Last update: October 2, 2012

Adversarial Search

CMSC 421, Chapter 5

We'll start with a restricted class of games

◇ **Finite**: finitely many players, actions, states

◇ **Perfect information**: Every agent always knows exactly what the current state is, and what the actions will do

- ◇ No simultaneous actions: players move one at a time
- Includes most (but not all) board games
- Excludes most card games and video games

◇ **Deterministic**: no chance elements

- Includes chess, checkers, go, tic-tac-toe, mancala (awari, kalah), Othello (Reversi), Connect-Four, Qubic, Quoridor, . . .
- Excludes backgammon, parcheesi, Monopoly, Yahtzee, Risk, Carcassonne, . . .

◇ **Zero-sum**: $\sum\{\text{the players' payoffs}\} = 0$

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

Terminology

- *Utility*: numeric measure of how much a player likes an outcome of a game
- Usually we'll assume this is the same as the game's payoff
  - When is this assumption correct?
- A *strategy* specifies what action an agent choose in every possible situation
  - *pure* strategy: the choice is always deterministic
  - *mixed* strategy: probability distribution over pure strategies

- Consider a game G between two players (Max and Min)
- Let $U(s, t)$ be Max's *expected utility* if Max's and Min's strategies are s and t
- If G is a zero-sum game, then Min's utility is always $-U(s, t)$
  - Max wants to maximize U and Min wants to minimize it

The Minimax Theorem (von Neumann, 1928)

◇ **Minimax theorem:** If G is a finite, two-player, zero-sum game, 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.,

$$\max_s U(s, t^*) = V_G$$

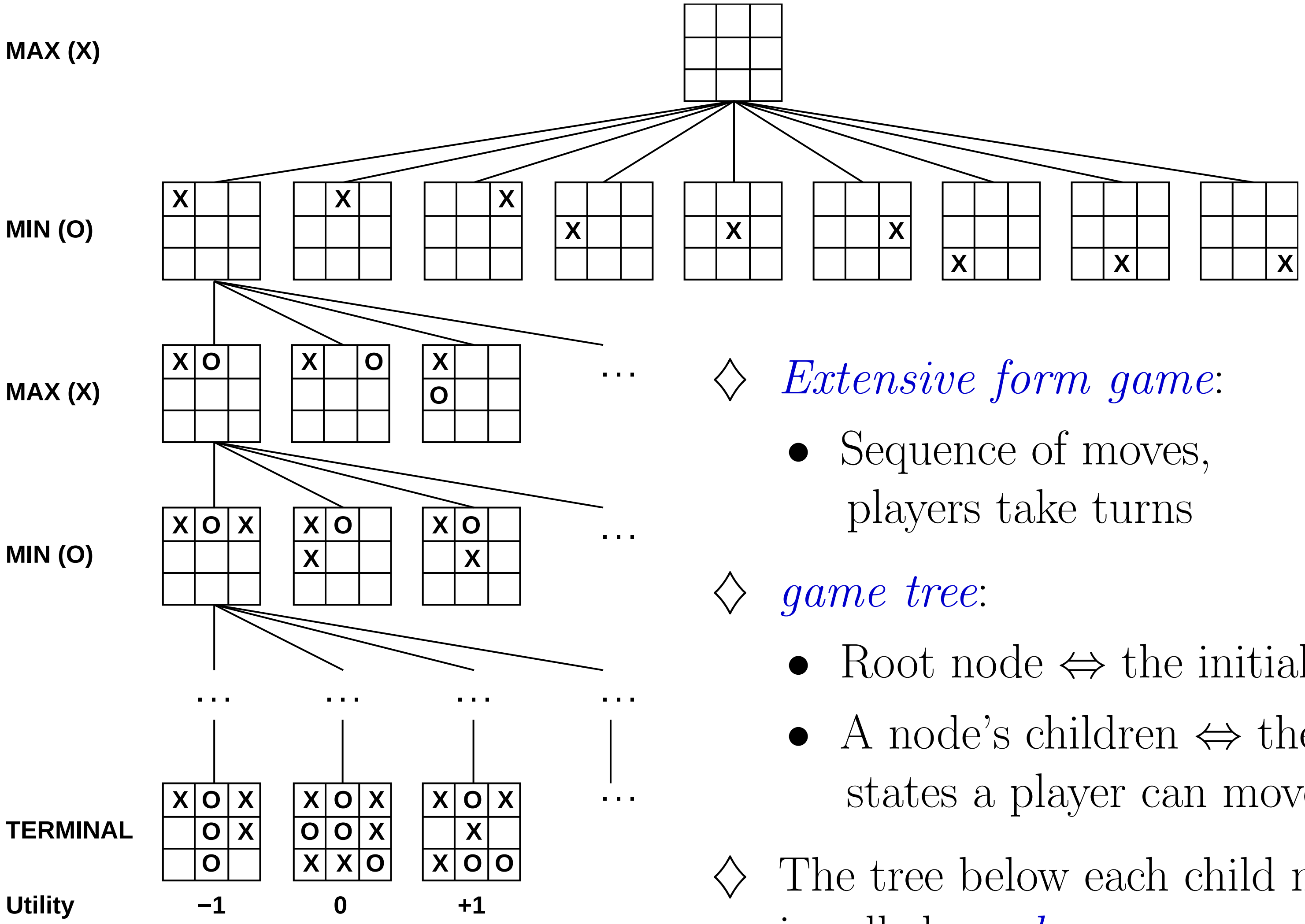
- If Max uses s^*, Max's expected utility is $\geq V_G$, i.e.,

$$\min_t U(s^*, t) = V_G$$

◇ **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



♢ *Extensive form game*:

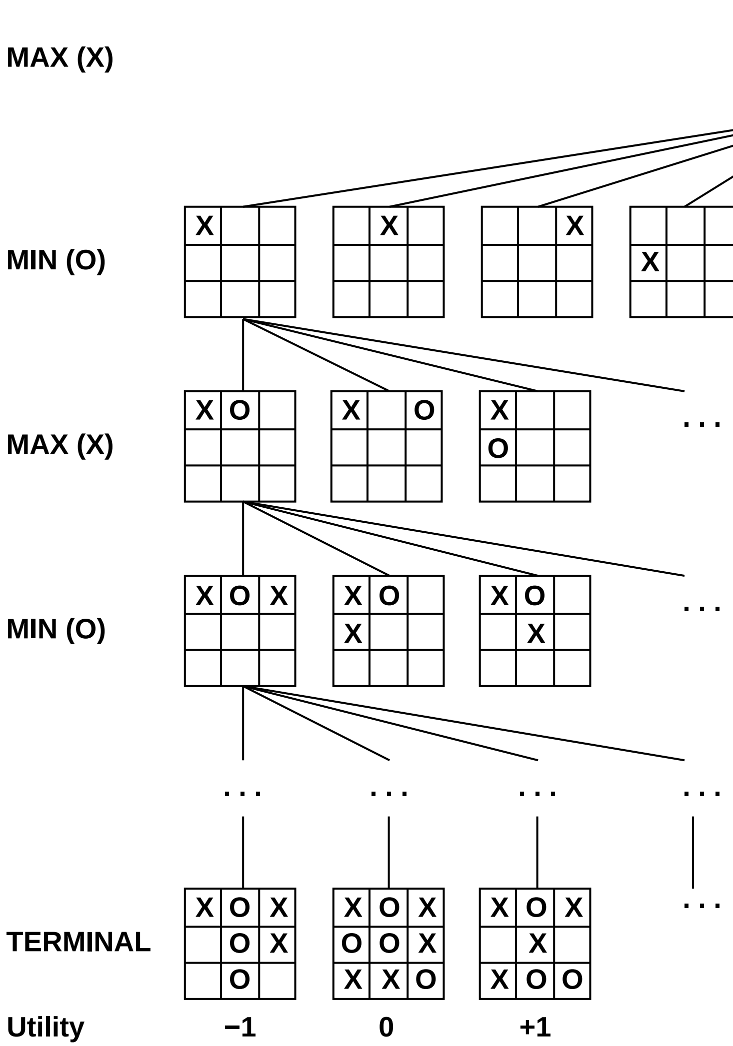
- Sequence of moves, players take turns

♢ *game tree*:

- Root node ⇔ the initial state
- A node's children ⇔ the states a player can move to

♢ The tree below each child node is called a *subgame*

Strategies on game trees



◇ Pure strategy for Max:

- One branch at each of Max's choice nodes
- There are $\Theta(b^h)$ of these

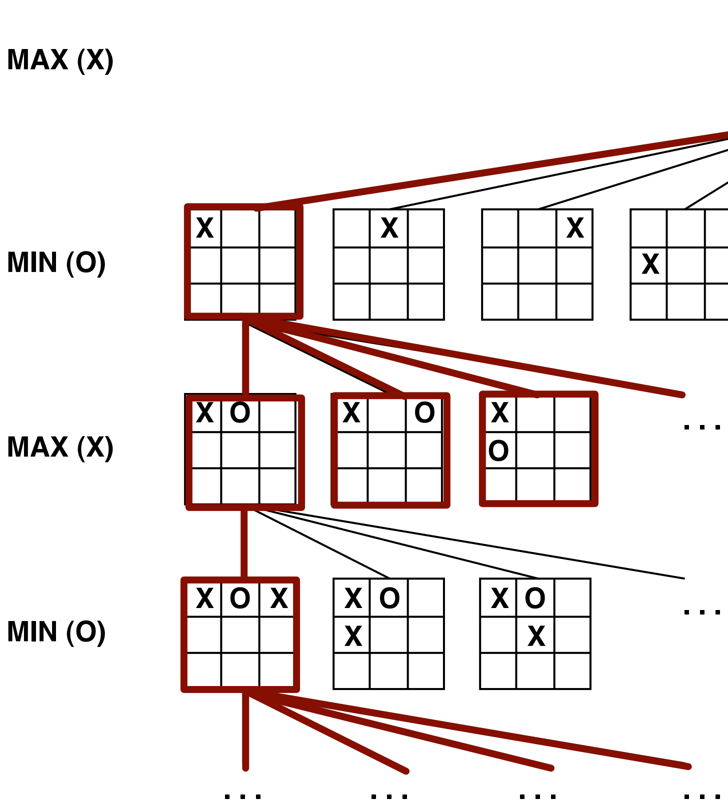
◇ Pure strategy for Min:

- One branch at each of Min's choice nodes
- There are $\Theta(b^h)$ of these

b = the *branching factor* (max. number of children of any node)

h = the tree's *height* (max. depth of any node)

Strategies on game trees



♢ Two strategies are equivalent if they differ only at unreachable nodes

♢ For **Max**:

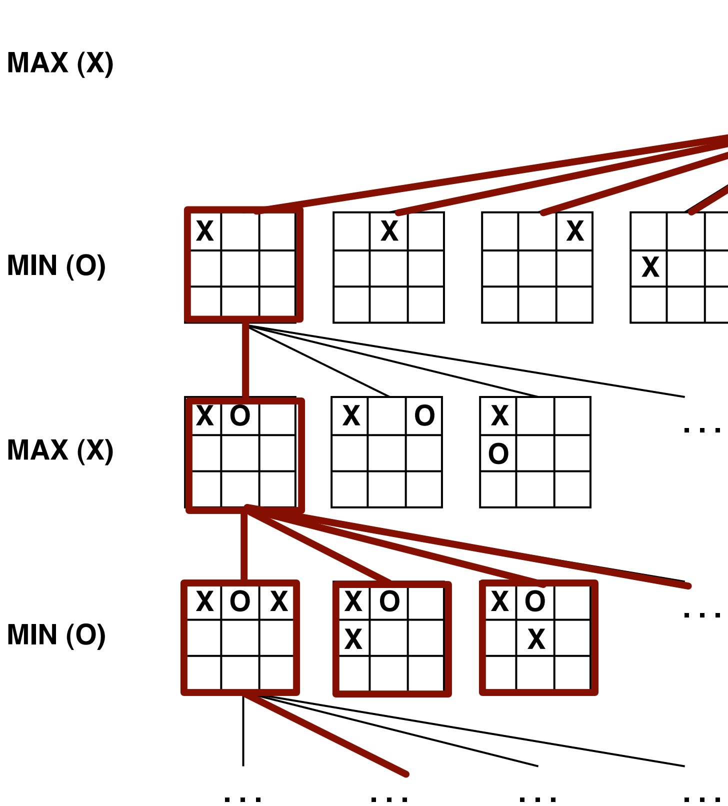
- At each of Max's choice nodes, choose one branch
- At each of Min's choice nodes, include all branches

