Game Playing and Adversarial Search

Chapter 5, Sections 1–5

Outline

◇ Perfect play

◇ Resource limits

◇ α–β pruning

◇ Games of chance

Games vs. search problems

"Unpredictable" opponent ⇒ solution is a contingency plan

Time limits ⇒ unlikely to find goal, must approximate

Plan of attack:

- algorithm for perfect play (Von Neumann, 1944)
- finite horizon, approximate evaluation (Zuse, 1945; Shannon, 1950; Samuel, 1952–57)
- pruning to reduce costs (McCarthy, 1956)

Types of games

randomness

observability

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

two agents.

selfish agent

limited resource

best return

Representing a game as a search problem

We can formally define a strategic two-player game by:

- initial state
- actions
- terminal test (i.e. win / lose / draw)
- utility function (i.e. numeric reward for outcome)
  chess: +1, 0, -1
  poker: cash won or lost

In a zero-sum game with 2 players:
each player's utility for a state are equal and opposite

Minimax

Perfect play for deterministic, perfect-information games

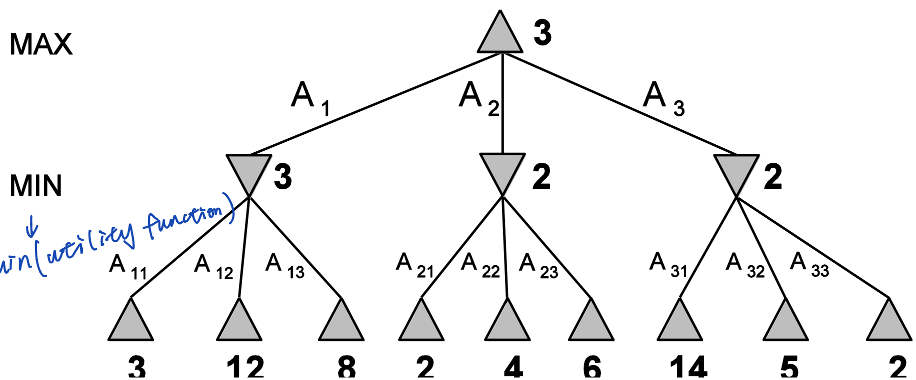
Idea: choose move to position with highest *minimax value*
= best achievable payoff against best play

E.g., 2-ply game:

2-rounds ⇒ determine the depth of tree

forward path (no minmax value at parent value to the terminal state and utility value)

min(utility function)



backward path propagate back find minmax value of parent node

Minimax algorithm

function MINIMAX-DECISION(*game*) **returns** *an operator* lookat each operation

  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*)

otherwise in MAX or MIN node

expand in DFS manner

Properties of minimax

Complete??

Optimal??

Time complexity??

Space complexity??

Properties of minimax

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

rational

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

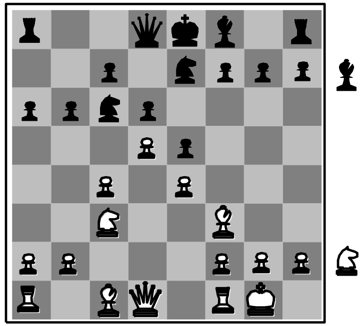
Resource limits

Suppose we have 100 seconds, explore 10^4 nodes/second
$\Rightarrow \underline{10^6}$ nodes per move

Standard approach:

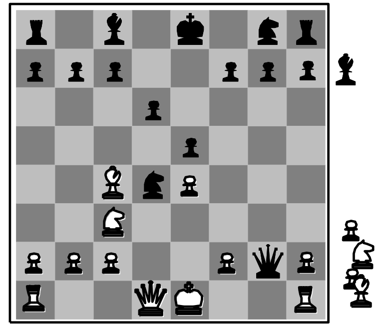
- *cutoff test*
  e.g., depth limit (perhaps add *quiescence search*)
- *evaluation function* don't get to terminal state, can't use utility function
  = estimated desirability of position

