Adversarial Search

AIMA.3rd Chapter 5

Outline

◇ Games

◇ Optimal decisions in games
 – minimax decisions
 – α–β pruning

◇ Imperfect real-time decisions

◇ Stochastic games - game of chance

◇ Partially observable games - games of imperfect information

Games vs. search problems

Game theory: a branch of economics which views any multiagent environment (competitive or cooperative) as a game (provided that the impact of the agents on each other is significant).
The presence of an opponent introduces uncertainty which in turn makes the decision problem more complicated than search problems.
Unpredictable opponent $\longrightarrow$ solution is a strategy specifying a move for every possible opponent reply

Characteristics of adversarial search problems, also known as games:

$\diamondsuit$ multiagent environment

$\diamondsuit$ competitive environment
in which the agents' goals are in conflict

$\diamondsuit$ stochastic "Unpredictable" opponent $\rightarrow$ solution is a strategy specifying a move for every possible opponent reply

Game description

A game can be formally defined as a kind of search problem with:

- ♢ initial state: specifies how the game is set up at the start.
- ♢ PLAYES(s): Defines whiwh palyer has the move in a state
- ♢ ACTIONS(s): set of operators (which define the legal moves)
- ♢ RESULT(s,a): transition model that defines the result of the move
- ♢ TERMINAL-TEST(s): terminal test (goal test)
- ♢ UTILITY(s,p): utility function final numeric value for the outcome of a game in terminal s for player p

Ex. backgammon (+1, -1, +2); Chess (win, lose, draw)...

Game description

♢ Zero-sum games: if one opponent gains, the other loses an equal amount (i.e. they are using opposite utility functions). More general concept is constant-sum games.
♢ Non-zerosum games: opponents may join forces to increase their gains together.

The initial state, ACRTIONS function, and RESULT function define the game tree – a three where the nodes are game states and the edges are moves.
Search tree is a tree that is subtree of the full game tree that traces the nodes and edges examined by a player to determine what move to make.

Games vs. search problems: time constraints

Real problem is that games are usually much too hard to solve:

In chess:
- ♢ Average branching factor: 35
- ♢ Games go to about 50 moves by each player
  $\longrightarrow$ 35100 nodes! 1040 different legal positions

Time limits $\longrightarrow$ unlikely to find goal, must approximate

The complexity of games introduces a new kind of uncertainty:
not due to lack of information but because one does not have time to calculate the exact consequences of any move

On the other hand tic-tac-toe is boring because it is too simple to determine the best move.

