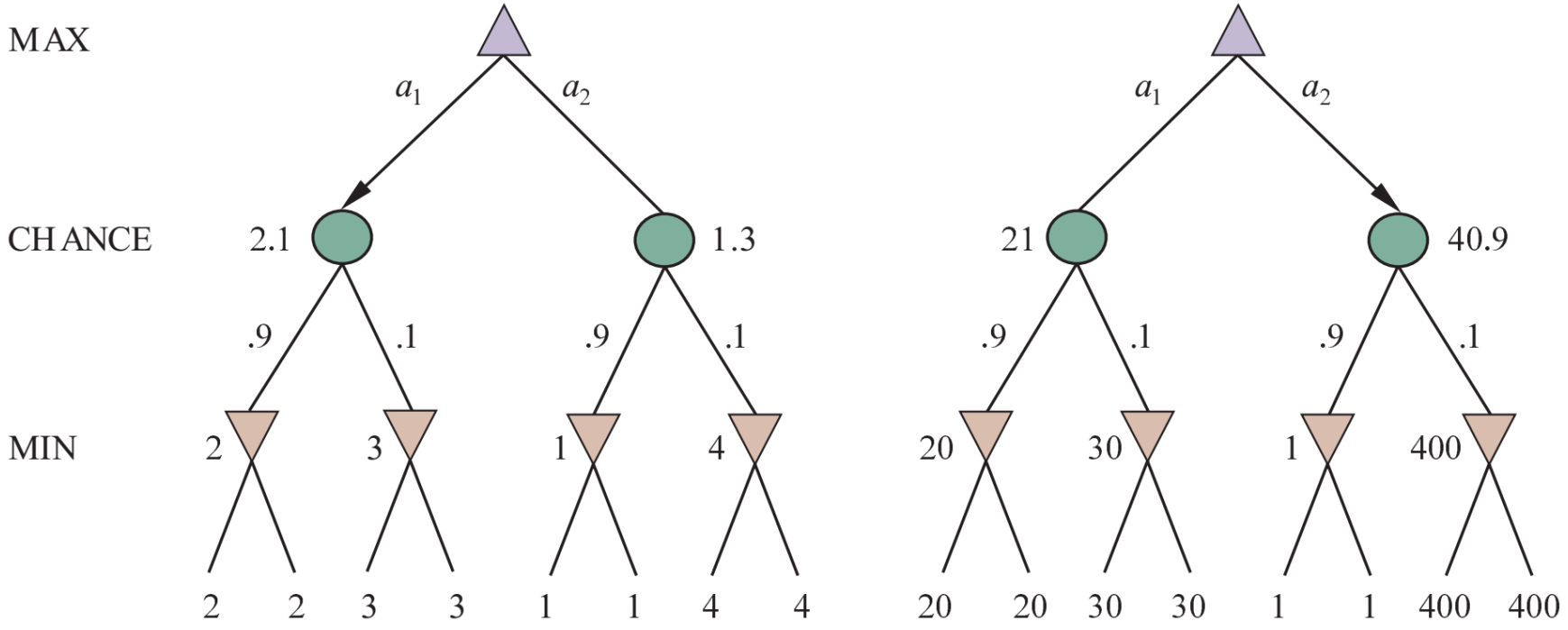
Expected value: expectiminimax value

$$\textsc{Expectiminimax}(s) = \begin{cases} \textsc{Utility}(s,\textsc{max}) & \text{if } \textsc{Is-Terminal}(s) \\ \max_a \text{if } \textsc{Expectiminimax}(\textsc{Result}(s,a)) & \text{if } \textsc{To-Move}(s) = \textsc{max} \\ \min_a \text{if } \textsc{Expectiminimax}(\textsc{Result}(s,a)) & \text{if } \textsc{To-Move}(s) = \textsc{min} \\ \sum_r P(r)\textsc{Expectiminimax}\Big(\textsc{Result}\Big(s,r\Big)\Big) & \text{if } \textsc{To-Move}(s) = \textsc{chance} \end{cases}$$

Stochastic games: evaluation functions

if *game*.IS-CUTOFF(*state*, *depth*) **then return** *game*.EVAL(*state*, *player*), *null*

Problem: "Higher values to better postions" is not enough.



Fix: Evaluation function should be positive linear transformation of the expected utility.

Stochastic games: Complexity of expectiminimax

Time complexity: $O(b^m n^m)$

- b - branching factor
- m - maximum depoth of the game tree
- n - number of distinct rolls

Example: Backgammon

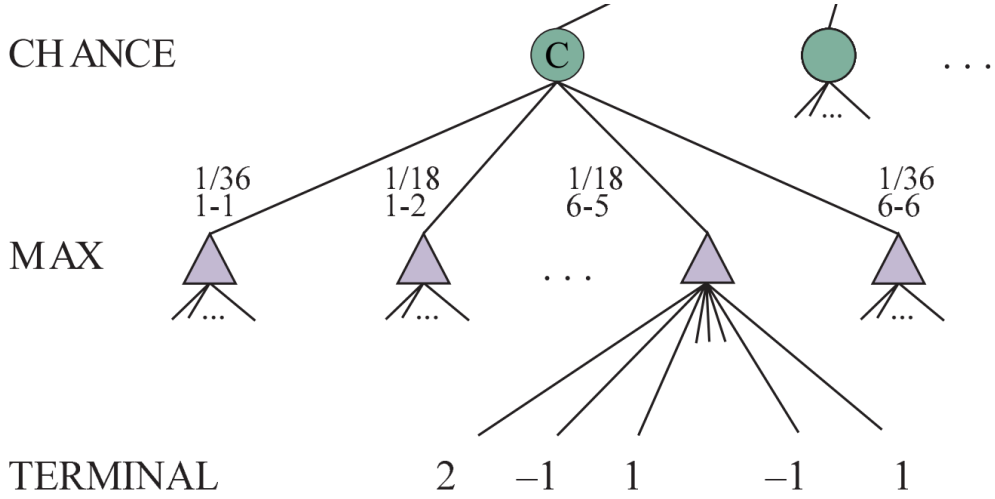
- n = 21
- b = 20 (can be up to 4000)
- Max search depth $\approx$ 3

Uncertainty multiplies possibilities!

