Playing games

Outline

- ♢ Games
- ♢ Perfect play
  - – minimax decisions
  - – α–β pruning
- ♢ Resource limits and approximate evaluation
- ♢ Games of chance
- ♢ Games of imperfect information

Games vs. search problems

Adversarial search: our first example of a (competitive) multi-agent setting !

"Unpredictable" opponent ⇒ solution is a strategy that specifies a move for every possible opponent's reply

There are time limits ⇒ we are unlikely to find goal and must approximate

One of the first AI problems studied:

- Computer considers possible lines of play (Babbage, 1846)
- Algorithm for perfect play (Zermelo, 1912; Von Neumann, 1944)
- Finite horizon, approximate evaluation (Zuse, 1945; Wiener, 1948; Shannon, 1950)
- First chess program (Turing, 1951)
- Machine learning to improve evaluation accuracy (Samuel, 1952–57)
- Pruning to allow deeper search (McCarthy, 1956)

Types of games

	deterministic	**chance**
perfect information	chess, checkers, Go, Othello	backgammon, Monopoly
imperfect information	battleships, blind tic-tac-toe	bridge, poker, Scrabble, nuclear war

Two-player, fully observable, zero-sum games

Players are commonly called MIN and MAX
- take turns (each turn is called ply)
- make one move at each turn

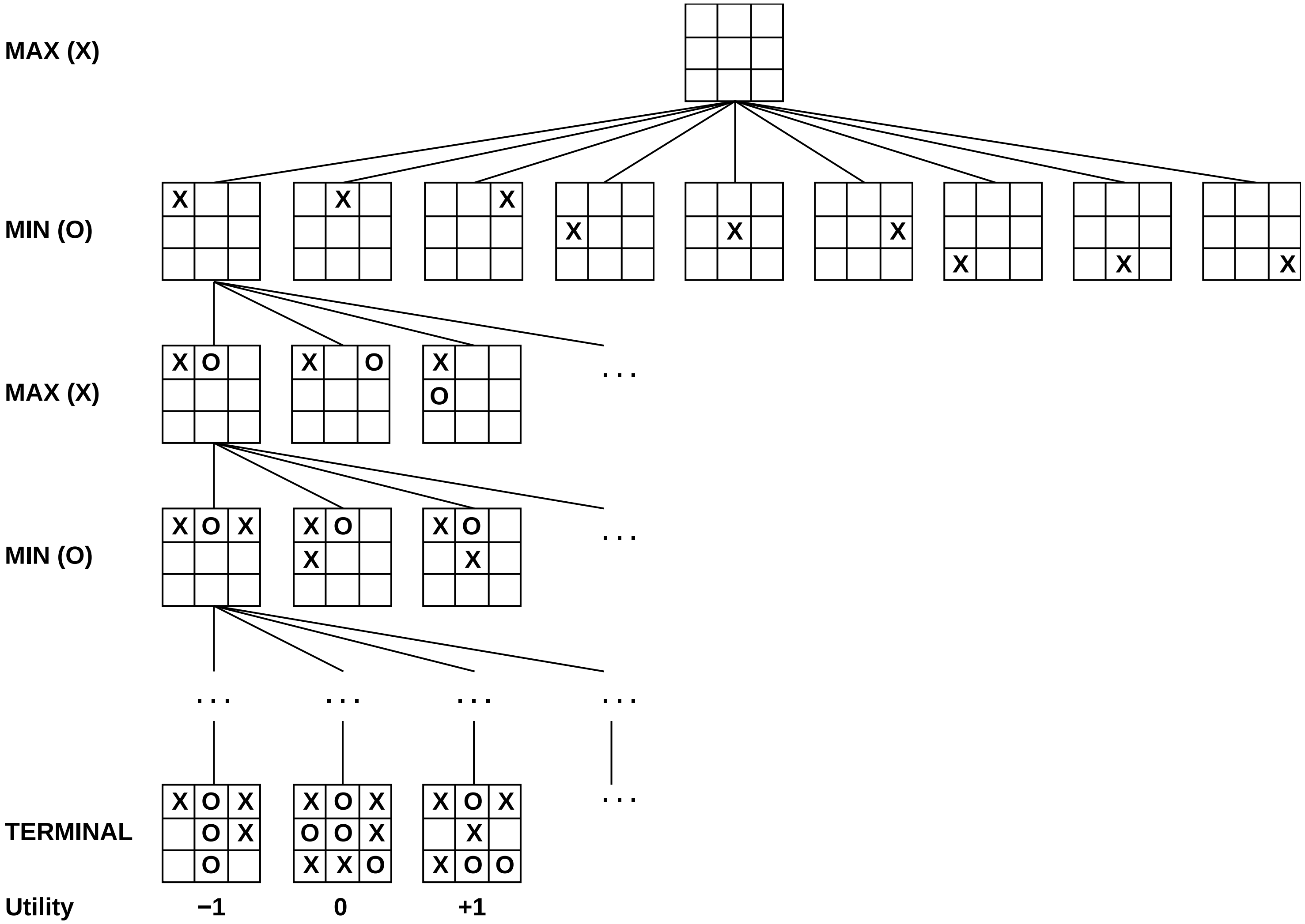
The game is zero-sum: one player's loss is another player's win
- Total reward is fixed, to be divided between players.

Example: chess, checkers, Go, Othello

⇒ All possible plays can be described by a game tree
- root contains the initial board position
- nodes in levels are labeled with MIN and MAX (alternation)
  - usually MAX makes the first move
- labels leaves with a utility function value (from MAX's point of view)
  - e.g., wins are labeled with +1, draws with 0, and losses with -1

Game tree can be used to identify an optimal strategy
- determines the outcome of the game even without playing!

Game tree



Optimal Strategies

In a search problem, optimal solution is a sequence of moves leading to goal.

In a game, MAX must find a **contingent strategy** specifying
– MAX's move in the initial state,
– MAX's moves in the states resulting from each response from MIN to those moves
– and so on. . .

Optimal Strategy: leads to outcomes at least as good as any other strategy
– even when one is playing with an infallible oponent.

Minimax algorithm

Computes the optimal strategy

Idea: propagate utility from leaves upwards (= $\textsc{Minimax}$)

- $\textsc{max}$ should maximize utility
- $\textsc{min}$ should minimize utility

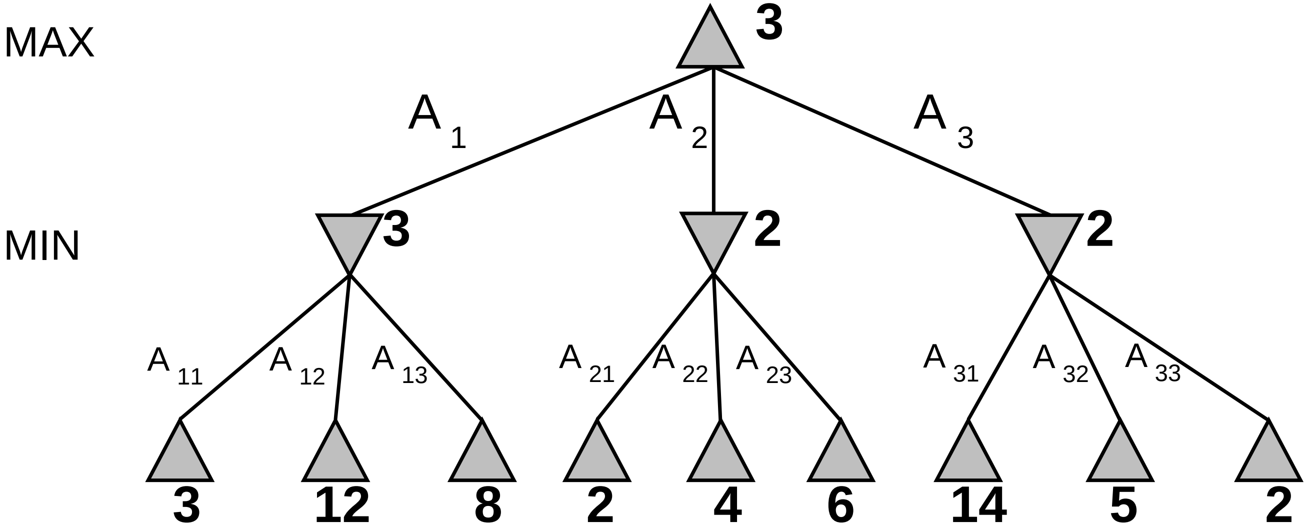
$$\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } s \text{ is a terminal node} \\ \max_{a \in \textsc{Actions}(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } s \text{ is a } \textsc{max} \text{ node} \\ \min_{a \in \textsc{Actions}(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } s \text{ is a } \textsc{min} \text{ node} \end{cases}$$

Optimal strategy determined by examining minimax value for each node.