♢ Number of **non-equivalent** pure strategies for Max is $\Theta(b^{h/2})$

---

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

Strategies on game trees



◇ Two strategies are equivalent if they differ only at unreachable nodes

◇ For **Min**:

- At each of Min's choice nodes, choose one branch
- At each of Min's choice nodes, include all branches

◇ Number of **non-equivalent** pure strategies for Min is $\Theta(b^{h/2})$

---

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

Finding the best strategy

◇ Brute-force approach

- Let S and T be the sets of pure strategies for Max and Min
- Compare every combination, choose the ones that work best:

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

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

◇ Each player has $O(b^h)$ strategies, each strategy has size $O(b^h)$

◇ Number of comparisons is $O(b^{2h})$

- If we keep all strategies in memory, each comparison takes time $O(h)$
  - ◇ $O(hb^{2h})$ time and $O(b^{2h})$ space
- If we generate strategies on the fly, each comparison takes time $O(hb^h)$
  - ◇ $O(hb^{3h})$ time and $O(b^h)$ space

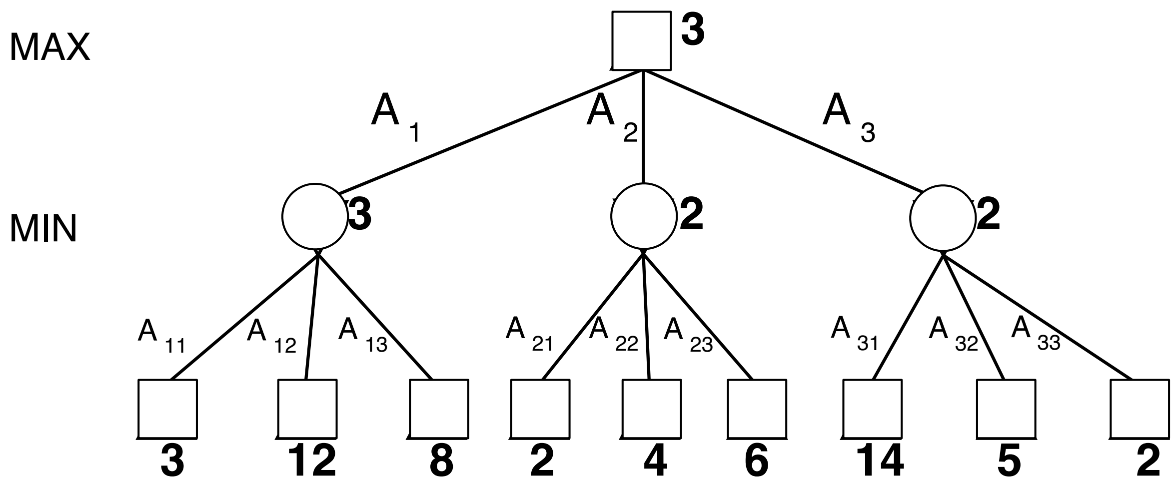
◇ If we only include reachable nodes, replace h with $h/2$ above

◇ But there's an easier way

Minimax Algorithm

◇ Compute minimax value recursively: time $O(b^h)$, space $O(bh)$

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 $\max\{$MINIMAX(result(a, s)) : a is applicable to $s\}$
  else return $\min\{$MINIMAX(result(a, s)) : a is applicable to $s\}$



◇ To get the next move, 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

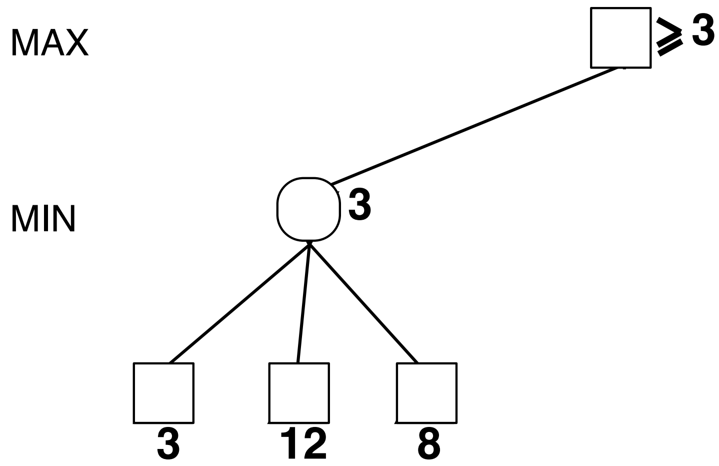
◇ *Time complexity?*

Properties of the minimax algorithm

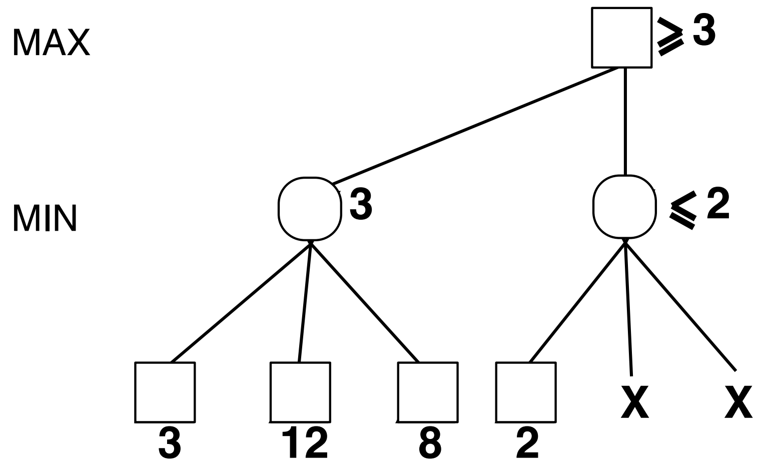
- *Is it sound?* I.e., when it returns answers, are they correct?
  - Yes (can prove this by induction)
- *Is it complete?* I.e., does it always return an answer when one exists?
  - Yes on **finite** trees (e.g., chess has specific rules for this).
- *Space complexity?* $O(bh)$, where b and h are as defined earlier
- *Time complexity?* $O(b^h)$

- For chess, $b \approx 35$, $h \approx 100$ for "reasonable" games
  - $35^{100} \approx 10^{135}$ nodes
- 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



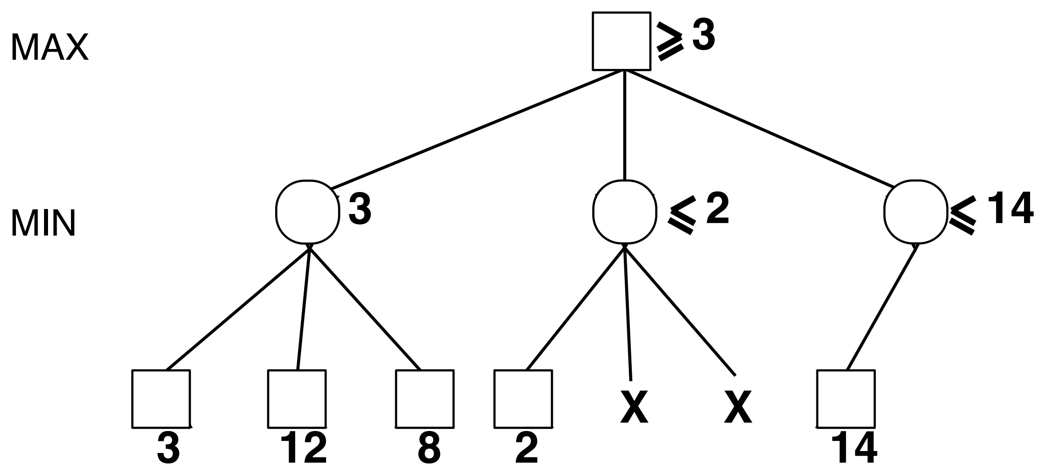
Pruning example 1



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

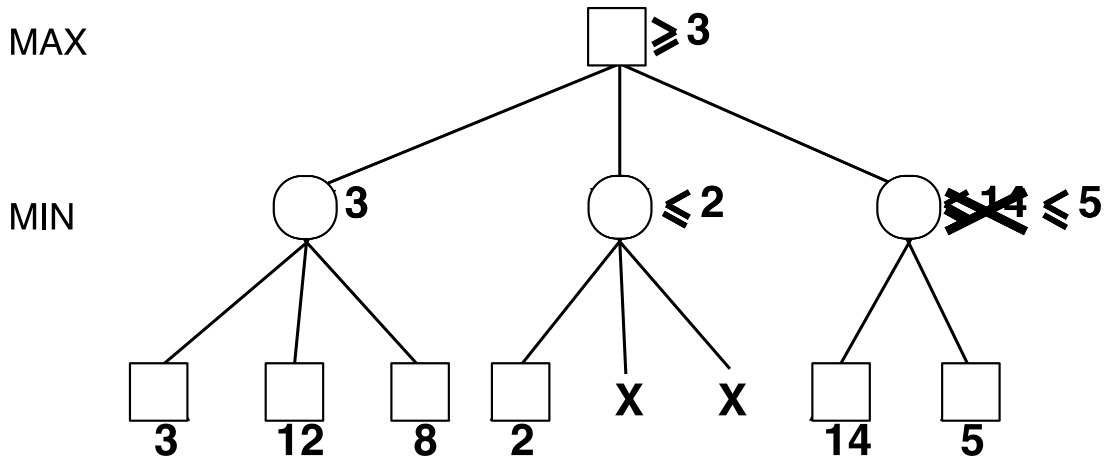
◇ Thus we don't need to figure out this node's minimax value

Pruning example 1



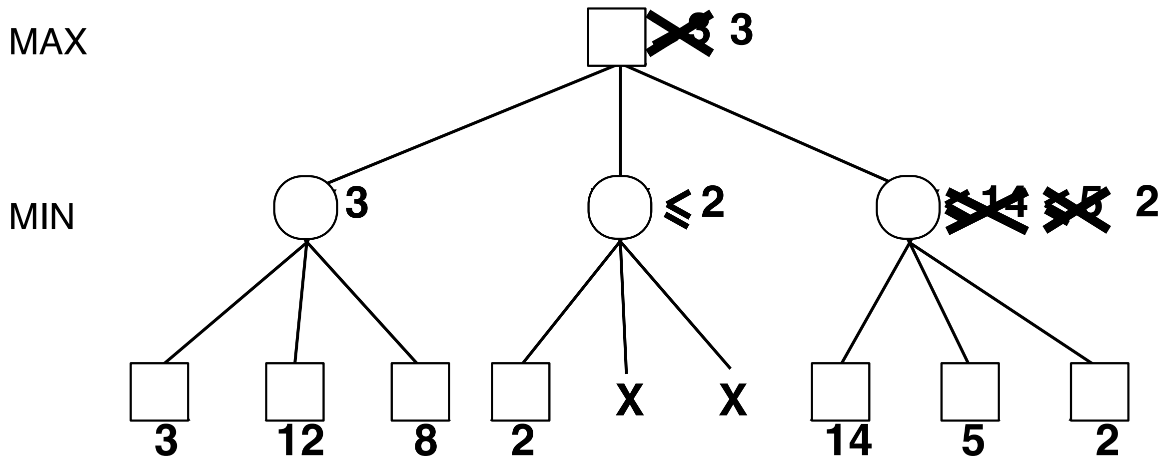
This node might be better than the first one

