Foundations of Artificial Intelligence

6. Board Games

Search Strategies for Games, Games with Chance, State of the Art

Joschka Boedecker and Wolfram Burgard and
Frank Hutter and Bernhard Nebel

Albert-Ludwigs-Universität Freiburg

May 09, 2018

Contents

Why Board Games?

Playing board games is one of the oldest research areas in AI (Shannon and Turing 1950).

- Board games present a very abstract and pure form of competition between two opponents and clearly require a form of "intelligence".
- The states of a game are easy to represent.
- The possible actions of the players are well-defined.

$\rightarrow$ Implementation of the game as a kind of search problem

$\rightarrow$ The individual states are fully accessible

$\rightarrow$ It is nonetheless a contingency problem, because the actions of the opponent are not under the control of the player

Problems

Board games are not only difficult because they are contingency problems, but also because the state space can become astronomically large.

Examples:

- **Chess**: On average 35 possible actions from every position; often, games have 50 moves per player, resulting in a search depth of 100: $\rightarrow 35^{100} \approx 10^{150}$ nodes in the search tree (with "only" 10^{40} legal chess positions).
- **Go**: On average 200 possible actions with ca. 300 moves $\rightarrow 200^{300} \approx 10^{700}$ nodes.

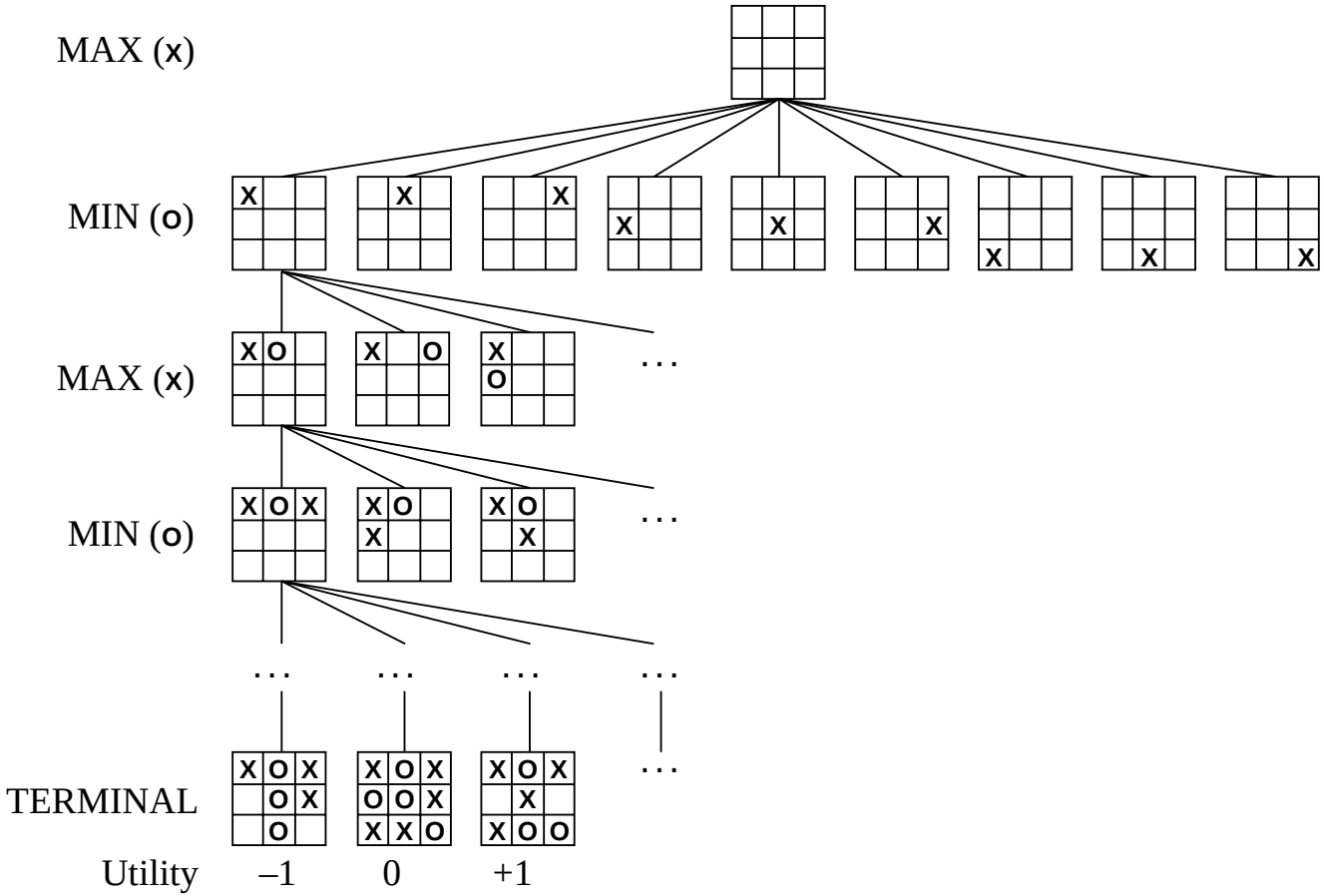
Good game programs have the properties that they

- delete irrelevant branches of the tree,
- use good evaluation functions for in-between states, and
- look ahead as many moves as possible.

Terminology of Two-Person Board Games

- Players are MAX and MIN, where MAX begins.
- Initial position (e.g., board arrangement)
- Operators (= legal moves)
- Game tree is the search tree generated from the possible (alternate) moves
- Termination test, determines when the game is over and what the value of the final state is
- Strategy. In contrast to regular searches, where a path from beginning to end is a solution, MAX must come up with a strategy to reach a favorable terminal state *regardless of what* MIN *does* $\rightarrow$ all of MIN's moves must be considered and reactions to them must be computed

Tic-Tac-Toe Example



Every level of the game tree is given the player's name whose turn it is (MAX- and MIN-steps).

