Negamax & Proof Trees

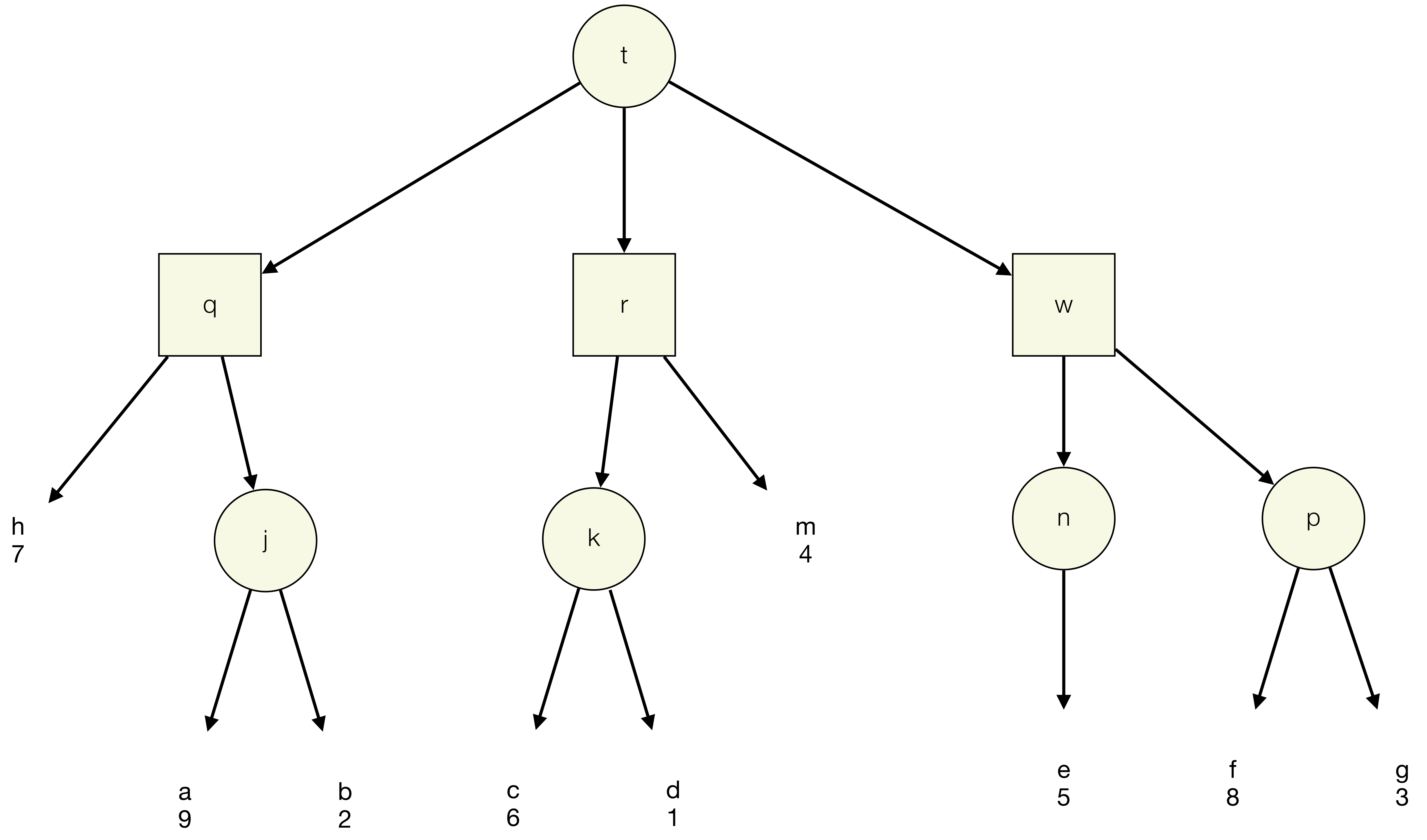
CMPUT 355: Games, Puzzles, and Algorithms