Types of games

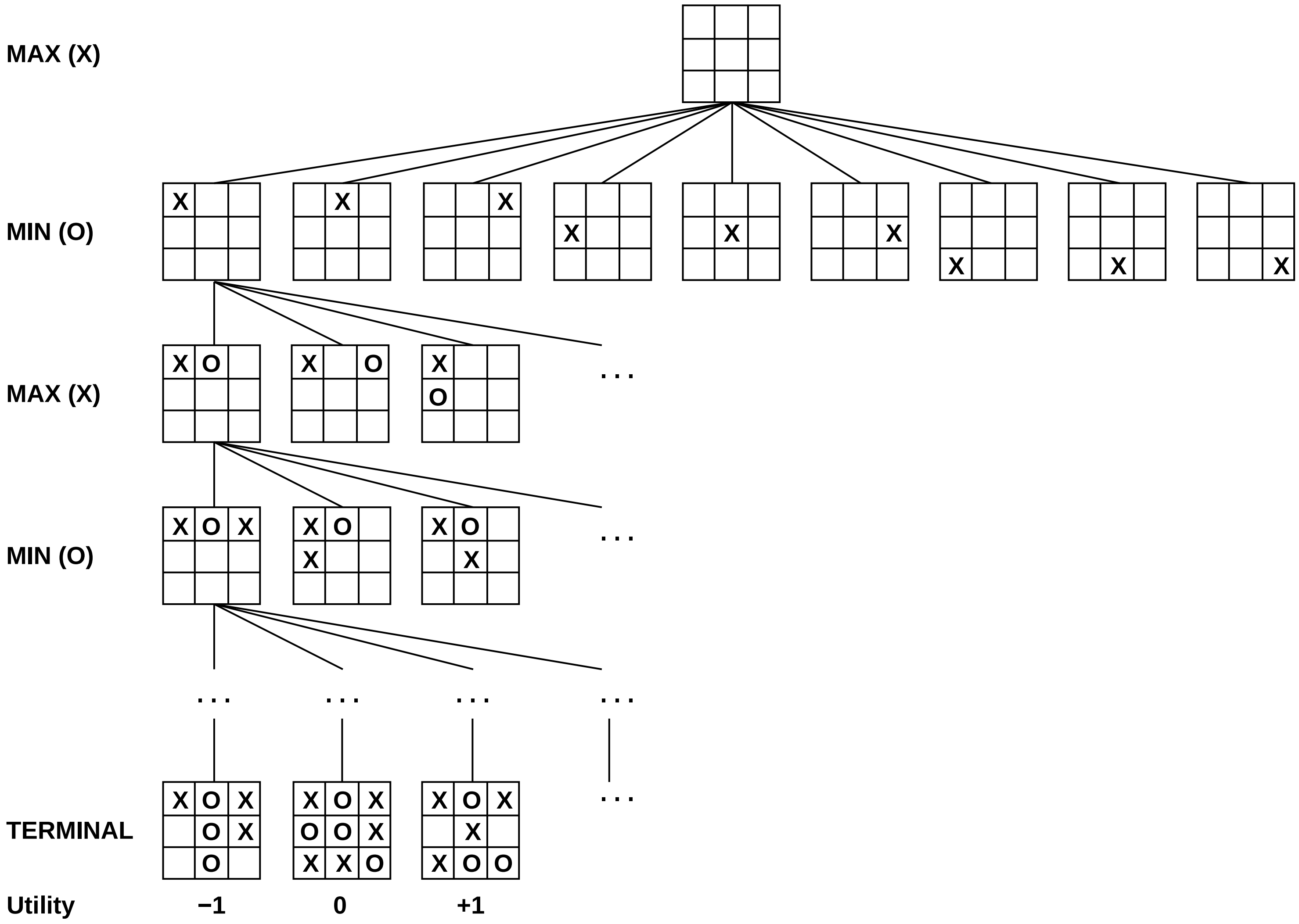
	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information	battleships, blind tictactoe	bridge, poker, scrabble nuclear war

Two-person games

Players: MAX and MIN taking turns until game is over

We can view MAX as the agent: in other words, MAX is constructing the search tree at each move and plays so as to maximize its gains assuming a $perfect$ opponent

Game tree (2-player, deterministic, turns)

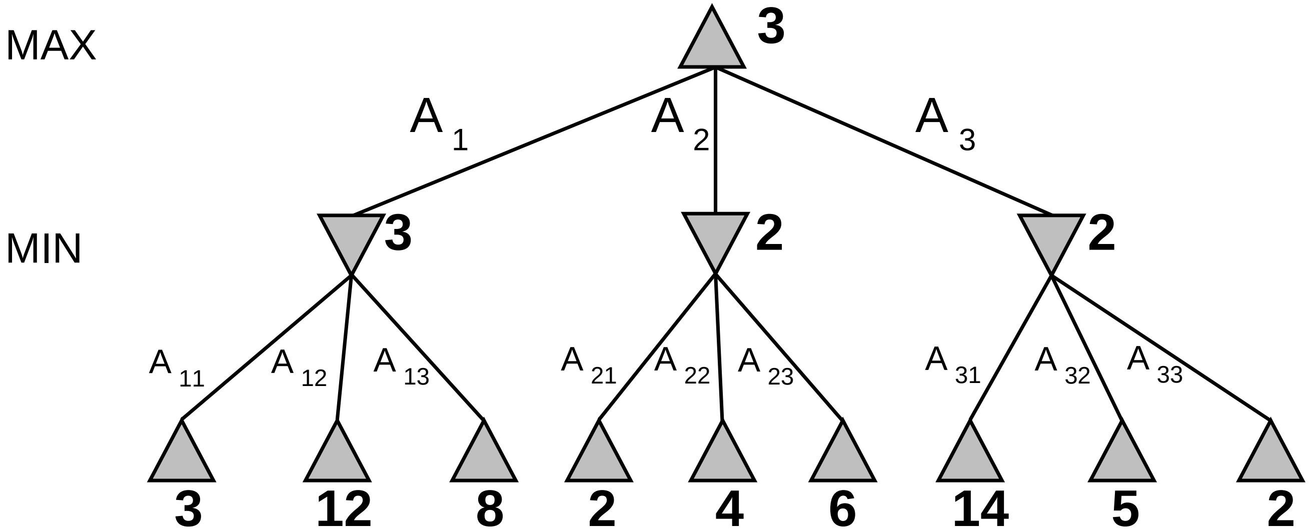


Minimax

Minimax algorithm is designed to determine the optimal strategy for MAX:
Perfect play for *deterministic and perfect-information games*

Idea: choose move to position with 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

Minimax algorithm

♢ Maximizes the utility under the assumption that the opponent will play perfectly to minimize it.

