Game playing

Chapter 5, Sections 1–6

Outline

◇ Games

◇ Perfect play
 – minimax decisions
 – α–β pruning

◇ Resource limits and approximate evaluation

◇ Games of chance (briefly)

Games as search problems

The main difference to the previous slides:
now we have **more than one** agent that have **different** goals.

- All possible game sequences are represented in a game tree.
- The nodes are the states of the game, e.g. the board position in chess.
- Initial state and terminal nodes.
- States are connected if there is a legal move/ply.
- Utility function (payoff function).
- Terminal nodes have utility values 0, 1 or -1.

Types of games

	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information	battleships, blind tictactoe	bridge, poker, scrabble nuclear war

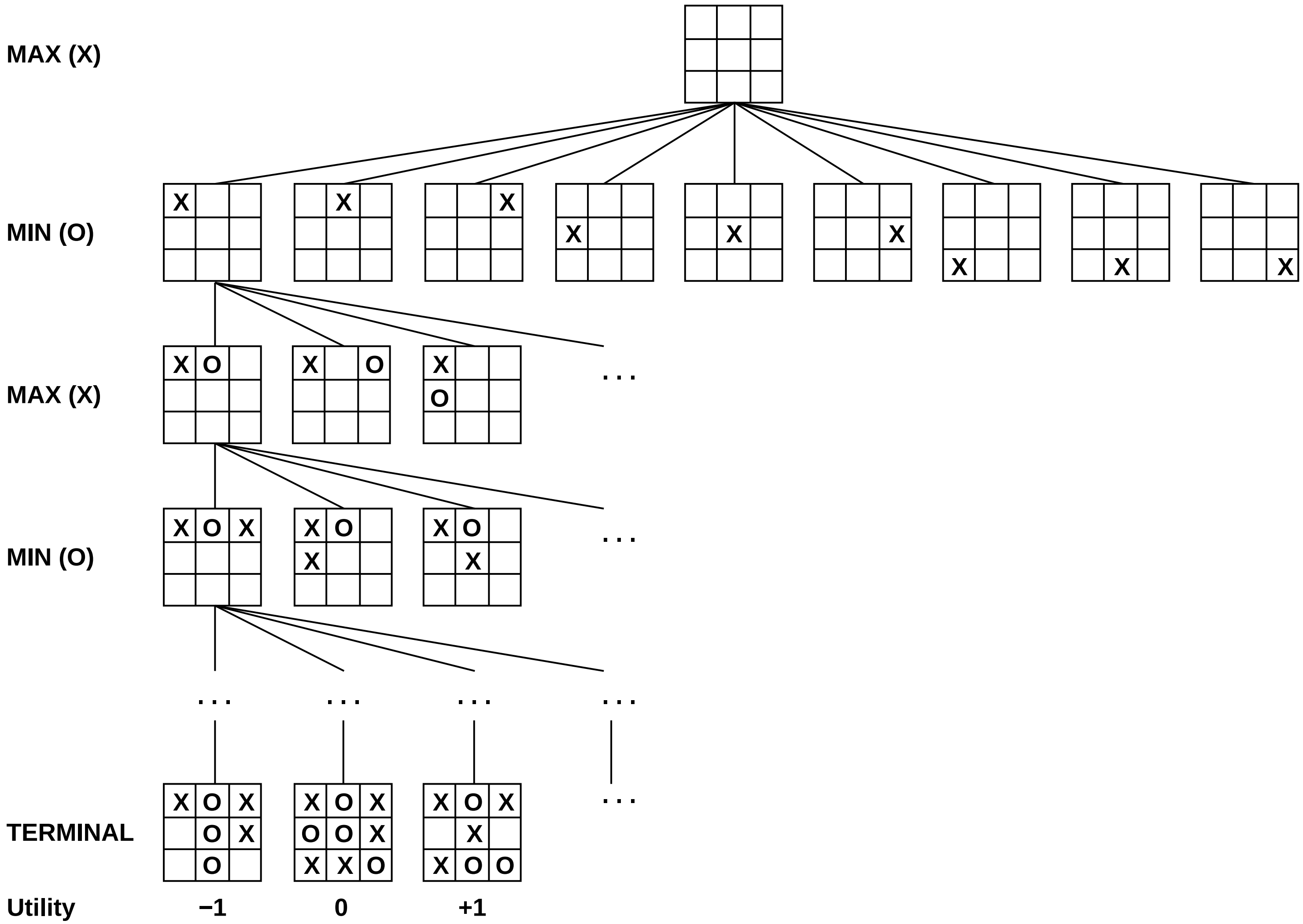
Strategies for Two-Player Games

Given two players called MAX and MIN, MAX wants to maximize the utility value. Since MIN wants to minimize the same value, MAX should choose the alternative that maximizes given that MIN minimized.

Minimax algorithm

MINIMAX(*state*) =
    if TERMINAL-TEST(*state*) **then**
        return UTILITY(*state*)
