Artificial Intelligence

Games

Summary

Russell & Norvig: Chap 5, Sections 1 – 4

♢ Perfect decisions
  – minimax search
  – α–β pruning
  ♢ Monte Carlo Search ♢ Stochastic games

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

Games

chess, checkers, go, backgammon, bridge, poker
abstract, and **static**
chance e.g. dice, cards
imperfect information e.g. cards

♢ **real** soccer, exploration, search: environment partially observable, dynamic, incomplete information

In game theory chess is a two-player, deterministic, turn-taking, zero-sum game of perfect information (0+1,1+0,1/2+1/2)

Games vs. search problems

"Unpredictable" opponent:

solution is a strategy
specifying a move for every possible opponent reply

Time limits make it unlikely to find goal, one must approximate.

An old challenge ...

- 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)

That is more and more central to the field

- ...
- Deep blue (IBM Watson, 1997)
- Alpha GO (Google Deep Mind, 2016)
- Deep stack (Bowling, 2017)
- Pluribus (Brown, 2019)

Types of games

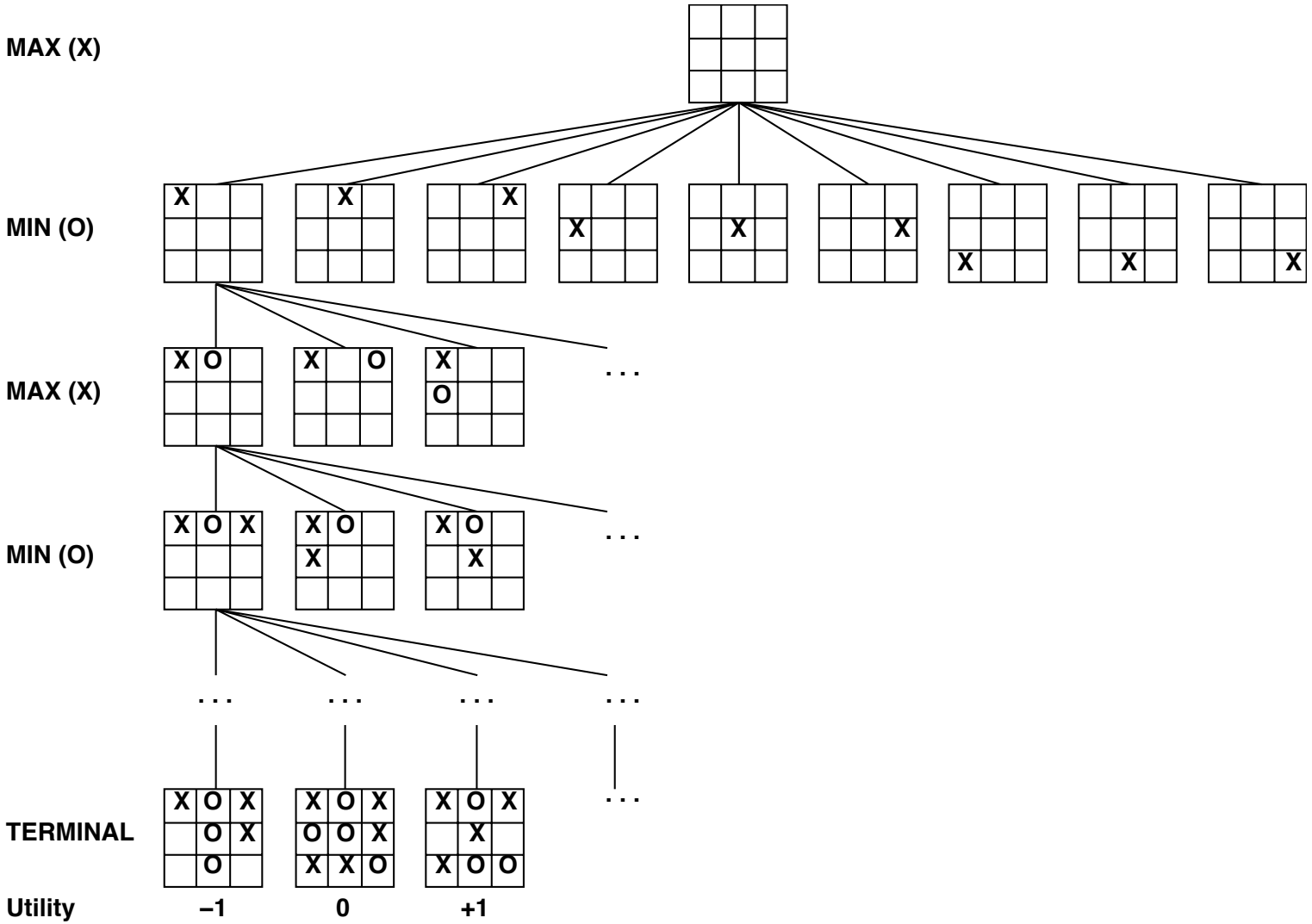
	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information	battleships, blind tictactoe	bridge, poker, scrabble nuclear war

Problem definition

Games as search problems

- The **initial state**, with the positions on the board.
- The next player.
- A set of **actions** defining admissible moves in a given state.
- A **transition model**, defining the state transitions determined by the moves.
- A **termination test**, determining when the game is over (**terminal states**).
- A **utility function**, to compute the numerical result of a game.

