CMSC 474, Game Theory

4b. Game-Tree Search

Dana Nau
University of Maryland

Finite perfect-information zero-sum games

- **Finite**:
  - finitely many agents, actions, states, histories
- **Perfect information**:
  - Every agent knows
    - all of the players' utility functions
    - all of the players' actions and what they do
    - the history and current state
  - No simultaneous actions – agents move one-at-a-time
- **Constant sum** (or **zero-sum**):
  - Constant k such that regardless of how the game ends,
    - $\Sigma_{i=1,\ldots,n}\, u_i = k$
  - For every such game, there's an equivalent game in which $k = 0$

Examples

- **Deterministic**:
  - outcomes depend only on the players' moves
  - tic-tac-toe, qubic, connect-four, mancala, reversi (othello), chess, checkers, go
- **Stochastic**:
  - outcomes depend partly on chance (e.g., dice rolls)
  - backgammon, monopoly, yahtzee, parcheesi, roulette, craps
- For now, we'll consider just the deterministic games

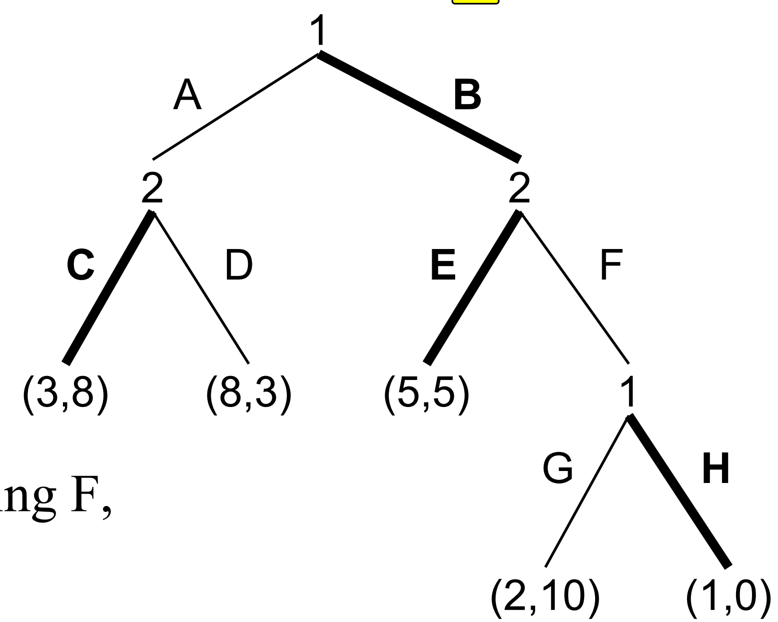
Outline

- Non-credible threats
- Restatement of the Minimax Theorem
- Game trees
- The minimax algorithm
- α-β pruning
- Resource limits, approximate evaluation

- Most of this isn't in the game-theory book
  - I'll post some material on Piazza
- For further information, look at one of the following
  - Russell & Norvig's *Artificial Intelligence: A Modern Approach*
    - Chapter 6 of the 2^{nd} edition
    - Chapter 5 of the 3^{rd} edition

No Non-Credible Threats

- Recall this example from Chapter 4:
- If the agents can announce their strategies beforehand, agent 1 might want to announce (B,H)
- Non-credible threat:
  - Agent 1 is trying to keep 2 from choosing F, by threatening to retaliate with H
  - But if 2 chooses F anyway, then it wouldn't be rational for 1 to do H, because H also hurts 1



- In zero-sum games, non-credible threats can't occur
  - In a zero-sum game, if H hurts 2 then it helps 1

Minimax Theorem (Restated)

- Instead of u_1 and u_2, write u and $-u$
  - Call player 1 Max (wants to maximize u), call Max's strategy s
  - Call player 2 Min (wants to minimize u), call Min's strategy t
- **Theorem.** Let G be any finite two-player zero-sum game. Then
  - Max's expected utility in any Nash equilibrium
    = Max's maxmin value = Max's minmax value
  - In other words, for every Nash equilibrium (s^*,t^*),
    - $u(s^*,t^*) = \min_s \max_t u(s,t) = \max_s \min_t u(s,t)$
- **Corollary.** There are pure strategies s^* and t^* that satisfy the theorem.
- Terminology
  - $u(s^*,t^*)$ is the called the *minimax value* of G
  - s^* and t^* are sometimes called *optimal strategies* or *perfect play*

Terminology

- **Root** node: where the game starts
- **Max** (or **Min**) node:
  - a node where it's Max's (or Min's) move
  - usually draw Max nodes as squares, Min nodes as circles
- A node's **children**: $\{\sigma(h,a) \mid a \in \chi(h)\}$
  - **Branching factor** $b = \max_h \chi(h)$
- **Terminal** nodes: where game ends

H = {nonterminal nodes}

Z = {terminal nodes}

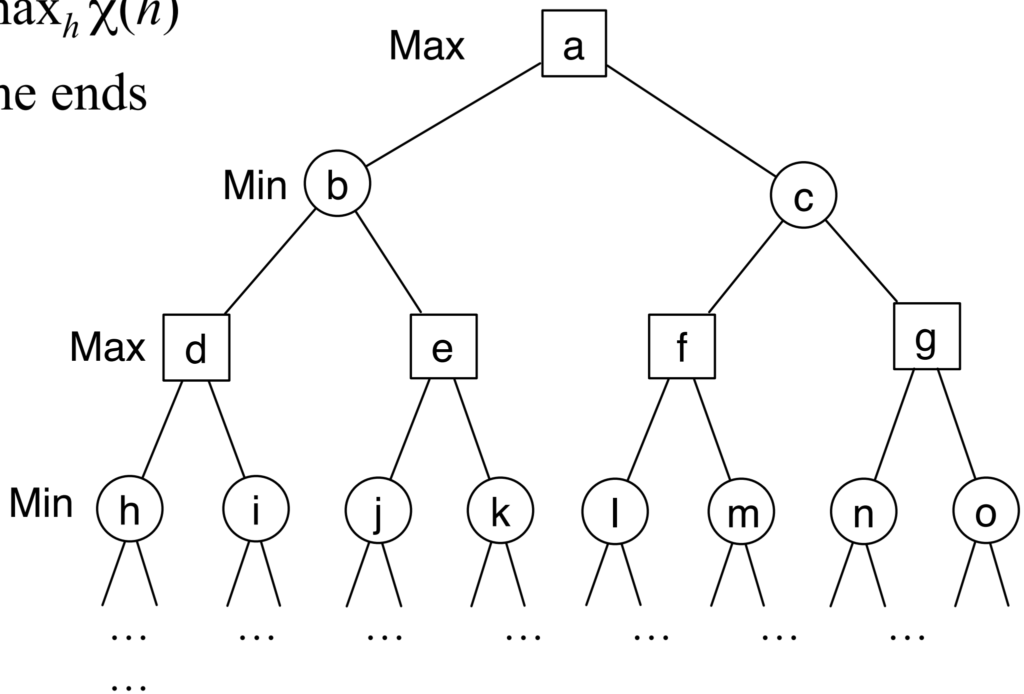
$\rho(h)$ = the player to move at h

$\chi(h)$ = {available actions at h}

$\sigma(h,a)$ = child of h produced by a

$\mathbf{u}(h)$ = utility profile for h

$\mathbf{v}[i]$ = i'th element of $\mathbf{v}$



Minimax Algorithm

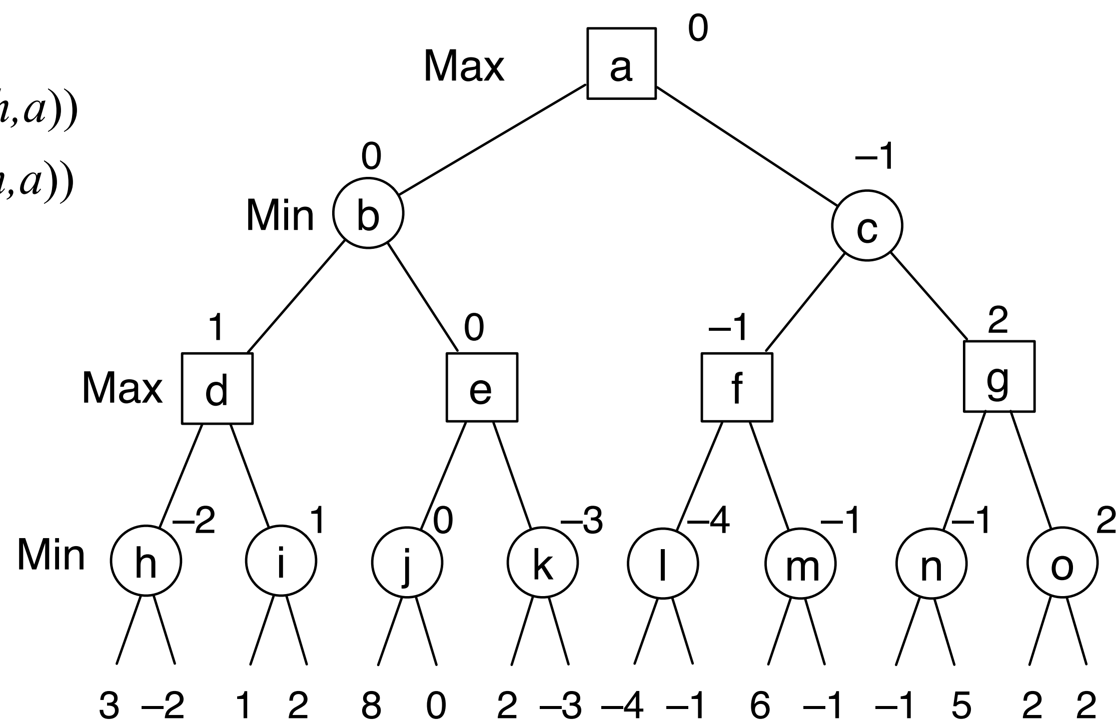
- Backward induction (Chapter 4)
  - Simplified for 2-player zero-sum games

function Minimax(h)
  if $h \in Z$ then return $u(h)$
  else $\rho(h)$ = Max then
    return $\max_{a \in \chi(h)}$ Minimax($\sigma(h,a)$)
  else return $\min_{a \in \chi(h)}$ Minimax($\sigma(h,a)$)

Z = {terminal nodes}
$\rho(h)$ = the player to move at h
$\chi(h)$ = {available actions at h}
$\sigma(h,a)$ = child of h produced by a

function Minimax-choice(h)
  if $h \in Z$ then return *error*
  else if $\rho(h)$ = Max then
    return $\arg\max_{a \in \chi(h)}$ Minimax($\sigma(h,a)$))
  else return $\arg\min_{a \in \chi(h)}$ Minimax($\sigma(h,a)$))

Poll 4.2: what would Minimax-choice return at nodes b and c?



Complexity Analysis

- Let b = branching factor, D = height of tree

- Space complexity
  = (max path length) × (space needed to store the path)
  = $O(bD)$
- Time complexity = size of the game tree = $O(b^D)$

- Example: chess
  - for "reasonable" chess games, b ≈ 35, D ≈ 100
  - $b^h \approx 35^{100} \approx 10^{135}$ nodes
- $\approx 10^{87}$ particles in the universe
  - 10^{135} nodes is $\approx 10^{48}$ times the number of particles in the universe

Limited-Depth Minimax

- Upper bound d on search depth
  - Running time $O(b^d)$
  - Space $O(bd)$
- If $d \geq \text{height}(h)$, get $u(h)$
  - Otherwise, approximation
- **static evaluation function** $e(h)$
  - returns estimate of $u(h)$

H = {nonterminal nodes}

Z = {terminal nodes}

$\rho(h)$ = the player to move at h

$\chi(h)$ = {available actions at h}

$\sigma(h,a)$ = child of h produced by a

$\mathbf{u}(h)$ = utility profile for h

$\mathbf{v}[i]$ = i'th element of $\mathbf{v}$

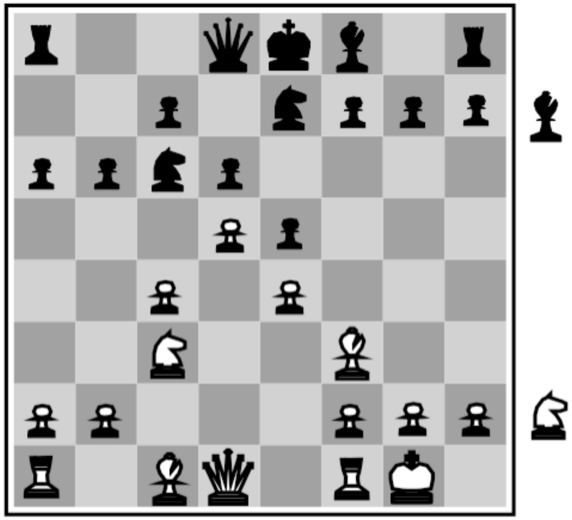
function LD-minimax(h,d)
  if $h \in Z$ then return $u(h)$
  else if $d = 0$ then return $e(h)$
  else if $\rho(h)$ = Max then
    return $\max_{a \in \chi(h)}$LD-minimax($\sigma(h,a)$), d–1)
  else return $\min_{a \in \chi(h)}$LD-minimax($\sigma(h,a)$), d–1)

function LD-minimax-choice(h,d)
  if $h \in Z$ or $d = 0$ then return *error*
  else if $\rho(h)$ = Max then
    return $\arg\max_{a \in \chi(h)}$LD-minimax($\sigma(h,a)$), d–1)
  else return $\arg\min_{a \in \chi(h)}$LD-minimax($\sigma(h,a)$), d–1)

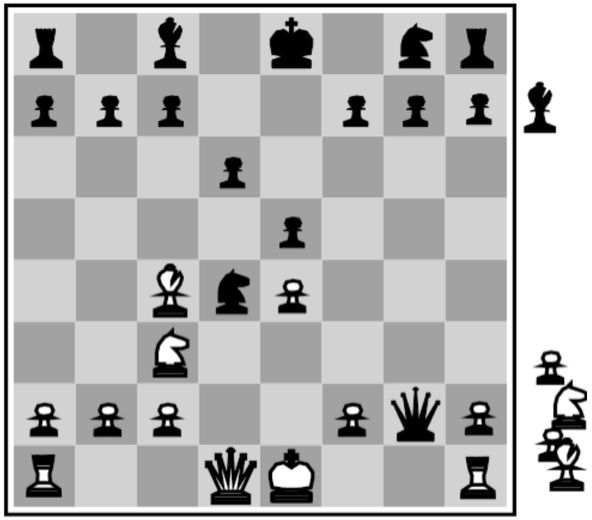
- LD-minimax-choice returns an action
- Call it again at every move
  - Why?

Evaluation Functions

- $e(h)$ is often a weighted sum of features
  - $e(h) = c_1 f_1(h) + c_2 f_2(h) + \ldots + c_n f_n(h)$
- E.g., chess
  - $c_1 \times$ material $+ \, c_2 \times$ mobility $+ \, c_3 \times$ king safety $+ \, c_4 \times$ center control + ...
    - material = 1×(your pawns − opponent's pawns) + 3×(your knights − opponent's knights) + …

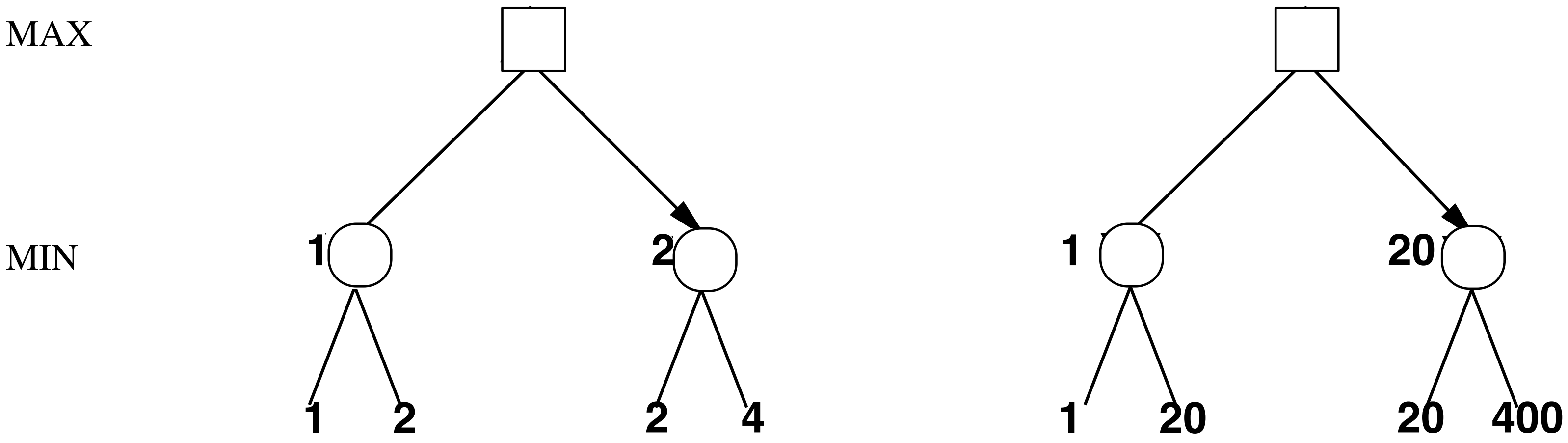


Black to move
White slightly better



White to move
Black winning

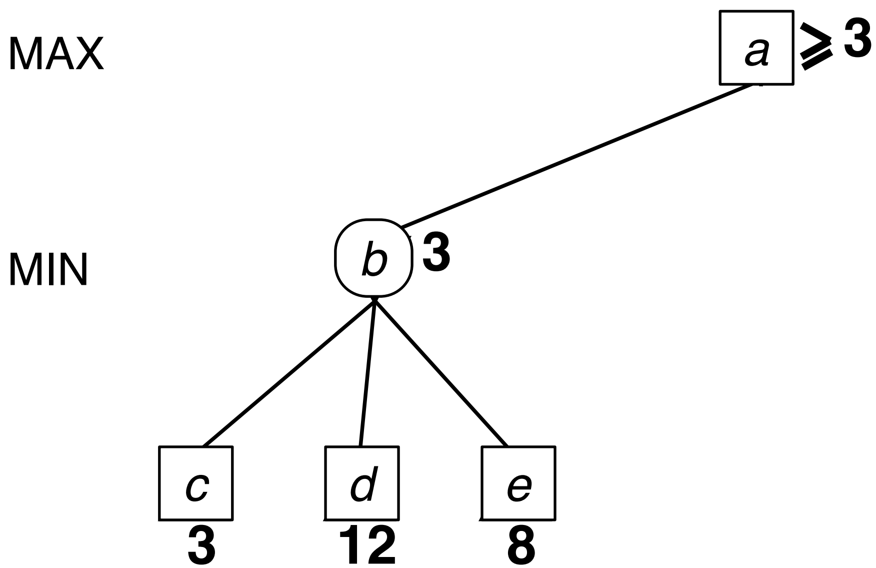
Exact Values for *e* Don't Matter



- Behavior is preserved under any **monotonic** transformation of *e*
  - Only the order matters

Pruning

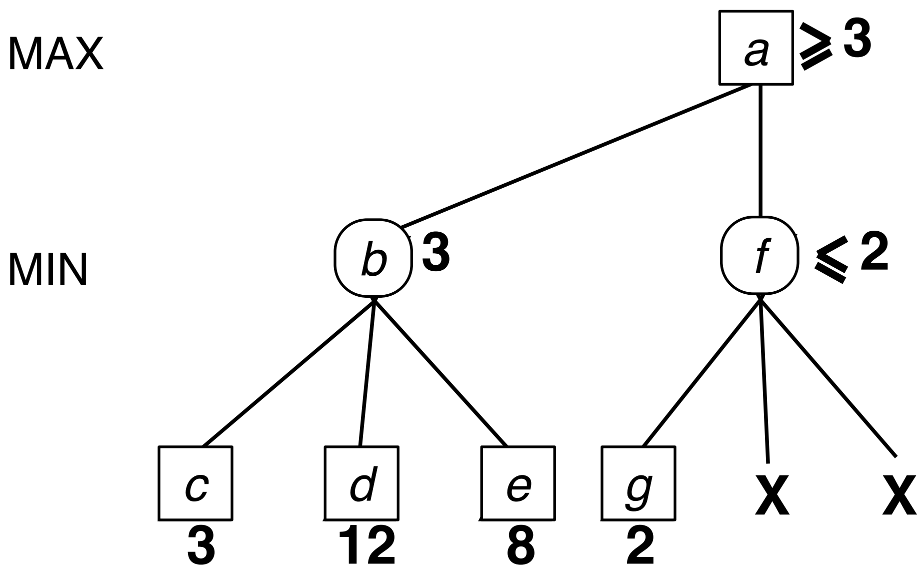
- Minimax and LD-minimax examine nodes that don't need to be examined



function LD-minimax(h,d)
  if $h \in Z$ then return $u(h)$
  else if $d = 0$ then return $e(h)$
  else if $\rho(h)$ = Max then
    $v^* \leftarrow -\infty$
    for every $a \in \chi(h)$ do
      v $\leftarrow$ LD-minimax($\sigma(h,a)$), d–1)
      if $v^* < v$ then $v^* \leftarrow v$
  else
    $v^* \leftarrow \infty$
    for every $a \in \chi(h)$ do
      v $\leftarrow$ LD-minimax($\sigma(h,a)$), d–1)
      if $v^* > v$ then $v^* \leftarrow v$
  return v

Pruning

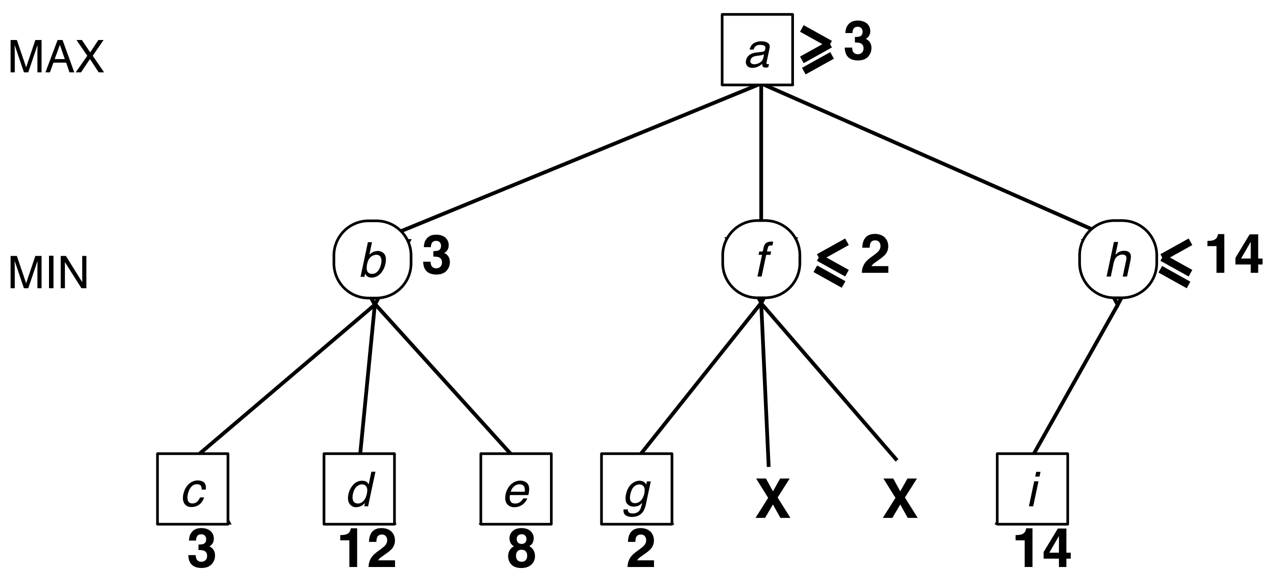
- For Max, b is better than f
  - If Max is rational then Max will never choose f
  - Don't need to examine any more nodes below f



function LD-minimax(h,d)
  if $h \in Z$ then return $u(h)$
  else if $d = 0$ then return $e(h)$
  else if $\rho(h)$ = Max then
    $v^* \leftarrow -\infty$
    for every $a \in \chi(h)$ do
      v ← LD-minimax($\sigma(h,a)), d{-}1$)
      if $v^* < v$ then $v^* \leftarrow v$
  else
    $v^* \leftarrow \infty$
    for every $a \in \chi(h)$ do
      v ← LD-minimax($\sigma(h,a)), d{-}1$)
      if $v^* > v$ then $v^* \leftarrow v$
  return v

