Last update: March 9, 2010

Game playing

CMSC 421, Chapter 6

Finite perfect-information zero-sum games

Finite: finitely many agents, actions, states

Perfect information: every agent knows the current state,
all of the actions, and what they do
No simultaneous actions – players move one-at-a-time

Constant-sum: regardless of how the game ends, $\sum\{\text{agents' utilities}\} = k$.
For every such game, there's an equivalent game in which $(k = 0)$.
Thus constant-sum games usually are called **zero-sum** games

Examples:

Deterministic: chess, checkers, go, othello (reversi), connect-four, qubic, mancala (awari, kalah), 9 men's morris (merelles, morels, mill)

Stochastic: backgammon, monopoly, yahtzee, parcheesi, roulette, craps

We'll start with deterministic games

Outline

- ♢ A brief history of work on this topic
- ♢ The minimax theorem
- ♢ Game trees
- ♢ The minimax algorithm
- ♢ α-β pruning
- ♢ Resource limits and approximate evaluation

A brief history

1846 (Babbage): machine to play tic-tac-toe

1928 (von Neumann): minimax theorem

1944 (von Neumann & Morgenstern): backward-induction algorithm (produces perfect play)

1950 (Shannon): minimax algorithm (finite horizon, approximate evaluation)

1951 (Turing): program (on paper) for playing chess

1952–7 (Samuel): checkers program, capable of beating its creator

1956 (McCarthy): pruning to allow deeper search

1957 (Bernstein): first complete chess program, on an IBM 704 vacuum-tube computer, could examine about 350 positions/minute

A brief history, continued

1967 (Greenblatt): first program to compete in human chess tournaments:
3 wins, 3 draws, 12 losses

1992 (Schaeffer): Chinook won the 1992 US Open checkers tournament

1994 (Schaeffer): Chinook became world checkers champion;
Tinsley (human champion) withdrew for health reasons

1997 (Hsu *et al*): Deep Blue won 6-game chess match against
world chess champion Gary Kasparov

2007 (Schaeffer *et al*, 2007): Checkers solved: with perfect play,
it's a draw. This took 10^{14} calculations over 18 years

Basics

- ♢ A *strategy* specifies what an agent will do in every possible situation
- ♢ Strategies may be *pure* (deterministic) or *mixed* (probabilistic)

Suppose agents A and B use strategies s and t to play a two-person zero-sum game G. Then

- A's expected utility is $U_A(s,t)$
  From now on, we'll just call this $U(s,t)$
- Since G is zero-sum, $U_B(s,t) = -U(s,t)$

Instead of A and B, we'll call the agents Max and Min

Max wants to maximize U and Min wants to minimize it

The Minimax Theorem (von Neumann, 1928)

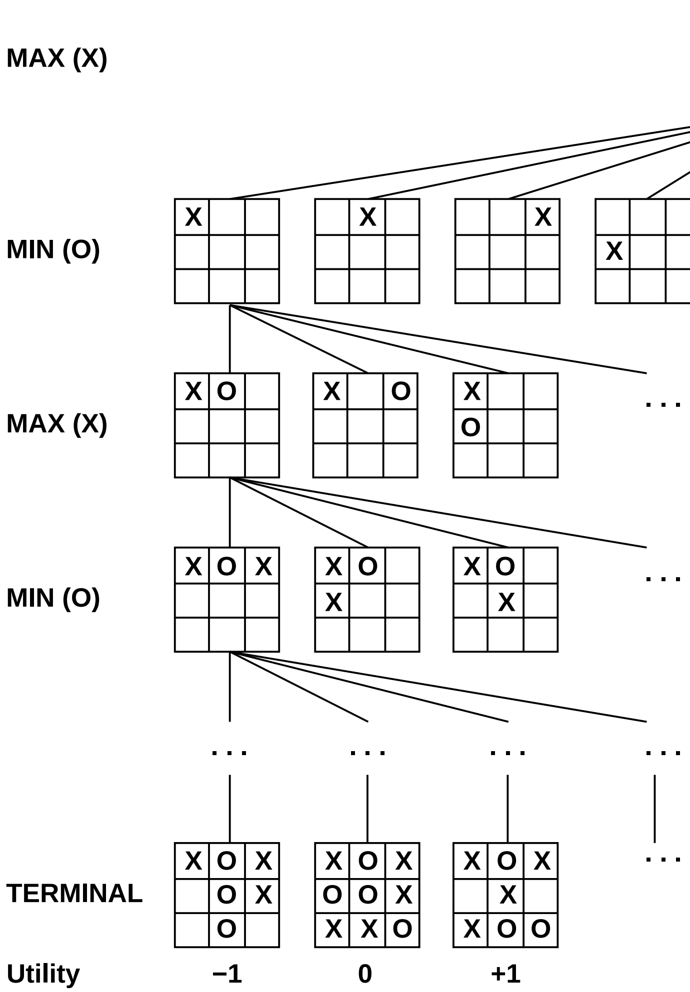
Minimax theorem: Let G be a two-person finite zero-sum game with players Max and Min. Then there are strategies s^* and t^*, and a number V_G called G's *minimax value*, such that

- If Min uses t^*, Max's expected utility is $\leq V_G$, i.e., $\max_s U(s, t^*) = V_G$
- If Max uses s^*, Max's expected utility is $\geq V_G$, i.e., $\min_t U(s^*, t) = V_G$

Corollary 1: $U(s^*, t^*) = V_G$.

Corollary 2: If G is a perfect-information game,
then there are *pure* strategies s^* and t^* that satisfy the theorem.

Game trees

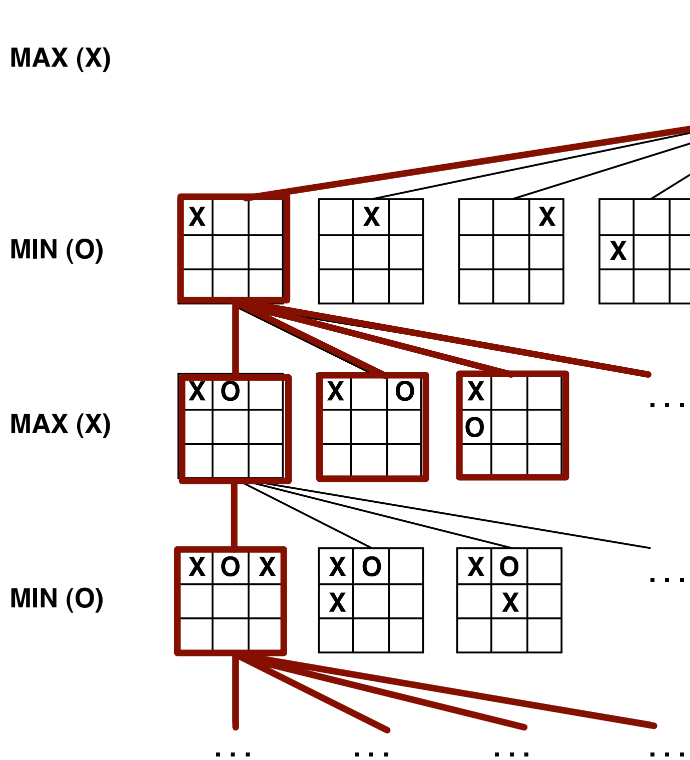


The name *game tree* comes from AI. Mathematical game theorists call this the *extensive form* of a game.

Root node ⇔ the initial state

Children of a node ⇔ the states a player can move to

Strategies on game trees



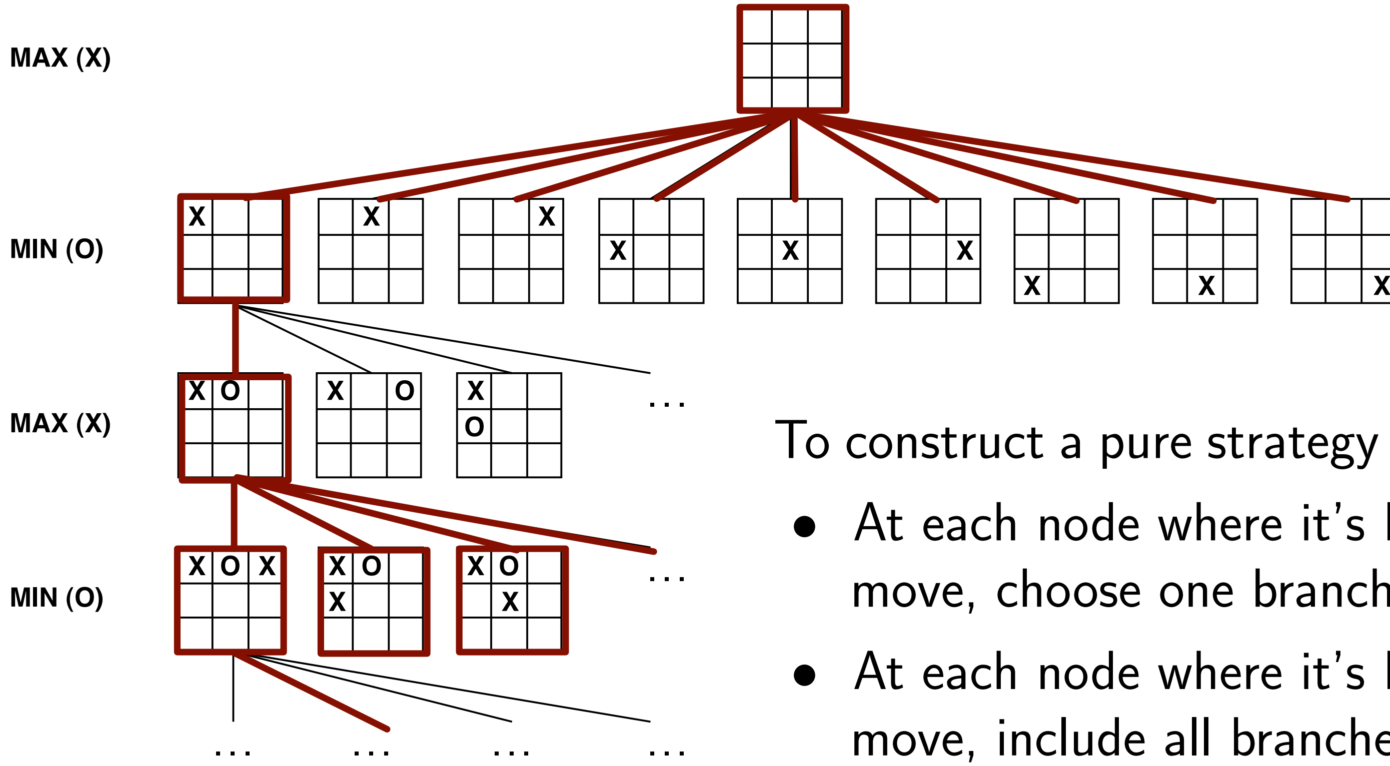
To construct a pure strategy for **Max**:

- At each node where it's Max's move, choose one branch
- At each node where it's Min's move, include all branches

Let $b =$ the *branching factor* (max. number of children of any node)
$h =$ the tree's *height* (max. depth of any node)

The number of pure strategies for Max $\leq b^{\lceil h/2 \rceil}$,
with equality if every node of height $< h$ node has b children

Strategies on game trees



To construct a pure strategy for **Min**:

- At each node where it's Min's move, choose one branch
- At each node where it's Max's move, include all branches

The number of pure strategies for Min $\leq b^{\lceil h/2 \rceil}$
with equality if every node of height $< h$ node has b children

Finding the best strategy

Brute-force way to find Max's and Min's best strategies:

Construct the sets S and T of all of Max's and Min's pure strategies, then choose

$$s^* = \arg\max_{s \in S} \min_{t \in T} U_{\mathsf{Max}}(s,t)$$

$$t^* = \arg\min_{t \in T} \max_{s \in S} U_{\mathsf{Max}}(s,t)$$

Complexity analysis:

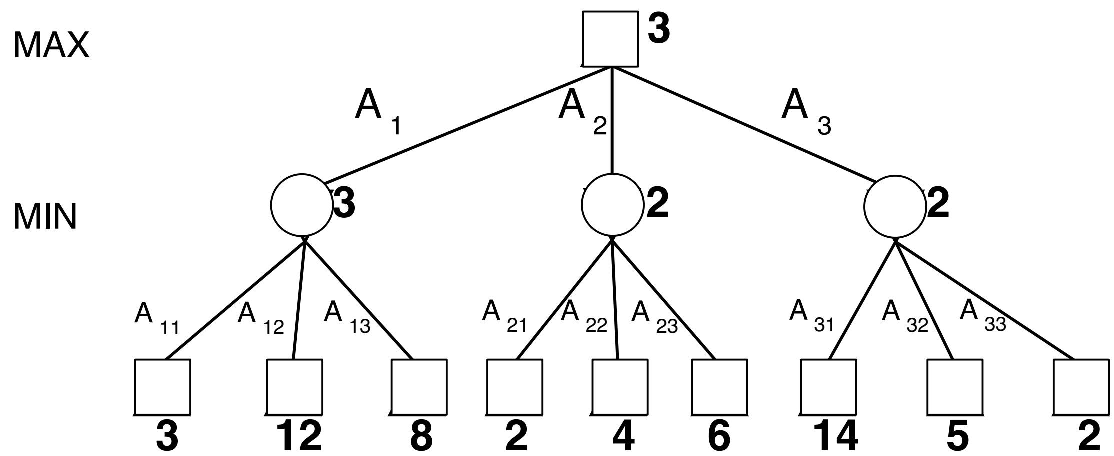
- Need to construct and store $O(b^{h/2} + b^{h/2}) = O(b^{h/2})$ strategies
- Each strategy is a tree that has $O(b^{h/2})$ nodes
- Thus space complexity is $O(b^{h/2} b^{h/2}) = O(b^h)$

- Time complexity is slightly worse

But there's an easier way to find the strategies

Minimax Algorithm

Compute a game's minimax value recursively from the minimax values of its subgames:



function MINIMAX(s) **returns** a utility value
  if s is a terminal state **then return** Max's payoff at s
  else if it is Max's move in s **then**
    return $\max\{\text{MINIMAX}(\text{result}(a, s)) : a$ is applicable to $s\}$
  else return $\min\{\text{MINIMAX}(\text{result}(a, s)) : a$ is applicable to $s\}$

To get the next action, return *argmax* and *argmin* instead of *max* and *min*

Properties of the minimax algorithm

Is it sound? I.e., when it returns answers, are they correct?

Properties of the minimax algorithm

Is it sound? I.e., when it returns answers, are they correct?
Yes (can prove this by induction)

Is it complete? I.e., does it always return an answer when one exists?

Properties of the minimax algorithm

Is it sound? I.e., when it returns answers, are they correct?
Yes (can prove this by induction)

Is it complete? I.e., does it always return an answer when one exists?
Yes on **finite** trees (e.g., chess has specific rules for this).

Space complexity?

Properties of the minimax algorithm

Is it sound? I.e., when it returns answers, are they correct?
Yes (can prove this by induction)

Is it complete? I.e., does it always return an answer when one exists?
Yes on **finite** trees (e.g., chess has specific rules for this).

Space complexity? $O(bh)$, where b and h are as defined earlier

Time complexity?

Properties of the minimax algorithm

Is it sound? I.e., when it returns answers, are they correct?
Yes (can prove this by induction)

Is it complete? I.e., does it always return an answer when one exists?
Yes on **finite** trees (e.g., chess has specific rules for this).

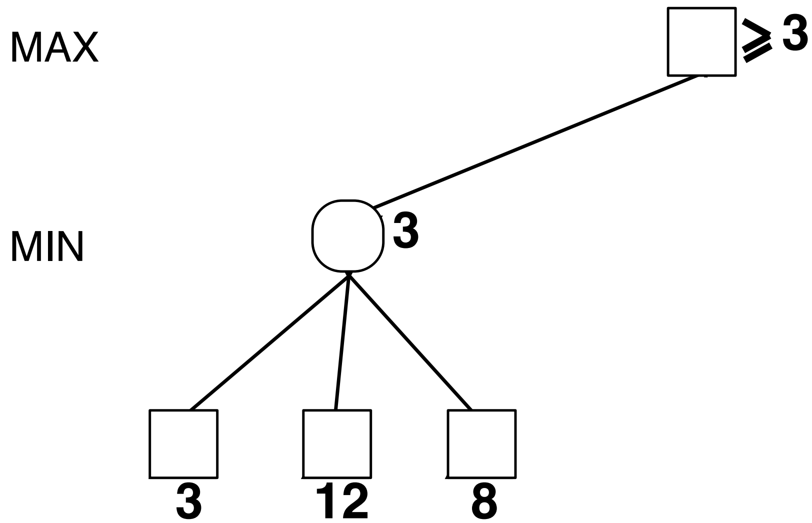
Space complexity? $O(bh)$, where b and h are as defined earlier

Time complexity? $O(b^h)$

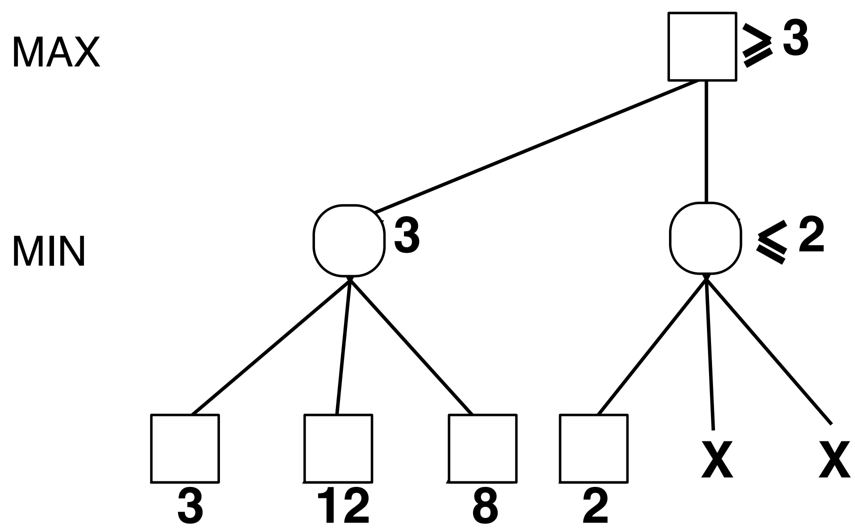
For chess, $b \approx 35$, $h \approx 100$ for "reasonable" games
$35^{100} \approx 10^{135}$ nodes
This is about 10^{55} times the number of particles in the universe (about 10^{87})
$\Rightarrow$ no way to examine every node!