Minimax decision for $\textsc{max}$: play a move at root that yields a successor with the largest $\textsc{Minimax}$ value

- guarantees optimal result even if $\textsc{min}$ plays optimally
  - i.e., if $\textsc{min}$ plays moves yielding the smallest $\textsc{Minimax}$
- even better if $\textsc{min}$ does not play optimally

Minimax example: Simple 2-ply game



Minimax algorithm

```
function MINIMAX-DECISION($s$) returns an action
    return $a$ in ACTIONS($s$) that maximizes MIN-VALUE(RESULT($s, a$))

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

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

Properties of minimax

Remember it amounts to DFS exploration of game tree.

Complete?? Yes, if tree is finite (chess has specific rules for this)

Optimal?? Yes, even 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
$\Rightarrow$ exact solution completely infeasible

But do we need to explore every path?

α–β **pruning**

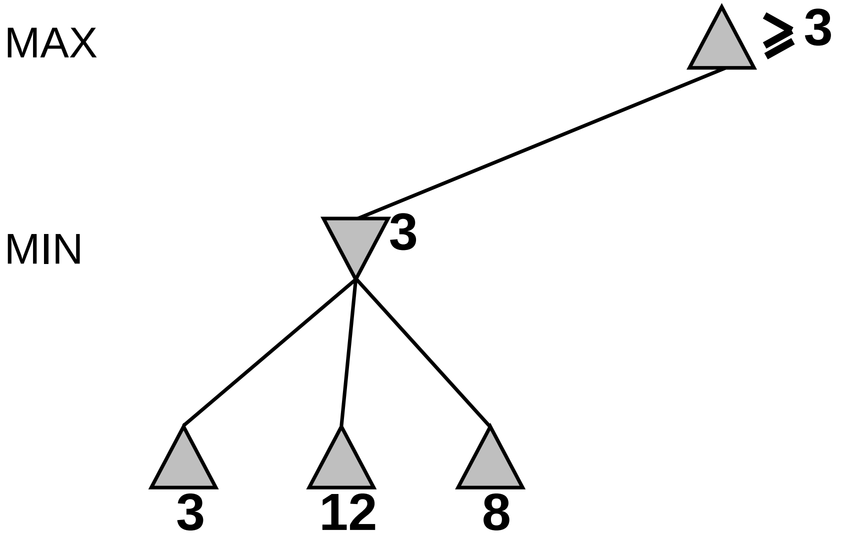
Idea: Prune away branches that cannot possibly influence decision at root.

Consider node n in game tree
– If player has a better choice m either at the parent of n or at any other choice further up, then n will never be reached in actual play
– Prune n as soon as we have found out enough about n (by examining some of its descendants) to reach that conclusion.

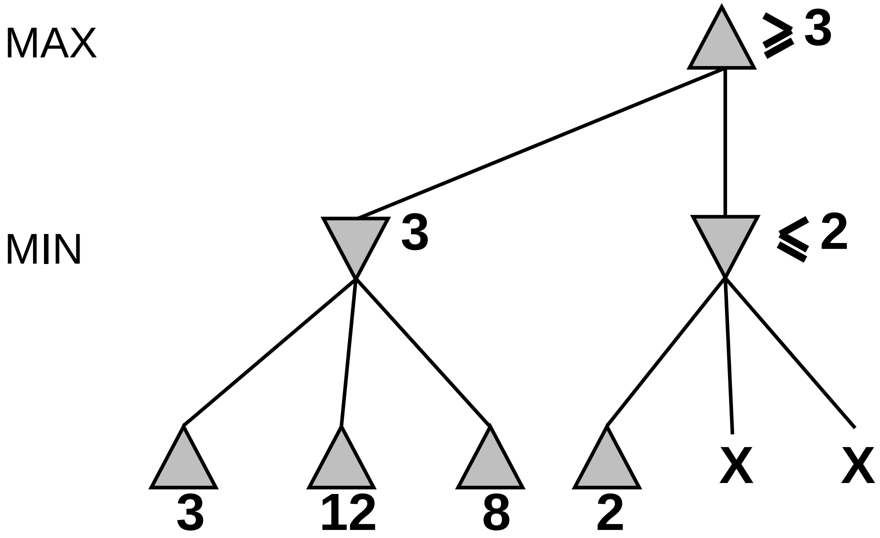
For this, keep a range of posible values for each node.
– α: value of best (highest) choice found so far at any choice point along path for MAX
– β: value of best (lowest) choice found so far at any choice point along path for MIN

Search updates values of α and β and prunes remaining branches at node as soon as value of node is worse than the current α or β value for MAX or MIN, respectively.

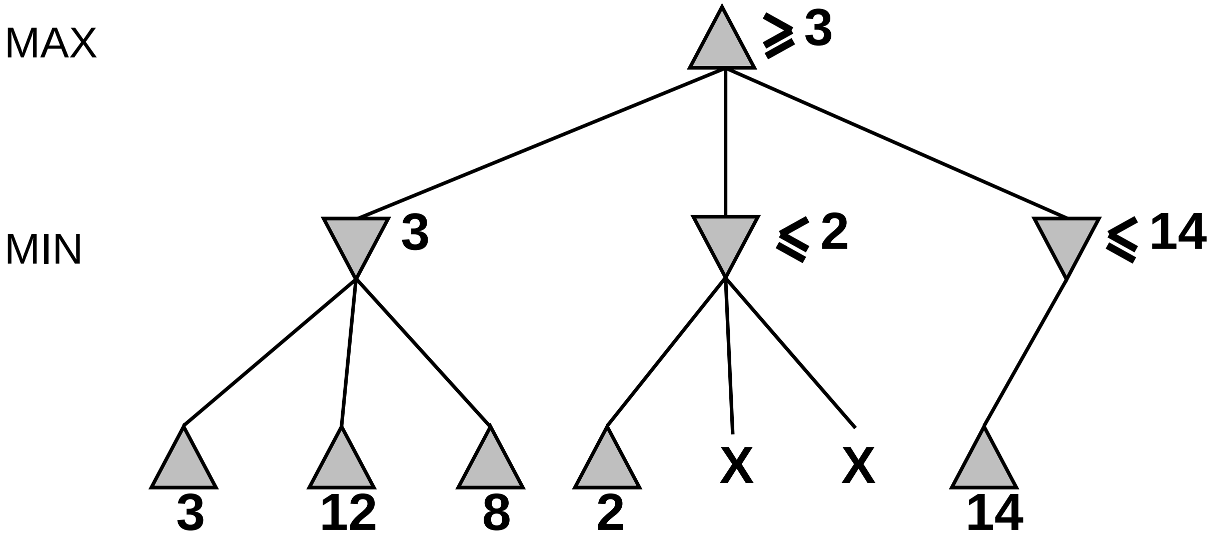
α–β pruning example



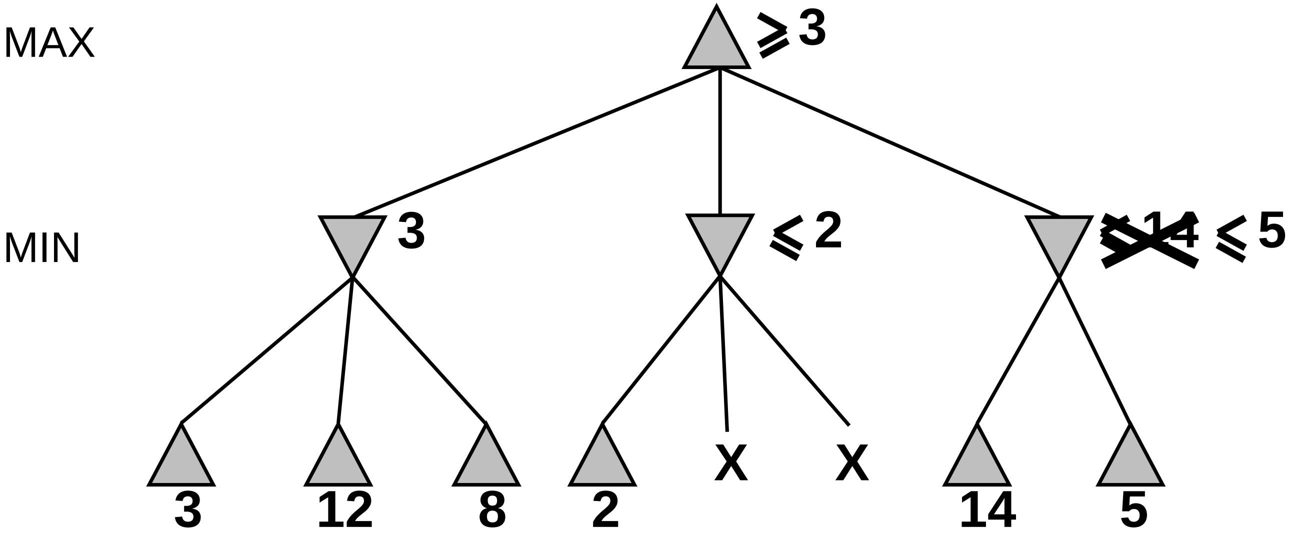
α–β pruning example



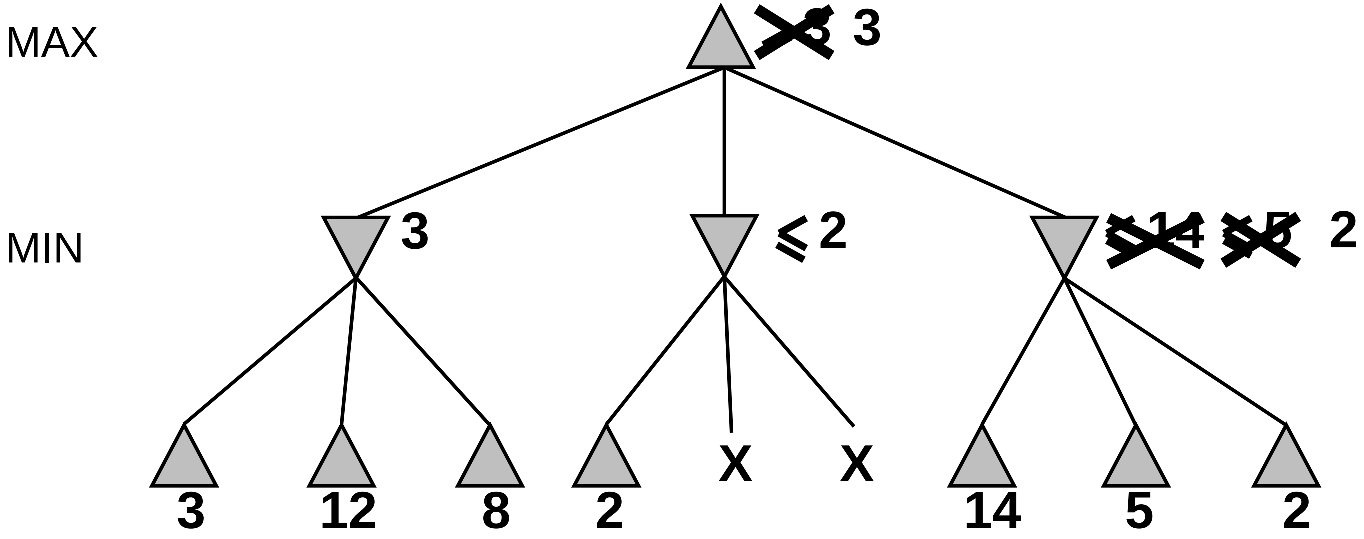
α–β pruning example



α–β pruning example



α–β pruning example



The α–β algorithm