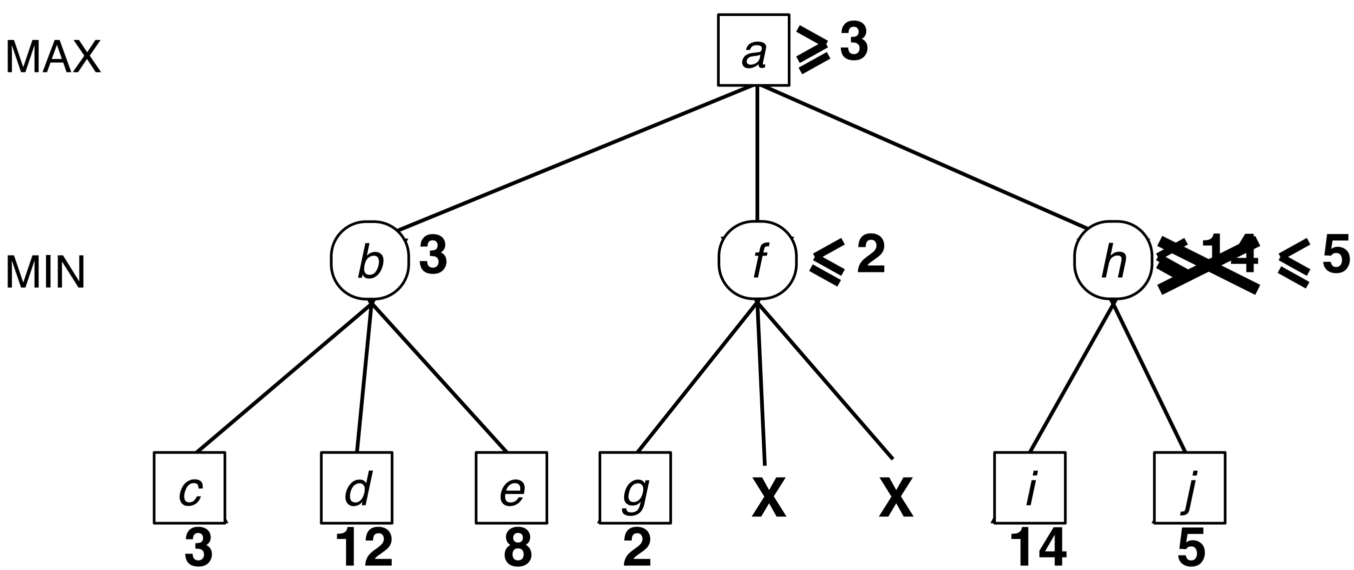
Pruning

- Don't know whether *h* is better or worse than *b*



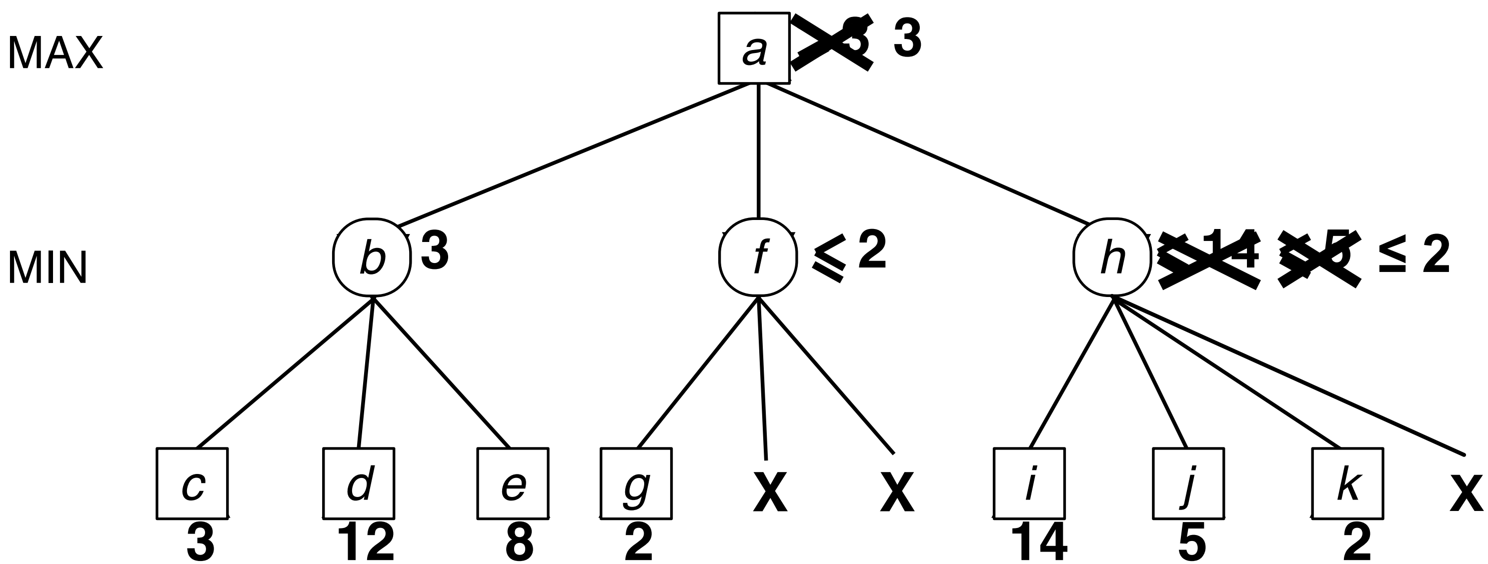
Pruning

- Still don't know whether *h* is better or worse than *b*



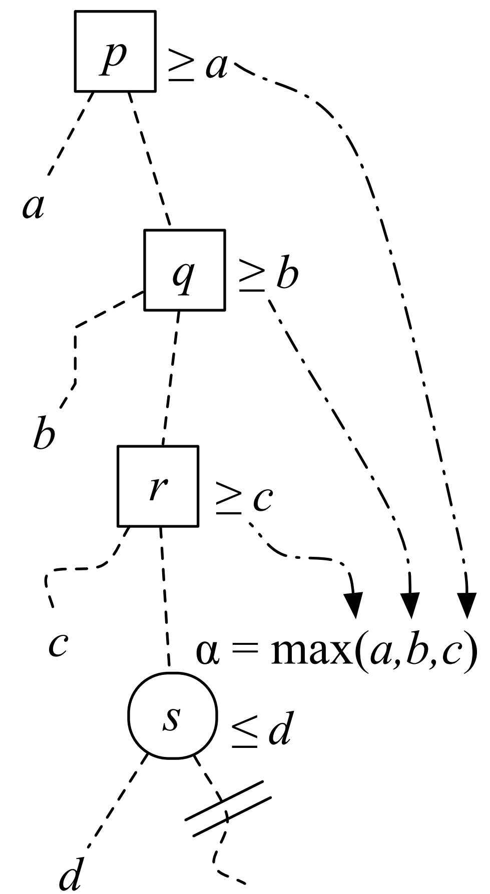
Pruning

- h is worse than b
  - Don't need to examine any more nodes below h
- $v(a) = 3$



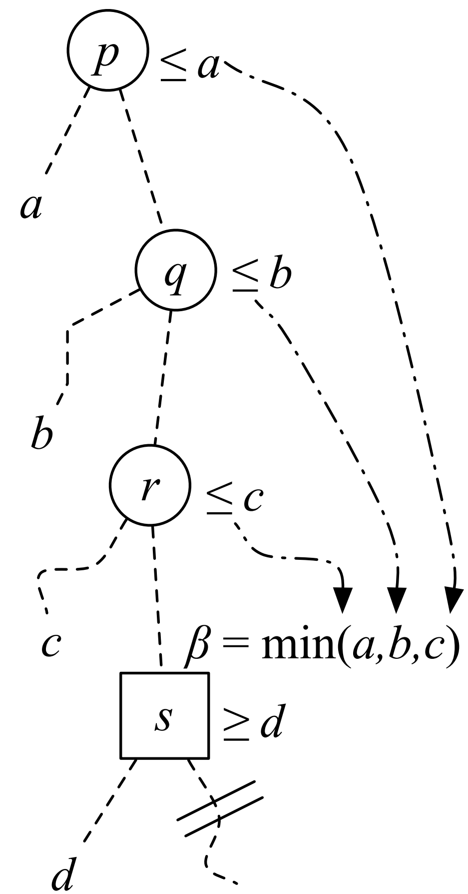
Alpha Cutoff

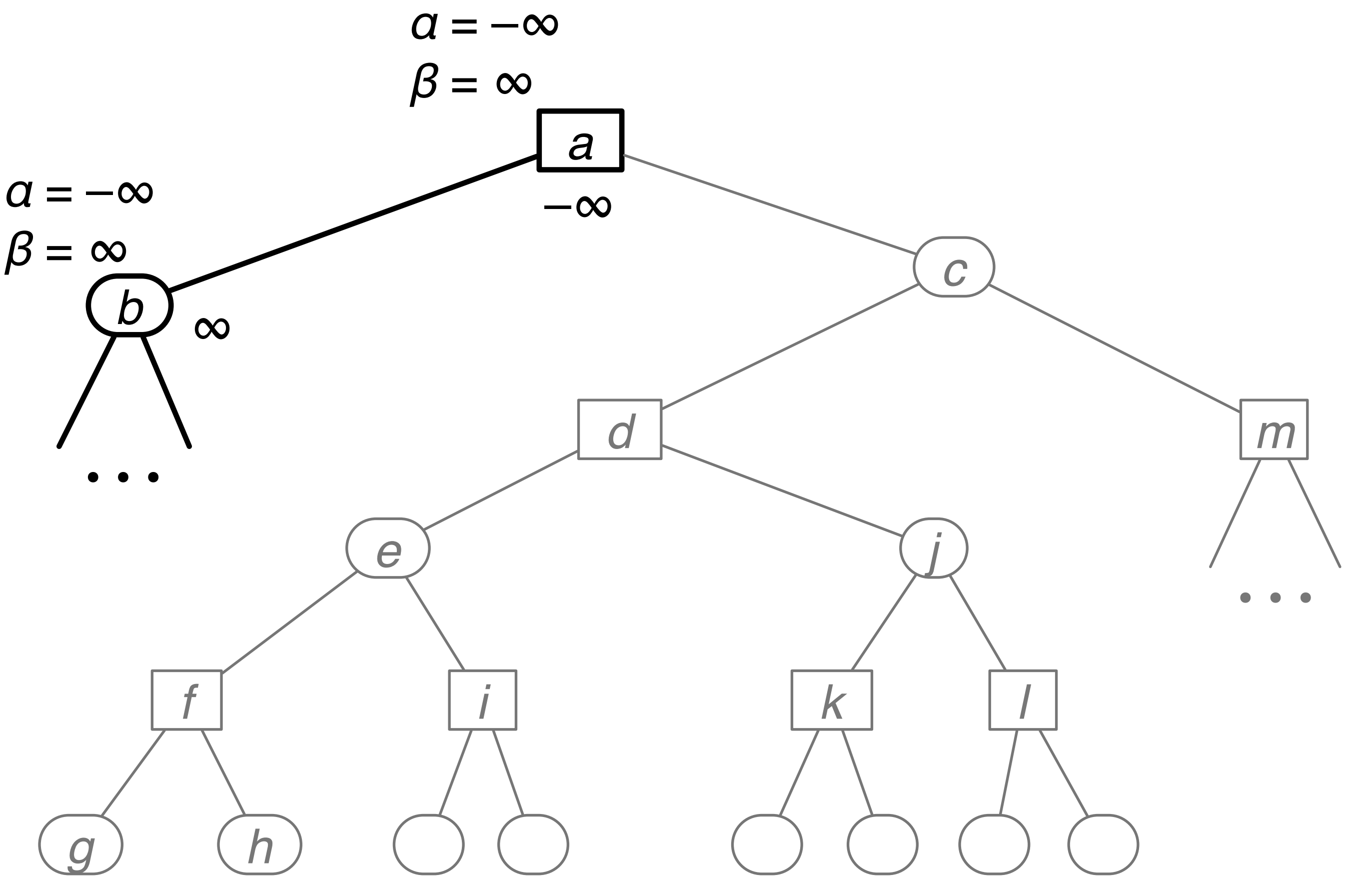
- At Min nodes, let α = largest lower bound on ancestors
  - At s, $\alpha = \max(a,b,c)$
- Suppose $d < \alpha$
  - If the game ever reaches s, Max's payoff will be $< \alpha$
- To get payoff $\geq \alpha$,
  - Max will move elsewhere at p, q, or r
  - The game won't ever reach s
- What if $d = \alpha$?



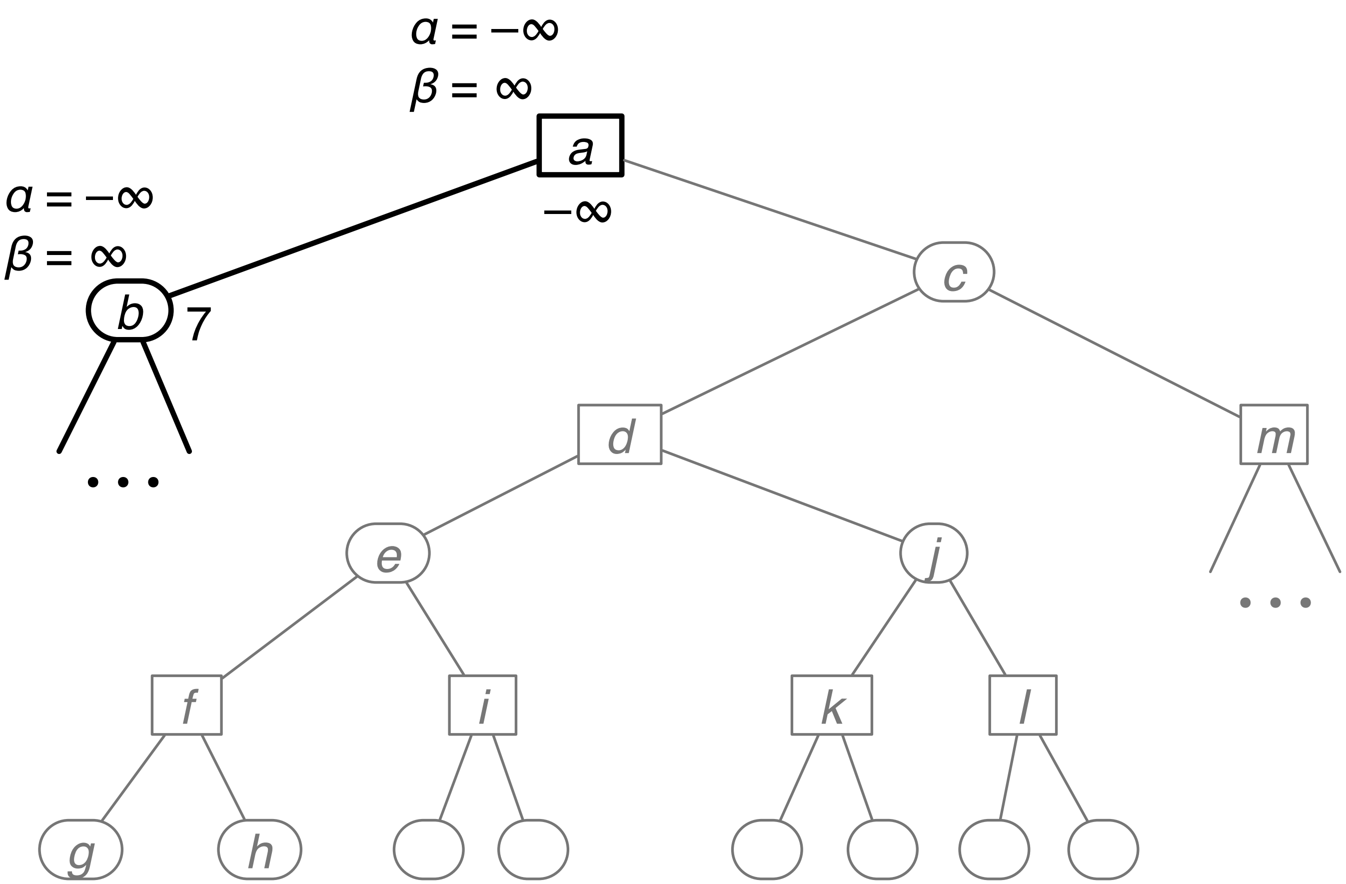
Beta Cutoff

- At Max nodes, let β = smallest upper bound on ancestors
  - At s, $\beta = \min(a,b,c)$
- Suppose $d > \beta$
  - If the game ever reaches s, Max's payoff will be $> \beta$
- To make Max's payoff $\leq \beta$
  - Min will move elsewhere at p, q, or r
  - The game won't ever reach s
- What if $d = \beta$?

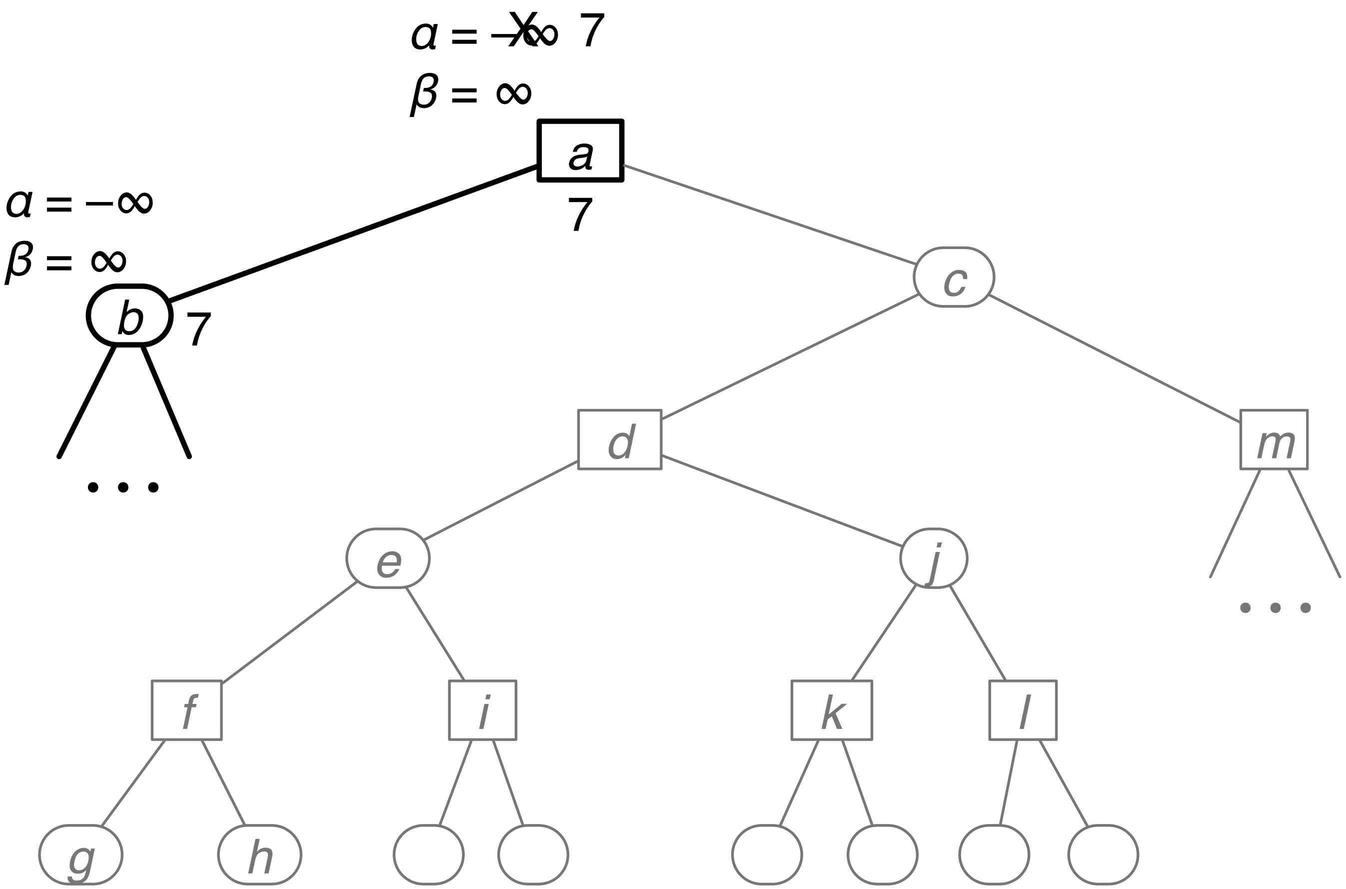




function Alpha-Beta(h, d, α, β)
  if $h \in Z$ then return $u(h)$
  else if $d = 0$ then return $e(h)$
  else if $\rho(h) = \text{Max}$ then
    $v \leftarrow -\infty$
    for every $a \in \chi(h)$ do
      $v \leftarrow \max(v,$Alpha-Beta$(\sigma(h,a),d-1,\alpha,\beta)$
      if $v \geq \beta$ then return v // β cutoff
      else $\alpha \leftarrow \max(\alpha, v)$
    return v
  else
    $v \leftarrow \infty$
    for every $a \in \chi(h)$ do
      $v \leftarrow \min(v,$Alpha-Beta$(\sigma(h,a),d-1,\alpha,\beta)$
      if $v \leq \alpha$ then return v // α cutoff
      else $\beta \leftarrow \min(\beta,v)$
    return v




```
function Alpha-Beta(h, d, α, β)
    if h ∈ Z then return u(h)
    else if d = 0 then return e(h)
    else if ρ(h) = Max then
      v ← −∞
      for every a ∈ χ(h) do
          v ← max(v,Alpha-Beta(σ(h,a),d−1,α,β))
          if v ≥ β then return v       // β cutoff
          else α ← max(α, v)
      return v
    else
      v ← ∞
      for every a ∈ χ(h) do
          v ← min(v,Alpha-Beta(σ(h,a),d−1,α,β))
          if v ≤ α then return v       // α cutoff
          else β ← min(β,v)
      return v
```



function Alpha-Beta(h, d, α, β)