But do we really need to examine every node?

Pruning example 1



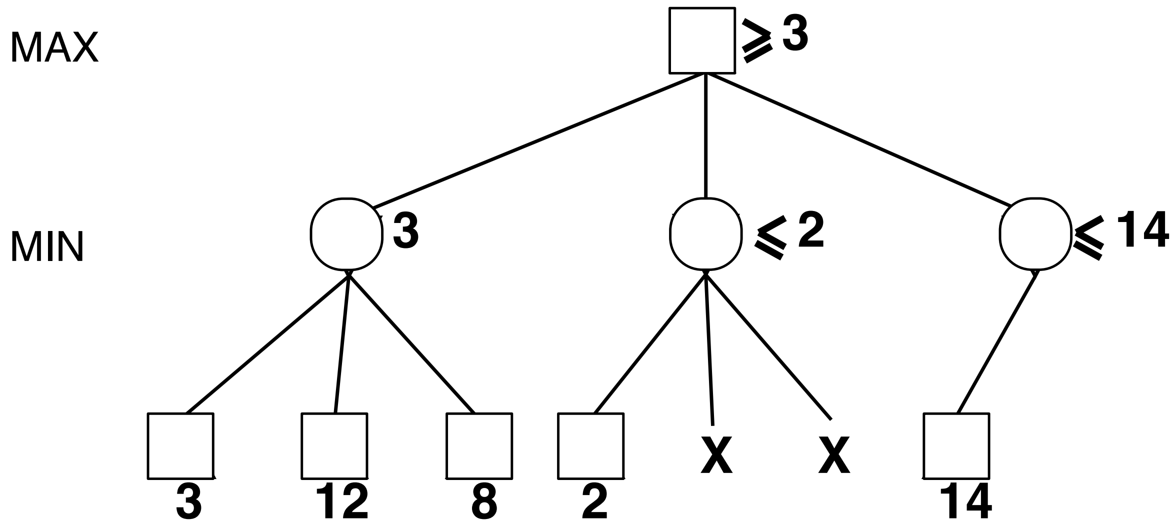
Pruning example 1



Max will never move to this node, because Max can do better by moving to the first one

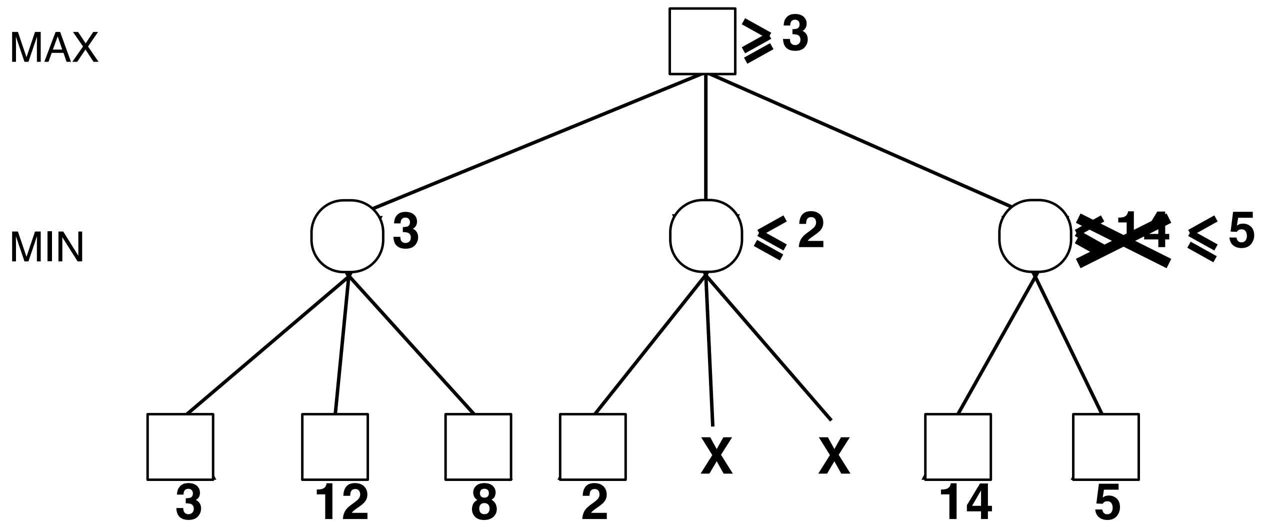
Thus we don't need to figure out this node's minimax value

Pruning example 1



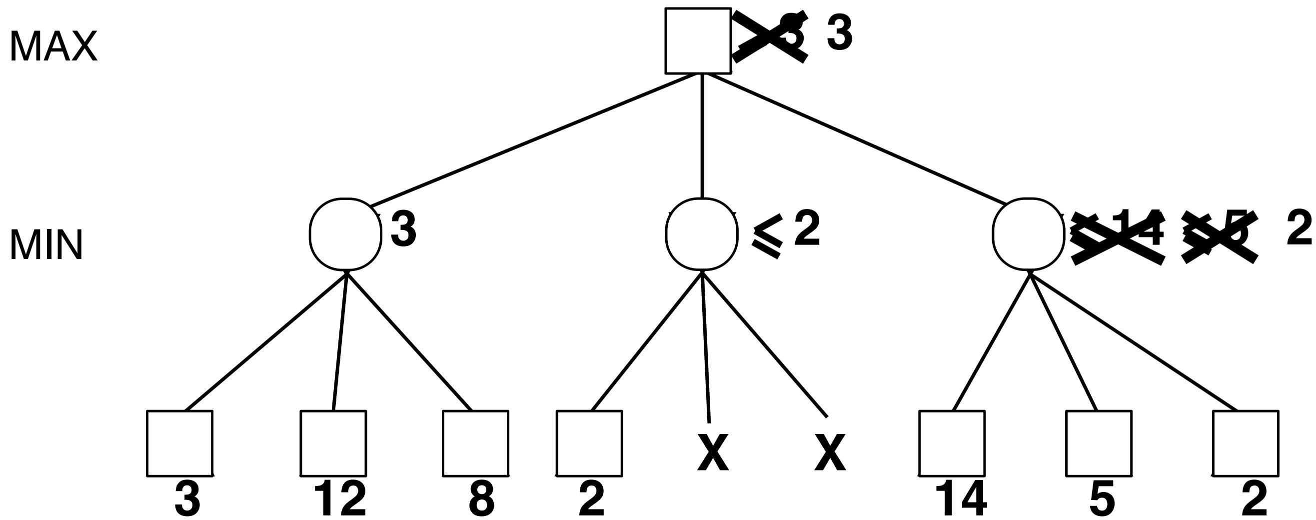
This node might be better than the first one

Pruning example 1



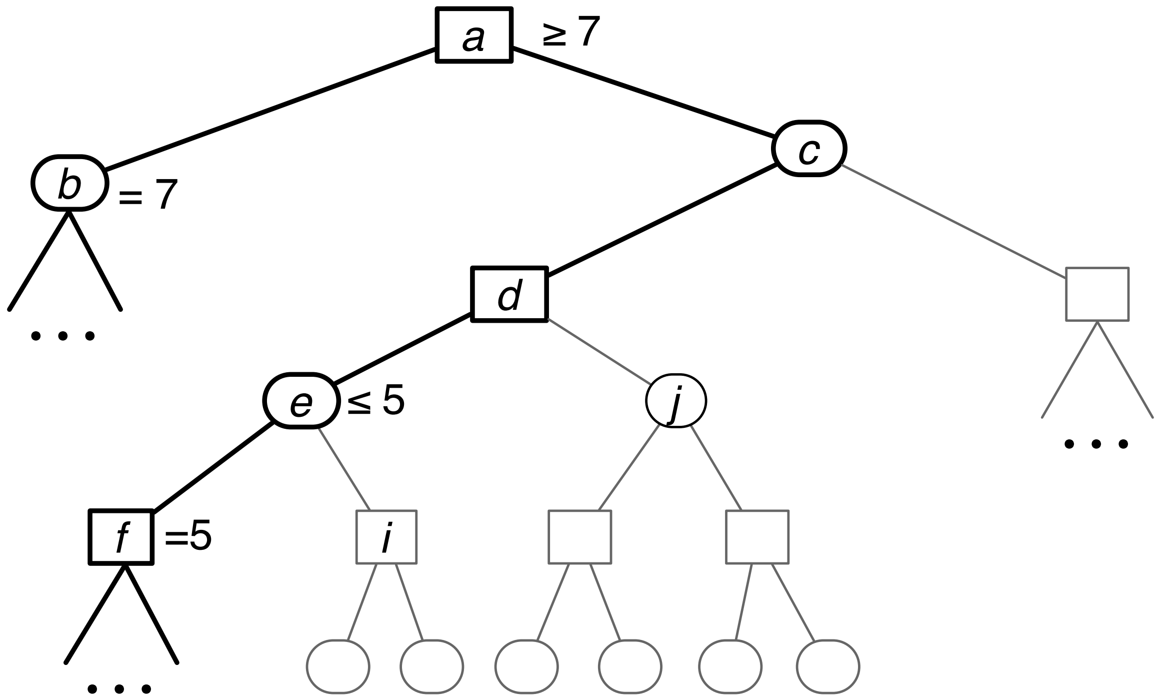
It still might be better than the first one

Pruning example 1



No, it isn't

Pruning example 2



Same idea works farther down in the tree

Max won't move to e, because Max can do better by going to b
Don't need e's exact value, because it won't change minimax(a)
So stop searching below e

