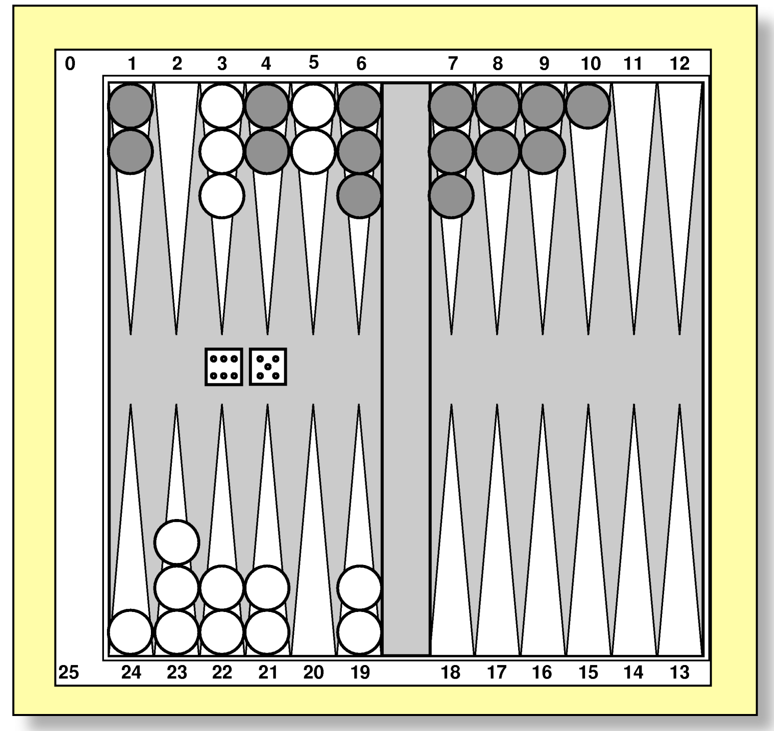
Adversarial Search

In which we examine the problems that arise
when we try to plan ahead in a world
where other agents are planning against us.

Outline

1. Games
2. Optimal Decisions in Games
3. Alpha-Beta Pruning
4. Imperfect, Real-Time Decisions
5. Games that include an Element of Chance
6. State-of-the-Art Game Programs
7. Summary



Search Strategies for Games

- Difference to general search problems
  - Imperfect Information: opponent not deterministic
  - Time: approximate algorithms

- Early fundamental results
  - Algorithm for perfect game von Neumann (1944)
  - Approximation through evaluation Zuse (1945), Shannon (1950)

	deterministic	*random*
perfect information	Checkers, Chess, Go	Backgammon, Monopoly
incomplete information	?	Bridge, Poker, Scrabble

- Our terminology:
  - deterministic, fully accessible information

Games as Search Problems

- Justification: Games are search problems with an opponent
- Imperfection through actions of opponent: possible results...
- Games hard to solve; exhaustive:
  - Average branching factor chess: 35
  - ≈ 50 steps per player → 10^{154} nodes in search tree
  - But "Only" 10^{40} allowed positions
- Games as playground for serious research
- How can we determine the best next step/action?
  - Cutting branches („pruning")
  - Evaluation functions for approximation of utility function

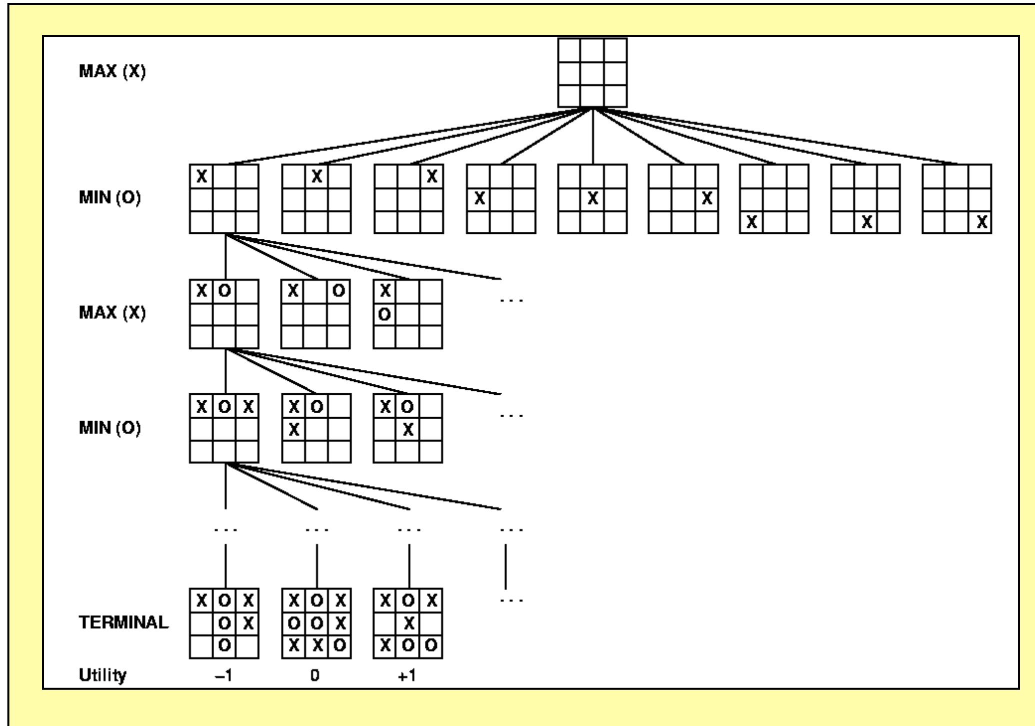
Search Problem

- 2-player games
  - Player MAX
  - Player MIN
  - MAX moves first; players then take turns

- Example: Chess
  - Search tree 10^{154} nodes
  - Only 10^{40} valid positions

- Search problem
  - Initial state
    - Board, positions, first player
  - Successor function
    - Lists of (move, state)-pairs
  - Goal test
    - Checks whether games is terminated
  - Evaluation function
    - Result of game
      e.g. +1,0,-1 (zero sum games)
    - also:payoff function

Example: Tic-Tac-Toe

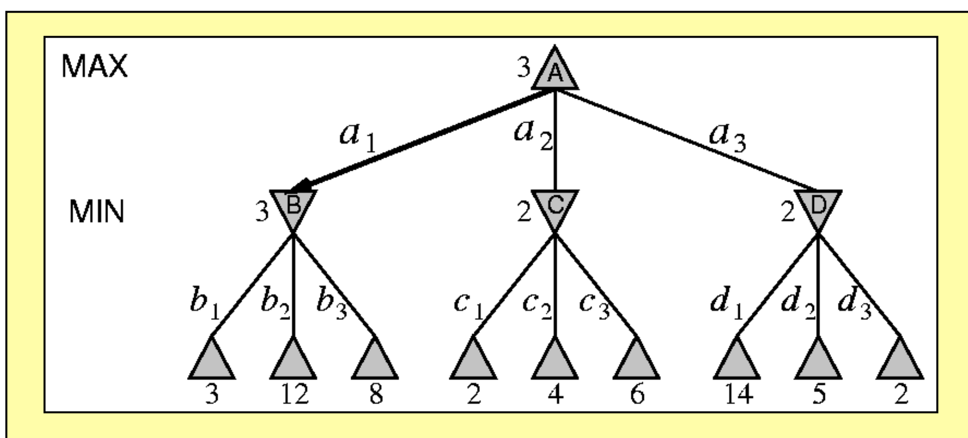


Partial search tree for Tic-Tac-Toe, MAX moves first (x)