    if *state* is a MAX node **then**
        return $\max_s$ MINIMAX(RESULT(*state*, s))
    if *state* is a MIN node **then**
        return $\min_s$ MINIMAX(RESULT(*state*, s))

Game tree (2-player, deterministic, turns)

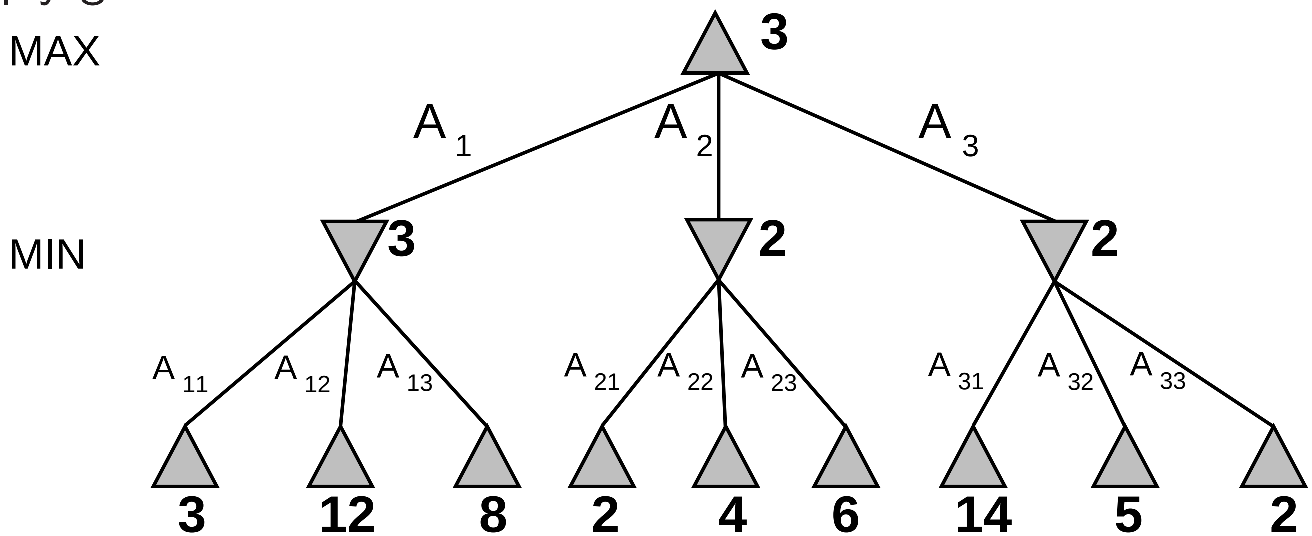


Minimax

Gives perfect play for deterministic, perfect-information games

Idea: choose the move with the highest minimax value
= best achievable payoff against best play

E.g., 2-ply game:



Minimax algorithm

function MINIMAX-DECISION(*state*) **returns** *an action*
  inputs: *state*, current state in game

  return the *a* in ACTIONS(*state*) maximizing MIN-VALUE(RESULT(*a*, *state*))

---

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

---

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

Properties of minimax

Complete?? Yes, if the game tree is finite

Optimal?? Yes, against an optimal opponent

Time complexity?? $O(b^m)$

Space complexity?? $O(bm)$ (depth-first exploration)

For chess, $b \approx 35$, $m \approx 100$ for "reasonable" games
$\Rightarrow$ an exact solution is completely infeasible

But do we need to explore every path?