Alpha-beta pruning

Start a minimax search at node c

Let $\alpha =$ biggest lower bound on any ancestor of f

$\alpha = \max(-2, 4, 0) = 4$ in the example

If the game reaches f, Max will get utility ≤ 3

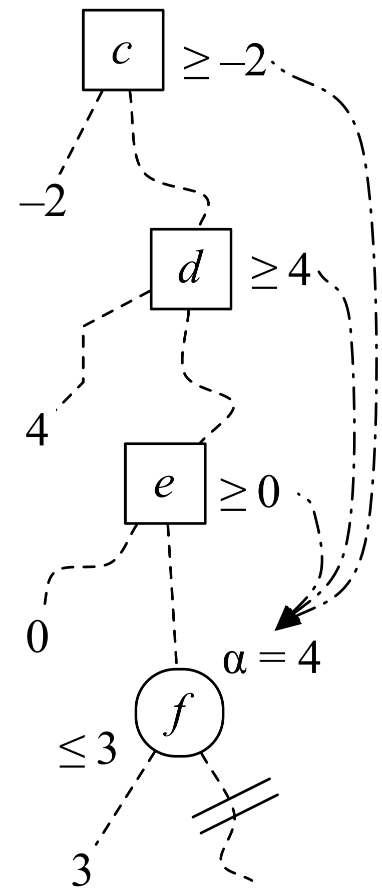
To reach f, the game must go through d

But if the game reaches d, Max can get utility ≥ 4 by moving off of the path to f

So the game will never reach f

We can stop trying to compute $u^*(f)$, because it can't affect $u^*(c)$

This is called an *alpha cutoff*



Alpha-beta pruning

Start a minimax search at node a

Let $\beta =$ smallest upper bound on any ancestor of d

$\beta = \min(5, -2, 3) = -2$ in the example

If the game reaches d, Max will get utility ≥ 0

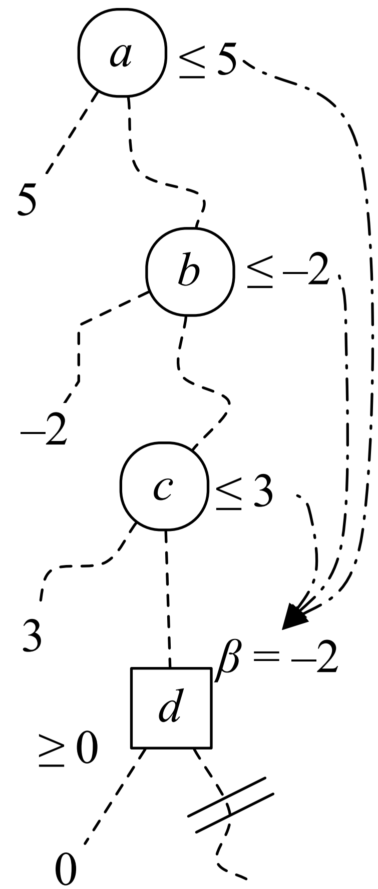
To reach d, the game must go through b

But if the game reaches b, Min can make Max's utility ≤ -2 by moving off of the path to d

So the game will never reach d

We can stop trying to compute $u^*(d)$, because it can't affect $u^*(a)$

This is called a *beta cutoff*

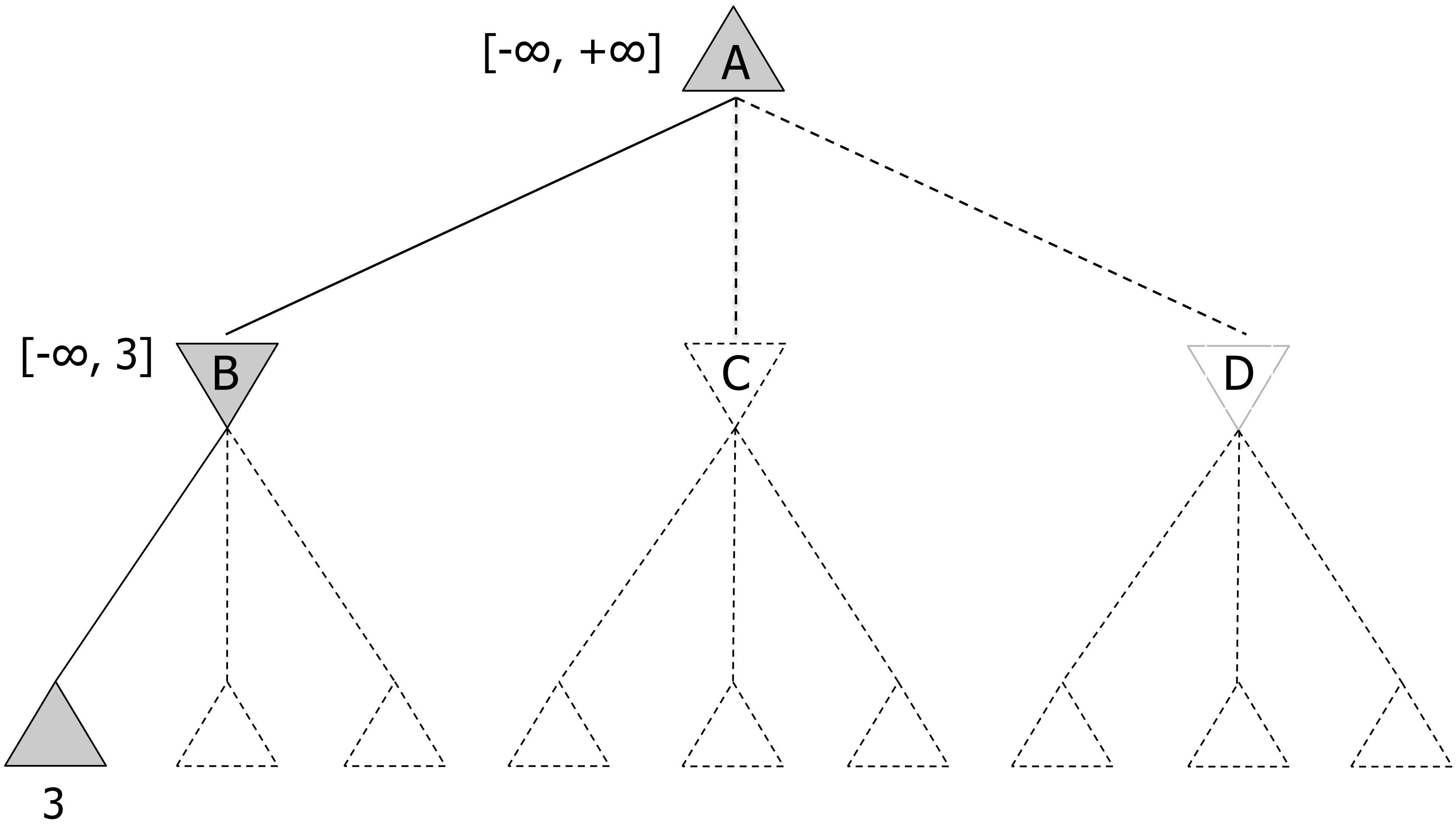


The alpha-beta algorithm

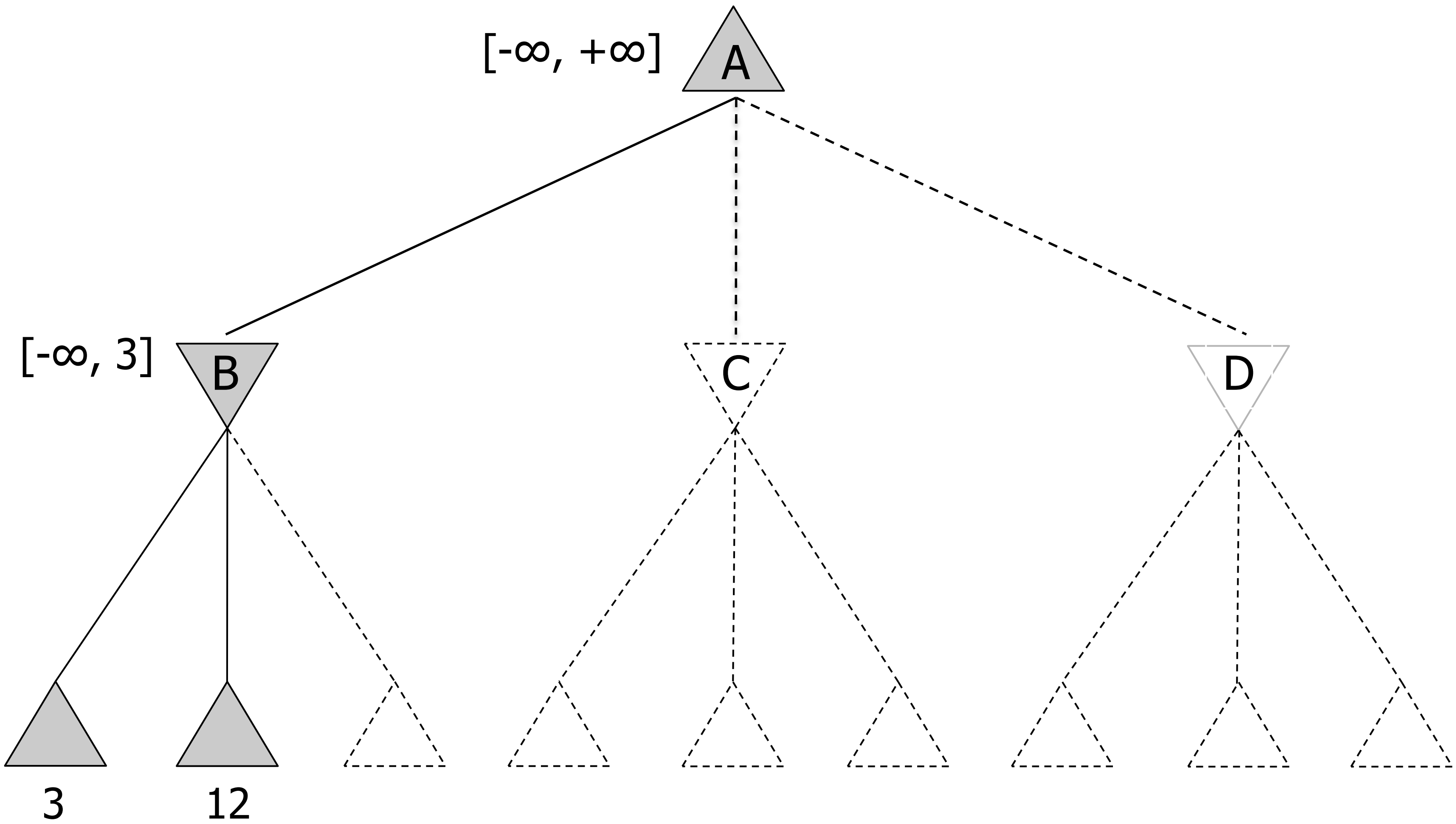
function ALPHA-BETA(s, α, β) **returns** a utility value
  inputs: s, current state in game
    α, the value of the best alternative for MAX along the path to s
    β, the value of the best alternative for MIN along the path to s