Pruning example 1



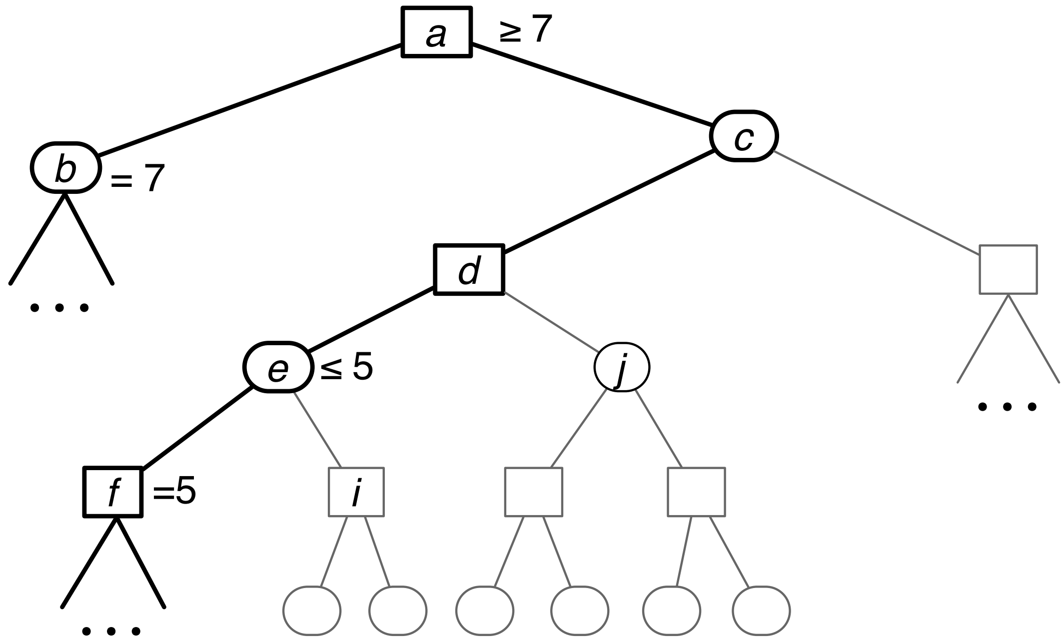
It still might be better than the first one

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 cutoff

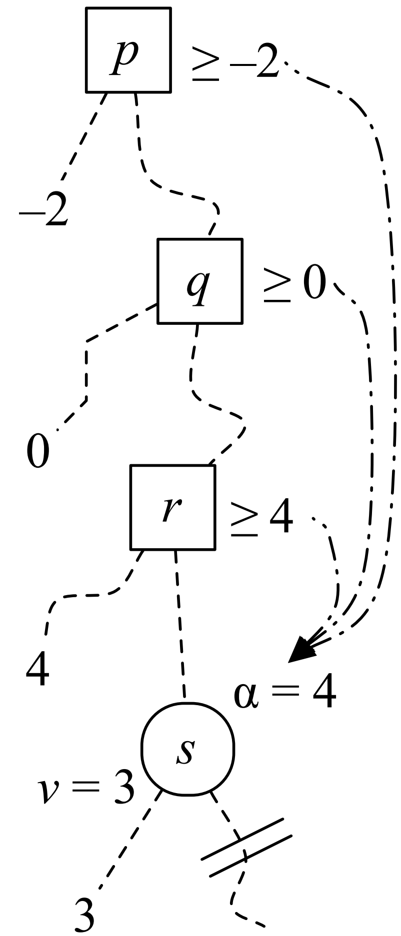
◇ Let s be any state where it's Min's move

◇ If we have visited some of s's children, then we have an upper bound $v \geq u(s)$

- Let $\alpha =$ lower bound on the best alternative for Max along the path to s
- If $v \leq \alpha$, then Max can do at least as well by moving off of the path to s
  - ◇ So stop searching below s
- This is called an *alpha cutoff*

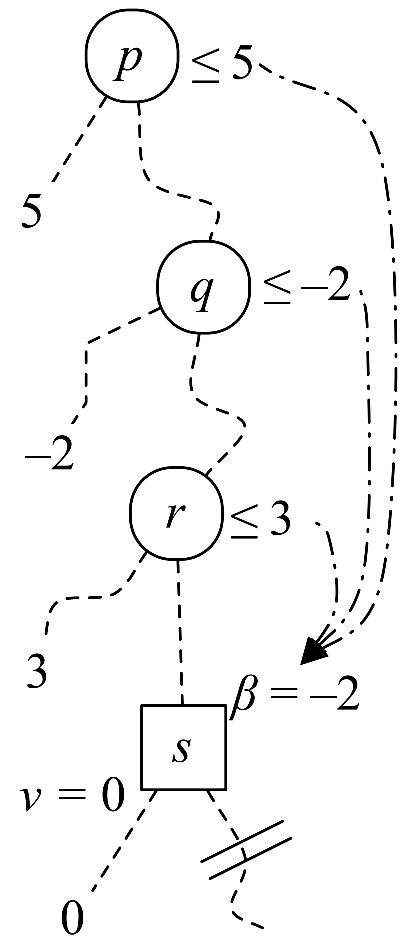
◇ Example:

- In the figure, $\alpha = \max(-2, 0, 4) = 4$
- $v = 3 < \alpha$, so stop searching below s
- Max can do better by moving to r



Beta cutoff

- ◇ Let s be any state where it's Max's move
- ◇ Let $\beta =$ upper bound on Min's best alternative along the path to s
- ◇ If we have visited some of s's children, then we have a lower bound $v \leq u(s)$
  - If $v \geq \beta$, then Min can do at least as well by moving off of the path to s
    - ◇ So stop searching below s
  - This is called a *beta cutoff*
- ◇ Example:
  - In the figure, $\beta = \min(5, -2, 3) = -2$
  - $v = 0 > \beta$, so stop searching below s
  - Min can do better by moving to q