♢ The optimal strategy can be determined by examining the minimax value of each node.

♢ MAX maximizes its worst-case outcome!

♢ Recursive search.

Properties of minimax

Complete??

Optimal??

Time complexity??

Space complexity??

Properties of minimax

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

Optimal?? Yes, against an optimal opponent. Otherwise??

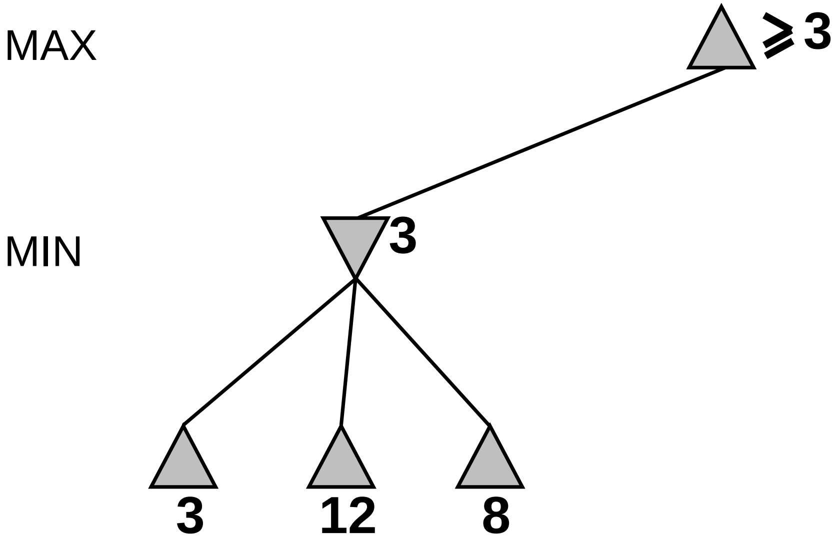
Time complexity?? $O(b^m)$

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

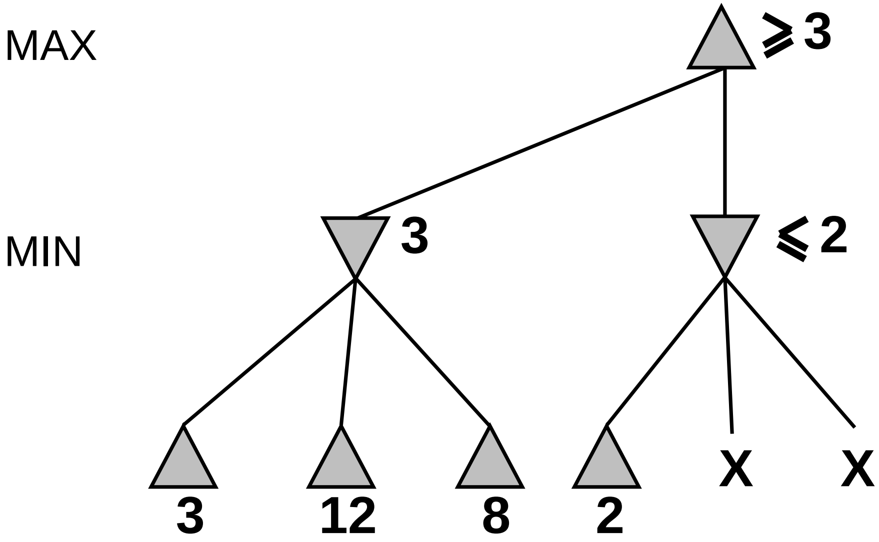
For chess, $b \approx 35$, $m \approx 100$ for "reasonable" games
$\rightarrow$ exact solution completely infeasible

But do we need to explore every path?