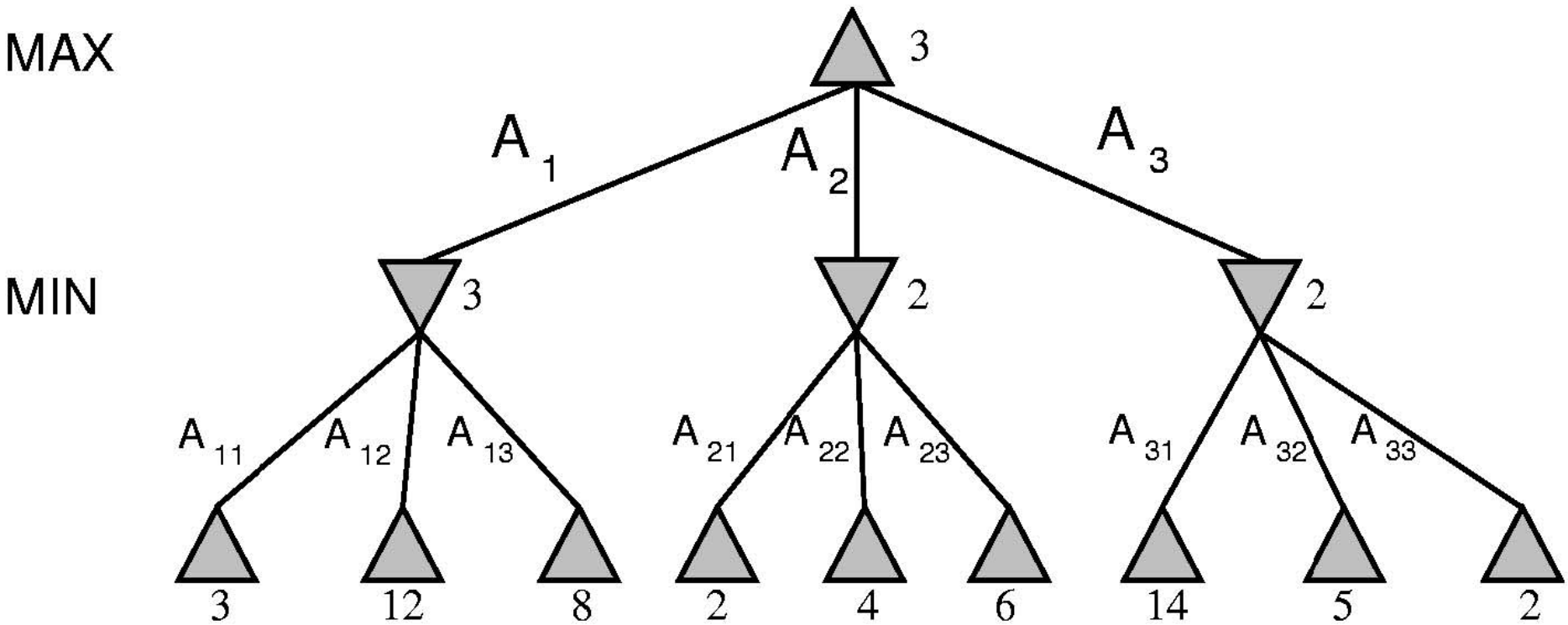
When it is possible, as it is here, to produce the full game tree, the minimax algorithm delivers an optimal strategy for MAX.

Minimax

1. Generate the complete game tree using depth-first search (we do not need the full tree in memory!)
2. Apply the utility function to each terminal state.
3. Beginning with the terminal states, determine the utility of the predecessor nodes as follows:
   - Node is a MIN-node
     Value is the minimum of the child nodes
   - Node is a MAX-node
     Value is the maximum of the child nodes
   - From the initial state (root of the game tree), MAX chooses the move that leads to the highest value (minimax decision).

Note: Minimax assumes that MIN plays perfectly. Every weakness (i.e., every mistake MIN makes) can only improve the result for MAX.

Minimax Example



Minimax Algorithm

Recursively calculates the best move from the initial state.

function MINIMAX-DECISION(*state*) **returns** *an action*
  return $\arg\max_{a \in \text{ACTIONS}(s)}$ MIN-VALUE(RESULT(*state*, *a*))

---

function MAX-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each *a* **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s, a)))
  return *v*

---

function MIN-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow \infty$
  for each *a* **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s, a)))
  return *v*

Note: Minimax can only be applied to game trees that are not too deep. Otherwise, the minimax value must be approximated at a certain level.

Evaluation Function

When the search tree is too large, it can be expanded to a certain depth only. The art is to correctly evaluate the playing position of the leaves of the tree at that depth.

Example of simple evaluation criteria in chess:

- Material value: pawn 1, knight/bishop 3, rook 5, queen 9
- Other: king safety, good pawn structure
- Rule of thumb: three-point advantage = certain victory

The design of the evaluation function is important!

The value assigned to a state of play should reflect the chances of winning, i.e., the chance of winning with a one-point advantage should be less than with a three-point advantage.

Evaluation Function—General

The preferred evaluation functions are weighted, linear functions:

$$w_1 f_1 + w_2 f_2 + \cdots + w_n f_n$$

where the w's are the weights, and the f's are the features. [e.g., $w_1 = 3$, $f_1 =$ number of our own knights on the board]

The above linear sum makes the strong assumption that the contributions of all features are independent. (not true: e.g., bishops in the endgame are more powerful, when there is more space)

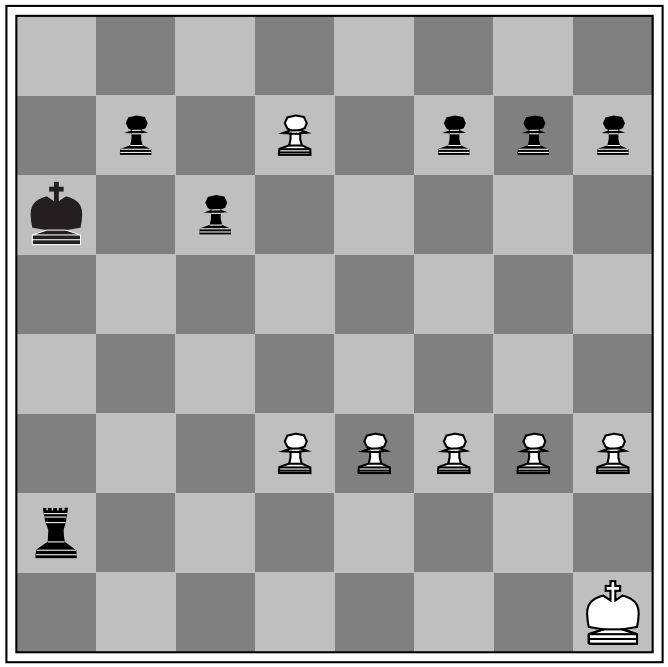
The weights can be learned. The features, however, are often designed by human intuition and understanding

When Should We Stop Growing the Tree?