α–β pruning

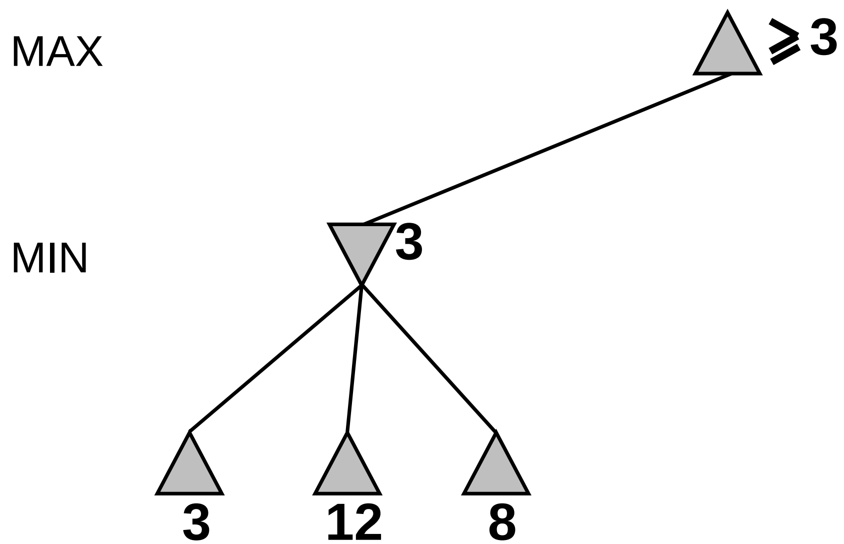
Suppose, we reach a node t in the game tree which has leaves $t_1, \ldots, t_k$ corresponding to moves of player MIN.

Let α be the best value of a position on a path from the root node to t.

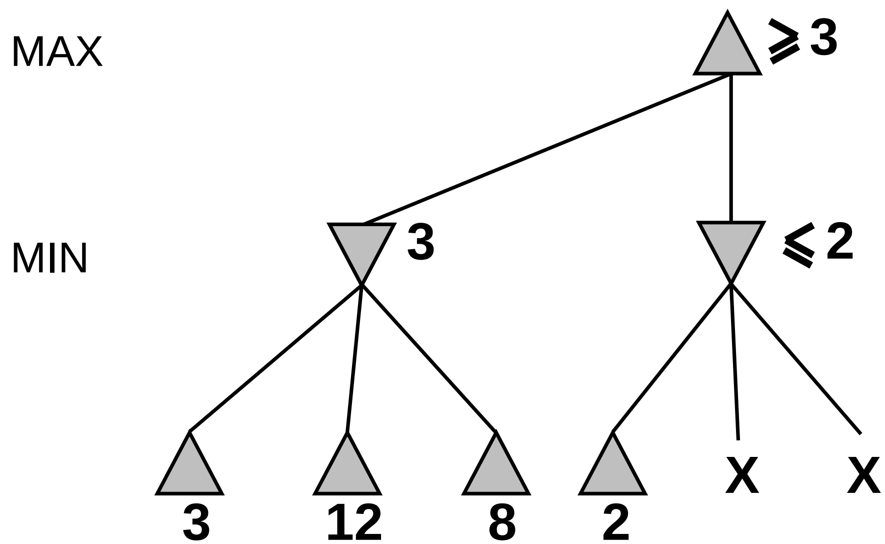
Then, if any of the leaves evaluates to $f(t_i) \leq \alpha$, we can discard t, because any further evaluation will not improve the value of t.

