Overview

5. Adversarial Search and Games

- Game Theory
- Optimal Decisions in Games
- Heuristic Alpha-Beta Tree Search
- Monte Carlo Tree Search
- Stochastic Games
- Partially Observable Games
- Limitations of Game Search Algorithms

Game Theory

- Games are **competitive environments with two or more agents** having conflicting goals
- Types of games
  - **Classical games** like chess, Go, poker etc.
  - **Physical games** like soccer, football etc.
  - **Economic planning games**
- Concentration in this lecture on **classical games** being simpler than other games:
  - State of game easy to represent,
  - Agents restricted to small number of actions
  - Effect of actions defined by precise rules
  - Well defined goal
- Physical games: Mostly robot soccer
- Planning games: Usually an environment for training people

Key Concepts

- Approaches to deal with multiagent environments
  - Aggregate view as an economy, e.g. increasing demand might cause prices to rise
  - Other agents are part of the environment, but do not act randomly (like rain), rather follow their own, often adversarial goals
  - Explicit modeling of adversarial agents (focus in this chapter)
    - Adversarial search
    - Pruning of irrelevant parts of the search tree
    - Evaluation function to assess states (positions)
    - Dealing with imperfect information

Two-player Zero-sum Games

- Most commonly type of game studied in AI (like e.g. chess and Go)
- Deterministic, fully observable (perfect information)
- Zero-sum means, what is good for one agent is bad for the other
  - Players (agents) are therefore usually called „Max“ and „Min“
- Terminology: „move“ ≡ „action“; „position“≡ „state“

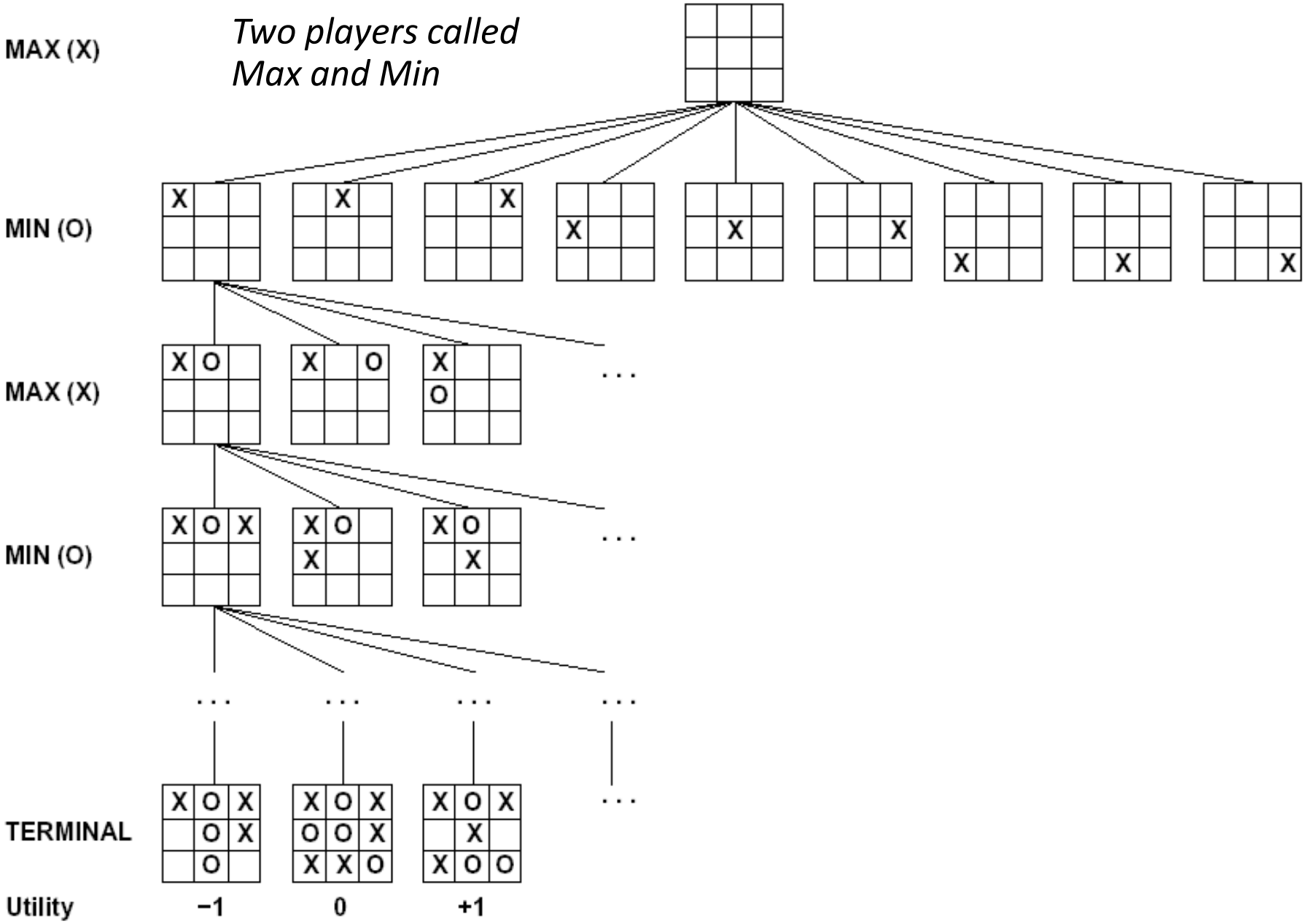
Example of Two-player Zero-sum Game

Tic-tac-to:

- Game tree (right) has 9! = 362 880 terminal nodes (paths).
- With only 5 478 distinct states

Chess:

- ca. 35^{100} paths in game tree (game with 50 moves for each player)
- ca. 10^{40} distinct states



Some Definitions

- S_0: **initial state**
- To-Move(s): Player to move
- Actions (s): Legal moves in state s
- Results (s, a): Effect of an action in a state; **transition model**
- Is-Terminal (s): A terminal test, whether the game has ended in a **terminal state**
- Utility (s, p): **Utility function** (payoff function) defining game outcome (e.g. in chess: 1, 0, ½)