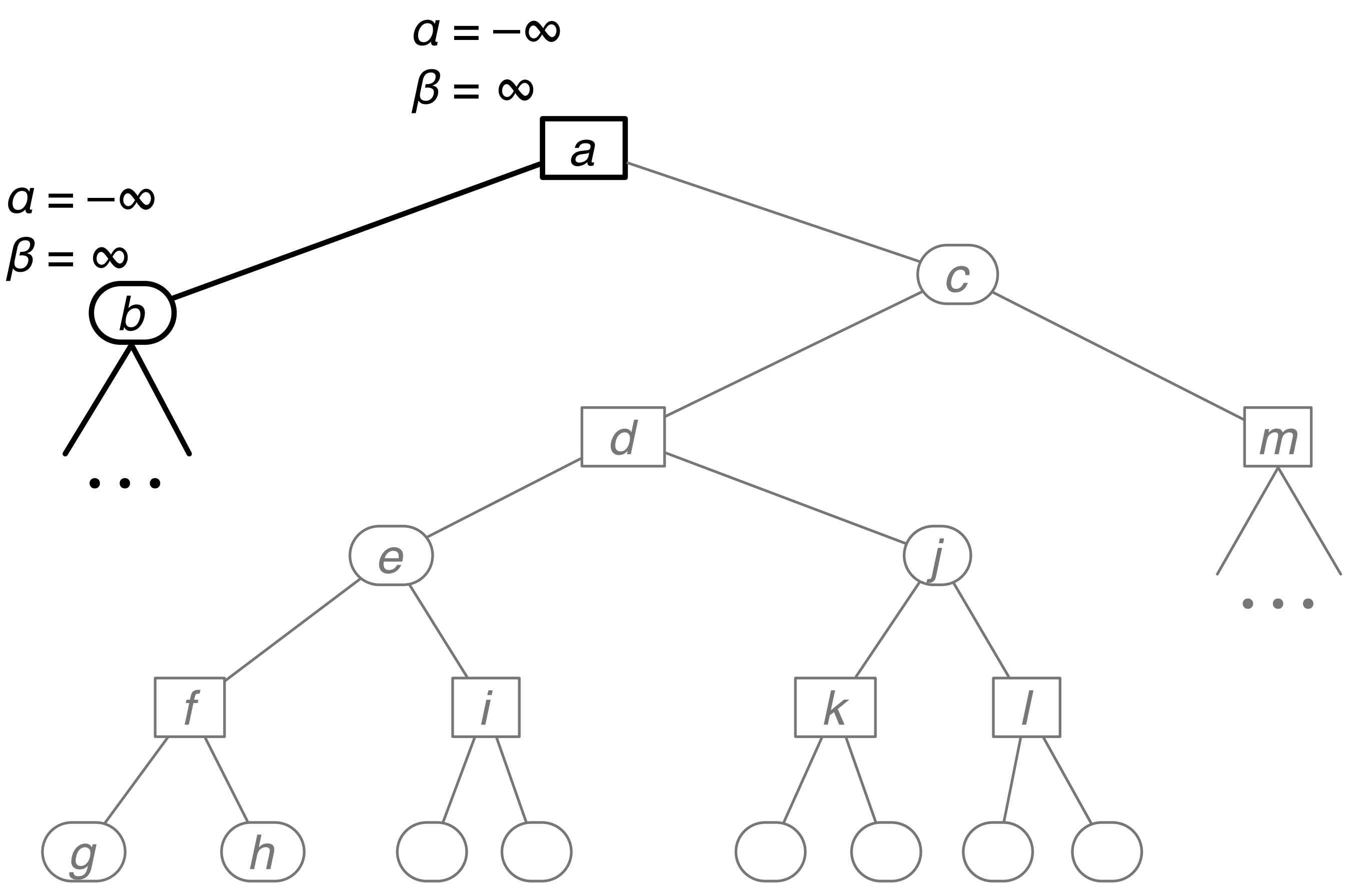


The alpha-beta algorithm

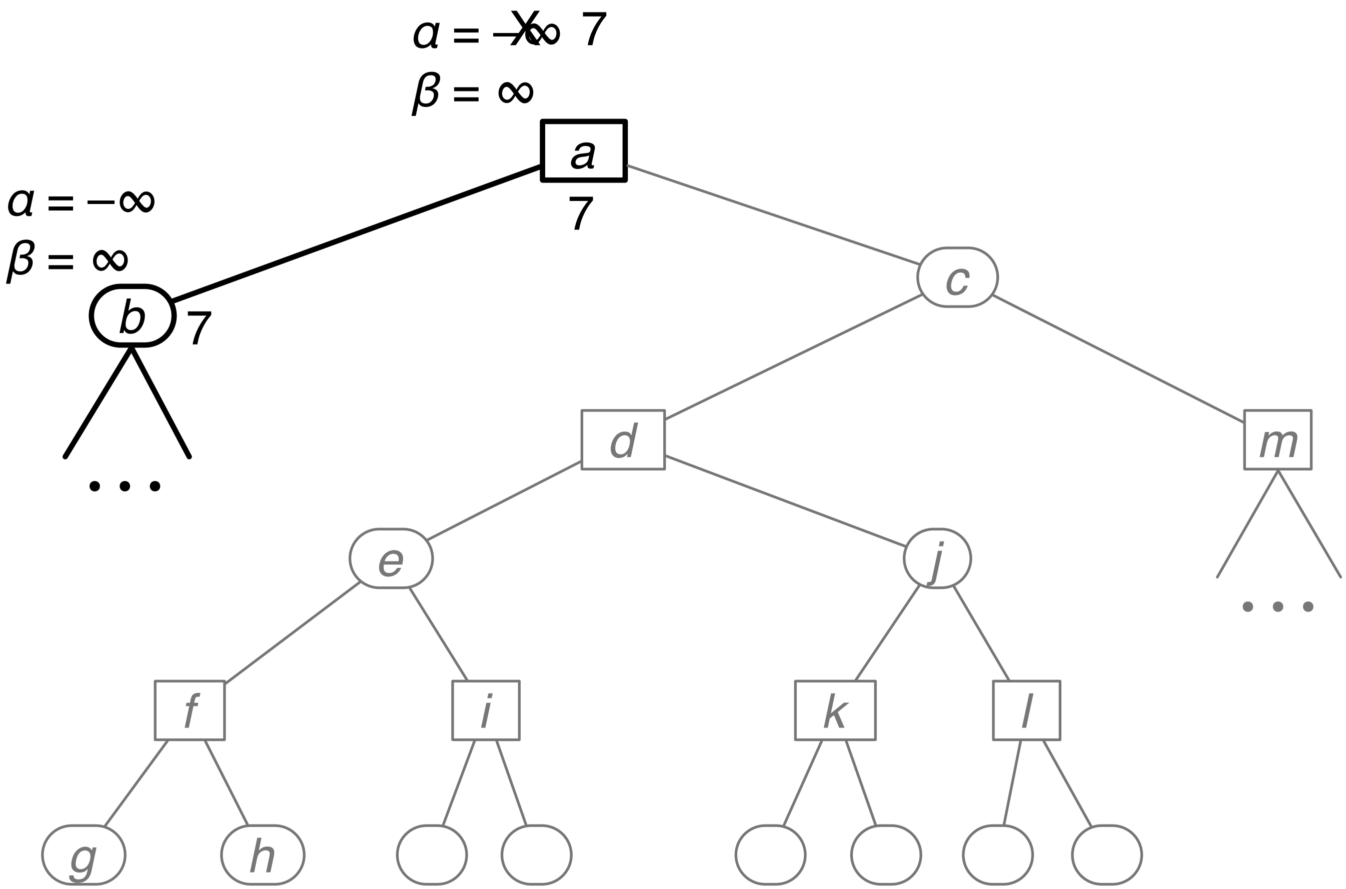
```
function ALPHA-BETA(s, α, β)
  inputs: s, current state
          α, lower bound on Max's best alternative along the path to s
          β, upper bound on Min's best alternative along the path to s

  if s is a terminal state then return Max's payoff at s
  else if it is Max's move at s then
    v ← −∞
    for every action a applicable to s do
      v ← max(v, ALPHA-BETA(result(a, s), α, β))
      if v ≥ β then return v
      α ← max(α, v)     // Max's best alternative along the path to descendants of s
  else
    v ← ∞
    for every action a applicable to s do
      v ← min(v, ALPHA-BETA(result(a, s), α, β))
      if v ≤ α then return v
      β ← min(β, v)     // Min's best alternative along the path to descendants of s
  return v
```