Motivation: Return an answer within the allocated time.

- Fixed-depth search.
- Better: iterative deepening search (stop, when time is over).
- But only stop and evaluate in "quiescent" positions that will not cause large fluctuations in the evaluation function in the following moves. For example, if one can capture a figure, then the position is not "quiescent" because this action might change the evaluation substantially. It is better to continue the search in non-quiescent positions, preferably by only allowing certain types of moves (e.g., capturing) to reduce search effort, until a quiescent position is reached.
- There still is the problem of limited depth search: horizon effect (see next slide).

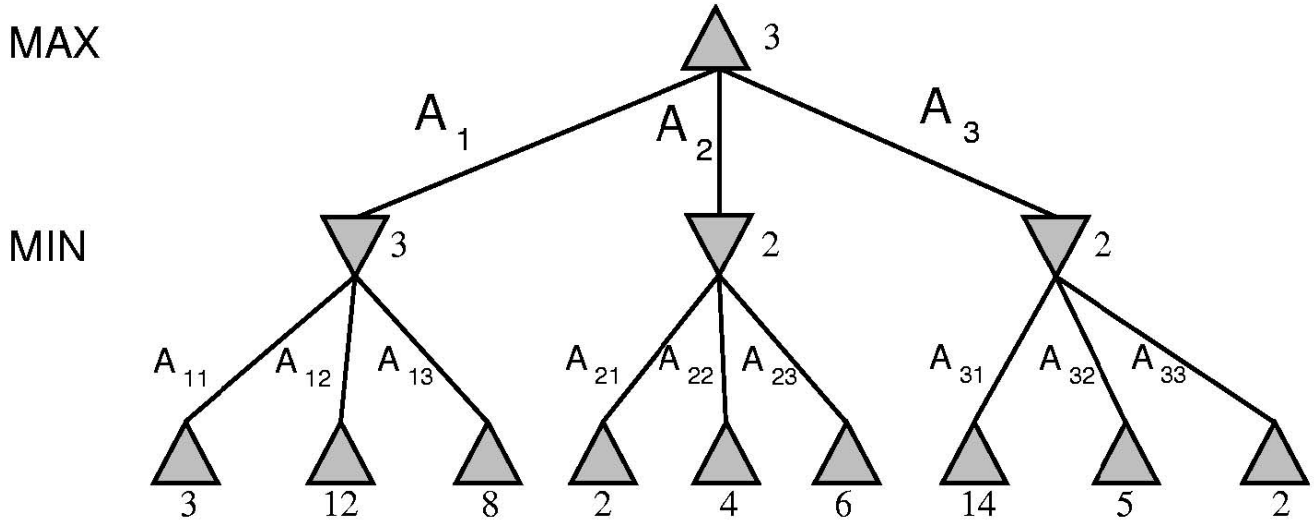
Horizon Problem



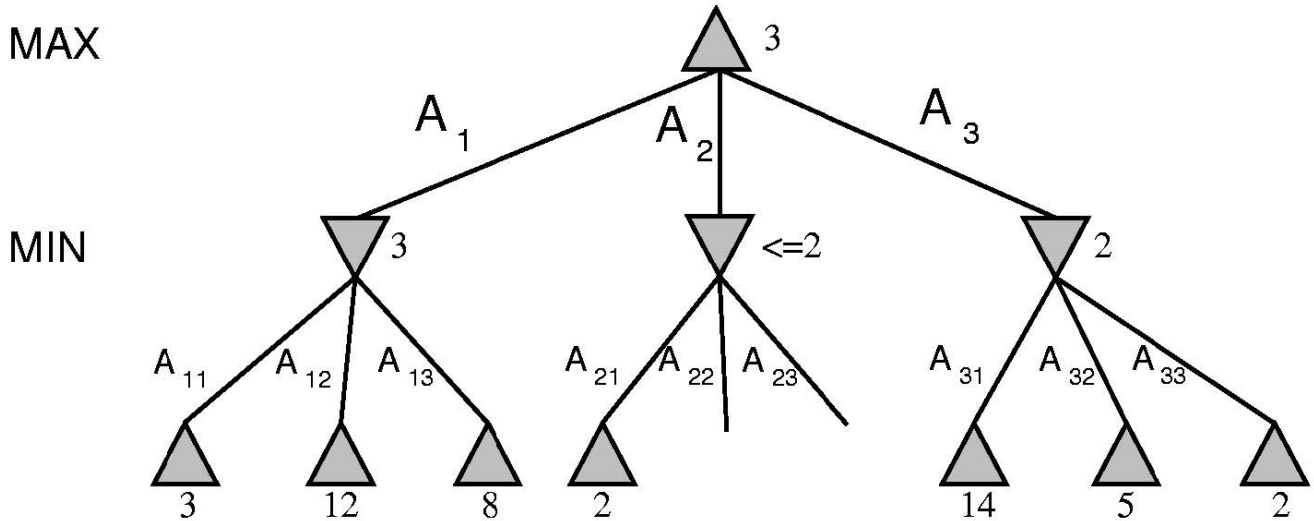
Black to move

- Black has a slight material advantage
- . . . but will eventually lose (pawn becomes a queen).
- A fixed-depth search cannot detect this because it thinks it can avoid it (on the other side of the horizon—because black is concentrating on the check with the rook, to which white must react).