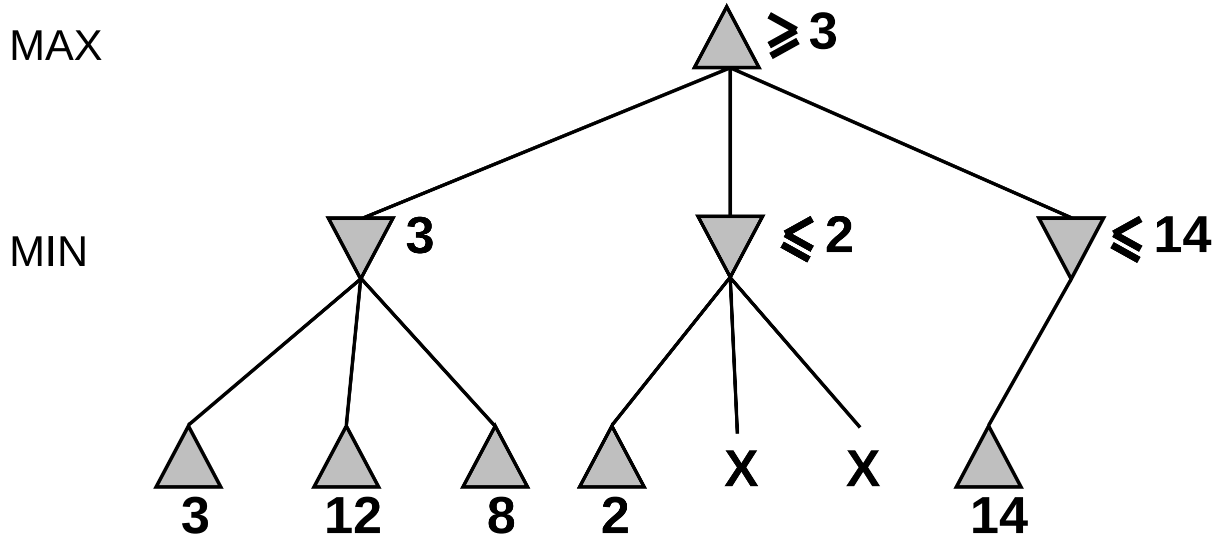
α–β pruning example



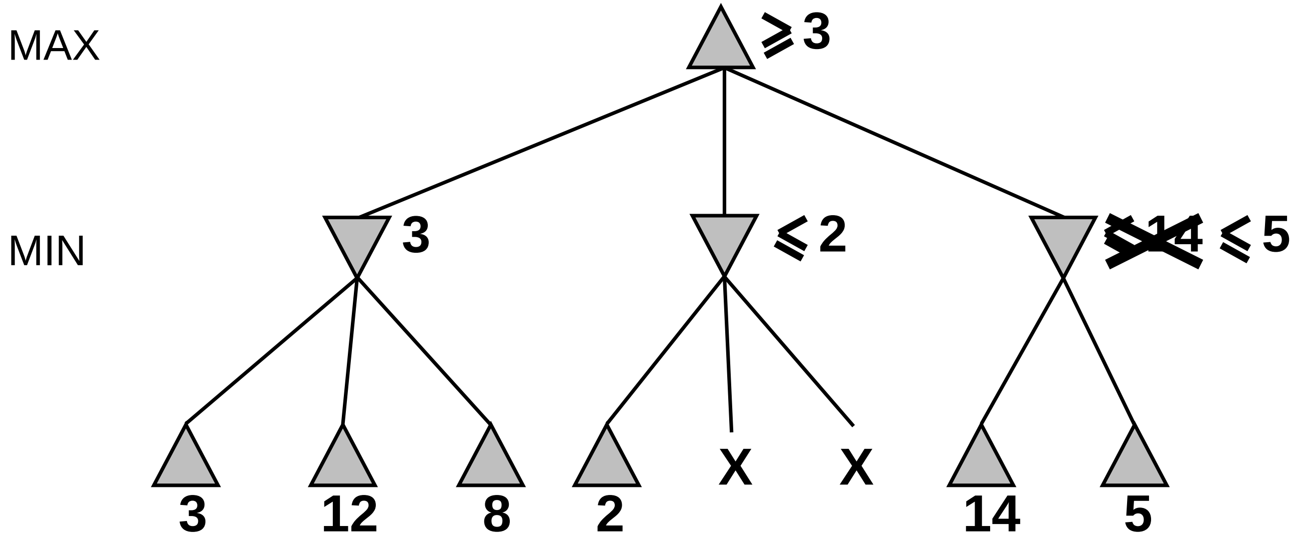
α–β pruning example



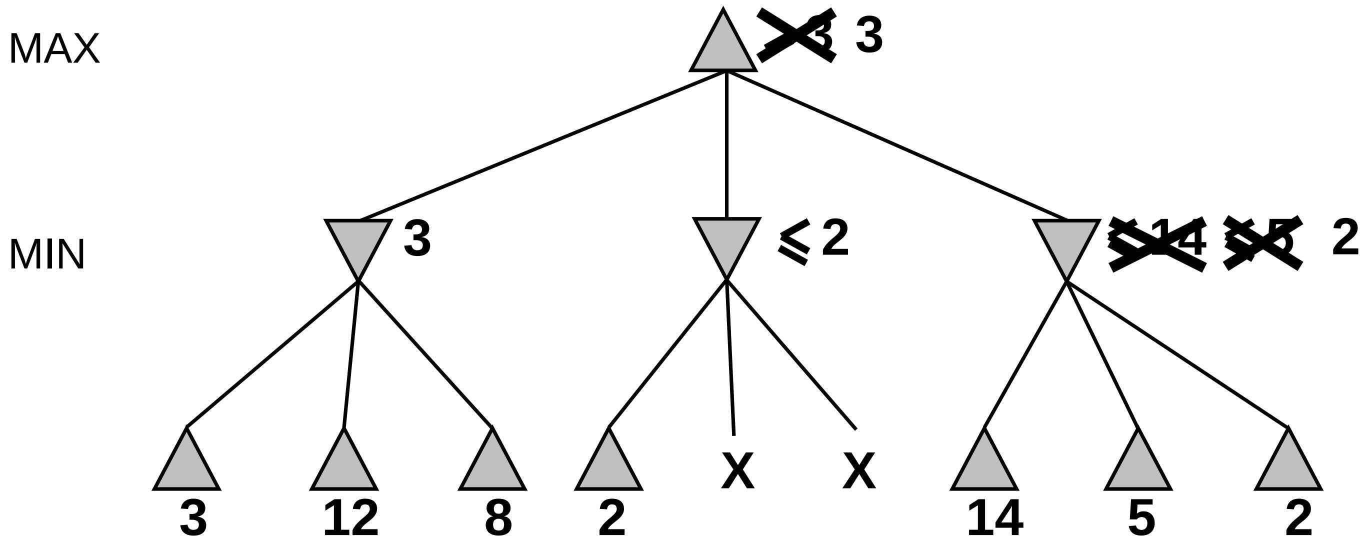
α–β pruning example



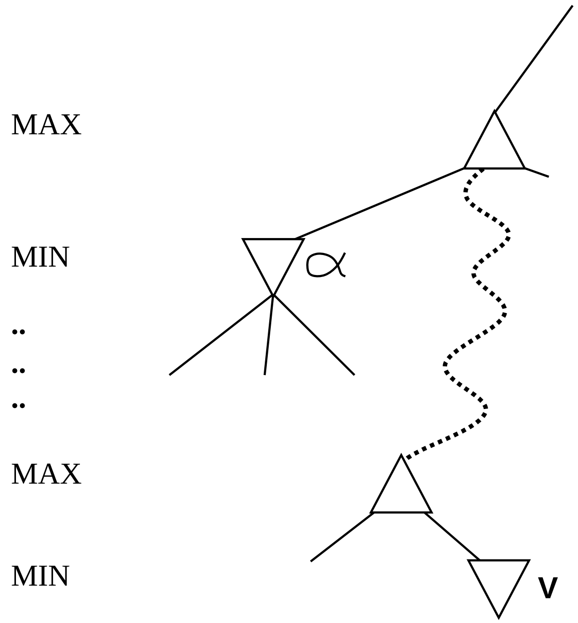
α–β pruning example



α–β pruning example



Why is it called α–β?



α is the best value (i.e., highest-value) we have found so far off the current path for MAX

If V is worse than α, MAX will avoid it $\rightarrow$ prune that branch

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

The α–β algorithm

```
function ALPHA-BETA-SEARCH(state) returns an action
   v ← MIN-VALUE(state, -∞, +∞)
   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 ← −∞
   foreach 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
   same as MAX-VALUE but with roles of α, β reversed
```

Properties of α–β

Pruning **does not** affect final result

Effectivenss is highly dependent on the ordering which the states are examined.

Good move ordering improves effectiveness of pruning

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

Good ordering will be the one that examine first the succesors that are likely to be the best. (cannot be done perfectly. we can try depth limiting approach for finding such successor)

Imperfect decisions

The minimax algorithm assumes that the program has time to search all the way down to terminal states which is exponential in the depth of the game tree.

α–β algorithm allows us to prune but still has to search all the way to the terminal state.