```
function ALPHA-BETA-SEARCH($s$) returns an action
  return $a$ in ACTIONS($s$) that maximizes MIN-VALUE(RESULT($s, a$), $-\infty, +\infty$)
```

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

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

Properties of α–β

Pruning **does not** affect the final result

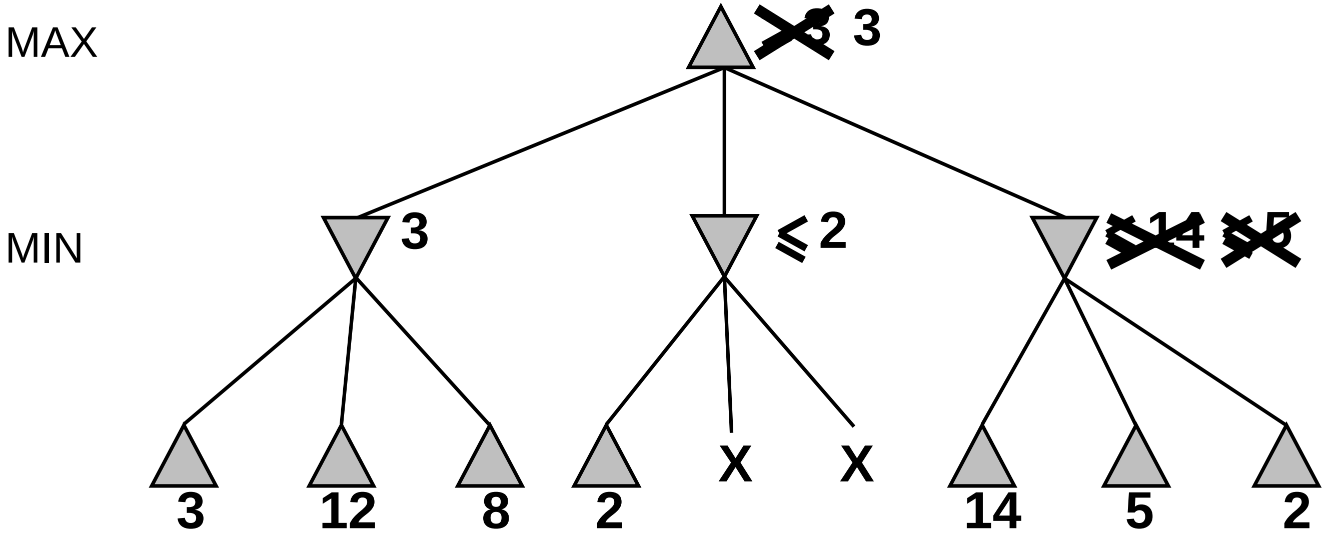
Good move ordering improves the effectiveness of pruning

With "perfect ordering" we get time complexity = $O(b^{m/2})$
$\Rightarrow$ **doubles** solvable depth

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

Unfortunately, 35^{50} still does not allow us to solve chess!

Move ordering



On the last branch, had we first examined the right-most move (resulting in 2), we could have avoided looking at moves leading to 14 and 5

⇒ Good move ordering significantly improves the search

– use hard-coded heuristics

– learn form past moves

Dealing with resource limits

Standard approach:

– use Cutoff-Test instead of Terminal-Test

e.g., depth limit

– use Eval instead of Utility

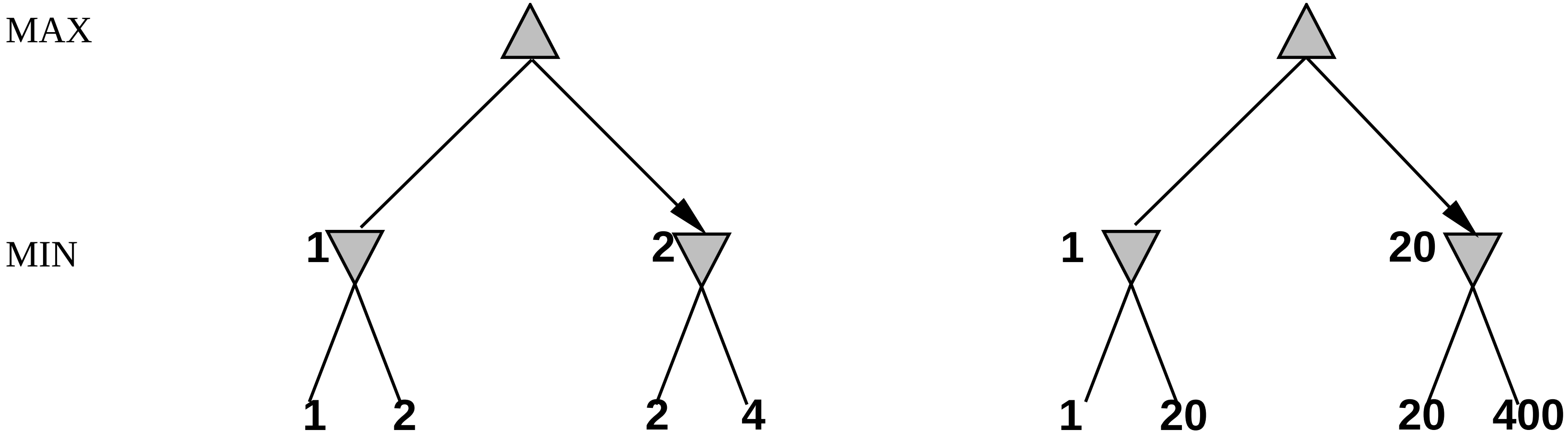
i.e., evaluation function that estimates desirability of position

Suppose we have 100 seconds and we explore 10^4 nodes/second

$\Rightarrow$ 10^6 nodes per move $\approx 35^{8/2}$

$\Rightarrow$ α–β reaches depth 8 $\Rightarrow$ pretty good chess program