- Initial state and legal moves define game tree
- MAX has nine options
- Games continues until one of the players has 3 x or 3 o in a row, column or diagonal or none of the fields is empty
- Number at leaves is utility value of final states from MAX' view (high values are good)

Optimal Strategies

- Normal search problems
  - Final states deliver result (Win)
  - MIN against this
  - Thus, MAX needs a contingent strategy
  - First move
  - Moves after MIN has moved
- Tic-Tac-Toe
  - Too complex to show complete tree. Here trivial game: ends after one move each
  - One move deep, two half-moves, each is called a **ply**

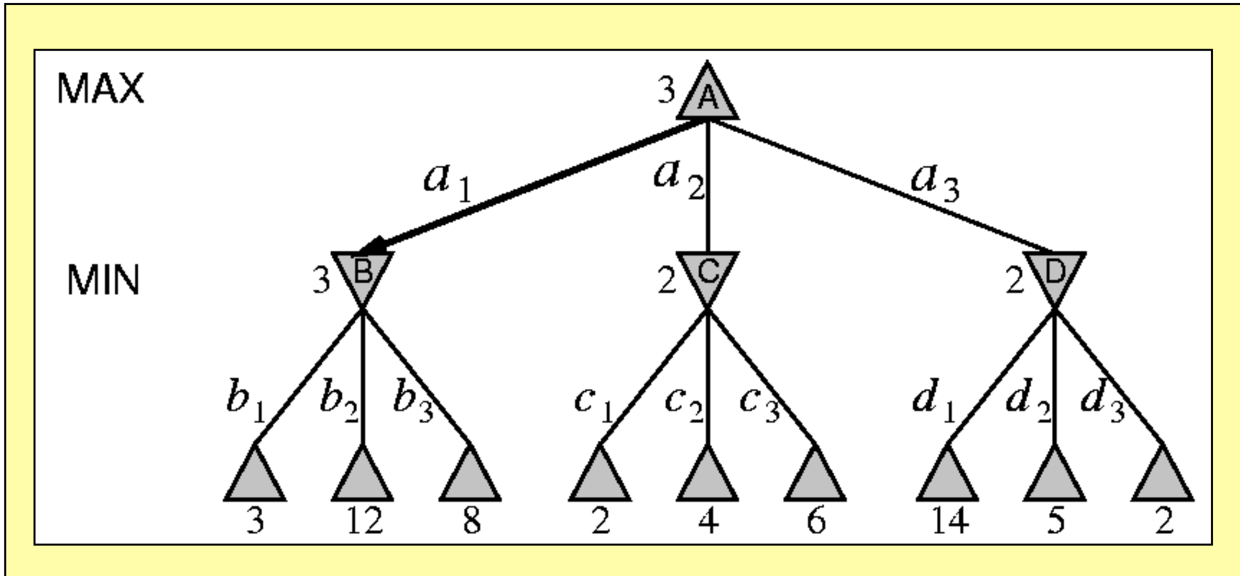


Minimax-Value

- Optimal strategy with game tree: determine the min/max value of each node
  - Minimax-Value(n)
- Minimax-Algorithm for determination of optimal strategy and for best first move.
- Minimax-Decision: maximizes utility under the assumption that MIN plays perfectly (to minimize utility).

$$\text{Minimax-Value(n)} = \begin{cases} \text{UTILITY}(n) & \text{if } n \text{ is a terminal state} \\ \max_{s \in Successors(n)} \text{MINIMAX-VALUE}(s) & \text{if } n \text{ is a MAX node} \\ \min_{s \in Successors(n)} \text{MINIMAX-VALUE}(s) & \text{if } n \text{ is a MIN node.} \end{cases}$$

Trivial Tic-Tac-Toe



- a_1-a_3 are MAX legal moves
- MIN can answer with $b_{1\text{-}3}$, $c_{1\text{-}3}$, $d_{1\text{-}3}$
- One move = 2 half moves = 2 plies
- ∇ - nodes: MIN moves
- Terminal values are utility values for MAX, other values are determined through utility-values of successors

Minimax-Algorithm

- Create search tree of game
  - All the way to the end!
- Evaluate leaves
  - Utility for each end of game
- Propagate evaluations to root
  - MAX chooses maximal utility
  - MIN chooses minimal utility

- Depth-first for whole tree
- Time complexity: max depth *m* and *b* legal moves at each point: $O(b^m)$
- Space complexity $O(bm)$, if all successors are calculated
- Real Games: Time complexity completely different!

Minimax-Algorithm

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

  $v \leftarrow$ MAX-VALUE(state)
  return the *action* in SUCCESSORS(*state*) with value v

---

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

---

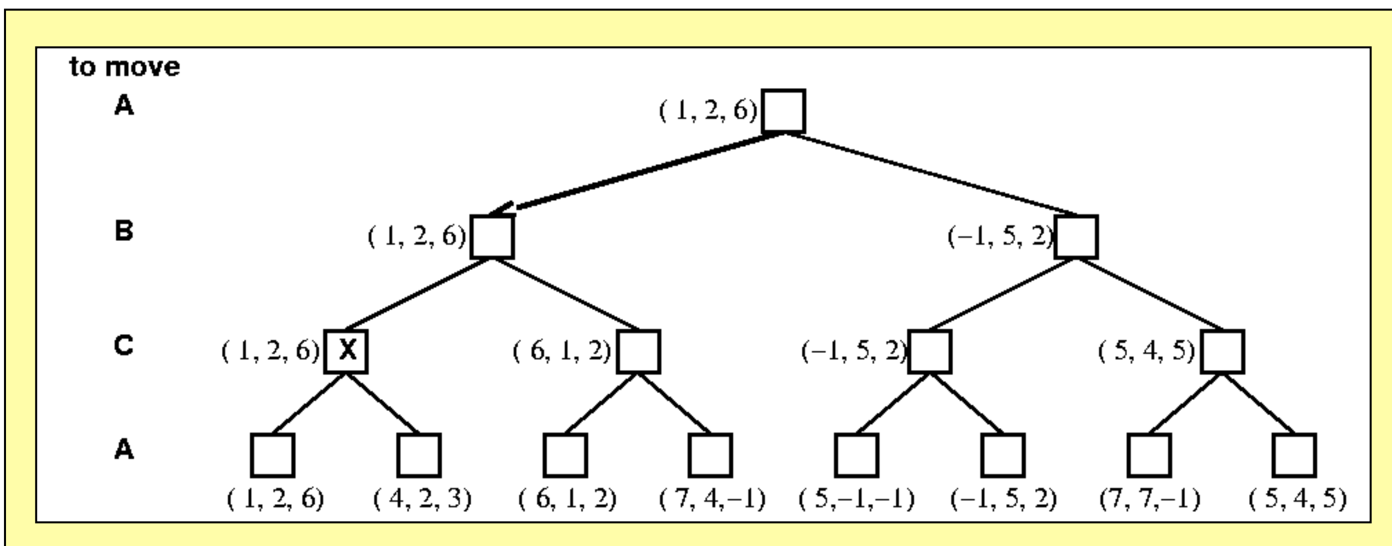
function MIN-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow \infty$
  for a, s in SUCCESSORS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(s))
  return v

- Minimax decisions
- maximize/minimize utility
- Action selection accordingly
- Assumption: Max/ Min always play optimal!