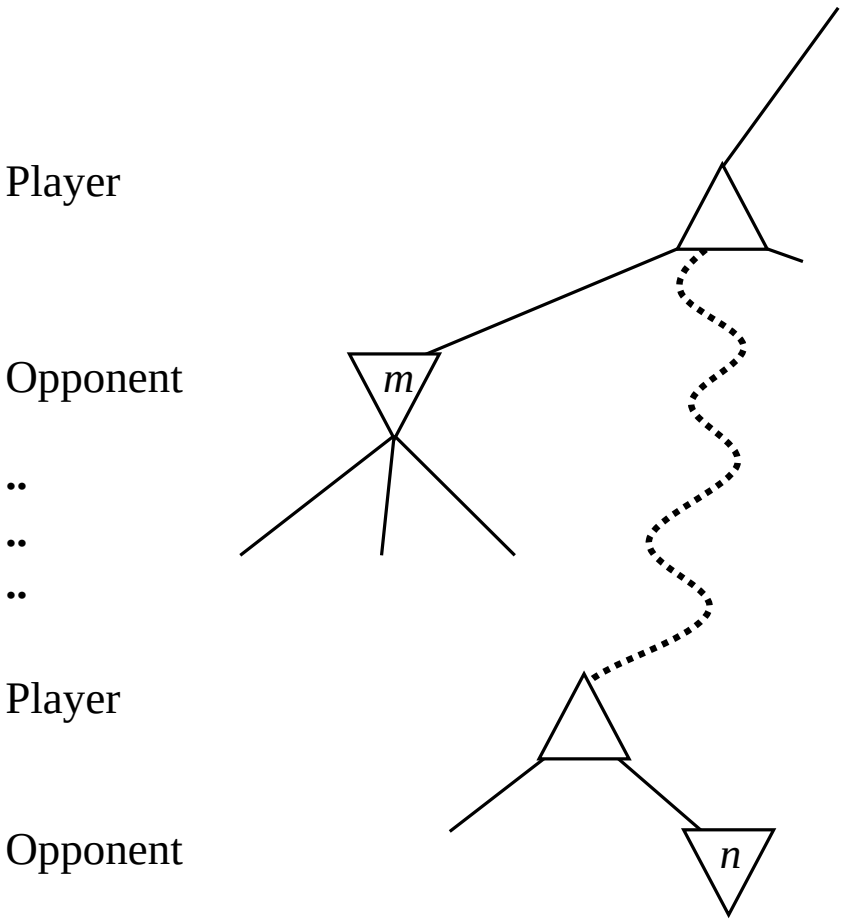
Alpha-Beta Pruning



Can we improve this? We do not need to consider all nodes.



Alpha-Beta Pruning: General



If $m > n$ we will never reach node n in the game.

Alpha-Beta Pruning

Minimax algorithm with depth-first search

α = the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.

β = the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.

When Can we Prune?

The following applies:

α values of MAX nodes can never decrease

β values of MIN nodes can never increase

(1) Prune below the MIN node whose β-bound is less than or equal to the α-bound of its MAX-predecessor node.

(2) Prune below the MAX node whose α-bound is greater than or equal to the β-bound of its MIN-predecessor node.

$\rightarrow$ Provides the same results as the complete minimax search to the same depth (because only irrelevant nodes are eliminated).

Alpha-Beta Search Algorithm

function ALPHA-BETA-SEARCH($state$) **returns** an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the $action$ in ACTIONS($state$) with value v

---

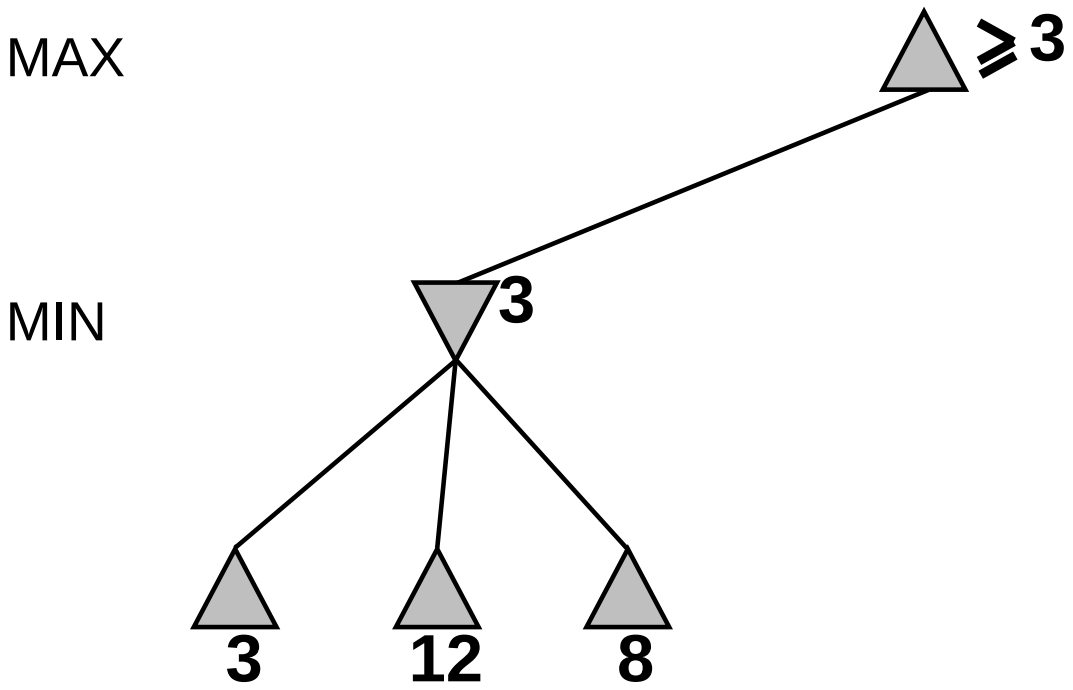
function MAX-VALUE($state, \alpha, \beta$) **returns** a $utility$ $value$
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

---

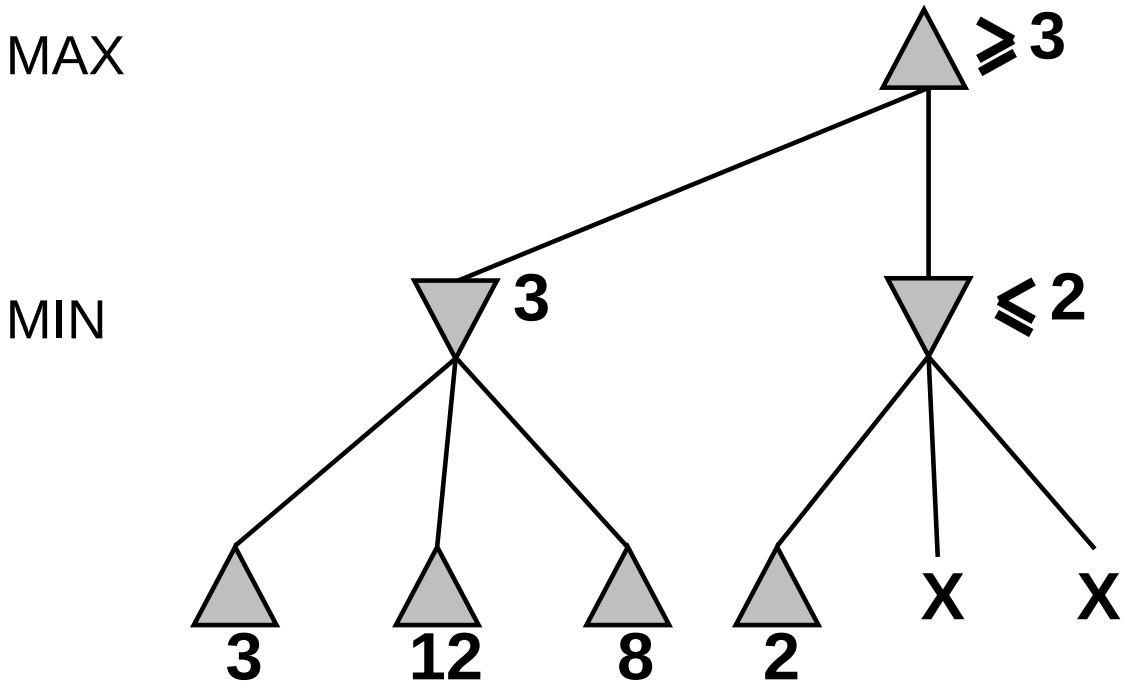
function MIN-VALUE($state, \alpha, \beta$) **returns** a $utility$ $value$
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Initial call with MAX-VALUE($initial\text{-}state$, $-\infty$, $+\infty$)

