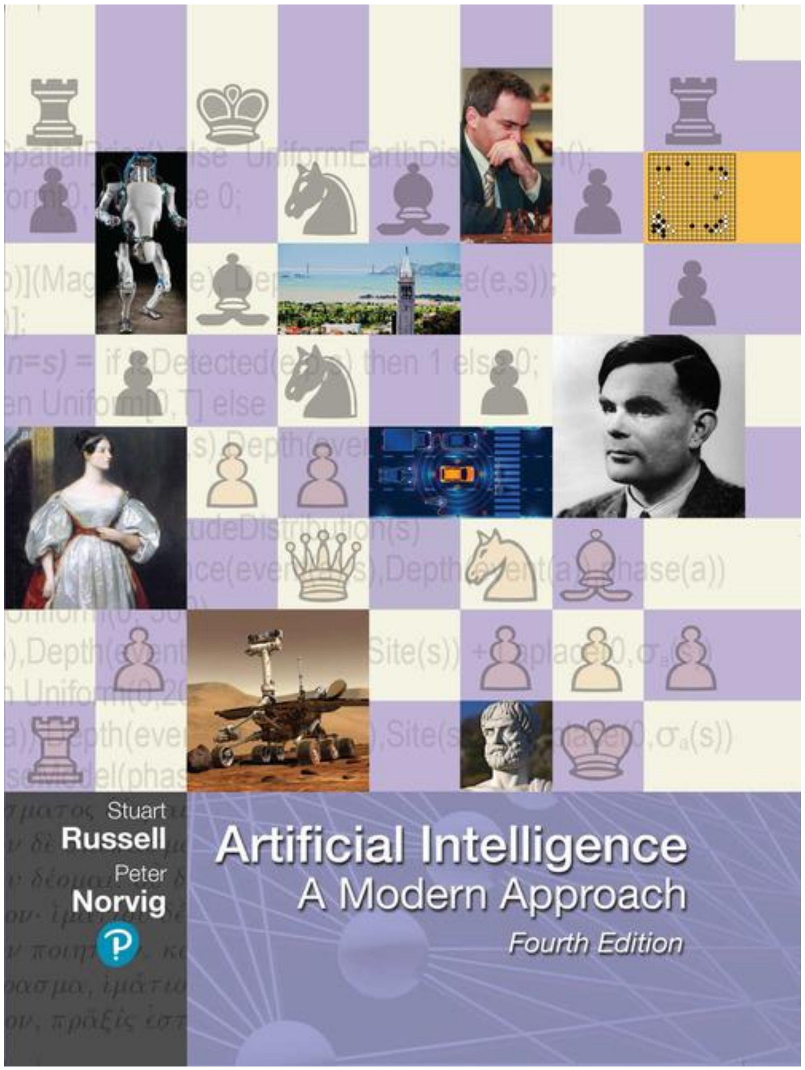
Artificial Intelligence Fundamentals

2024 - 2025



"There is a silver lining on the bias issue. For example, say you have an algorithm trying to predict who should get a promotion. And say there was a supermarket chain that, statistically speaking, didn't promote women as often as men. It might be easier to fix an algorithm than fix the minds of 10,000 store managers."

- Richard Socher

AIMA Chapter 6

Adversarial Search And Games

Outline

- Game Theory
- Optimal Decisions in Games
  - minimax decisions
  - α–β pruning
  - Monte Carlo Tree Search (MCTS)
- Resource limits and approximate evaluation
- Games of chance
- Games of imperfect information
- Limitations of Game Search Algorithms

Games Theory

In this lecture we cover **competitive environments**, in which **two or more agents** have conflicting goals, giving rise to *adversarial search problems*.

For simplicity we consider:

Two players

- Max-min
- Taking turns, fully observable

Moves: Action

Position: state

Zero sum:

- good for one player, bad for another
- No *win-win* outcome.

Components

S_0: The initial state of the game

TO-MOVE(s): player to move in state s.

ACTIONS(s): The set of legal moves in state s.

RESULT(s, a): The transition model, resulting state

IS-TERMINAL(s): A terminal test to detect when the game is over

UTILITY(s; p): A utility function (objective/payoff)

Pearson

Games vs. search problems

"**Unpredictable**" opponent ⇒ solution is a strategy specifying a move for every possible opponent reply

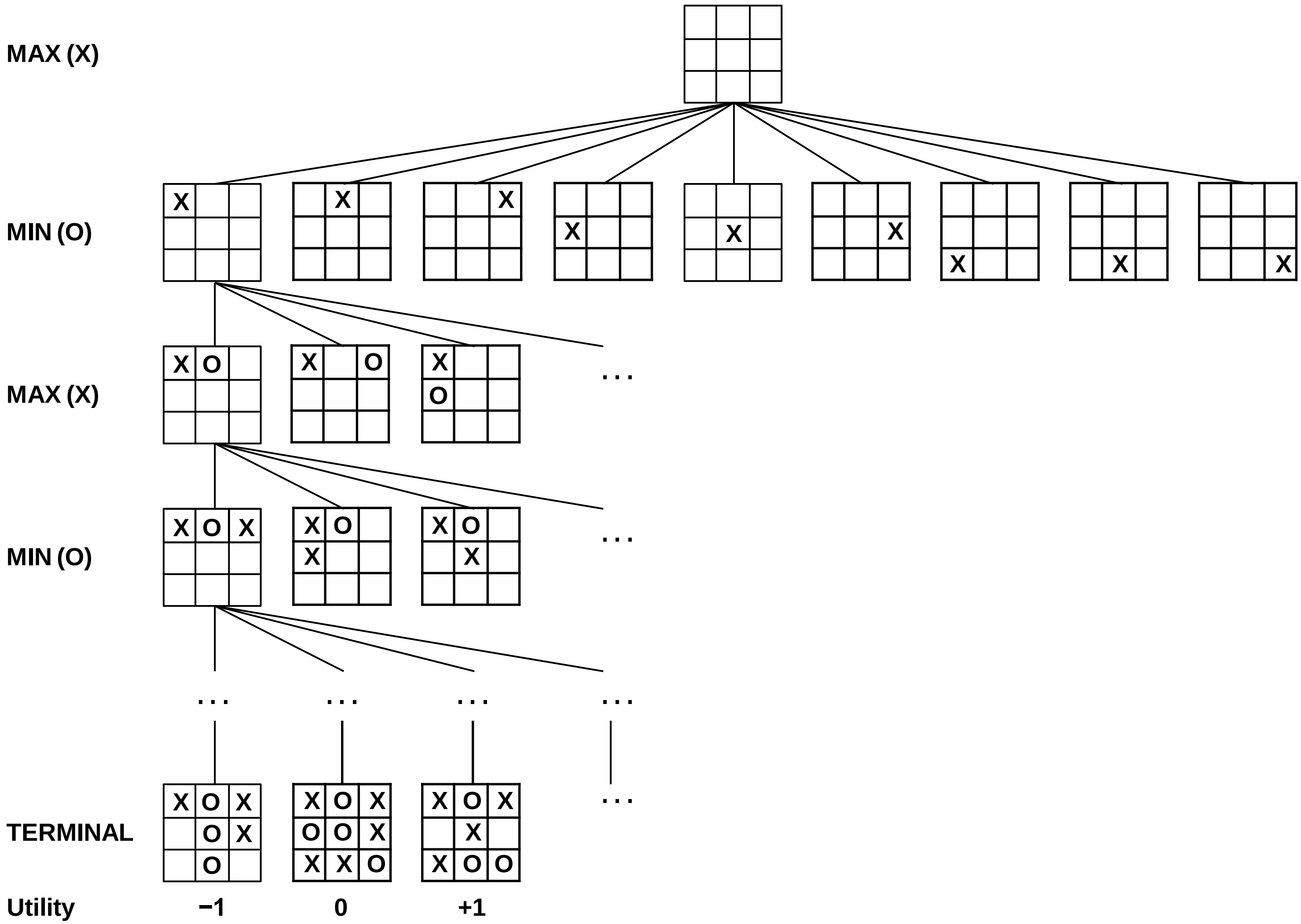
Time limits ⇒ unlikely to find goal, must design **approximate** *Plan of attack*:

- 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 tictactoe	bridge, poker, scrabble nuclear war

Game tree (2-player, deterministic, turns)

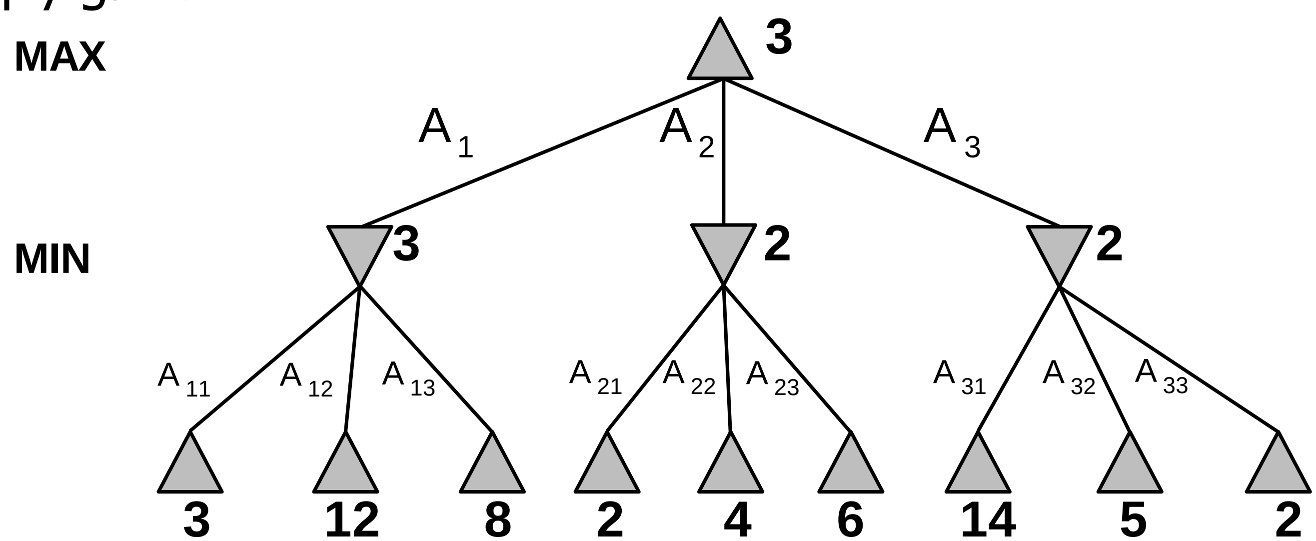


Minimax

Perfect play for deterministic, perfect-information games

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

E.g., 2-ply game:



In some games, the word "move" means that both players have taken an action; therefore the word "**ply**" is used to unambiguously mean one move by one player, bringing us one level deeper in the game tree

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 ← −∞
    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

Complete?

Properties of minimax

Complete? Only if tree is finite (chess has specific rules for this).
NB a finite strategy can exist even in an infinite tree!

Optimal?

Properties of minimax

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

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

Time 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?