  if $h \in Z$ then return $u(h)$

  else if $d = 0$ then return $e(h)$

  else if $\rho(h) = \text{Max}$ then

    $v \leftarrow -\infty$

    for every $a \in \chi(h)$ do

      $v \leftarrow \max(v,$Alpha-Beta$(\sigma(h,a), d-1, \alpha, \beta))$

      if $v \geq \beta$ then return v // β cutoff

      else $\alpha \leftarrow \max(\alpha, v)$

    return v

  else

    $v \leftarrow \infty$

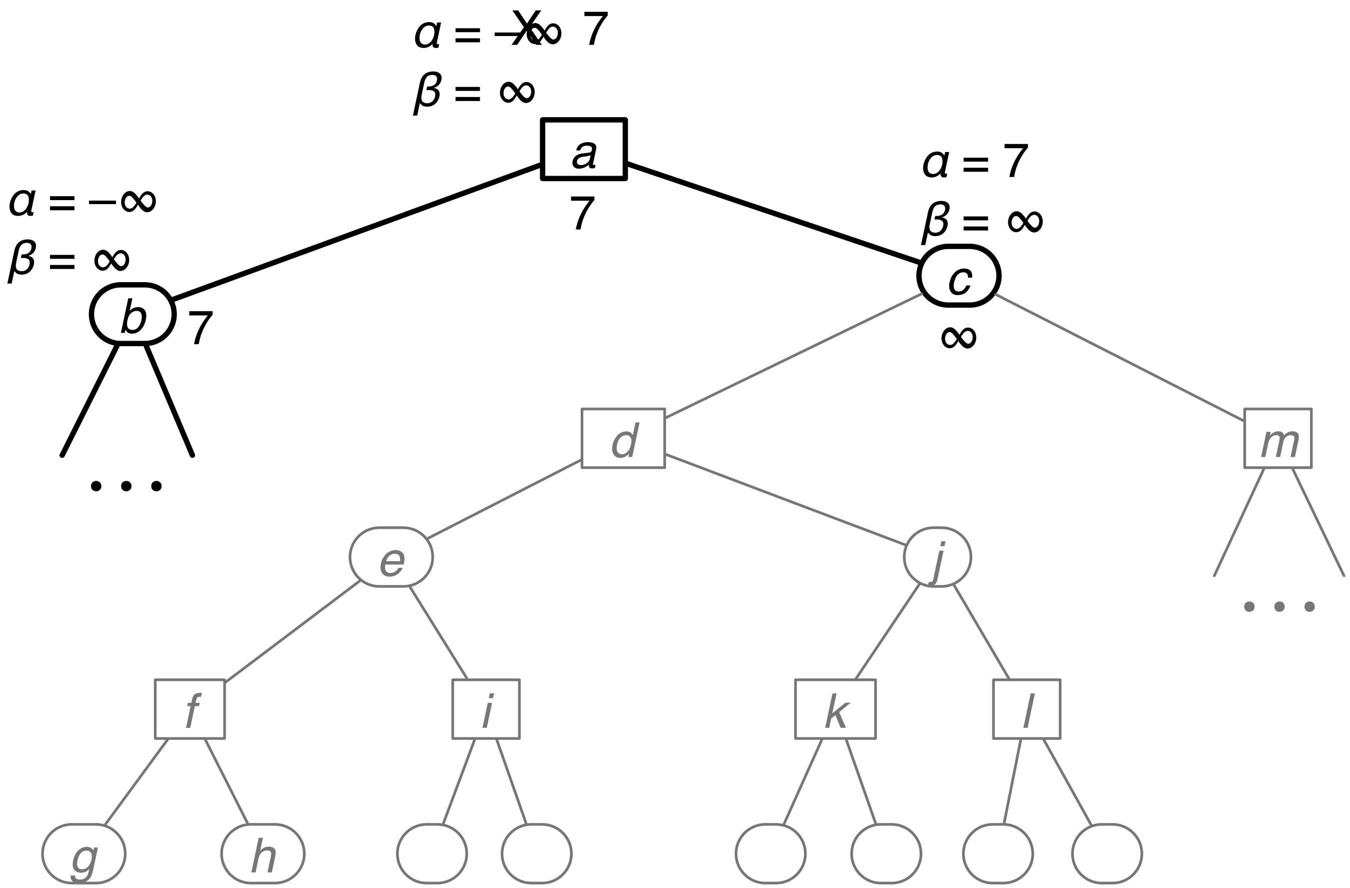
    for every $a \in \chi(h)$ do

      $v \leftarrow \min(v,$Alpha-Beta$(\sigma(h,a), d-1, \alpha, \beta))$

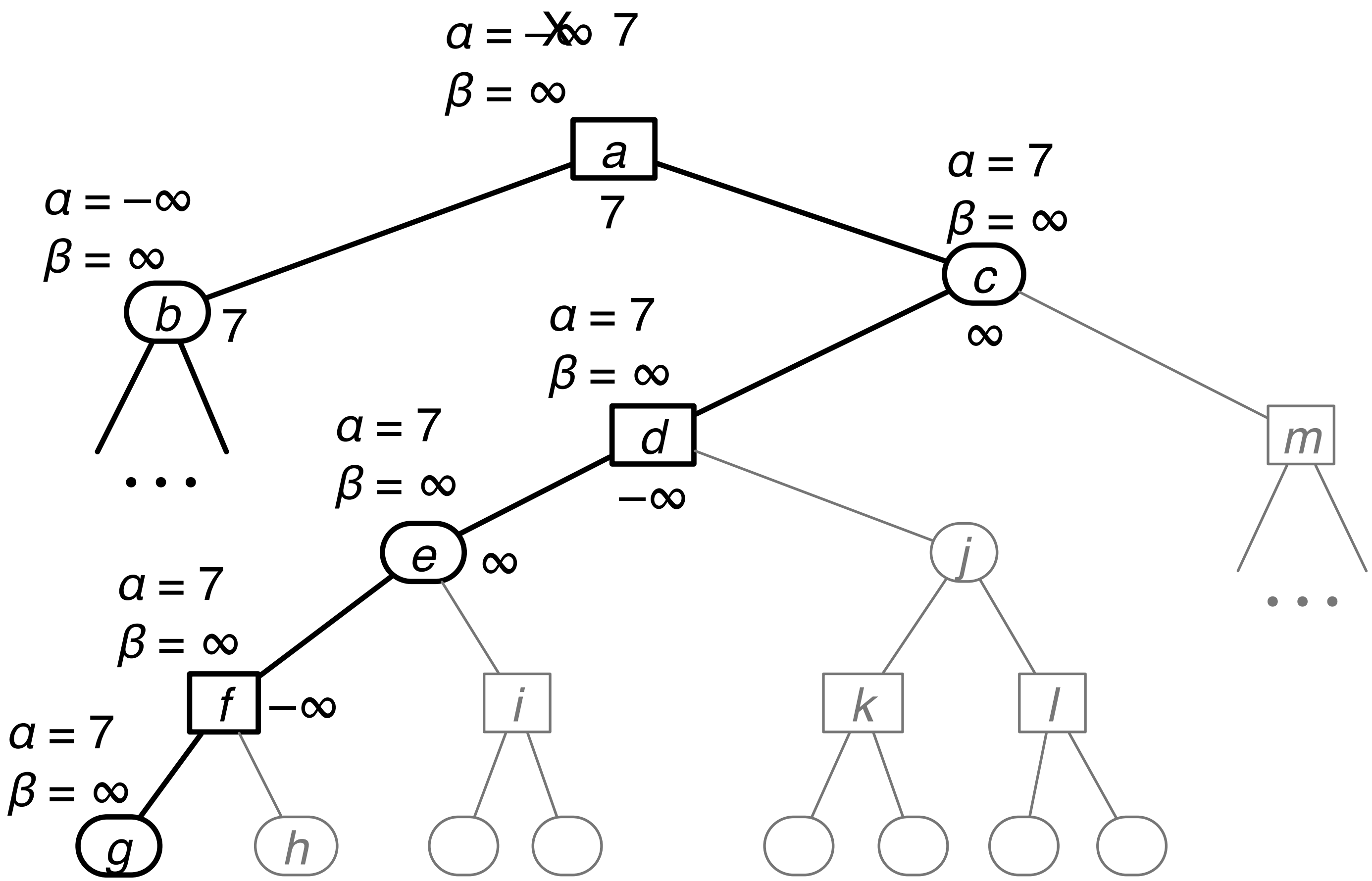
      if $v \leq \alpha$ then return v // α cutoff

      else $\beta \leftarrow \min(\beta, v)$

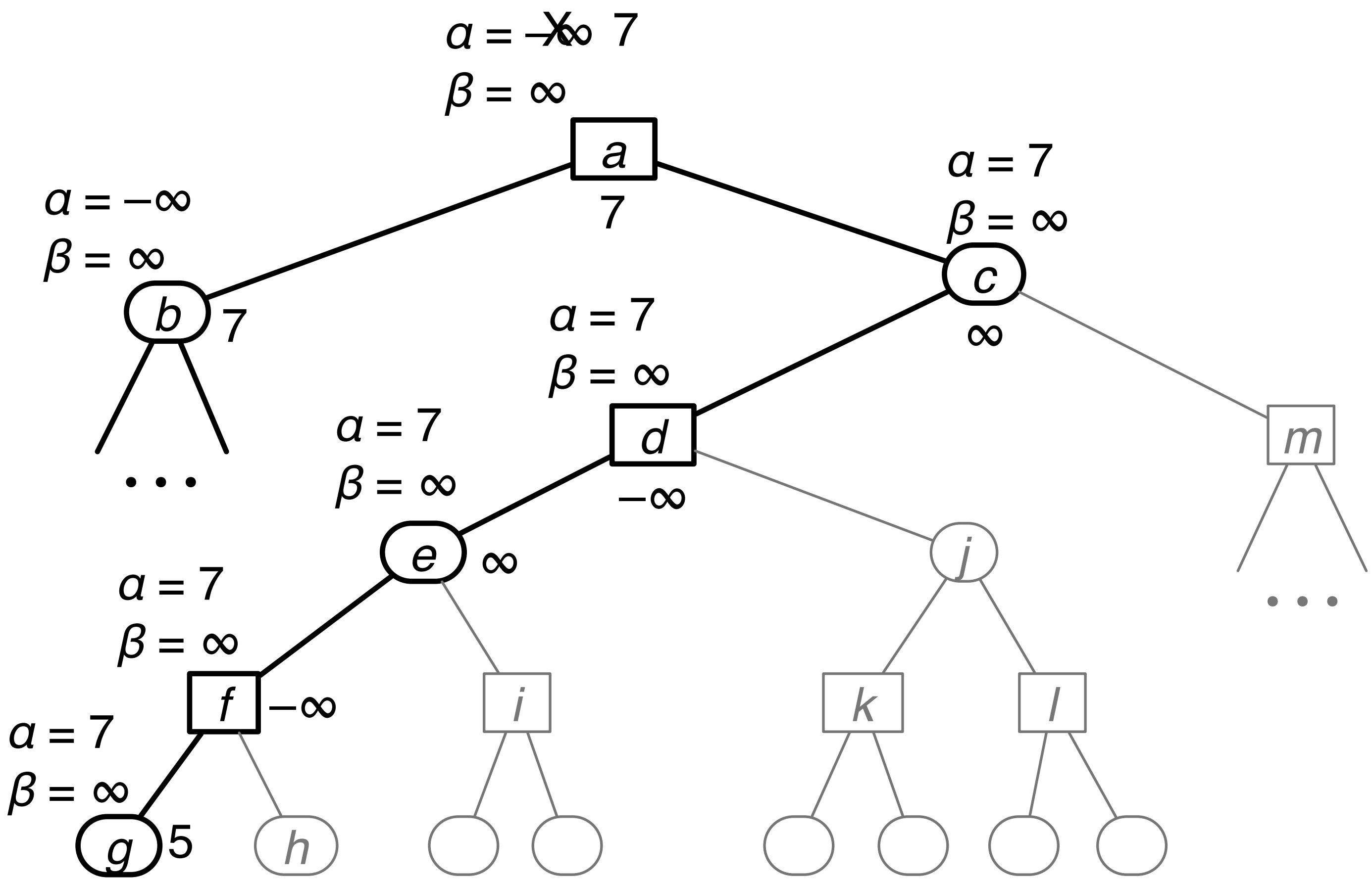
    return v



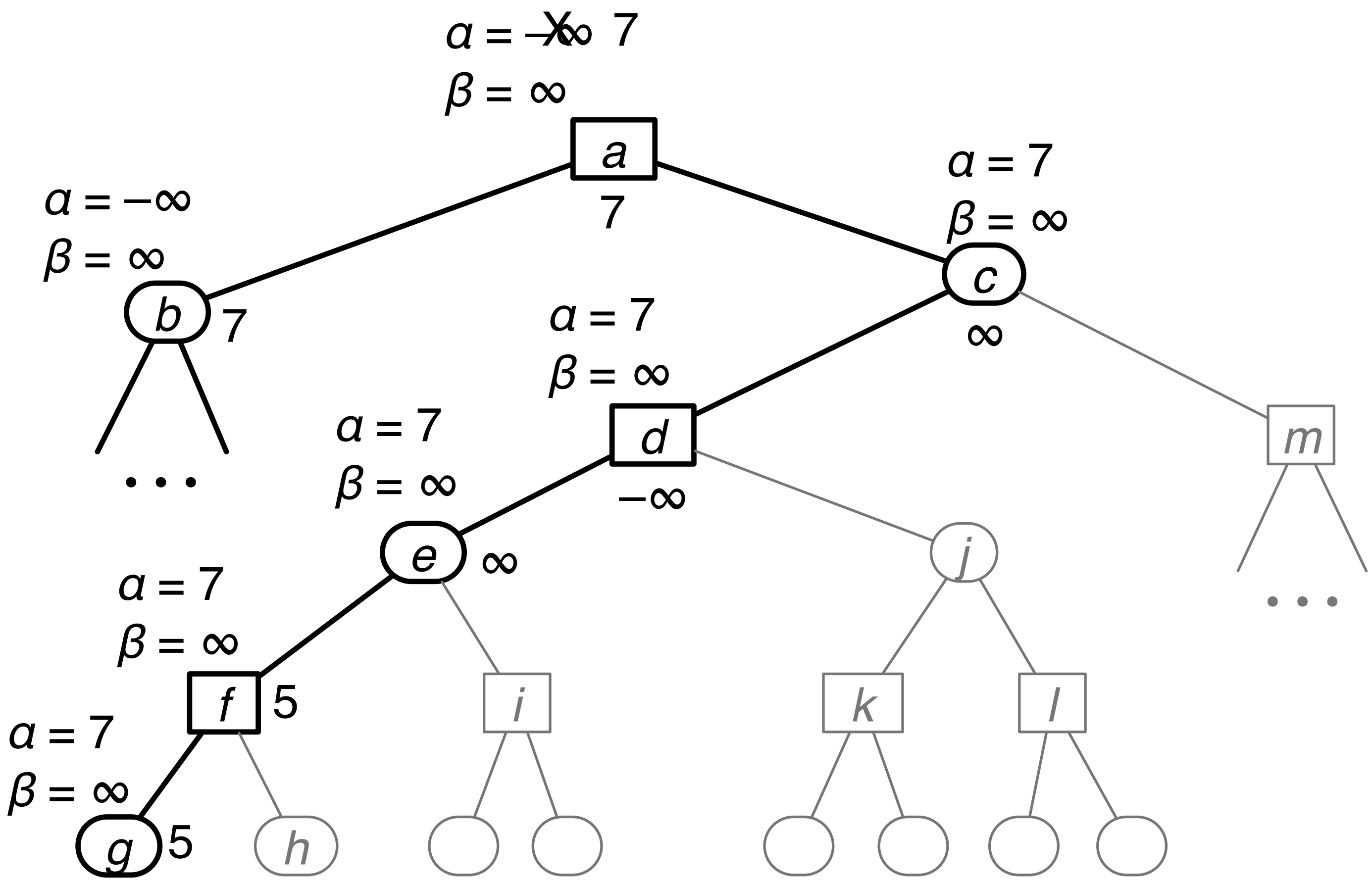

```
function Alpha-Beta(h, d, α, β)
    if h ∈ Z then return u(h)
    else if d = 0 then return e(h)
    else if ρ(h) = Max then
      v ← −∞
      for every a ∈ χ(h) do
          v ← max(v,Alpha-Beta(σ(h,a),d−1,α,β)
          if v ≥ β then return v        // β cutoff
          else α ← max(α, v)
      return v
    else
      v ← ∞
      for every a ∈ χ(h) do
          v ← min(v,Alpha-Beta(σ(h,a),d−1,α,β)
          if v ≤ α then return v        // α cutoff
          else β ← min(β,v)
      return v
```



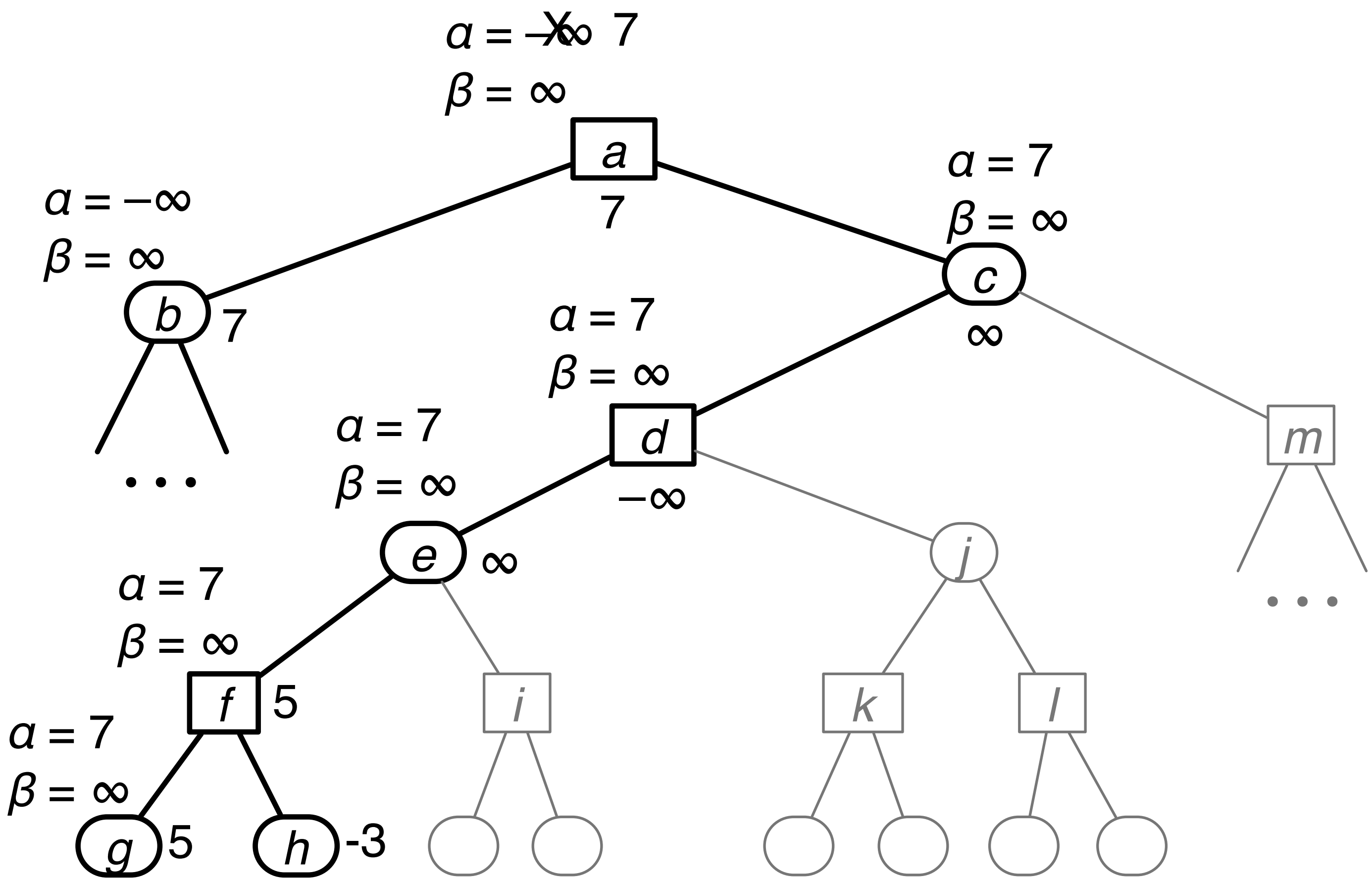
function Alpha-Beta(h, d, α, β)
    if $h \in Z$ then return $u(h)$
    else if $d = 0$ then return $e(h)$
    else if $\rho(h) = \text{Max}$ then
        $v \leftarrow -\infty$
        for every $a \in \chi(h)$ do
            $v \leftarrow \max(v, \text{Alpha-Beta}(\sigma(h,a), d-1, \alpha, \beta))$
            if $v \geq \beta$ then return v // β cutoff
            else $\alpha \leftarrow \max(\alpha, v)$
        return v
    else
        $v \leftarrow \infty$
        for every $a \in \chi(h)$ do
            $v \leftarrow \min(v, \text{Alpha-Beta}(\sigma(h,a), d-1, \alpha, \beta))$
            if $v \leq \alpha$ then return v // α cutoff
            else $\beta \leftarrow \min(\beta, v)$
        return v



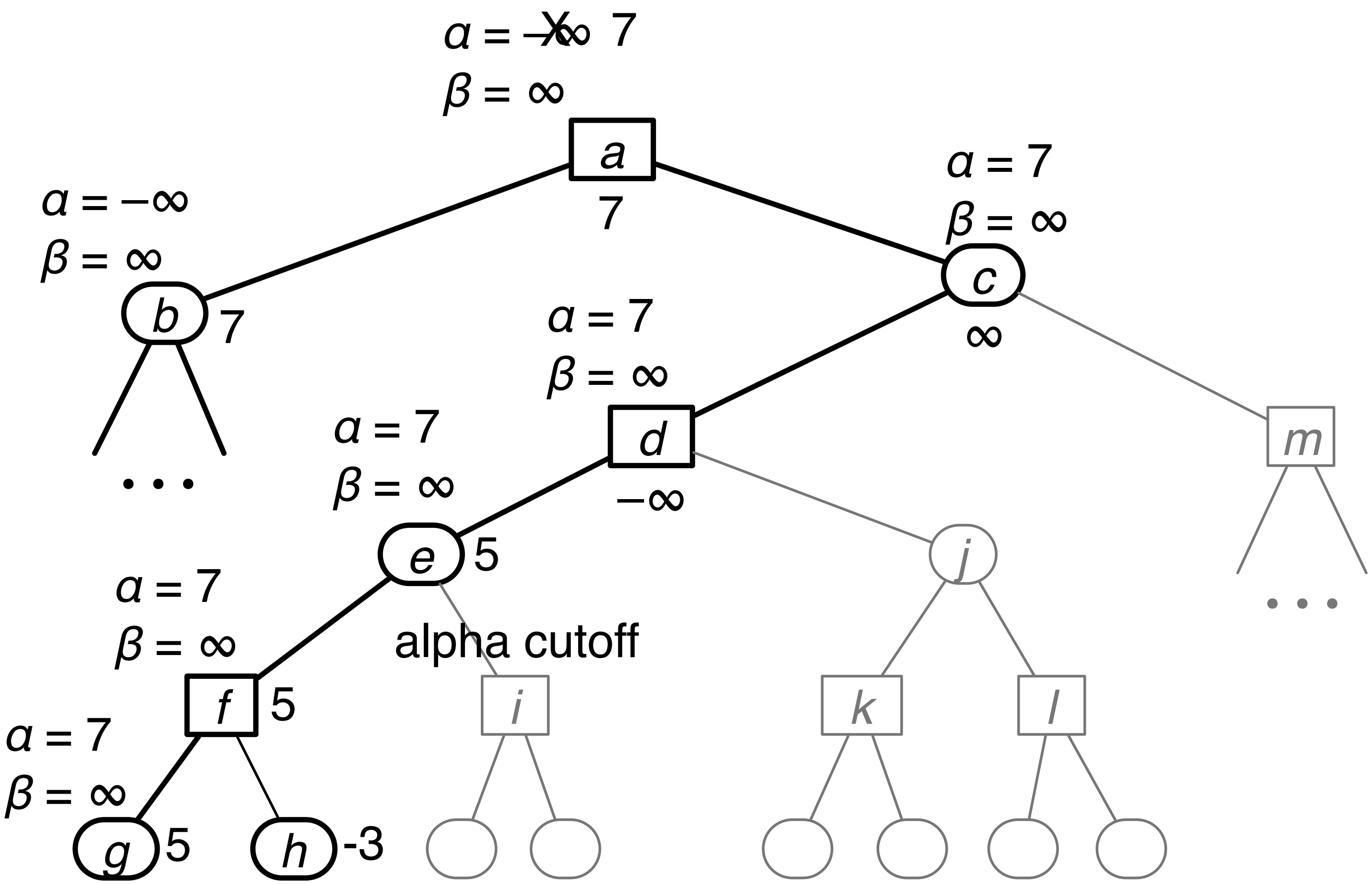

```
function Alpha-Beta(h, d, α, β)
    if h ∈ Z then return u(h)
    else if d = 0 then return e(h)
    else if ρ(h) = Max then
        v ← −∞
        for every a ∈ χ(h) do
            v ← max(v,Alpha-Beta(σ(h,a),d−1,α,β)
            if v ≥ β then return v        // β cutoff
            else α ← max(α, v)
        return v
    else
        v ← ∞
        for every a ∈ χ(h) do
            v ← min(v,Alpha-Beta(σ(h,a),d−1,α,β)
            if v ≤ α then return v        // α cutoff
            else β ← min(β,v)
        return v
```