  if s is a terminal state **then return** Max's payoff at s
  else if it is Max's move at s **then**
    $v \leftarrow -\infty$
    for every action a applicable to s **do**
      $v \leftarrow \max(v, \text{ALPHA-BETA}(\text{result}(a, s), \alpha, \beta))$
      if $v \geq \beta$ **then return** v
      $\alpha \leftarrow \max(\alpha, v)$
  else
    $v \leftarrow \infty$
    for every action a applicable to s **do**
      $v \leftarrow \min(v, \text{ALPHA-BETA}(\text{result}(a, s), \alpha, \beta))$
      if $v \leq \alpha$ **then return** v
      $\beta \leftarrow \min(\beta, v)$
  return v

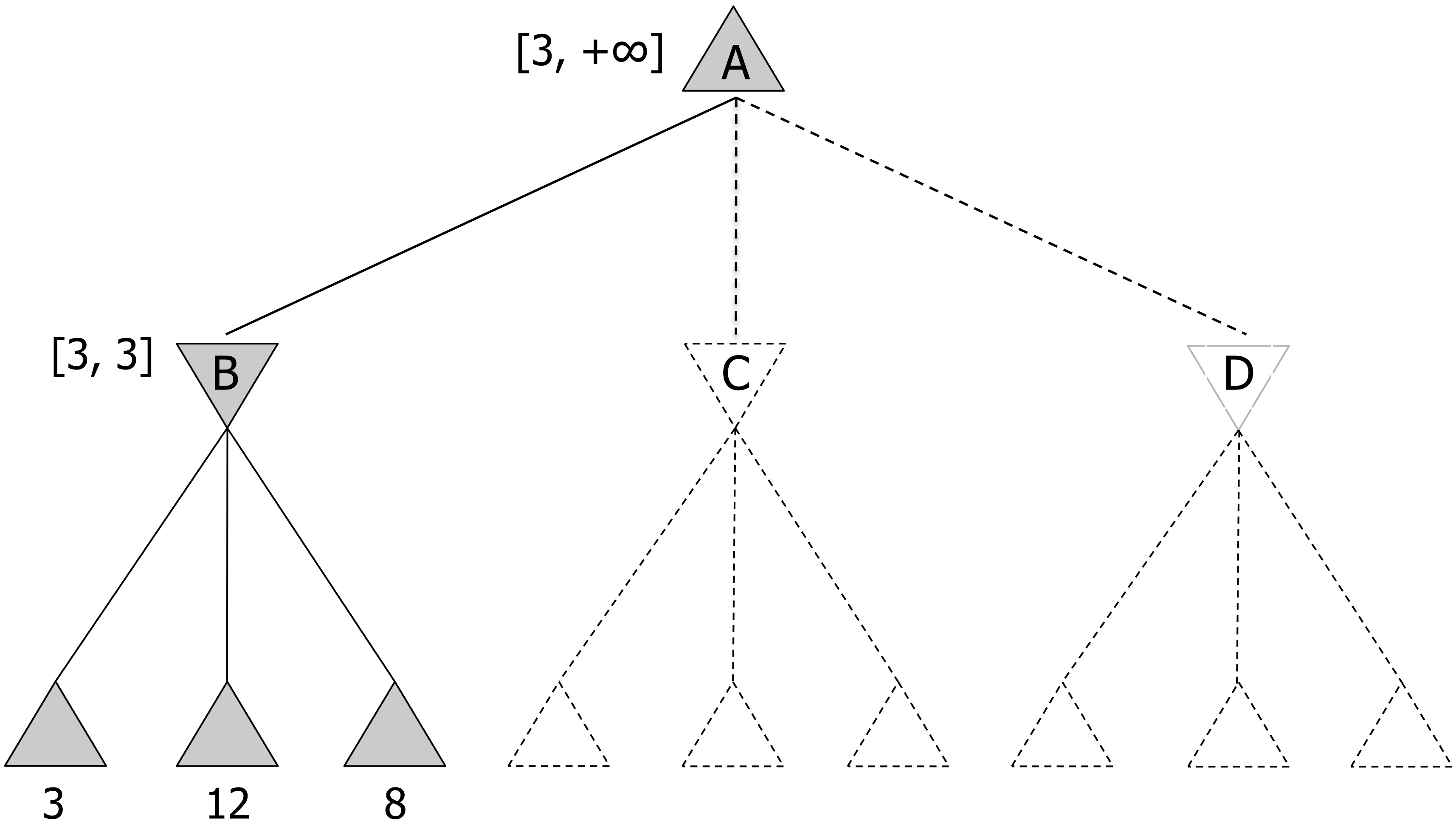
Example: alpha-beta pruning



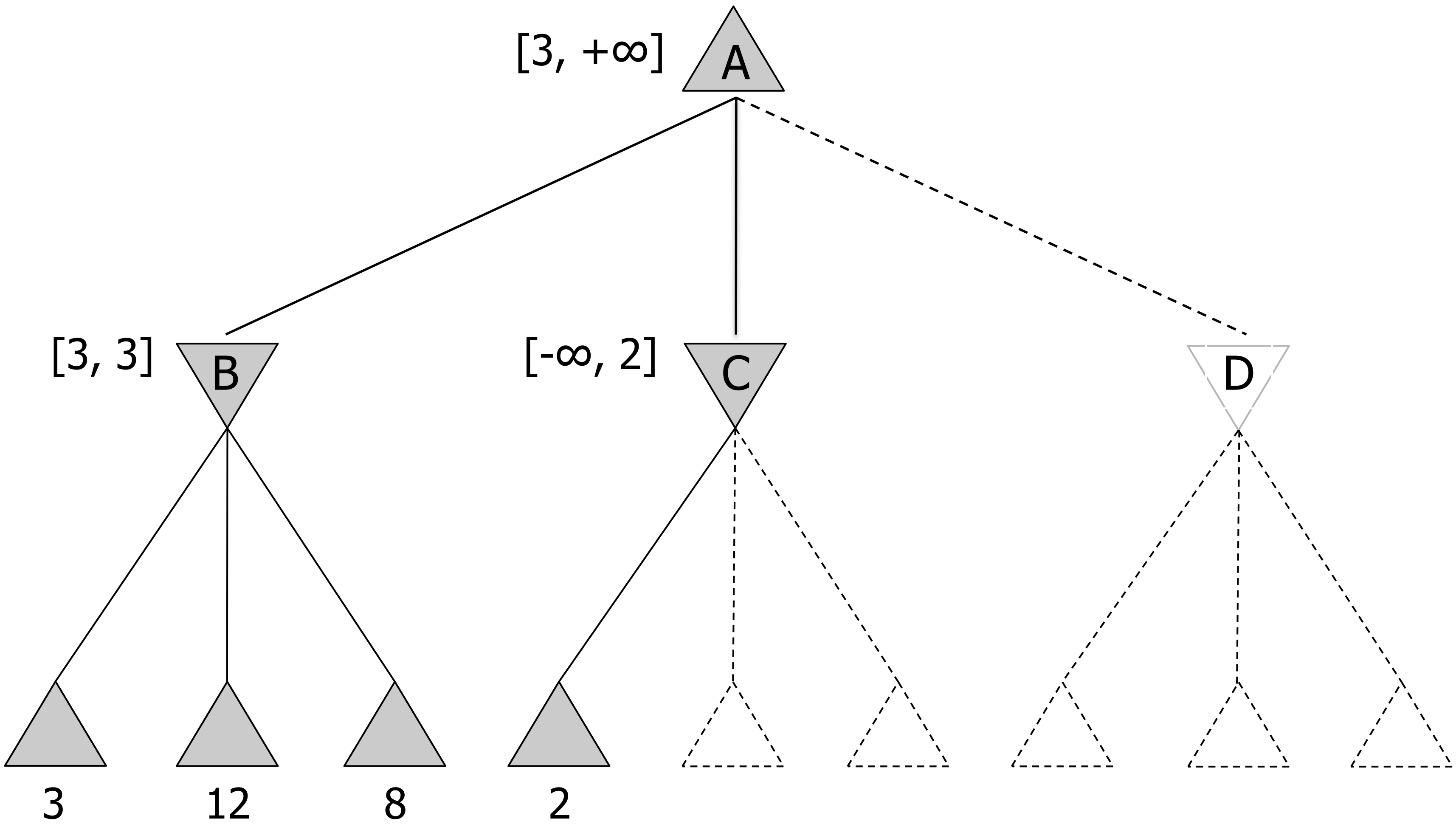
Example: alpha-beta pruning



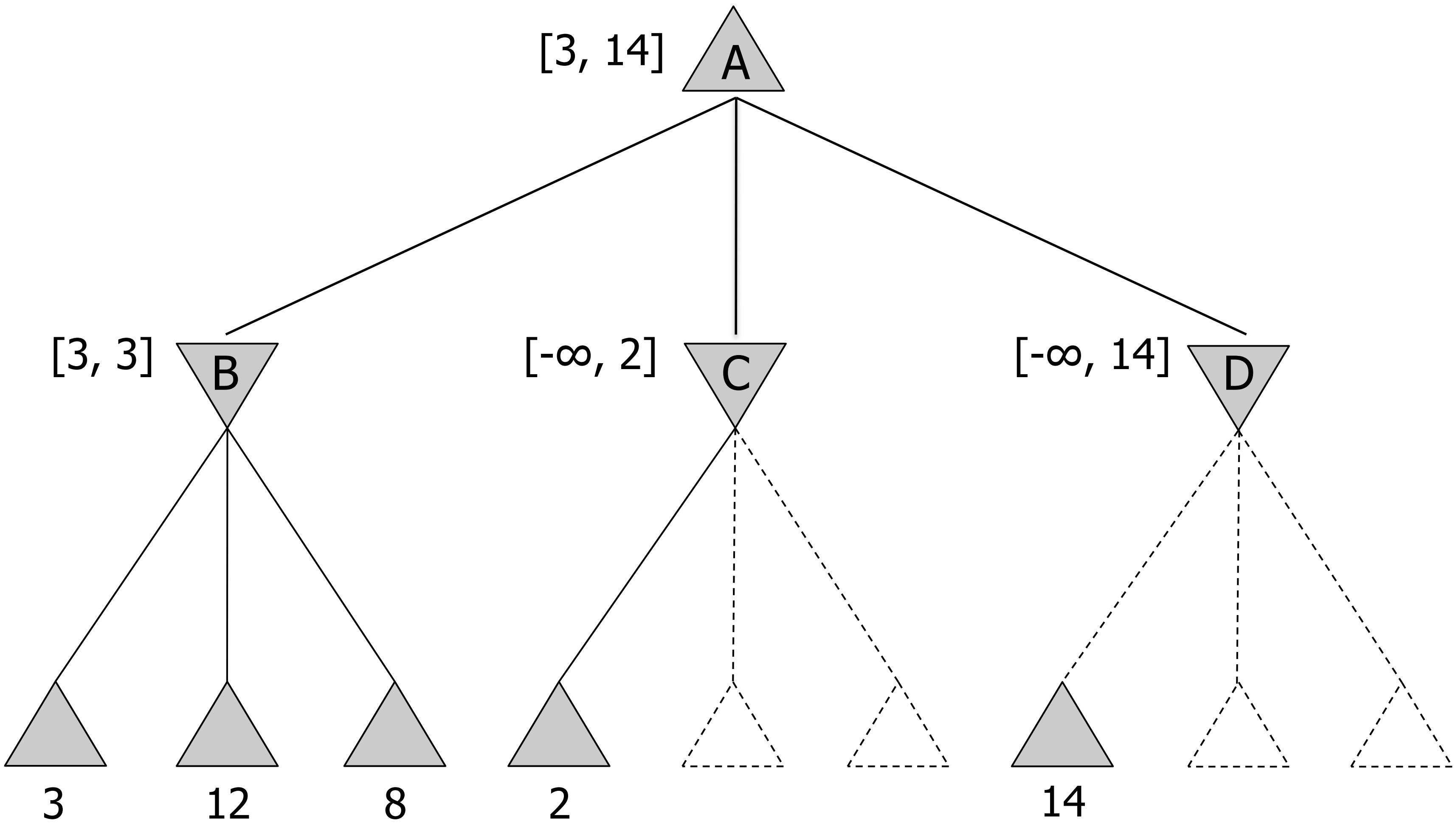
Example: alpha-beta pruning



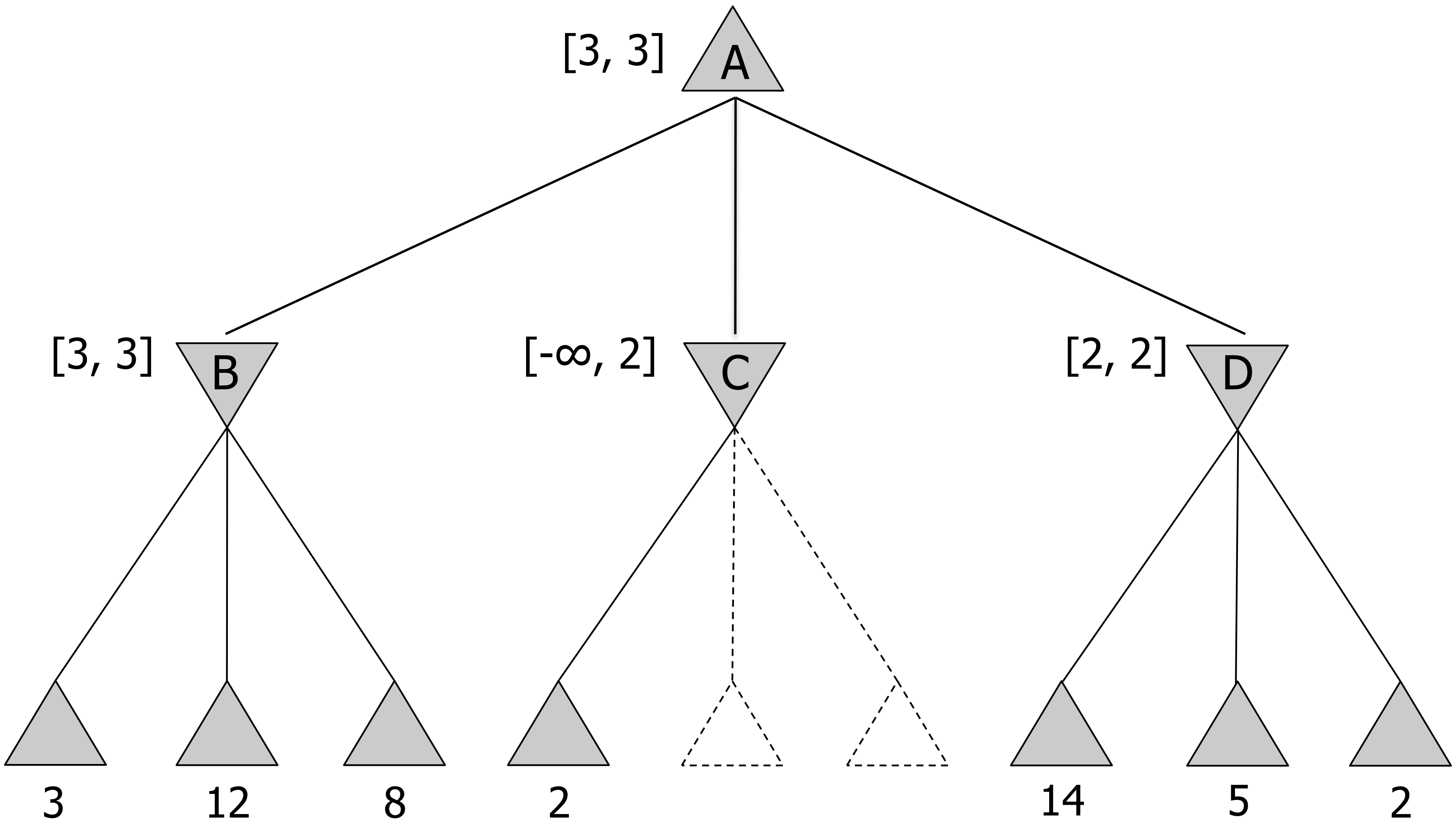
Example: alpha-beta pruning



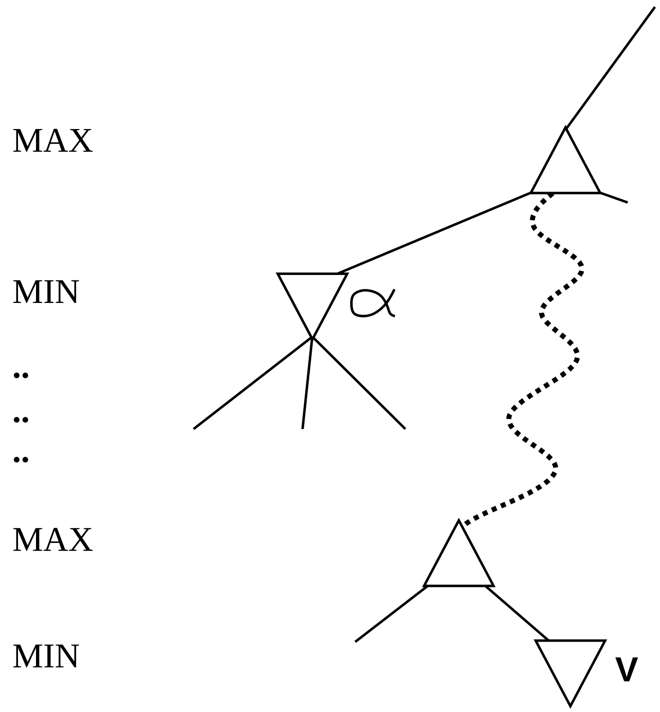
Example: alpha-beta pruning



Example: alpha-beta pruning



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

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!