Lecture Outline

1. Recap
2. Alpha-beta search example
3. Negamax
4. Tic-tac-toe
5. Proof trees

Recap: Minimax Search

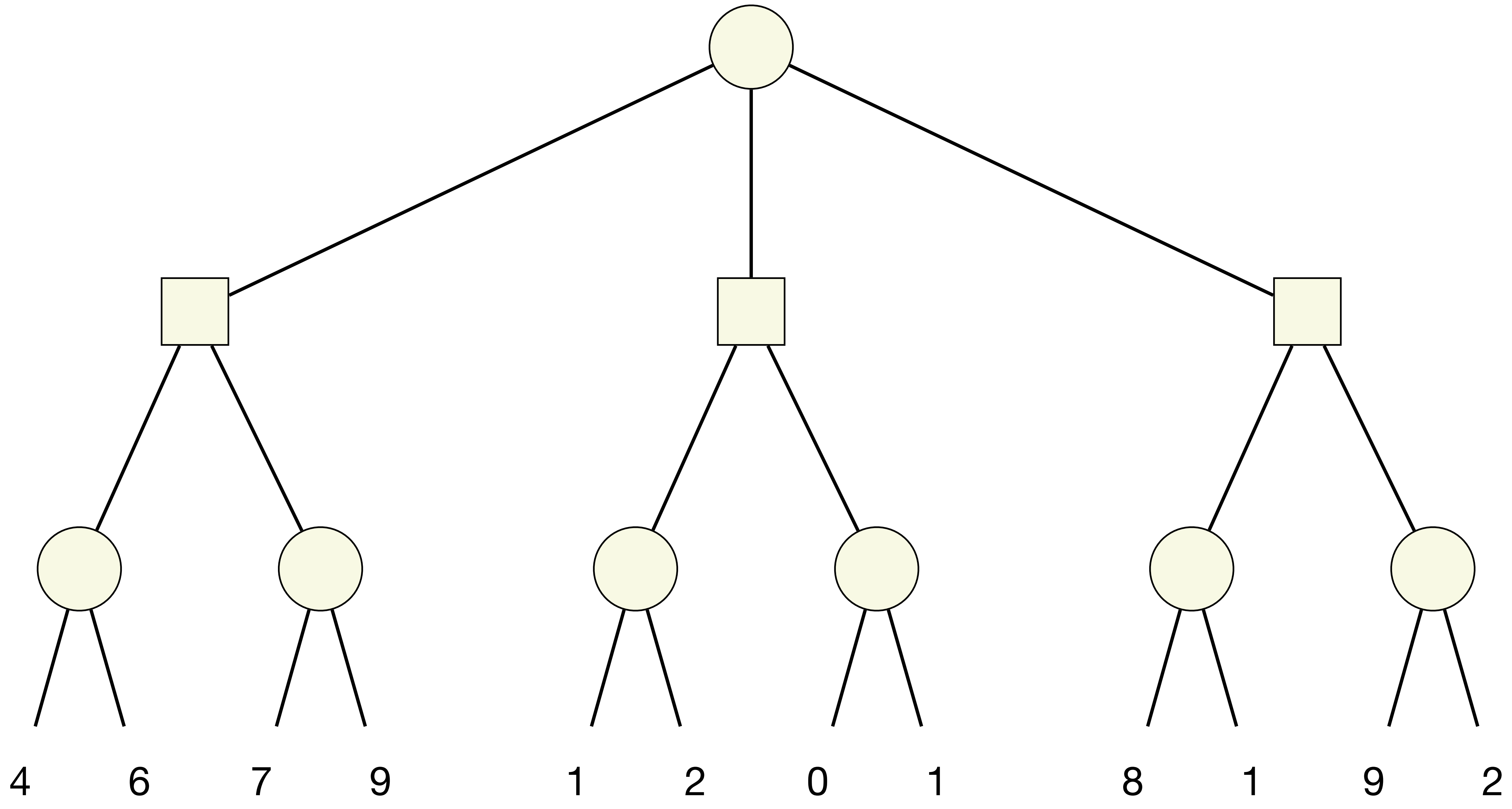
```
assume P1 plays at root
assume players alternate turns

def score(s):
    return P1's score at state s

def minimax(s):
    if terminal(s):
        return score(s)
    if player(s) == 1:
        return max{minimax(c) for
all c in children(s)}
    if player(s) == 2:
        return min{minimax(c) for
all c in children(s)}
```