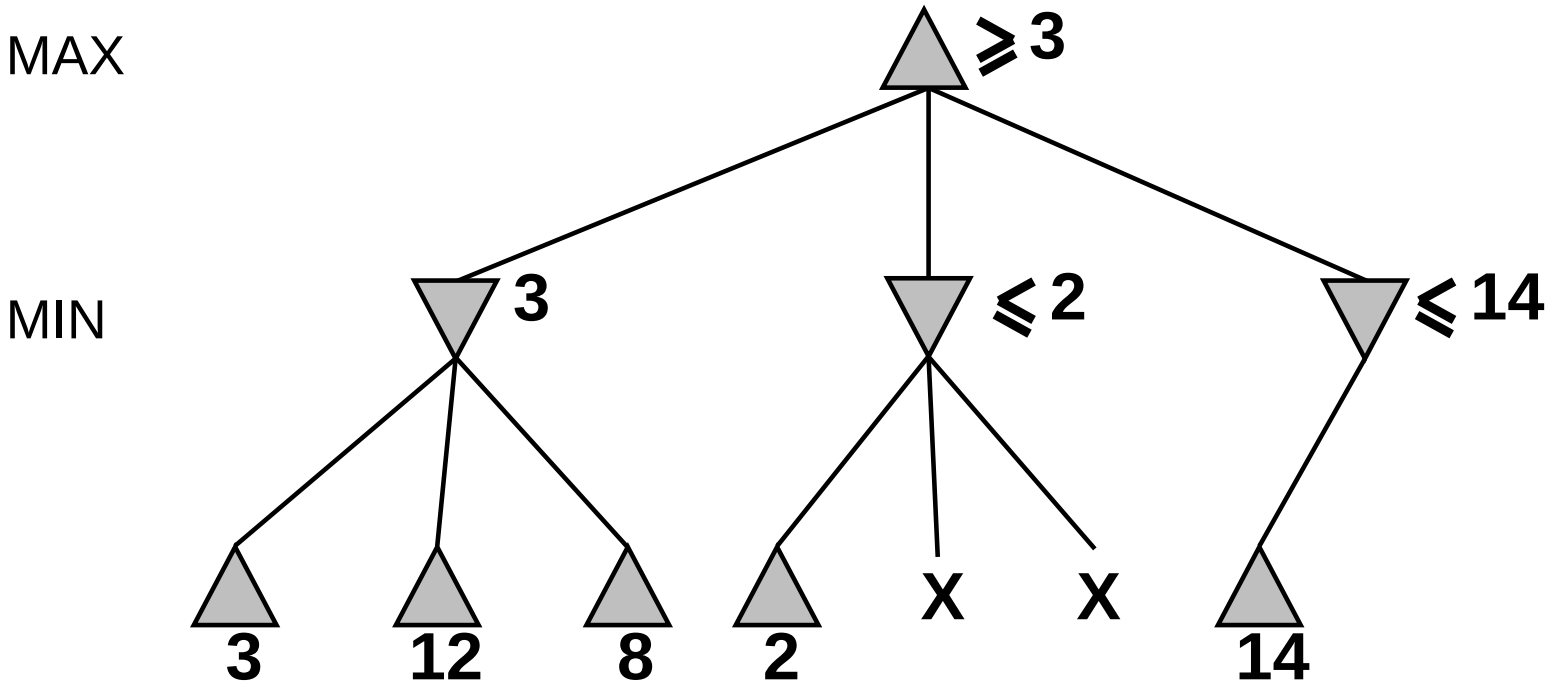
Alpha-Beta Pruning Example



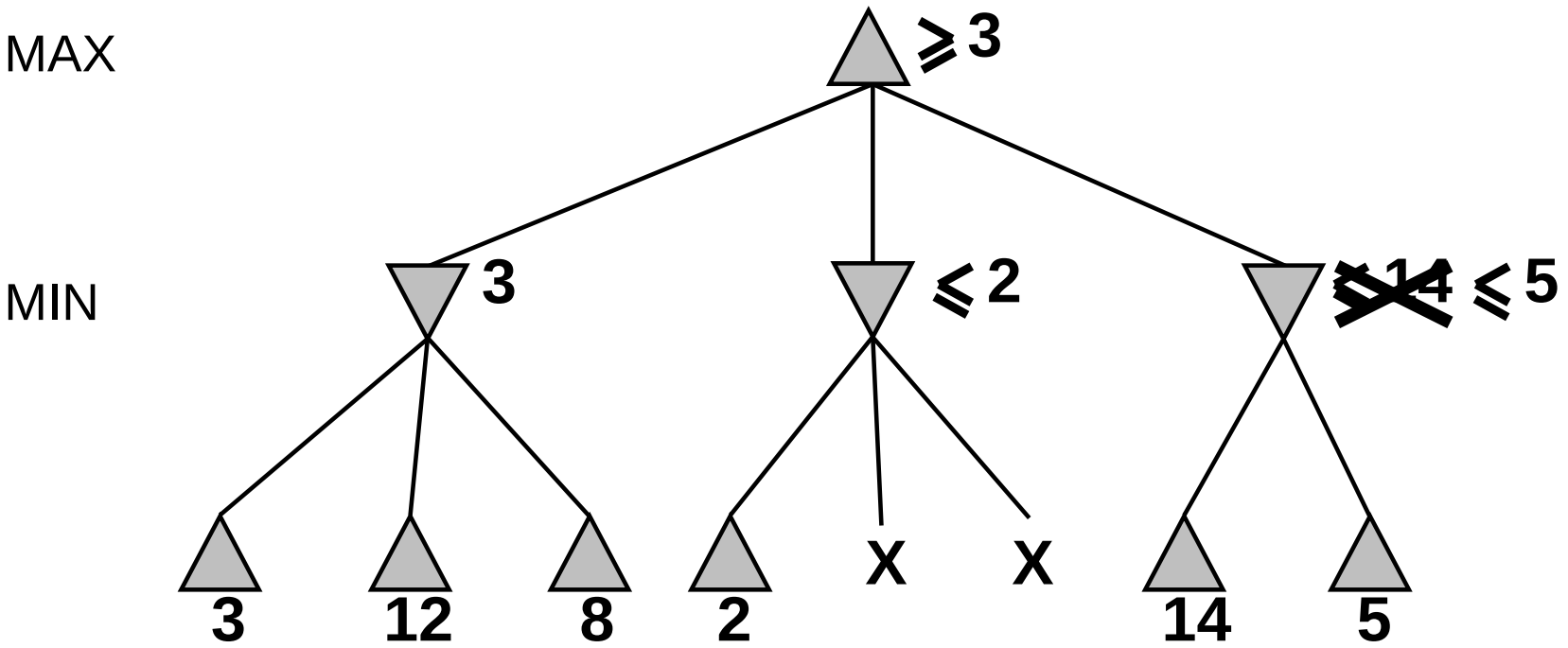
Alpha-Beta Pruning Example



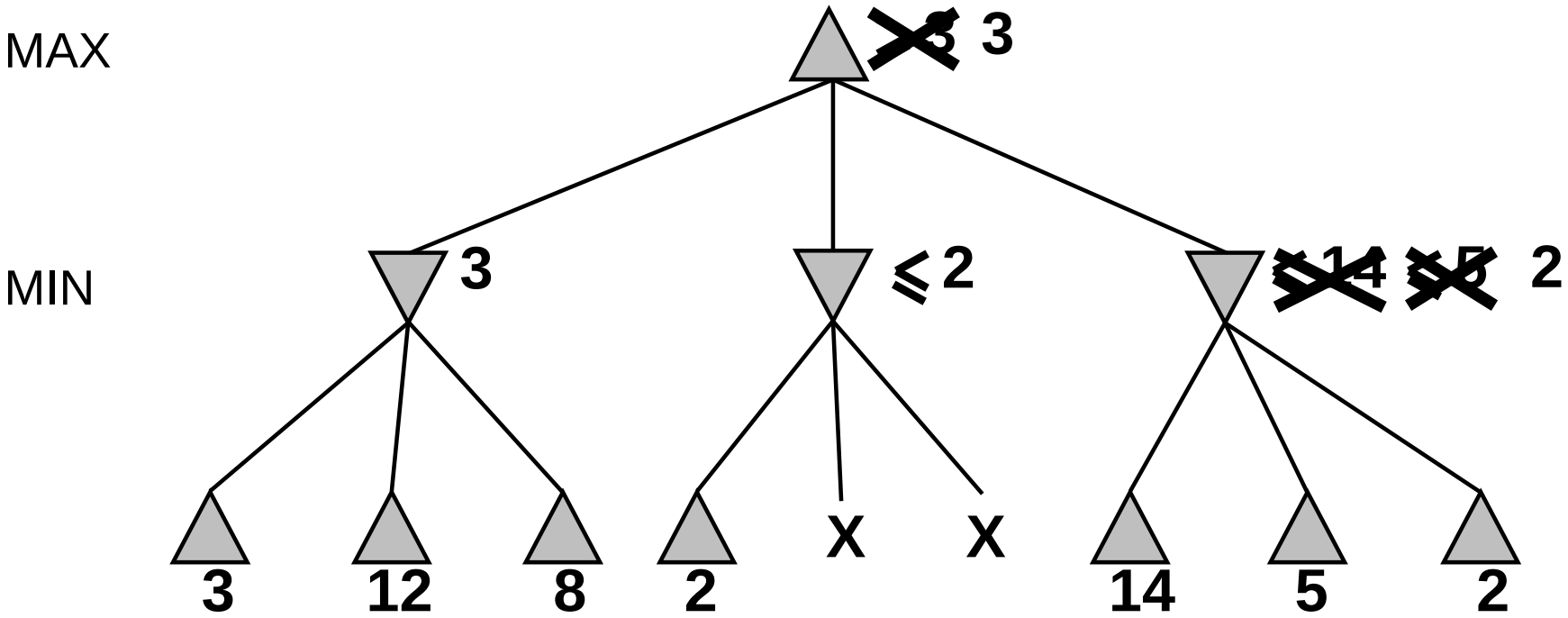
Alpha-Beta Pruning Example



Alpha-Beta Pruning Example



Alpha-Beta Pruning Example

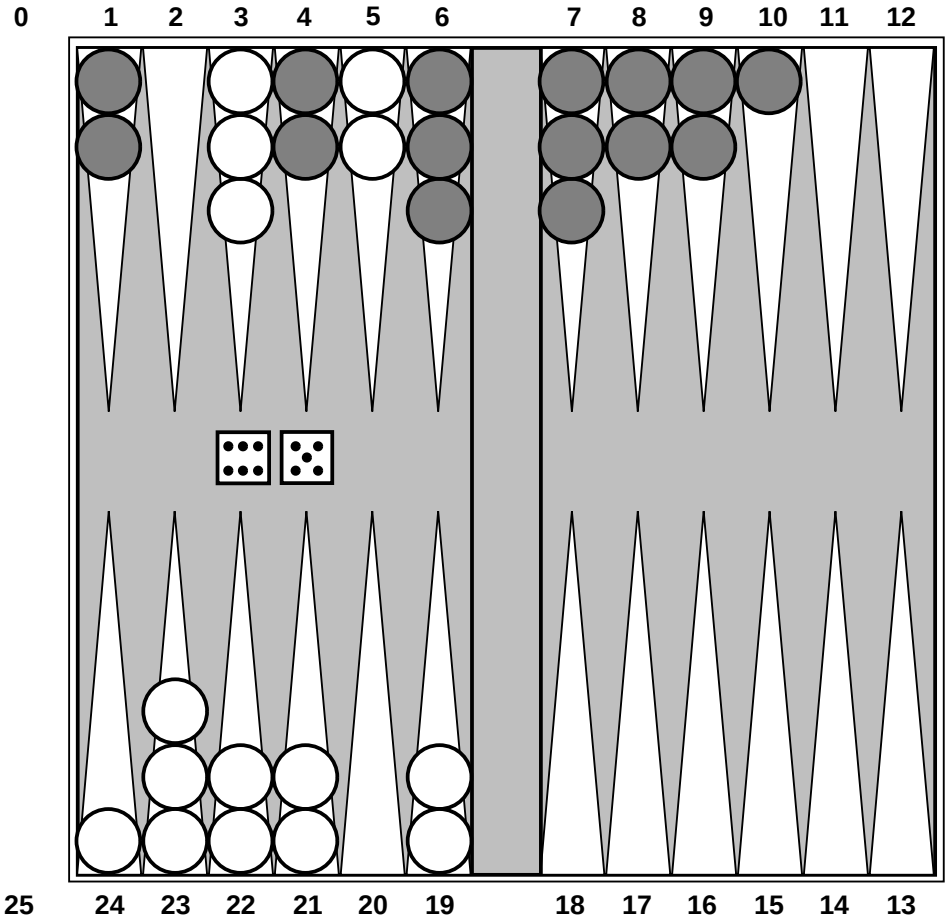


Efficiency Gain

- The alpha-beta search cuts the largest amount off the tree when we examine the best move first.