Game tree (2-player, deterministic, turns)



Minimax

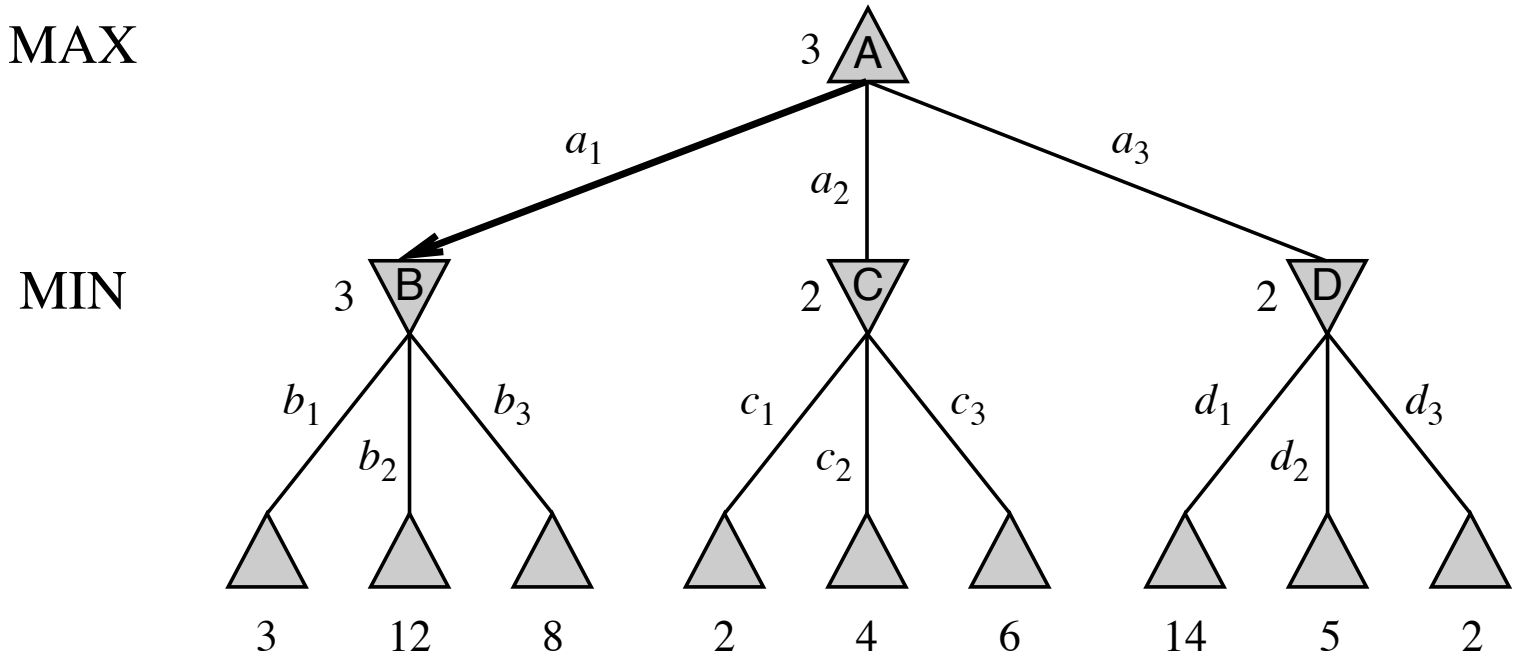
Perfect play for deterministic, perfect-information games

Idea: choose move to position with highest *minimax value*
= best achievable payoff against best play

1. generate the game tree
2. compute utility for each terminal state
3. use utility at level $N+1$ to compute it at level N, Choosing min for the opponent's moves and max for the own moves.

Minimax

E.g., 2-ply game:



The best move for player 1 in A is a_1,
The best move for player 2 in B is b_1

```
function MINIMAX-SEARCH(game, state) returns an action
    player ← game.TO-MOVE(state)
    value, move ← MAX-VALUE(game, state)
    return move

function MAX-VALUE(game, state) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v, move ← −∞
    for each a in game.ACTIONS(state) do
        v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
        if v2 > v then
            v, move ← v2, a
    return v, move

function MIN-VALUE(game, state) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v, move ← +∞
    for each a in game.ACTIONS(state) do
        v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
        if v2 < v then
            v, move ← v2, a
    return v, move
```

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, $b \approx 35$, $m \approx 80$ for "reasonable" games

