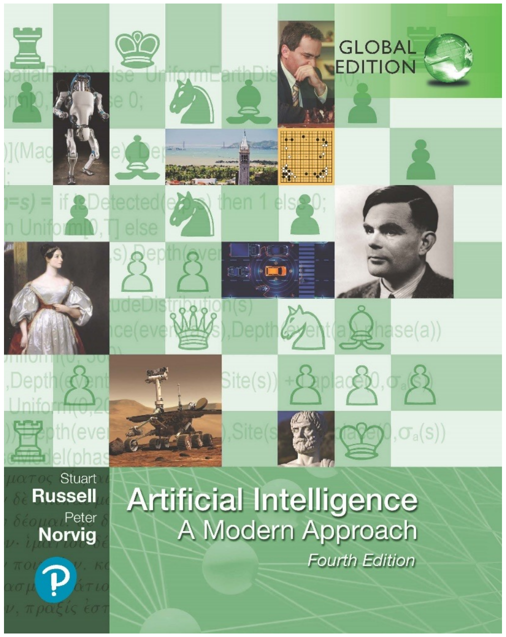
Artificial Intelligence: A Modern Approach

Fourth Edition, Global Edition



Adversarial Search And Games

3rd Edition – Chapter 5
4th Edition - Chapter 6

Outline

◊ Games

◊ Perfect play
  – minimax decisions
  – α–β pruning

◊ Resource limits and approximate evaluation

◊ Games of chance

◊ Games of imperfect information

Games

One of the oldest areas of AI

Chess programs were especially chosen because success would be a proof of a machine doing something intelligent.

Types of games

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

Games vs. Search problems: Uncertainty

- The presence of an opponent that introduces uncertainty makes the decision problem more complicated than regular search problems.

- “Unpredictable” opponent: solution is a strategy specifying a move for every possible opponent reply

Optimal Decisions with Minimax

3rd ed: Section 5.2

4th ed: pp. [193-196]

Maximizes the worst-case outcome for MAX

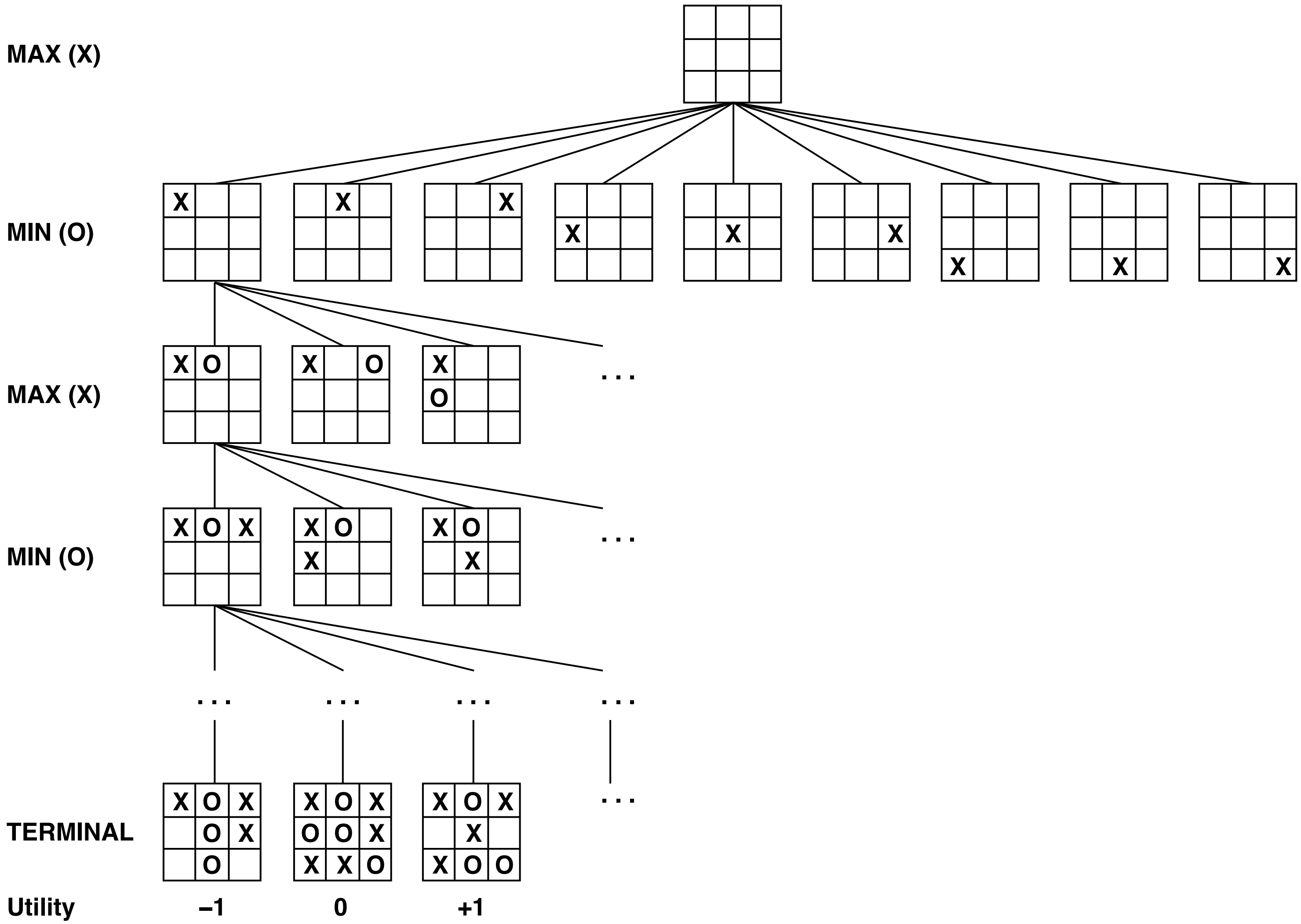
Perfect Decisions in Two-Person Games

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

- ♢ initial state (of the board and whose turn it is)
- ♢ set of operators (which define the legal moves)
- ♢ terminal test (goal test)
- ♢ utility function: (numeric value for the outcome of a game)