Less meaningful to make detailed plans.

Stochastic games: Reducing complexity

1. Alpha-beta pruning with utility bounds.
2. Forward prunning
3. Monte Carlo tree search



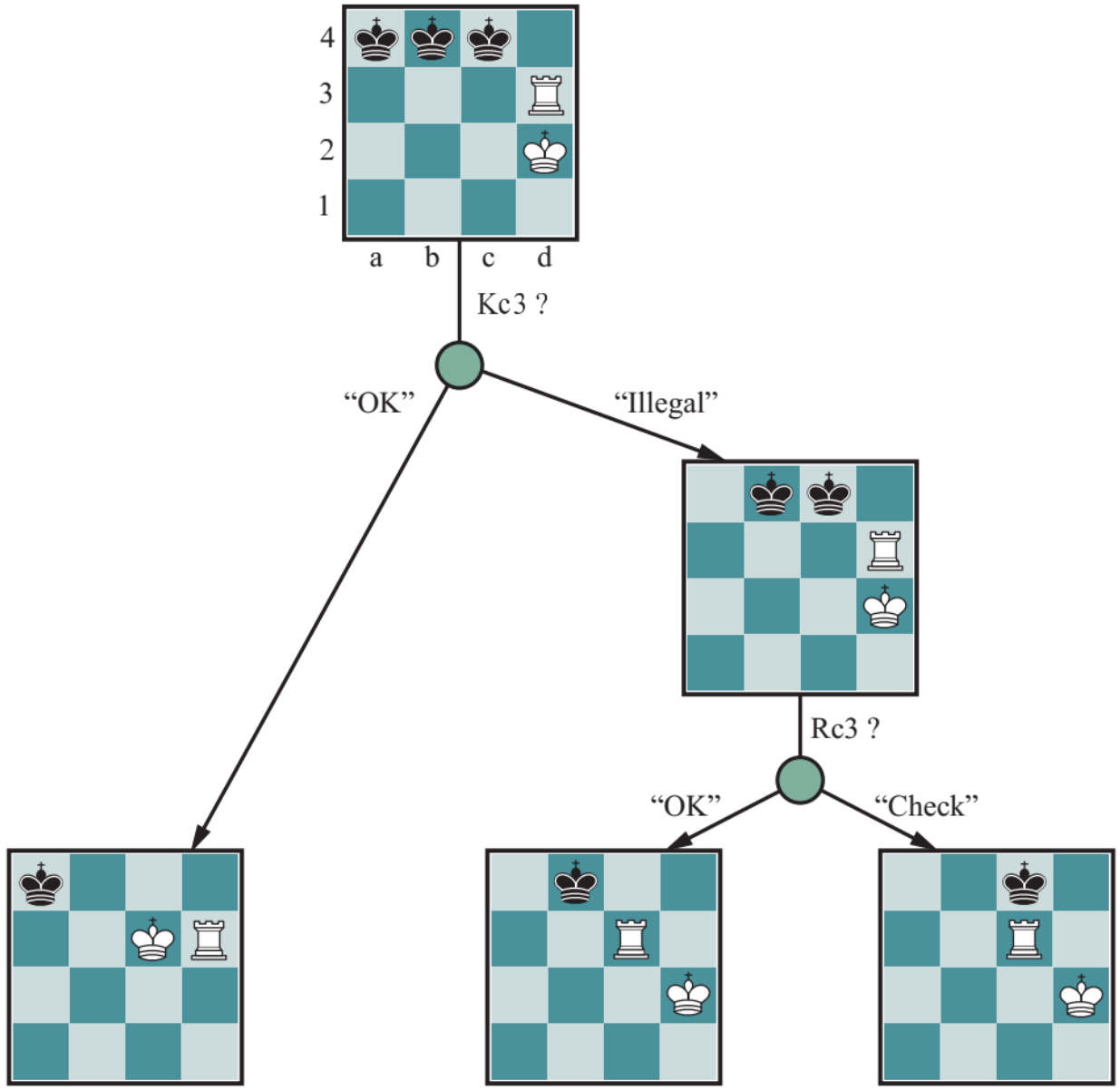
Partially obervable games

Fog of war - (partially) hidden opponent state and actions

Krigspiel:
Chess with invisible opponent pieces.

AND-OR search for partially observable games

- Belief states
- Guaranteed checkmate
- Probabalistic checkmate
- Accidental checkmate
- Confusing random moves



Card games - Averaging over clairvoyance

Stages:

1. Dealing of cards - chance node.
2. Then use **expectiminimax**

Problem: Large number of possible deals
e.g. in bridge 10 million

Tricks to reduce complexity:

1. Abstraction - treat similar hands as identical
2. Forward prunning - consider sample of deals.
3. Heuristic search with depth cutoff

Limitations of Game Search Algorithms

Limitations of alpha-beta search

1. Too slow for games with high branching factor.
2. Vulnarable to errors in heuristic function.

Limitations of alpha-beta search and Monte Carlo:

1. Don't consider the *utility of a node expansion*, **metareasoning**.
2. No abstract level reasoning with high-level goal.
3. Limited use of **machine learning**