$\Rightarrow$ exact solution completely infeasible

Considering duplicate states the state space is 35^{80}

Extension to multiple players

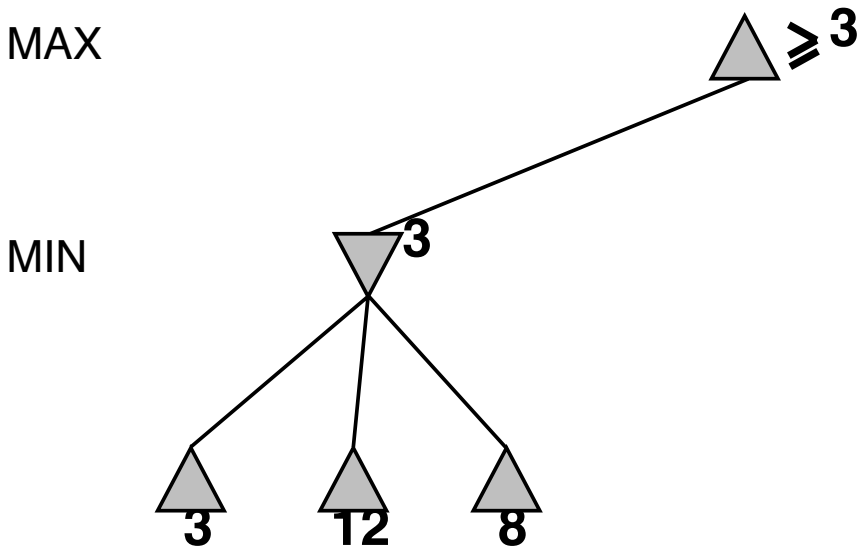
The notion of minimax game trees extends to multiple players.

The value of the game becomes a vector with a value for each player.

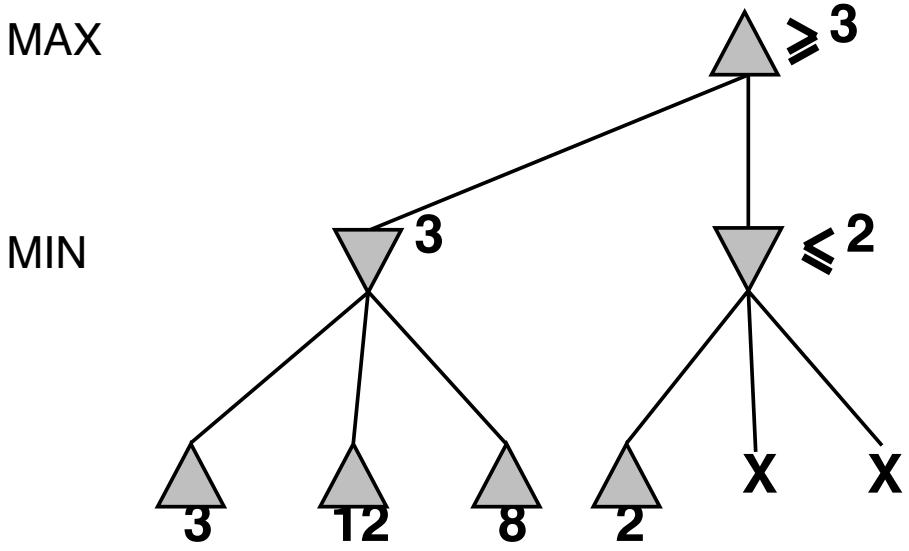
At each step each player maximizes its own utility, but more complex strategies are needed:

- ♢ alliances
- ♢ cooperation (non zero-sum games)