Properties of minimax

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

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

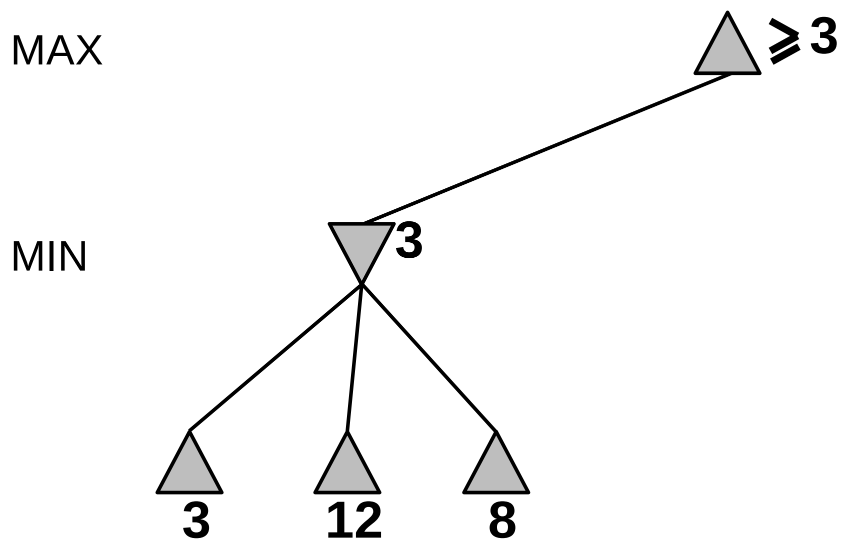
Time complexity? $O(b^m)$

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

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

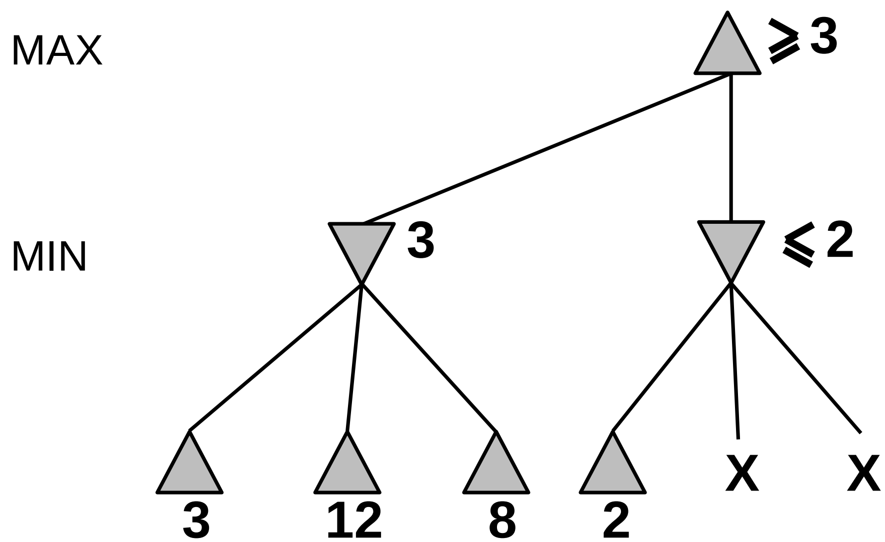
But *do we need to explore every path?*

α–β pruning example

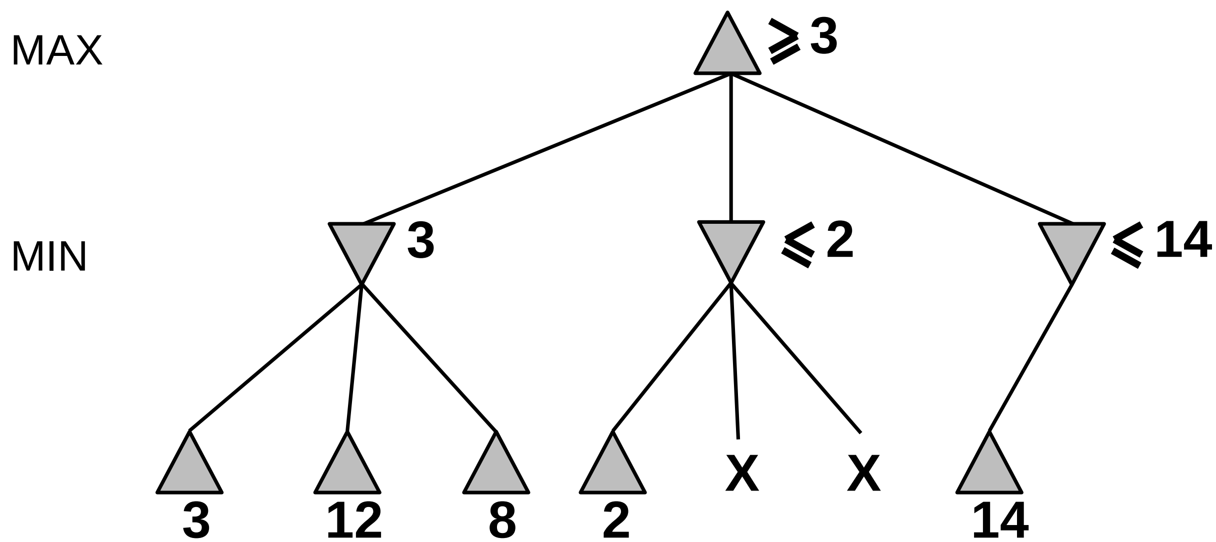


Pearson

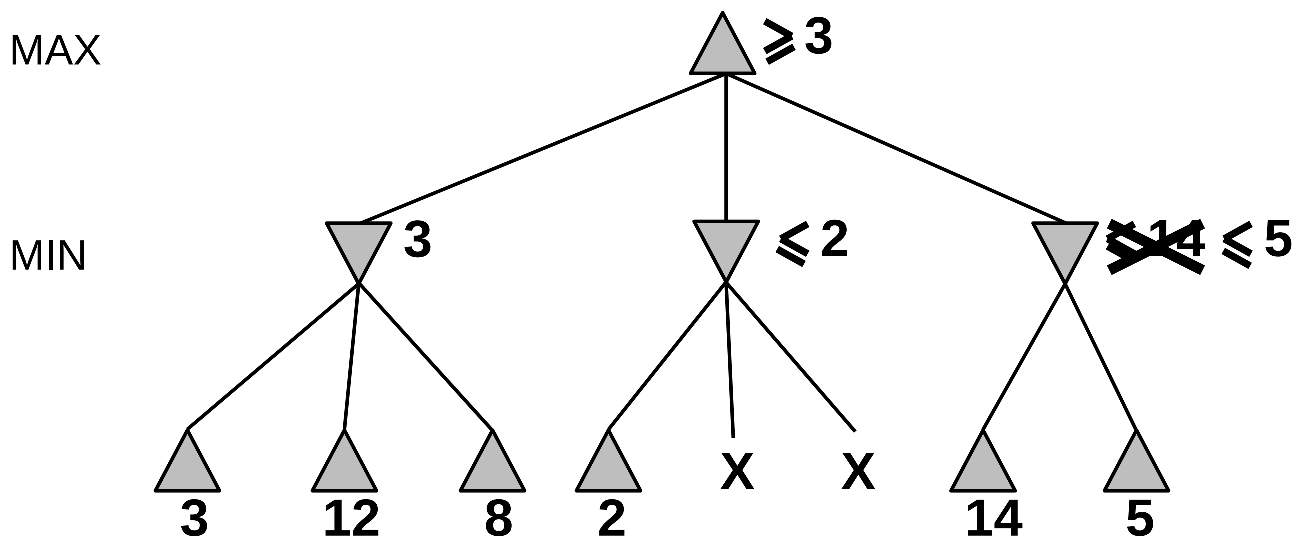
α–β pruning example



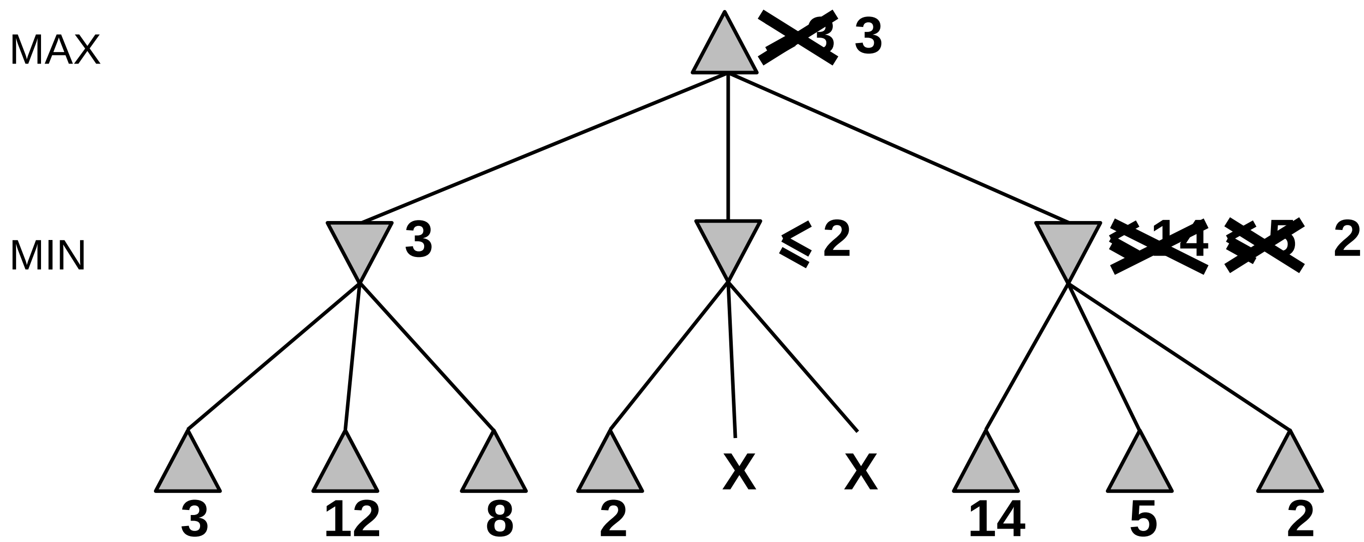
α–β pruning example



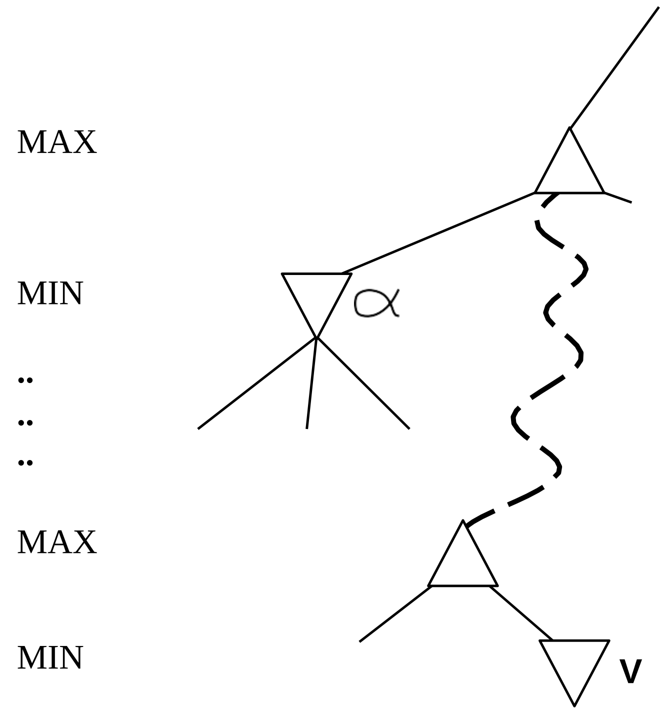
α–β pruning example



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 $\min$

The α–β algorithm

```
function Alpha-Beta-Decision(state) returns an action
   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 ← −∞
   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
   same as Max-Value but with roles of α, β reversed
```

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 solvable depth

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

Unfortunately, 35^{50} is still impossible!

Monte Carlo Tree Search

The basic Monte Carlo Tree Search (MCTS) strategy does not use a heuristic evaluation function. Value of a state is estimated as the *average utility over number of simulations*

- **Playout**: simulation that chooses moves until terminal position reached.

Main iterative steps:

- **Selection**: Start of root, choose move (selection policy) repeated moving down the tree
- **Expansion**: Search tree grows by generating a new child of selected node
- **Simulation**: playout from generated child node