Ex. backgammon (+1, -1, +2); Chess (win, lose, draw)... which is a zero-sum game.

Game tree (2-player, deterministic, turns)



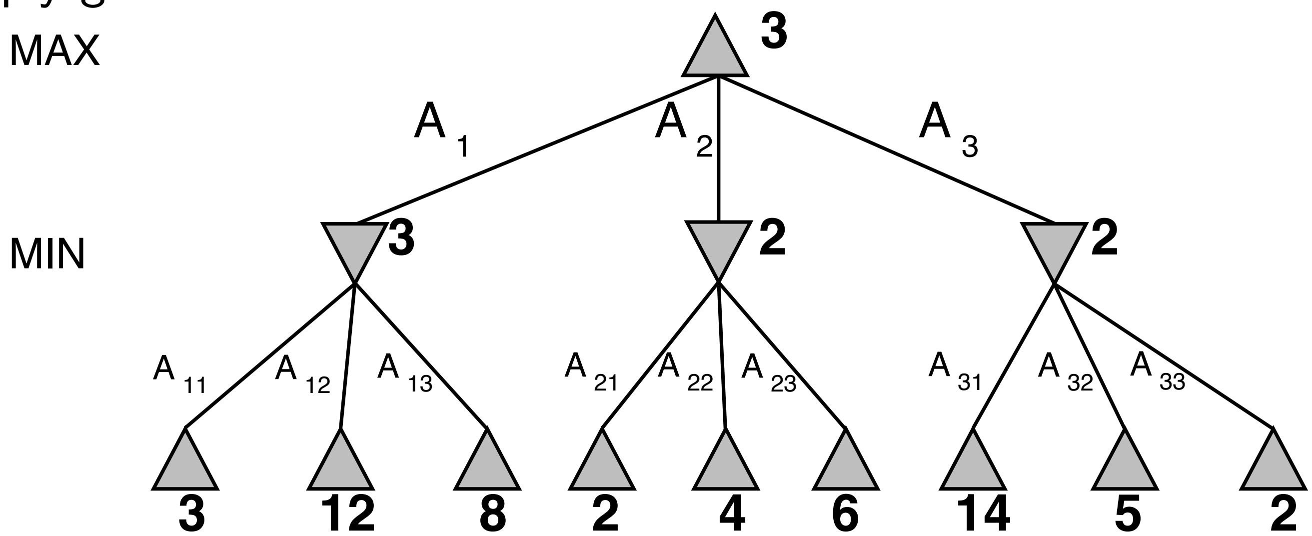
Minimax

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

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

Minimax

E.g., 2-ply game:



Minimax algorithm

function MINIMAX-DECISION(*game*) **returns** *an operator*

    for each *op* **in** OPERATORS[*game*] **do**
        VALUE[*op*] ← MINIMAX-VALUE(APPLY(*op*, *game*), *game*)
    end
    return the *op* with the highest VALUE[*op*]

---

function MINIMAX-VALUE(*state*, *game*) **returns** *a utility value*

    if TERMINAL-TEST[*game*](*state*) **then**
        return UTILITY[*game*](*state*)
    else if MAX is to move in *state* **then**
        return the highest MINIMAX-VALUE of SUCCESSORS(*state*)
    else
        return the lowest MINIMAX-VALUE of SUCCESSORS(*state*)

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

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?? 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

Optimal Decisions with Minimax & Alpha-Beta Pruning

3rd ed: Section 5.3

4th ed: pp. [198-201)

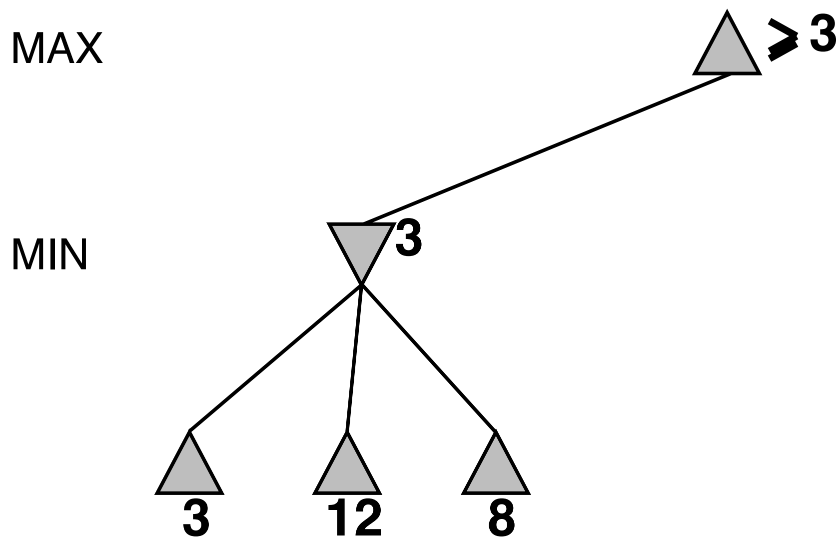
Still optimal – Prunes subtrees that would not be selected