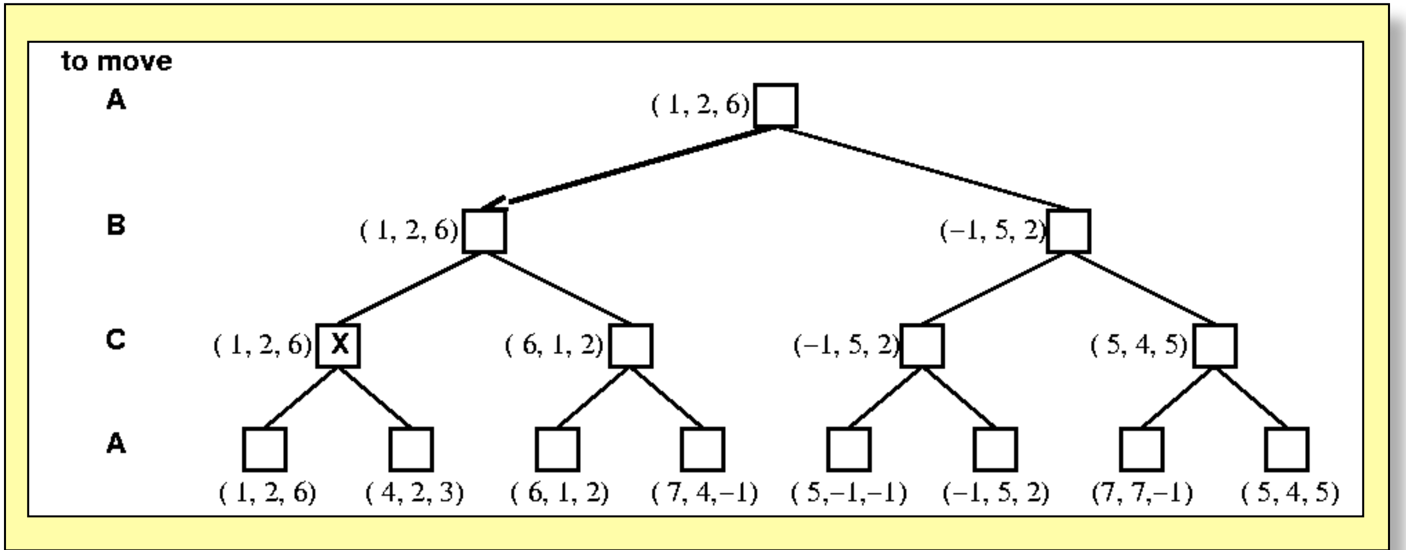
Optimal Decisions in Games with more than 2 Players

- Extension of minimax algorithm possible
- Single value of node substituted by vector of values

- Example.:
  - 3 players A,B,C: 3 vectors per node $\{v_A, v_B, v_C\}$
  - Final states: values from viewpoint of each player
  - Nodes in tree



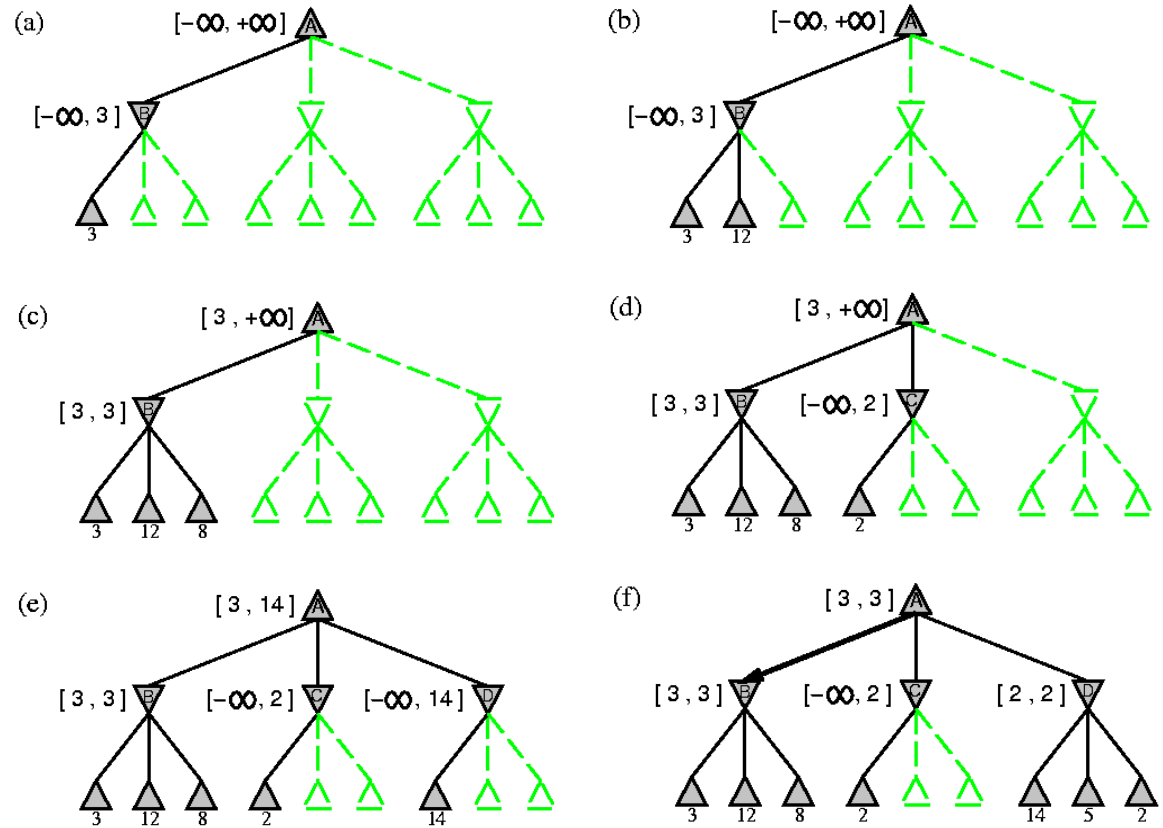
Example



- C decides about next move
  - Two options: $\{v_A=1, v_B=2, v_C=6\}$, $\{v_A=4, v_B=2, v_C=3\}$
  - Because 6 > 3, the first option should be taken, i.e. if X is reached, (1,2,6) is final state
- Alliances as problem