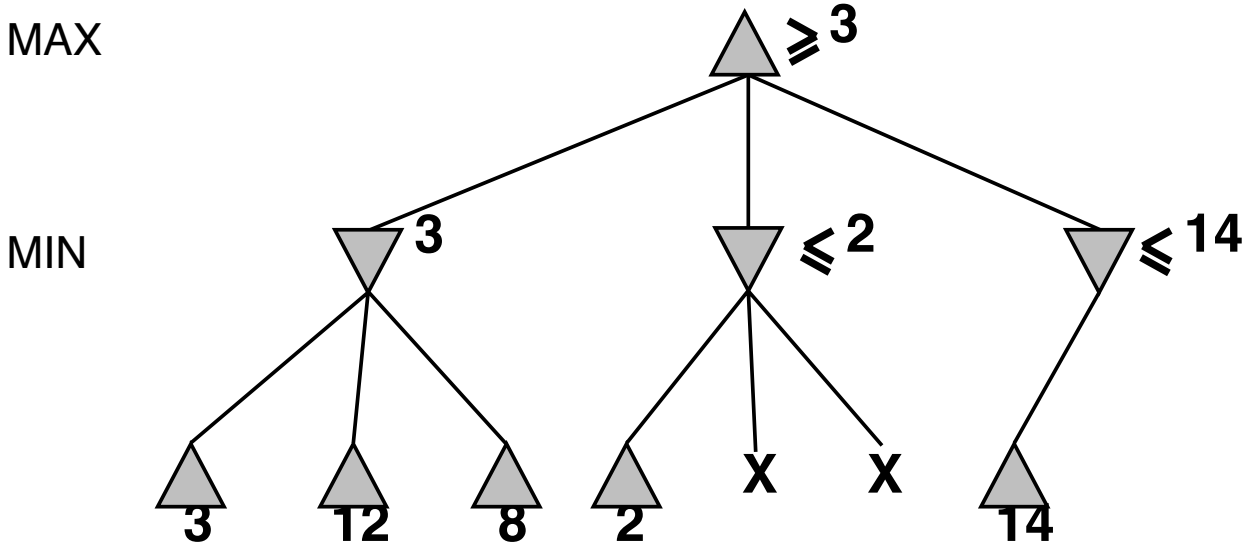
α–β pruning example



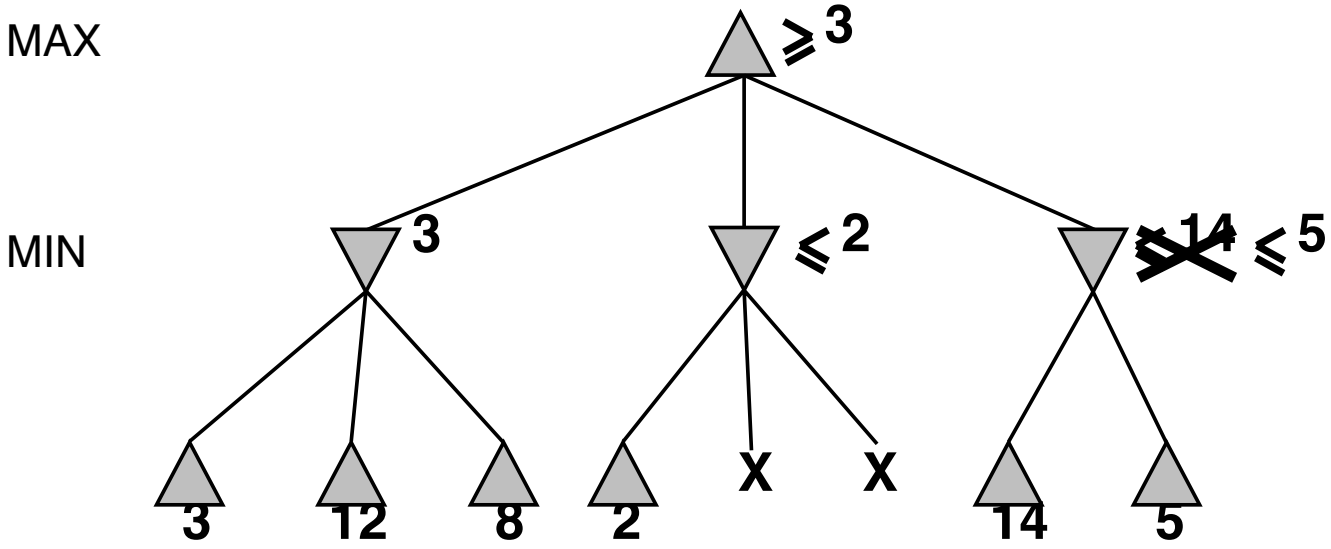
α–β pruning example



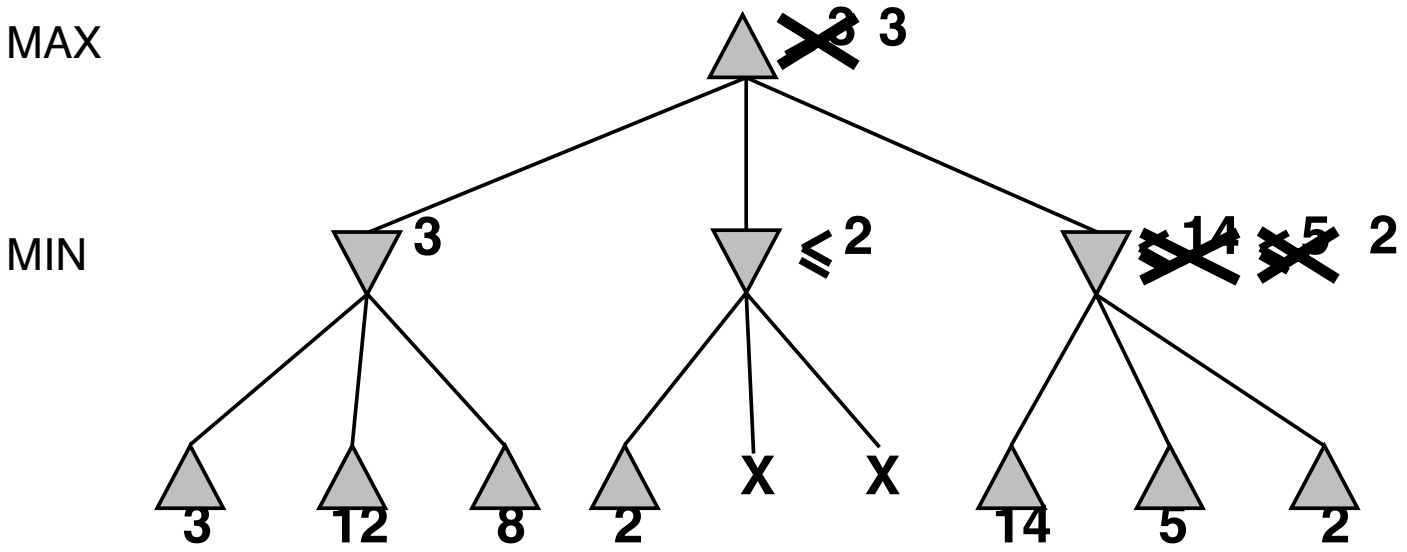
α–β pruning example



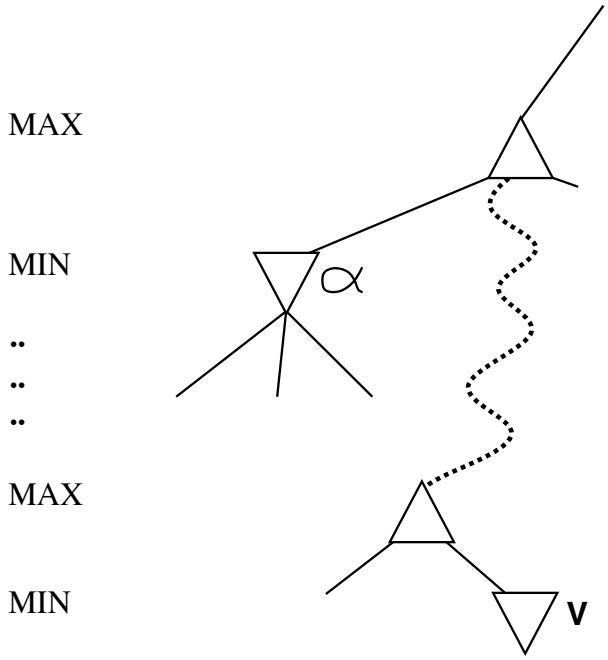
α–β pruning example



α–β pruning example



Why is it called α–β?



α is the best value (to MAX) found so far off the current path
If V is worse than α, MAX will avoid it $\Rightarrow$ prune that branch

Define β similarly for MIN

```
function ALPHA-BETA-SEARCH(game, state) returns an action
    player ← game.TO-MOVE(state)
    value, move ← MAX-VALUE(game, state, −∞, +∞)
    return move
```

```
function MAX-VALUE(game, state, α, β) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v ← −∞
    for each a in game.ACTIONS(state) do
        v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), α, β)
        if v2 > v then
            v, move ← v2, a
            α ← MAX(α, v)
        if v ≥ β then return v, move
    return v, move
```

```
function MIN-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), α, β)
    if v2 < v then
      v, move ← v2, a
      β ← MIN(β, v)
    if v ≤ α then return v, move
  return v, move
```

α: "at least

β: "at most

Properties of α–β

Pruning *does not* affect final result

Good move ordering improves effectiveness of pruning

With "perfect ordering," time complexity $= O(b^{m/2})$

$\Rightarrow$ *doubles* depth of search

$\Rightarrow$ can easily reach depth 8 and play good chess

Average ordering $O(b^{3m/4})$

Properties of α–β

Iterative deepening helps finding the correct ordering for the moves:

killer moves are the best moves

transposition table: caches the heuristic values of states

Is perfect ordering achievable?

Metareasoning: reasoning about which computations are relevant

Varying strategy

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

$\Rightarrow 10^6$ nodes per move

♢ **Type A strategy**: expand the whole tree up to a certain depth and then use an evaluation function (as breadth-first)

♢ **Type B strategy**: ignore moves that look bad and focus on promising moves (as depth-first)

Type A strategy

Standard approach:

- *cutoff test*

  e.g., depth limit

- *evaluation function*

  = estimated desirability of position (statistical/heuristic)

Cutting off search

MinimaxCutoff is identical to MinimaxValue except

1. Terminal? is replaced by Cutoff?
2. Utility is replaced by Eval

Does it work in practice?

$$b^m = 10^6, \quad b = 35 \quad \Rightarrow \quad m = 4$$

4-ply lookahead is a hopeless chess player!

