Spezielle Themen der Künstlichen Intelligenz

3. Termin: Game Playing

Dr. Stefan Kopp
Center of Excellence „Cognitive Interaction Technology“
AG Sociable Agents

Outline

- What are games?
- Optimal decisions in games
  - Which strategy leads to success?
- α-β pruning and various improvements
- Games & uncertainty
  - games of imperfect information
  - games that include an element of chance

Games = Search ?

Search – no adversary (opponent)

- solution is (heuristic) method for finding goal
- heuristics and CSP techniques can find *optimal* solution
- evaluation function: estimate of cost from start to goal through given node
- *examples*: path planning, scheduling

Games – adversary

- solution is strategy = specifies move for every possible opponent reply
- time limits force an *approximate* solution
- evaluation function: evaluate "goodness" of game position
- *examples*: chess, checkers, Othello, backgammon

Types of Games

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

Game setup

Two players: MAX and MIN

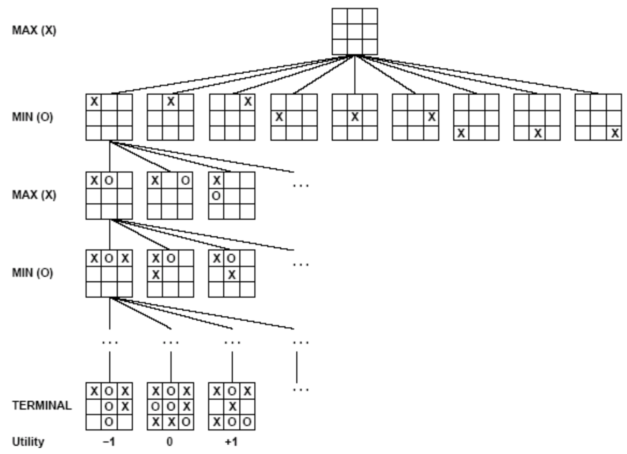
- MAX moves first and they take turns until the game is over
- winner gets award, looser gets penalty

Game as search:

- Initial state: board configuration of chess
- Successor function: list of (move,state) pairs specifying legal moves
- Goal test: is the game finished?
- Utility function: gives numerical value of terminal states. e.g. win (+1), loose (-1) and draw (0) in tic-tac-toe

MAX uses search tree to determine next move

Partial Game Tree for Tic-Tac-Toe



Optimal strategies

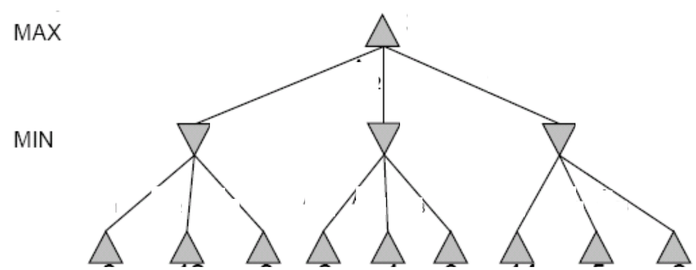
Find the contingent strategy for MAX assuming

- infallible MIN opponent
- both players play optimally

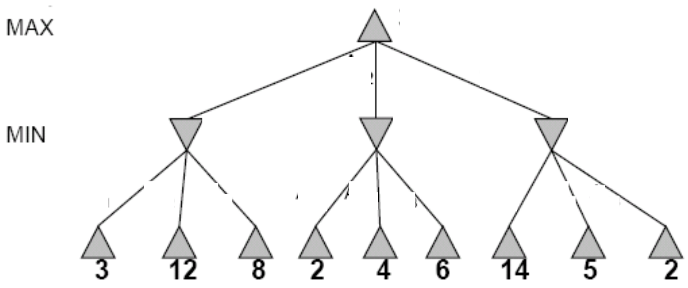
Given a game tree, the optimal strategy can be determined using the minimax value of each node:

$$\text{MINIMAX-VALUE}(n) = \begin{cases} \text{UTILITY}(n) & \text{If } n \text{ is a terminal} \\ \max_{s \in successors(n)} \text{MINIMAX-VALUE}(s) & \text{If } n \text{ is a MAX node} \\ \min_{s \in successors(n)} \text{MINIMAX-VALUE}(s) & \text{If } n \text{ is a MIN node} \end{cases}$$