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

α-β with a bound d on the search depth

function Alpha-Beta(s, α, β, d) **returns** a utility value
  inputs: s, α, β, same as before
    d, an upper bound on the search depth

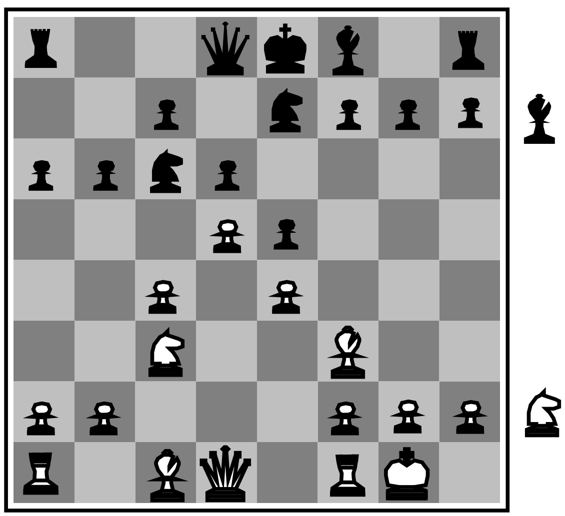
  if s is a terminal state **then return** Max's payoff at s
  else if $d = 0$ **then return** Eval(s)
  else if it is Max's move at s **then**
    $v \leftarrow -\infty$
    for every action a applicable to s **do**
      $v \leftarrow \max(v, \text{Alpha-Beta}(\text{result}(a, s), \alpha, \beta, d-1))$
      if $v \geq \beta$ **then return** v
      $\alpha \leftarrow \max(\alpha, v)$
  else
    $v \leftarrow \infty$
    for every action a applicable to s **do**
      $v \leftarrow \min(v, \text{Alpha-Beta}(\text{result}(a, s), \alpha, \beta, d-1))$
      if $v \leq \alpha$ **then return** v
      $\beta \leftarrow \min(\alpha, v)$
  return v

Evaluation functions

$\textsc{Eval}(s)$ is supposed to return an approximation of s's minimax value

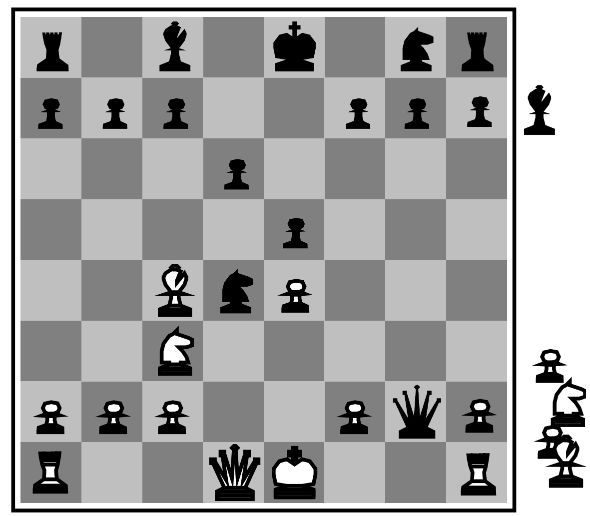
$\textsc{Eval}$ is often a weighted sum of *features*

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



Black to move

White slightly better



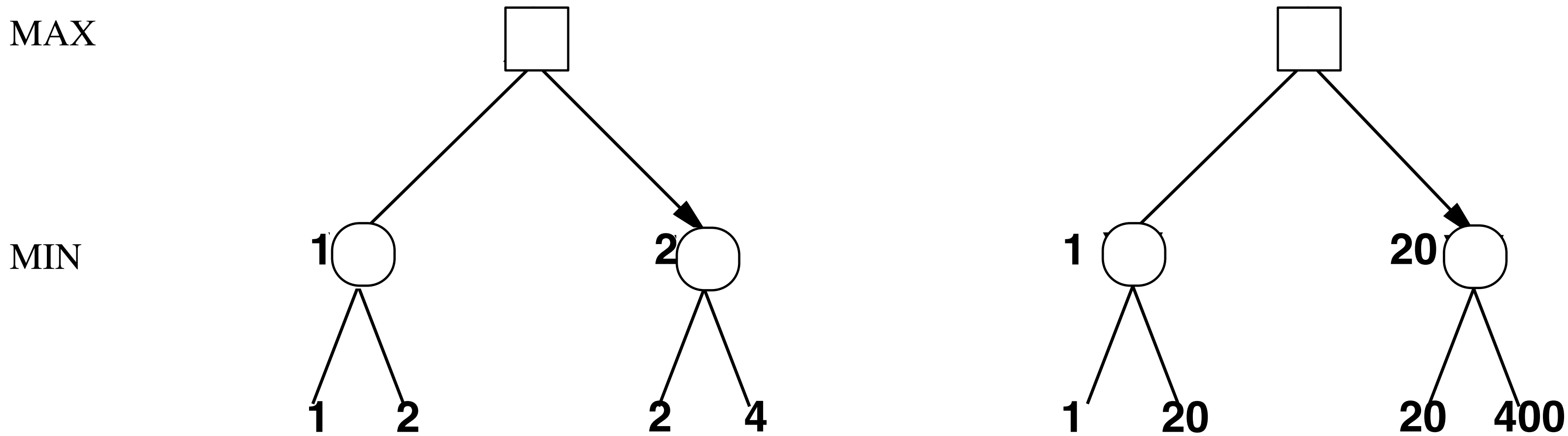
White to move

Black winning

E.g.,

$$\begin{aligned}
&\ 1(\text{number of white pawns} - \text{number of black pawns}) \\
&+ 3(\text{number of white knights} - \text{number of black knights}) \\
&+ \ldots
\end{aligned}$$

Exact values for EVAL don't matter



Behavior is preserved under any **monotonic** transformation of EVAL

Only the order matters:

In deterministic games, payoff acts as an *ordinal utility* function

Discussion

Deeper lookahead (i.e., larger depth bound d) usually gives better decisions

Exceptions do exist

Main result in my PhD dissertation (30 years ago!):

"pathological" games in which deeper lookahead gives worse decisions

But such games hardly ever occur in practice

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

$\Rightarrow 10^6 \approx 35^{8/2}$ nodes per move

$\Rightarrow \alpha$-β reaches depth 8 $\Rightarrow$ pretty good chess program

Some modifications that can improve the accuracy or computation time:

node ordering (see next slide)

quiescence search

biasing

transposition tables

thinking on the opponent's time

...

Game-tree search in practice

Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994.

Checkers was *solved* in April 2007: from the standard starting position, both players can guarantee a draw with perfect play. This took 10^{14} calculations over 18 years. Checkers has a search space of size 5×10^{20}.

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: until recently, human champions didn't compete against computers because the computers were too **bad.** But that has changed . . .

Game-tree search in the game of go

A game tree's size grows exponentially with both its depth and its branching factor

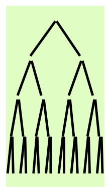
Go is much too big for a normal game-tree search:

branching factor = about 200
game length = about 250 to 300 moves
number of paths in the game tree = 10^{525} to 10^{620}

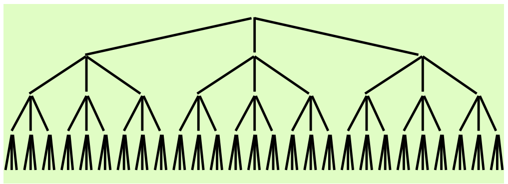
For comparison:

Number of atoms in universe = about 10^{80}
Number of particles in universe = about 10^{87}

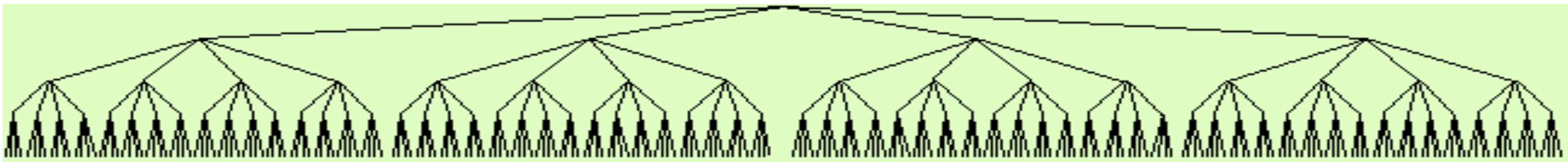
b =2



b =3



b =4



Game-tree search in the game of go

During the past couple years, go programs have gotten much better

Main reason: **Monte Carlo roll-outs**

Basic idea: do a minimax search of a randomly selected subtree

At each node that the algorithm visits,

- ◊ It randomly selects some of the children
  There are some heuristics for deciding how many
- ◊ Calls itself recursively on these, ignores the others