Alpha-Beta Pruning

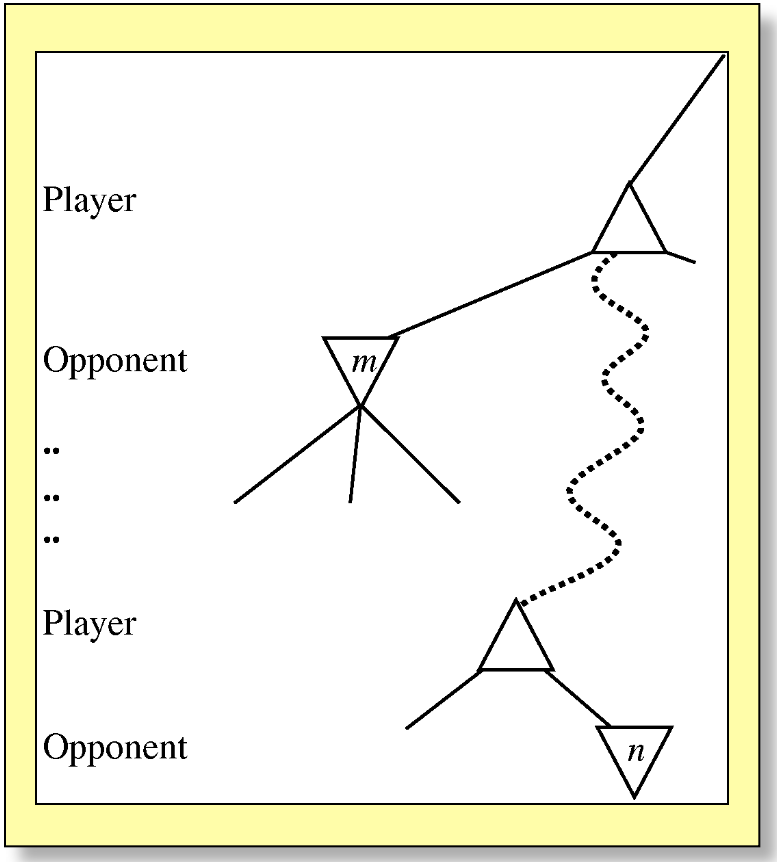
- Problem Minimax:
  - Search space exponential in number of moves
- Shortening search
  - Idea of Alpha-Beta-Pruning: Cut off branches that cannot influence decision



Range of values given per node

General Case

- Alpha-Beta-Pruning: Cut off branches that cannot influence decision
- Principle of Alpha-Beta-Pruning: if *m* better as *n*, we never get to *n*



Algorithm

- Only two different lines of code w.r.t. Minimax

- Effectivity: *O(b^{d/2})* Consider nodes only if the best successor nodes are analyzed first.

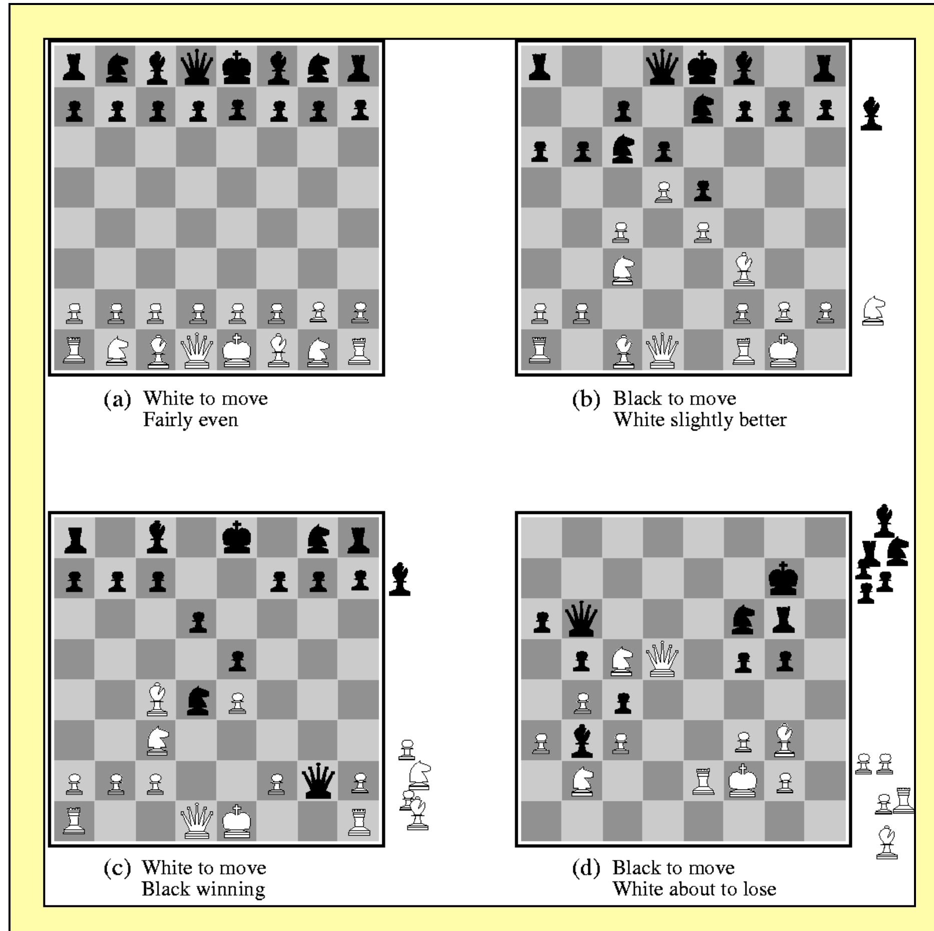
```
function ALPHA-BETA-SEARCH(state) returns an action
   inputs: state, current state in game

   v ← MAX-VALUE(state, −∞, +∞)
   return the action in SUCCESSORS(state) with value v

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 ← −∞
   for a, s in SUCCESSORS(state) do
      v ← MAX(v, MIN-VALUE(s, α, β))
      if v ≥ β then return v
      α ← MAX(α, v)
   return v

function MIN-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 ← +∞
   for a, s in SUCCESSORS(state) do
      v ← MIN(v, MAX-VALUE(s, α, β))
      if v ≤ α then return v
      β ← MIN(β, v)
   return v
```

Imperfect, real-time decisions

- Minimax searches whole tree
- Alpha-Beta Pruning helps to shorten significant part
- However, search whole tree down to leaves not practical in most of the times
- Better: heuristic evaluation function that makes non-terminal node temporarily to terminals
- Minimax or Alpha-Beta algorithms are substituted in two ways
  - Heuristic evaluation function instead utility function
  - Shortening; Cutoff-Test instead of goal test. Cutoff-Test decides when to use evaluation function

Heuristic Evaluation Function

- Substitution of utility function through heuristic evaluation function
- Early cut-off of branches
- Requirements of evaluation function:
  - e.g. measure value of 'material' in chess:
    Pawn = 1, Knight/Bishop = 3,
    Rook = 5, Queen = 9,
    others (e.g. King safety =
    1/2 Pawn