4-ply $\approx$ human novice
8-ply $\approx$ typical PC, human master
12-14-ply $\approx$ Deep Blue, Kasparov

Evaluation functions

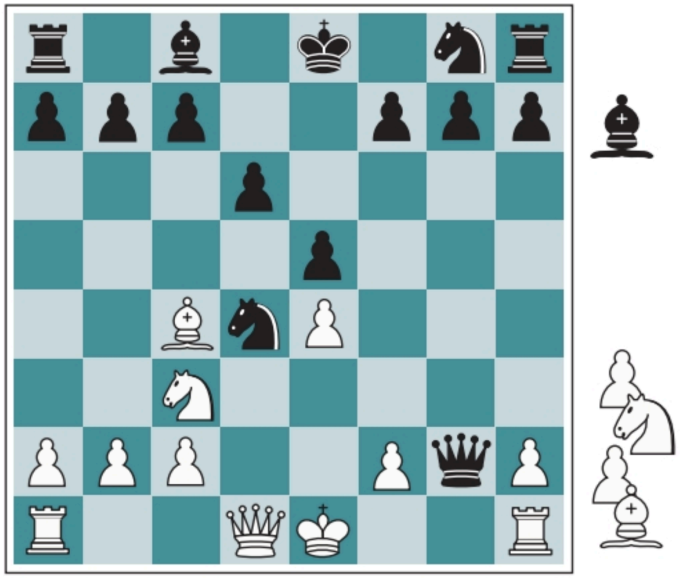
- order terminal states as true utility
- fast computation
- good correlation with actual chances of winning

Behaviour is preserved under any *monotonic* transformation of EVAL

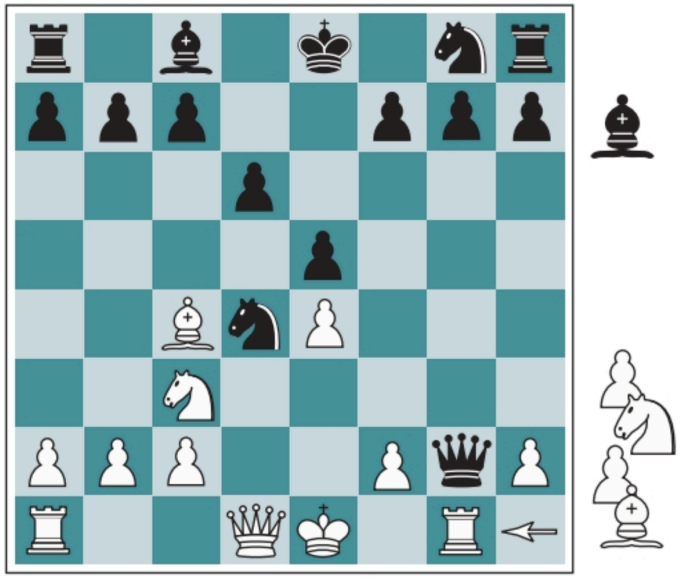
Exact values don't matter, only the order matters:

payoff in deterministic games acts as an *ordinal utility* function

Heuristic evaluation functions



(a) White to move



(b) White to move

For chess, typically *linear* weighted sum of features

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

Cut-off search

Replace $TERMINAL-TEST$ with:

if $CUTOFF-TEST(state, depth)$ **then return** $EVAL(state)$

Iterative deepening can be used when time is limited.

Quiescience search: to check whether the move that is cut off does not lead to a dramatic change in the evaluation function.

Horizon Effect: to delay a bad move that is inevitable by suffering other "minor losses.

Further improvements

♢ **Forward pruning**: evaluating which moves are unlikely to win (estimate which moves are likely to be outside the $\alpha - \beta$ range, using ML to find the estimates)

♢ **Lookup tables**: For openings and the terminal part of the games with few remaining pieces (games with more than 6 pieces have been fully analyzed – in checkers the whole game)

Monte Carlo Tree Search

Underlying principle

- Does simulations (rollout) of complete games to estimate the heuristic values of moves.

- Keeps playing against itself and compute the win percentage.