Shannon proposed that instead of going all the way down to terminal states and using the utility function, the program should **cut-off** the search earlier, and apply a heuristic **evaluation function**, turn the nonterminal nodes to terminal nodes using a .

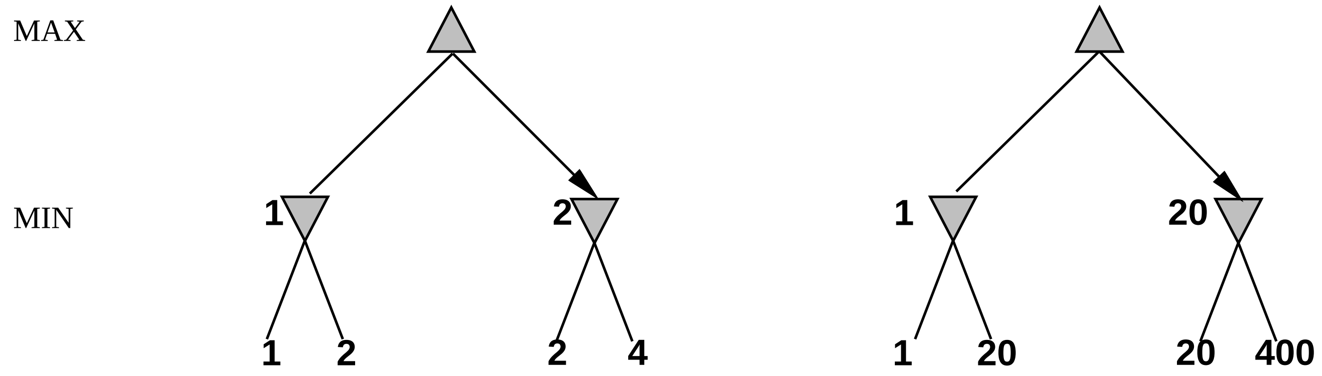
Standard approach:

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

Evaluation functions

◇ EVAL should order the *terminal* states in the same way as the tru utility function.
I.e., Behaviour is preserved under any **monotonic** transformation of EVAL
Only the order matters:

payoff in deterministic games acts as an ordinal utility function

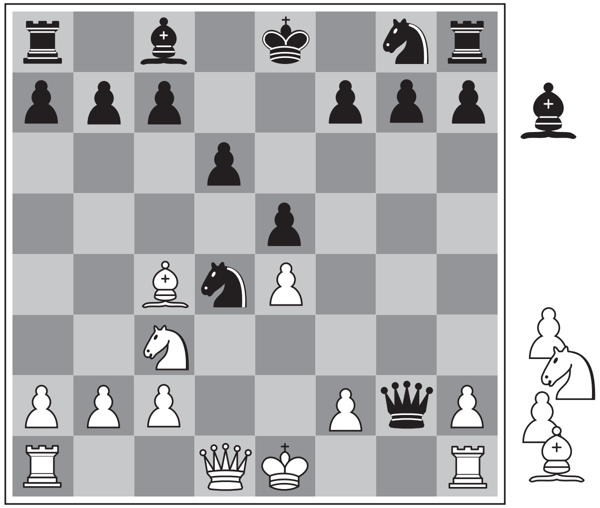


◇ The computation must not take too long

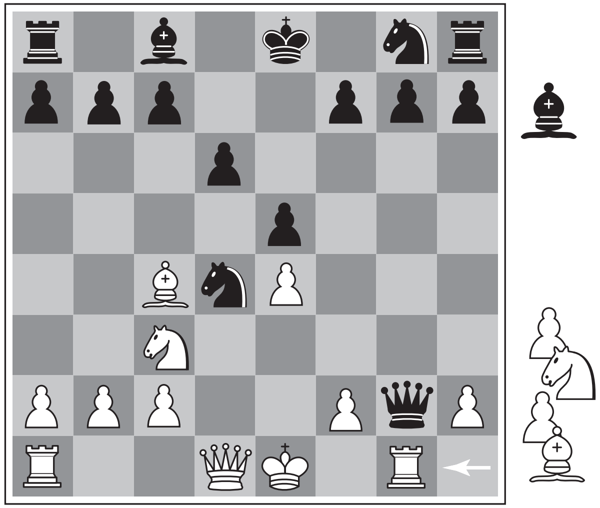
◇ For nonterminal states, the EVAL should be strongly correlated with the actual chances of winning.

Evaluation functions: chess example

Returns *estimate* of the expected utility of the game from a given position.