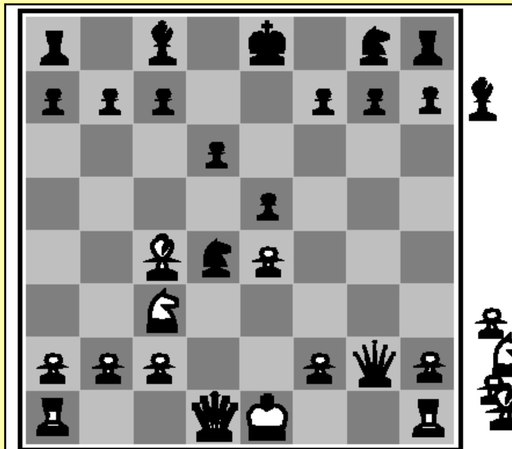
Requirements of Evaluation Function

- Conformance with utility function for leaves
- Performance!
- Depiction of winning chances
- "Evaluation function should represent winning options for an arbitrary position of a material category"

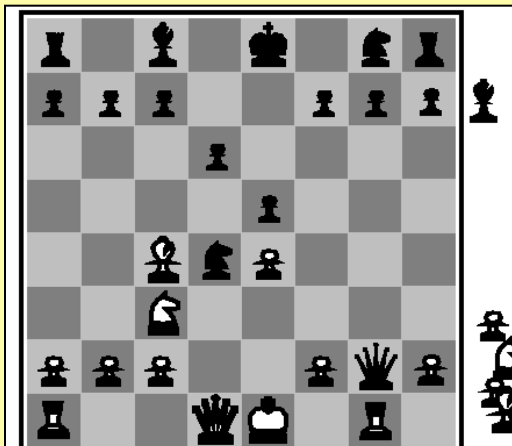
- Example: weighted linear evaluation function:
  - $w_1 f_1 + w_2 f_2 + \ldots + w_n f_n$
    with: *w* = weights (values for pieces, e.g. 1 for pawns, 3 for knight) and *f* = number of play elements

Cut-Off Search

- … with fixed limit, i.e. Cut-off-Test for all nodes until limit successful
- Goal: apply evaluation function only on '**quiescent**' positions
- Example:
  - Assumption: evaluation function based on material advantage, program searches until limit of depth, reaches position b)
  - Evaluation function would probably say that win is likely
  - However, white can beat queen in one move
  - Search from unstable states for stable states
  - Material function, i.e. apply evaluation function only of positions are quiescent



(a) White to move

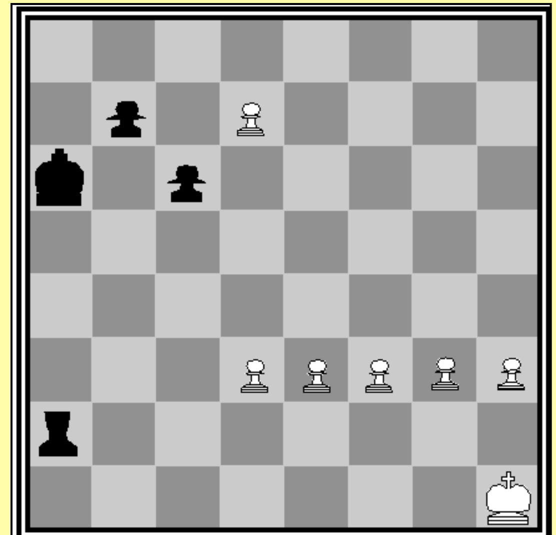


(b) White to move

Horizon Problem

- Horizon problem
  - Opponent moves, move is significant damage and cannot be avoided

- Example:
  - It looks like black has light advantages. If white brings pawn in 8th row a queen will be given and white will win
  - Black can forestall this by checking with the rook. Stalling moves pushes the inevitable queening move “over the search horizon” to a place where it cannot be detected.
  - A limited depth-first search cannot foresee move (pawn-queen)



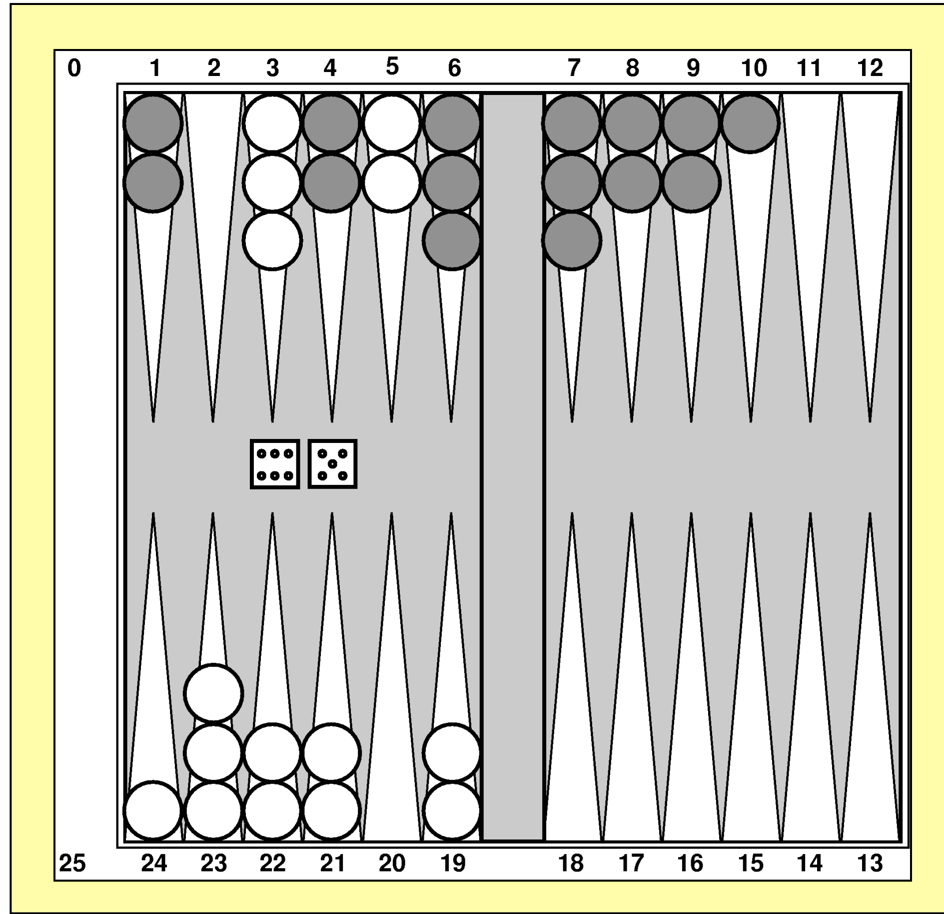
Black to move

Example: Chess

- Assumption:
  - Evaluation function implemented
  - “Reasonable” Cut-Off-Test for stable states
  - Large “transposition table” (repeated states in a hash-table)
  - ≈ 1 Mio nodes can generated and evaluated (on 2 GHz PC)
  - ~200 Mio moves per standard time control (3 min)
- Test
  - b = 35 for chess, with 35^5 ~ 50 Mio, i.e. 5 plies for evaluation
    - Average chess player would win
  - With Alpha-Beta-Pruning: 10 plies for evaluation
    - Expert level
  - With more pruning techniques 14 plies
    - Grandmaster level