Exploring the search space

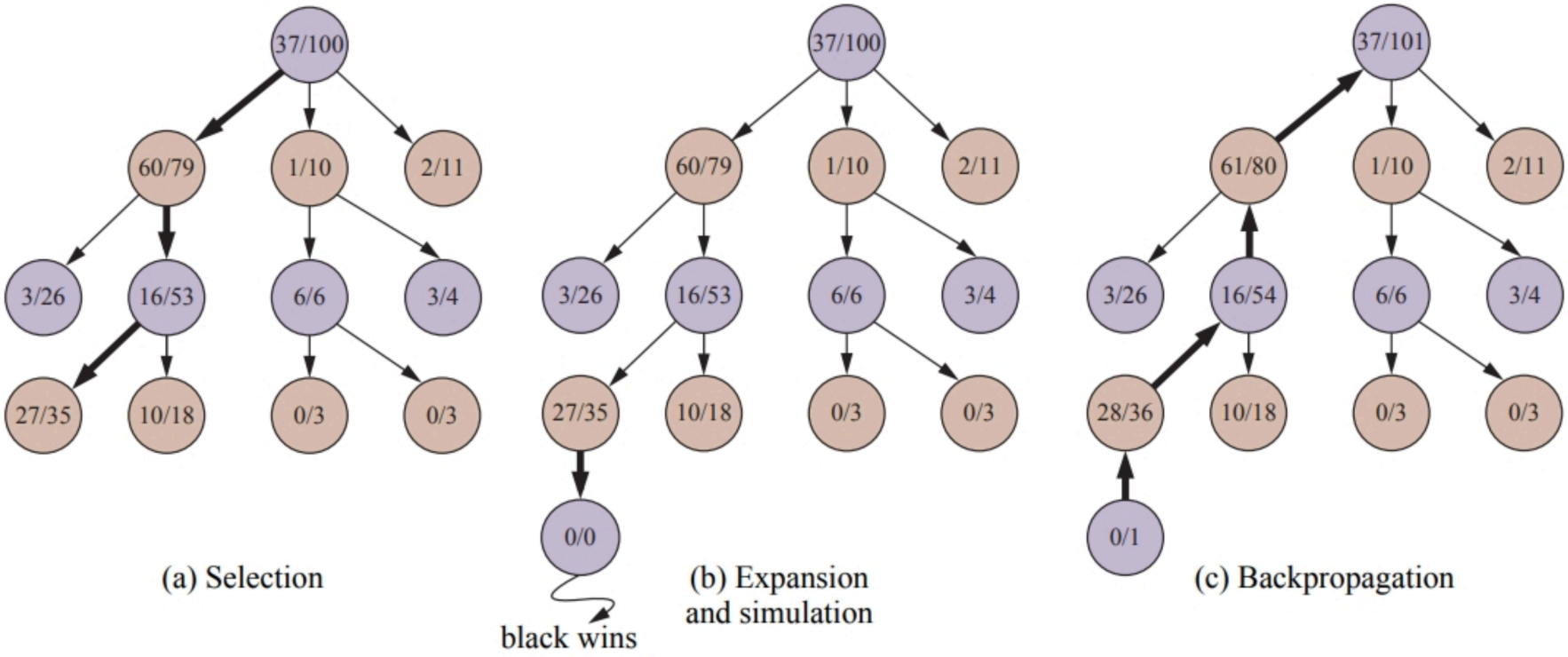
♢ **Playout policy**: random exploration is inefficient, a good choice of the moves could be learned (go) or based on game heuristics.

♢ **Selection Policy**: focus the computation **Exploration** vs **Exploration**

MCTS sketch

1. **Selection**: choose the next action
2. **Expansion**: expands a node
3. **Simulation**: completes the simulation recording only the final outcome
4. **Back-propagation**: update the search tree nodes

MCTS example



MCTS propeties

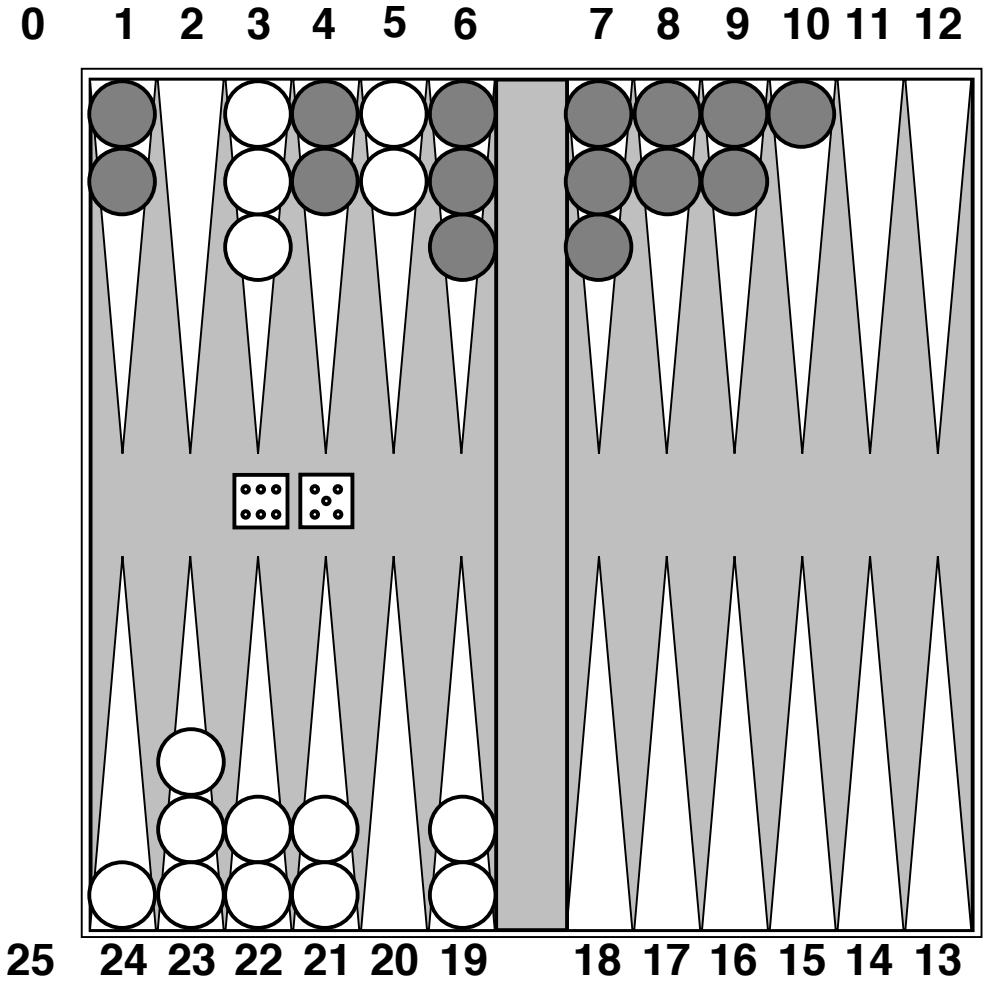
♢ MTCS is chosen when branching factor becomes very high and the evaluation function is difficult