Evaluation functions



Black to move

White slightly better



White to move

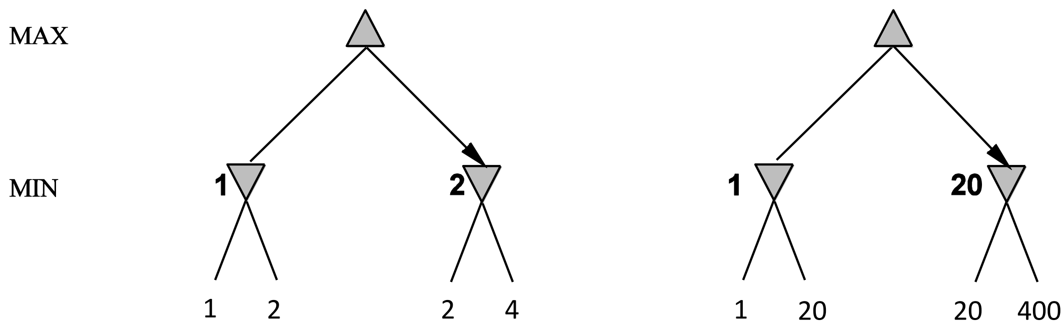
Black winning

For chess, typically *linear* weighted sum of features

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

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

Cutting off search

MinimaxCutoff is identical to MinimaxValue except

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

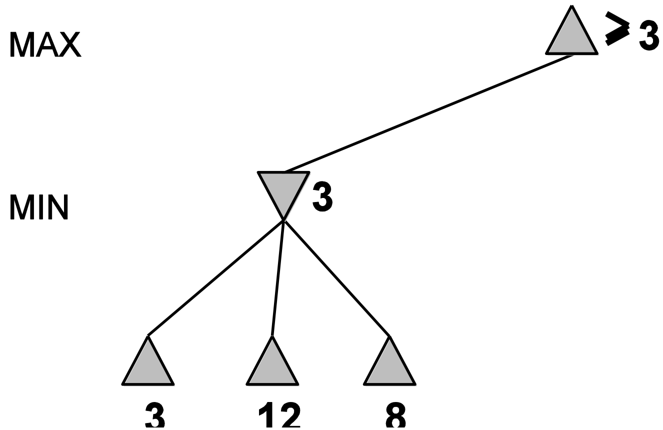
Does it work in practice?

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

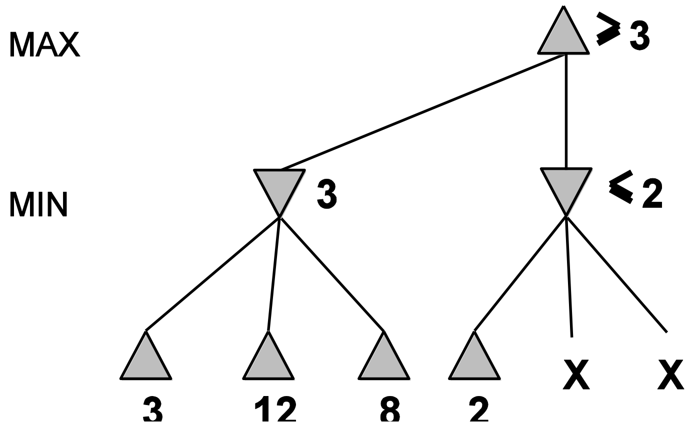
4-ply lookahead is a hopeless chess player!

4-ply ≈ human novice
8-ply ≈ typical PC, human master
12-ply ≈ IBM's Deep Blue, Kasparov