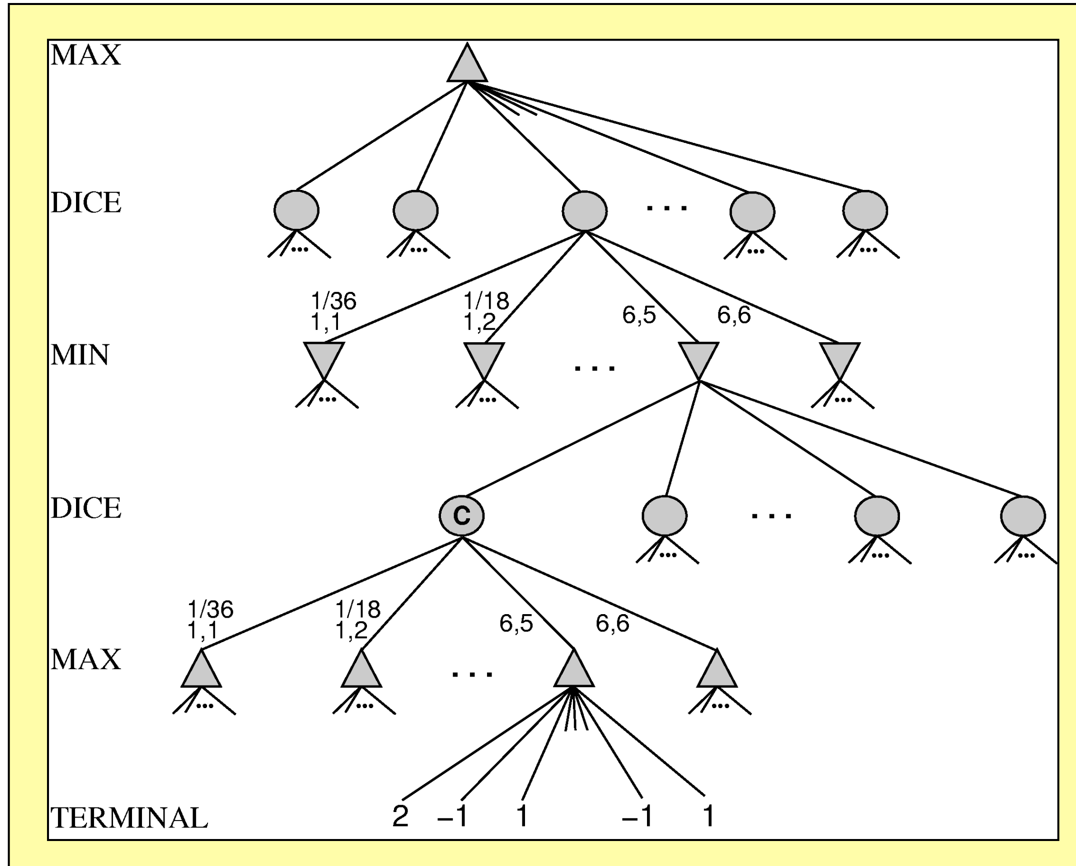
Games with Element of Chance

- Unpredictable events bring new situations
- Knowledge and luck (dice)
- Example:
  White has diced a 6 and a 5,
  four options now: (5-10,5-11),
  (5-11,19-24), (5-10,10-16),
  (5-11,11-16)
- Problem:
  White does not know what Black will dice nor what Black will do
- Construction of complete search tree not possible
- → Chance nodes in addition



Chance Nodes (CN)

- CN as circles
  - We cannot calculate best move
  - But: we can calculate an average for all possible dice rolls
- Leaves (final states)
  - As in deterministic games
- Chance C:
  - Suppose d_i is a dice roll and $P(d_i)$ the a priori probability. For each dice roll calculate, sum-up and weight utility for best moves



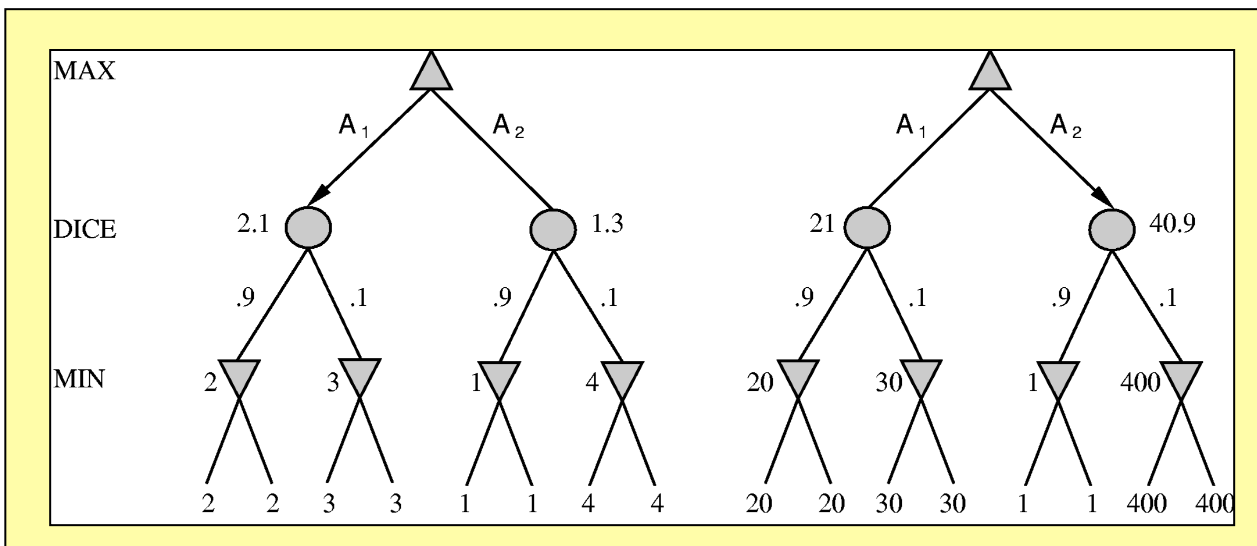
Expectiminimax Value

- Expectiminimax Value
  - Minimax value for games with Chance Nodes
- But:
  - Values are not "real" Minimax-values
  - Only: expected (probabilistic) value
- Probability
  - over dice roll occurrence
- Generalization
  - of Minimax value to Expectiminimax value

$$\text{EXPECTIMINIMAX}(n) = \begin{cases} \text{UTILITY}(n) & \text{if } n \text{ is a terminal state} \\ \max_{s \in Successors(n)} \text{EXPECTIMINIMAX}(s) & \text{if } n \text{ is a MAX node} \\ \min_{s \in Successors(n)} \text{EXPECTIMINIMAX}(s) & \text{if } n \text{ is a MIN node} \\ \sum_{s \in Successors(n)} P(s) \cdot \text{EXPECTIMINIMAX}(s) & \text{if } n \text{ is a chance node} \end{cases}$$