- **Back-propagation**: use the result of the simulation to update all the search tree nodes going up to the root

Monte Carlo Tree Search

UCT: Effective selection policy is called "***U****pper* ***C****onfidence bounds applied to* ***T****rees*"

UCT ranks each possible move based on an upper confidence bound formula UCT called UCB1

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(\textsc{Parent}(n))}{N(n)}}$$

where **U(n)** is the total utility of all playouts that went through node **n**, **N(n)** is the number of playouts through node **n**, and **PARENT(n)** is the parent node of **n** in the tree.

Monte Carlo Tree Search

UCT: Effective selection policy is called “upper confidence bounds applied to trees”

UCT ranks each possible move based on an upper confidence bound formula UCT called UCB1

$$UCB1(n) = \boxed{\frac{U(n)}{N(n)}} + C \times \sqrt{\frac{\log N(\textsc{Parent}(n))}{N(n)}}$$

where U(n) is the total utility of all playouts that went through node n, N(n) is the number of playouts through node n, and PARENT(n) is the parent node of n in the tree.

Thus U(n) / N(n) is the **exploitation term**: the average utility of n.

Monte Carlo Tree Search

UCT: Effective selection policy is called “upper confidence bounds applied to trees”

UCT ranks each possible move based on an upper confidence bound formula UCT called UCB1

$$UCB1(n) = \frac{U(n)}{N(n)} \boxed{+ C \times \sqrt{\frac{\log N(\textsc{Parent}(n))}{N(n)}}}$$

where U(n) is the total utility of all playouts that went through node n, N(n) is the number of playouts through node n, and PARENT(n) is the parent node of n in the tree.