♢ MTCS is more robust than Type A search (but evaluation can be used also in MCTS)

♢ MTCS can be applied to unknown games

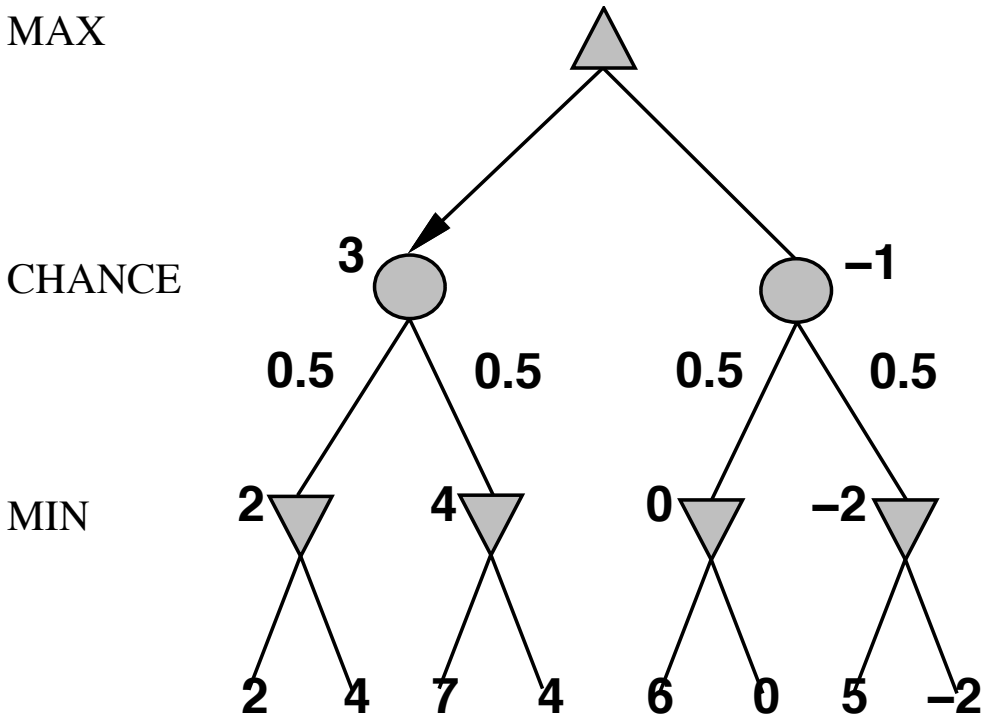
♢ MTCS is a form of **reinforcement learning**

Nondeterministic games: backgammon



Nondeterministic games in general

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



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)
...
```

Properties of Expectiminimax

Completeness and **Optimality** can not be achieved because of the random steps, where the expected value is computed

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

Space complexity $O(bmnm)$ (depth-first exploration)

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

$\Rightarrow m = 3?$

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.

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, which are too good.

Go

Chess: b=35, d=80
Go: b=250, d=150

Before **AlphaGo** best programs used Montecarlo Tree Search, but reach only amateur level.

AlphaGo (Google DeepMind, 2016) wins against world champion after combining MC search and learning with Neural Networks

Position evaluation, which reduces the depth
Policy to select the best move, which limits breadth

Poker

Card games include an additional issue **Partial osservability**.

DeepStack (M. Bowling et al. 2017) wins against best poker players 2-players.
Pluribus (M. Brown and T. Sandholm 2019) wins against best poker players 6-players.

Learning:

- Policies for forward pruning (evaluation function)
- Counterfactual regret minimization (self-play)

Concluding

Games are fun to work on! (and dangerous)

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