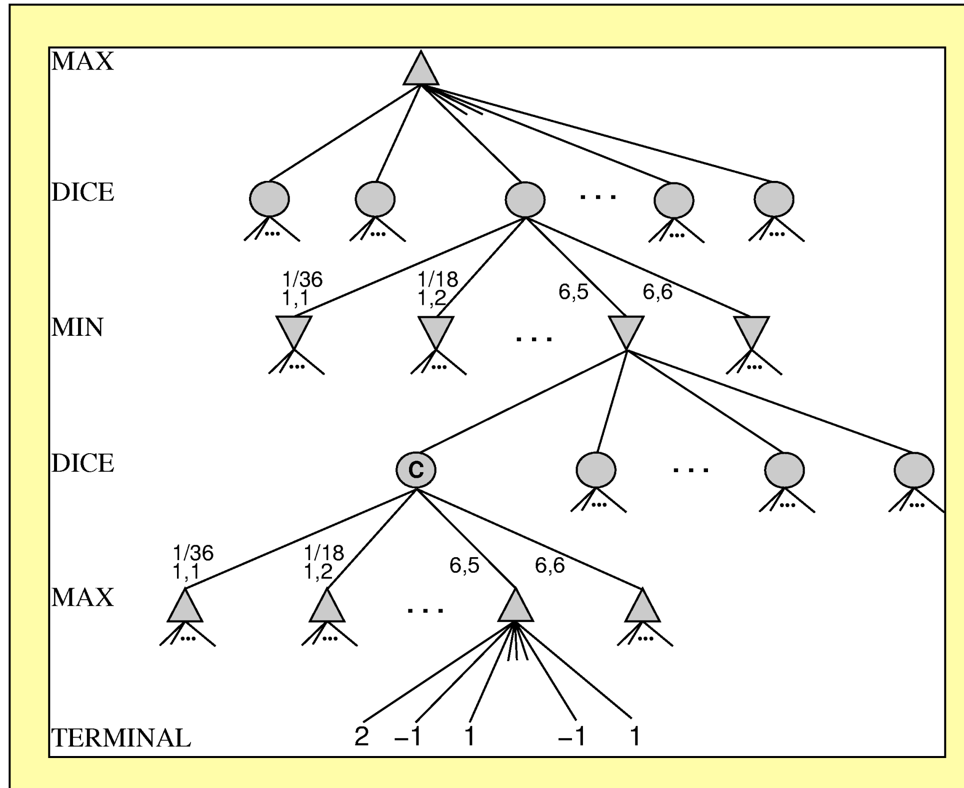
Position Evaluation

- Obvious: Cut-Off for search apply evaluation function at each leaf
- With Minimax: order preserving transformation of leaves does not make difference (1,2,3,4) vs (1,20,30,400),
  → Free to choose a function
- With randomness we loose this freedom: (1,2,3,4) is A_1 is best choice. (1,20,30,400): A_2 is better
  → program operates differently!
- Avoidance: Evaluation function can only be a **positive linear transformation of the probability of winning from a position**
- Important and general property in situations where uncertainty is



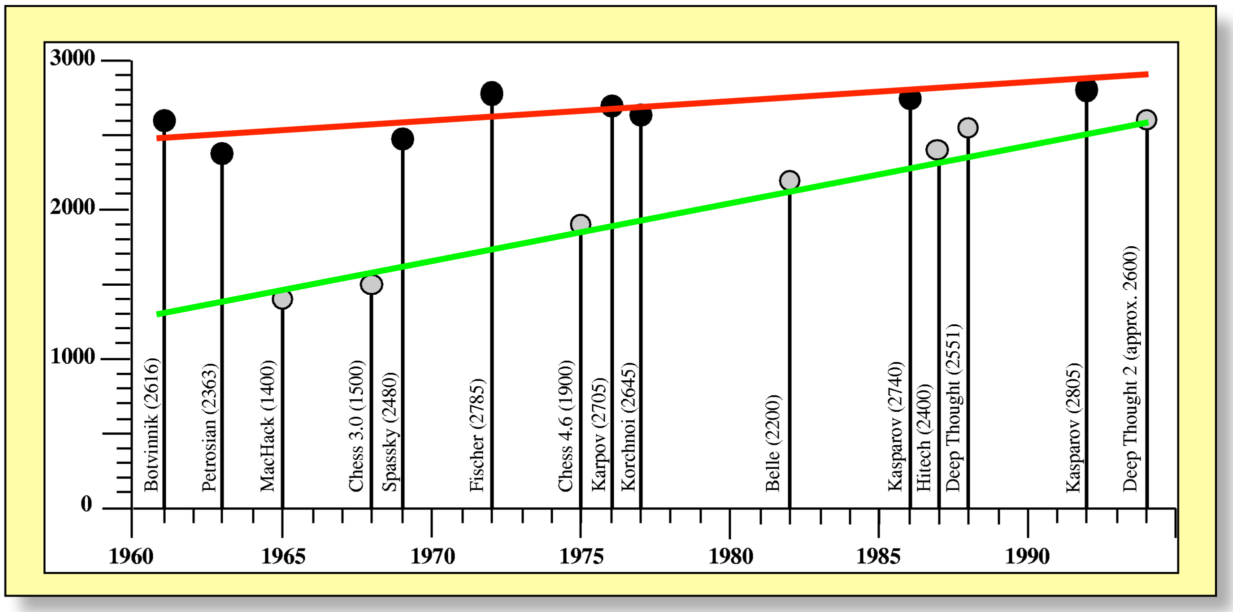
Complexity of Expectiminimax

- Minimax $O(b^m)$
- Expectiminimax $O(b^m n^m)$
  - n is number of dice rolls
  - lot of extra costs (e.g. for Backgammon $n = 21, b \approx 20$), sometimes even $b=4,000$ (doubles))
- Alpha-Beta Pruning
  - Upper bound for C?
  - Possible if upper bound for utility function given



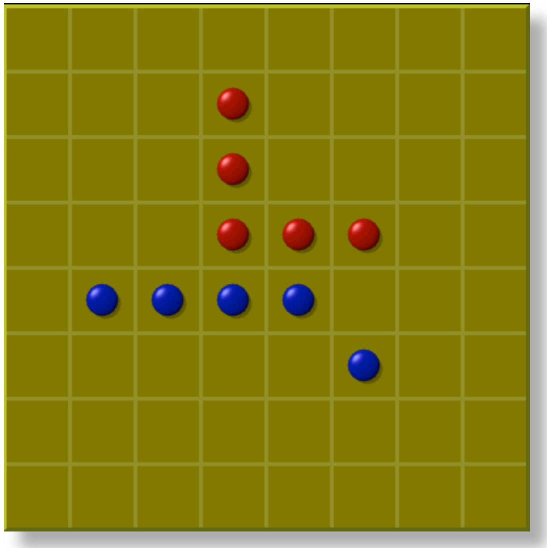
State-of-the-art Programs

- Two goals with game development:
  - Action selection in complex domains with uncertain result
  - Development of high-performance systems for special games
- Here: the latter (Chess)
  - Concentration on chess extremely distinctive
  - Speed-Chess (5 and 25 min) Computer wins against Kasparov
  - In normal tournament a little less good