Analogously, define β values for evaluating response moves of MAX.

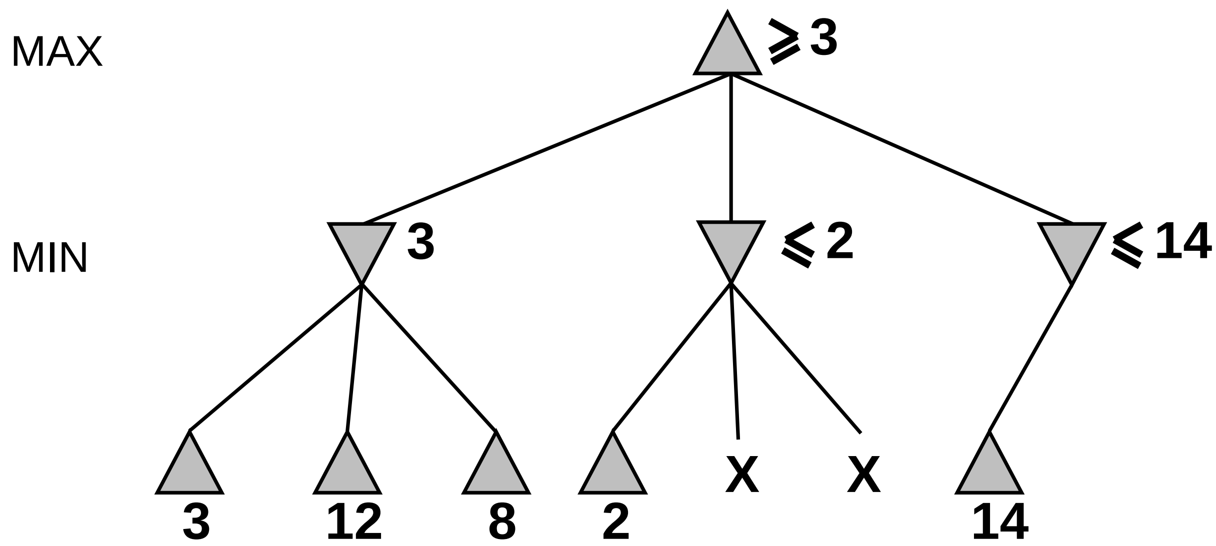
α–β pruning example



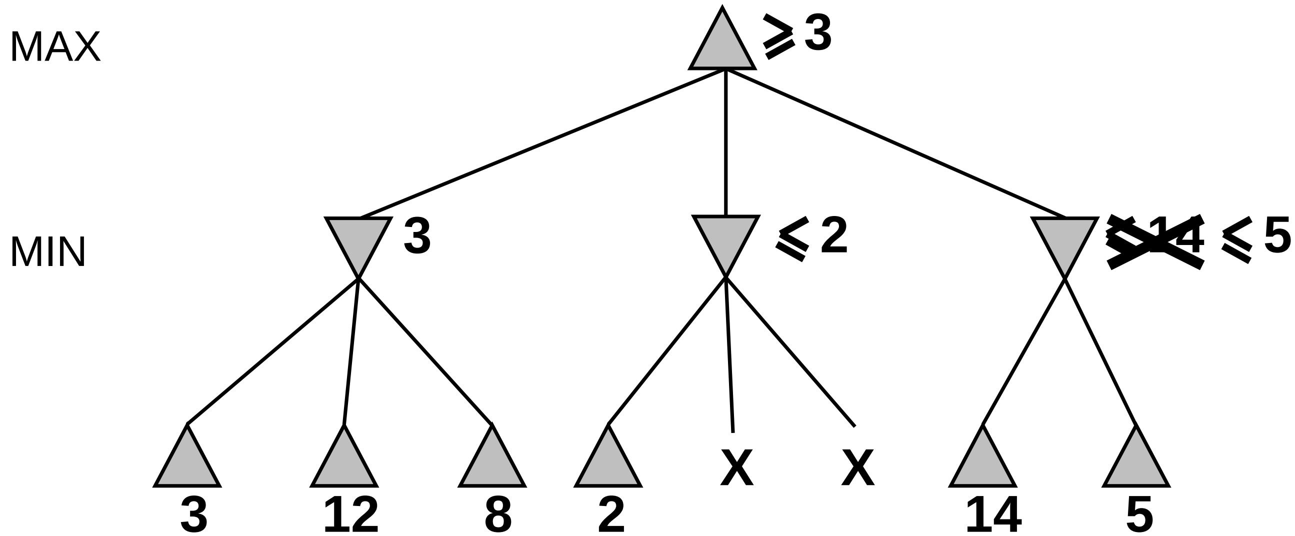
α–β pruning example



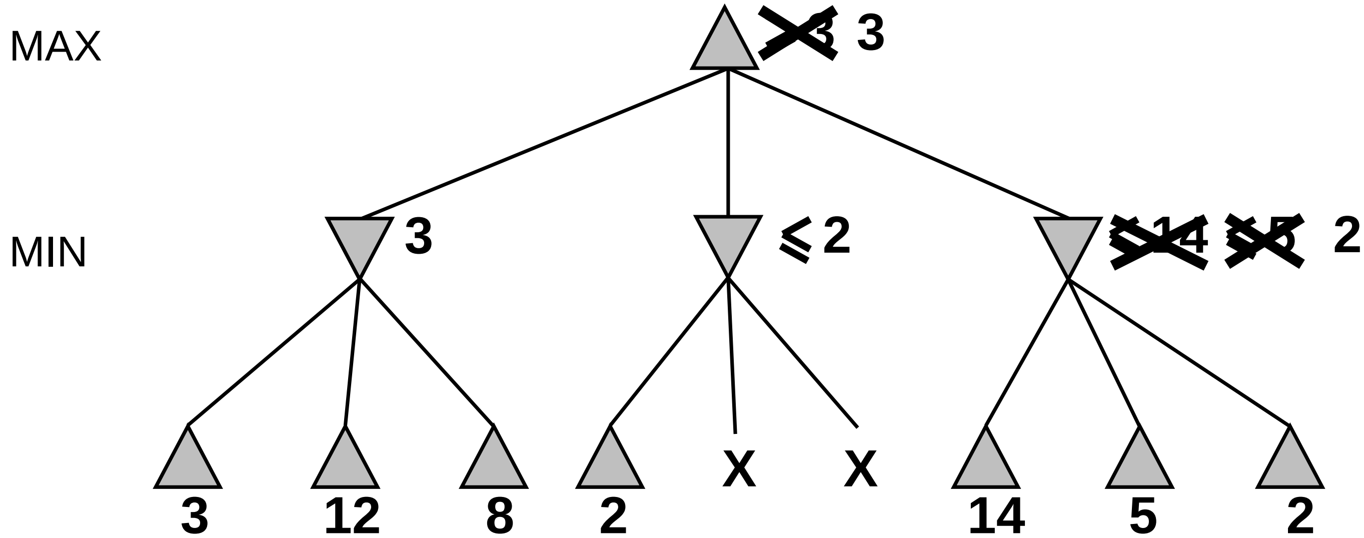
α–β pruning example



α–β pruning example



α–β pruning example



The α–β algorithm

function ALPHA-BETA-DECISION(*state*) **returns** an action
  return the *a* in ACTIONS(*state*) maximizing MIN-VALUE(RESULT(*a*, *state*))

---

function MAX-VALUE(*state*, α, β) **returns** *a utility value*
  inputs: *state*, current state in game
    α, the value of the best alternative for MAX along the path to *state*
    β, the value of the best alternative for MIN along the path to *state*

  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for *a*, *s* in SUCCESSORS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(s, α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

---

function MIN-VALUE(*state*, α, β) **returns** *a utility value*
  same as MAX-VALUE but with roles of α, β reversed

Properties of α–β pruning

Pruning **does not** affect the final result

A good move ordering improves the effectiveness of pruning

With "perfect ordering", the time complexity becomes $O(b^{m/2})$
⇒ this **doubles** the solvable depth

This is a simple example of the value of reasoning about which computations are relevant (a form of metareasoning)

Unfortunately, 35^{50} is still impossible!