The term with the square root is the **exploration term**: it has the count N(n) in the denominator, which means the term will be high for nodes that have only been explored a few times. In the numerator it has the log of the number of times we have explored *the parent of n*. This means that if we are selecting n some nonzero percentage of the time, the exploration term goes to zero as the counts increase, and eventually the playouts are given to the node with highest average utility.

Monte Carlo Tree Search

function MONTE-CARLO-TREE-SEARCH(state) **returns** an action
    tree ← NODE(state)
    while IS-TIME-REMAINING() **do**
        leaf ← SELECT(tree)
        child ← EXPAND(leaf)
        result ← SIMULATE(child)
        BACK-PROPAGATE(result, child)
    return the move in ACTIONS(state) whose node has highest number of playouts

Monte Carlo Tree Search

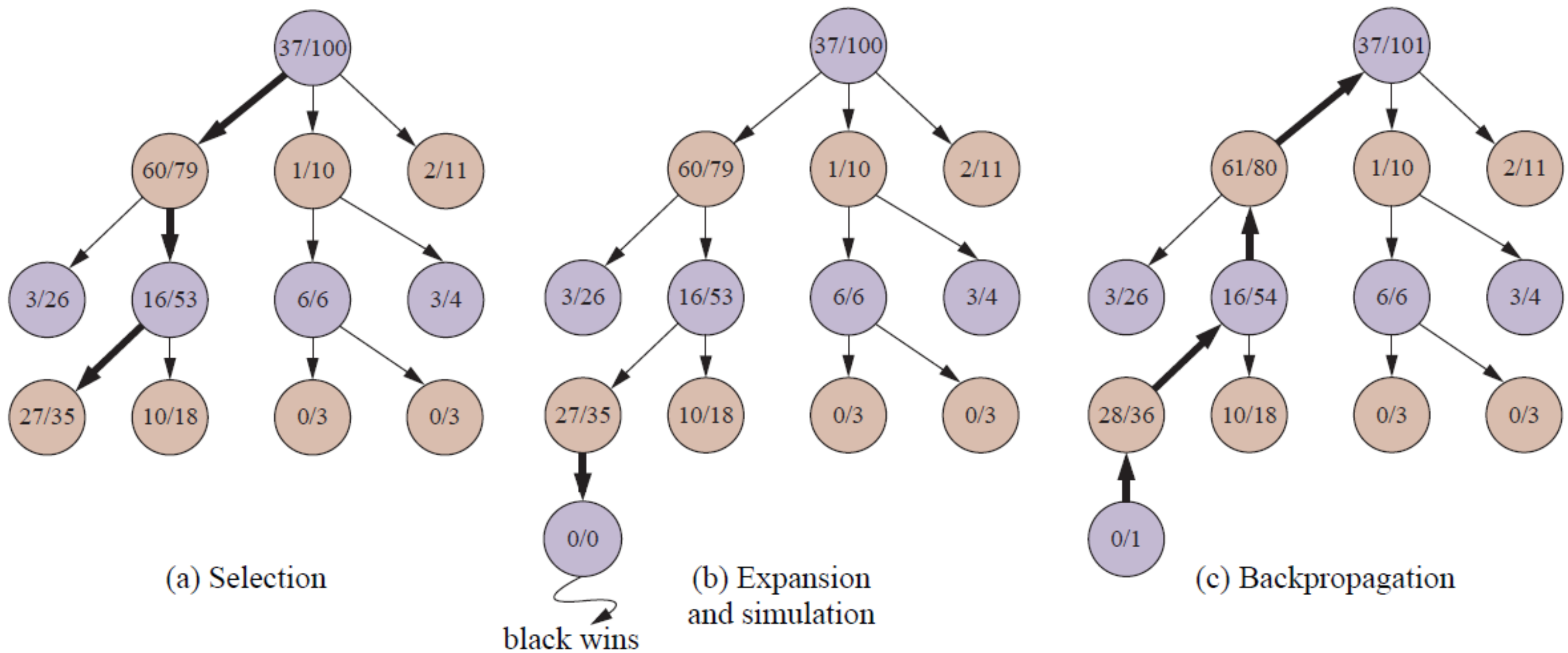


Figure 6.10 One iteration of the process of choosing a move with Monte Carlo tree search (MCTS) using the upper confidence bounds applied to trees (UCT) selection metric, shown after 100 iterations have already been done. In (a) we select moves, all the way down the tree, ending at the leaf node marked 27/35 (for 27 wins for black out of 35 playouts). In (b) we expand the selected node and do a simulation (playout), which ends in a win for black. In (c), the results of the simulation are back-propagated up the tree.

Pearson

Resource limits

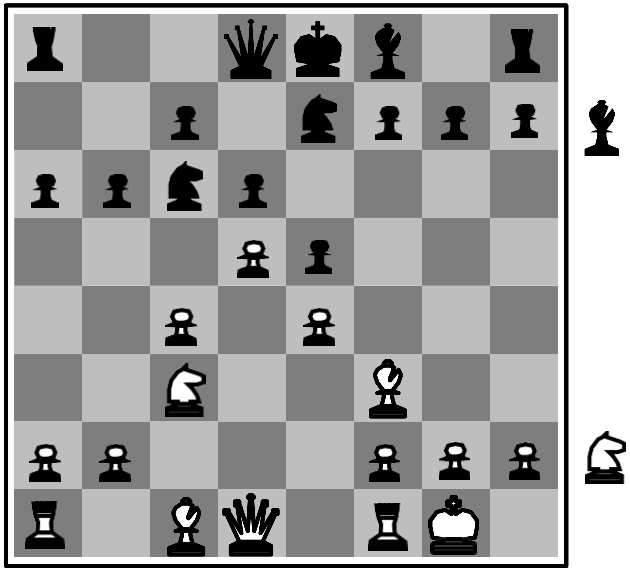
Standard approach:

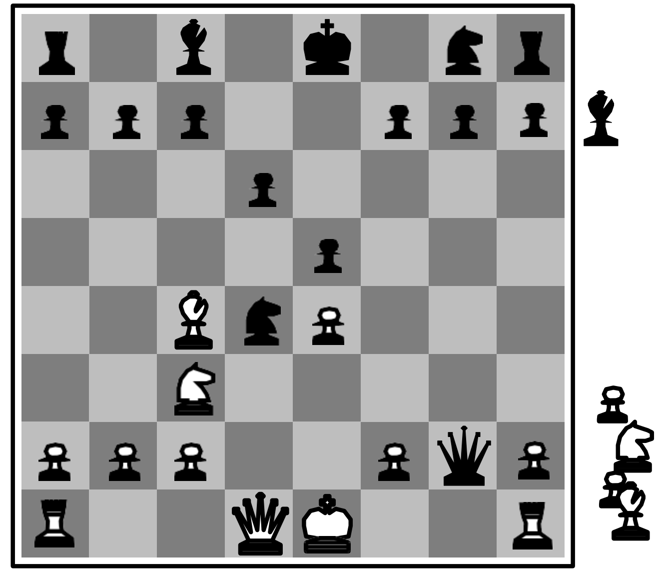
- Use Cutoff-Test instead of Terminal-Test
  e.g., depth limit (perhaps add quiescence search)
- Use Eval instead of Utility
  i.e., evaluation function that estimates desirability of position

Suppose we have 100 seconds, explore 10^4 nodes/second
$\Rightarrow 10^6$ nodes per move $\approx 35^{8/2}$
$\Rightarrow \alpha\text{–}\beta$ reaches depth 8 $\Rightarrow$ pretty good chess program

"I see only one move ahead, but it is always the correct one" – Jose R. Capablanca, world chess champion from 1921-1927

Evaluation functions





Black to move

White slightly better

White to move

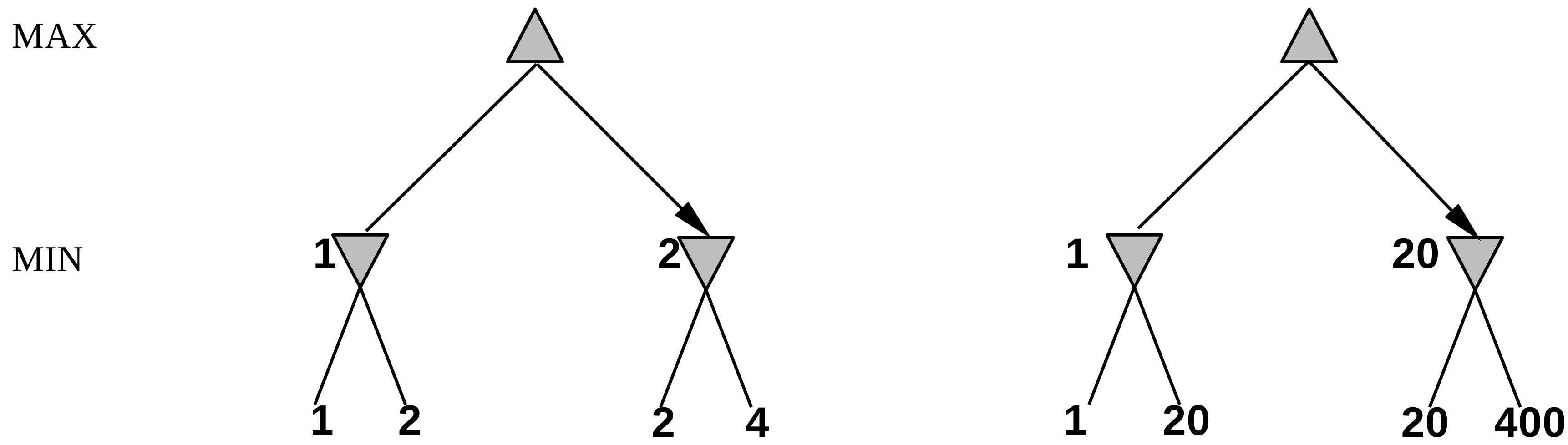
Black winning

For chess, typically linear weighted sum of features

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

e.g., $w_1 = 9$ with
$f_1(s)$ = (number of white queens) – (number of black queens), etc.

Digression: Exact values don't matter



Behaviour is preserved under any monotonic transformation of Eval

Only the order matters:

payoff in deterministic games acts as an ordinal utility function

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

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: human champions refuse to compete against computers, which are too bad. In go,~~ $\cancel{b > 300}$~~, so most programs use pattern knowledge bases to suggest plausible moves~~