State-of-the-art Programs

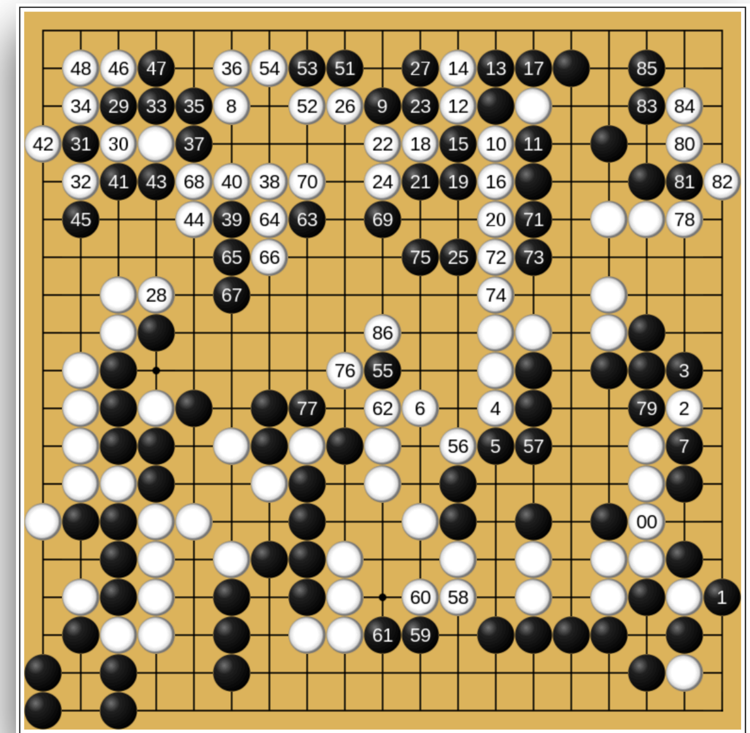
- Chess
  - Deep Blue: 30 billion positions per move, depth >14
  - HYDRA successor using FPGA, 18 plies
  - RYBKA, won 2008/9, unknown eval-function which is the key
  - Komodo, Stockfish, Houdini as commercial products
  - Stockfish 9, Houdini 6, or Komodo 11.2 highest rated on CCRL ('18) won every game of 6 game match against WC Magnus Carlson ('15)
  - https://ccrl.chessdom.com/ccrl/4040/



- Othello/Reversi
  - Search space less than chess
  - 5 -15 legal moves
  - Computer way better than humans
  - 1997 Logistello (Buro, 2002) 6:1 against WC
  - Saio, Edax and Cyrano (2011) much faster than Logistello
- Backgammon
  - Uncertainty through dice rolls, search thus expensive
  - TD-Gammon (Gerry Tesauro) on ANN & RL basis
  - 1 Mio training games against itself
  - Top 3 of the world
  - GNU Backgammon, BGBlitz, Palamedes winners of 2015 computer olympiad

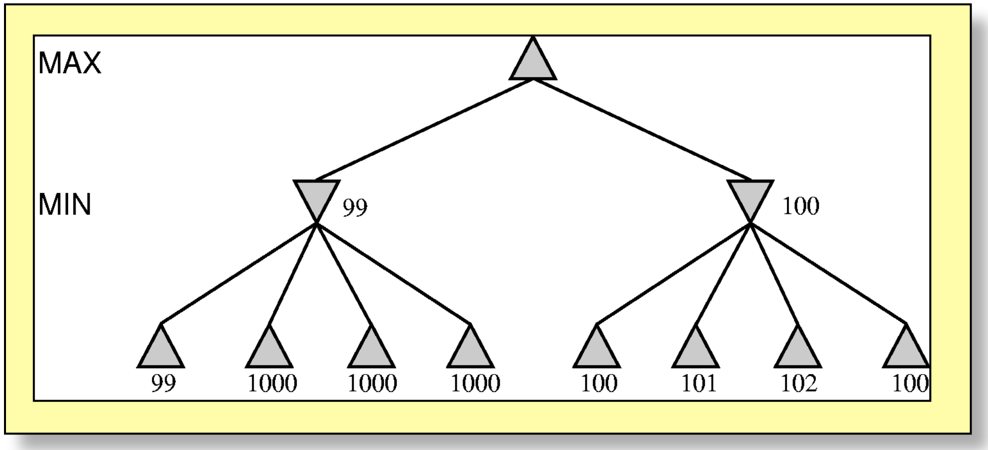
State-of-the-art Programs

- Go
  - *b* reaches 360 on 19x19 board, regular search impossible
  - Systems based on knowledge-based approach, until 1997 no good programs
  - MoGo program (runs on 800 processor 15 Tflop supercomputer (1000x DeepBlue)
  - AlphaGo, first computer program beating a human professional, 2018: ELO > 5,000, 2017 Nature article: https://www.nature.com/articles/nature24270
  - Uses a combination of ML and tree search techniques, extensive training (both human and computer play).



Discussion

- Optimal decisions in games mostly inefficient (intractable in most cases)
- Thus: algorithms operate with assumptions and approximations
  - Standard approach, based on Minimax, Evaluation function and Alpha-Beta Pruning
  - Minimax is optimal method for next move if search tree is given and evaluation of leaves are correct.
  - Reality: only estimations, in figure Minimax seems not to be a good choice
  - Algorithm decides for right branch, but it is more likely that left branch is better in reality
  - Minimax assumption: all right nodes are better that 99 on the left



Summary

- Games for AI like Formula 1 in Motorsports. Here are the most important ideas:
  - A game can be defined by the **initial state** (how the board is set up), the legal **actions** in each state, a **terminal test** (which says when the game is over), and a **utility function** that applies to terminal states.
  - In two-player zero-sum games with **perfect information**, the **minimax** algorithm can select optimal moves using a depth-first enumeration of the game tree.
  - The **alpha-beta** search algorithm computes the same optimal move as minimax, but achieves much greater efficiency by eliminating subtrees that are provably irrelevant.
  - Usually, it is not feasible to consider the whole game tree (even with alpha-beta), so we need to cut the search off at some point and apply an **evaluation function** that gives an estimate of the utility of a state.

Summary 2

- Games of chance can be handled by an extension to the minimax algorithm that evaluates a **chance node** by taking the average utility of all its children nodes, weighted by the probability of each child.
- Optimal play in games of **imperfect information**, such as bridge, requires reasoning about the current and future **belief states** of each player. A simple approximation can be obtained by averaging the value of an action over each possible configuration of missing information.
- Programs can match or beat the best human players in Checkers, Othello, and Backgammon, and are close behind in bridge. A program has beaten the world chess champion in one exhibition match. Programs remain at the amateur level in Go.