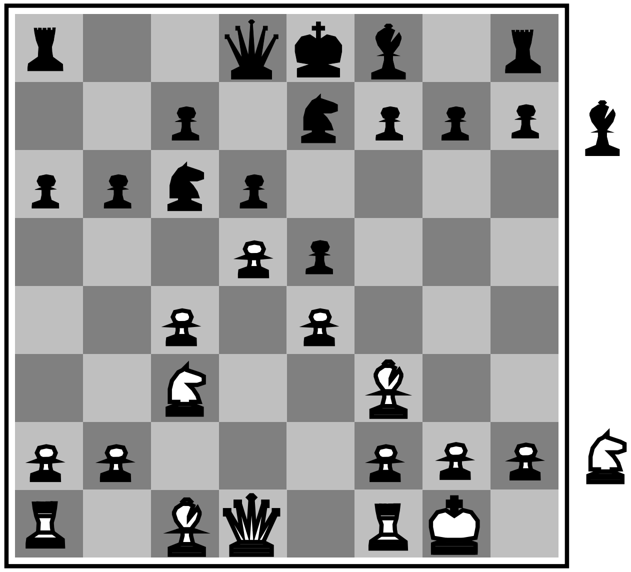
Resource limits

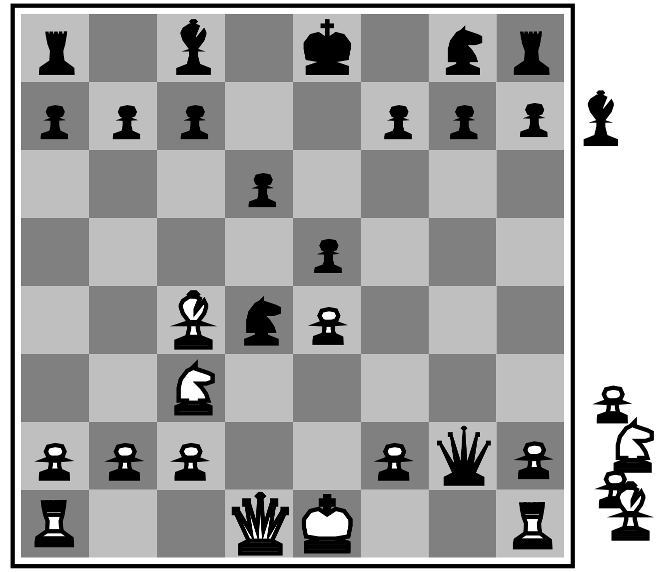
The standard approach is to cutoff the search at some point:

- Use CUTOFF-TEST instead of TERMINAL-TEST
  - use a depth limit
  - perhaps add quiescence search
- Use EVAL instead of UTILITY
  - i.e., an evaluation function that estimates desirability of position

Suppose we have 10 seconds per move, and can explore 10^5 nodes/second
– 10^6 nodes per move $\approx 35^{8/2}$ nodes
– α–β pruning reaches depth 8 $\Rightarrow$ pretty good chess program

Evaluation functions





Black to move

White slightly better

White to move

Black winning

For chess, the evaluation function is typically linear weighted sum of features

$$\text{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

e.g., $w_1 = 9$ with

$f_1(s) =$ (number of white queens) – (number of black queens)