Minimax Search Algorithm

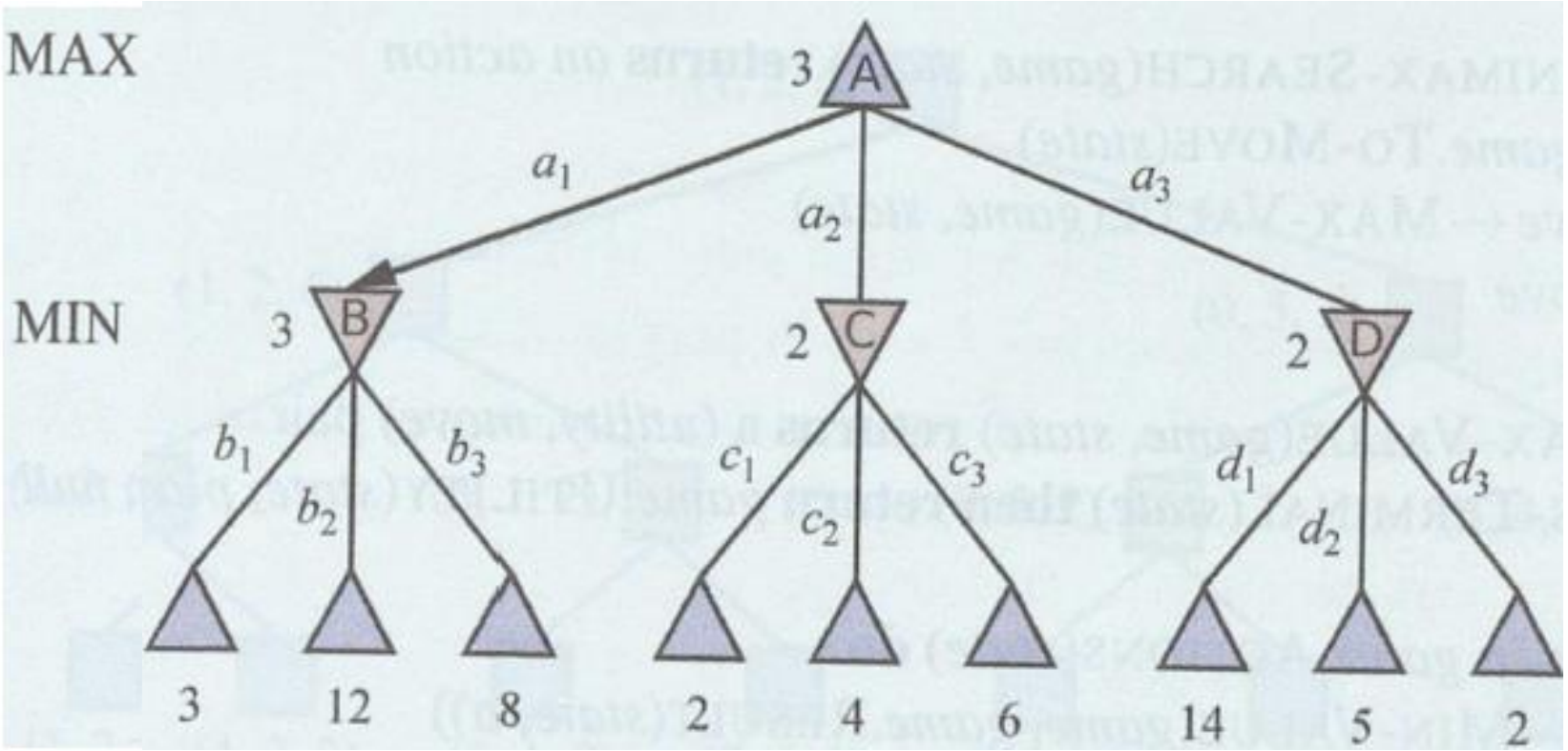
- Generate full search tree
- Apply utility function on each terminal node
- Compute utilities of a node from its sucessors:
  - If Min is to-move, take minimum of utilities
  - If Max is to-move, take maximum of utilities
- Apply this procecure recursively until the root of the search tree
- Choose move with highest utility if Max is to-move, otherwise move with smallest utility.

```
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
```

Example for Minimax Search Algorithm



Minimax (s) =

- Utility (s, Max)
  if Is-Terminal (s)
- $\max_{a \in \text{actions (s)}}$ (Minimax (Result (s,a))
  if To-move(s) = Max
- $\min_{a \in \text{actions (s)}}$ (Minimax (Result (s,a))
  if To-move(s) = Min

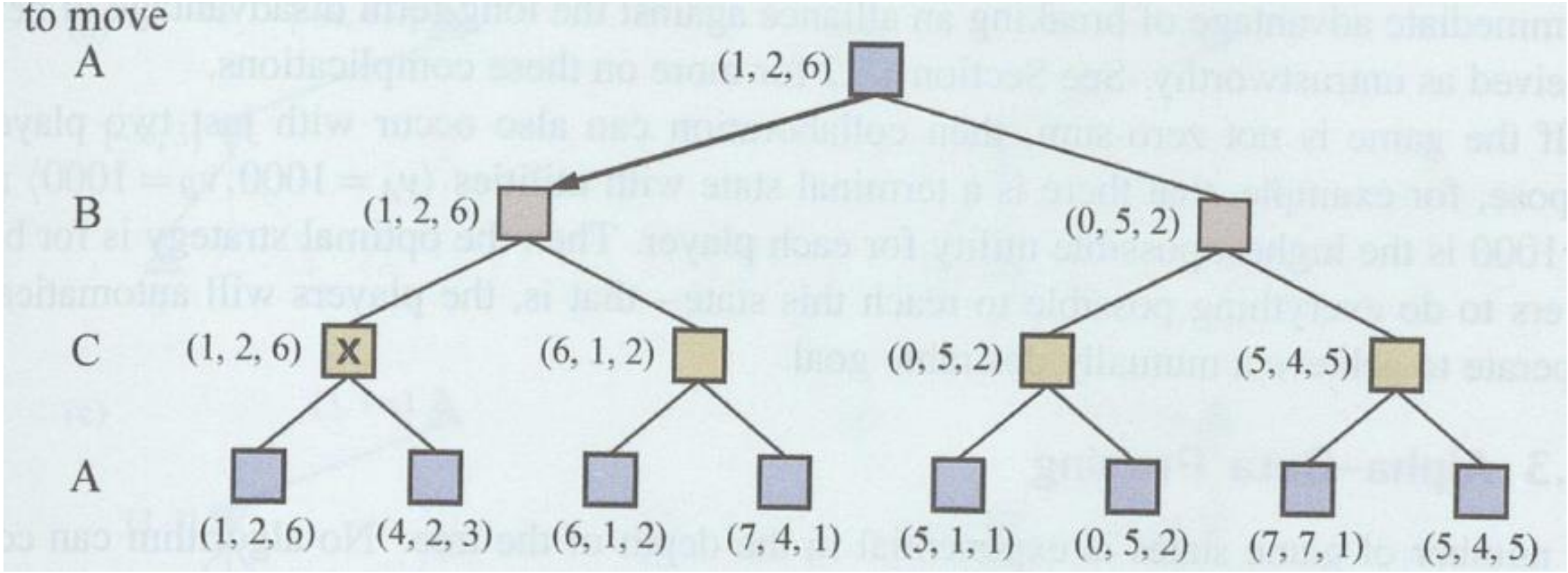
Max selects the move B with the highest utility from its three options (3, 2, 2)

Min then selects the leftmost move with the lowest utility from its three option (3, 12, 8)

Optimal Decisions in Multiplayer Games

- From technical viewpoint, a simple extension to minimax is sufficient.
- From strategic viewpoint, new considerations are necessary.