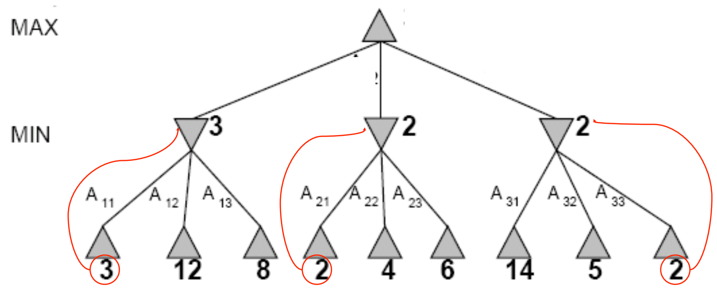
Two-Player Game Tree



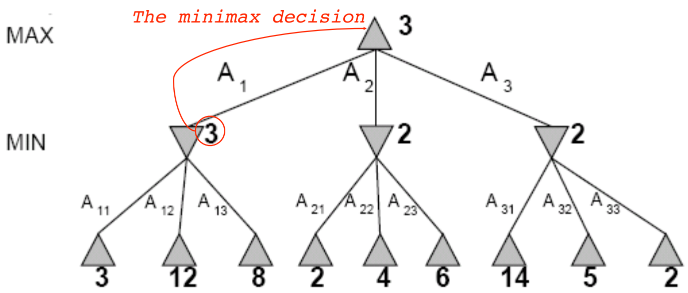
Two-Ply Game Tree



Two-Ply Game Tree



Two-Ply Game Tree



Minimax maximizes the worst-case outcome for max

Minimax Algorithm

Minimax ~backward induction

Max- and Min-Value are dual-recursive

```
function MINIMAX-DECISION(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) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← -∞
  for a,s in SUCCESSORS(state) do
    v ← MAX(v,MIN-VALUE(s))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a,s in SUCCESSORS(state) do
    v ← MIN(v,MAX-VALUE(s))
  return v
```

Properties of Minimax

Criterion	Minimax
Complete?	Yes ☺
Time	$O(b^m)$ ☹
Space	$O(bm)$ ☺
Optimal?	Yes ☺

Explores the entire tree...

...in a depth-first manner

m: max. depth
b: #legal moves

What if MIN does *not* play optimally?

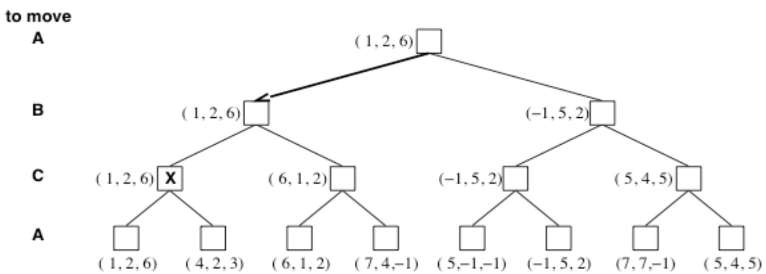
Definition of optimal play for MAX assumes MIN plays optimally
- maximizes worst-case outcome for MAX

But if MIN does not play optimally, MAX will do even better
[can be proved]

Minimax is the optimal strategy against optimal opponents, and still a very good one for suboptimal opponents

Multiplayer games

Games allow more than two players
- minimax values become vectors



Problem of Minimax search: complexity

Number of game states is *exponential* to the number of moves
- chess: average branching factor is~35 (=number of moves possible at a given), 35^5~50.000.000
- Minimax could look ahead only 5 moves (~novice level)