Forward pruning in chess

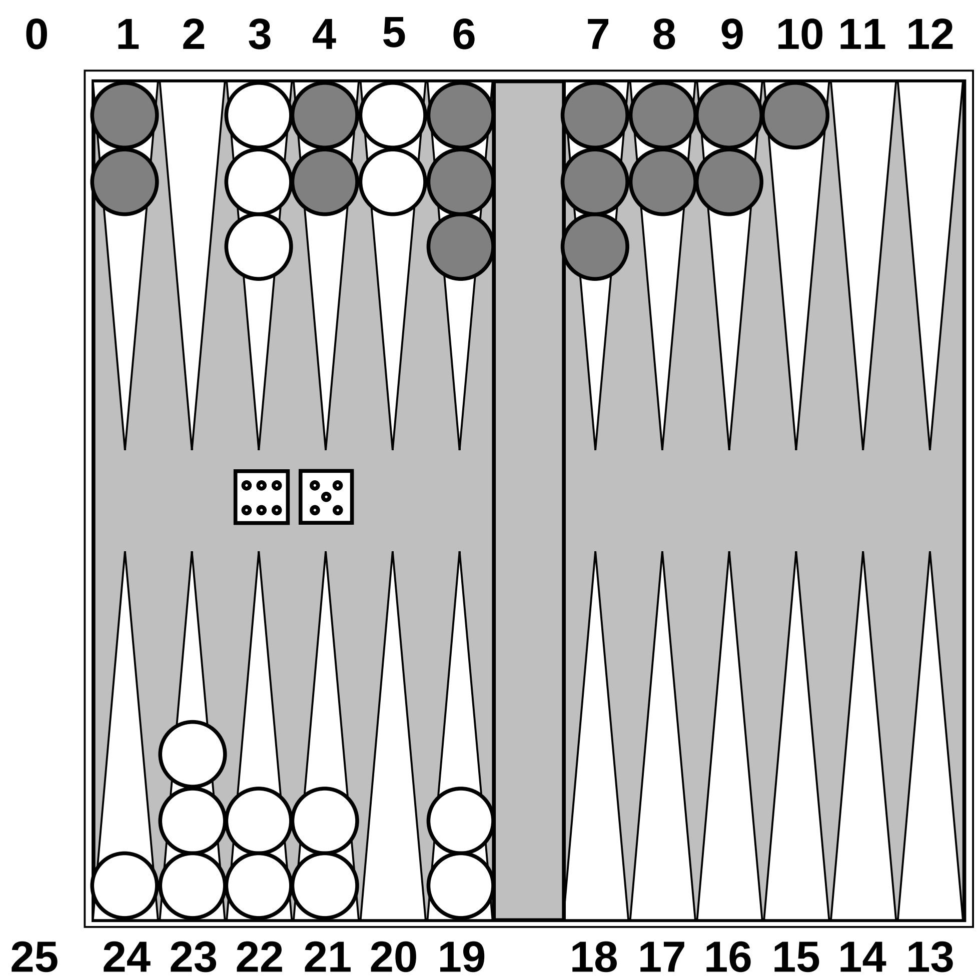
Back in the 1970s, some similar ideas were tried in chess

The approach was called **forward pruning**
Main difference: select the children heuristically rather than randomly
It didn't work as well as brute-force alpha-beta, so people abandoned it

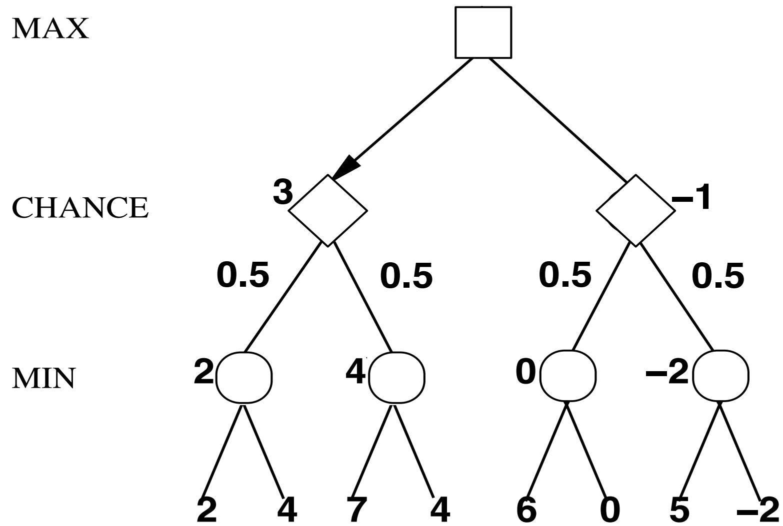
Why does a similar idea work so much better in go?

Perfect-information nondeterministic games

Backgammon: chance is introduced by dice



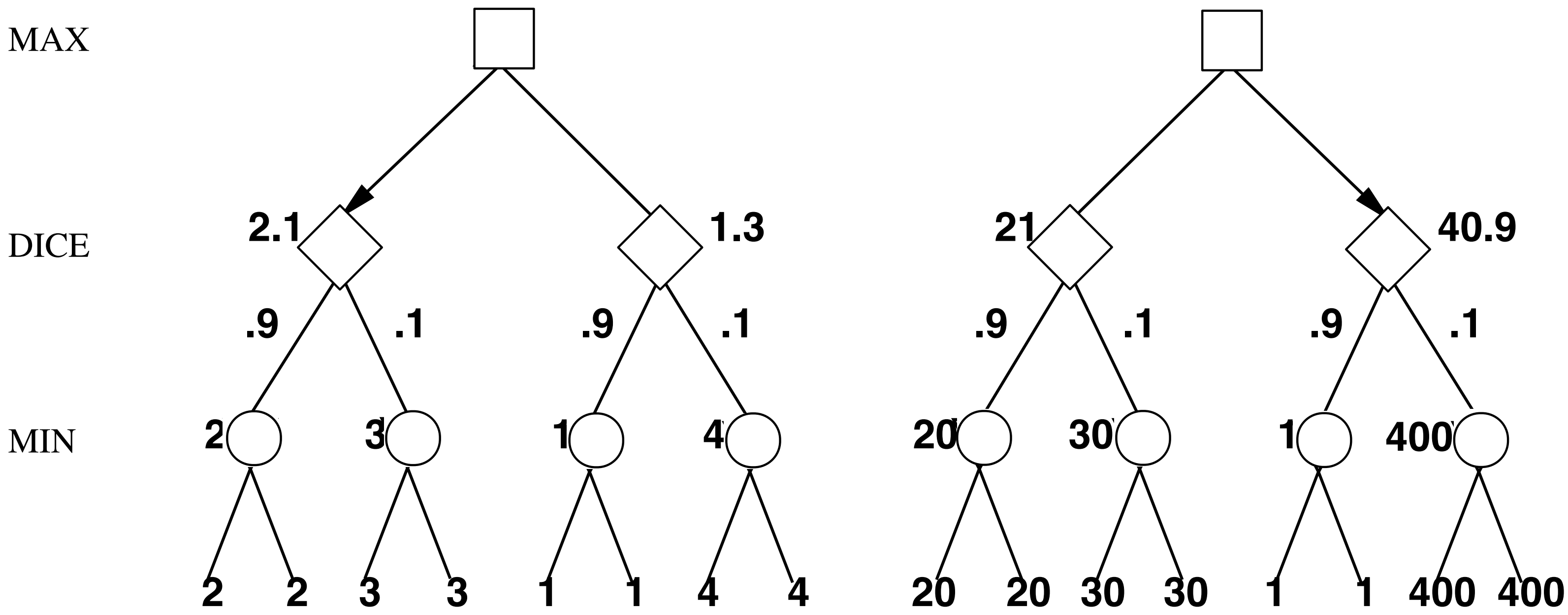
Expectiminimax



function EXPECTIMINIMAX(s) **returns** an expected utility
  if s is a terminal state **then return** Max's payoff at s
  if s is a "chance" node **then**
    return $\sum_{s'} P(s'|s)\textsc{ExpectiMinimax}(s')$
  else if it is Max's move at s **then**
    return $\max\{\textsc{ExpectiMinimax}(\text{result}(a,s)) : a$ is applicable to $s\}$
  else return $\min\{\textsc{ExpectiMinimax}(\text{result}(a,s)) : a$ is applicable to $s\}$

This gives optimal play (i.e., highest expected utility)

With nondeterminism, exact values do matter



At "chance" nodes, we need to compute weighted averages

Behavior is preserved only by positive linear transformations of EVAL
Hence EVAL should be proportional to the expected payoff

Summary

We looked at games that have the following characteristics:

two players
zero sum
perfect information
deterministic
finite

In these games, can do a game-tree search
minimax values, alpha-beta pruning

In sufficiently complicated games, perfection is unattainable
$\Rightarrow$ must approximate: limited search depth, static evaluation function

In games that are even more complicated, further approximation is needed
$\Rightarrow$ Monte Carlo roll-outs

If we add an element of chance (e.g., dice rolls), expectiminimax