Technical Extension of Multiplayer Minimax



- Single utility value for each node is replaced by a vector of values for each player
- Each player chooses the move with its highest utility
- Example of node marked with an X:
  - C chooses between two moves with ratings 6 and 3 (leftmost node)
  - B chooses between 2 and 1 prefering 2 and A chooses between 1 and 0 prefering 1 (bold arrow)

Strategic Considerations in Multiplayer Games

Players may form alliances

- Weak players usually form alliances against a strong player
  - If weak players become stronger, alliances may be broken
    - Breaking alliances might cause a social stigma
      - Difficult decision

Different considerations, if game ist not zero-sum

- Cooperation can be beneficial for some or all players

Improvement to MiniMax: Alpha-Beta Pruning

- Number of game states is exponential in the depth of the tree
- Exponent can be cut in half by alpha-beta pruning
- Core idea: Guess the best moves and show, that other moves are worse
- The better the guess, the more efficient is the technique (even random guess cuts the exponent substantially)
- Terminology:
  - Current best option along a path for Max-nodes is called **alpha-value**
  - Current best option along a path for Min-nodes is called **beta-value**
- Max can safely ignore in the current branch of the search tree all moves lower than the alpha-value
- Min can safely ignore in the current branch of the search tree all moves higher than the beta-value

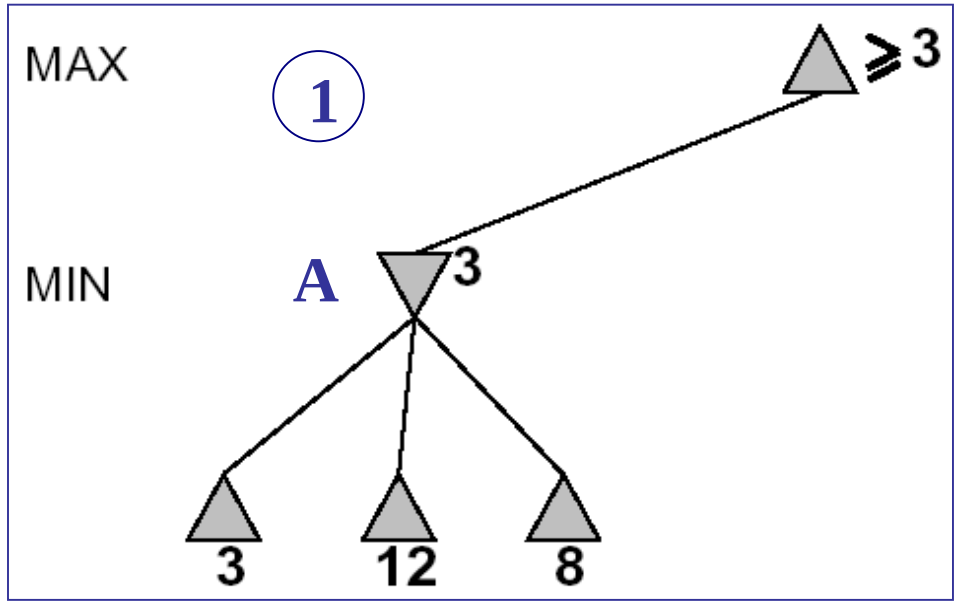
Example for Alpha-Beta Pruning

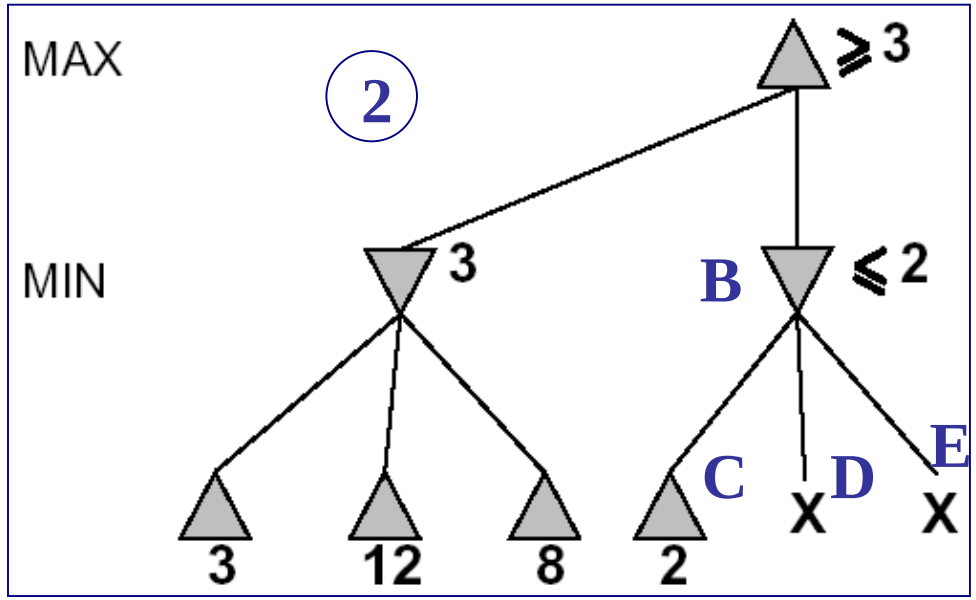
1: Value (A) = 3
1: Alpha-value (root) = 3

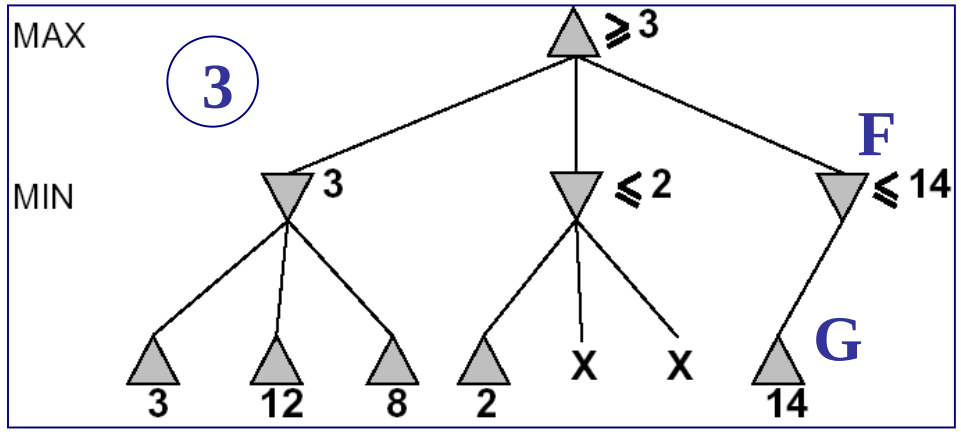
2: Value (B) ≤ 2
because value (C) = 2
value (D) irrelevant
value (E) irrelevant