Solution: Do not examine every node!
- „Alpha-beta pruning“
  - Alpha = best score that can be forced for MAX, anything *worse* can be ignored because MAX can and will avoid it
  - Beta = worst-case scenario for MIN to endure, anything *better* can be ignored because MIN can and will avoid it

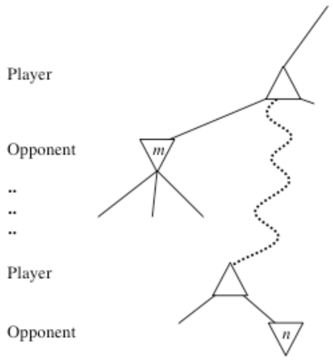
General alpha-beta pruning

Consider node n in the tree:

If *player* has a better choice at
- parent node of n
- or any choice point further up

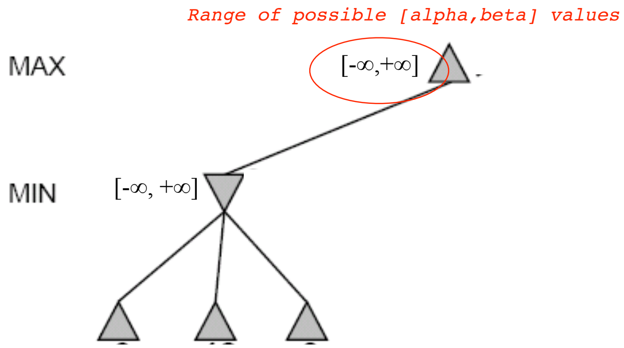
then n will never be reached in actual play and its child nodes simply don't matter

Hence when enough is known about n, it can be pruned

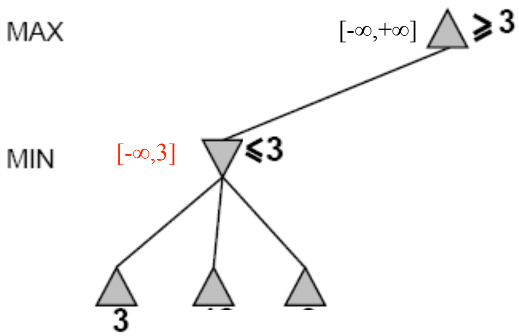


Alpha-Beta Example

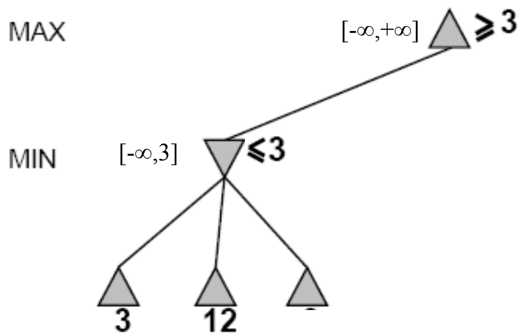
Do DF-search until first leaf



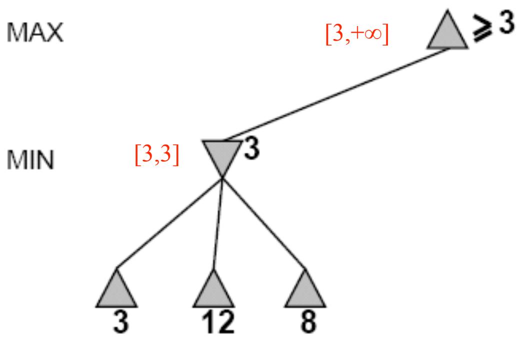
Alpha-Beta Example (continued)