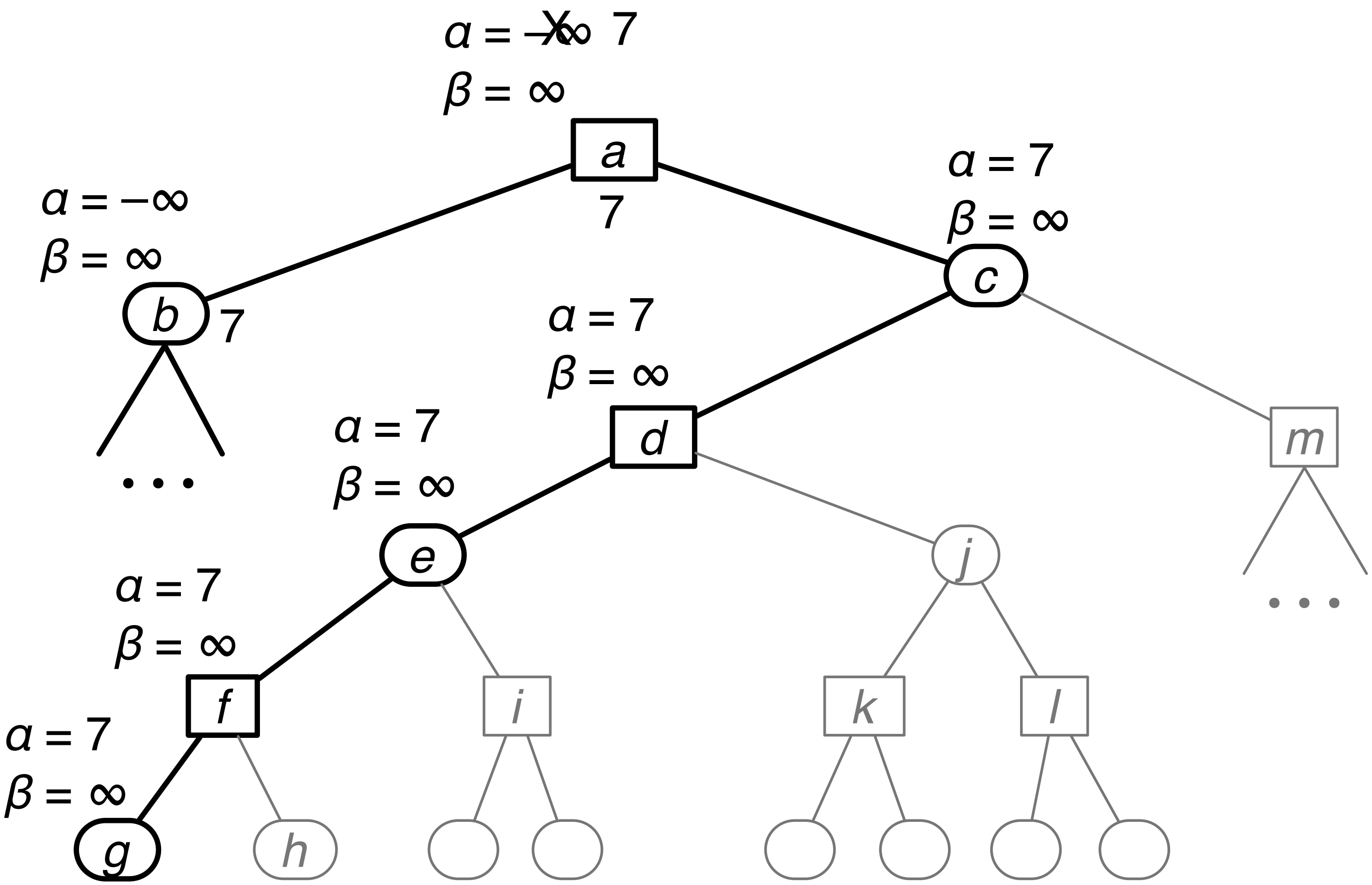
α-β pruning example



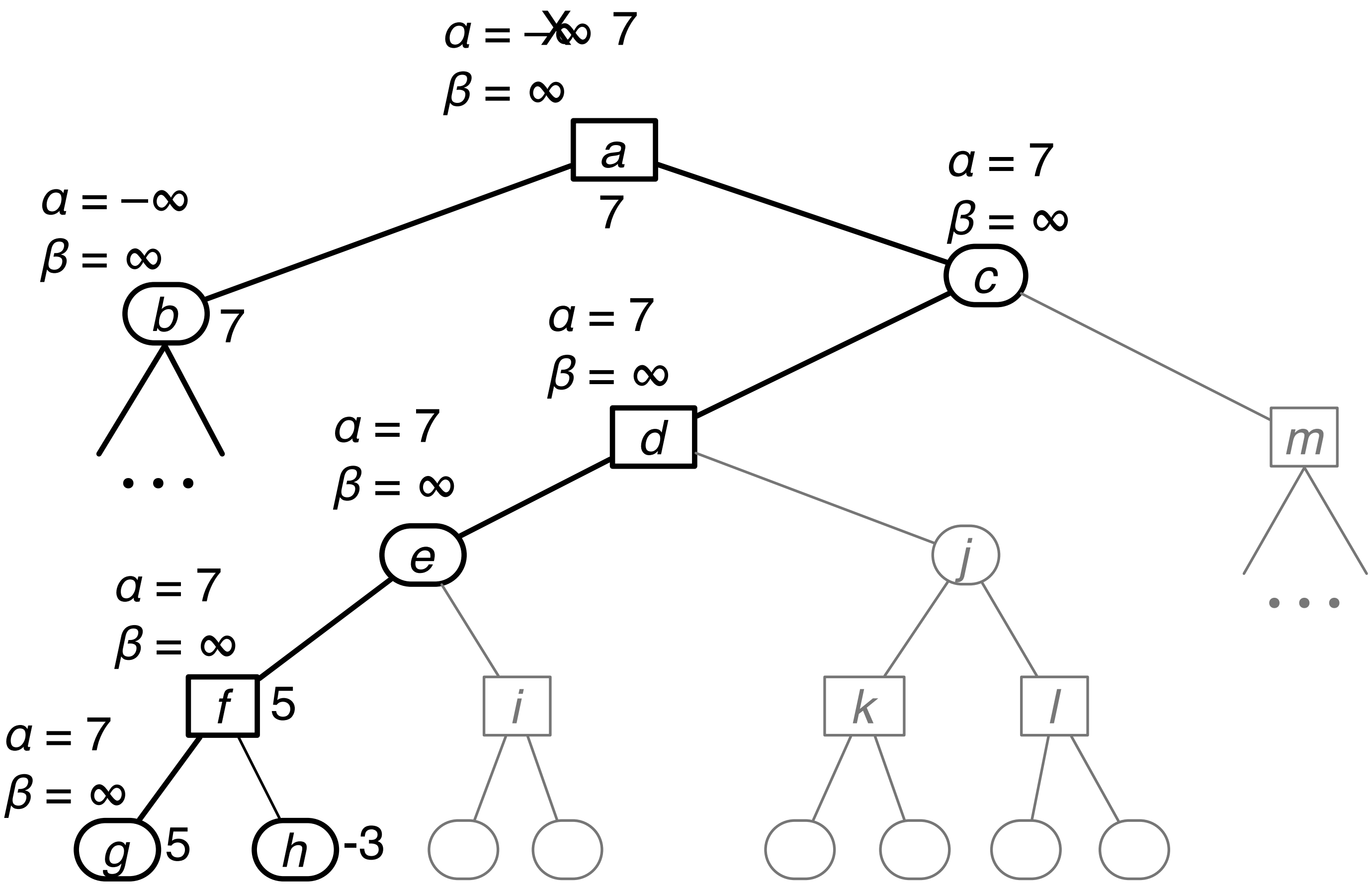
α-β pruning example



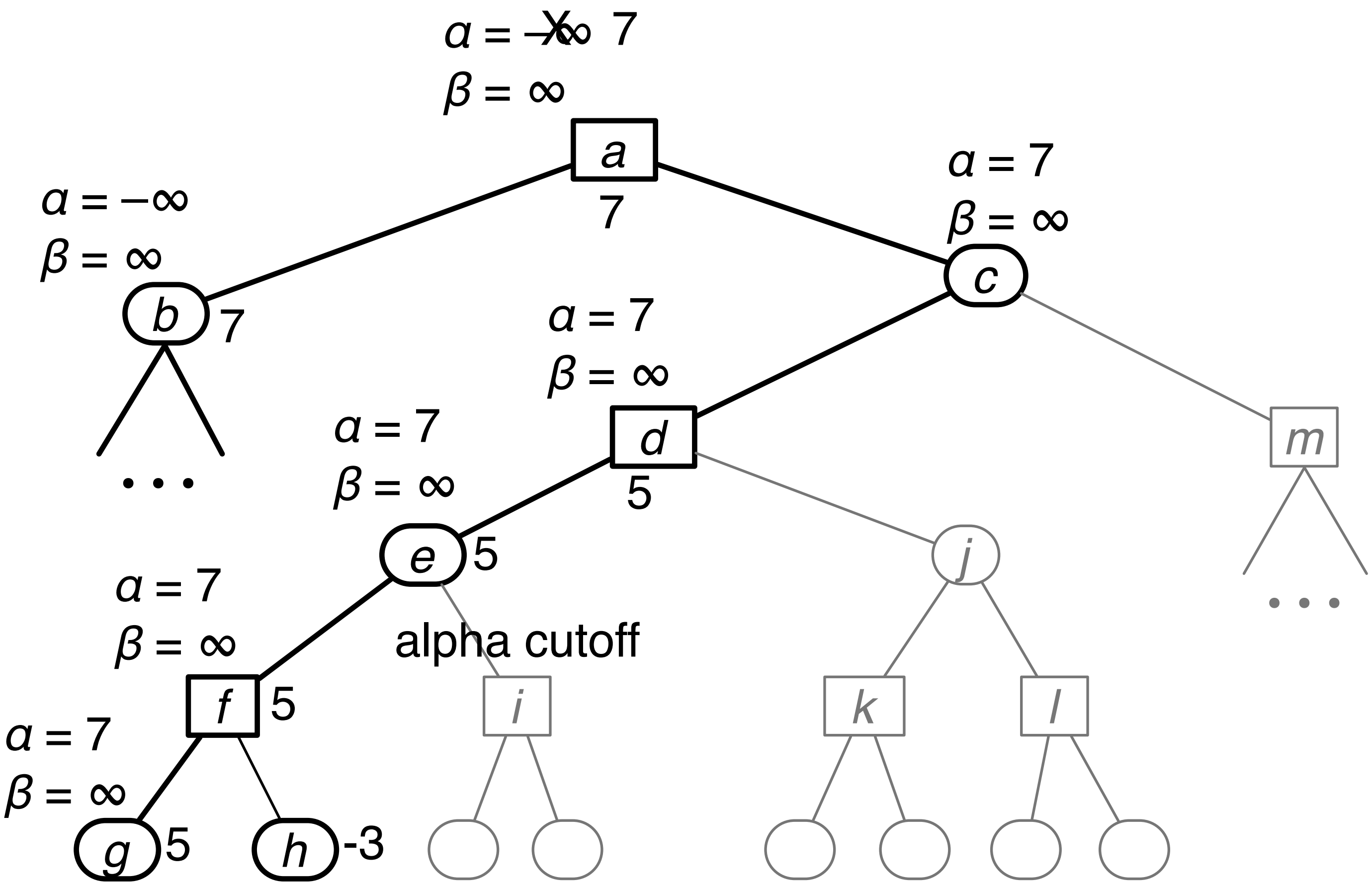
α-β pruning example



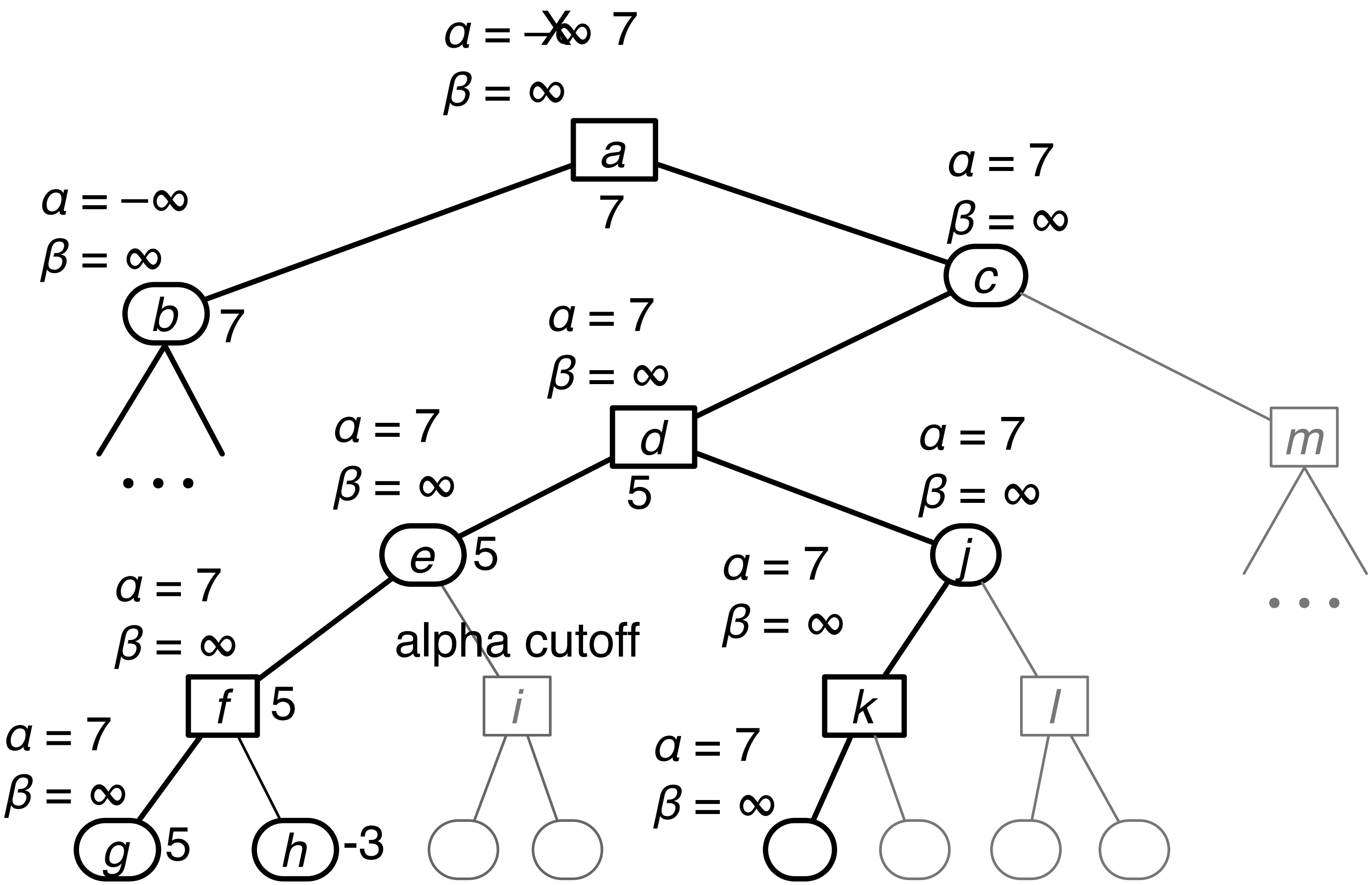
α-β pruning example



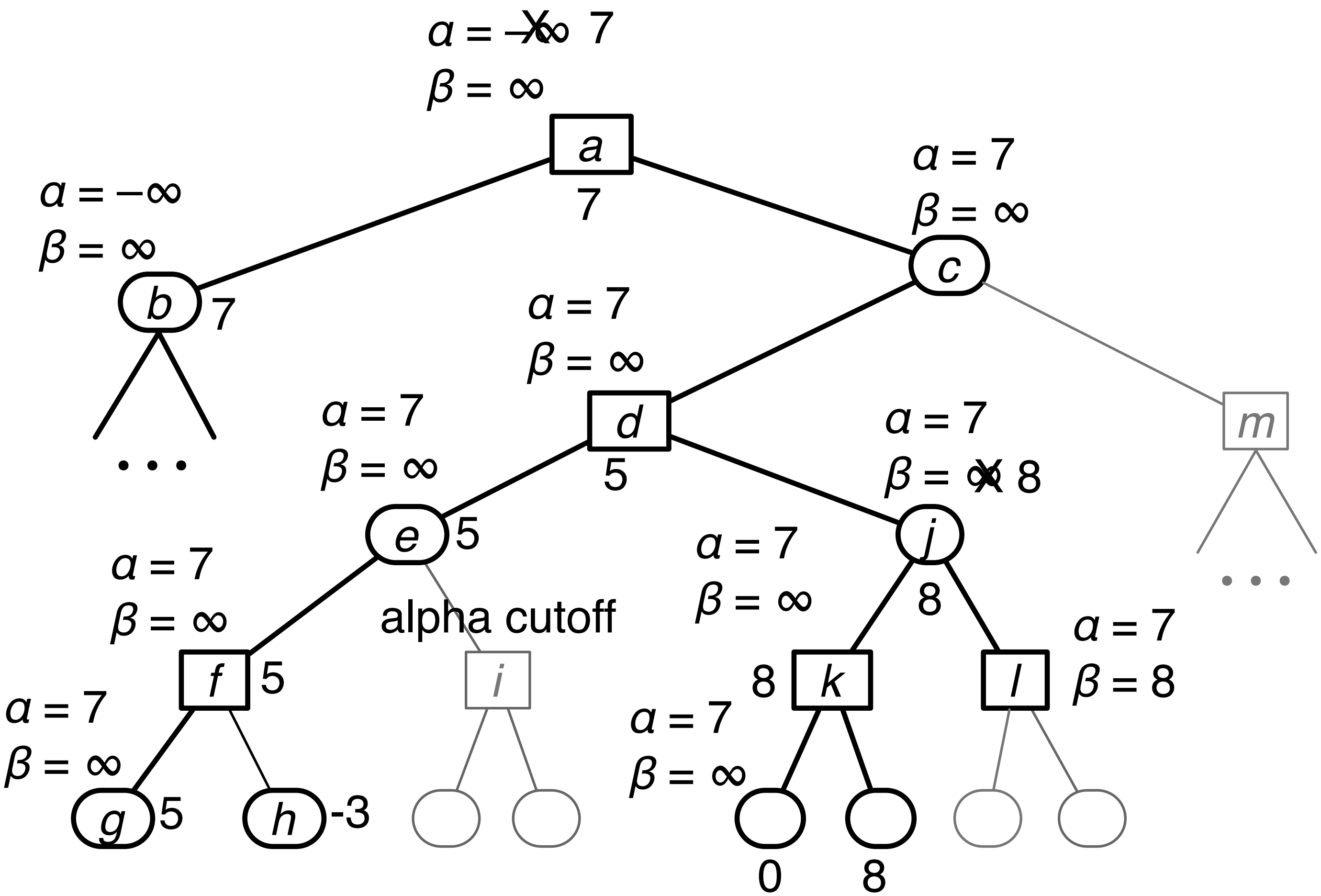
α-β pruning example



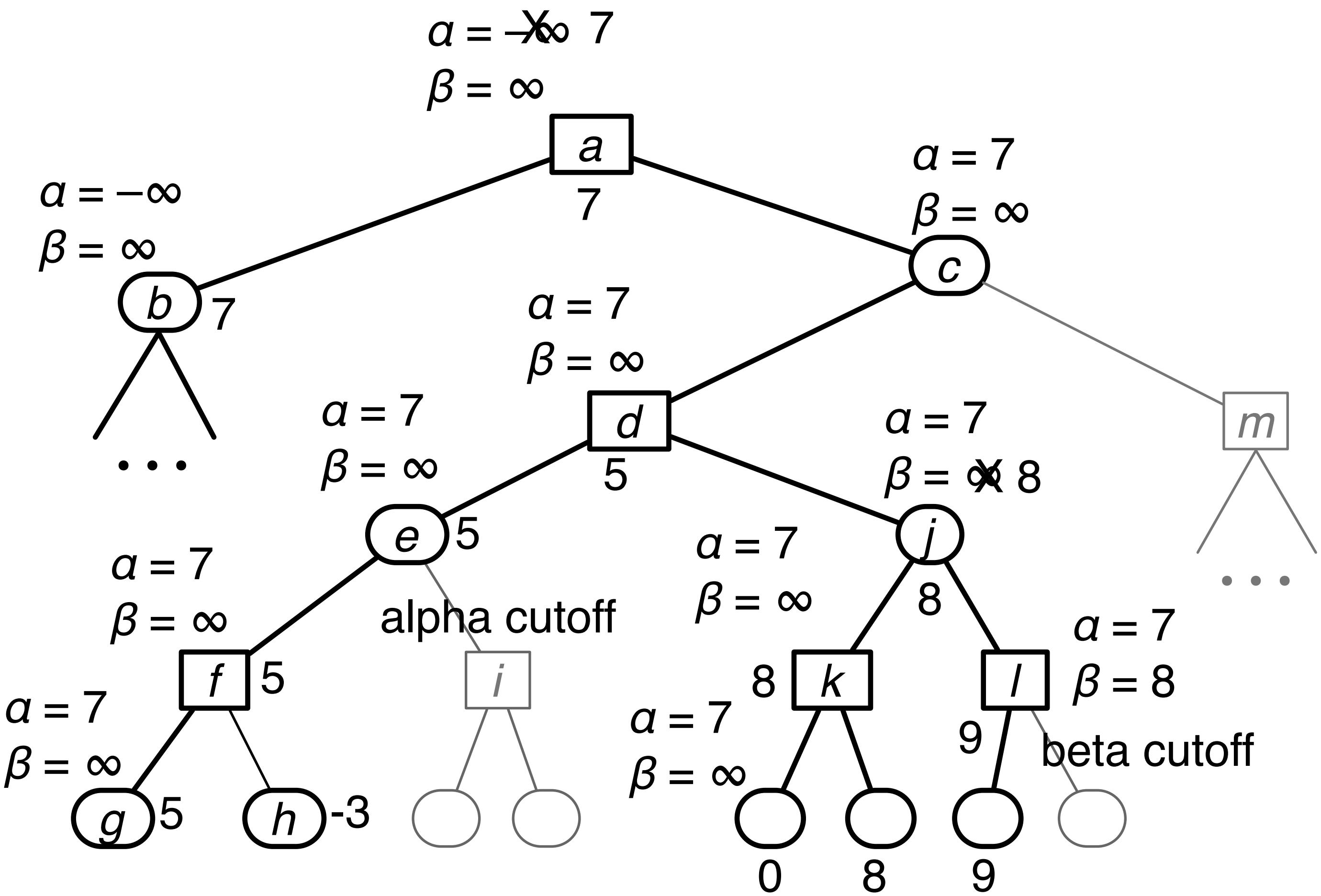
α-β pruning example



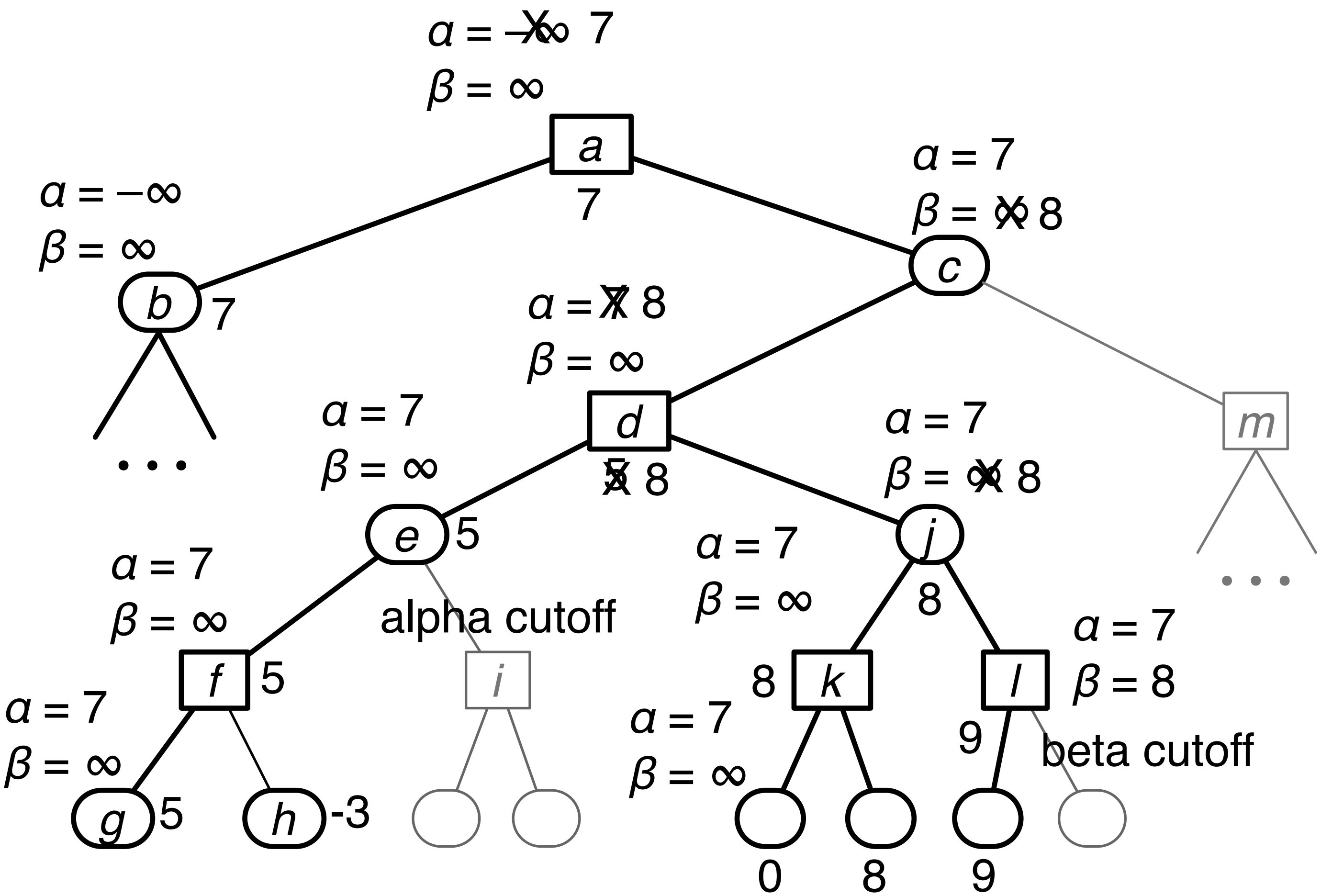
α-β pruning example



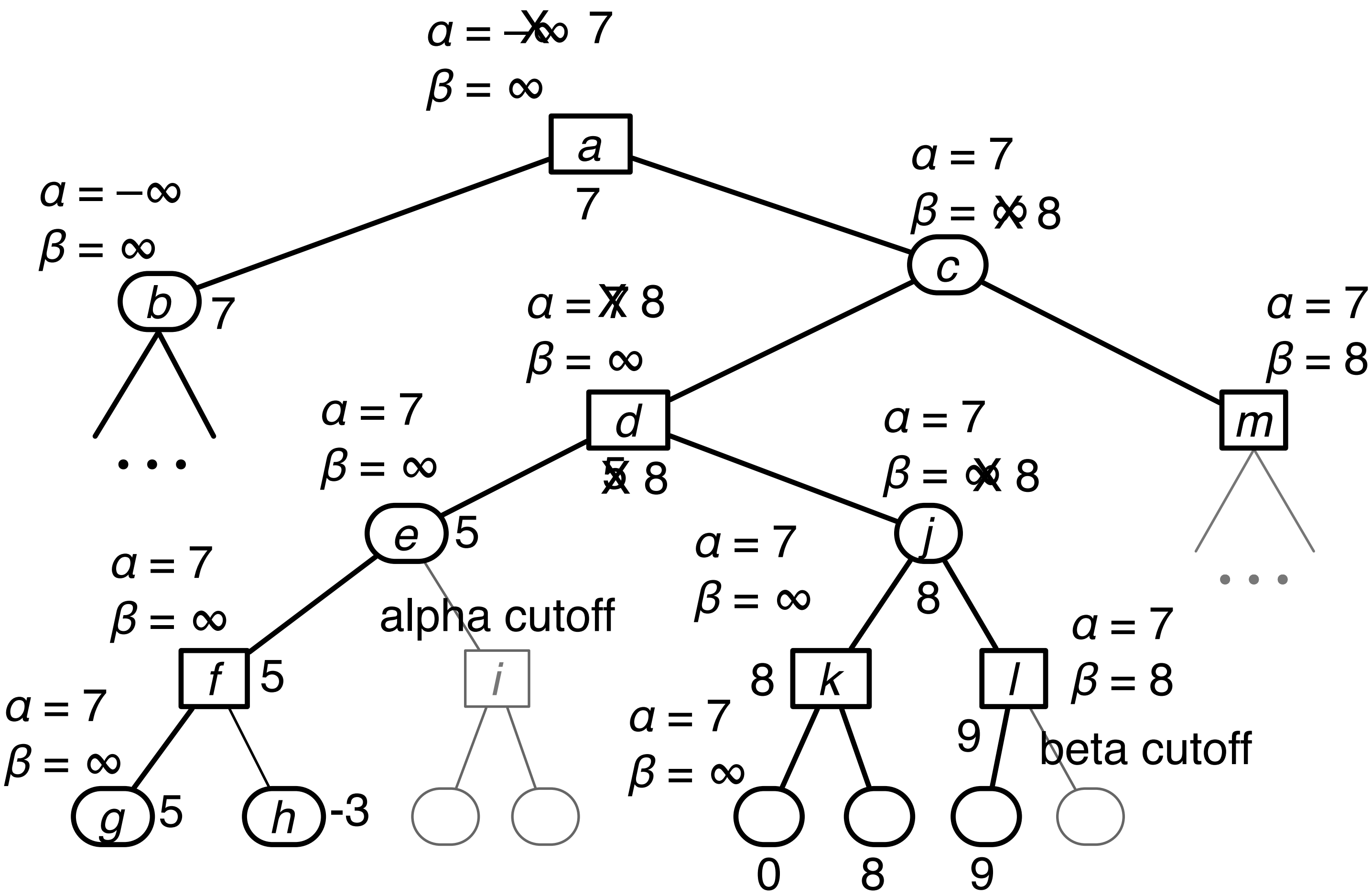
α-β pruning example



α-β pruning example



α-β pruning example



Properties of α-β

- The alpha-beta algorithm is a simple example of reasoning about which computations are relevant (a form of *metareasoning*)
  - if $\alpha \leq \text{minimax}(s) \leq \beta$, then alpha-beta returns $\text{minimax}(s)$
  - if $\text{minimax}(s) \leq \alpha$, then alpha-beta returns a value $\leq \alpha$
  - if $\text{minimax}(s) \geq \beta$, then alpha-beta returns a value $\geq \beta$
- If we start with $\alpha = -\infty$ and $\beta = \infty$, then alpha-beta will always return $\text{minimax}(s)$
- Good move ordering can enable us to prune more nodes
  - Best case is if
    - at nodes where it's Max's move, children are largest-value first
    - at nodes where it's Min's move, children are smallest-value first
    - In this case time complexity $= O(b^{h/2}) \Rightarrow$ twice the solvable depth
  - Worst case is the reverse
    - In this case, α-β will search every node