α–β pruning

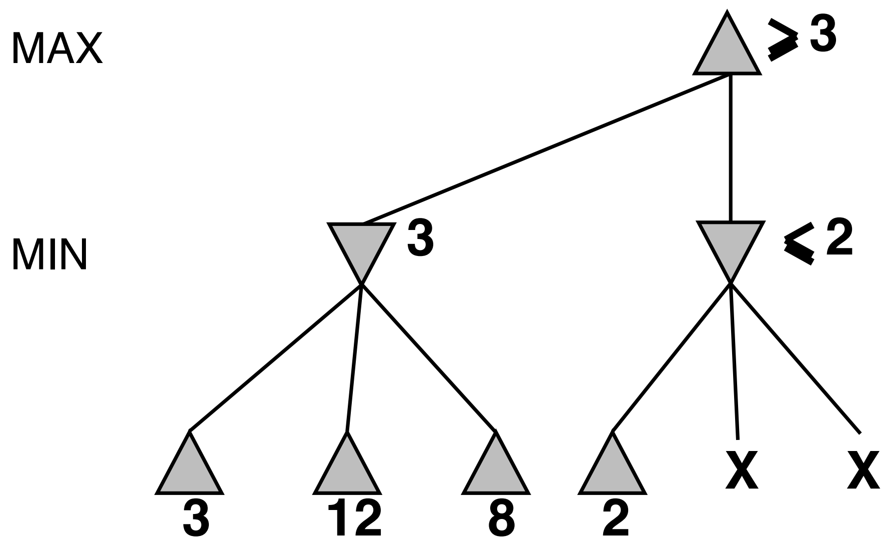
With minimax 4-ply possible, but even average human players can make plans 6-8 ply ahead!

How can we improve minimax search?