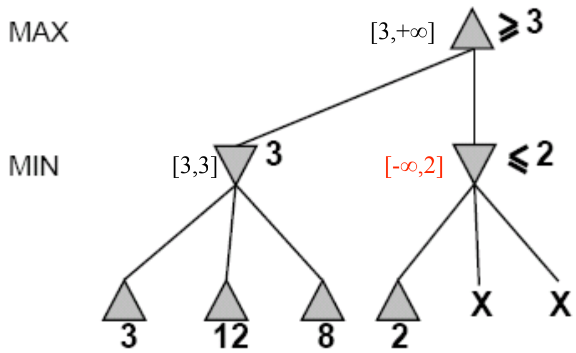
Alpha-Beta Example (continued)



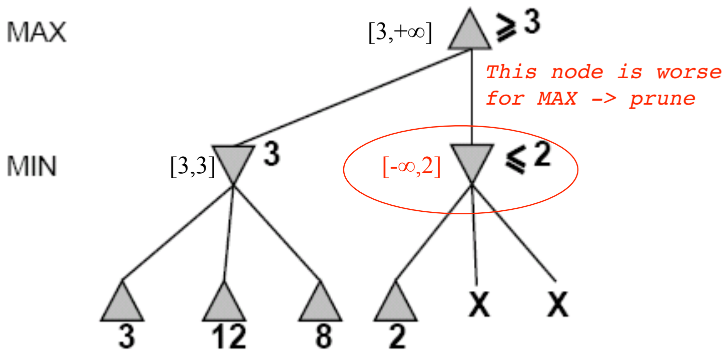
Alpha-Beta Example (continued)



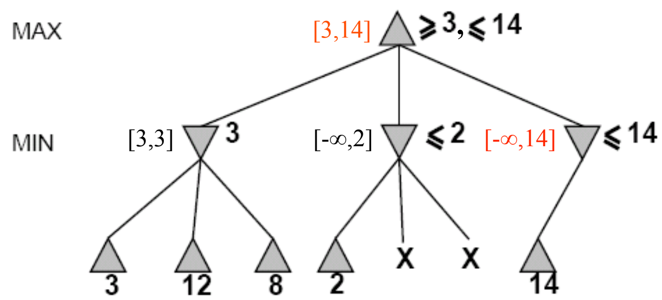
Alpha-Beta Example (continued)



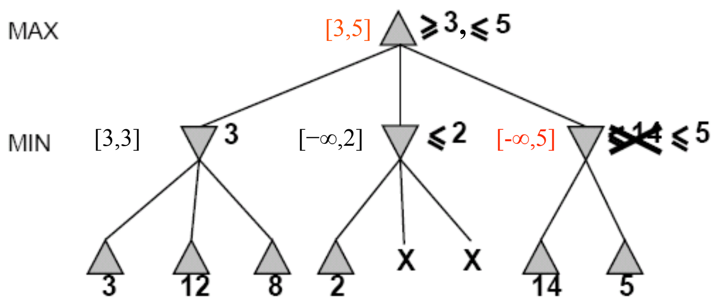
Alpha-Beta Example (continued)



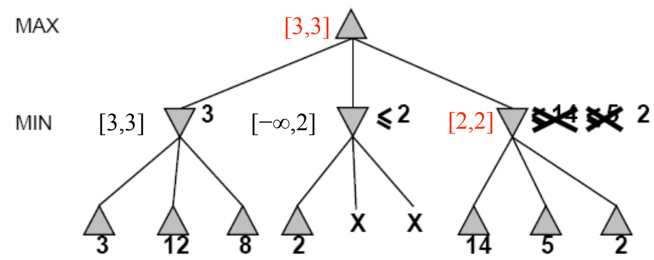
Alpha-Beta Example (continued)



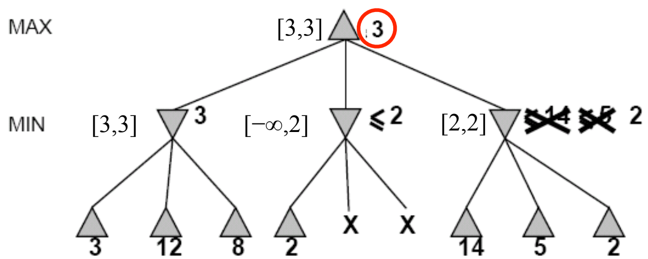
Alpha-Beta Example (continued)