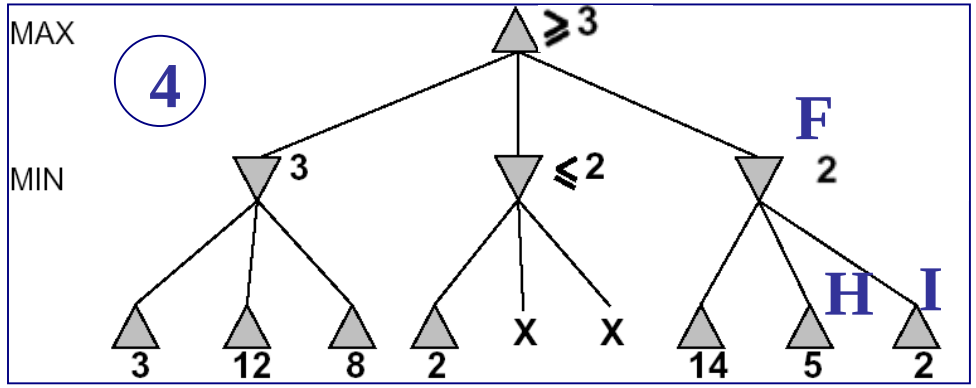
3: Value (F) ≤ 14 ≥ 3
because value (G) = 14

4: Value (F) = 2
because value (I) = 2

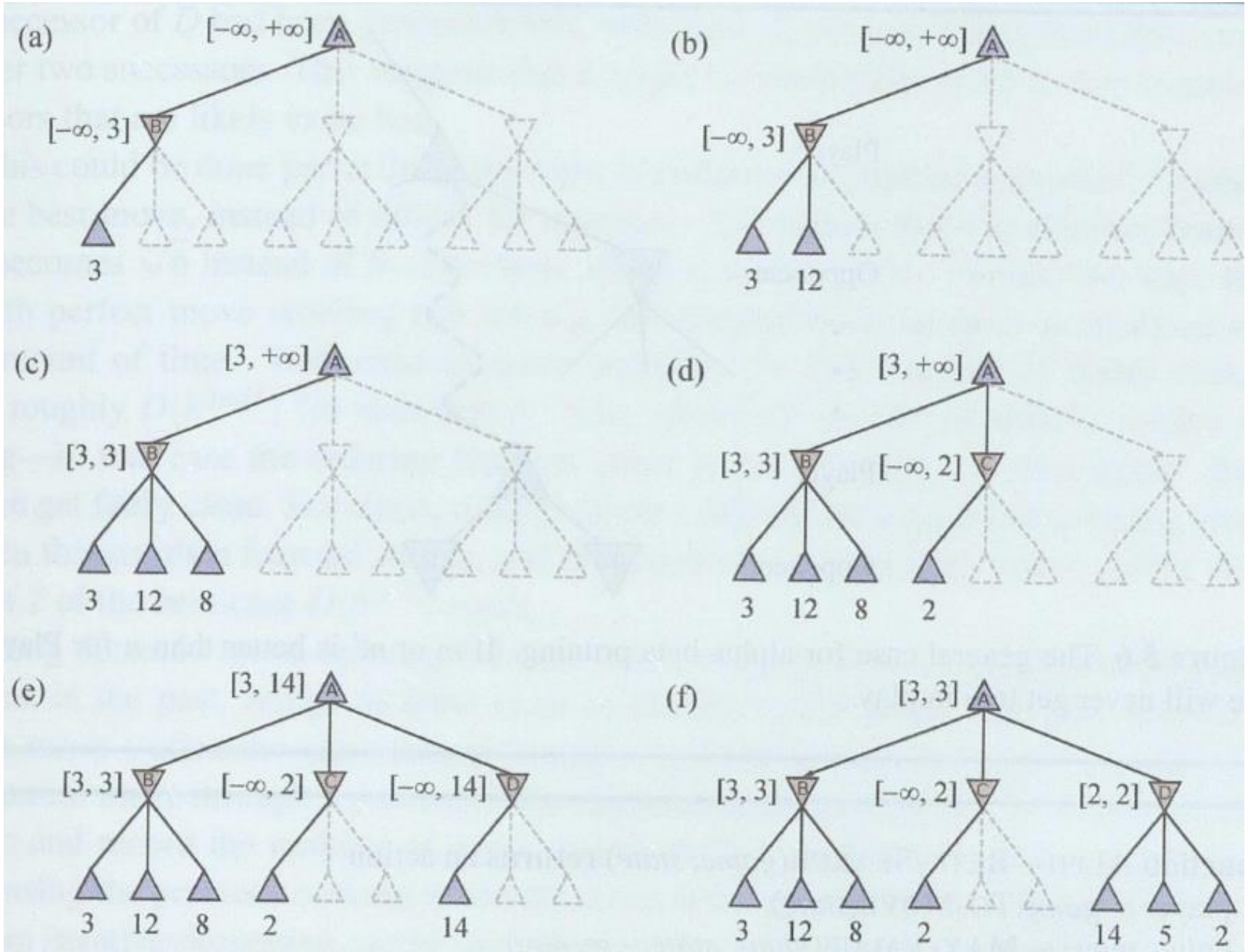








Example with more details



Improvement of Minimax by Alpha-Beta Cut

Store in each node current best value for Max: α (init: $-\infty$)

Store in each node current best value for Min: β (init: $+\infty$)

Let α the value of node X

Let β the value of node X_k

Let X_1 .. X_n the sucessors of X, from which Max chooses

Let X_{k1} .. X_{km} the sucessors of X_k, from which Min chooses

Let X_{kj1} .. X_{kjo} the sucessors of X_{kj}, from which Max chooses

If Eval (X_{kj}) < α, then cut search branch for X_k

If Eval (X_k) > α, then set α in X as Eval (X_k)

If Eval (X_{kji}) > β, then cut search branch for X_{kj}

If Eval (X_{kj}) < β, then set β in X_k as Eval (X_{kj})

Efficiency of Alpha-Beta Cut

- If always the best move is guessed, the exponent of Minimax decreases from $O(b^d)$ to $O(b^{d/2})$
- With random selection, the exponent of Minimax decreases from $O(b^d)$ to $O(b^{3d/4})$

- With simple heuristics the optimal situation can be approximated:
  - Use iterative deepening search and
  - Try the best move from the last iteration first in the next iteration

Further Improvements

- In many games, there are far more paths than positions
  - ➢ Store positions in a tansposition table and check for newly generated positions, whether they are known
- In most games, a search for terminal nodes is impossible, even with alpha-beta pruning
  - Use a heuristic evaluation function for the „end“-positions in search tree
  - The deeper the search tree, the more reliable is the evaluation function