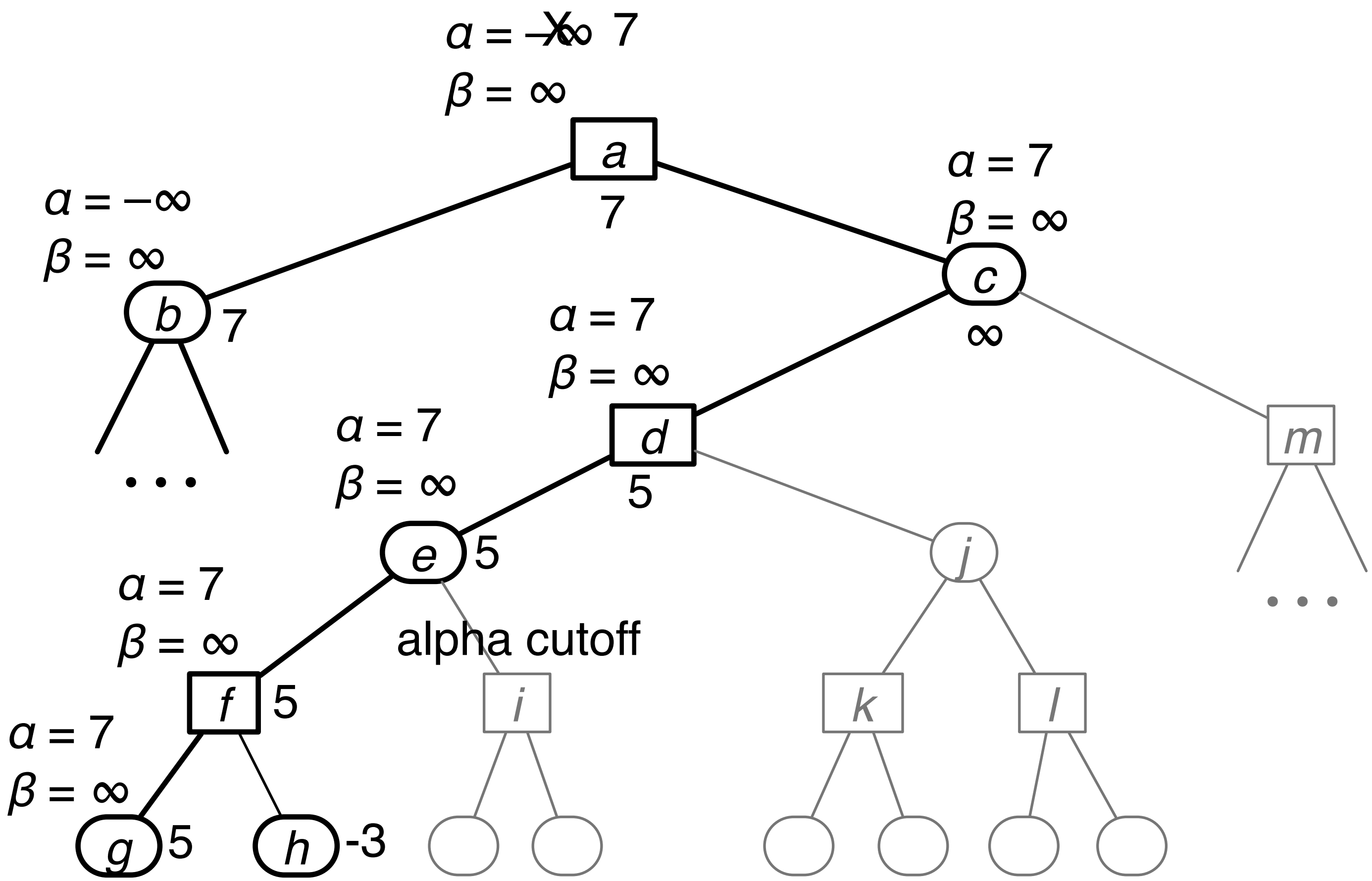

```
function Alpha-Beta(h, d, α, β)
    if h ∈ Z then return u(h)
    else if d = 0 then return e(h)
    else if ρ(h) = Max then
      v ← −∞
      for every a ∈ χ(h) do
          v ← max(v,Alpha-Beta(σ(h,a),d−1,α,β))
          if v ≥ β then return v        // β cutoff
          else α ← max(α, v)
      return v
    else
      v ← ∞
      for every a ∈ χ(h) do
          v ← min(v,Alpha-Beta(σ(h,a),d−1,α,β))
          if v ≤ α then return v        // α cutoff
          else β ← min(β,v)
      return v
```



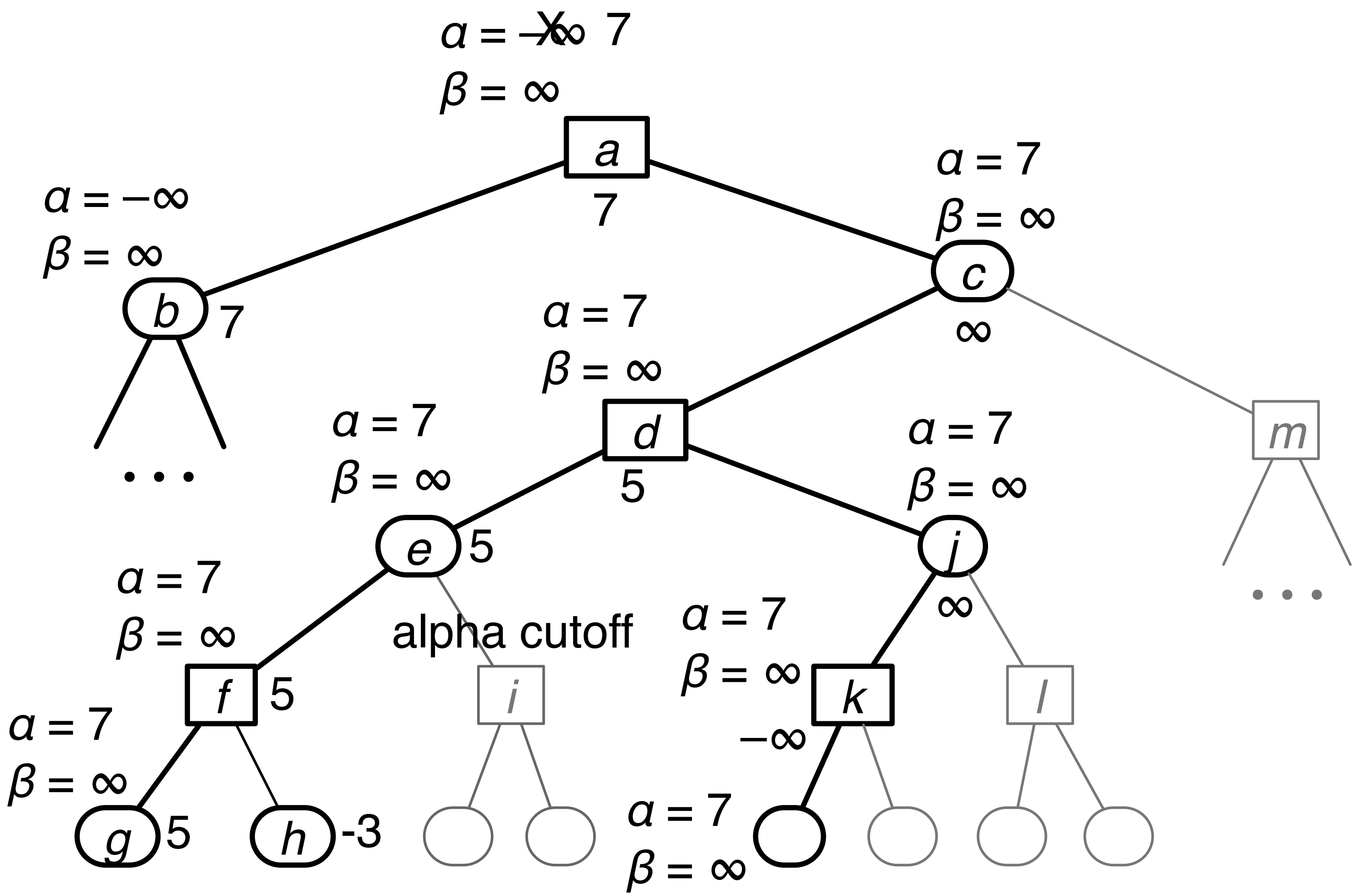
function Alpha-Beta(h, d, α, β)
  if $h \in Z$ then return $u(h)$
  else if $d = 0$ then return $e(h)$
  else if $\rho(h)$ = Max then
    $v \leftarrow -\infty$
    for every $a \in \chi(h)$ do
      $v \leftarrow \max(v,$Alpha-Beta$(\sigma(h,a), d-1, \alpha, \beta)$
      if $v \geq \beta$ then return v // β cutoff
      else $\alpha \leftarrow \max(\alpha, v)$
    return v
  else
    $v \leftarrow \infty$
    for every $a \in \chi(h)$ do
      $v \leftarrow \min(v,$Alpha-Beta$(\sigma(h,a), d-1, \alpha, \beta)$
      if $v \leq \alpha$ then return v // α cutoff
      else $\beta \leftarrow \min(\beta, v)$
    return v




```
function Alpha-Beta(h, d, α, β)
    if h ∈ Z then return u(h)
    else if d = 0 then return e(h)
    else if ρ(h) = Max then
      v ← −∞
      for every a ∈ χ(h) do
          v ← max(v,Alpha-Beta(σ(h,a),d−1,α,β)
          if v ≥ β then return v        // β cutoff
          else α ← max(α, v)
      return v
    else
      v ← ∞
      for every a ∈ χ(h) do
          v ← min(v,Alpha-Beta(σ(h,a),d−1,α,β)
          if v ≤ α then return v        // α cutoff
          else β ← min(β,v)
      return v
```




```
function Alpha-Beta(h, d, α, β)
    if h ∈ Z then return u(h)
    else if d = 0 then return e(h)
    else if ρ(h) = Max then
        v ← −∞
        for every a ∈ χ(h) do
            v ← max(v,Alpha-Beta(σ(h,a),d−1,α,β)
            if v ≥ β then return v        // β cutoff
            else α ← max(α, v)
        return v
    else
        v ← ∞
        for every a ∈ χ(h) do
            v ← min(v,Alpha-Beta(σ(h,a),d−1,α,β)
            if v ≤ α then return v        // α cutoff
            else β ← min(β,v)
        return v
```



function Alpha-Beta(h, d, α, β)

  if $h \in Z$ then return $u(h)$

  else if $d = 0$ then return $e(h)$

  else if $\rho(h)$ = Max then

    $v \leftarrow -\infty$

    for every $a \in \chi(h)$ do

      $v \leftarrow \max(v,$Alpha-Beta$(\sigma(h,a),d-1,\alpha,\beta)$

      if $v \geq \beta$ then return v // β cutoff

      else $\alpha \leftarrow \max(\alpha, v)$

    return v

  else

    $v \leftarrow \infty$

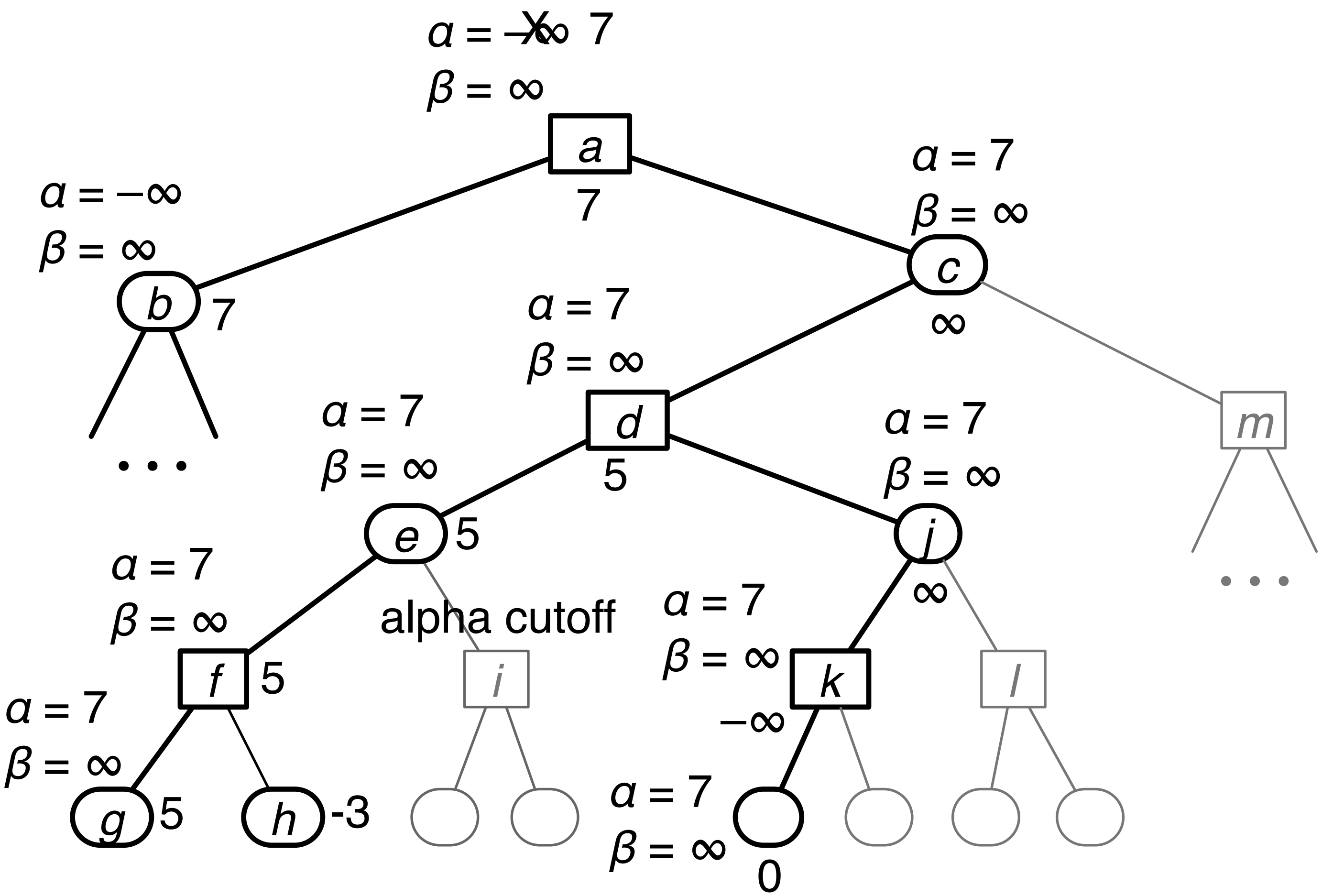
    for every $a \in \chi(h)$ do

      $v \leftarrow \min(v,$Alpha-Beta$(\sigma(h,a),d-1,\alpha,\beta)$

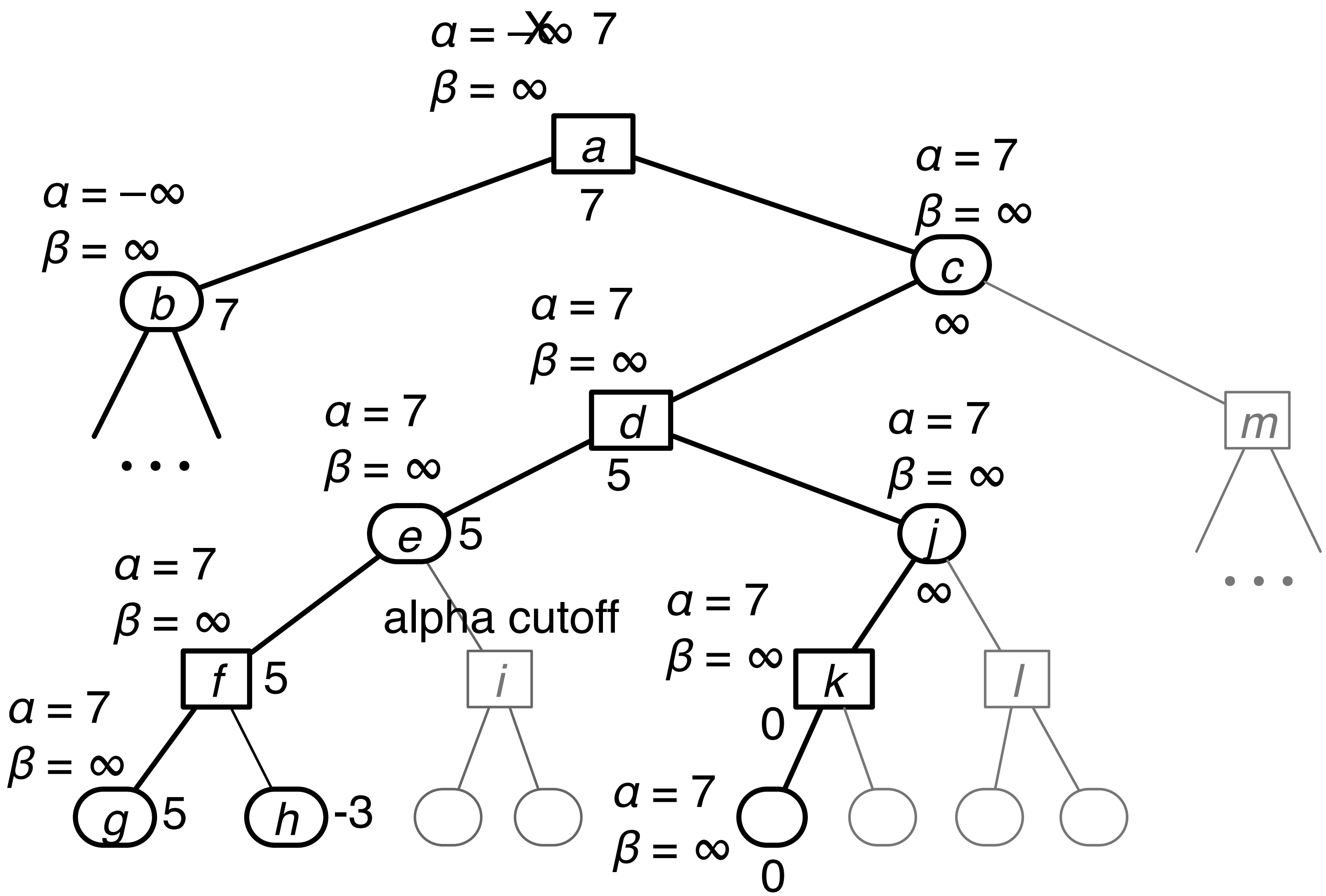
      if $v \leq \alpha$ then return v // α cutoff

      else $\beta \leftarrow \min(\beta, v)$

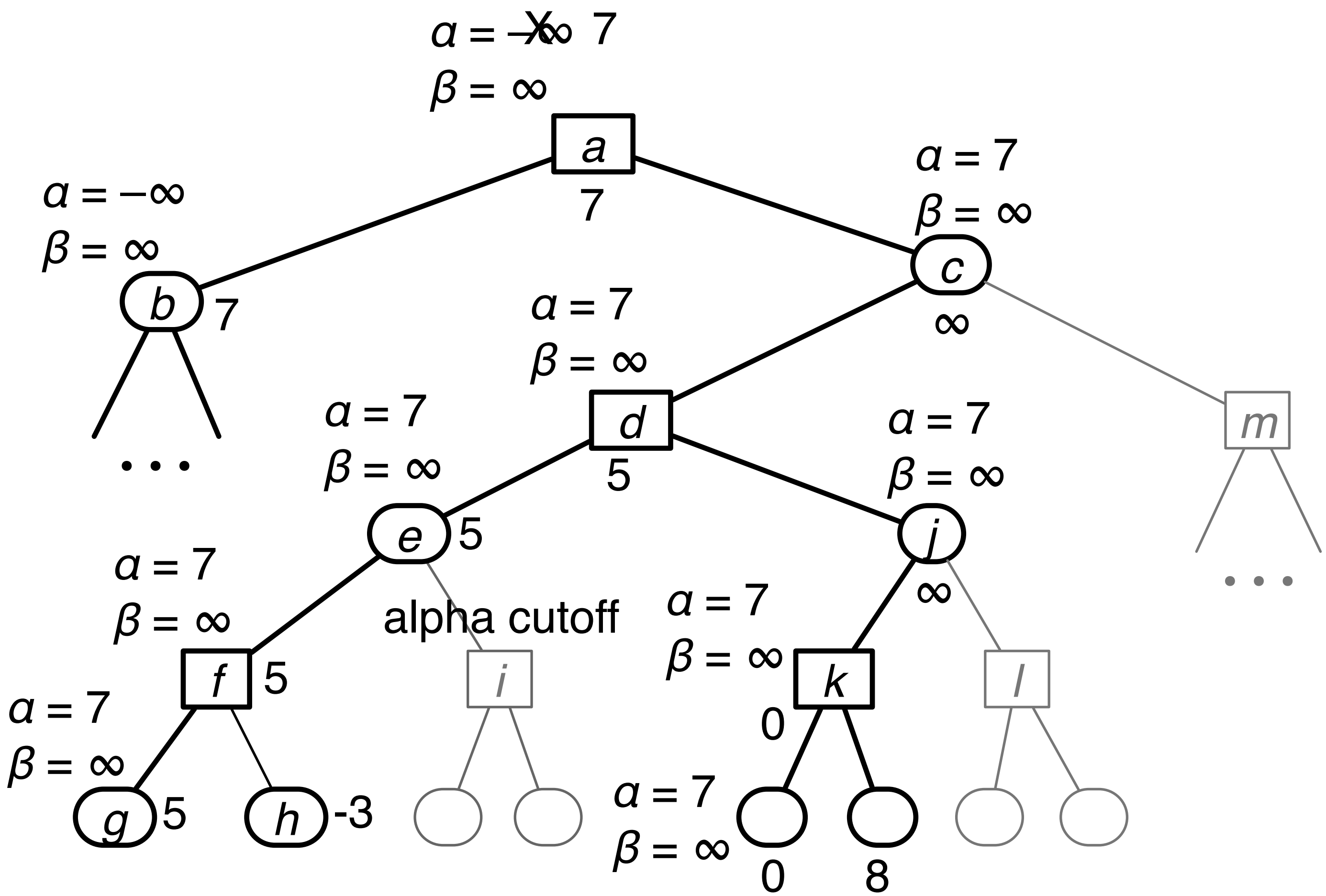
    return v



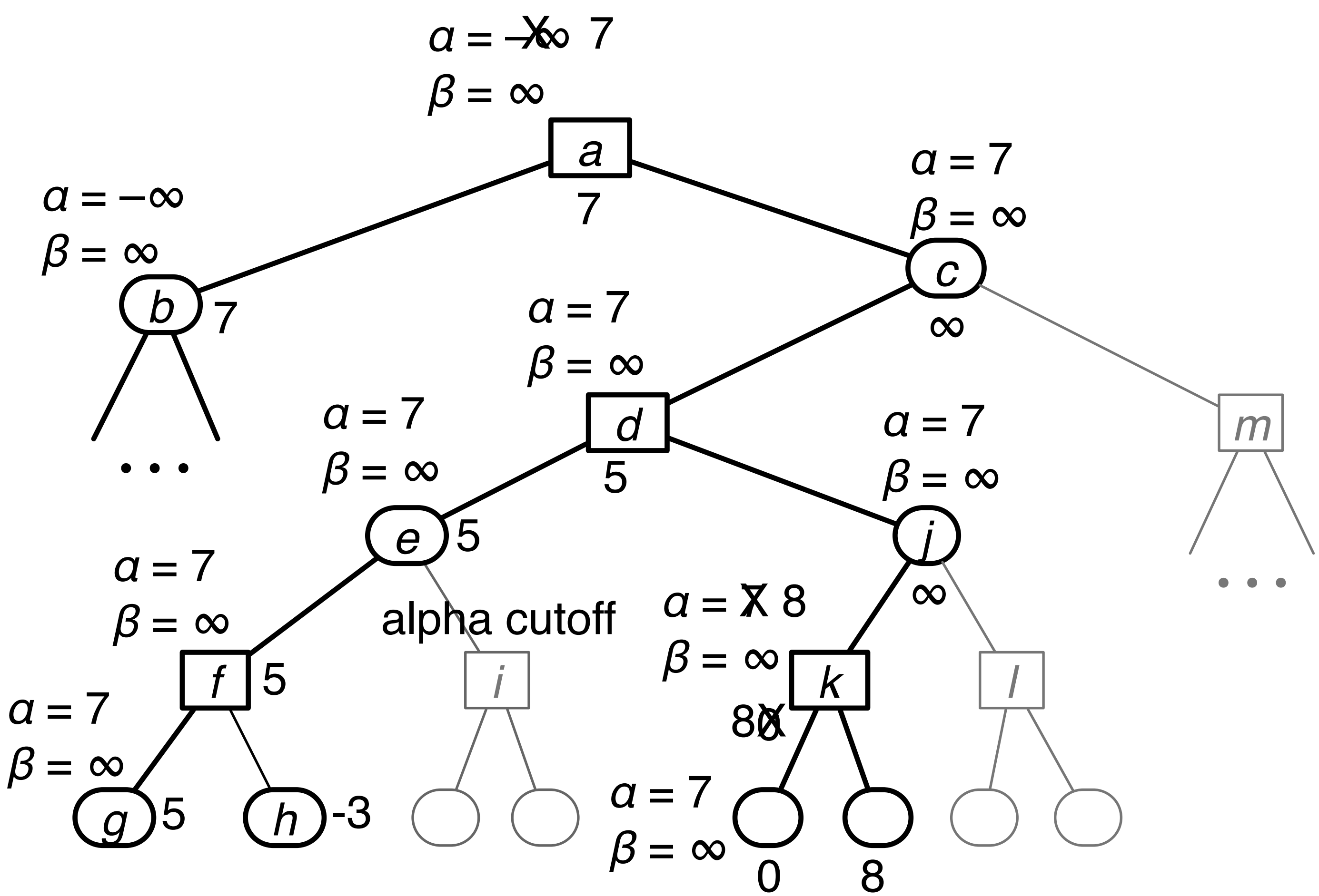

```
function Alpha-Beta(h, d, α, β)
    if h ∈ Z then return u(h)
    else if d = 0 then return e(h)
    else if ρ(h) = Max then
      v ← −∞
      for every a ∈ χ(h) do
          v ← max(v,Alpha-Beta(σ(h,a),d−1,α,β))
          if v ≥ β then return v        // β cutoff
          else α ← max(α, v)
      return v
    else
      v ← ∞
      for every a ∈ χ(h) do
          v ← min(v,Alpha-Beta(σ(h,a),d−1,α,β))
          if v ≤ α then return v        // α cutoff
          else β ← min(β,v)
      return v
```