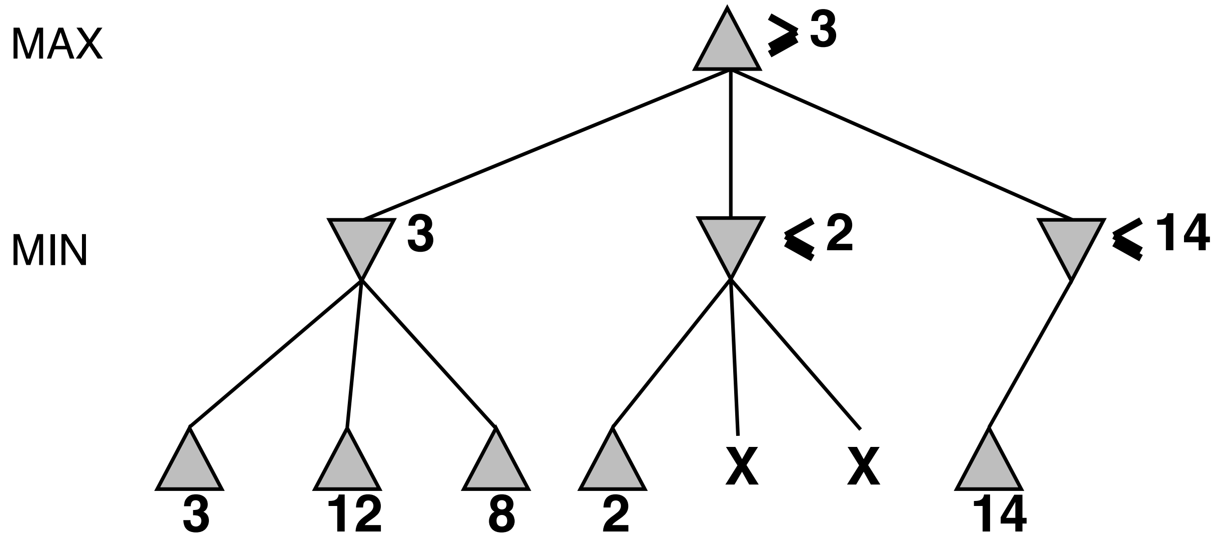
α–β pruning example



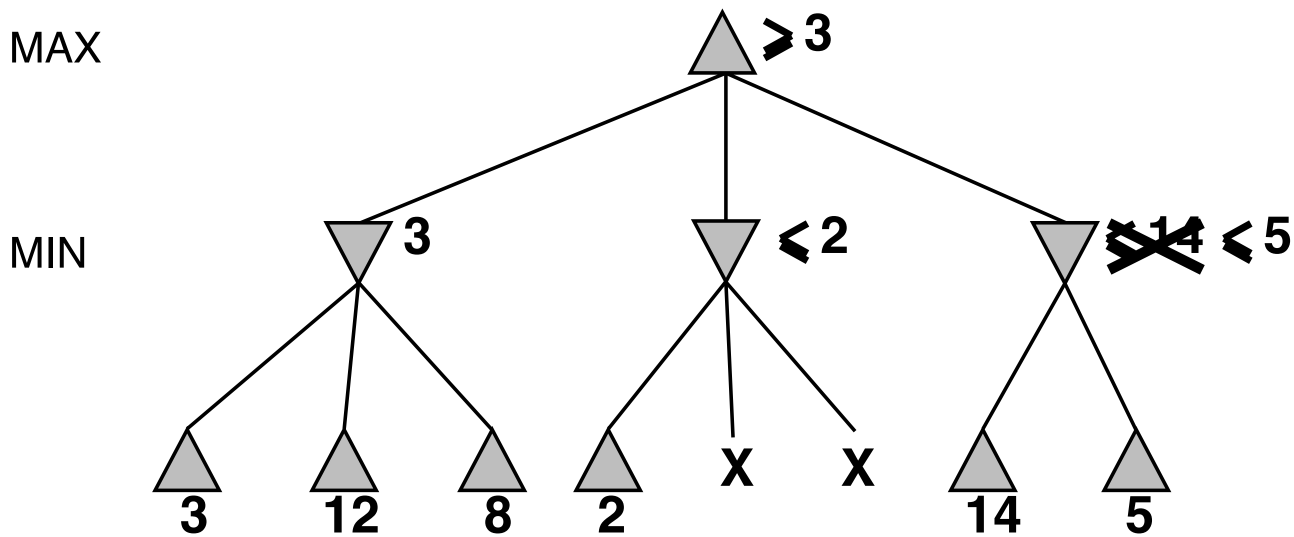
α–β pruning example



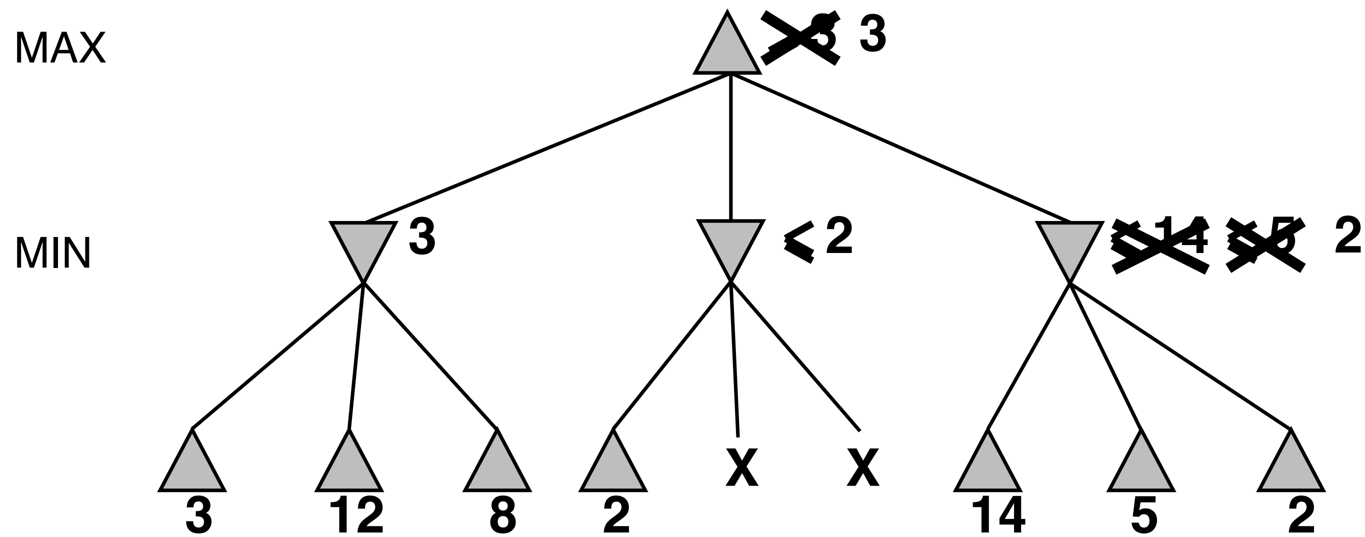
α–β pruning example



α–β pruning example



α–β pruning example



Alpha-Beta Pseudocode

Basically Minimax + keep track of α, β + prune

inputs: state, current game state
α, value of best alternative for MAX on path to state
β, value of best alternative for MIN on path to state
returns: a utility value

```
function MAX-VALUE(state,α,β)
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
```

At max node:
Prune if $v \geq \beta$;
Update $\alpha \leq$

```
function MIN-VALUE(state,α,β)
if  TERMINAL-TEST(state) then
      return UTILITY(state)
v ← +∞
for a, s in SUCCESSORS(state) do
      v ← MIN(v, MAX-VALUE(s,α,β))
      if v ≤ α then return v
      β ← MIN(β,v)
return v
```

At min node:
Prune if $\alpha \leq v$;
Update β

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!

Imperfect Decisions with Minimax

3rd ed: Section 5.4

4th ed: pp. [202-204]

Cutting Off Search with Eval Function

Imperfect decisions

The minimax algorithm assumes that the program has time to search all the way down to terminal states, which is usually not practical.

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** to the leaves of the tree.