Resource limits

- Even with alpha-beta, it can still be infeasible to search the entire game tree
  - e.g., recall chess has about 10^{135} nodes
  - $\Rightarrow$ need to limit the depth of the search
- Basic approach: have a maximum search depth d
  - Whenever we reach a node of depth $> d$
    - If we're at a terminal state, then return Max's payoff
    - Otherwise return an *estimate* of the node's utility value, computed by a **static evaluation function**

α-β with a bound d on the search depth

function ALPHA-BETA(s, α, β, d)
  inputs: s, current state
      α, lower bound on Max's best alternative along the path to s
      β, upper bound on Min's best alternative along the path to s

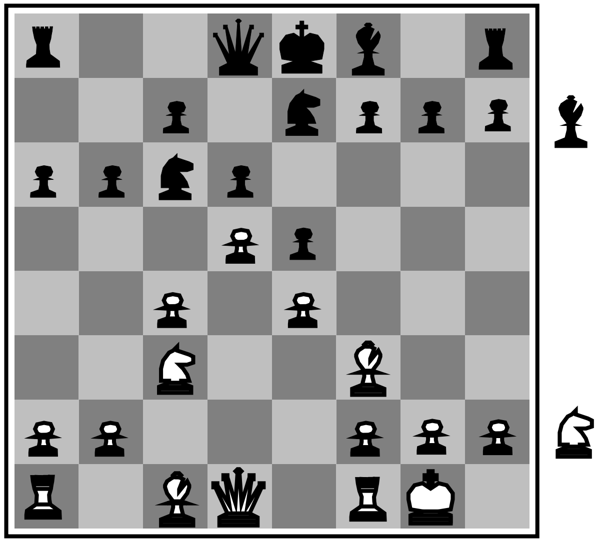
  if s is a terminal state **then return** Max's payoff at s
  else if $d = 0$ **then return** EVAL(s)
  else if it is Max's move at s **then**
    $v \leftarrow -\infty$
    for every action a applicable to s **do**
      $v \leftarrow \max(v, \text{ALPHA-BETA}(\text{result}(a, s), \alpha, \beta, d-1))$
      if $v \geq \beta$ **then return** v
      $\alpha \leftarrow \max(\alpha, v)$ // Max's best alternative along the path to descendants of s
  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
      $\beta \leftarrow \min(\beta, v)$ // Min's best alternative along the path to descendants of s
  return v

Evaluation functions

◇ $\text{EVAL}(s)$ is supposed to return an approximation of s's minimax value

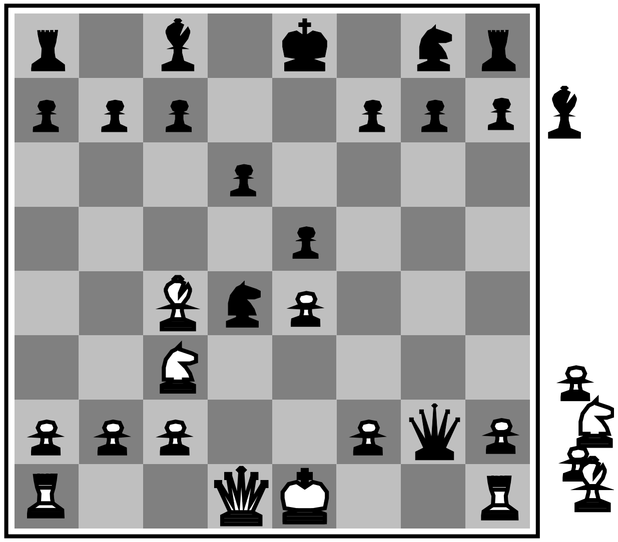
◇ EVAL is often a weighted sum of *features*

- $\text{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$
- E.g., $1 \times$ (number of white pawns − number of black pawns)
  $+ 3 \times$ (number of white knights − number of black knights)
  $+ \ldots$



Black to move

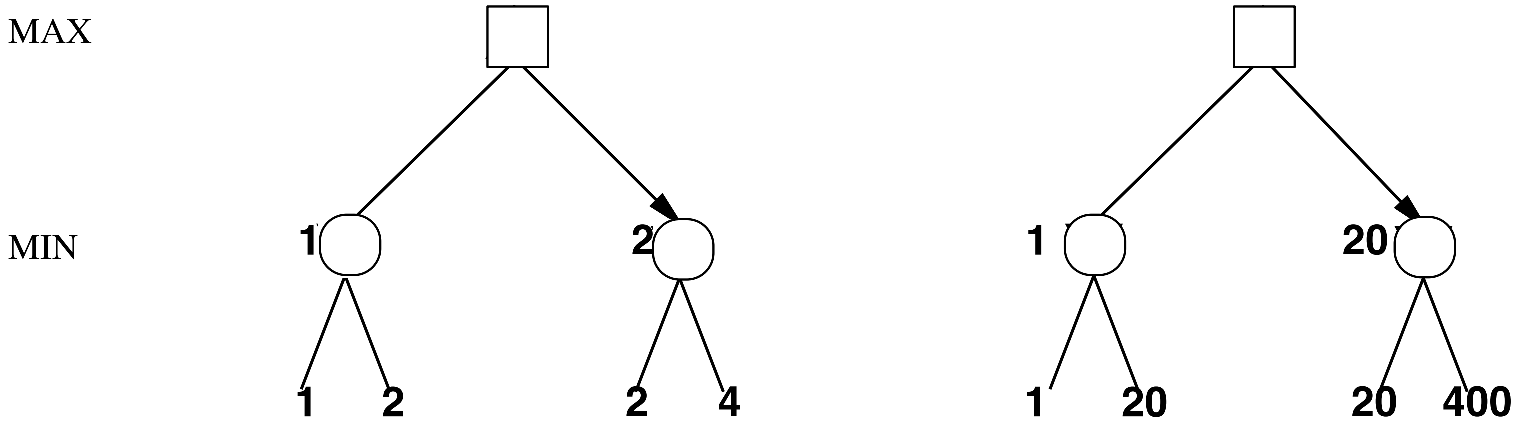
White slightly better



White to move

Black winning

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

- Increasing the search depth usually gives better decisions
- There are some exceptions
  - Main result in my PhD dissertation (more than 30 years ago!): "pathological" games in which deeper search 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
  ...

Node ordering

◇ Recall that I said:

- Best case is if
  - ◇ at nodes where it's Max's move, children are largest-value first
  - ◇ at nodes where it's Min's move, children are smallest-value first
  - ◇ In this case time complexity = $O(b^{h/2})$ $\Rightarrow$ twice the solvable depth
- Worst case is the reverse
  - ◇ In this case, α-β will search every node

◇ How to get closer to the best case:

- Every time you expand a state, apply EVAL to its children
- If it's Min's move, sort the children in order of their EVAL values
- If it's Max's move, sort the children in reverse order of their EVAL values

Quiescence search and biasing

- In a game like checkers or chess
  - The evaluation is based greatly on material pieces
  - It's likely to be inaccurate if there are pending captures
    - e.g., if someone is about to take your queen
- Search deeper to reach a position where there aren't pending captures
  - Evaluations will be more accurate here
- But it creates another problem
  - You're searching some paths to an even depth, others to an odd depth
  - Paths that end just after your opponent's move
    will generally look worse than paths that end just after your move
- Add or subtract a number called the "biasing factor" to try to fix this

Transposition tables

- Often there are multiple paths to the same state
  - i.e., the state space is a really graph rather than a tree
- Idea:
  - when you compute a node's minimax value, store it in a hash table
  - visit it again $\Rightarrow$ retrieve its value rather than computing it again
- The hash table is called a **transposition table**
  - Any idea why?
- Problem: far too many states to store all of them
  - Store some of the states, rather than all of them
  - Try to store the ones that you're most likely to need

Thinking on the opponent's time

- Current state s, children $s_1, \ldots, s_n$
- Compute their minimax values, move to the one that looks best
  - Suppose it's s_i
- You computed s_i's minimax value as the min of its children, $s_{i1}, \ldots, s_{im}$
- Let s_{ij} be the child that has the smallest minimax value
  - According to your analysis, that's where the opponent is likely to move
- While waiting for the opponent to move, do a minimax search at s_{ij}
  - If your opponent moves to s_{ij}
    - then you have a head start on figuring out your next move
  - If your opponent moves to s_{ij}
    - then its no worse than if you just waited

Game-tree search in practice

- **Checkers** was solved in April 2007; took 10^{14} calculations over 18 years
  - With perfect play, it's a draw
  - Search space of size 5×10^{20}
- **Chess**: Deep Blue searches 200 million positions per second
  - very sophisticated evaluation
  - undisclosed methods for extending some lines of search up to 40 ply
- **Othello** programs are much better than the best human players
- **Go**: Until about 5 years ago, computer programs were very bad
  - A different kind of tree search has improved them dramatically
  - Now, probably about as good as a good amateur

Game-tree search in the game of go

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

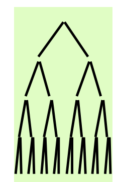
◇ Go is much too big for a normal game-tree search:

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

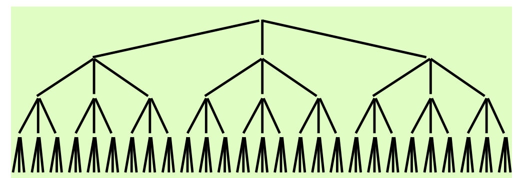
◇ For comparison, the size of the universe

- About 10^{80} atoms
- About 10^{87} particles

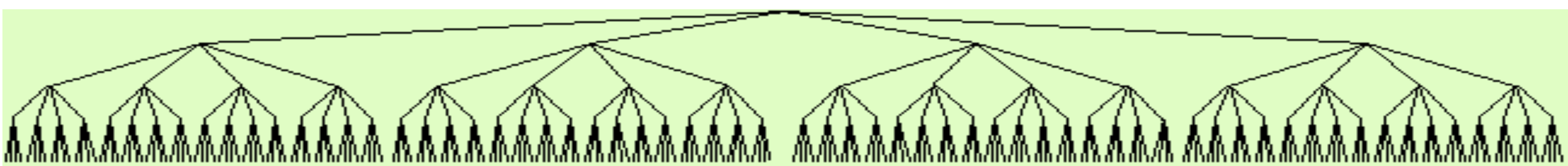
b =2



b =3



b =4



Game-tree search in the game of go

◇ During the past 4–5 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
- It calls itself recursively on these, ignores the others

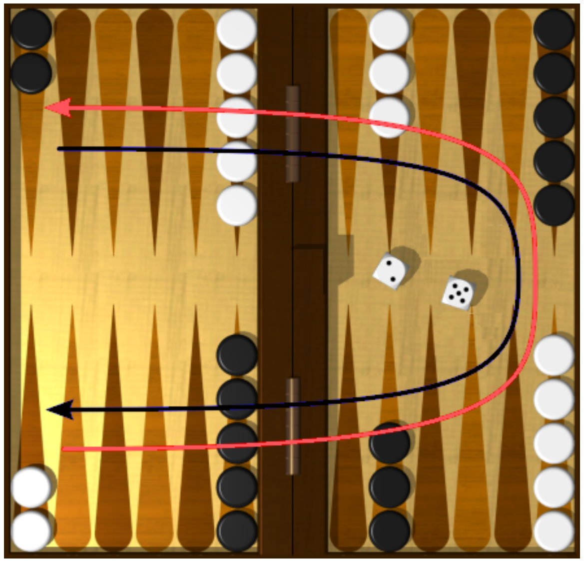
Forward pruning in chess

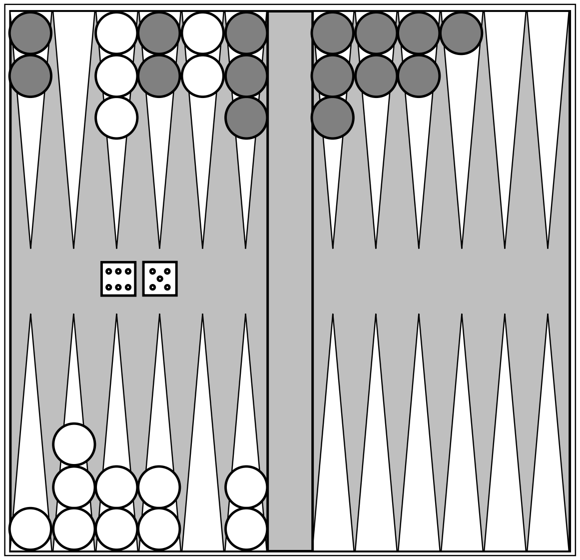
- ◇ Back in the 1970s, some similar ideas were tried in chess
- ◇ The approach was called **forward pruning**
  - Main difference: select the children heuristically rather than randomly
  - It didn't work as well as brute-force alpha-beta, so people abandoned it
- ◇ Why does a similar idea work so much better in go?