(a) White to move



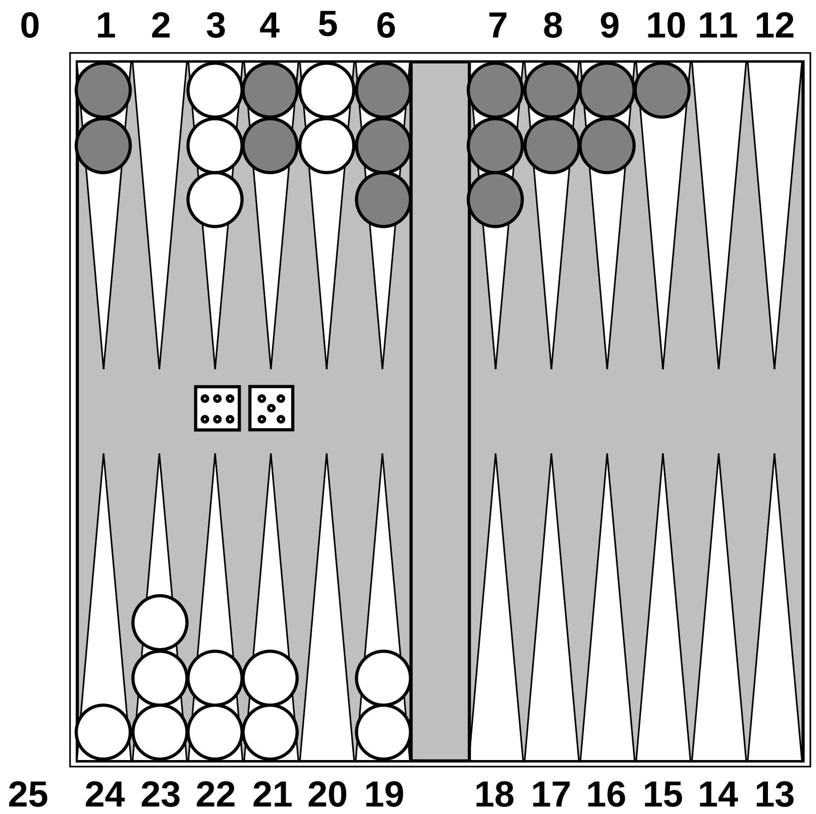
(b) White to move

For chess, weighted linear function can be used

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

where w_i is the weight (e.g., values of the pieces of type i in state s) and $f_i(s)$ is the value of feature i at state s (e.g., number of pieces of type i)

Stochastic (nondeterministic) games: e.g. backgammon

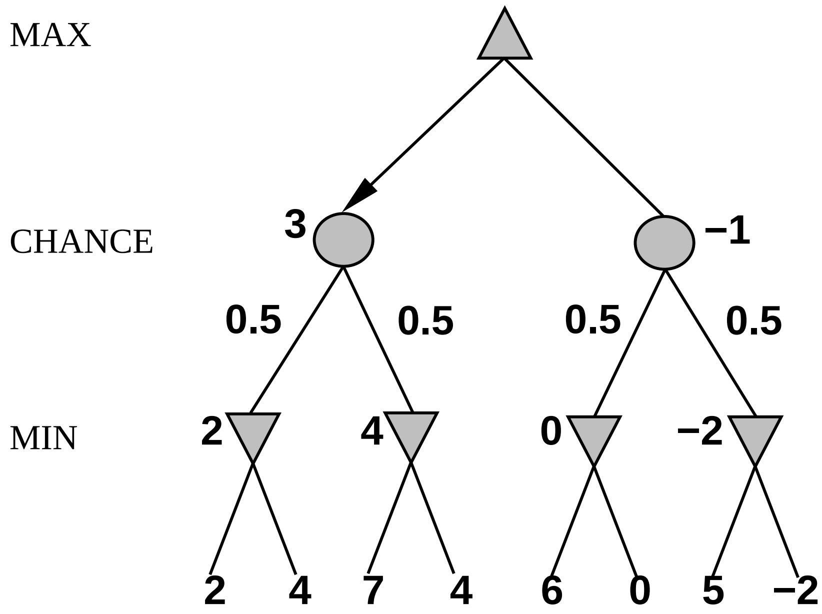


Dice rolling events cause random events to happen.

Stochastic games in general

In stochastic games, chance introduced by dice, card-shuffling

Simplified example with coin-flipping:



CHANCE NODE branches leading from each chance node denote the possible dice rolls; each branch is labeled with the roll and its probability.