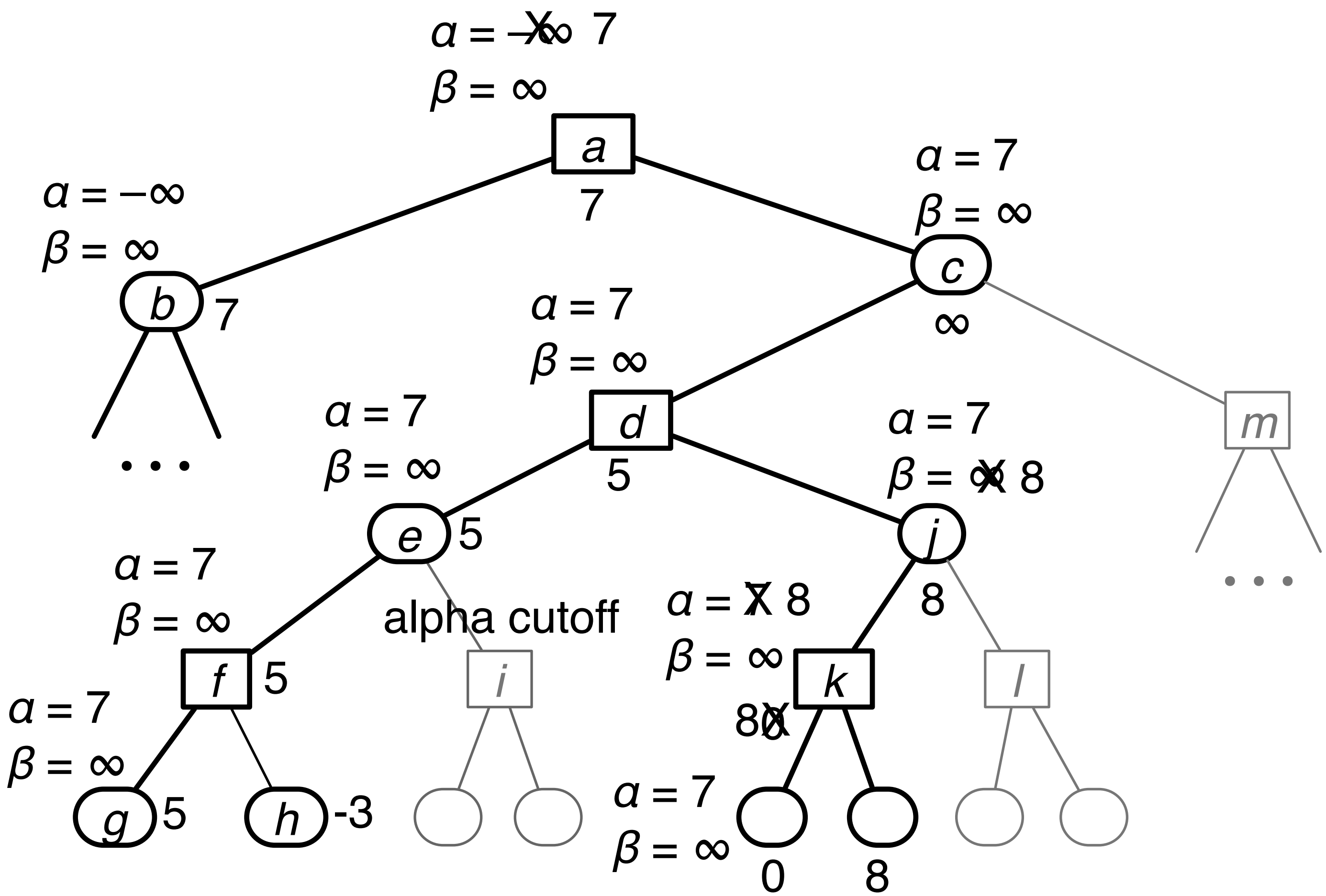

```
function Alpha-Beta(h, d, α, β)
    if h ∈ Z then return u(h)
    else if d = 0 then return e(h)
    else if ρ(h) = Max then
      v ← −∞
      for every a ∈ χ(h) do
          v ← max(v,Alpha-Beta(σ(h,a),d−1,α,β)
          if v ≥ β then return v        // β cutoff
          else α ← max(α, v)
      return v
    else
      v ← ∞
      for every a ∈ χ(h) do
          v ← min(v,Alpha-Beta(σ(h,a),d−1,α,β)
          if v ≤ α then return v        // α cutoff
          else β ← min(β,v)
      return v
```



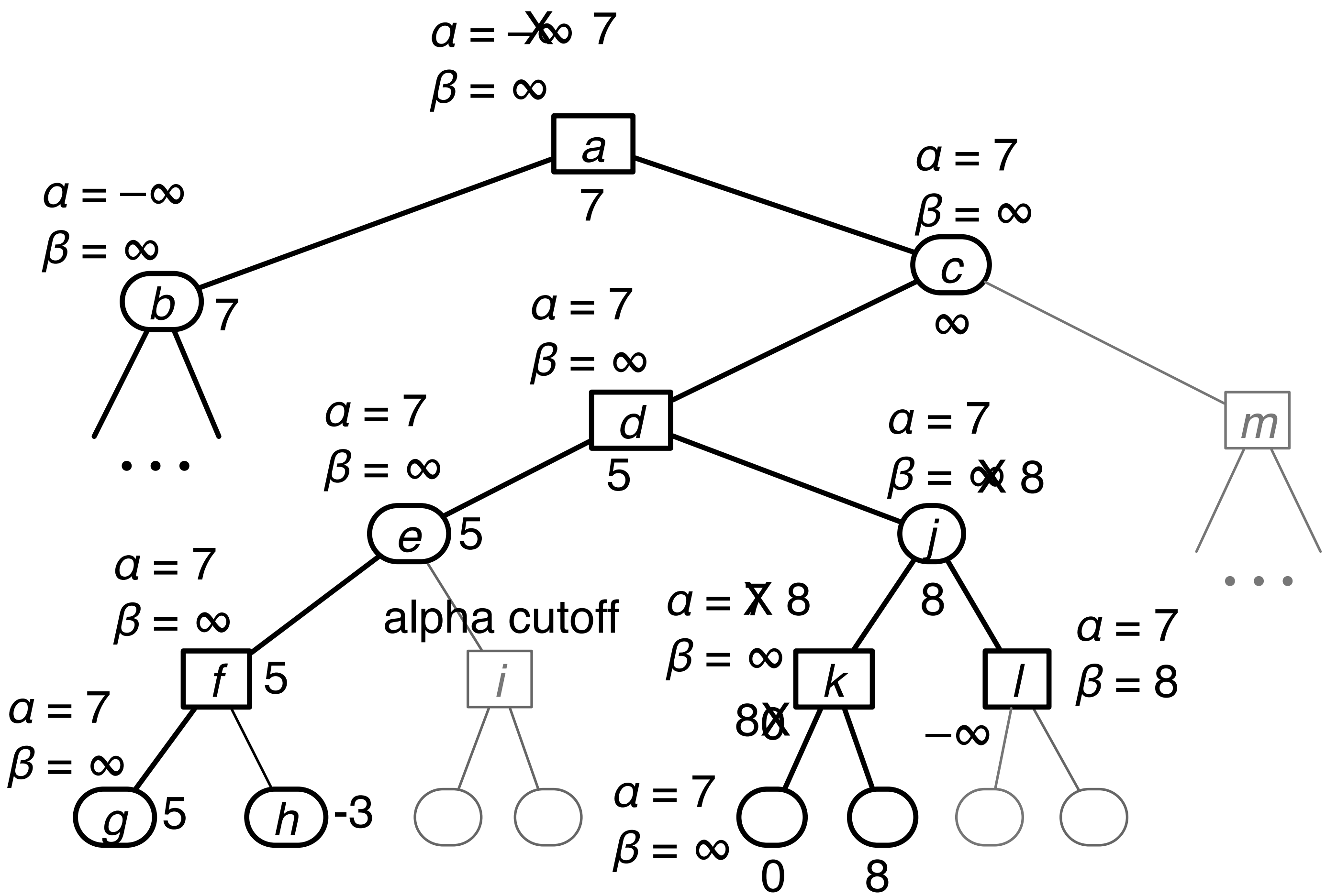

```
function Alpha-Beta(h, d, α, β)
  if h ∈ Z then return u(h)
  else if d = 0 then return e(h)
  else if ρ(h) = Max then
    v ← −∞
    for every a ∈ χ(h) do
      v ← max(v,Alpha-Beta(σ(h,a),d−1,α,β)
      if v ≥ β then return v       // β cutoff
      else α ← max(α, v)
    return v
  else
    v ← ∞
    for every a ∈ χ(h) do
      v ← min(v,Alpha-Beta(σ(h,a),d−1,α,β)
      if v ≤ α then return v       // α cutoff
      else β ← min(β,v)
    return v
```



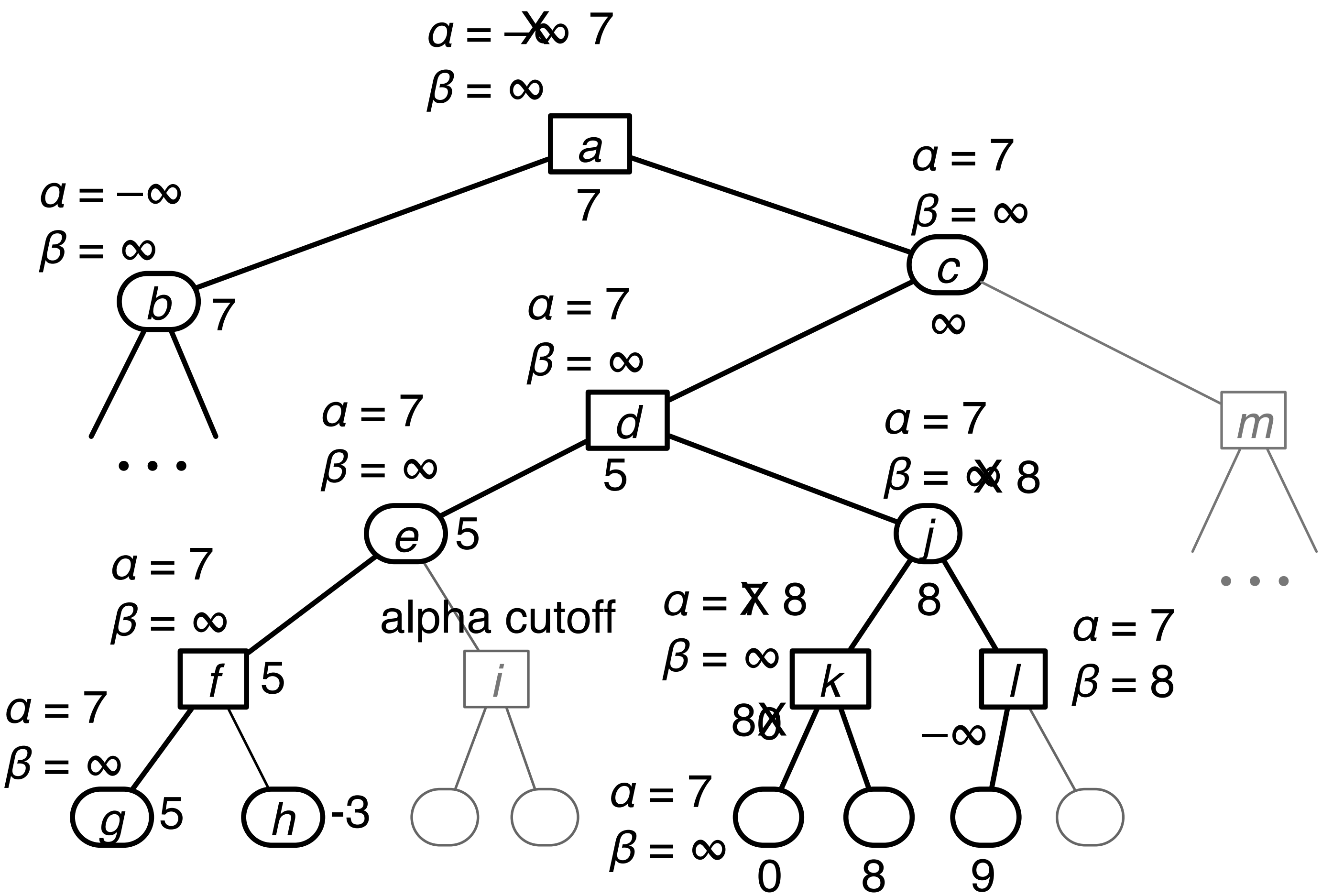

```
function Alpha-Beta(h, d, α, β)
    if h ∈ Z then return u(h)
    else if d = 0 then return e(h)
    else if ρ(h) = Max then
        v ← −∞
        for every a ∈ χ(h) do
            v ← max(v,Alpha-Beta(σ(h,a),d−1,α,β))
            if v ≥ β then return v        // β cutoff
            else α ← max(α, v)
        return v
    else
        v ← ∞
        for every a ∈ χ(h) do
            v ← min(v,Alpha-Beta(σ(h,a),d−1,α,β))
            if v ≤ α then return v        // α cutoff
            else β ← min(β,v)
        return v
```



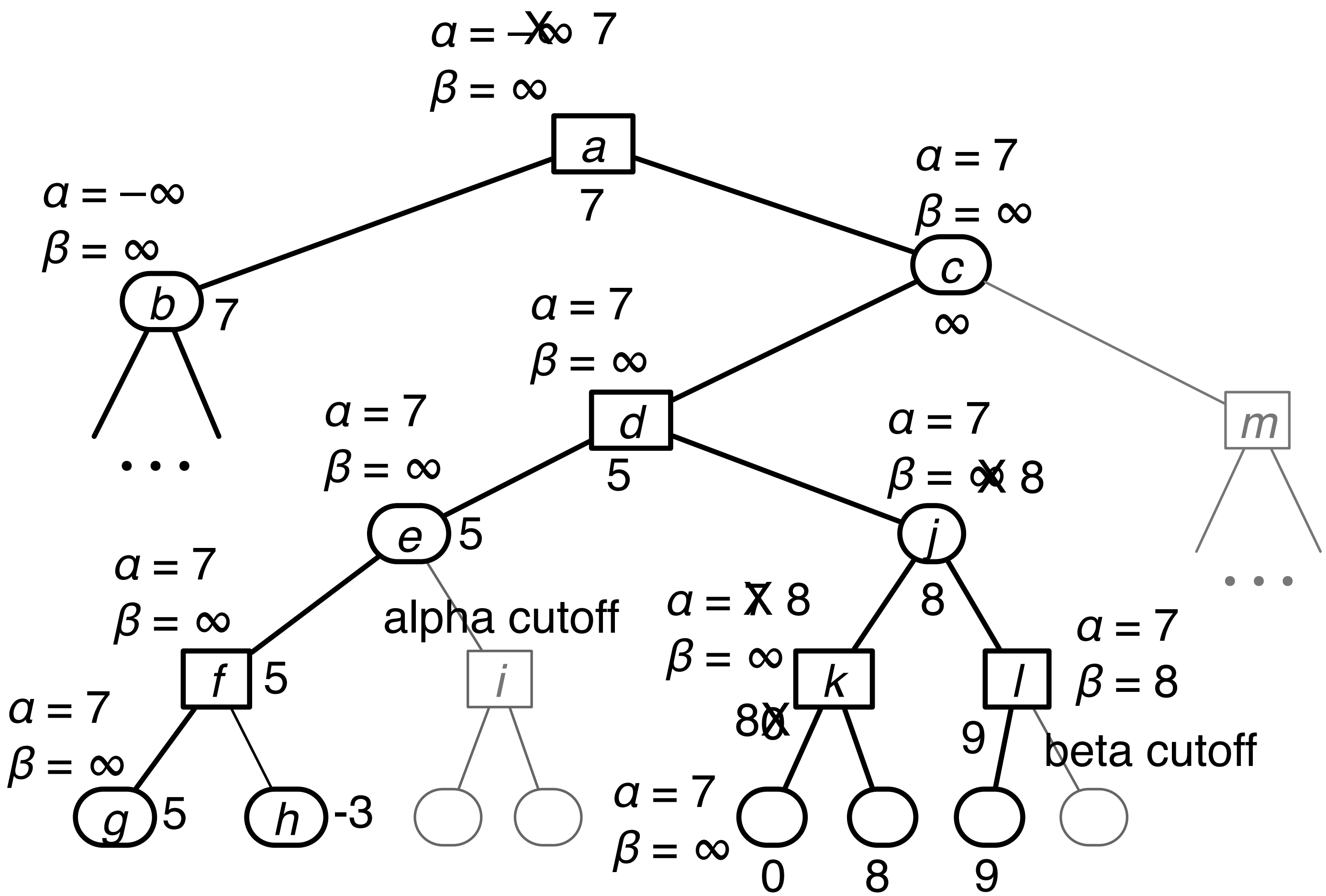

```
function Alpha-Beta(h, d, α, β)
    if h ∈ Z then return u(h)
    else if d = 0 then return e(h)
    else if ρ(h) = Max then
      v ← −∞
      for every a ∈ χ(h) do
          v ← max(v,Alpha-Beta(σ(h,a),d−1,α,β)
          if v ≥ β then return v        // β cutoff
          else α ← max(α, v)
      return v
    else
      v ← ∞
      for every a ∈ χ(h) do
          v ← min(v,Alpha-Beta(σ(h,a),d−1,α,β)
          if v ≤ α then return v        // α cutoff
          else β ← min(β,v)
      return v
```



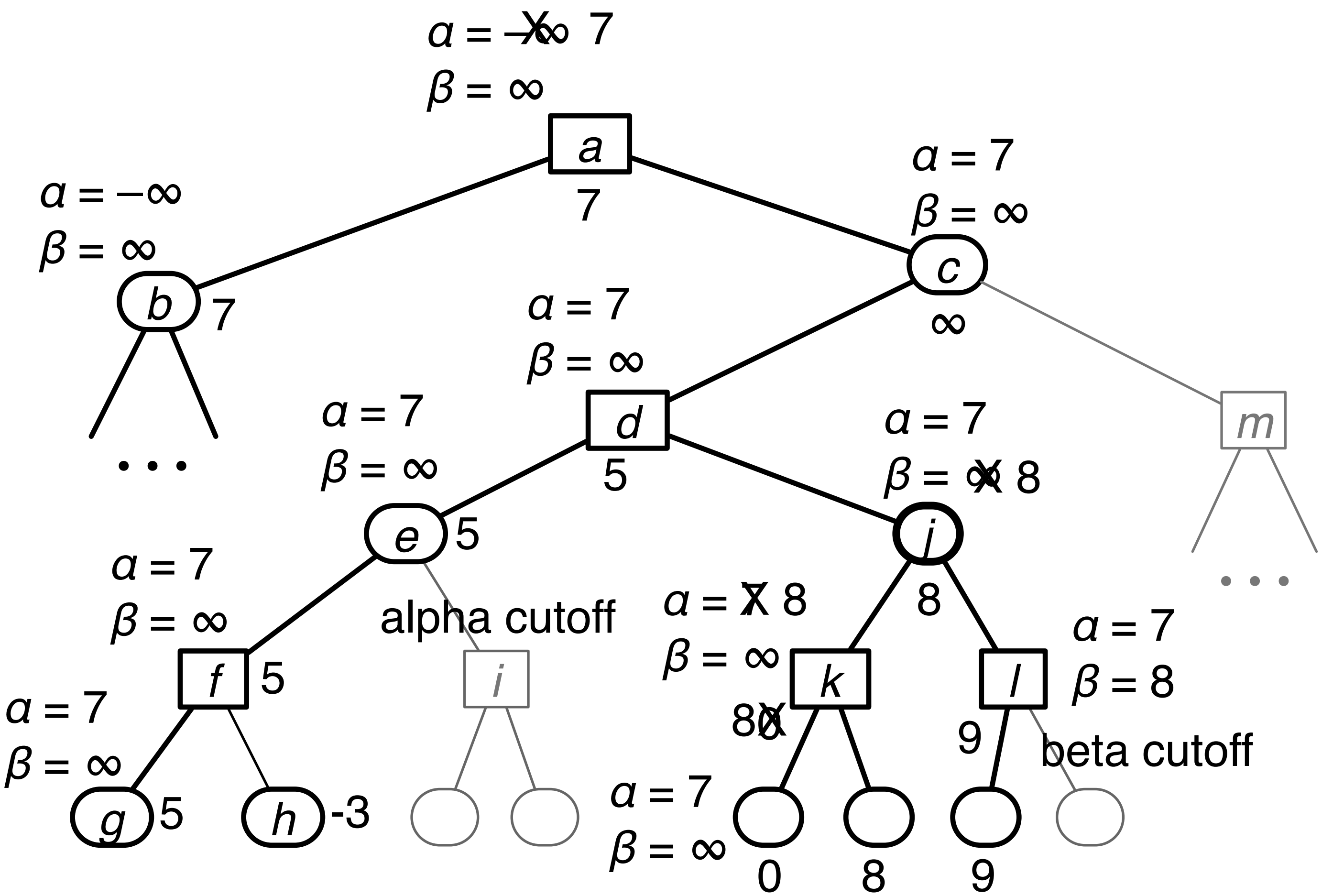

```
function Alpha-Beta(h, d, α, β)
    if h ∈ Z then return u(h)
    else if d = 0 then return e(h)
    else if ρ(h) = Max then
      v ← −∞
      for every a ∈ χ(h) do
          v ← max(v,Alpha-Beta(σ(h,a),d−1,α,β)
          if v ≥ β then return v        //  β cutoff
          else α ← max(α, v)
      return v
    else
      v ← ∞
      for every a ∈ χ(h) do
          v ← min(v,Alpha-Beta(σ(h,a),d−1,α,β)
          if v ≤ α then return v       //  α cutoff
          else β ← min(β,v)
      return v
```