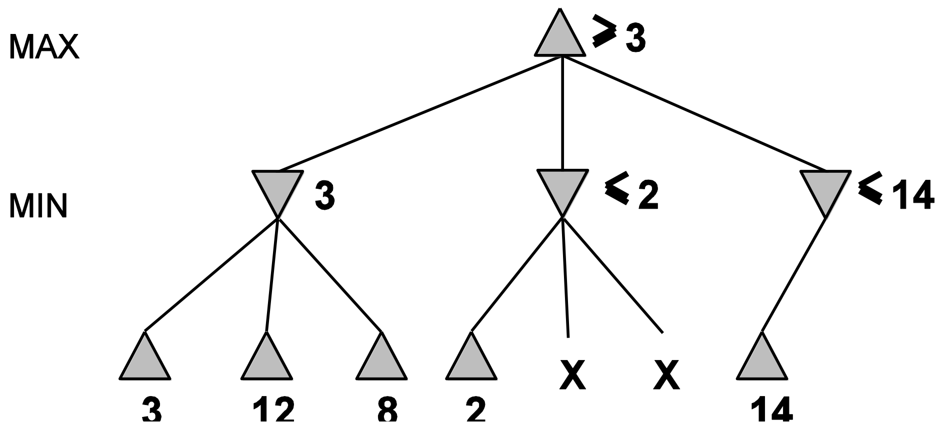
α–β pruning example



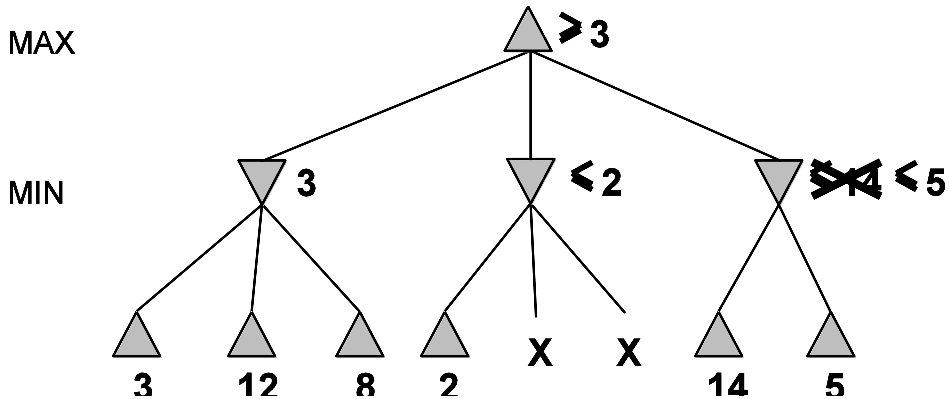
α–β pruning example



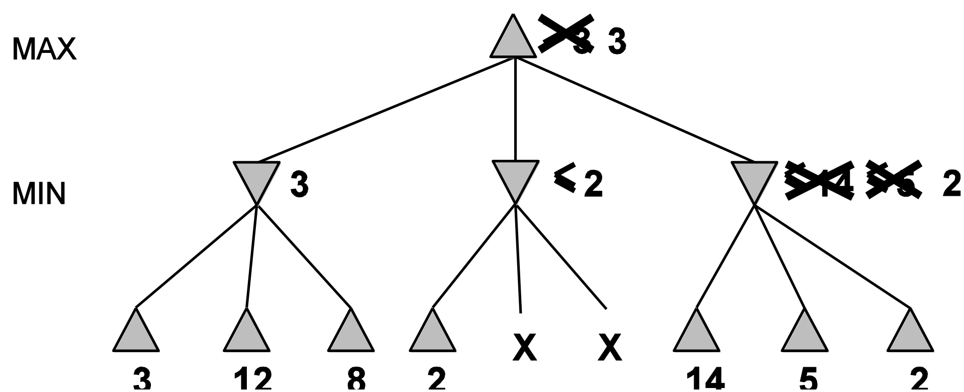
α–β pruning example



α–β pruning example



α–β pruning example



Properties of α–β

Pruning *does not* affect final result

Good move ordering improves effectiveness of pruning

from $O(b^m) \to$ to

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

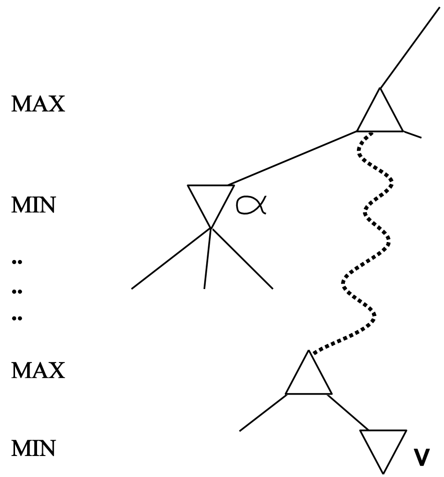
$\Rightarrow$ *doubles* depth of search

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

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

search control learning

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 for min β is the smallest (best value to min)

if $V > \beta$, min will avoid it

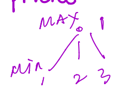
The α–β algorithm

Basically MINIMAX + keep track of α, β + prune

function MAX-VALUE(*state, game*, α, β) **returns** the minimax value of *state*
  inputs: *state*, current state in game
    game, game description
    α, the best score for MAX along the path to *state* $\in(-\infty,\infty)$
    β, the best score for MIN along the path to *state* $\in(-\infty,\infty)$

initially have no idea

  if CUTOFF-TEST(*state*) **then return** EVAL(*state*)
  for each s **in** SUCCESSORS(*state*) **do**
    $\alpha \leftarrow$ MAX(α, MIN-VALUE(s, *game*, α, β)) → if min-value > α, update α
    if $\alpha \geq \beta$ **then return** β → prune
  end
  return α



$\alpha=-\infty$
$\beta=\infty$.

如果现在的max值为α，min node选β
$\alpha\geq\beta$，则 max node不会选β
prune the branch

---

function MIN-VALUE(*state, game*, α, β) **returns** the minimax value of *state*

  if CUTOFF-TEST(*state*) **then return** EVAL(*state*)
  for each s **in** SUCCESSORS(*state*) **do**
    $\beta \leftarrow$ MIN(β, MAX-VALUE(s, *game*, α, β))
    if $\beta \leq \alpha$ **then return** α
  end
  return β

Some drawbacks of α–β pruning

◇ It is in most cases not feasible to search the entire game tree, a depth limit needs to be set.