Pearson

Go: the last board game

https://www.youtube.com/watch?v=WXuK6gekU1Y&ab_channel=DeepMind

Go: the last board game

Published: 27 January 2016

Mastering the game of Go with deep neural networks and tree search

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel & Demis Hassabis

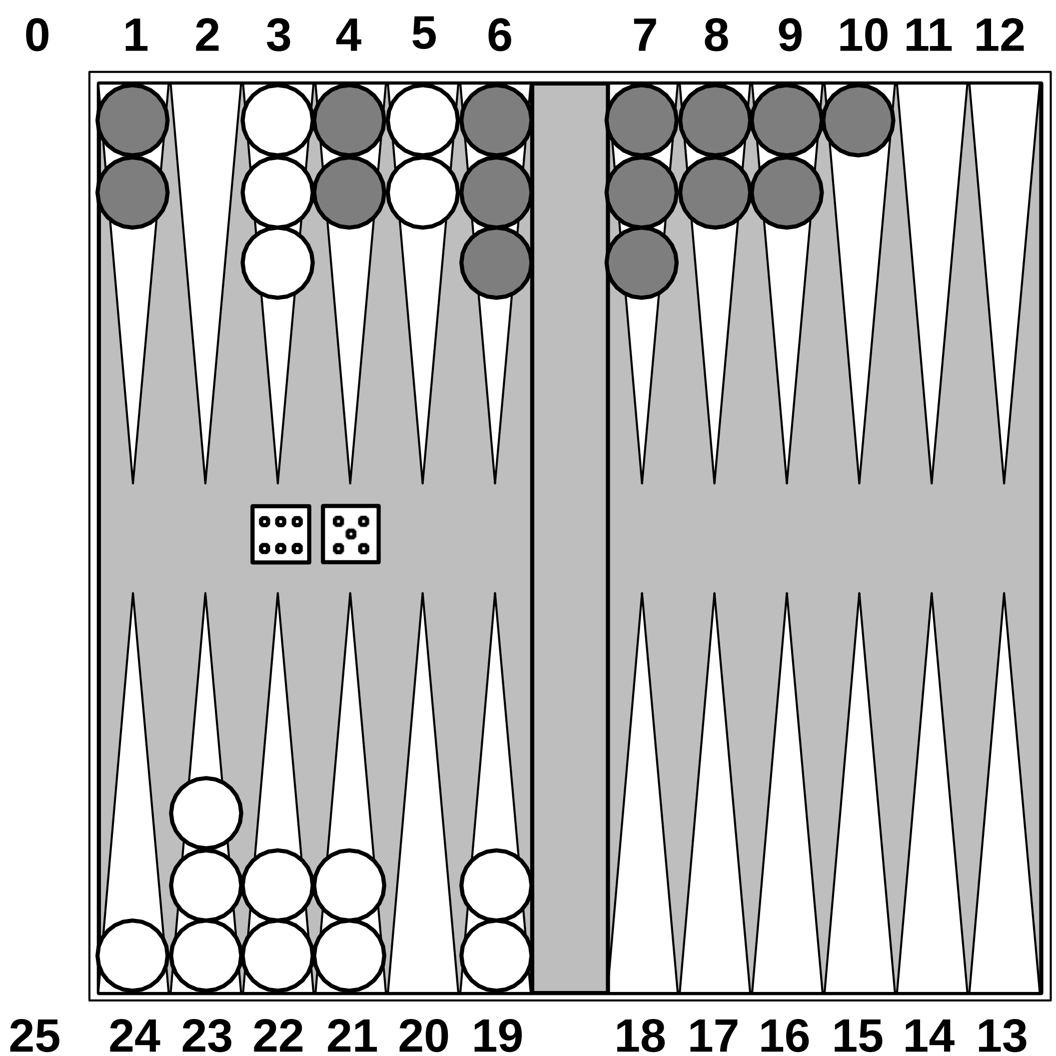
Nature **529**, 484–489 (2016) | Cite this article

Abstract

The game of Go has long been viewed as the most challenging of classic games for artificial intelligence owing to its enormous search space and the difficulty of evaluating board positions and moves. Here we introduce a new approach to computer Go that uses 'value networks' to evaluate board positions and 'policy networks' to select moves. These deep neural networks are trained by a novel combination of supervised learning from human expert games, and reinforcement learning from games of self-play. Without any lookahead search, the neural networks play Go at the level of state-of-the-art Monte Carlo tree search programs that simulate thousands of random games of self-play. We also introduce a new search algorithm that combines Monte Carlo simulation with value and policy networks. Using this search algorithm, our program AlphaGo achieved a 99.8% winning rate against other Go programs, and defeated the human European Go champion by 5 games to 0. This is the first time that a computer program has defeated a human professional player in the full-sized game of Go, a feat previously thought to be at least a decade away.

Nature paper: https://www.nature.com/articles/nature16961

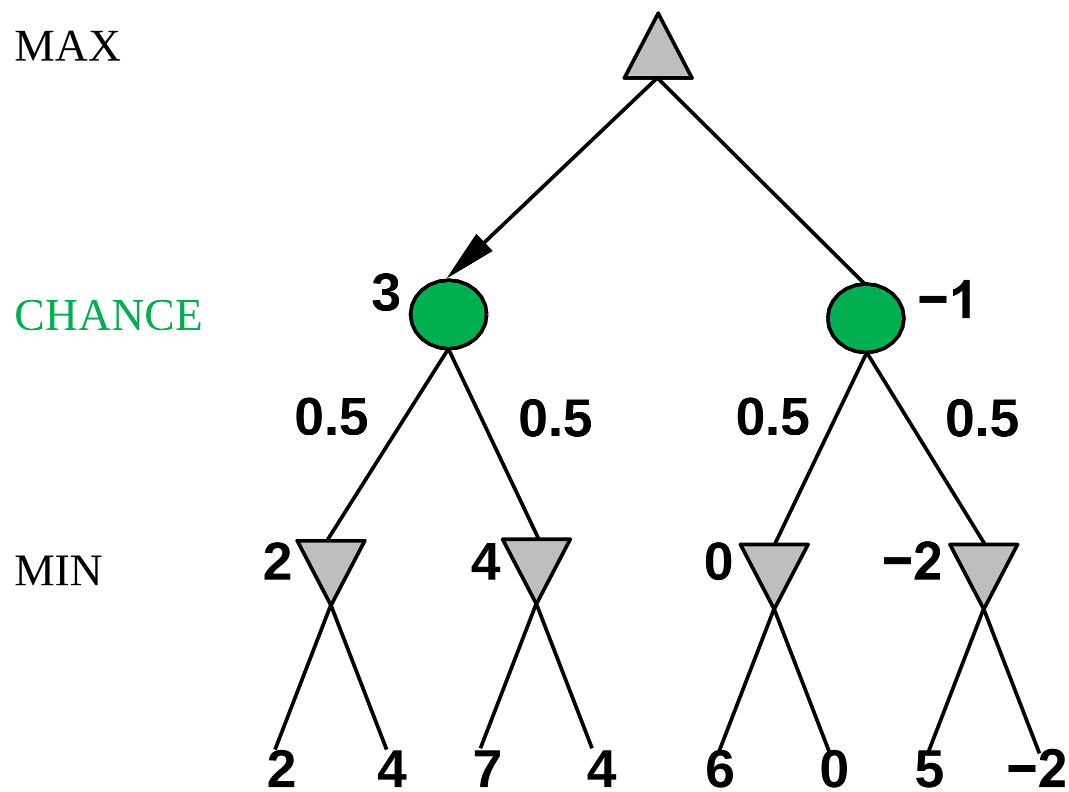
Nondeterministic games: backgammon



Nondeterministic games in general

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

Simplified example with coin-flipping:



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)