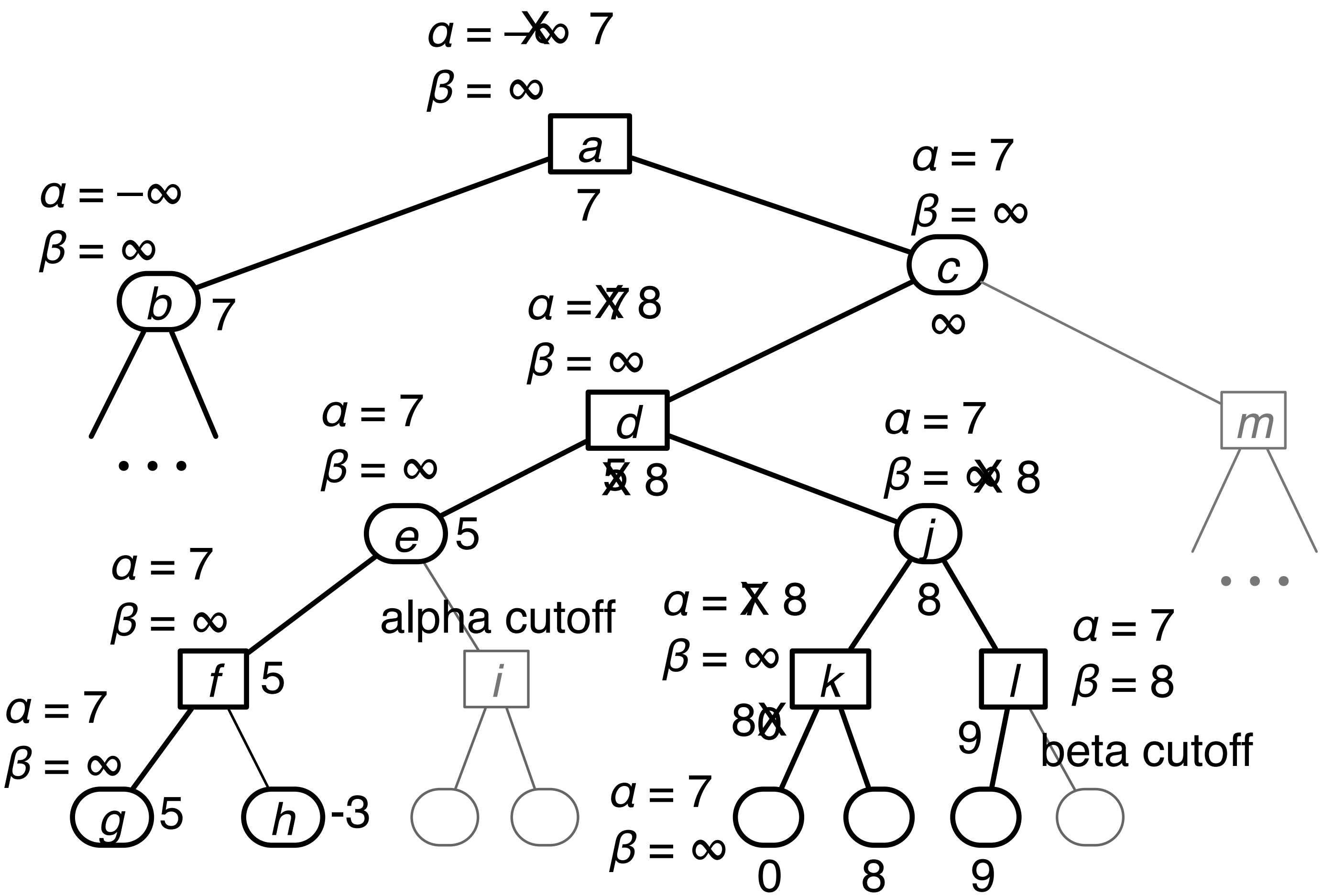

```
function Alpha-Beta(h, d, α, β)
    if h ∈ Z then return u(h)
    else if d = 0 then return e(h)
    else if ρ(h) = Max then
        v ← −∞
        for every a ∈ χ(h) do
            v ← max(v,Alpha-Beta(σ(h,a),d−1,α,β)
            if v ≥ β then return v        //  β cutoff
            else α ← max(α, v)
        return v
    else
        v ← ∞
        for every a ∈ χ(h) do
            v ← min(v,Alpha-Beta(σ(h,a),d−1,α,β)
            if v ≤ α then return v        //  α cutoff
            else β ← min(β,v)
        return v
```




```
function Alpha-Beta(h, d, α, β)
    if h ∈ Z then return u(h)
    else if d = 0 then return e(h)
    else if ρ(h) = Max then
        v ← −∞
        for every a ∈ χ(h) do
            v ← max(v,Alpha-Beta(σ(h,a),d−1,α,β)
            if v ≥ β then return v        // β cutoff
            else α ← max(α, v)
        return v
    else
        v ← ∞
        for every a ∈ χ(h) do
            v ← min(v,Alpha-Beta(σ(h,a),d−1,α,β)
            if v ≤ α then return v        // α cutoff
            else β ← min(β,v)
        return v
```




```
function Alpha-Beta(h, d, α, β)
    if h ∈ Z then return u(h)
    else if d = 0 then return e(h)
    else if ρ(h) = Max then
      v ← −∞
      for every a ∈ χ(h) do
          v ← max(v,Alpha-Beta(σ(h,a),d−1,α,β))
          if v ≥ β then return v        //  β cutoff
          else α ← max(α, v)
      return v
    else
      v ← ∞
      for every a ∈ χ(h) do
          v ← min(v,Alpha-Beta(σ(h,a),d−1,α,β))
          if v ≤ α then return v       //  α cutoff
          else β ← min(β,v)
      return v
```



function Alpha-Beta(h, d, α, β)

  if $h \in Z$ then return $u(h)$

  else if $d = 0$ then return $e(h)$

  else if $\rho(h) =$ Max then

    $v \leftarrow -\infty$

    for every $a \in \chi(h)$ do

      $v \leftarrow \max(v,$ Alpha-Beta$(\sigma(h,a), d-1, \alpha, \beta))$

      if $v \geq \beta$ then return v // β cutoff

      else $\alpha \leftarrow \max(\alpha, v)$

    return v

  else

    $v \leftarrow \infty$

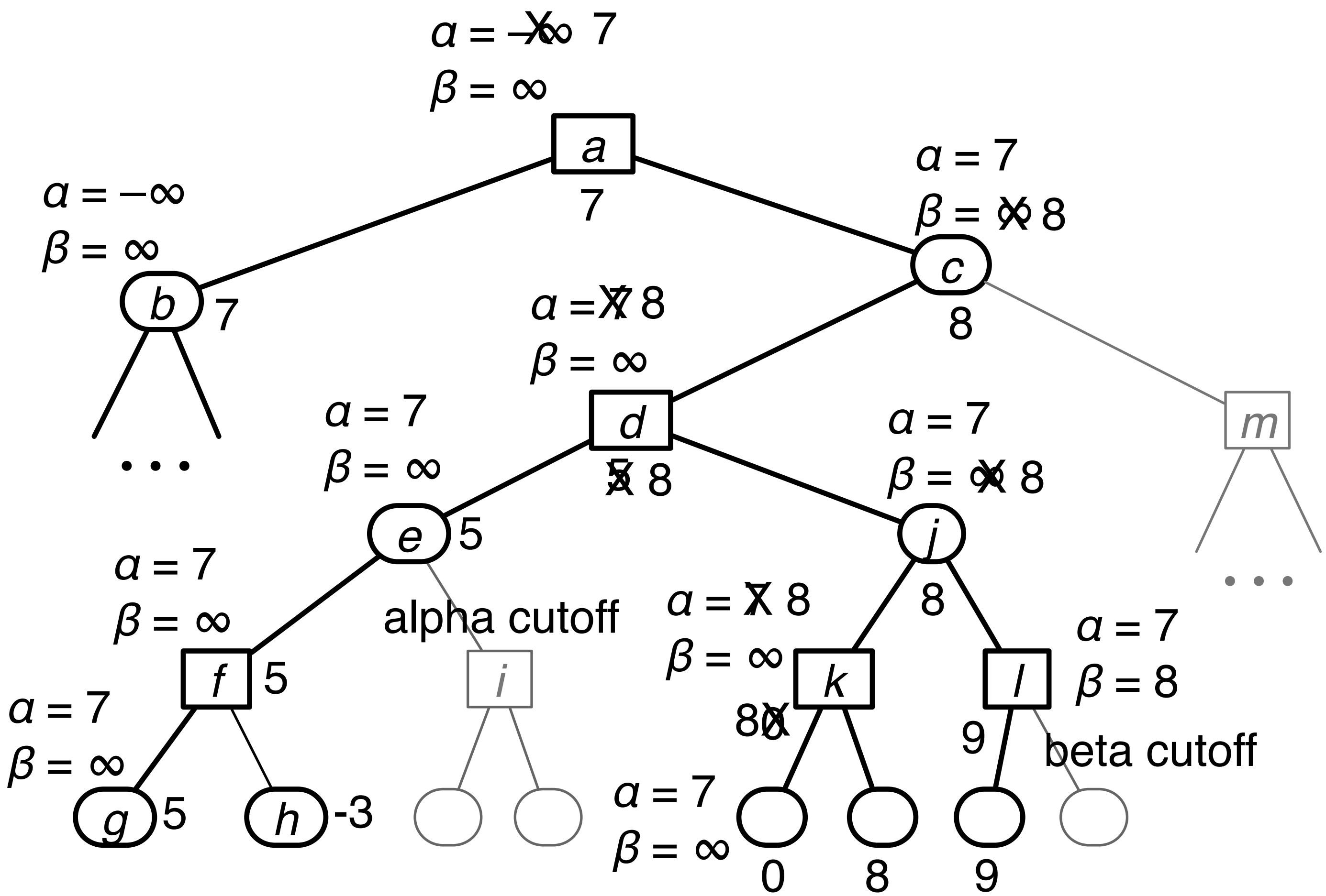
    for every $a \in \chi(h)$ do

      $v \leftarrow \min(v,$ Alpha-Beta$(\sigma(h,a), d-1, \alpha, \beta))$

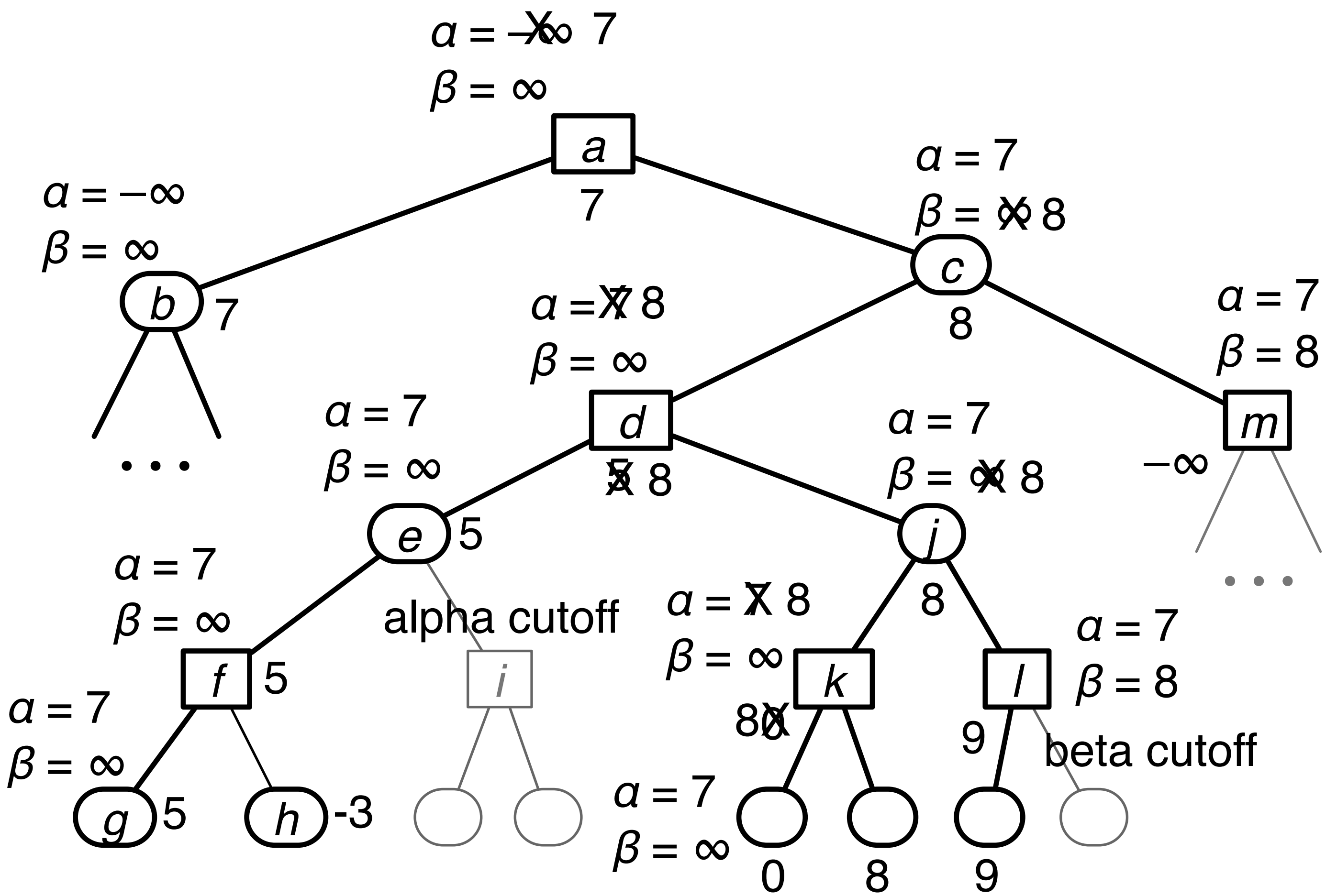
      if $v \leq \alpha$ then return v // α cutoff

      else $\beta \leftarrow \min(\beta, v)$

    return v




```
function Alpha-Beta(h, d, α, β)
    if h ∈ Z then return u(h)
    else if d = 0 then return e(h)
    else if ρ(h) = Max then
      v ← −∞
      for every a ∈ χ(h) do
          v ← max(v,Alpha-Beta(σ(h,a),d−1,α,β)
          if v ≥ β then return v       // β cutoff
          else α ← max(α, v)
      return v
    else
      v ← ∞
      for every a ∈ χ(h) do
          v ← min(v,Alpha-Beta(σ(h,a),d−1,α,β)
          if v ≤ α then return v       // α cutoff
          else β ← min(β,v)
      return v
```




```
function Alpha-Beta(h, d, α, β)
    if h ∈ Z then return u(h)
    else if d = 0 then return e(h)
    else if ρ(h) = Max then
        v ← −∞
        for every a ∈ χ(h) do
            v ← max(v,Alpha-Beta(σ(h,a),d−1,α,β))
            if v ≥ β then return v        // β cutoff
            else α ← max(α, v)
        return v
    else
        v ← ∞
        for every a ∈ χ(h) do
            v ← min(v,Alpha-Beta(σ(h,a),d−1,α,β))
            if v ≤ α then return v        // α cutoff
            else β ← min(β,v)
        return v
```

Properties of Alpha-Beta

- Alpha-beta pruning reasons about which computations are relevant
  - A form of **metareasoning**

Theorem:

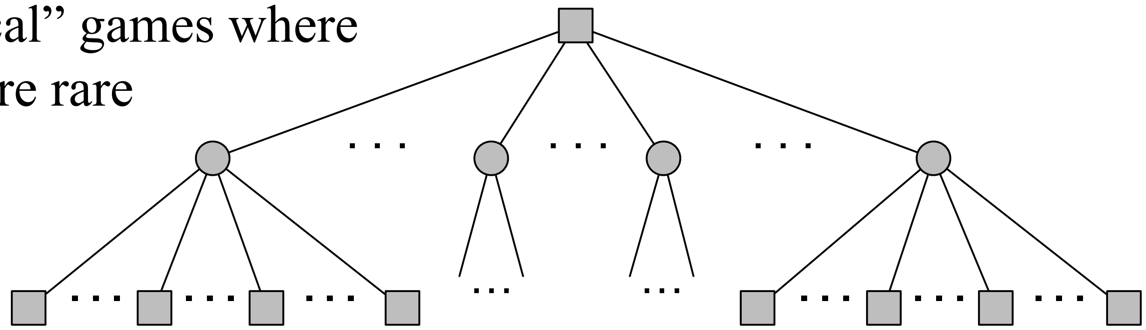
- If LD-minimax(h, d) returns a value in $[\alpha,\beta]$
  - Alpha-Beta(h, d, α, β) returns the same value
- If LD-minimax(h, d) returns a value $\leq \alpha$
  - Alpha-Beta(h, d, α, β) returns a value $\leq \alpha$
- If LD-minimax(h, d) returns a value $\geq \beta$
  - Alpha-Beta(h, d, α, β) returns a value $\geq \beta$

Corollary:

- Alpha-Beta$(h, d, -\infty, \infty)$ returns the same value as LD-minimax(h, d)
- Alpha-Beta$(h, \infty, -\infty, \infty)$ returns $u(h)$

Node Ordering

- Deeper search (larger d) usually gives better decisions
  - There are "pathological" games where it doesn't, but those are rare
- How much deeper can Alpha-Beta search?



- Worst case:
  - children of Max nodes are smallest-value-first
  - children of Min nodes are greatest-value-first
  - Alpha-Beta visits all nodes of depth $\leq d$
    - running time $O(b^d)$
- Best case:
  - children of Max nodes are greatest-value-first
  - children of Min nodes are smallest-value-first
  - Alpha-Beta's running time is $O(b^{d/2})$ $\Rightarrow$ doubles the solvable depth

Node Ordering

- How to get closer to the best case:
  - Every time you expand a state s, apply *e* to its children
  - When it's Max's move, sort the children in order of largest *e* first
  - When it's Min's move, sort the children in order of smallest *e* first

- Suppose we have 100 seconds, explore 10^4 nodes/second
  - 10^6 nodes per move
  - Chess midgame $b \approx 35$
    - LD-minimax: time $b^d = 35^d = 10^6 \rightarrow d \approx 4$
    - Alpha-Beta: time $b^{d/2} \approx 35^{8/2} \rightarrow d \approx 8$

Other Modifications

- Several other ways to improve accuracy or running time:
  - quiescence search and biasing
  - transposition tables
  - thinking on the opponent's time
  - table lookup of "book moves"
  - iterative deepening
  - forward pruning

Quiescence Search and Biasing

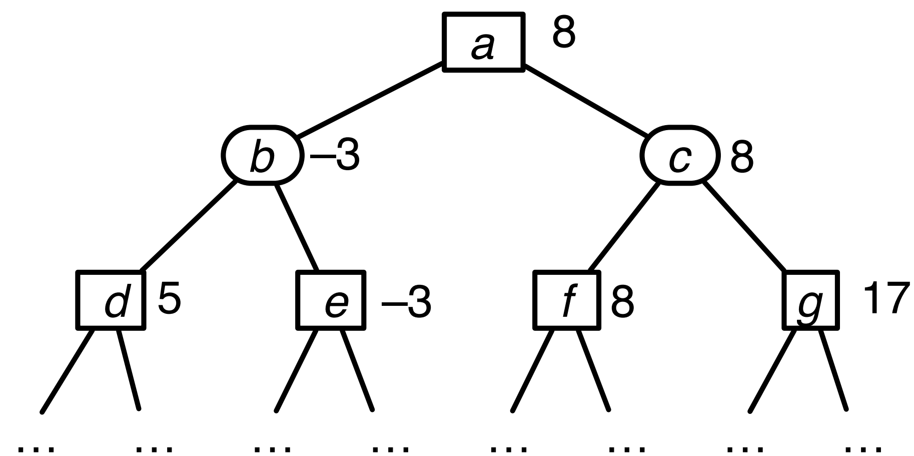
- In a game like checkers or chess, *e* is based greatly on material pieces
  - Likely to be inaccurate if there are pending captures
- Search deeper to reach a position where there aren't pending captures
  - *e* will be more accurate here
- That creates another problem
  - You're searching some paths to an even depth, others to an odd depth
  - Paths that end just after your opponent's move will look worse than paths that end just after your move
- To compensate, add or subtract a number called the "biasing factor"

Transposition Tables

- Multiple paths to the same state (state space is a graph rather than a tree)
- Idea:
  - when you compute *s*'s minimax value, store it in a hash table
  - visit *s* again $\Rightarrow$ retrieve its value rather than computing it again
- The hash table is called a **transposition table**

- Problem: can't store exponentially many states
  - Only store some – the ones that you're most likely to need

Thinking on the Opponent's Time



- Suppose you're at node *a*
  - Use Alpha-Beta to estimate *u*(*b*) and *u*(*c*)
  - *c* looks better, so move there
- Consider your estimates of *u*(*f*) and *u*(*g*)
  - They suggest your opponent is likely to move to *f*
  - While waiting for the opponent to move, start an alpha-beta search below *f*
- If the opponent moves to *f*, then you've already done a lot of the work of figuring out your next move

Book Moves

- In some games, experts have spent lots of time analyzing **openings**
  - Sequences of moves one might play at the start of the game
  - Best responses to those sequences
- Some of these are cataloged in standard reference works
  - e.g., the *Encyclopaedia of Chess Openings*
- Store these in a lookup table
  - Respond almost immediately, as long as the opponent sticks to a sequence that's in the book

- A technique humans can use when playing against such a program
  - Deliberately make a move that isn't in the book
  - This may weaken the human's position, but the computer will (i) start taking longer and (ii) stop playing as well

Iterative Deepening

- How deeply should you search a game tree?
  - When you call Alpha-Beta($h, d, -\infty, \infty$), what to use for d?
- small d => don't make as good a decision
- large d => run out of time without knowing what move to make
- Solution: **iterative deepening**

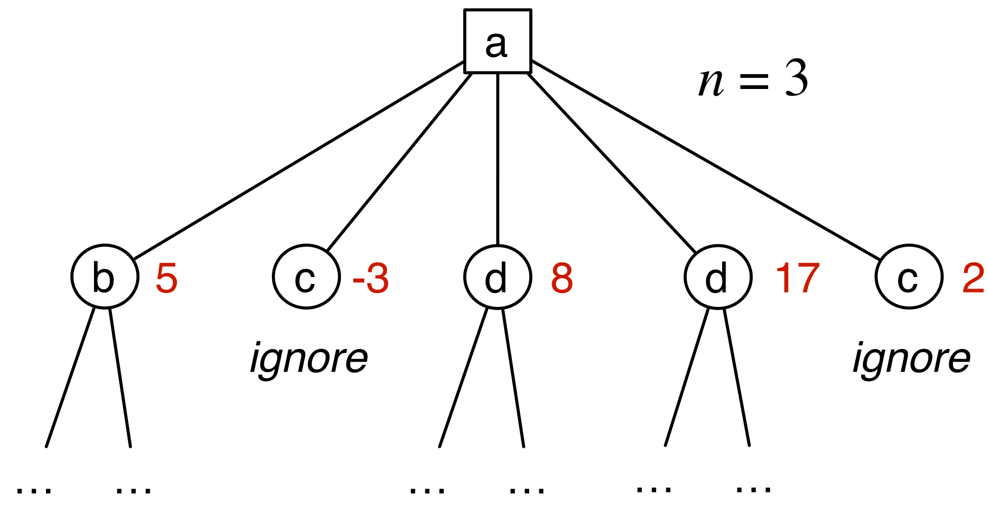
for $d = 1$ by 1 until you run out of time

$m \leftarrow$ the move returned by Alpha-Beta($h, d, -\infty, \infty$)

- Why this works:
  - Time complexity is $O(b^1 + b^2 + \ldots + b^d) = O(b^d)$
  - For large b, each iteration takes much more time than the total of *all the previous iterations*

Forward Pruning

- **Tapered search**:
  - Instead of looking at all of a node's children, just look at the n best ones
    - the n highest $e(h)$ values
  - Decrease the value of n as you go deeper in the tree
  - Drawback: may exclude an important move at a low level of the tree



- **Marginal forward pruning**:
  - Ignore every child h such that $e(h)$ is smaller than the current best values from the nodes you've already visited
  - Not reliable, should be avoided

Forward Pruning

- Until the mid-1970s most computer-chess researchers tried to get programs to search "like people think"
  - Use extensive chess knowledge at each node to select a few "plausible" moves; prune the others
  - Serious tactical shortcomings
- Brute-force search programs did better, dominated computer chess for about 20 years
- Early 1990s: development of some forward-pruning techniques that worked well
  - Null-move pruning
- Today most chess programs use some kind of forward-pruning
  - null-move pruning is one of the most popular

Game-Tree Search in Practice

- **Othello**: since 1980, the best computer programs have easily defeated the best humans
- **Checkers**: In 1994, Chinook ended 40-year-reign of human world champion Marion Tinsley
  - Tinsley withdrew for health reasons, died a few months later
- In 2007, Checkers was solved
  - With perfect play, it's a draw
  - This took 10^{14} calculations over 18 years
  - Search space size 5×10^{20}
- **Chess**: In 1997, Deep Blue defeated Gary Kasparov in a six-game match
  - Used special hardware, could search 200 million positions per second
  - Modern programs don't use special hardware
  - Barred from world-championship matches (except against other computers)