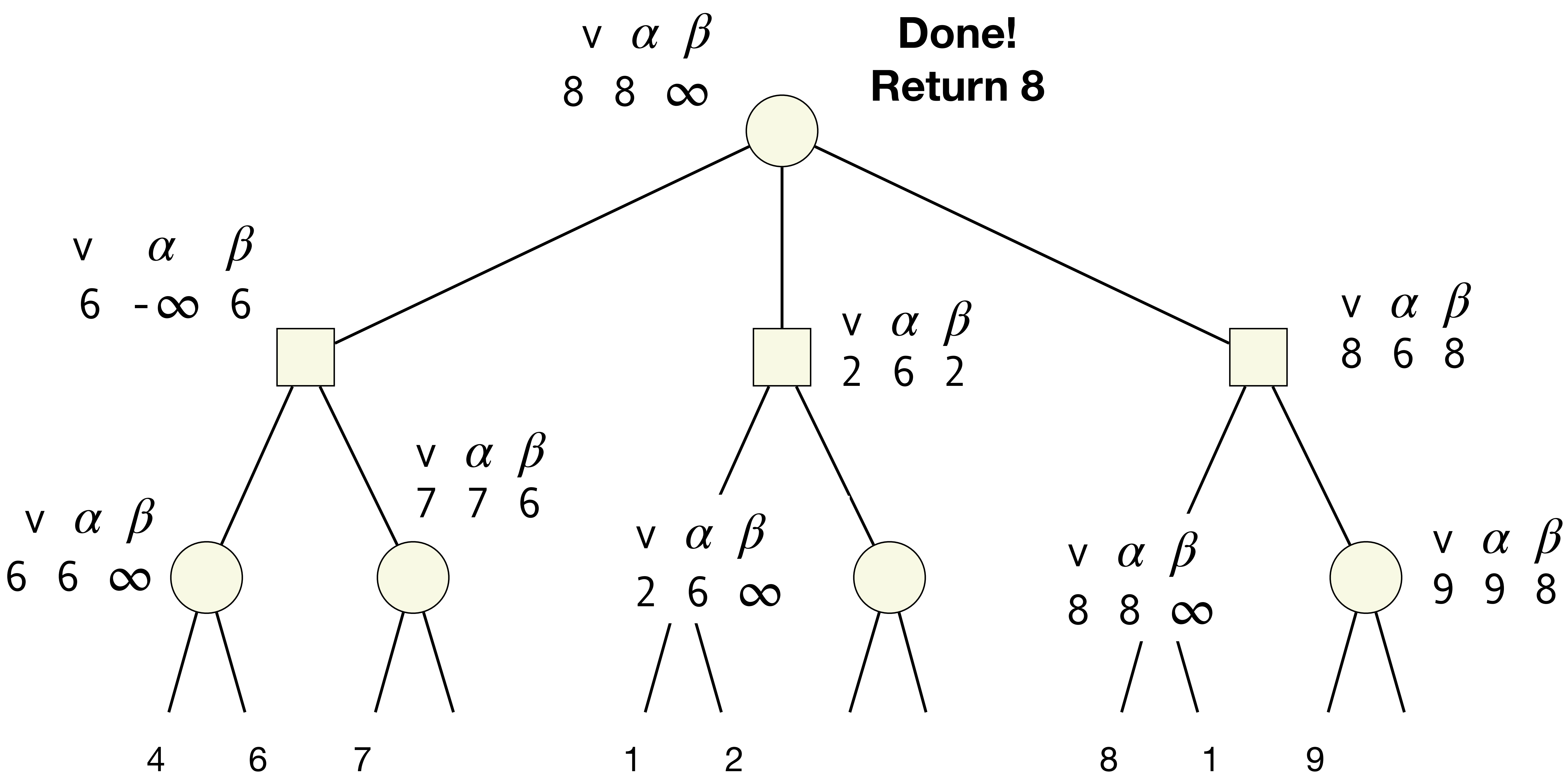
Recap: Alpha-Beta Search

```
def alphabeta(s, alpha, beta):
  if terminal(s):
    return score(s)
  if player(s) = 1: # MAX player
    val = -inf
    for c in children(s):
      ab = alphabeta(c, alpha, beta)
      alpha = max(alpha, ab)
      val = max(val, ab)
      if alpha >= beta:
        return val # prune remaining children
  if player(s) = 2:
    val = inf
    for c in children(s):
      ab = alphabeta(c, alpha, beta)
      beta = min(beta, ab)
      val = min(val, ab)
      if alpha >= beta:
        return val # prune remaining children
  return val
```