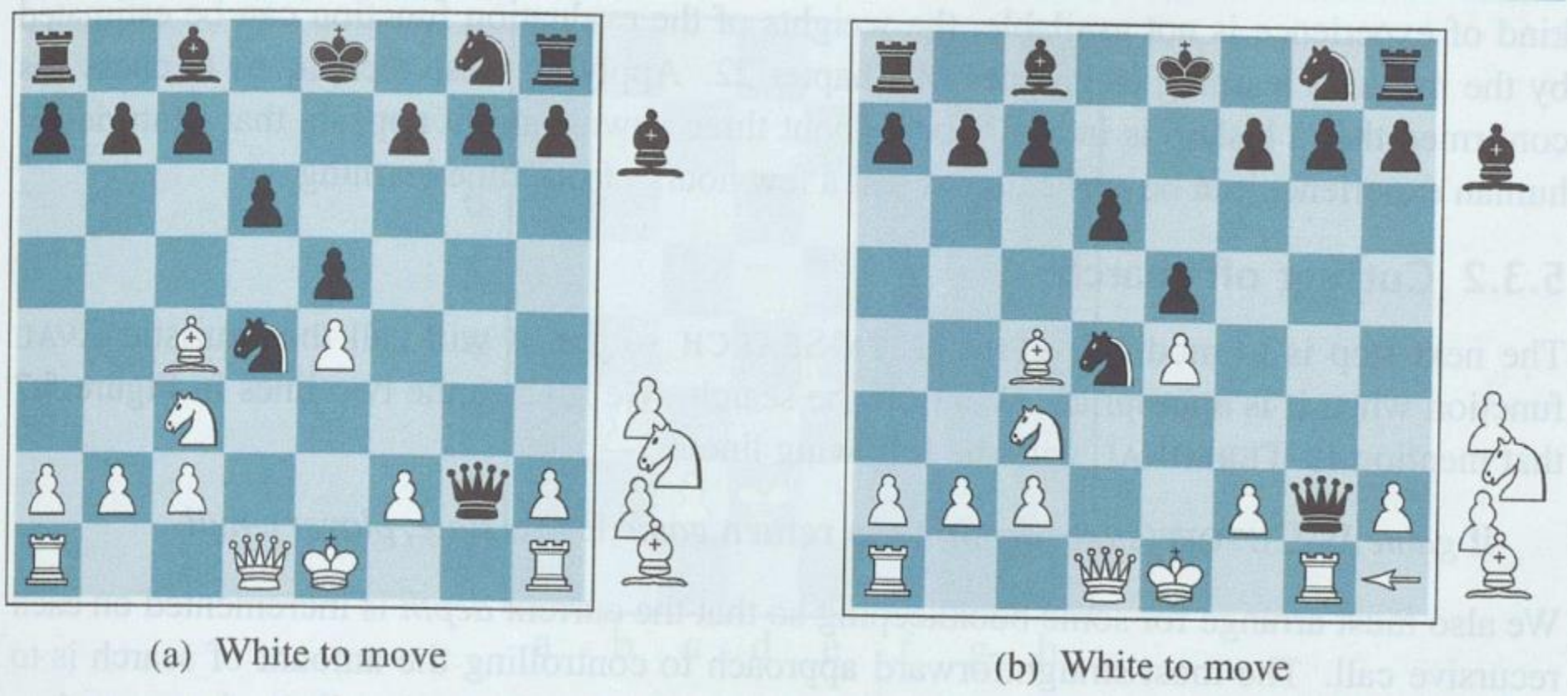
- Use different search strategies:
  - Type A: Search all possible moves to a certain depth (improvement „quiescence search“)
  - Type B: Ignore moves that look bad and follow promising lines „as far as possible“.

Heuristic Evaluation Functions for Positions

- Evaluation result describe a chance for winning
- Conflicting requirements:
  - Correct
  - Fast

Type of evaluation functions

- Usually weighted linear functions are used:
  - Eval (s) = $w_1f_1(s) + w_2f_2(s) + \ldots w_nf_n(s) = \sum_{i=1}^{n} w_i f_i(s)$
  - w = weights, f = features of state (s)



Cutting off Search

- Relevant for type A strategies (complete search)!
- Dynamic cut-off by iterative deepening
- The evaluation function should be applied only to position that are quiet (not to position b in last slide)
  - **Quiescence search** extents search for certain types of moves (in chess e.g. capture moves, check moves etc.)
- **Horizon effect:**
  - An inevitable bad move of the opponent cannot be avoided but postoned beyond the search horizon (see right)
  - Possible solution: **singular extension:** extend search by moves that are „clearly better“ than other moves

from: https://www.chessprogramming.org/Horizon_Effect

Forward Pruning

- Prune moves, that appear to be poor moves at the risk, that they might turn out to be good
  - Type B strategy (selective search)
  - Most human chess players do this
  - Kind of beam search strategy
- Can be combined with alpha-beta search
  - ProbCut (1995) is a forward pruning version of alpha-beta search, that uses statistics gained from prior experience to lessen the chance that the best move will be pruned

Search versus Lookup

- Transposition tables avoid generating the same position several times
- For certain types of situations, complete search is possible or background knowledge is available and the optimal results can be stored
  - e.g. in chess endgames and openings
  - For some positions types, simple rules for optimal play can be defined (e.g. King, Bishop, and Knight vs King)

Monte Carlo Tree Search (MTCS)

- The game of Go illustrates two major weaknesses of heuristic alpha-beta tree search:
  - Go has a huge branching factor starting with 361 (19*19) resulting in a search depth of only 4 to 5 ply (as opposed to chess with about 14 ply or move-actions)
  - It is difficult to define a good evaluation function for GO, because material value is not a strong indicator
- Alternative strategy: Monte Carlo Tree Search
  - Does not use a heuristic evaluation function
  - Takes average utility of a number of simulations (**playouts**) for estimating value of a state
    - Simulation chooses moves for each player till a terminal state is reached
      - Some strategy for biasing appearently good moves necessary („**playout policy**“)
        - Can been **learned by self play with neural networks**
        - Also game-specific heuristics possible