Game-Tree Search in Practice

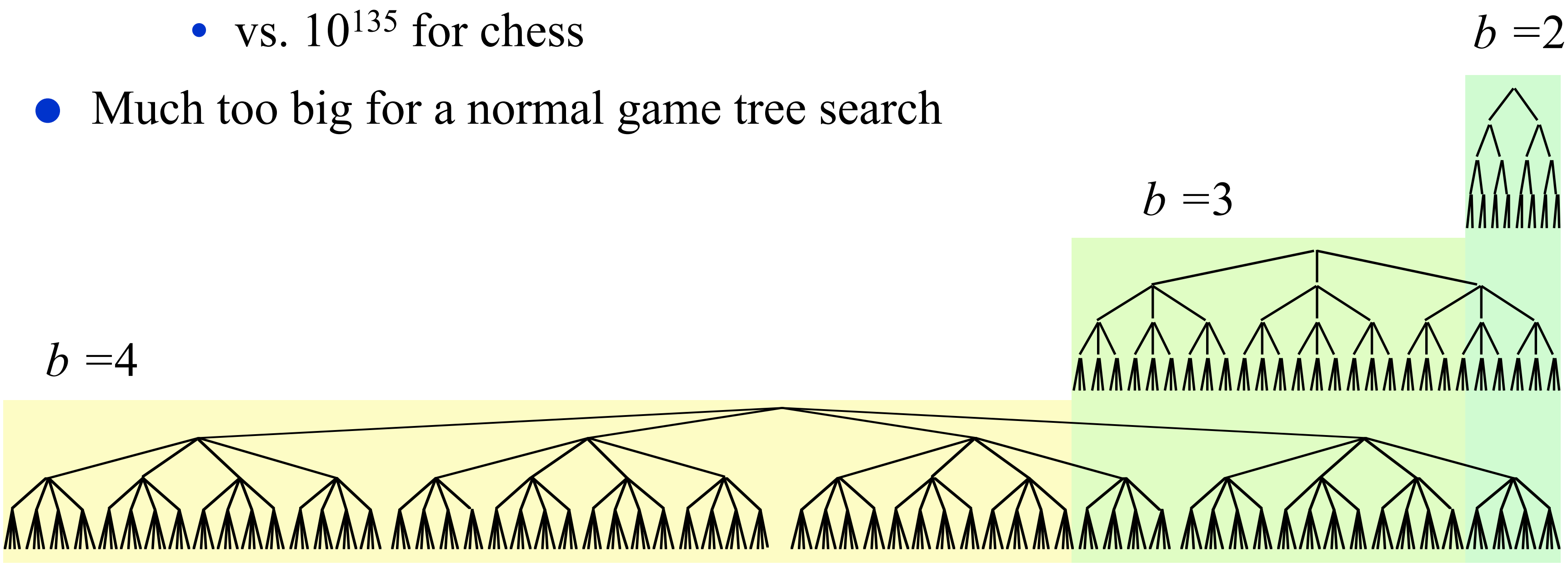
- **Go**: before 2006, good amateurs could easily beat the best go programs, even when given a handicap
- Two *big* jumps in performance
  - 2006: Monte Carlo rollouts
    - brought go programs up to a master level, better than strong amateurs
  - 2015: Monte Carlo rollouts combined with deep learning
    - AlphaGo: first go program to beat professional human players

Rules of Go (Abbreviated)

- Go-board: 19 × 19 locations (intersections on a grid)
- Back and White take turns
- Black has the first move
- Each move consists of placing a stone at an unoccupied location
  - You just put them on the board; you don't move them around
- Adjacent stones of the same color are called a **string**.
  - **Liberties** are the empty locations next to the string
  - A string is removed if its number of liberties is 0
- Score: territories (number of occupied or surrounded locations)

Why Go is Difficult for Computers

- A game tree's size grows exponentially with both its depth and its branching factor
- The game tree for go:
  - branching factor ≈ 200
  - game length ≈ 250 to 300 moves
  - number of nodes in the game tree ≈ 10^{525} to 10^{620}
    - vs. 10^{135} for chess
- Much too big for a normal game tree search



Why Go is Difficult for Computers

- Writing an evaluation function for chess
  - Mainly piece count, plus some positional considerations
    - isolated/doubled pawns, rooks on open files (columns), pawns in the center of the board, etc.
  - As the game progresses, pieces get removed => evaluation gets easier

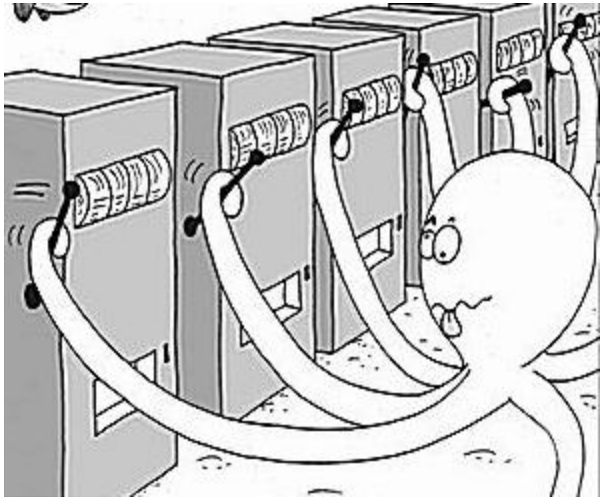
- For Go, much more complicated
    - whether or not a group is alive
    - which stones can be connected to one another
    - the extent to which a position has influence or can be attacked
  - As the game progresses, pieces get added => evaluation gets more complicated

Monte Carlo Roll-Outs

- Basic idea:
  - Generate a list of potential moves $A = \{a_1, \ldots, a_n\}$
  - For each move a_i in A:
    - Starting at $\sigma(h,a_i)$, generate a set of random games G_i in which the two players make all their moves at random
  - Choose the move in A that produces the best results
- Whether this works depends on **how** you generate the random games

Multi-Arm Bandit

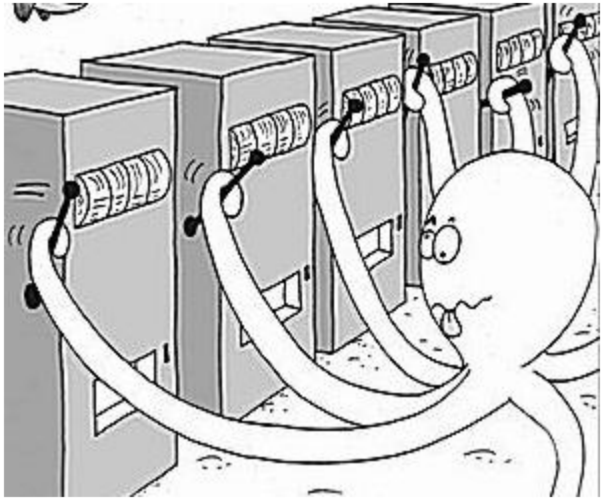
- Statistical model of sequential experiments
  - Name comes from a traditional slot machine (one-armed bandit)



- Multiple actions
  - Each action provides a reward from a probability distribution associated with that specific action
  - Objective: maximize the expected utility of a sequence of actions
- Exploitation vs exploration dilemma:
  - ***Exploitation***: choosing an action that you already know about, because you think it's likely to give you a high reward
  - ***Exploration***: choosing an action that you don't know much about, in hopes that maybe it will produce a better reward than the actions you already know about

UCB (Upper Confidence Bound) Algorithm

- Let
  - r_i = average reward you've gotten from arm i
  - t_i = number of times you've tried arm i;
  - t = total number of tries = $\sum_i t_i$



- **loop**
  - **if** there are one or more arms that have not been played
  - **then** play one of them
  - **else** play the arm i that has the highest value of $r_i + 2\sqrt{(\log t)/t_i}$

UCT (UCB for Trees)

- Adaptation of UCB for game-tree search
- First time it was used in go: Mogo, 2006
  - Won the go tournament at the 2007 Computer Olympiad, and several other computer go tournaments
- Now used in most computer go programs

- Next page: basic idea (omitting several improvements)

UCT (Basic Idea)

global $Seen \leftarrow \varnothing$

function UCT(h)

if $h \notin Seen$ then

add h to $Seen$

$r_h \leftarrow 0$; $t_h \leftarrow 0$

$h' \leftarrow$ UCB-choose($\{\sigma(h,a) \mid a \in \chi(h)\}$)

$\mathbf{v} \leftarrow$ UCT(h')

$r_h \leftarrow (r_h\, t_h + \mathbf{v}[\rho(h)]) \,/\, (t_h + 1)$

$t_h \leftarrow t_h + 1$

return $\mathbf{v}$

> $Seen$ = {all nodes seen so far}
> t_h = no. of times we've tried h
> r_h = avg. reward for $\rho(h)$ at h
> $\rho(h)$ = player to move at h
> $\chi(h)$ = {available actions at h}
> $\sigma(h,a)$ = child of h produced by a

function UCB-choose(H)

if $H \setminus Seen \neq \varnothing$ then return a randomly chosen member of $H \setminus Seen$

else

$t \leftarrow \sum_{h \in H} t_h$

return the $h \in H$ with the highest value of $r_h + 2\sqrt{(\log t) \,/\, t_h}$

UCT (Basic Idea)

global $Seen \leftarrow \varnothing$

function UCT(h)
- if $h \notin Seen$ then
  - add h to $Seen$
  - $r_h \leftarrow 0;\ \ t_h \leftarrow 0$
- $h' \leftarrow$ UCB-choose($\{\sigma(h,a) \mid a \in \chi(h)\}$)
- $\mathbf{v} \leftarrow$ UCT(h')
- $r_h \leftarrow (r_h\, t_h + \mathbf{v}[\rho(h)]) \,/\, (t_h + 1)$
- $t_h \leftarrow t_h + 1$
- return $\mathbf{v}$

- As the number of iterations $\rightarrow \infty$, each $r_h \rightarrow u(h)$

function UCB-choose(H)
- if $H \setminus Seen \neq \varnothing$ then return a randomly chosen member of $H \setminus Seen$
- else
  - $t \leftarrow \sum_{h \in H} t_h$
  - return the $h \in H$ with the highest value of $r_h + 2\sqrt{(\log t) \,/\, t_h}$

UCT (Basic Idea)

global $Seen \leftarrow \varnothing$

function UCT(h)
  if $h \notin Seen$ then
    add h to $Seen$
    $r_h \leftarrow 0;\ \ t_h \leftarrow 0$
  $h' \leftarrow$ UCB-choose($\{\sigma(h,a) \mid a \in \chi(h)\}$)
  $\mathbf{v} \leftarrow$ UCT(h')
  $r_h \leftarrow (r_h\, t_h + \mathbf{v}[\rho(h)]) \,/\, (t_h + 1)$
  $t_h \leftarrow t_h + 1$
  return $\mathbf{v}$

function UCB-choose(H)
  if $H \setminus Seen \neq \varnothing$ then return a randomly chosen member of $H \setminus Seen$
  else
    $t \leftarrow \sum_{h \in H} t_h$
    return the $h \in H$ with the highest value of $r_h + 2\sqrt{(\log t) \,/\, t_h}$

- As the number of iterations $\rightarrow \infty$, each $r_h \rightarrow u(h)$
- Question: how can this be true?
  - $u(h)$ should mean each player chooses their max payoff
  - But r_h is *average* payoff

global $Seen \leftarrow \varnothing$

UCT (Basic Idea)

function UCT(h)
    if $h \notin Seen$ then
        add h to $Seen$
        $r_h \leftarrow 0;\ \ t_h \leftarrow 0$
    $h' \leftarrow$ UCB-choose($\{\sigma(h,a) \mid a \in \chi(h)\}$)
    $\mathbf{v} \leftarrow$ UCT(h')
    $r_h \leftarrow (r_h\, t_h + \mathbf{v}[\rho(h)]) \,/\, (t_h + 1)$
    $t_h \leftarrow t_h + 1$
    return $\mathbf{v}$

function UCB-choose(H)
    if $H \setminus Seen \neq \varnothing$ then return a randomly chosen member of $H \setminus Seen$
    else
        $t \leftarrow \sum_{h \in H} t_h$
        return the $h \in H$ with the highest value of $r_h + 2\sqrt{(\log t) / t_h}$

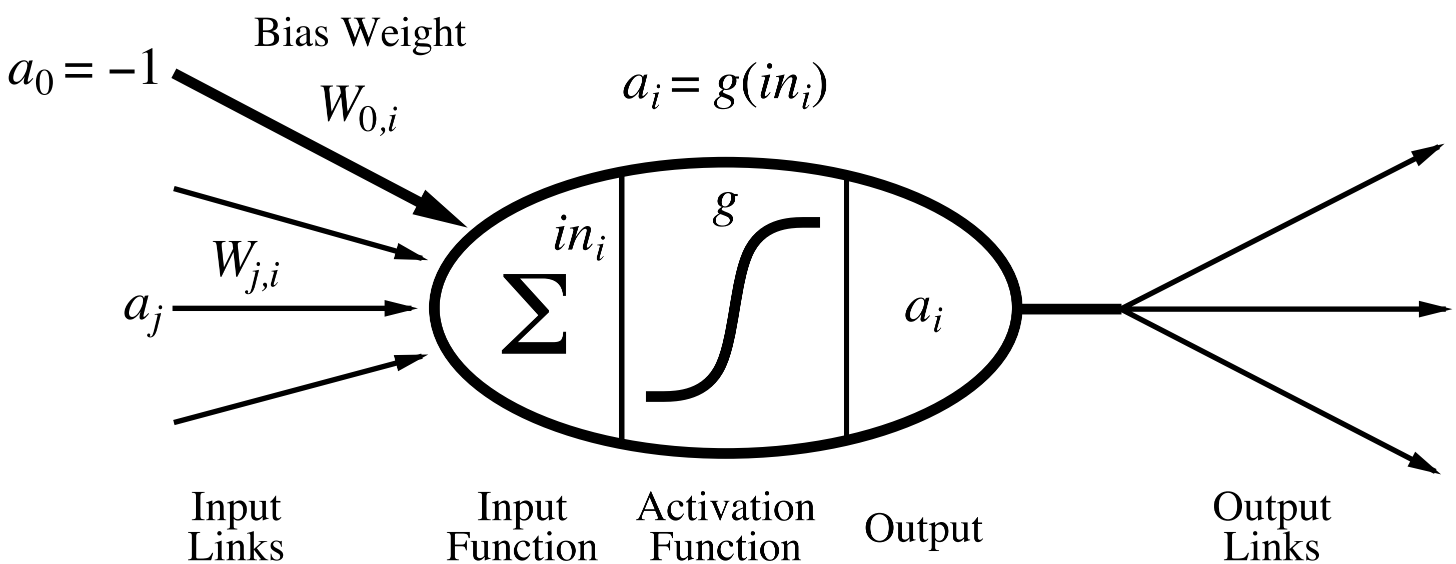
- As the number of iterations $\rightarrow \infty$, each $r_h \rightarrow u(h)$
- Question: how can this be true?
  - $u(h)$ should mean each player chooses their max payoff
  - But r_h is *average* payoff
- Answer: as iterations $\rightarrow \infty$, $\sqrt{(\log t) / t_h} \rightarrow 0$
  - UCB-choose converges to choosing *highest* avg. payoff

AlphaGo: Game Playing

- 2015: first go program to beat a professional go player
- 2016: first program to beat one of the world's top go players
- Uses a combination of Monte-Carlo with artificial neural networks

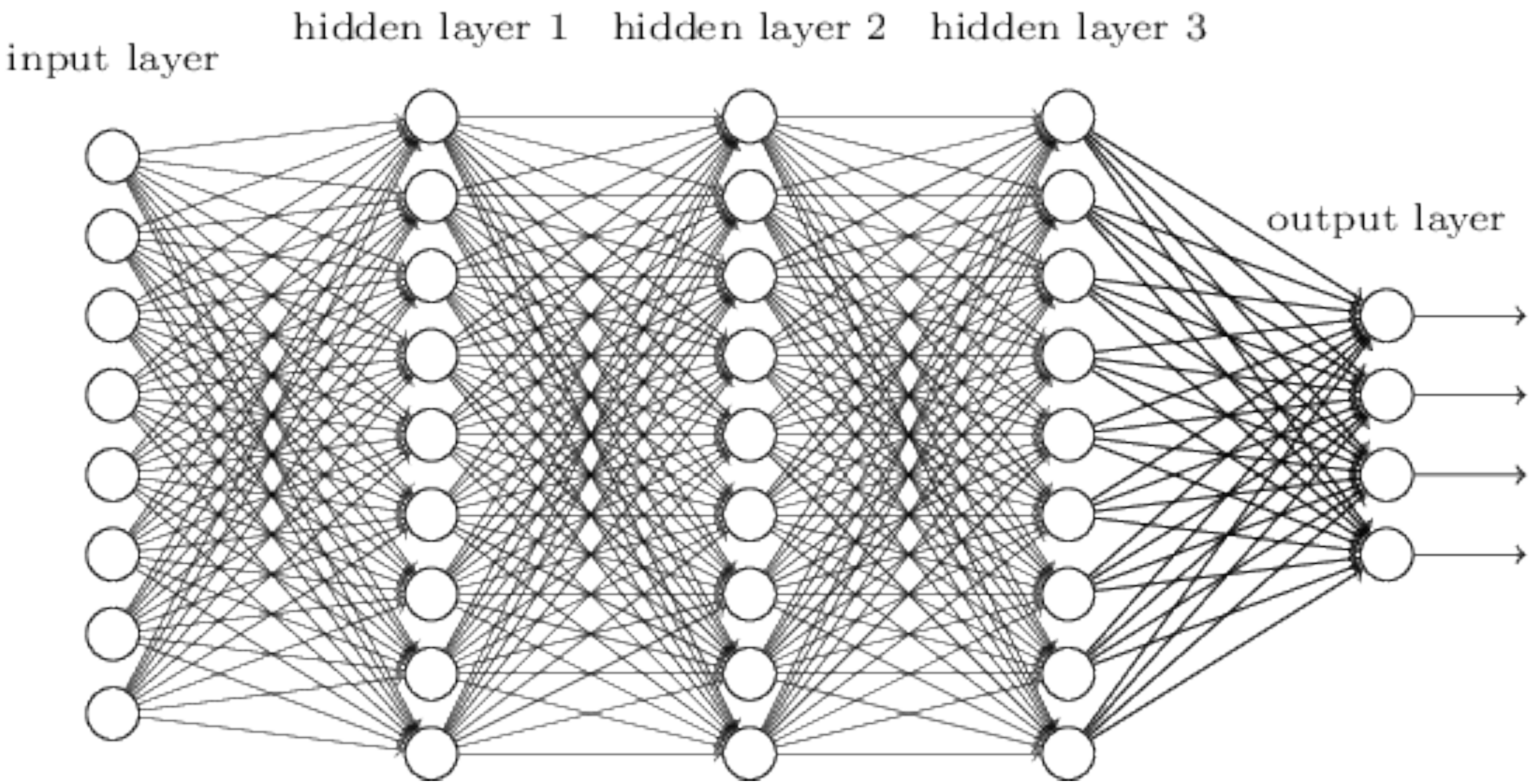
Artificial Neural Networks

- Collection of "units" that act a little like biological neurons
- Each unit:



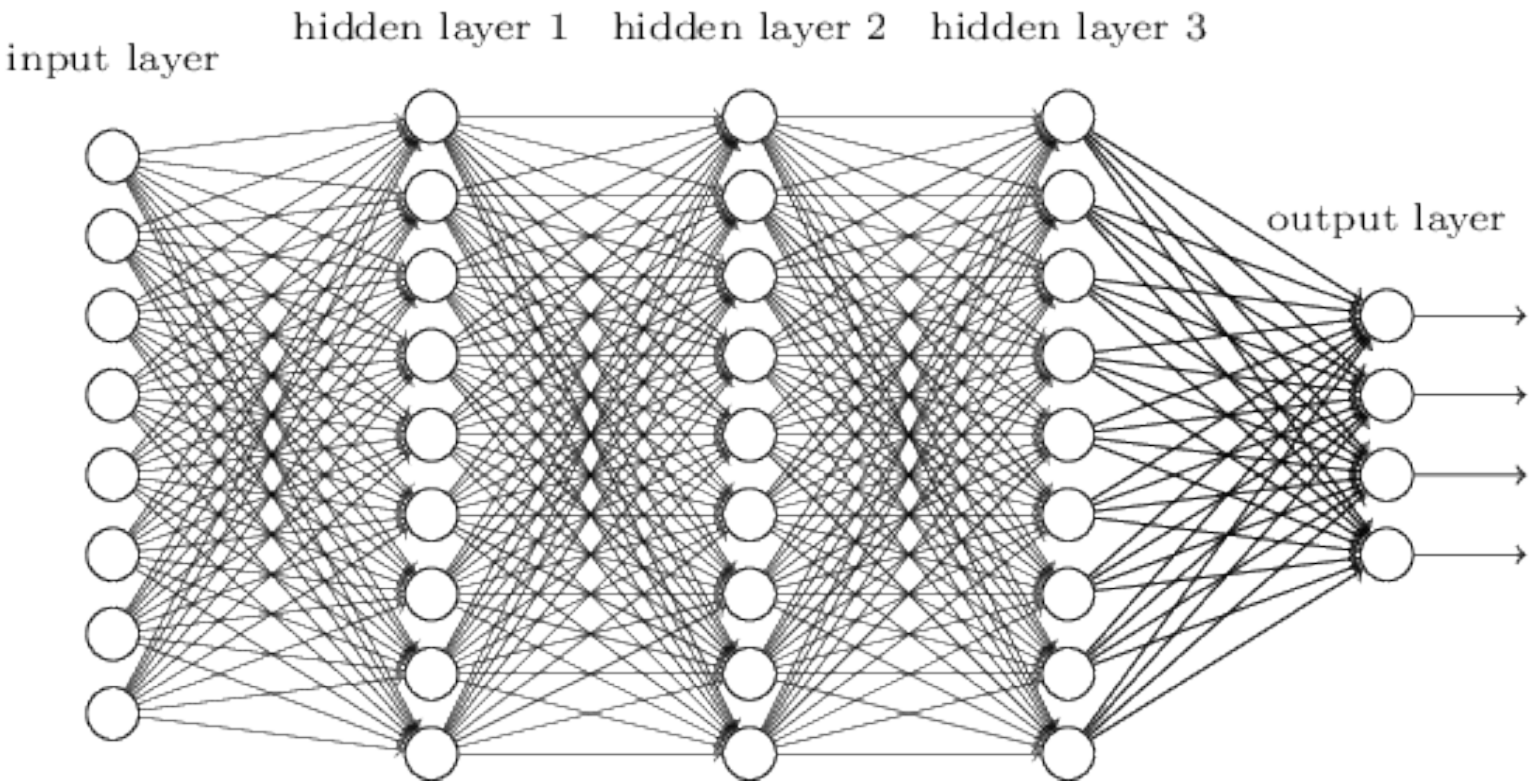
Artificial Neural Networks

- Generally organized into layers
- Can train them to compute a wide variety of functions
  - Give them lots of examples: (*input, output*) pairs
- Formulas for adjusting the bias weights to try to do well on all the pairs



Artificial Neural Networks

- *Convolutional* neural network: a particular way to organize the layers
- *Deep* neural network: many layers
- *Deep learning*: learning using a deep neural network



AlphaGo: Training

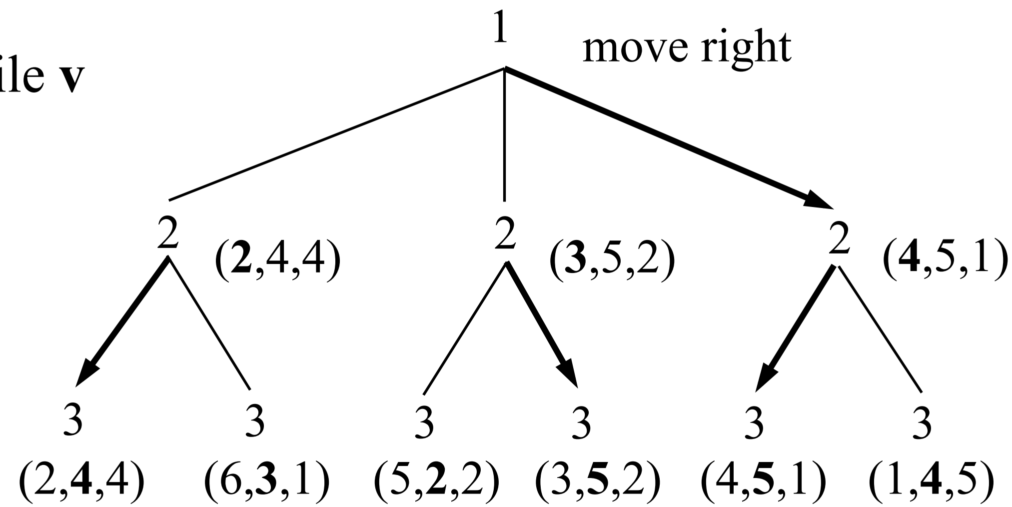
- Policy network
  - Give it examples of moves experts would make in various situations
    - (*input, output*) pair: (position, expert move)
  - Resulting neural network:
    - input: position → output: probability distribution over moves
  - Do some optimization to make it work better
    - I'll skip the details
- Value network
  - Use the policy network to play games against itself
  - Use the results to train another deep neural network
    - (*input, output*) pair: (position, who won)
  - Resulting neural network:
    - input: position → output: expected utility