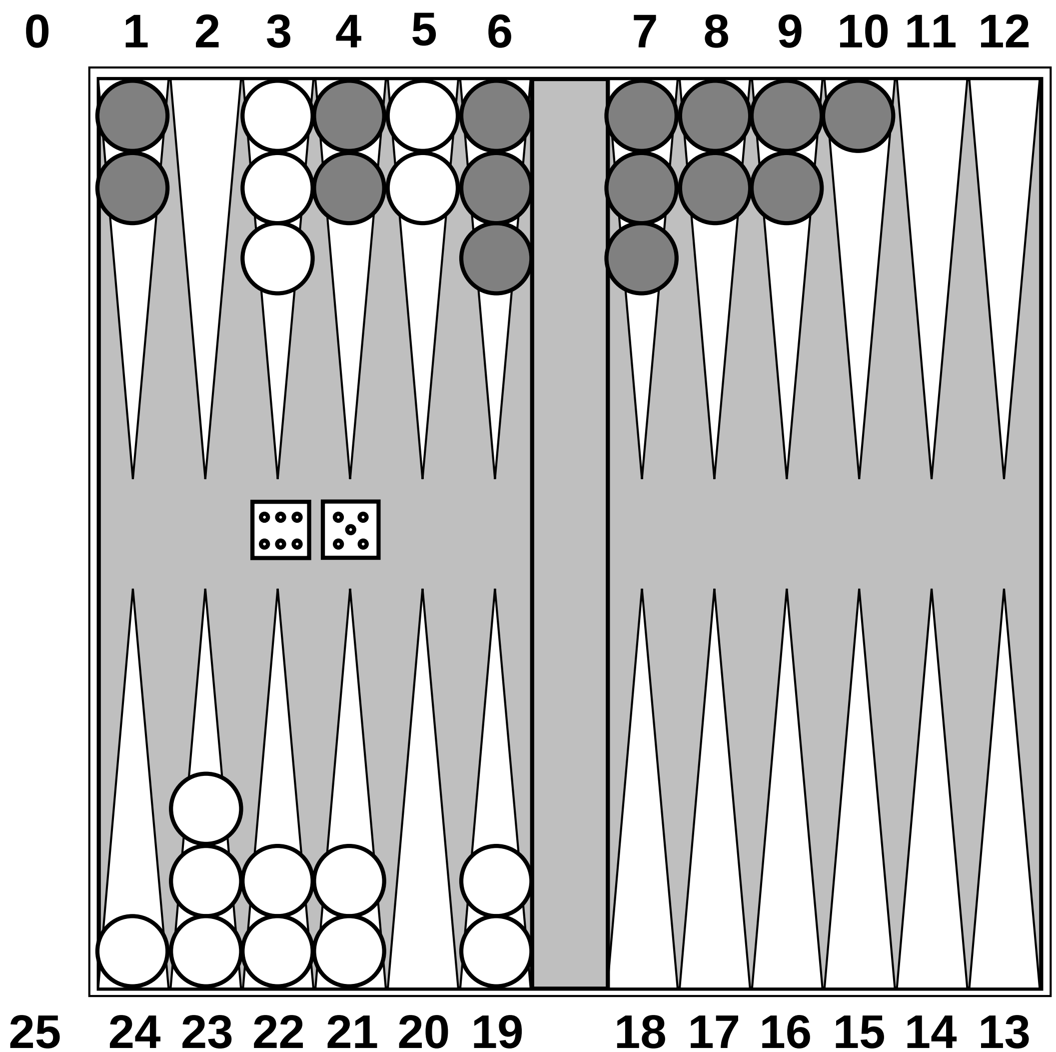
Deterministic games in practice

Chess: Deep Blue (IBM) beats chess world champion Garry Kasparov, 1997.
– Modern chess programs: Houdini, Critter, Stockfish.

Checkers/Othello/Reversi:
Human champions refuse to compete—computers are too good.
– Chinook plays checkers perfectly, 2007. It uses an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 443,748,401,247 positions.
– Logistello beats the world champion in Othello/Reversi, 1997.