Alpha-Beta Example (continued)



Alpha-Beta Example (continued)



Alpha-Beta Algorithm

function ALPHA-BETA-SEARCH(*state*) **returns** *action*
  inputs: *state*, current state in game
  $v \leftarrow$ MAX-VALUE(state, $-\infty$, $+\infty$)
  return the action in SUCCESSORS(state) with value v

function MAX-VALUE(*state*, α, β) **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, 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*
  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, α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Comments an alpha-beta

- Entire subtrees can be pruned, but pruning *must not* affect the final results
- *[alpha,beta]* is called search window; only moves with scores in this window are considered, all others are pruned
- Usually used in alpha-beta negamax variant
  - no need for different MAX-VALUES, MIN-VALUES functions
  - swaps and inverts alpha-beta between levels

```
int AlphaBeta(int depth, int alpha, int beta)
{
  if (depth == 0)  return Evaluate();
  GenerateLegalMoves();
  while (MovesLeft()) {
    MakeNextMove();
    val = -AlphaBeta(depth-1, -beta, -alpha);
    UnmakeMove();
    if (val >= beta)  return beta;
    if (val > alpha)
      alpha = val;
  }
  return alpha;
}
```


Comments an alpha-beta

Move order affects the effectiveness of pruning

- with worst ordering, it equates Minimax (no pruning effectively)
- with perfect ordering, complexity is $O(b^{m/2})$
  - Branching factor of sqrt(b), e.g. chess: 6 instead of ~35
  - Alpha-beta pruning can look *twice* as far as minimax in the same amount of time

Repeated states are still possible

- store evaluations in memory = transposition table
- can have dramatic effects, e.g. double search depth

Imperfect, real-time decisions

Minimax and alpha-beta pruning require too much leaf-node evaluations

- often impractical within a reasonable amount of time

Classical idea (Shannon 1950; for chess): Depth-limited game search

- Fixed-depth limit so that the amount of time will not exceed what the rules of the game allow
- Cut off search and use evaluation heuristic
  - replace
    if TERMINAL-TEST(**state**) **then return** UTILITY(**state**)
  - by
    if CUTOFF-TEST(**state, depth**) **then return** EVAL(**state**)

Using a heuristic EVAL function

EVAL function crucial, e.g., for pruning

- performance depends on quality of EVAL

Idea: produce an estimate of the expected utility of the game from a given position

Requirements:

- EVAL should order terminal-nodes in the same way as UTILITY
- computation must not take too long
- for non-terminal states, EVAL should be strongly correlated with the actual chance of winning

Heuristic EVAL example

Calculate features of the state, which would lead to wins/ draws/losses

- e.g. #pawns, #bishops, good pawn stucture, etc.
- weighted linear function of the features

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

Addition presumes feature independence

- better often use non-linear combinations

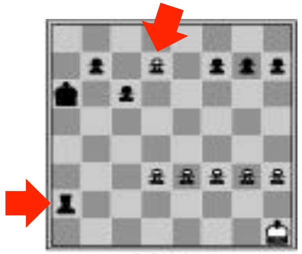
Features & weights encode game experience, not rules

- could be estimated by machine learning

Cut-off & heuristic difficulties

Horizon effect: moves that cause damage, but may eventually be unavoidable

- may be forestalled by own moves
- when pushed over the search horizon (depth limit), search doesn‘t see it anymore, thinks they have been avoided
- *singular extension:* search only moves that outperform all other; get deeper with branching factor 1



Black to move

Fixed depth search for Black thinks it can avoid the queening move by checking white king