Further aspects of MCTS

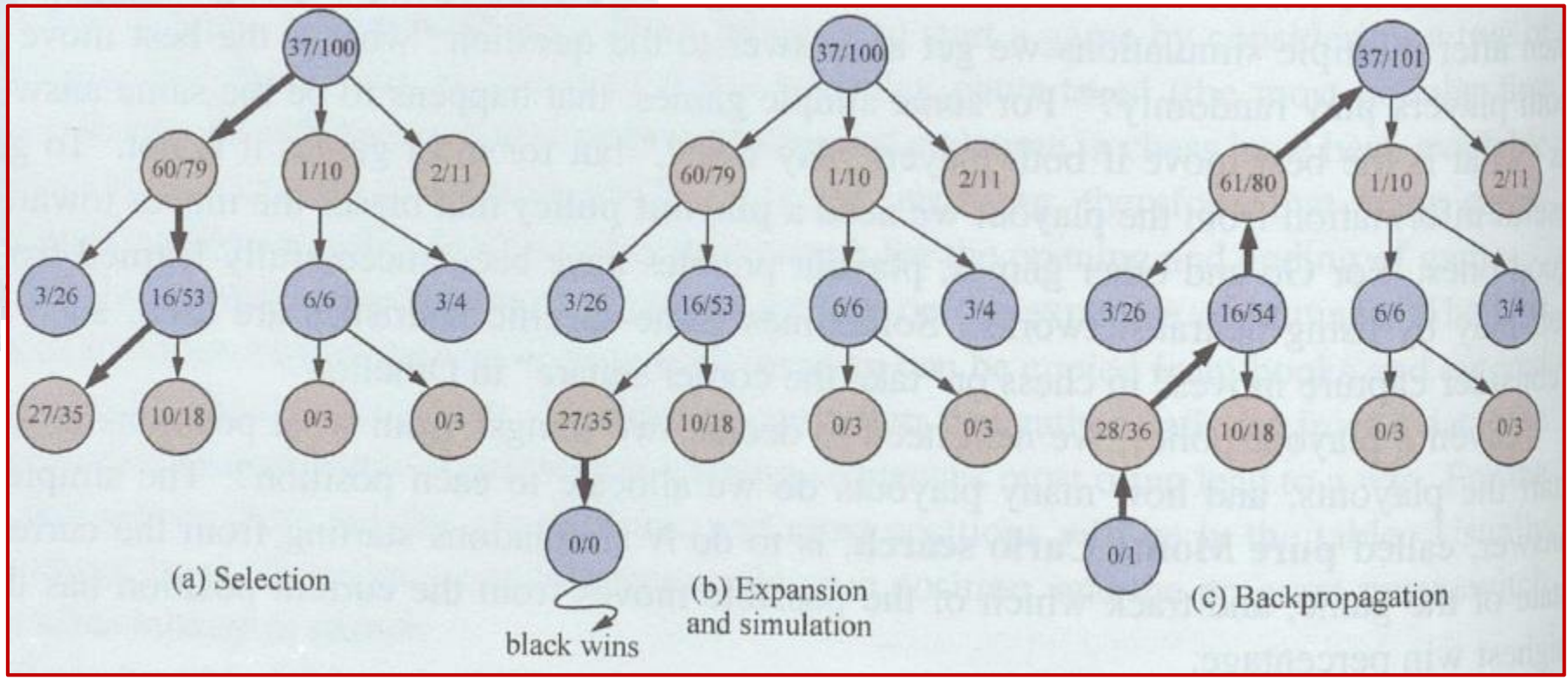
- From what position should a playout be started?
- How many playouts should be allocated to each position?

- Pure Monte Carlo Search: no strategy
  - N playouts from the current position for each move and choosing the best with highest win percentage
- Selection policy: focusing the resources on the important parts in the game tree
  - Balances two factors:
    - Exploration of states with few playouts
    - Exploitation of promising states to get a more accurate estimate of their value