Go: Human champions refuse to compete—computers are too bad.
– In Go, $b > 300$, so most programs use pattern knowledge bases to suggest plausible moves.
– Modern programs: MoGo, Zen, GNU Go

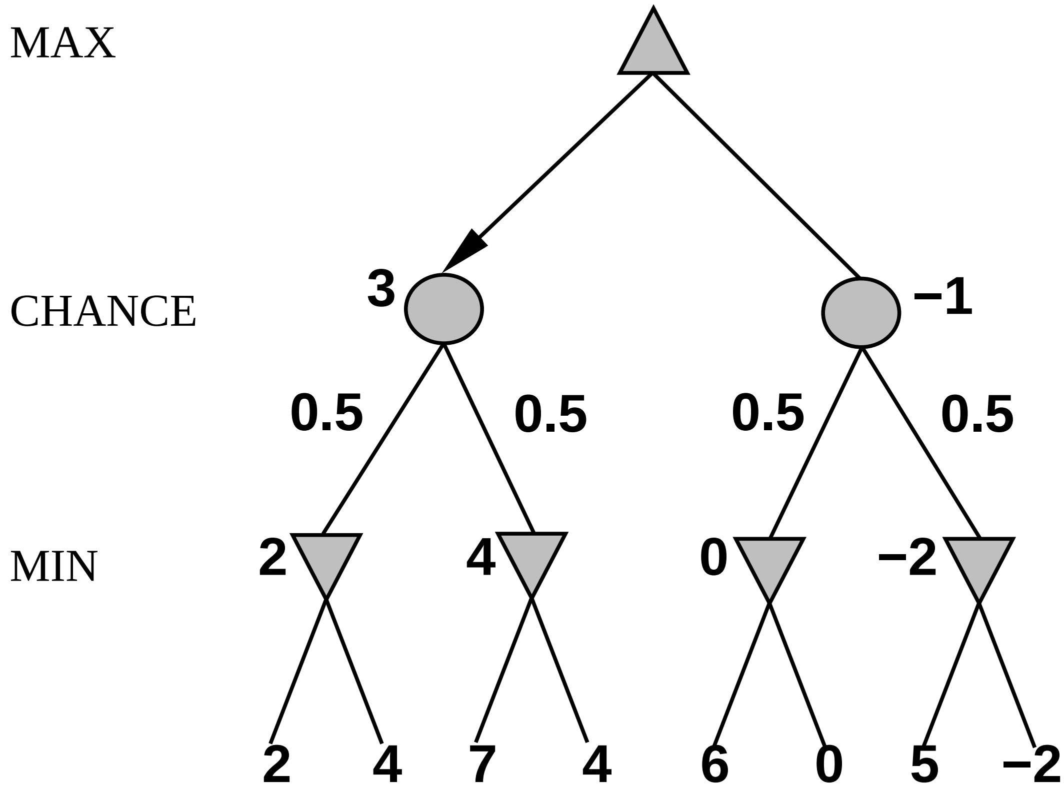
Nondeterministic games: backgammon



Nondeterministic games in general

In nondeterministic games, chance is introduced by dice, card-shuffling, etc.

Simplified example with coin-flipping:



Algorithm for nondeterministic games

ExpectiMinimax gives perfect play
– Just like Minimax, except we must also handle chance nodes

ExpectiMinimax(*state*) =
    if Terminal-Test(*state*) **then**
        return Utility(*state*)
    if *state* is a Max node **then**
        return $\max_s$ ExpectiMinimax(Result(*state*, s))
    if *state* is a Min node **then**
        return $\min_s$ ExpectiMinimax(Result(*state*, s))
    if *state* is a chance node **then**
        return $\Sigma_s\ P(s)$ ExpectiMinimax(Result(*state*, s))

where $P(s)$ is the probability that s occurs

Nondeterministic games in practice

Dice rolls increase the branching factor b:
– there are 21 possible rolls with 2 dice

Backgammon has $\approx$ 20 legal moves (can be up to 4,000 with double rolls)
– depth $4 \Rightarrow 20 \times (21 \times 20)^3 \approx 1.2 \times 10^9$ nodes

As depth increases, the probability of reaching a given node shrinks
– value of lookahead is diminished

α–β pruning is much less effective

TDGAMMON uses depth-2 search + very good EVAL
$\approx$ world-champion level