Resource limits

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

$\Rightarrow$ 10^6 nodes per move

Standard approach:

- *cutoff test*

  e.g., depth limit

- *evaluation function*

  = estimated desirability of position

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 $\approx 35^{8/2}$
$\Rightarrow$ α–β reaches depth 8 $\Rightarrow$ pretty good chess program

Cutting off search

Should look further:

♢ in positions where favorable captures can be made (non-quiescent positions)

♢ in positions where unavoidable (but beyond the horizon moves) will affect the situation drastically

e.g. pawn turning into queen

Evaluation functions

Estimate of the expected utility of the game from a given position.

E.g. material value for each piece: 1 for pawn, 3 for knight or bishop,...

Performance of a game-playing program is extremely dependent on the quality of its evaluation function.

Evaluation functions

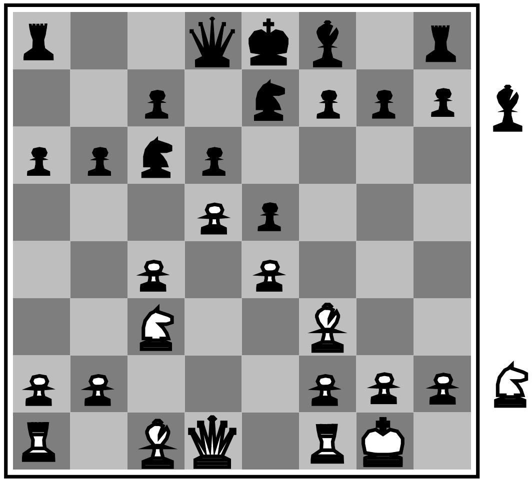
Evaluation functions:

♢ should agree with the utility function on terminal states.

♢ must not take too long to calculate!

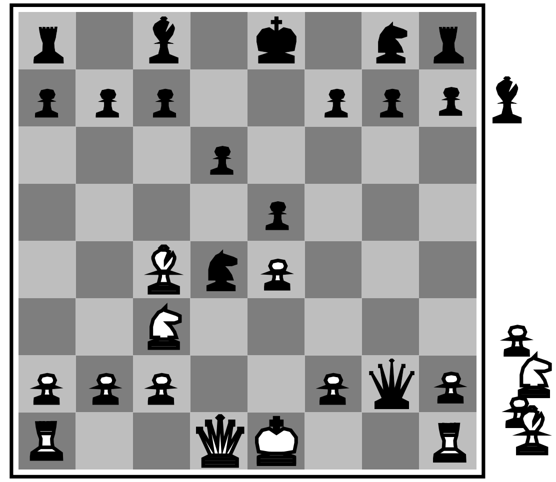
♢ should accurately reflect the *chances* of winning (if we have to cut-off, we do not know what will happen in subsequent moves)