Exact values of EVAL do not matter



Any **monotonic** transformation of UTILITY preserves optimality

$\text{EVAL}(n) \leq \text{EVAL}(n')$ if and only if $\text{UTILITY}(n) \leq \text{UTILITY}(n')$

– usually not the case in practice

⇒ In deterministic games we just need to **order** states

Practical evaluation functions

EVAL should order terminal states as UTILITY
– otherwise we could miss wins even if we can explore entire game tree

Should reasonably approximate UTILITY on nonterminal states

Computation should not take too long

Defining evaluation functions

Common approach: define Eval as a linear combination of features

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

In chess:

- f_i = the number of pieces on the board; w_i = the utility of pieces
- "good pawn structure"
- "king safety"
- …

We can also use nonlinear combinations of features — e.g., in chess…

- two pieces may be more useful together than each piece independently
- a piece may have different utility at various states of the game

⇒ Requires huge amounts of expertise!

Cutting off search

Straightforward approach: a priori depth limit

Improvement: iterative deepening

– increase depth until the time runs out
– return the best move detected thus far
⇒ can be adapted to learn good move orderings

Cut off at quiescent states (i.e., where Eval is unlikely to change rapidly)

Horizon effect: limit on search depth can postpone inevitable moves

Search vs. lookup

Idea:

– precompute a database of positions and the “best moves”

– use lookup during play instead of search

– often applicable to certain parts of the game

Chess openings were known for a century

– The Oxford Companion to Chess lists 1,327 named openings

Endings can be precomputed by backwards search

– start with a winning position and apply moves “backwards”

– better performance in chess than humans!

– solved checkers!

Deterministic games in practice

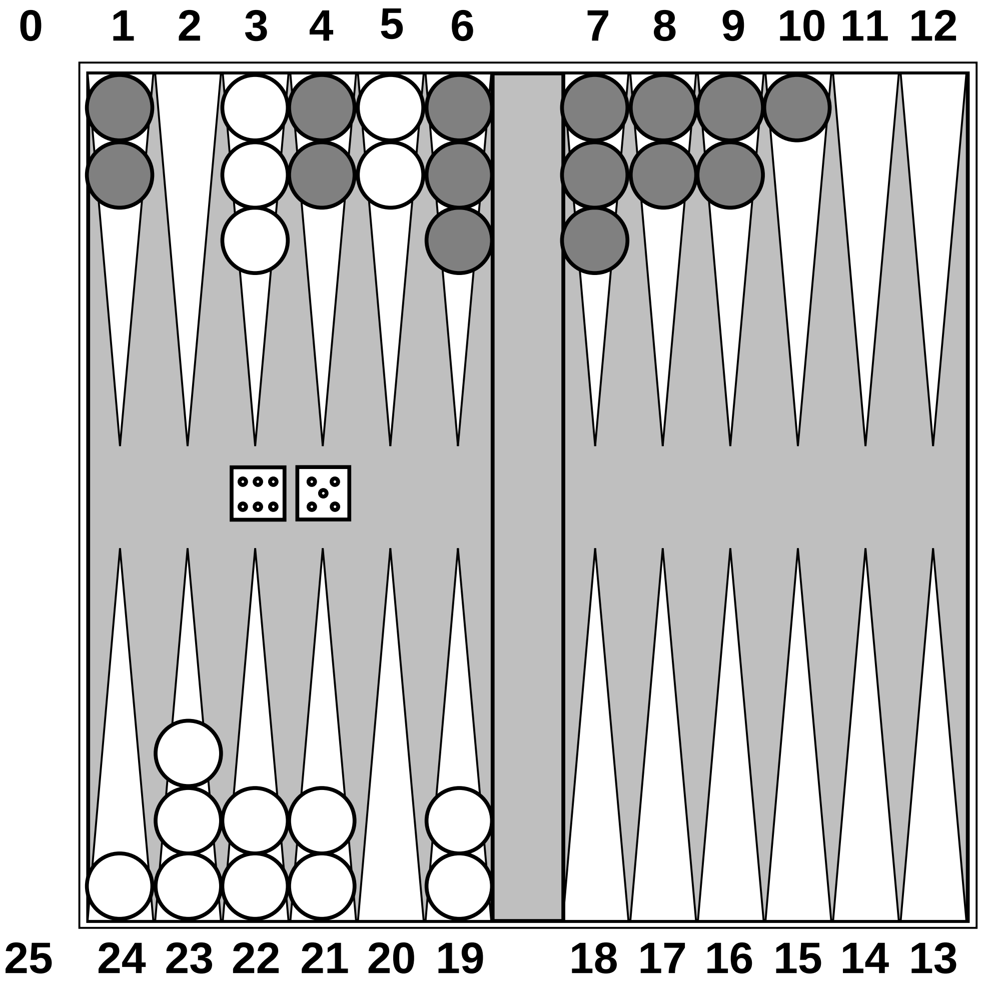
Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 443,748,401,247 positions. Current version of Chinook cannot lose.

Chess: Deep Blue defeated human world champion Gary Kasparov in a six-game match in 1997. Deep Blue searches 200 million positions per second, uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply.

Othello: human champions refuse to compete against computers, who are too good.

Go: human champions refuse to compete against computers, who are too bad. In Go, $b > 300$, so most programs use pattern knowledge bases to suggest plausible moves.

Nondeterministic games: backgammon

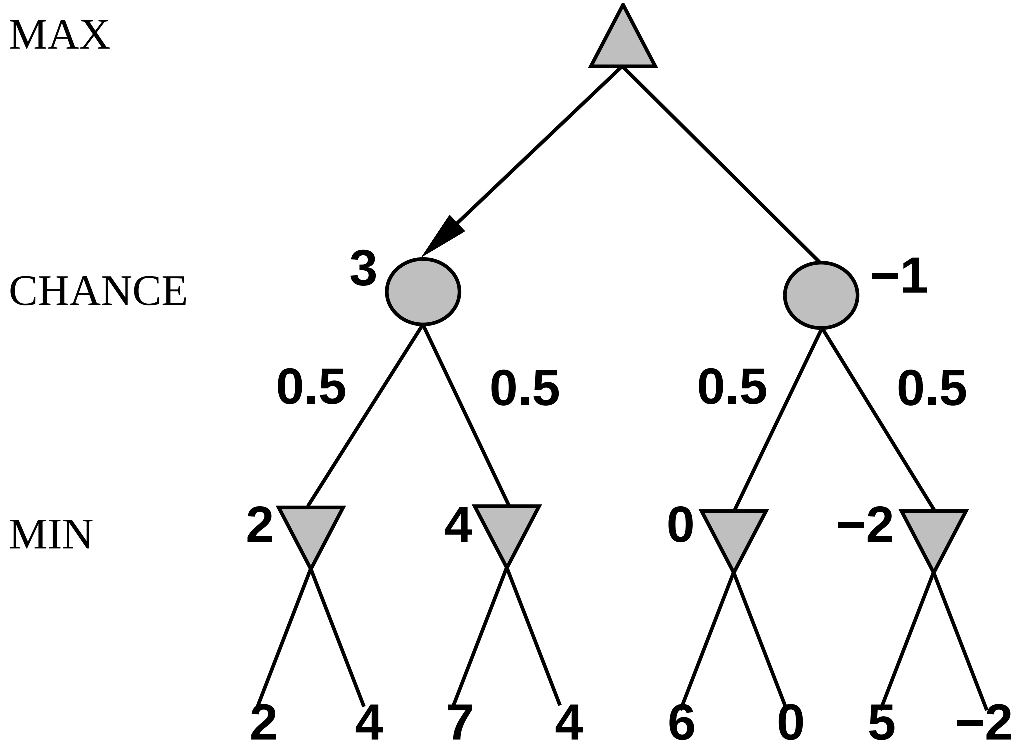