Algorithmic Steps in MCTS

- **Selection:** Select move in search tree guided by selection policy (bold arrow in (a))
- **Expansion:** Grow search tree by generating new child of selected node (node „0/0“ in (b))
- **Simulation:** Full playout guided by playout policy („Black wins“ in (b))
- **Back-Propagation:** Update all the search nodes going up to the root (bold arrows in (c))

```
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

- The selection policy should balance
  - exploitation: utility of a node n: U(n) and
  - exploration: knowledge about a node, i.e. number of visits of a node: N(n)
- Popular formular is the upper confidence bound formula UCB1:

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

To avoid dividing by 0, a 1 is added in the denominator

- First term: Prefers nodes with high utility
- Second term with square root: prefers nodes with low frequency and frequent parents
- Constant C balances both terms, typically $\sqrt{2}$ or higher

Reflections (1)

- Advantages of Monte Carlo search over alpha-beta search:
  - Suited for all games (even with high branching factor and bad evaluation functions)
  - Needs no external knowledge about evalution function
- Disadvantages:
  - Might fail to consider a single very good move (game changer) due to its stochastic nature
  - Cannot recognize an „obvious“ win for one player
- Combinations possible

Reflections (2)

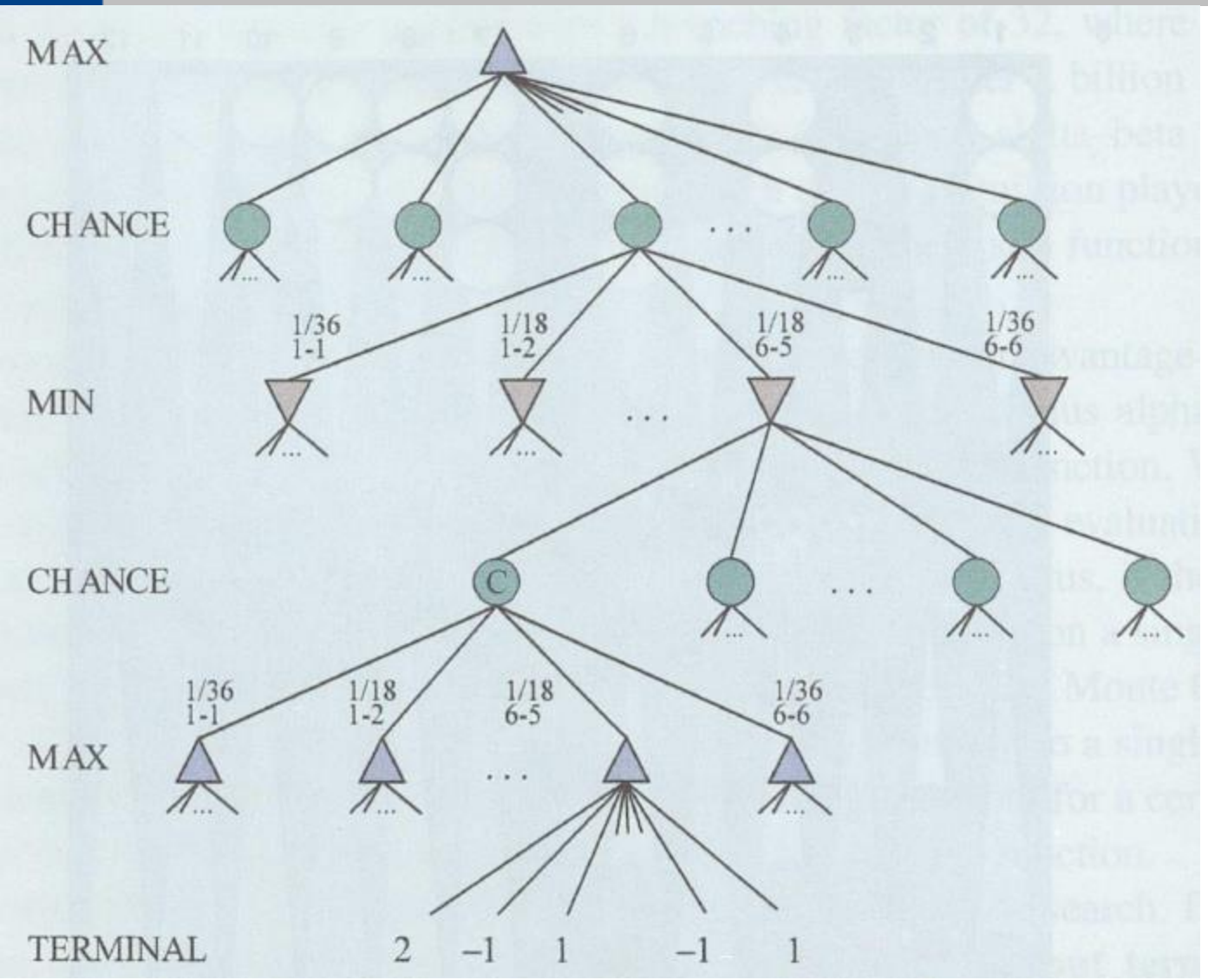
- But Monte Carlo approaches have recently also succeeded in games like chess being dominated by alpha-beta.
  - Due to neural networks for getting heuristics by self play
  - AlphaGo used two neural networks:
    - Policy network: predicts next move in order to narrow down the search by focusing on good moves
    - Value network: Estimates the winner in each position in order to avoid a full play-out

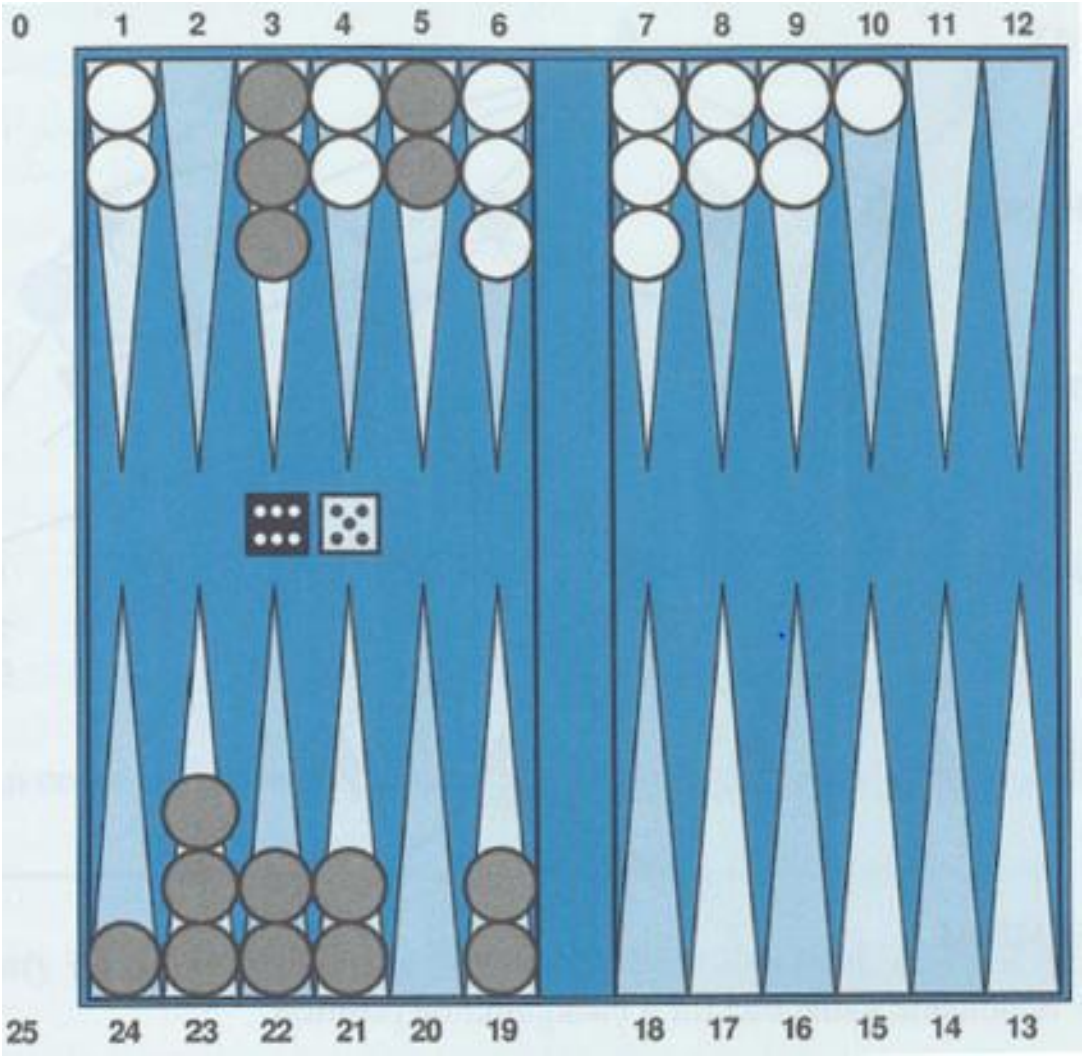
- Monte Carlo Search might be viewed as instance of reinforcement learning

Stochastic Games

- Closer to unpredictability of real life by including a random element, e.g. throwing a dice
- Utility value of a node depends on chance
  - Inclusion of chance nodes
  - Expected value of a position is average over all possible outcomes of the chance nodes
- Generalization of Minimax to Expectiminimax algorithm
  - Introduction of chance nodes
  - Their expected values is the sum of all possible outcomes, weighted by the probability of each chance action

Example: Search Tree in Backgammon





Computation of Expected Values

$$\text{EXPECTIMINIMAX}(s) = \begin{cases} \text{UTILITY}(s, \text{MAX}) & \text{if IS-TERMINAL}(s) \\ \max_a \text{EXPECTIMINIMAX}(\text{RESULT}(s, a)) & \text{if TO-MOVE}(s) = \text{MAX} \\ \min_a \text{EXPECTIMINIMAX}(\text{RESULT}(s, a)) & \text{if TO-MOVE}(s) = \text{MIN} \\ \sum_r P(r)\text{EXPECTIMINIMAX}(\text{RESULT}(s, r)) & \text{if TO-MOVE}(s) = \text{CHANCE} \end{cases}$$

s = state
a = action
r = possible dice roll or other chance event
Result (s, r) = the result of a state s given a certain dice roll r

- Expected value should represent a **chance of winning**, not an arbitrary number (which was sufficient in Minimax algorithm).

Algorithms for Stochastic Games

- Transfer from Minimax to Expectiminimas straightforward (with different utility function, see last slide)
  - Problem: only small lookahead possible due to chance nodes (e.g. in Backgammon 3 ply)
- Alpha-Beta-Pruning not very effective
  - Idea is, that many branches can be pruned with little effort