◇ Alpha-Beta is designed to select a good move but it also calculates the values of all legal moves.

A better method maybe to use what is called the *utility of a node expansion*. In this way a good search algorithm could select a node that had a high utility to expand (these will hopefully lead to better moves). This could lead to a faster decision by searching through a smaller decision space. A extension on those abilities would be the use of another technique called *goal-directed reasoning*. This technique focuses on having a certain goal in mind like capturing the queen in chess.

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.

all possible moves for state that are close to terminal state

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.

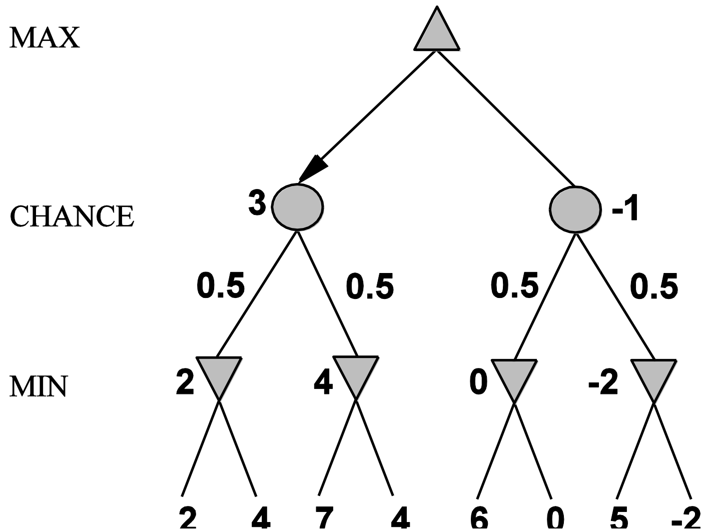
Othello: human champions refuse to compete against computers, who are too good.

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

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*)
...

A version of α–β pruning is possible
but only if the leaf values are bounded. Why??

?

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)

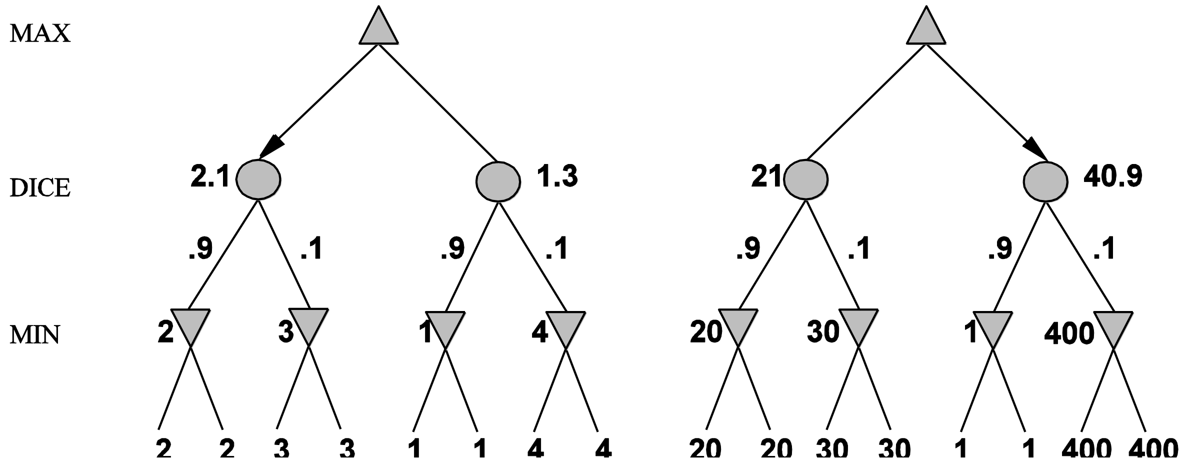
$$\text{depth } 4 = 20 \times (21 \times 20)^3 \approx 1.2 \times 10^9$$

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

Summary

Games illustrate several important points about AI

◇ perfection is unattainable ⇒ must approximate and make trade-offs

◇ uncertainty limits the value of look-ahead

◇ can programs learn for themselves as they play? (stay tuned...)

Examples of skills expected:

◇ Demonstrate operation of game search algorithms

◇ Discuss and evaluate the properties of game search algorithms

◇ Design suitable evaluation functions for a game

◇ Explain how to search in nondeterministic games