Alpha-Beta Search Example #2



Alpha-Beta Search Example #2



Negamax Pseudocode

```
assume P1 plays at root
assume players alternate turns

def score(s):
    return P1's score at state s

def minimax(s):
    if terminal(s):
        return score(s)
    if player(s) == 1:
        return max{minimax(c) for all c in children(s)}
    if player(s) == 2:
        return min{minimax(c) for all c in children(s)}
```

Minimax:

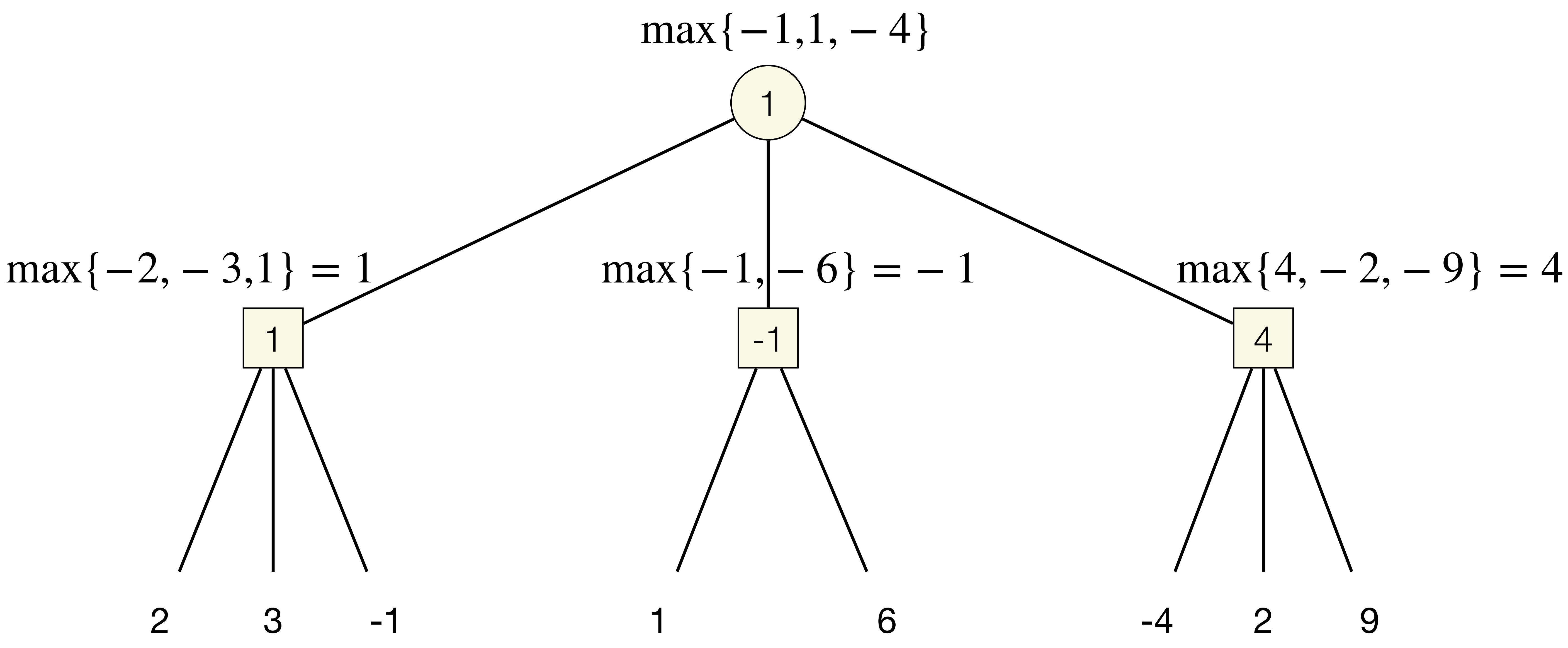
- Every terminal node labelled with P1's score
- Code for P1 and P2 is nearly identical

```
assume P1 plays at root
assume players alternate turns

def ptm_score(s):
    return player-to-move's score at state s

def negamax(s):
    if terminal(s):
        return ptm_score(s)
    return max{-negamax(c) for c in children(s)}
```

Negamax:

- Every terminal node labelled with player-to-move's score (**why?**)
- Only one case for nonterminal nodes