  - Due to chance, pruning is rarely possible and takes more computational effort
- **Monte Carlo Search** usually best algorithm
  - No substantial change of basic algorithm necessary

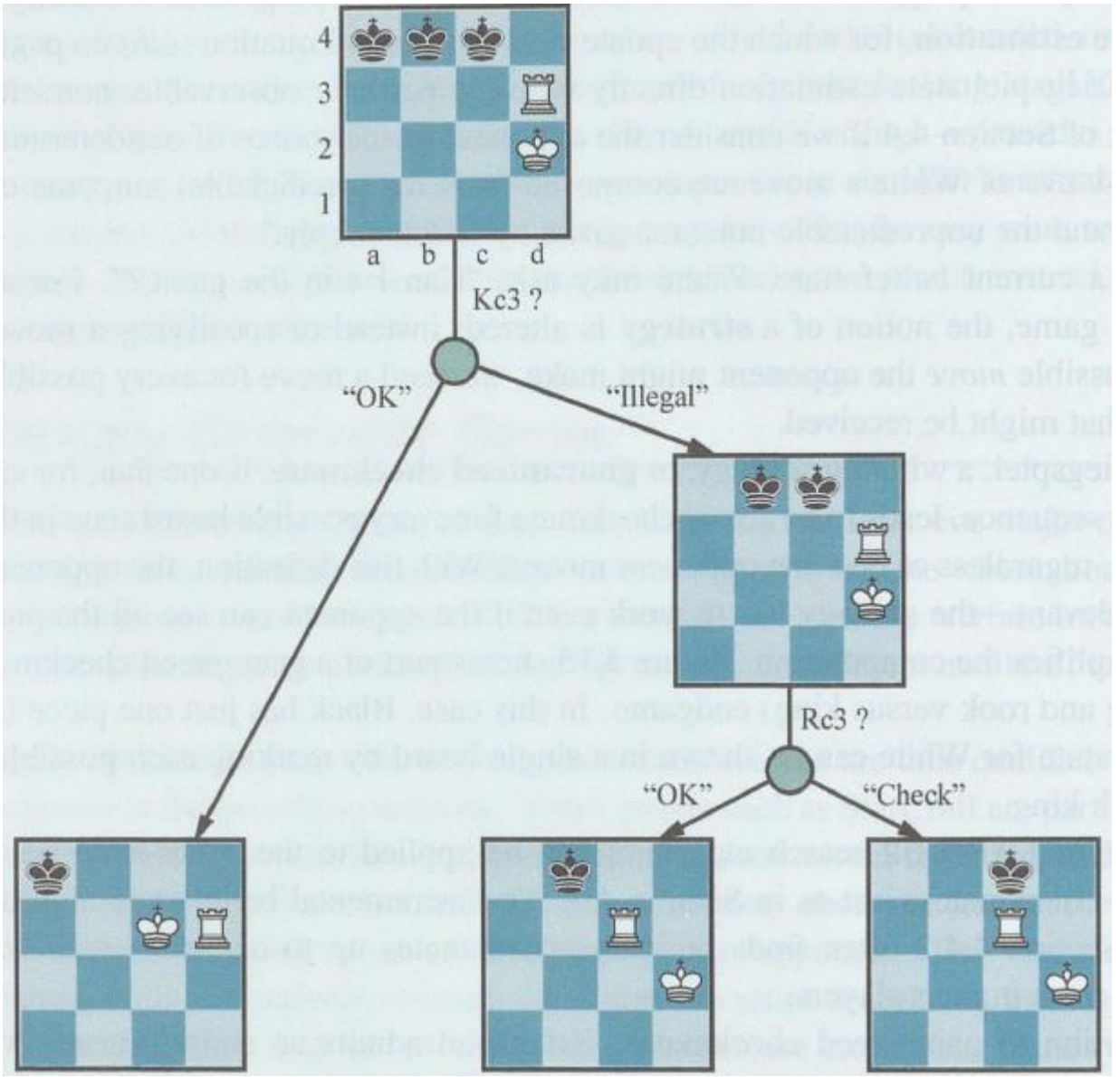
Partial Observable Games

- Closer to unpredictability of real life by dealing with the problem of information gain to reduce chance
- Examples:
  - Board games like Battleship, Stratego, **Kriegsspiel** (chess without seeing the opposite‘s pieces)
  - Card games like brigde, poker, skat, schafskopf, doppelkopf (stochastic and partial observability, where missing information is generated once at the beginning by random dealing of cards)
  - Video games like StarCraft (partial observable, non-deterministic, multi-agent, dynamic, unknown)

Kriegsspiel

- Goal: Training of officers by imitating real battles with figures corresponding to available troops and their abilities.
- Simplified: chess without seeing the opposite‘s pieces
- Moves are told to a referee, who replies whether the move is possible und the consequences like check, check mate, stalemate, capture of figures.

- Can be modelled with And-Or-Graph (right)
  - Additional nodes for each „percept“

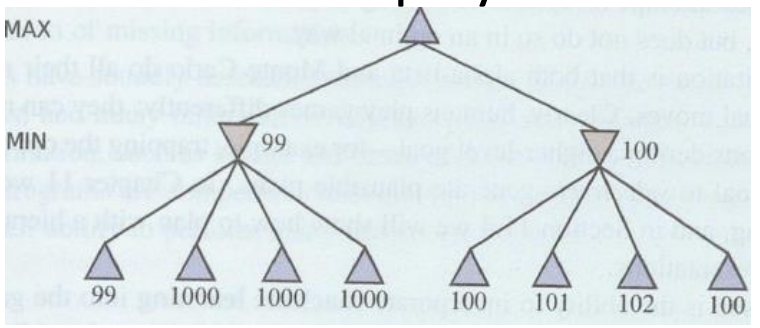


Strategies

- Explicit representation of information hiding and information gain necessary
- Kriegsspiel more difficult than card games, because in card games, the initial card distribution is unknown, but all tricks are visible, while in Kriegsspiel, also the moves of the opponent are (mostly) unknown.
- Belief state is helpful (set of all logically possible board states given the complete history of percepts)
  - Also explicit representation of belief state of opponent helpful
- Abstraction necessary to reduce the size of the belief state
- Search algorithms with adaptions still possible
- Example programm: Libratus for Poker
  - Reached super human performance (2017; no deep learning)
  - Tremendous resources consumption: 25 million CPU hours on a supercomputer

Limitations of Game Search Algorithms

- All algorithms must make assumptions and approximations
  - Alpha-Beta search uses heuristic evaluation function
  - Monte Carlo search computes average over a random or guided selection of playouts
- General limitations:
  - (Alpha-Beta only): vulnerability to errors in heuristic function
    - Extension: take standard deviations into account
  - Designed to calculate bounds on the value for legal moves
    - Unnessary, if there is only one move or one move is clearly the best or with symmetrical moves
    - Extension: Compute utility of node expansion first
      - Example for **Meta-reasoning**
  - Designed to do all reasoning on the level of individual moves
    - Extension: **Planning** on higher abstraction level (e.g. capturing the opponents queen)
  - Incorperation of **Machine Learning** (already done in Monte Carlo for playout-strategy)



State of the Art

- **Chess**
  - Deep Blue defeated world champion Kasparov 1997 with alpha-beta search
  - Current best programs (Stockfish, Komodo, Houdini) far exceed any human player
  - AlphaZero (learning from selfplay) defeated Stockfish (more traditional) 2018
- **GO**
  - AlphaGo defeated the world champion Lee Sedoal 2015
  - AlphaZero surpassed AlphaGo 2018
- **Poker** (similar for other card games):
  - Libratus beat champion poker players (2017)
  - Pluribus, AlphaZero even better
- **Video Games like StarCraft2**
  - AlphaStar defeated expert human players 10:1 (2019)
- **Physical Games liker soccer robot** much more difficult (currently no breakthrough)