Evaluation functions



Black to move

White slightly better



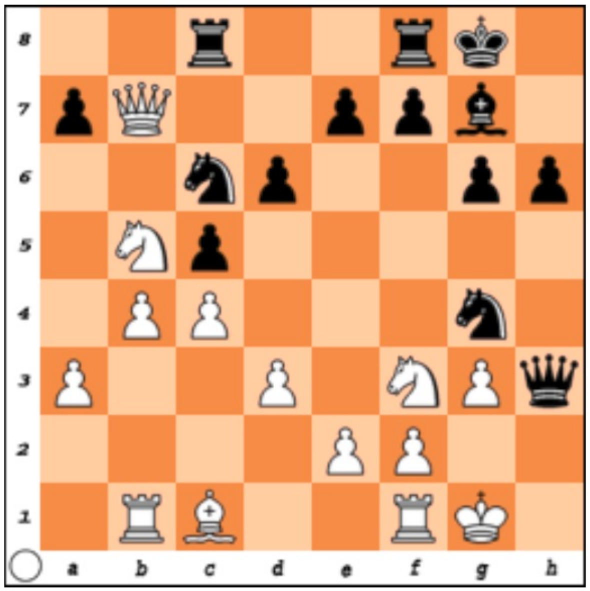
White to move

Black winning

For chess, typically *linear* weighted sum of <u>features</u>

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

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

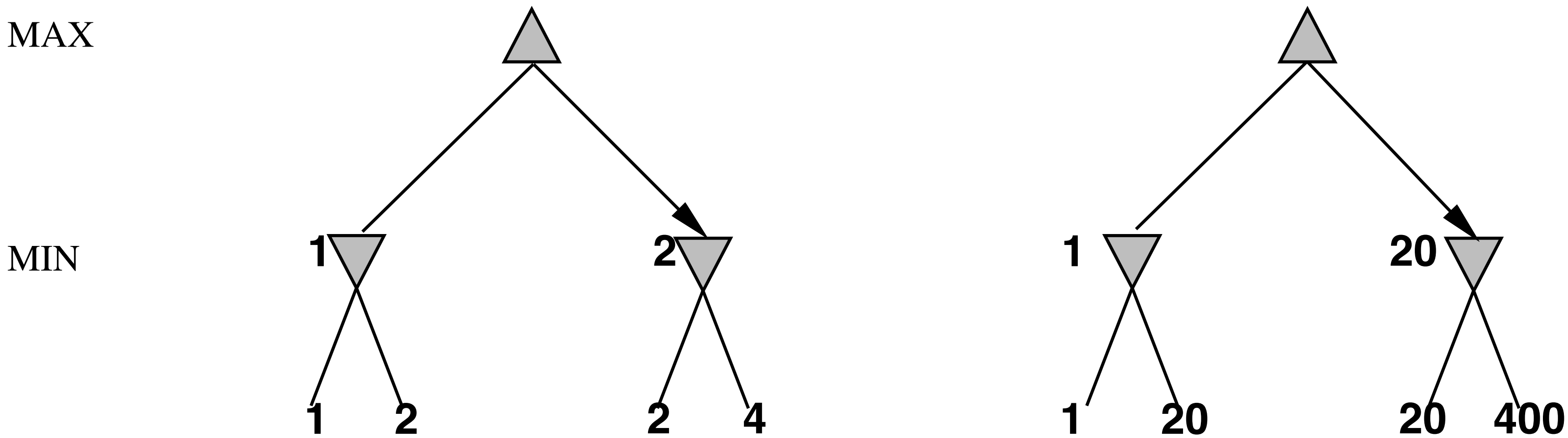


From: M. Lai, MS Thesis, Giraffe: Using Deep Reinforcement Learning to Play Chess

https://arxiv.org/abs/1509.01549

Feature	Value
Side to Move	White
White Long Castle	No
White Short Castle	No
Black Long Castle	No
Black Short Castle	No
White Queens	1
White Rooks	2
White Bishops	1
White Knights	2
White Pawns	7
Black Queens	1
Black Rooks	2
Black Bishops	1
Black Knights	2
Black Pawns	7
White Queen 1 Exists	Yes
White Queen 1 Position	b7
White Rook 1 Exists	Yes
White Rook 1 Position	b1
White Rook 2 Exists	Yes
White Rook 2 Position	f1
White Bishop 1 Exists	Yes
White Bishop 1 Position	c1
White Bishop 2 Exists	No
White Bishop 2 Position	N/A
...	

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

Evaluations with Monte Carlo Tree Search

4^{th} ed: pp. [207-208]

Monte Carlo tree search

- $\alpha-\beta$ pruning is not very effective in Go
  - Large branching factor (361), thus small depth (4-5 ply)
  - Hard to define a good evaluation function

- Use Monte-Carlo Tree Search
  - Instead of a heuristic evaluation function, play a large number of simulations/playouts
    - **Choose moves for the player and later for the opponent until a terminal position is reached**
  - Average utility of a state can be derived from the win percentages from that state

- Should we try random moves?
  - Playout policy biases moves towards good ones.

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. Baron von Kempelen's "Turk" in 1769!

Backgammon: First program to make a serious impact, BKG, used only a one-ply search but a very complicated evaluation function (1980). It plays a strong amateur level. In 1992, neural network techniques to learn evaluation function.

Othello/Reversi: Smaller search space ($b = 5 - 15$). 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.

Stochastic Games

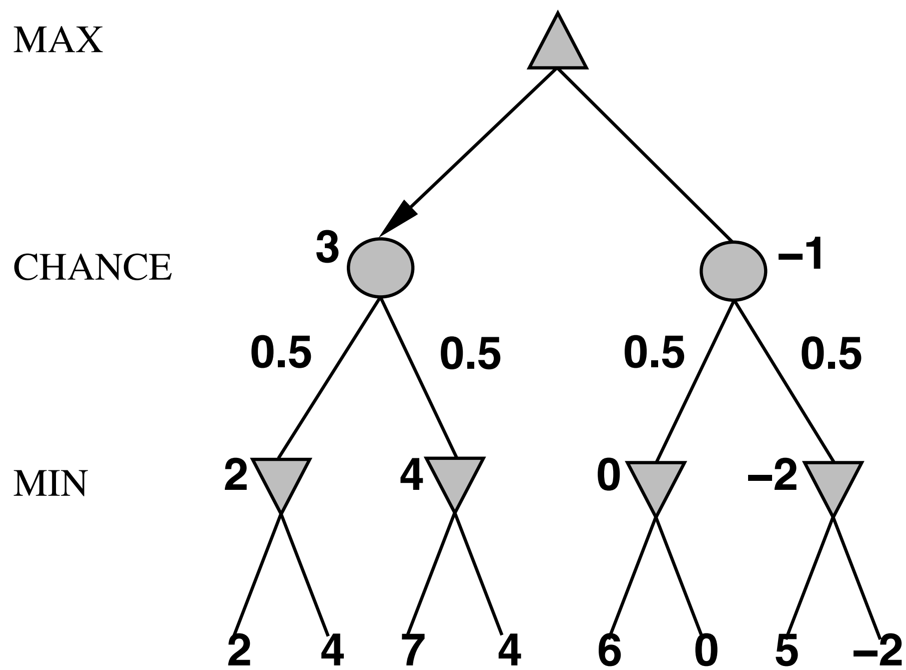
3rd ed: Section 5.5

4th ed: Section 6.5

Nondeterministic games

E..g, in backgammon, the dice rolls determine the legal moves

Simplified example with coin-flipping instead of dice-rolling:



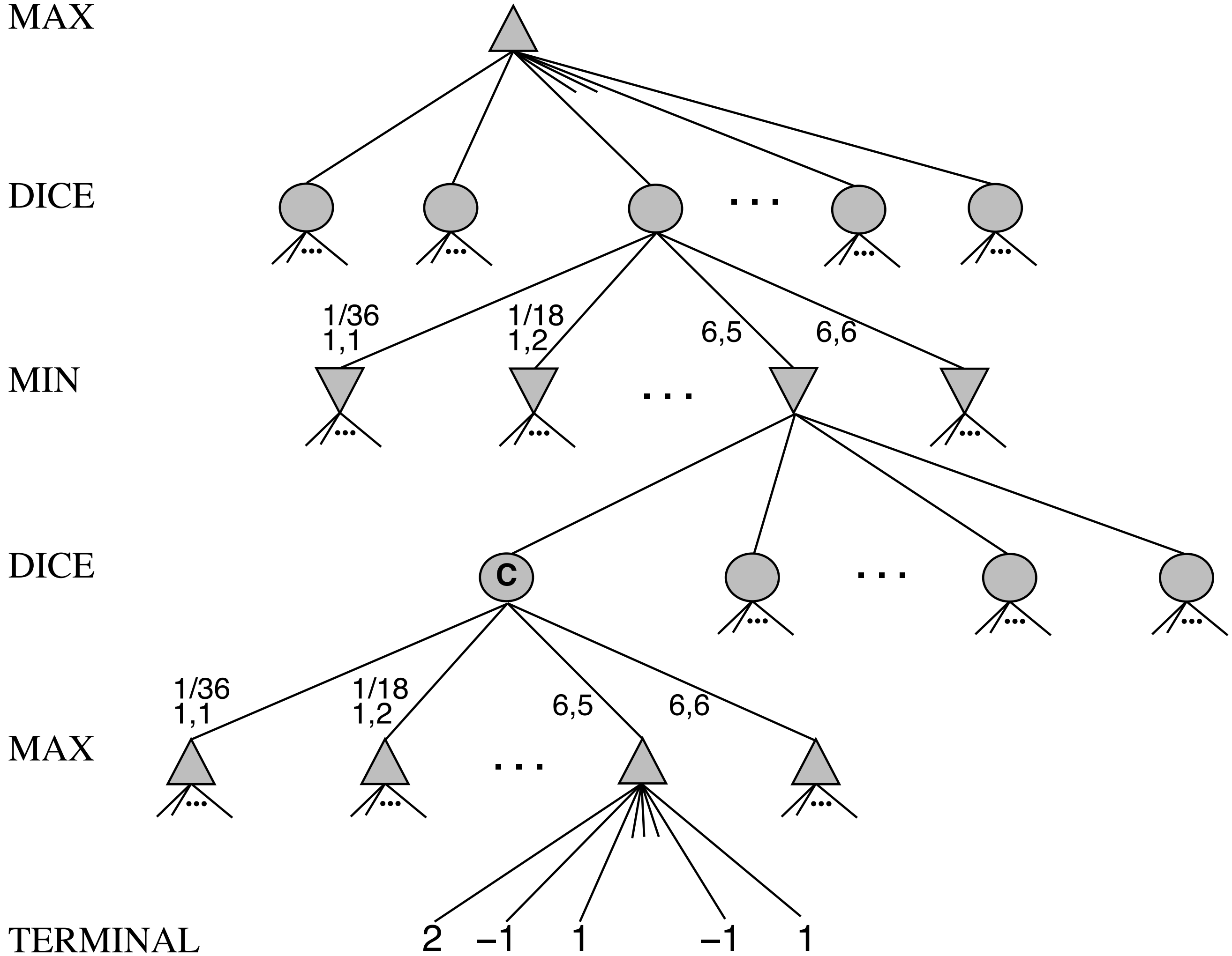
Algorithm for nondeterministic games

EXPECTIMINIMAX gives perfect play

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

. . .
if *state* is a chance node **then**
    return average of EXPECTIMINIMAX-VALUE of SUCCESSORS(*state*)

Nondeterministic games



Rest?

Nondeterministic games in practice

Dice rolls increase b: 21 possible rolls with 2 dice

Backgammon $\approx$ 20 legal moves

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

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

TDGammon uses depth-2 search + very good Eval
$\approx$ world-champion level

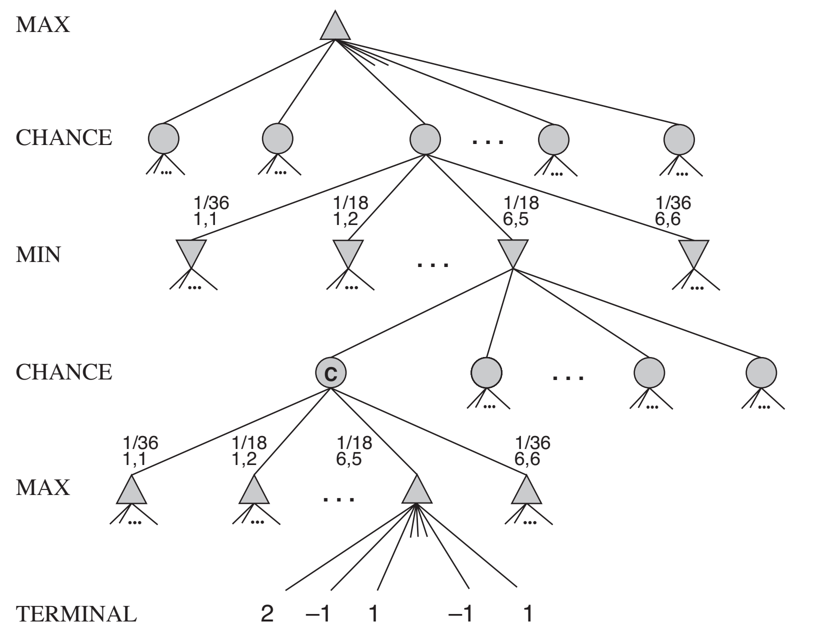


Figure 5.11 Schematic game tree for a backgammon position.