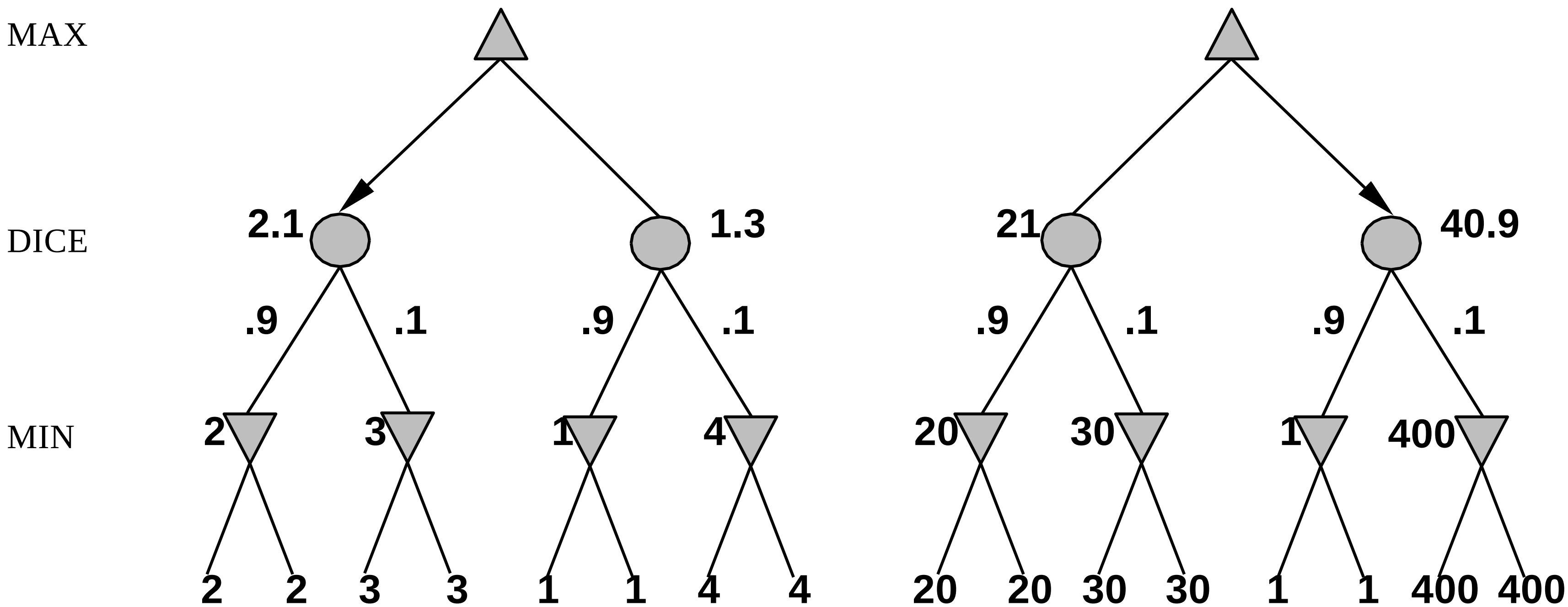
$$\text{depth } 4 = 20 \times (21 \times 20)^3 \approx 1.2 \times 10^9$$

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

Games of imperfect information

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 one big dice roll at the beginning of the game

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

it assumes that once the actual deal has occurred, the game becomes fully observable to both players.

"GIB", current best bridge program, approximates this idea by:

1) generating *100 deals* consistent with bidding information
2) picking the action that wins most tricks **on average**

Proper analysis

The intuition that the value of an action is the average of its values in all actual states is WRONG

With partial observability, value of an action depends on the information state or belief state the agent is in

Can generate and search a tree of information states

Leads to rational behaviors such as

- Acting to obtain information
- Signalling to one's partner
- Acting randomly to minimize information disclosure

Limitations of Game Search Algorithms

- Alpha–beta search vulnerable to errors in the heuristic function.
- Waste of computational time for deciding best move where it is obvious (meta-reasoning).
- Reasoning done on individual moves, while Humans reason on abstract levels (sometimes informally)
- Possibility to incorporate Machine Learning into game search process.

AIMA Notebook

1667 lines (1667 sloc) | 46.5 KB

Raw Blame

Game Tree Search

We start with defining the abstract class `Game`, for turn-taking *n*-player games. We rely on, but do not define yet, the concept of a `state` of the game; we'll see later how individual games define states. For now, all we require is that a state has a `state.to_move` attribute, which gives the name of the player whose turn it is. ("Name" will be something like `'X'` or `'O'` for tic-tac-toe.)

We also define `play_game`, which takes a game and a dictionary of `{player_name: strategy_function}` pairs, and plays out the game, on each turn checking `state.to_move` to see whose turn it is, and then getting the strategy function for that player and applying it to the game and the state to get a move.

In [1]:

```
from collections import namedtuple, Counter, defaultdict
import random
import math
import functools
cache = functools.lru_cache(10**6)
```

In [73]:

```
class Game:
    """A game is similar to a problem, but it has a terminal test instead of
    a goal test, and a utility for each terminal state. To create a game,
    subclass this class and implement `actions`, `result`, `is_terminal`,
    and `utility`. You will also need to set the .initial attribute to the
    initial state; this can be done in the constructor."""

    def actions(self, state):
        """Return a collection of the allowable moves from this state."""
        raise NotImplementedError

    def result(self, state, move):
        """Return the state that results from making a move from a state."""
        raise NotImplementedError

    def is_terminal(self, state):
        """Return True if this is a final state for the game."""
        return not self.actions(state)

    def utility(self, state, player):
        """Return the value of this final state to player."""
        raise NotImplementedError


def play_game(game, strategies: dict, verbose=False):
    """Play a turn-taking game. `strategies` is a {player_name: function} dict,
    where function(state, game) is used to get the player's move."""
    state = game.initial
    while not game.is_terminal(state):
        player = state.to_move
        move = strategies[player](game, state)
        state = game.result(state, move)
        if verbose:
```

https://github.com/aimacode/aima-python/blob/master/games4e.ipynb

Summary

Minimax algorithm: selects optimal moves by a depth-first enumeration of the game tree.

Alpha–beta algorithm: greater efficiency by eliminating subtrees

Evaluation function: a heuristic that estimates utility of state.

Monte Carlo tree search (MCTS): no heuristic, play game to the end with rules and repeated multiple times to determine optimal moves during playout.

In the next lecture...

- Knowledge-based agents
- Wumpus world
- Logic in general—models and entailment
- Propositional (Boolean) logic
- Equivalence, validity, satisfiability
- Inference rules and theorem proving
  - resolution
  - forward chaining
  - backward chaining
- Effective Propositional Model Checking