Perfect-information stochastic games

◊ Example: **backgammon**

- Two players who take turns
- At each turn, the set of available moves depends on the results of rolling the dice
- Each die specifies how far to move one of your pieces (except if you roll doubles)
- If your piece will land on a location that contains 2 or more of the opponent's piece you can't move there
- If your piece lands on a location that contains 1 of the opponent's pieces, that piece must start over

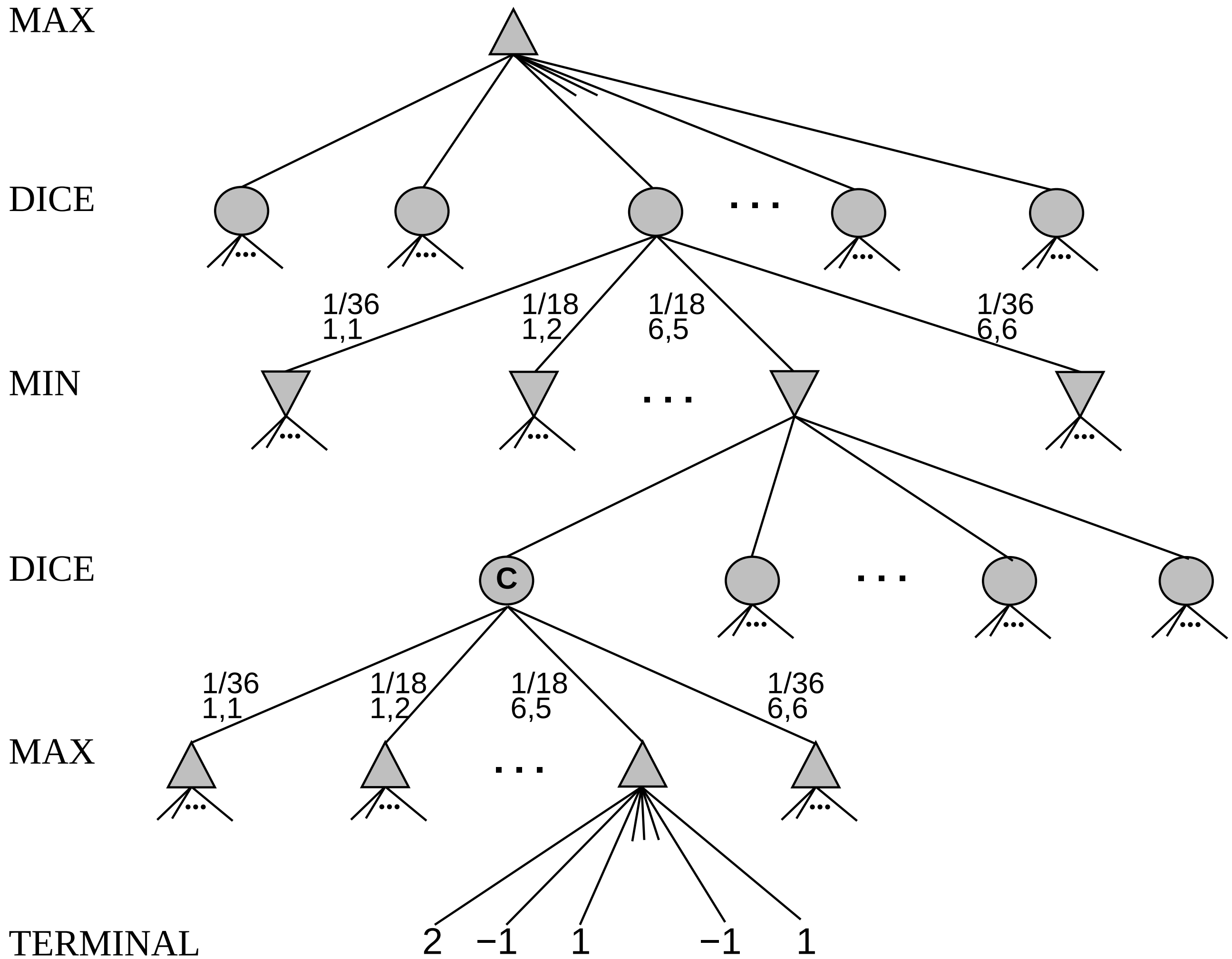




Backgammon game tree

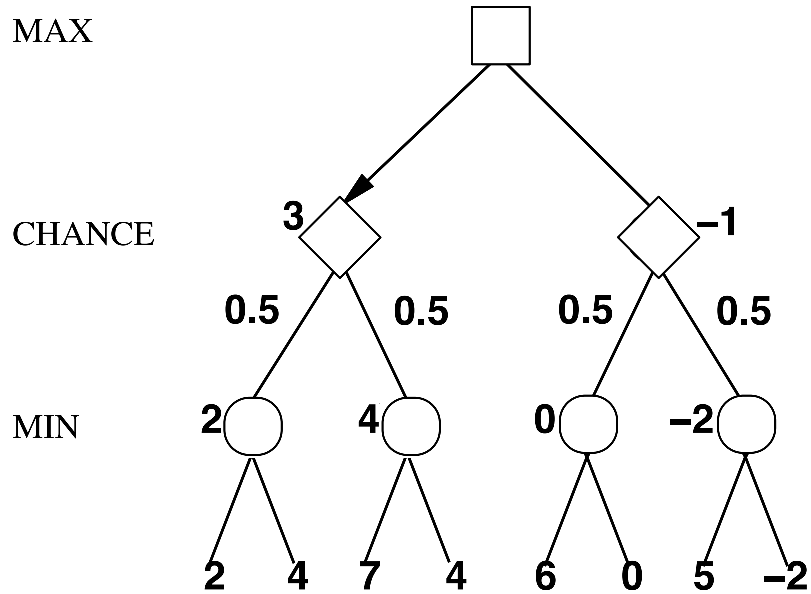
◊ The players' moves have deterministic outcomes

◊ The dice rolls have stochastic outcomes



Expectiminimax

- ◇ Returns expected minimax value
- ◇ Can be modified to return actions
- ◇ Can also be modified to do α-β pruning
  - But it's more complicated and less effective than in deterministic games

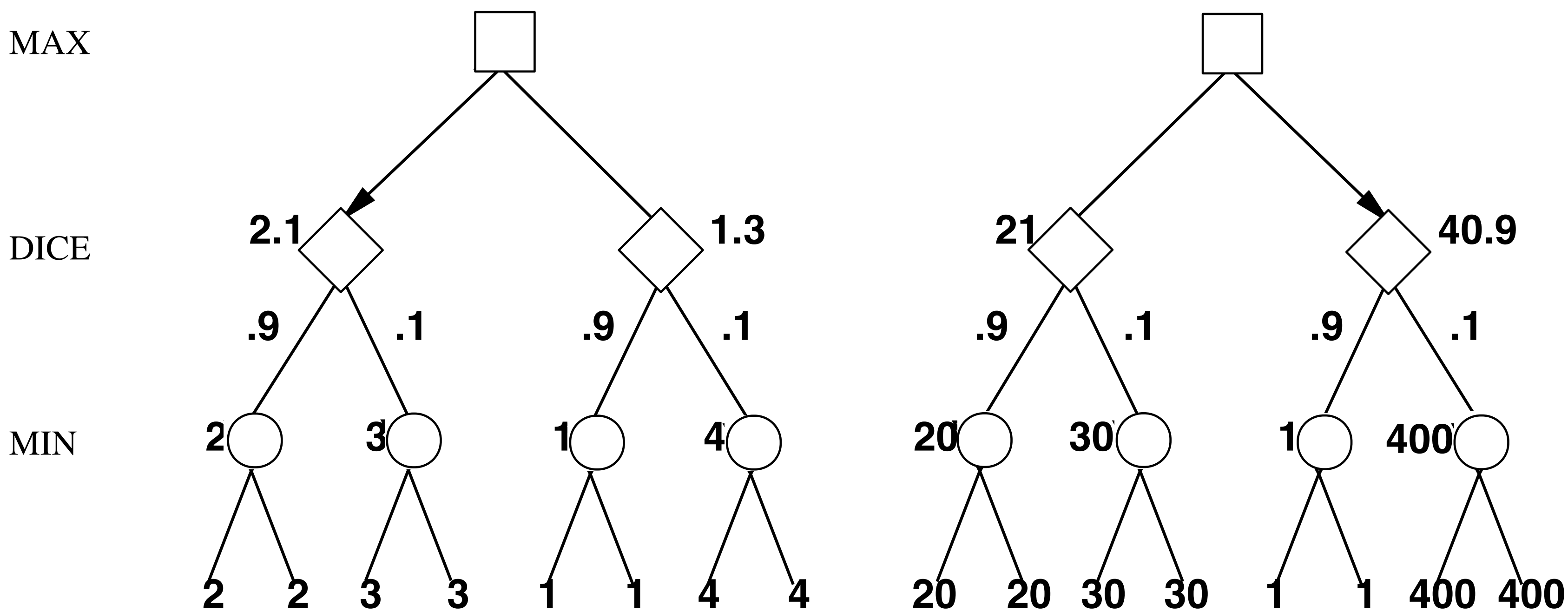


function EXPECTIMINIMAX(s, d)
  if s is a terminal state **then return** Max's payoff at s
  else if $d = 0$ **then return** EVAL(s)
  else if s is a "chance" node **then**
    return $\sum_{t \in \text{children}(s)} P(t|s)$ EXPECTIMINIMAX($t, d-1$)
  else if it is Max's move at s **then**
    return max{EXPECTIMINIMAX(result(a, s), $d-1$) : a is applicable to s}
  else return min{EXPECTIMINIMAX(result(a, s), $d-1$) : a is applicable to s}

In stochastic games, 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



In practice

◊ Dice rolls increase b: 21 possible rolls with 2 dice

◊ Given the dice roll, $\approx$ 20 legal moves on average

- (for some dice rolls, can be much higher)

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

◊ 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

◊ The evaluation function was created automatically using a machine-learning technique called Temporal Difference learning

- hence the TD in TDGammon

Summary

- We looked at games that have the following characteristics:
  - two players, zero sum, perfect information, finite
- Case 1: deterministic
  - In these games, can do a game-tree search
    - minimax values, alpha-beta pruning
  - In sufficiently complicated games, perfection is unattainable
    - approximate using limited search depth, static evaluation function
  - In some games, other techniques are better
    - Monte Carlo roll-outs
- Case 2: stochastic (e.g., dice rolls)
  - Expectiminimax

Reminder: midterm exam postponed

- October 9 was causing problems for too many people

- We discussed this in class last Tuesday, and decided to postpone it to Thursday, October 18