Algorithm for nondeterministic games

EXPECTIMINIMAX gives perfect play

Just like MINIMAX, except we must also handle chance nodes:

```
. . .
if state is a MAX node then
        return the highest EXPECTIMINIMAX-VALUE of SUCCESSORS(state)
if state is a MIN node then
        return the lowest EXPECTIMINIMAX-VALUE of SUCCESSORS(state)
if state is a chance node then
        return average of EXPECTIMINIMAX-VALUE of SUCCESSORS(state)
. . .
```

Nondeterministic games in practice

Dice rolls increase b: 21 possible rolls with 2 dice
Backgammon $\approx$ 20 legal moves (can be 6,000 with 1-1 roll)

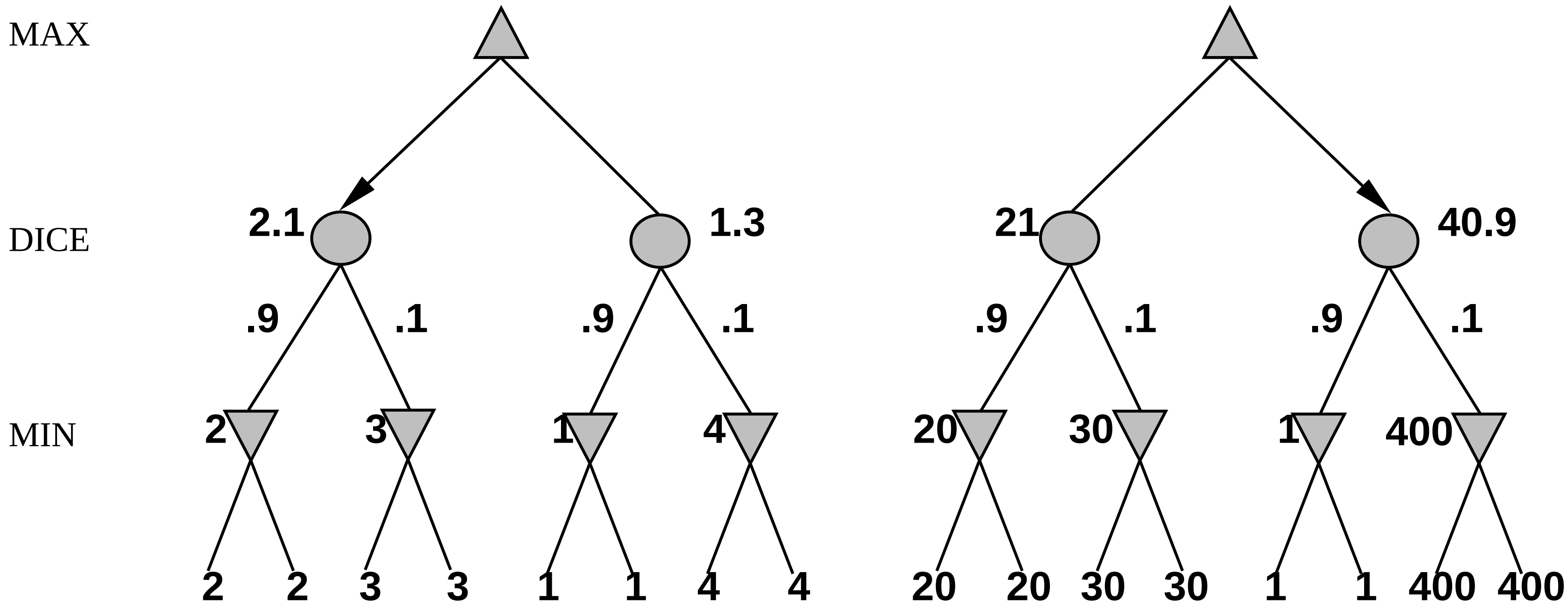
$$\text{depth } 4 = 20 \times (21 \times 20)^3 \approx 1.2 \times 10^9$$

As depth increases, probability of reaching a given node shrinks
$\rightarrow$ value of lookahead is diminished

α–β pruning is much less effective

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

Digression: Exact values DO matter



Behaviour is preserved only by positive linear transformation of EVAL

Hence EVAL should be proportional to the expected payoff

Stochastic partially observable games

E.g., card games, where opponent's initial cards are unknown

Typically we can calculate a probability for each possible deal

Seems just like having all the dice rolled at the beginning*

Idea: compute the minimax value of each action in each deal,
then choose the action with highest expected value over all deals*

Special case: if an action is optimal for all deals, it's optimal.*