- In the best case (always the best move first), the search expenditure is reduced to $O(b^{d/2}) \Rightarrow$ we can search twice as deep in the same amount of time.
- In the average case (randomly distributed moves), for moderate b ($b < 100$), we roughly have $O(b^{3d/4})$.
- However, the best move typically is not known. Practical case: A simple ordering heuristic brings the performance close to the best case $\Rightarrow$ In chess, we can thus reach a depth of 6–7 moves.

Good ordering for chess? Try captures first, then threats, then forward moves, then backward moves.

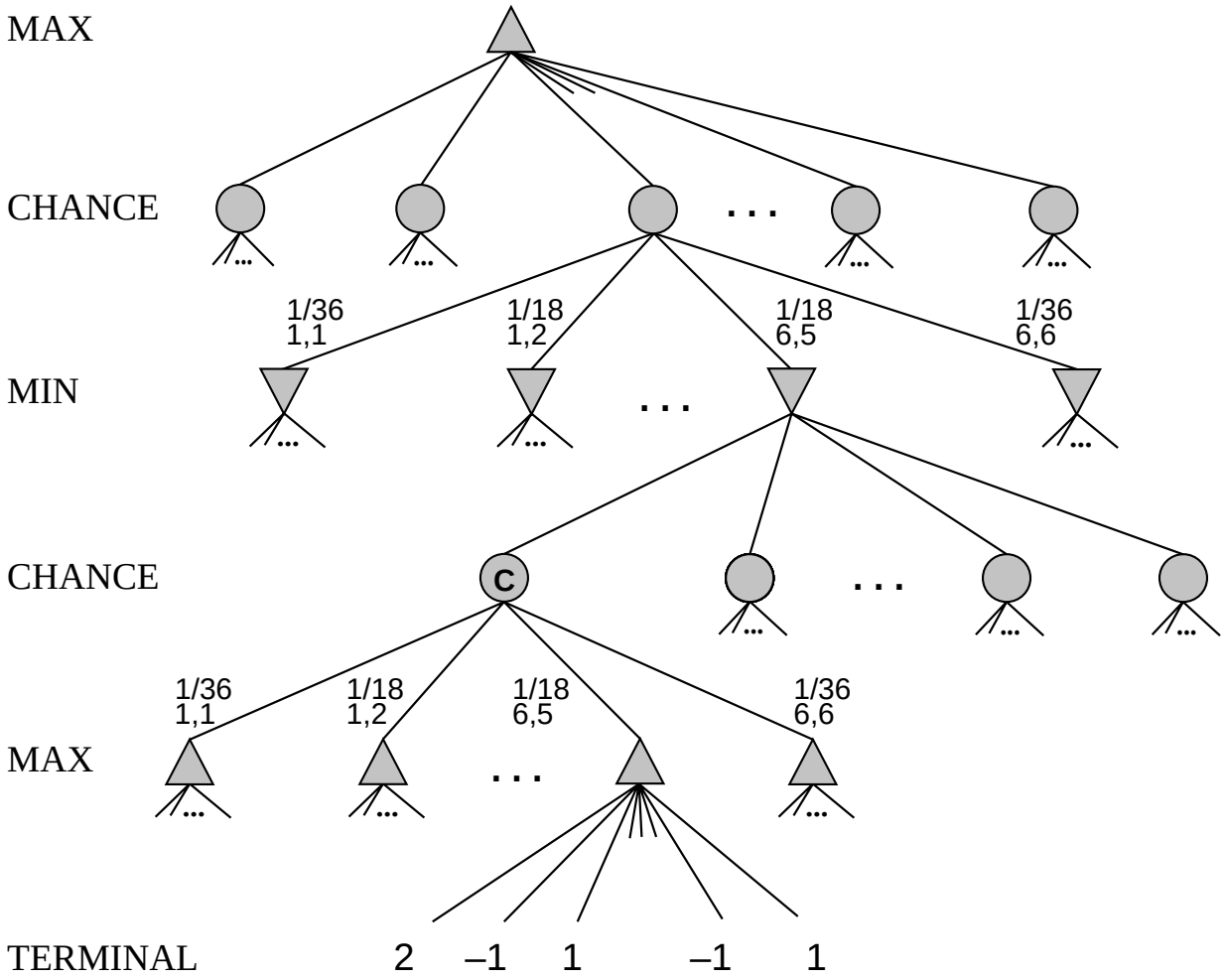
Games that Include an Element of Chance



White has just rolled a 6 and a 5 and has 4 legal moves.

Game Tree for Backgammon

In addition to MIN- and MAX nodes, we need chance nodes (for the dice).