Nondeterministic games in general

Chance is introduced by dice, shuffling of cards, . . .

Game tree includes chance nodes

⇒ have outgoing edges labeled with probabilities



Nondeterministic games in general

We still want to pick the move that leads to the best position
– but, resulting positions do not have definite minimax values !
– we can only calculate the **expected value**, where expectation is taken over all possible dice rolls.

Idea to extend minimax:
– evaluate terminal, MIN and MAX nodes as before
– evaluate chance nodes by taking weighted average of values resulting from all possible dice rolls.

Expectiminimax algorithm

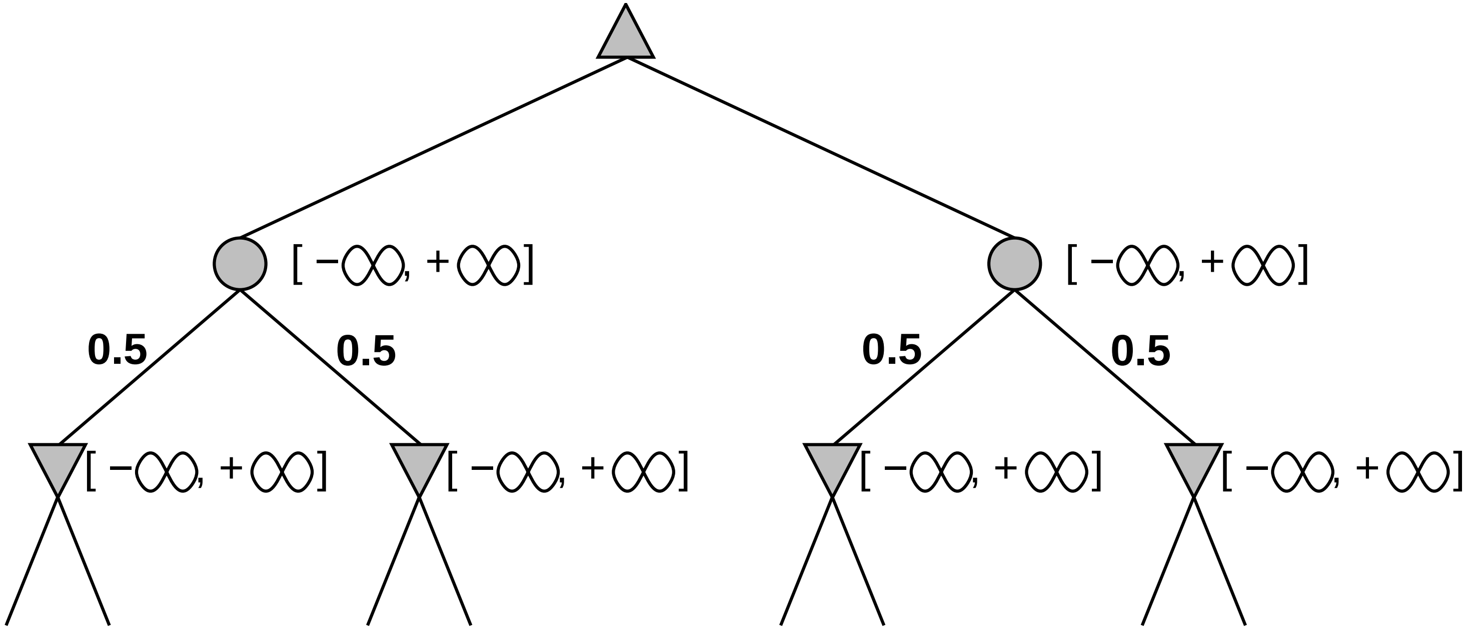
Similar to Minimax, but also handles chance nodes; yields perfect play (maximises expected values)

$$\textsc{ExpectiMM}(s) =$$

$$\begin{cases} \textsc{Utility}(s) & \text{if } s \text{ is a terminal node} \\ \max_{a \in \textsc{Actions}(s)} \textsc{ExpectiMM}(\textsc{Result}(s,a)) & \text{if } s \text{ is a } \textsc{max} \text{ node} \\ \min_{a \in \textsc{Actions}(s)} \textsc{ExpectiMM}(\textsc{Result}(s,a)) & \text{if } s \text{ is a } \textsc{min} \text{ node} \\ \Sigma_r P(r) \textsc{ExpectiMM}(\textsc{Result}(s,r)) & \text{if } s \text{ is a chance node} \end{cases}$$

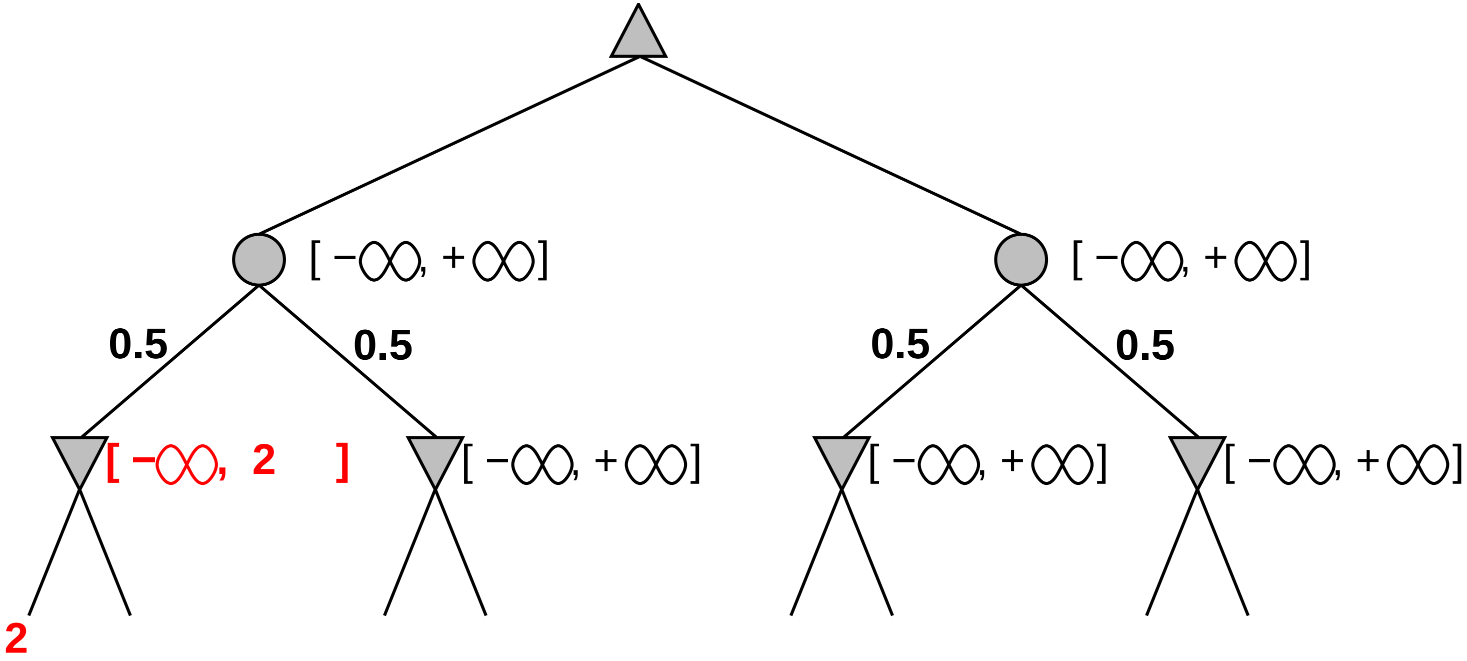