AlphaGo: Training

- Generate Monte Carlo rollouts
  - Somewhat like UCT – but in addition to r_h and t_h it involves
    - probability of h returned by the neural network
    - value of h returned by the value network
- I don't know the details

Multiplayer Games

- **Max^n** algorithm: backward induction for n players, with cutoff depth d and evaluation function $\mathbf{e}$
- Node evaluation is a payoff profile $\mathbf{v}$
  $\mathbf{v}[i]$ is player i's payoff
- Approximate SPE payoff profile
  - Exact if $d \geq$ height of h



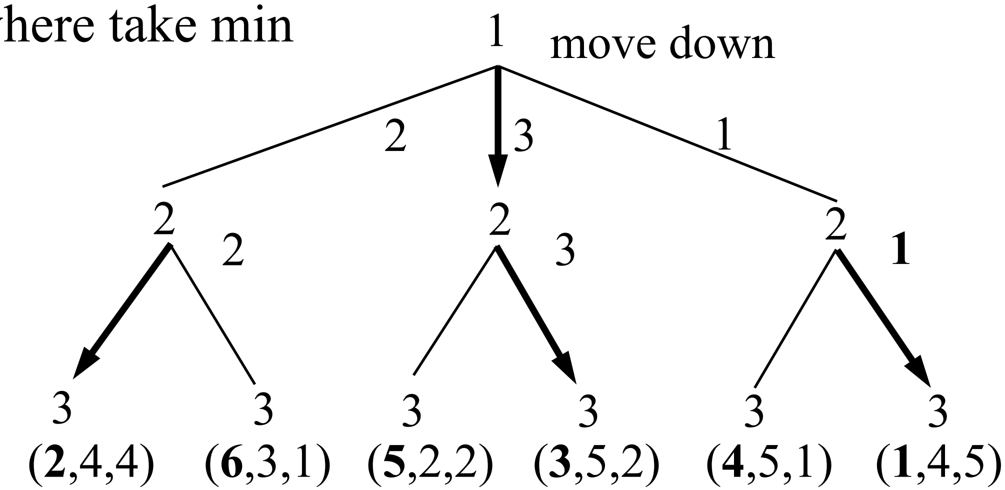
function Maxn (h,d)
  if $h \in Z$ then return $\mathbf{u}(h)$
  if $d = 0$ then return $\mathbf{e}(h)$
  $V = \{\text{Maxn}(\sigma(h,a), d-1) \mid a \in \chi(h)\}$
  return $\arg\max_{\mathbf{v} \in V} \mathbf{v}[\rho(h)]$

function Maxn-choice (h, d)
  if $h \in Z$ or $d = 0$ then return *error*
  return $\arg\max_{a \in \chi(h)} (\text{Maxn}(\sigma(h,a), d-1))[\rho(h)]$

H = {nonterminal nodes}
Z = {terminal nodes}
$\rho(h)$ = the player to move at h
$\chi(h)$ = {available actions at h}
$\sigma(h,a)$ = child of h produced by a
$\mathbf{u}(h)$ = utility profile for h
$\mathbf{v}[i]$ = i'th element of $\mathbf{v}$

Multiplayer Games

- The **Paranoid** algorithm
  - Cutoff depth d and evaluation function **e**
  - At i's move take max, at elsewhere take min
- Approximate maxmin value for i
  - exact value if $d \geq$ height of h



function Paranoid(i, h, d)

if $h \in Z$ then return $u_i(h)$

if $d = 0$ then return $e_i(h)$

if $\rho(h) = i$ then return $\max_{a \in \chi(h)}$ Paranoid($i, \sigma(h,a), d-1$)

else return $\min_{a \in \chi(h)}$ Paranoid($i, \sigma(h,a), d-1$)

function Paranoid-choice (i,h,d)

if $\rho(h) = i$ then return $\arg\max_{a \in \chi(h)}$ Paranoid($i, \sigma(h,a), d-1$)

else return *error*

Discussion

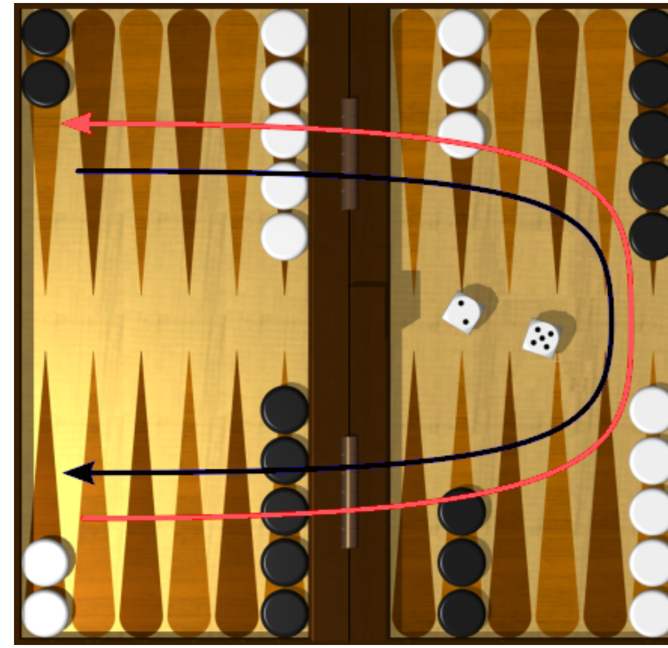
- Neither Max^n nor Paranoid has been entirely successful
- Partly due to dynamic relationships among players
  - Human players may hold grudges
  - Informal alliances form and dissolve over time
    - players who are behind may "gang up" on a player who's ahead

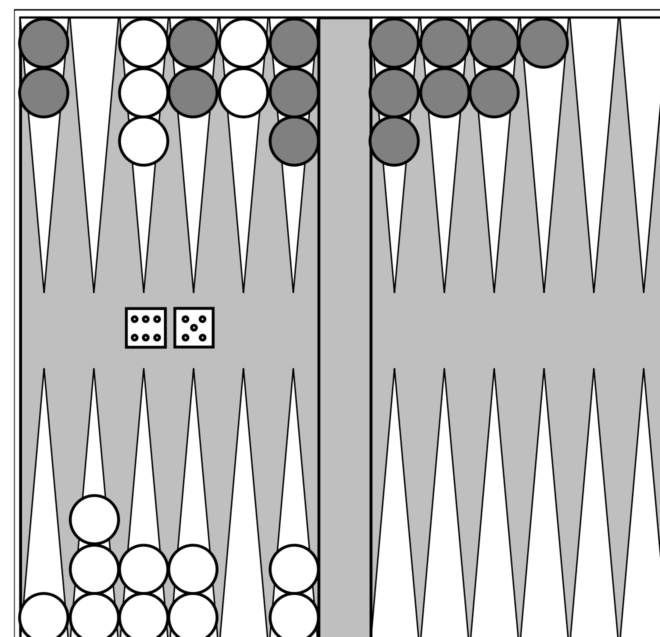
- Max^n and Paranoid don't model these relationships
  - But they can greatly influence the players' strategies

- For better play in a multi-player game, need ways
  - Decide when/whether to cooperate with others
  - Deduce each player's attitude toward the other players

Games with Chance Nodes

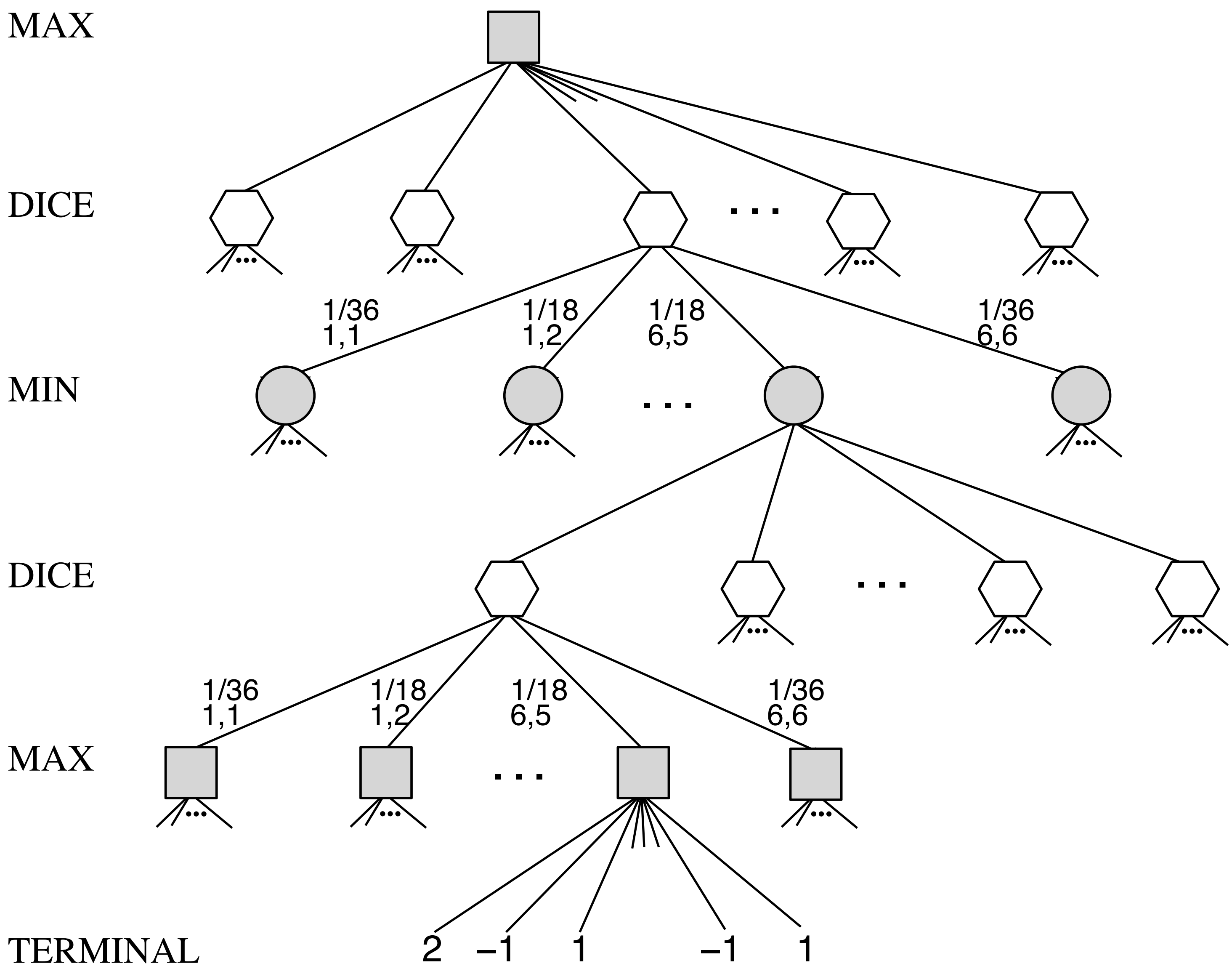
- Example: Backgammon
- Two agents who take turns
- At each turn, the set of available moves depends on the results of rolling the dice
  - Each die specifies how far to move one of your pieces
    - (except if you roll doubles)
  - You can’t move to a location where your opponent has 2 or more pieces
  - You can move to a location where your opponent has 1 piece
    - Knock that piece off the board, it must start over





Backgammon Game Tree

- Players' moves have deterministic outcomes
- Dice rolls have stochastic outcomes



ExpectiMinimax

Z = {terminal nodes}

$\rho(h)$ = the player to move at h

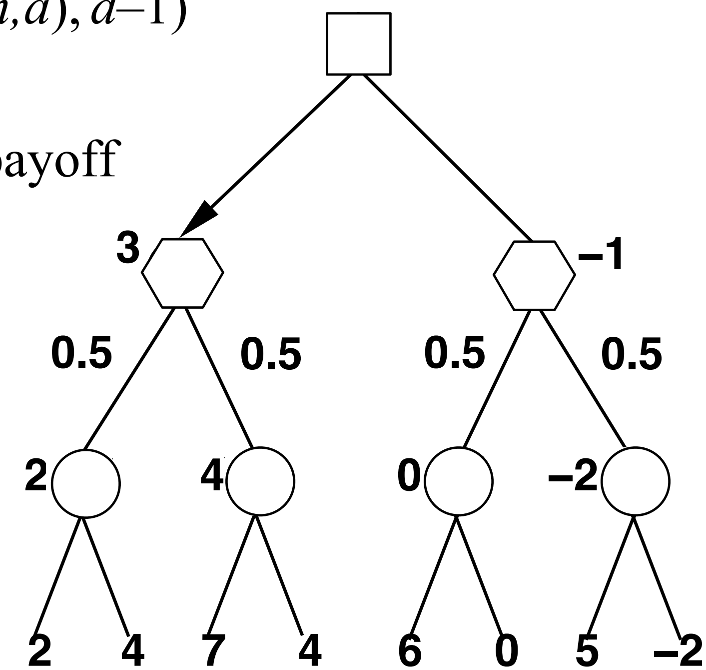
$\chi(h)$ = {available actions at h}

$\sigma(h,a)$ = child of h produced by a

$\mathbf{u}(h)$ = utility profile for h

$\mathbf{v}[i]$ = i'th element of $\mathbf{v}$

function ExpectiMinimax(h,d)
  if $h \in Z$ then return $u(h)$
  else if $d = 0$ then return $e(h)$
  else if $\rho(h)$ = Max then return $\max_{a \in \chi(h)}$ ExpectiMinimax($\sigma(h,a), d-1$)
  else if $\rho(h)$ = Min then return $\min_{a \in \chi(h)}$ ExpectiMinimax($\sigma(h,a), d-1$)
  else return $\Sigma_{a \in \chi(h)} \Pr[a \mid h]$ ExpectiMinimax($\sigma(h,a), d-1$)



- If $d \geq$ height(h), returns Max's SPE expected payoff
  - Otherwise returns an approximation
- Can be modified to return an action
- Can also be modified to do α-β pruning
  - But it's more complicated and less effective than in deterministic games

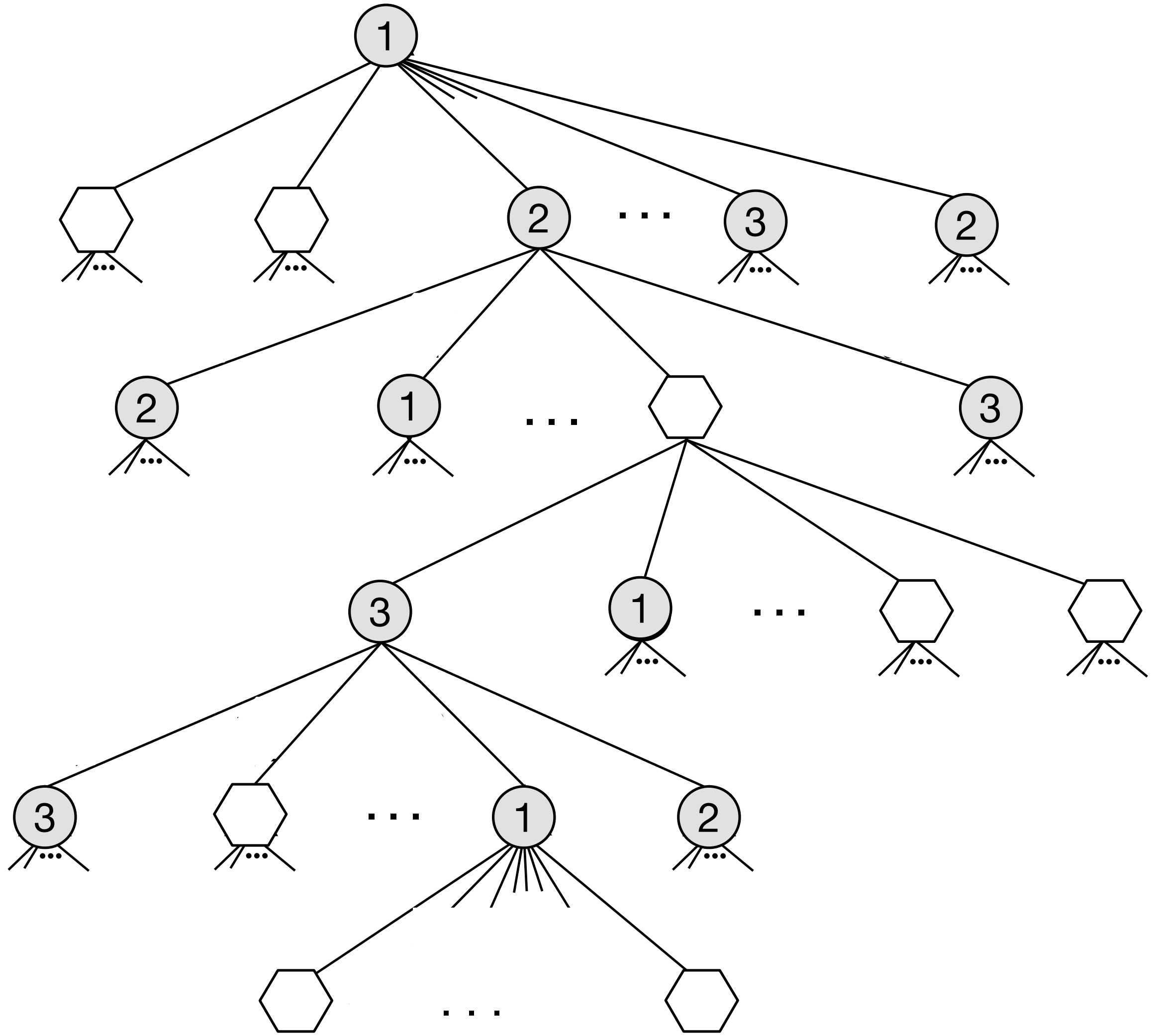
In Practice

- Dice rolls increase branching factor
  - 21 possible rolls with 2 dice
  - Given the dice roll, ≈ 20 legal moves on average
    - For some dice roles, can be much higher
  - At depth 4, number of nodes is $= 20 \times (21 \times 20)^3 \approx 1.2 \times 10^9$
  - As depth increases, probability of reaching a given node shrinks

    ⇒ value of lookahead is diminished
- *α-β* pruning is much less effective
- TDGammon (1992) used depth-2 search + very good evaluation function
  - Created its evaluation function automatically using a machine-learning technique called *Temporal Difference* learning
    - hence the *TD* in TDGammon
  - ≈ world-champion level

Generalize

- Finite perfect-information games with
  - Multiple players
  - Chance nodes
  - Nonzero-sum payoffs



Expectimax

Z = {terminal nodes}

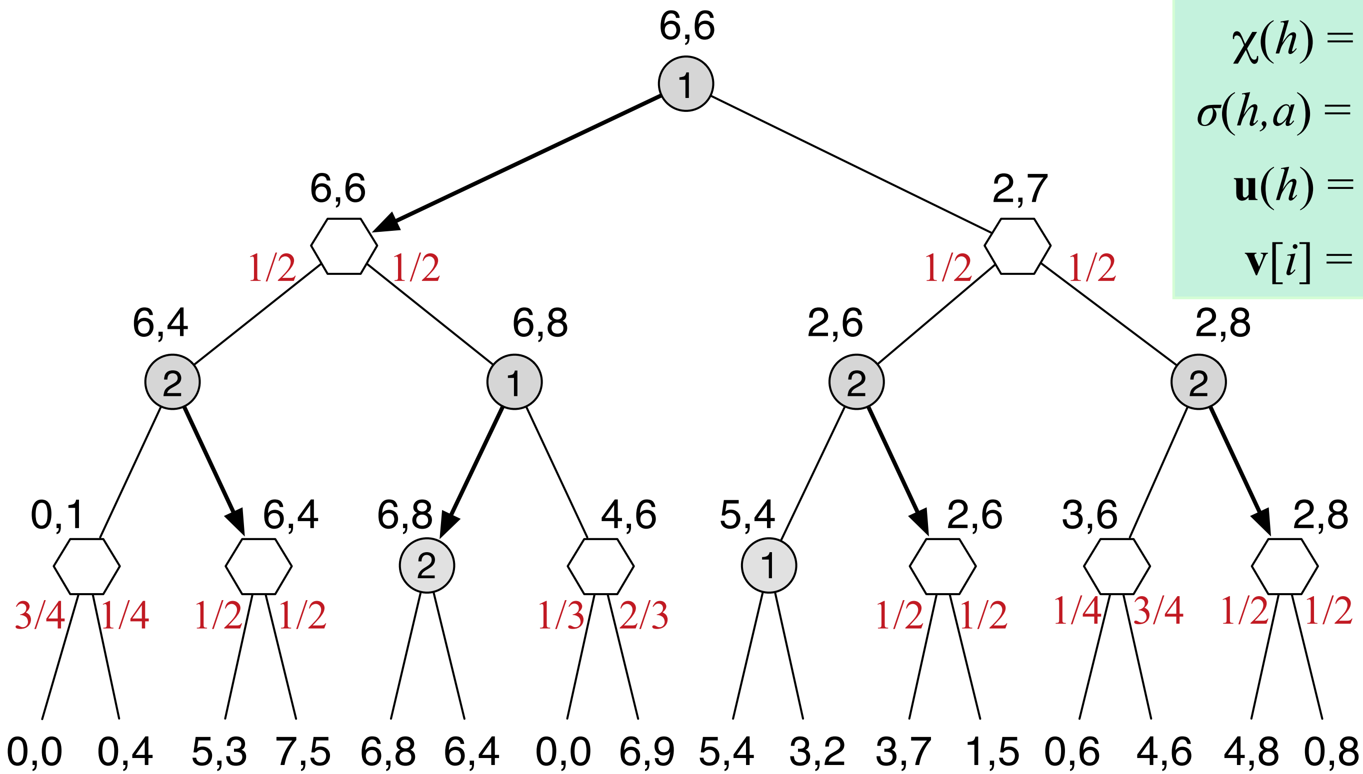
$\rho(h)$ = the player to move at h

$\chi(h)$ = {available actions at h}

$\sigma(h,a)$ = child of h produced by a

$\mathbf{u}(h)$ = utility profile for h

$\mathbf{v}[i]$ = i'th element of $\mathbf{v}$



function Expectimax(h,d)

if $h \in Z$ then return $\mathbf{u}(h)$

else if $d = 0$ then return $\mathbf{e}(h)$

if h is a chance node then return $\sum_{a \in \chi(h)} \Pr[a \mid h]$ Expectimax($\sigma(h,a), d-1$)

V = {Expectimax($\sigma(h,a), d-1$) | $a \in \chi(h)$}

return arg $\max_{\mathbf{v} \in V} \mathbf{v}[\rho(h)]$

Expectimax

Z = {terminal nodes}

$\rho(h)$ = the player to move at h

$\chi(h)$ = {available actions at h}

$\sigma(h,a)$ = child of h produced by a

$\mathbf{u}(h)$ = utility profile for h

$\mathbf{v}[i]$ = i'th element of $\mathbf{v}$

function Expectimax(h,d)

  if $h \in Z$ then return $\mathbf{u}(h)$

  else if $d = 0$ then return $\mathbf{e}(h)$

  if h is a chance node then return $\sum_{a \in \chi(h)} \Pr[a \mid h]$ Expectimax($\sigma(h,a), d-1$)

  V = {Expectimax($\sigma(h,a), d-1$) | $a \in \chi(h)$}

  return $\arg\max_{\mathbf{v} \in V} \mathbf{v}[\rho(h)]$

function Expectimax-choice(h,d)

  if $h \in Z$ then return *error*

  return $\arg\max_{a \in \chi(h)}$ (Expectimax($\sigma(h,a), d-1$))[$\rho(h)$]

- If $d \geq \text{height}(h)$ then
  - Expectimax returns the SPE payoff profile
  - Expectimax-choice returns the SPE action for player $\rho(h)$

Summary

- Finite two-player zero-sum perfect-information games
  - maxmin = minmax = Nash
  - only need to look at pure strategies
  - game-tree search
    - Minimax, LD-minimax, Alpha-Beta
    - limited search depth, static evaluation function
  - Monte Carlo roll-outs, UCT
- Multiplayer games
  - Max^n algorithm (depth-limited backward induction)
  - Paranoid algorithm (approximate maxmin)
- Games with chance nodes
  - Expectiminimax (2-player zero-sum)
  - Expectimax (n-player nonzero-sum)