Calculation of the Expected Value

Utility function for chance nodes C over MAX:

d_i: possible dice roll

$P(d_i)$: probability of obtaining that roll

$S(C, d_i)$: attainable positions from C with roll d_i

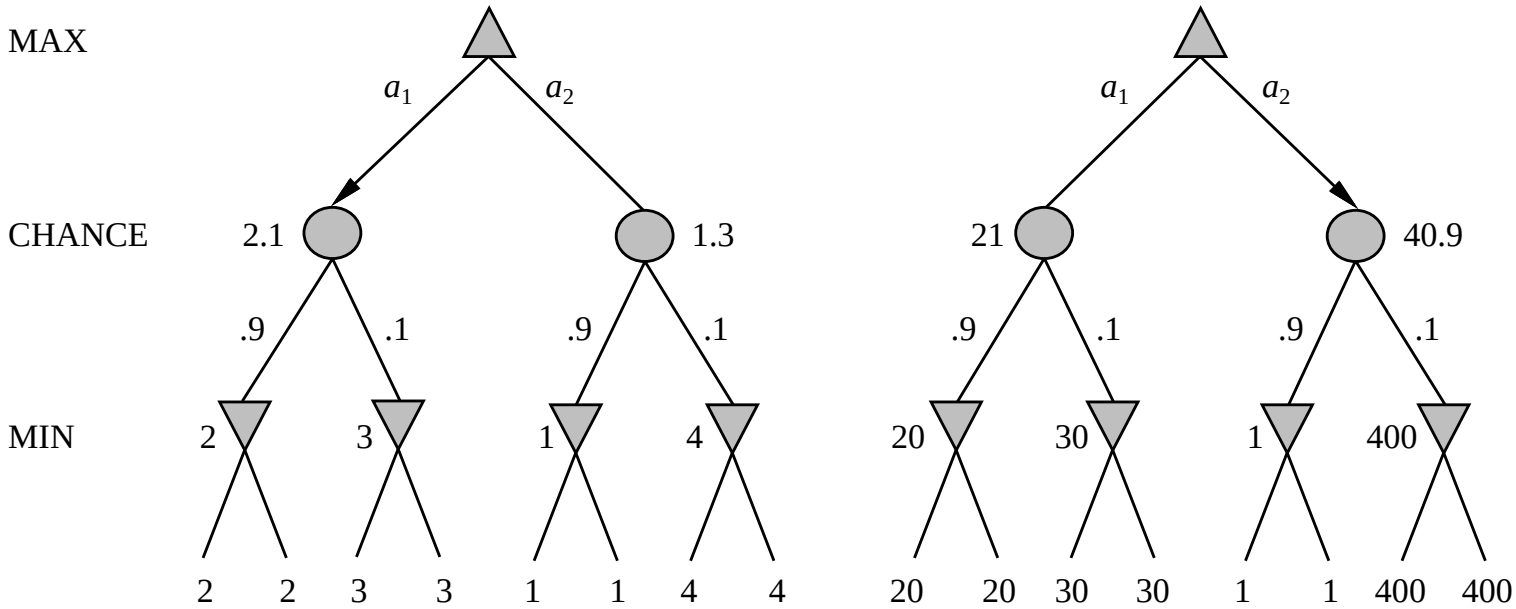
UTILITY(s): Evaluation of s

$$\text{EXPECTIMAX}(C) = \sum_i P(d_i) \max_{s \in S(C,d_i)} (\text{UTILITY}(s))$$

EXPECTIMIN likewise

Problems

- Order-preserving transformations on the evaluation values may change the best move:



- Search costs increase: Instead of $O(b^d)$, we get $O((b \times n)^d)$, where n is the number of possible dice outcomes.

$\rightarrow$ In Backgammon ($n = 21$, $b = 20$, can be 4000) the maximum for d is 2.

Card Games

- Recently card games such as bridge and poker have been addressed as well
- One approach: simulate play with open cards and then average over all possible plays (or make a Monte Carlo simulation) using minimax (perhaps modified)
- Pick the move with the best expected result (usually all moves will lead to a loss, but some give better results)
  - Averaging over clairvoyance
- Although "incorrect", appears to give reasonable results

State of the Art (1)

Backgammon: The *BKG* program defeated the official world champion in 1980. A newer program TD-Gammon is among the top 3 players.

Checkers, **draughts** (by international rules): A program called *Chinook* is the official world champion in man-computer competition (acknowledges by ACF and EDA) and is the highest-rated player:

Chinook: 2712	Ron King: 2632
Asa Long: 2631	Don Lafferty: 2625

In 1995, Chinook won a 32 game match against Don Lafferty.

Othello: Very good, even on normal computers. In 1997, the *Logistello* program defeated the human world champion.

Chess: In 1997, world chess master G. Kasparow was beaten by a computer in a match of 6 games by *Deep Blue* (IBM Thomas J. Watson Research Center).
Special hardware (32 processors with 8 chips, 2 Mi. calculations per second) and special chess knowledge.

State of the Art (2)

Go: The program *AlphaGo* was able to beat in March 2016 one of the best human players Lee Sedol (according to ELO ranking the 4th best player worldwide) 4:1.
AlphaGo used Monte Carlo search techniques (UCT) and deep learning techniques.

Poker: In January 2017, *Libratus* played against four top-class human poker players for 20 days **heads-up no-limit Texas hold 'em**. In the end, Libratus was more than 1.7 M$ ahead.
Libratus used a number of different techniques all based on game theory.

The Reasons for Success...

- Alpha-Beta-Search
- ... with dynamic decision-making for uncertain positions
- Good (but usually simple) evaluation functions
- Large databases of opening moves
- Very large game termination databases (for checkers, all ten-piece situations)
- For Go, Monte-Carlo and machine learning techniques proved to be successful.
- ...and very fast and parallel processors, huge memory, and plenty of plays.
- For Poker, game theoretic analysis together with extensive self-play (15 million core CPU hours) were important.

Summary

- A game can be defined by the initial state, the operators (legal moves), a terminal test and a utility function (outcome of the game).
- In two-player board games, the minimax algorithm can determine the best move by enumerating the entire game tree.
- The alpha-beta algorithm produces the same result but is more efficient because it prunes away irrelevant branches.
- Usually, it is not feasible to construct the complete game tree, so the utility of some states must be determined by an evaluation function.
- Games of chance can be handled by an extension of the alpha-beta algorithm.
- The success for different games is based on quite different methodolgies.