Pruning in nondeterministic game trees

A version of α-β pruning is possible:



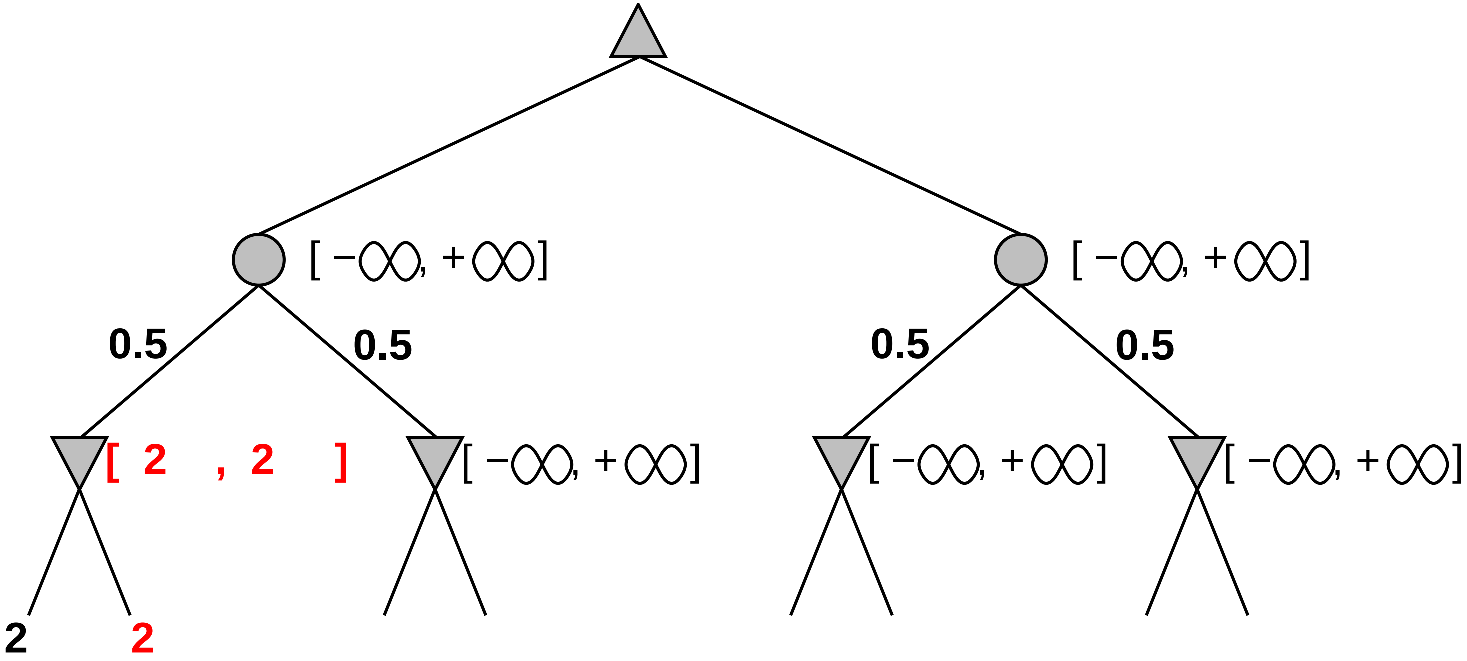
Pruning in nondeterministic game trees

A version of α-β pruning is possible:



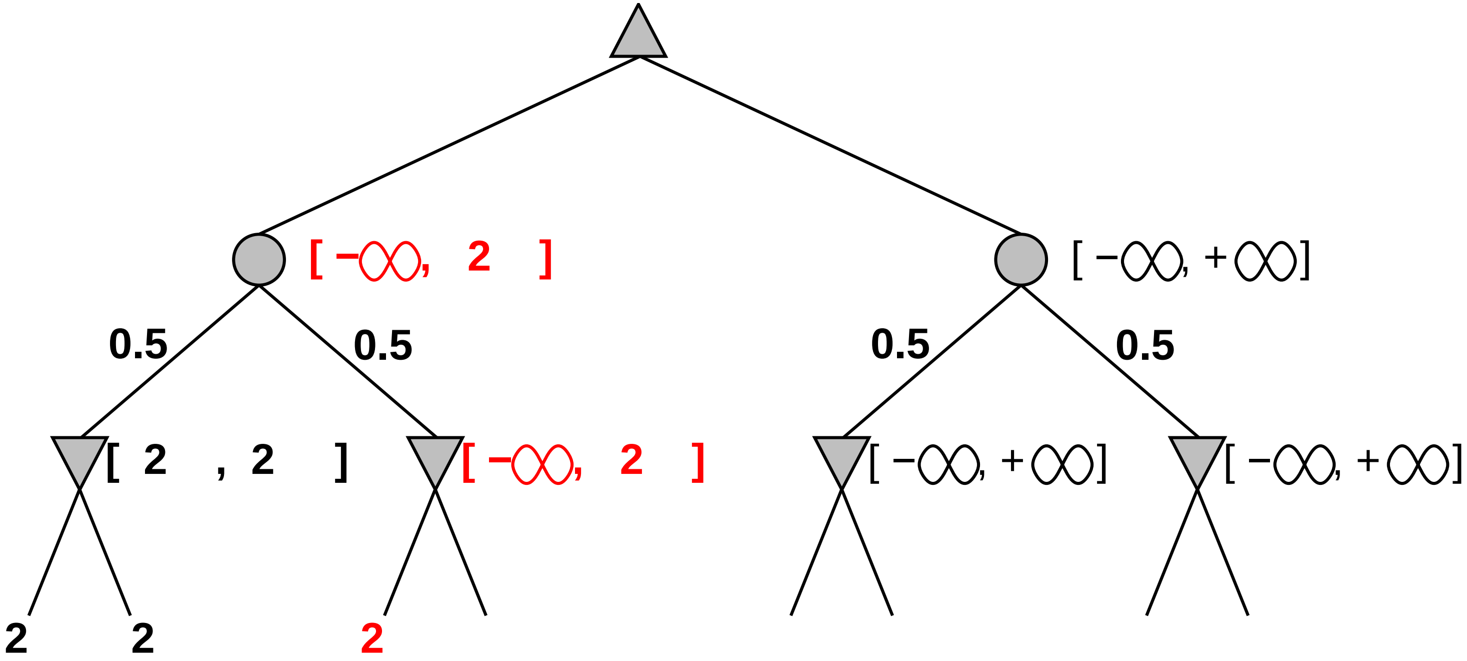
Pruning in nondeterministic game trees

A version of α-β pruning is possible:



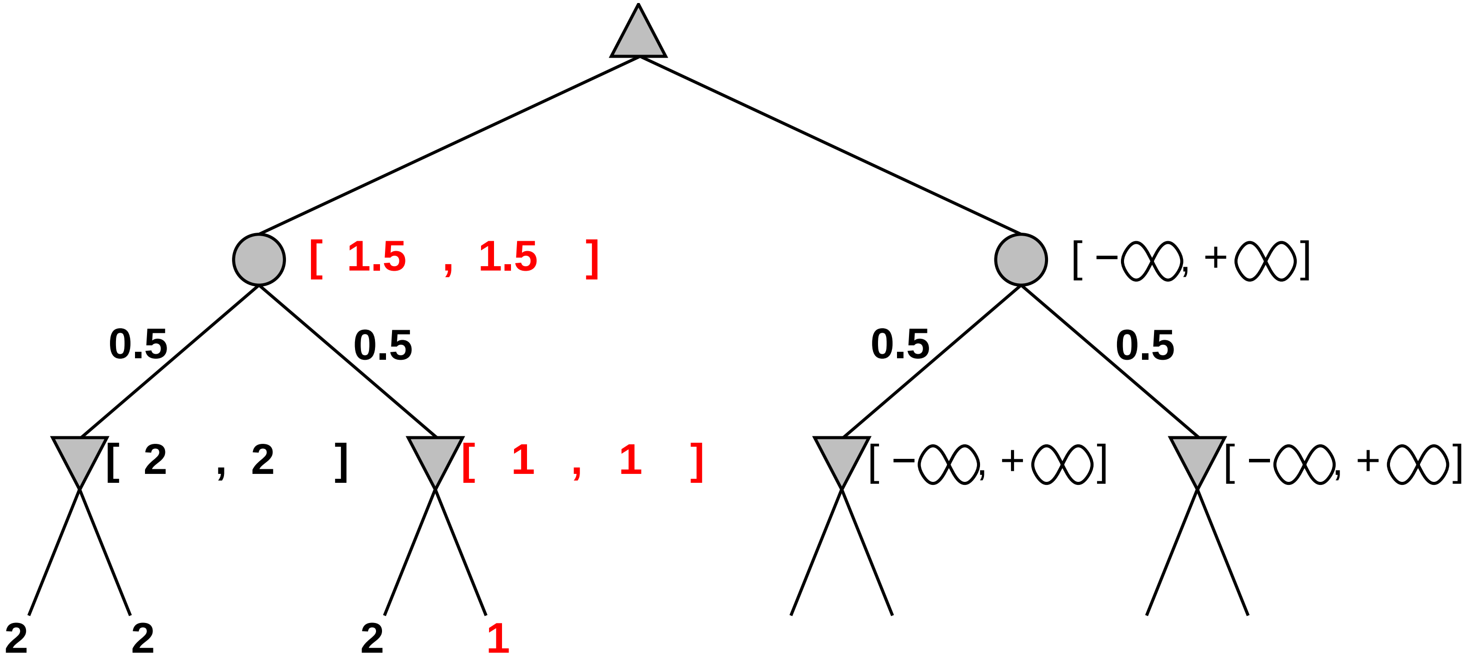
Pruning in nondeterministic game trees

A version of α-β pruning is possible:



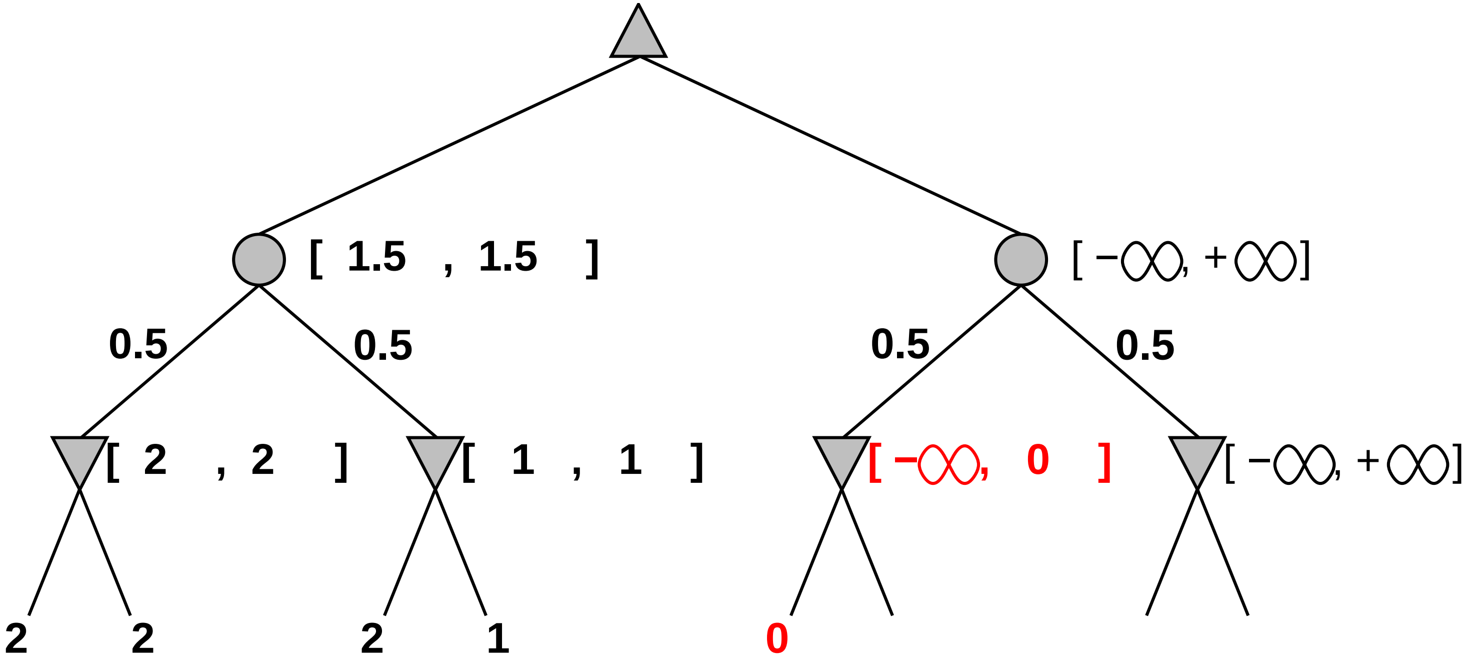
Pruning in nondeterministic game trees

A version of α-β pruning is possible:



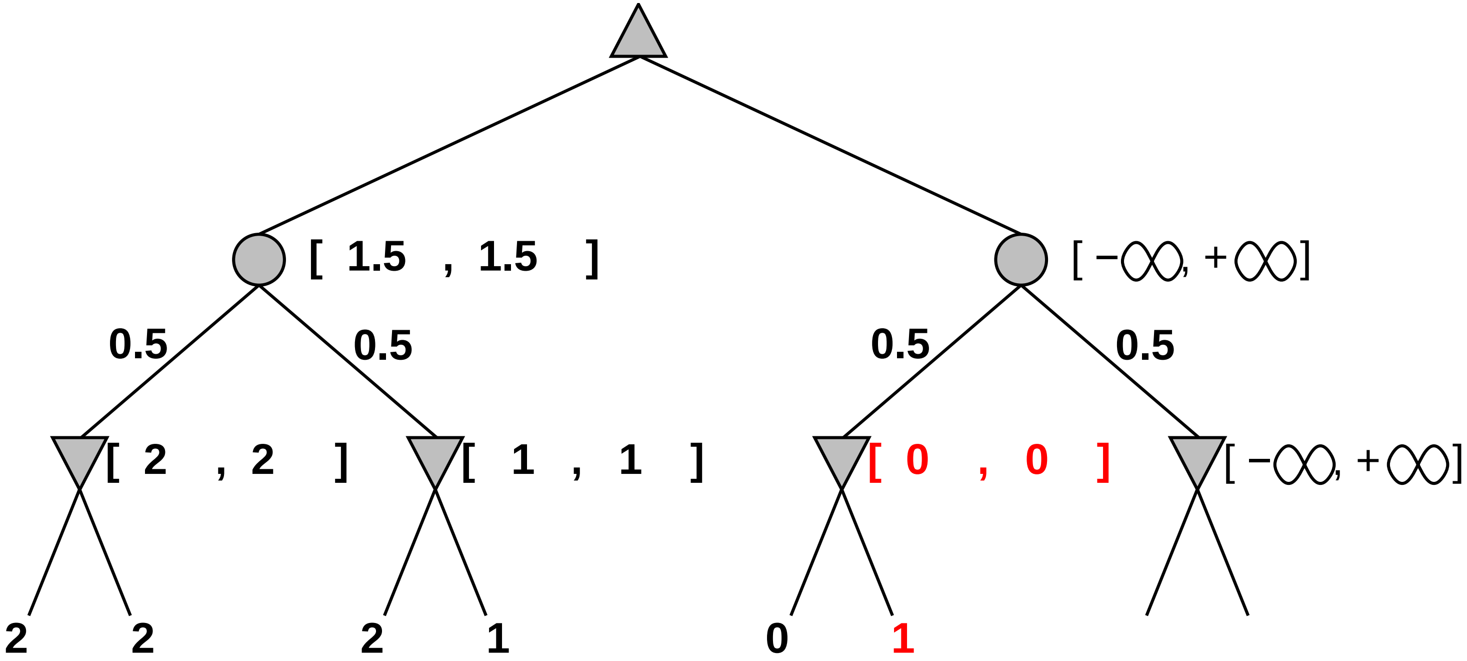
Pruning in nondeterministic game trees

A version of α-β pruning is possible:



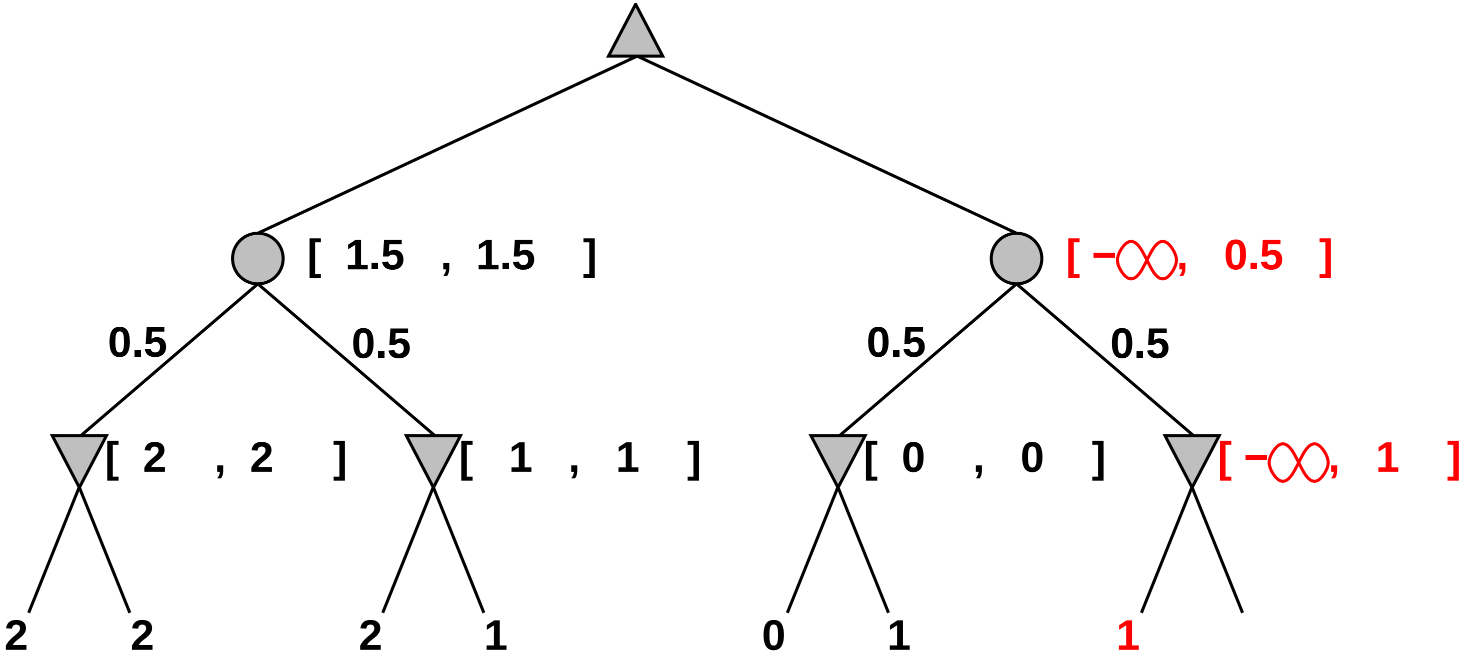
Pruning in nondeterministic game trees

A version of α-β pruning is possible:

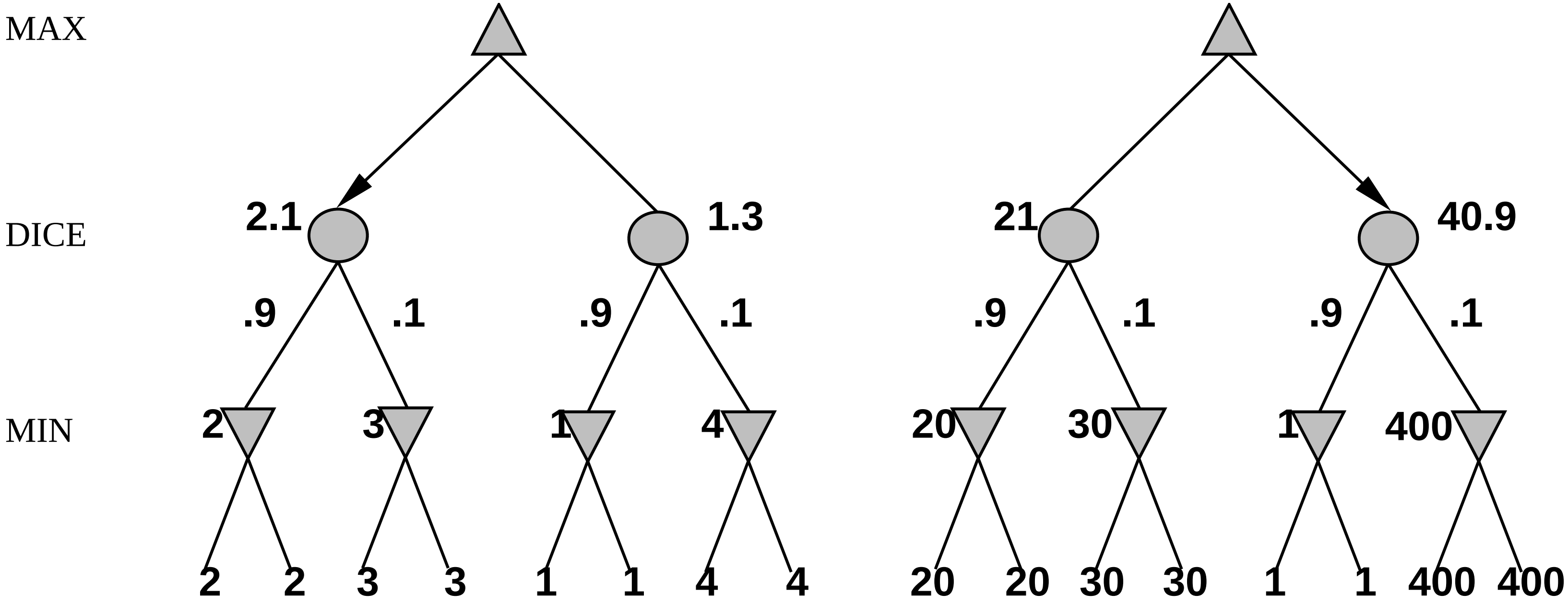


Pruning in nondeterministic game trees

A version of α-β pruning is possible:



Exact values of EVAL DO matter



Behavior is preserved only by **positive linear** transformation of UTILITY

Hence EVAL should be proportional to the expected payoff

Nondeterministic games in practice

Complexity of expectiminimax: $O(b^m n^m)$

– b: the branching factor for moves

– n: the number of rolls

– m: the maximum depth of the tree

⇒ dice rolls increase the branching factor:

Example: 21 possible rolls with 2 dice

Backgammon: 20 legal moves per roll on average (can be 6,000 for 1-1 roll)

⇒ for depth 4, we have $20 \times (21 \times 20)^3 \approx 1.2 \times 10^9$

As depth increases, 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

≈ world-champion level

Monte Carlo simulation

Applicable to deterministic and nondeterministic games

Determine the next move from position P:
- play a random game starting from P (= playout)
- record for the initial move the result of the game (win/loss/draw)
- repeat until no more time
- play the initial move that lead to wins most often

Playout types:
- truly random—each move is equally likely
- move probability is weighted based on utility

Surprisingly good results on simple games
- constant space complexity (no need to store search tree)

Monte Carlo tree search: probabilistic simulation + search
- eliminates choice nodes on nondeterministic games

Summary

Games are fun to work on!

They illustrate several important points about AI:

– perfection is unattainable ⇒ must approximate

– good idea to think about what to think about

– uncertainty constrains the assignment of values to states

– optimal decisions depend on information state, not real state

Games are to AI as grand prix racing is to automobile design