The next step is to understand how to make correct decisions. Obviously, we still want to pick the move that leads to the best position. However, positions do not have definite minimax values. Instead, we can only calculate the **expected value** of a position: the average over all possible outcomes of the chance nodes.

EXPECTED VALUE

This leads us to generalize the **minimax value** for deterministic games to an **expectiminimax value** for games with chance nodes. Terminal nodes and MAX and MIN nodes (for which the dice roll is known) work exactly the same way as before. For chance nodes we compute the expected value, which is the sum of the value over all outcomes, weighted by the probability of each chance action:

EXPECTIMINIMAX VALUE

$$\textsc{Expectiminimax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\ \max_a \textsc{Expectiminimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\ \min_a \textsc{Expectiminimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{min} \\ \sum_r P(r)\textsc{Expectiminimax}(\textsc{Result}(s,r)) & \text{if } \textsc{Player}(s) = \textsc{chance} \end{cases}$$

where r represents a possible dice roll (or other chance event) and $\textsc{Result}(s,r)$ is the same state as s, with the additional fact that the result of the dice roll is r.

5.5.1 Evaluation functions for games of chance

As with minimax, the obvious approximation to make with expectiminimax is to cut the search off at some point and apply an evaluation function to each leaf. One might think that evaluation functions for games such as backgammon should be just like evaluation functions

Nondeterministic games in practice

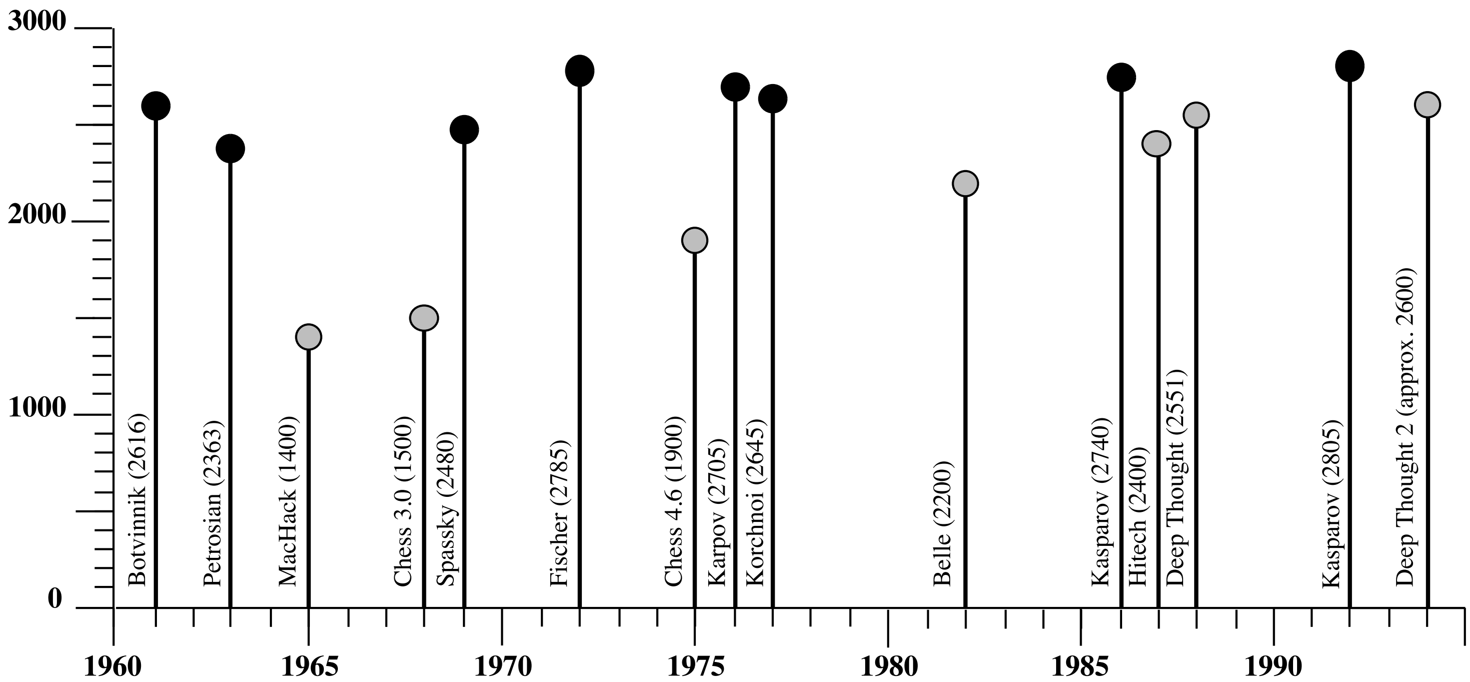
α–β pruning is much less effective:

The advantage of α–β pruning is that it ignores future developments that just are not going to happen, given best play.

Thus in concentrates on likely moves.

In games with dice, there are no likely sequences of moves, because for thos moves to take place, the dice would have to come out the right way to make them legal.

Nondeterministic games in practice



Extrapolate?

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*

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

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

Summary

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

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