Cut-off & heuristic difficulties

EVAL only useful for quiescent states = no wild swings in value in near future

Heuristic counts pieces won: (left) black ahead by one knight and two pawns and black will win, (right) white‘s next move will capture the black queen and black will loose

(a) White to move (b) White to move

Quiescent search

Non-quiescent states can be expanded until quiescent states are reached, usually testing moves like captures

When alpha-beta runs out of depth, a quiescent search function evaluates the position

- being careful to avoid overlooking obvious tactical conditions

*int **Quiesc**(state, α , β)*

- Calls EVAL for *state*
- If score is >β, a cutoff is immediately made (return β)
- If score isn't good enough to cause a cutoff, but is >α, α is updated
- „Good captures“ *s* are tried and tested with recursive call
  *v=-**Quiesc**(s, -α , -β)*
- When it comes back, check as above for β-cutoff

Can get deep if liberal definition of "good" capture is applied

Iterative Deepening

```
for (depth=1;; depth++) {
    val = AlphaBeta(depth, -∞, +∞);
    if (TimedOut())
        break;
}
```

Useful as a framework:

- alpha-beta is extremely sensitive to move ordering
- let alpha-beta return the move sequence predicted to be best for both sides: principal variation
- search it first in next iteration, because it tends to be very good
- can result in big improvements overall

Deep Blue (IBM, 1997)

generated 30 billion positions per move, reaching depth 14 routinely

iterative deepening alpa-beta search with transposition table, PLUS

- extensions beyond depth limit for interesting lines of moves (up to depth 40)
- heuristic evaluation function out of 8.000 features
- database of solved endgames (5-6 pieces)

Aspiration search

Speed up by small search (alpha-beta) windows!

Assumption: The value in the next iteration (depth+1) is not too much different from the value in the current iteration

Idea: Call alpha-beta with an artificially narrow aspiration window, centered around the previous search value. If the result is within that window, you've saved time

- alpha = previous - valWINDOW;
- beta = previous + valWINDOW;

If search fails, window must be widened again and search started again

Principal variation search (PVS)/NegaScout

Best variant of alpha-beta around, used in all good chess, checkers, etc. programs

Uses zero-width window if possible

Idea: If moves are in good order, all you need to do is to prove that first node is better than remaining nodes

- zero window size ~ test if actual score is equal to the guess

Also depends on node ordering !!

- techniques such as sorting the move list or storing best move in a hash table need to be employed

PVS/NegaScout algorithm

Searches first node with wide window, gives value *v*

- assuming that it is best, checks remaining nodes with null window *[v,v+1]* („scout test")
- if proof fails, 1st node was not best, repeat search with full-width window (like normal alpha-beta)

```
function NegaScout(node, depth, α, β)
  if node is terminal node or depth = 0
    return the heuristic value of node  (* cut-off *)
  b := β
  foreach child of node
    v := -NegaScout (child, depth-1, -b, -α)
    if α<v<β and not the first child   (* re-search *)
      v := -NegaScout(child, depth-1, -β, -v)
    α := max(α, v)
    if α≥β  return α   (* prune; cut-off *)
    b := α+1           (* set new null window *)
  return α
```

Aspiration NegaScout is at the heart of much of the best game-playing AI software around!

Recent trends: Memory-enhanced test algorithms

Make use of efficient memory (transposition tables!) and efficiency of runs with zero window size

Idea: Alpha-beta with zero-size window *[gamma,gamma+1]* will fail either high or low; this gives an upper or lower bound on minimax value

- run multiple times to converge on the real value
- need good first guess, often used with iterative deepening, re-using previous value as next first guess

The state of the art for some games

Chess:

- 1997: IBM Deep Blue defeats Kasparov
- … there is still debate about whether computers are really better

Checkers:

- Computer world champion since 1994
- … there was still debate about whether computers are really better…
- until 2007: checkers solved optimally by computer

Go:

- Computers still not very good, branching factor really high
- Some recent progress with heuristic probabilistic methods
  - e.g.: http://senseis.xmp.net/?UCT

Poker:

- Competitive with top humans in some 2-player games
- 3+ player case much less well-understood

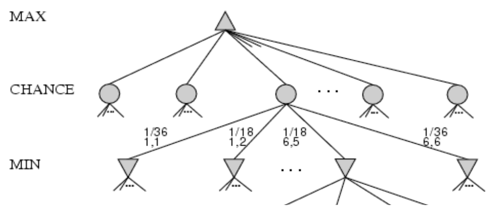
Games that include chance

Many games combine luck and skill, e.g. Backgammon

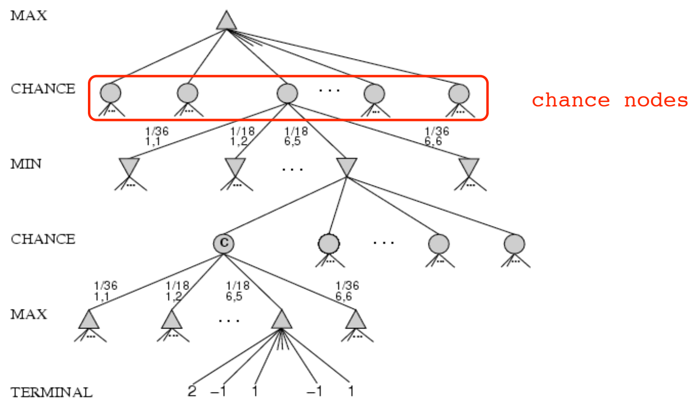
- impossible to construct standard game-tree, because opponent's legal moves are not clear

Build game tree with additional chance nodes

- branches denote possible dice rolls, labelled with chances they occur



Example: Backgammon



Possible moves (5-10,5-11), (5-11,19-24),(5-10,10-16) and (5-11,11-16)
[1,1], [6,6] chance 1/36, all other chance 1/18

Games that include chance

Can not calculate definite minimax value, only the expected value taken over all possible dice rolls

Generalize to: EXPECTIMINIMAX(n)=

$$\begin{cases} \text{UTILITY}(n) & \text{If } n \text{ is a terminal} \\ \max_{s \in successors(n)} \text{EXPECTIMINIMAX}(s) & \text{If } n \text{ is MAX node} \\ \min_{s \in successors(n)} \text{EXPECTIMINIMAX}(s) & \text{If } n \text{ is MIN node} \\ \boxed{\sum_{s \in successors(n)} P(s) \,.\, \text{EXPECTIMINIMAX}(s) \quad \text{If } n \text{ is chance node}} \end{cases}$$

Can be backed-up recursively all the way to the root of the game tree as in minimax

Games that include chance

EXPECTIMINIMAX takes $O(b^m n^m)$, where n is number of distinct dice rolls

- unealistic to look far ahead, e.g. Backgammon: sometimes not more than 3 plies

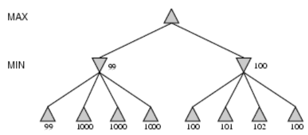
Problem:

- alpha-beta ignores suboptimal developments, concentrating on likely plays
- BUT with chance, there are no likely sequences of moves and possibilities are multiplied enormously

One can prune chance nodes:

- with bounds on utility function, one can have bounds on average
- Example: all utilities are +3...-3 -> can place upper bound on value of chance node without looking at its children

Summary



There is more than just taking the standard approach to the max

Minimax does not care about approximative nature of evaluations

- Better: evaluation gives probability distribution over possible values, may get expensive

Alpha-Beta pruning does much irrelevant calculations, e.g. computing bounds in a „clear favorite“ situation

- Better: consider utility of node expansion by some sort of metareasoning in decision making

Search algorithms construct all possible sequences

- Better: generate plausible plans for certain goals, based on experience