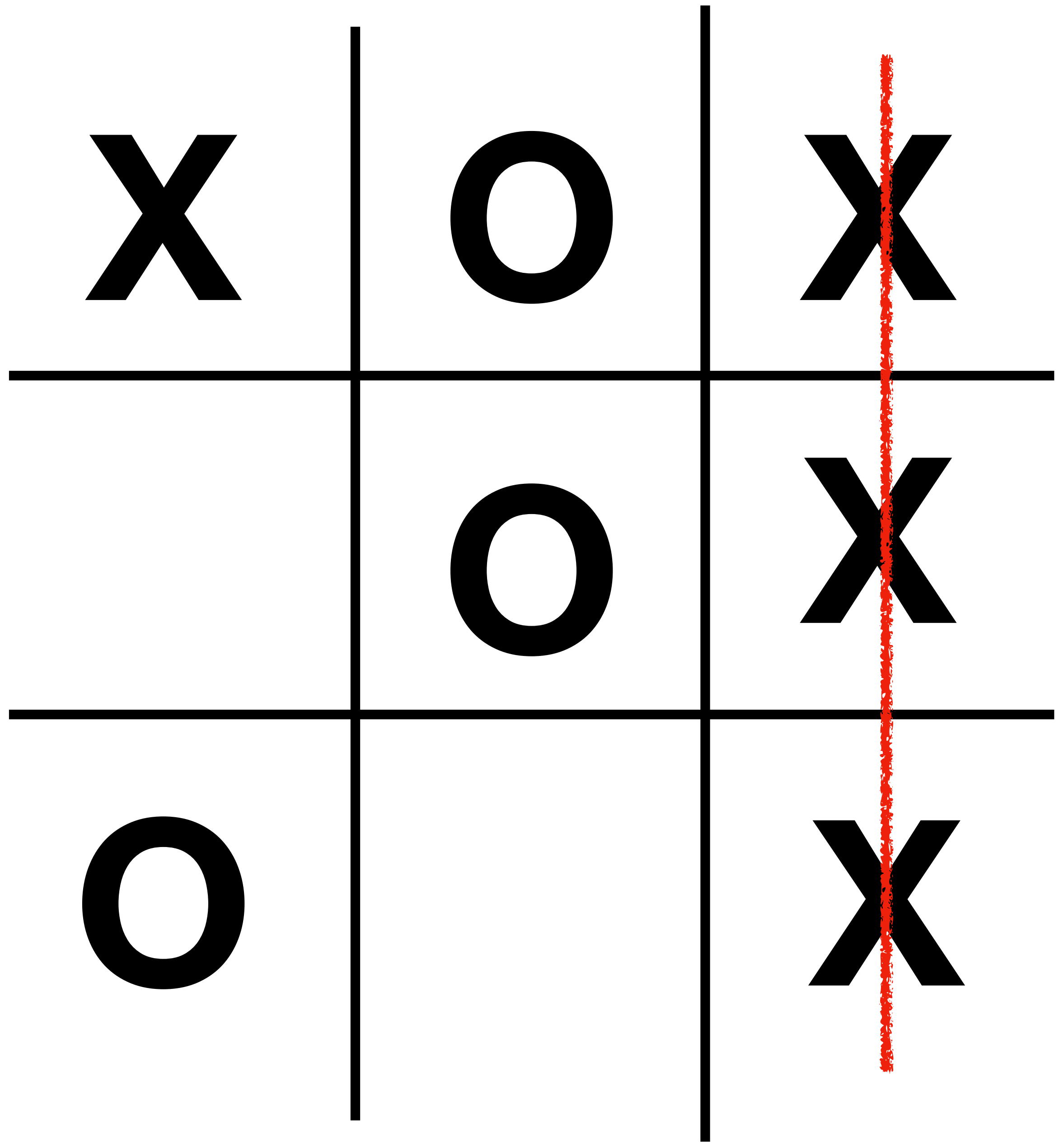
Negamax Example




```
assume P1 plays at root
assume players alternate turns

def ptm_score(s):
    return player-to-move's score at state s

def negamax(s):
    if terminal(s):
        return ptm_score(s)
    return max{-negamax(c) for c in children(s)}
```

Tic-Tac-Toe

- 9 cells
- Two players: P1 (X) and P2 (O)
- Alternating turns
- **Turn:** Place a mark in an empty cell
- First player to mark cells "in a row" **wins**
    - Horizontal, vertical, or diagonal line



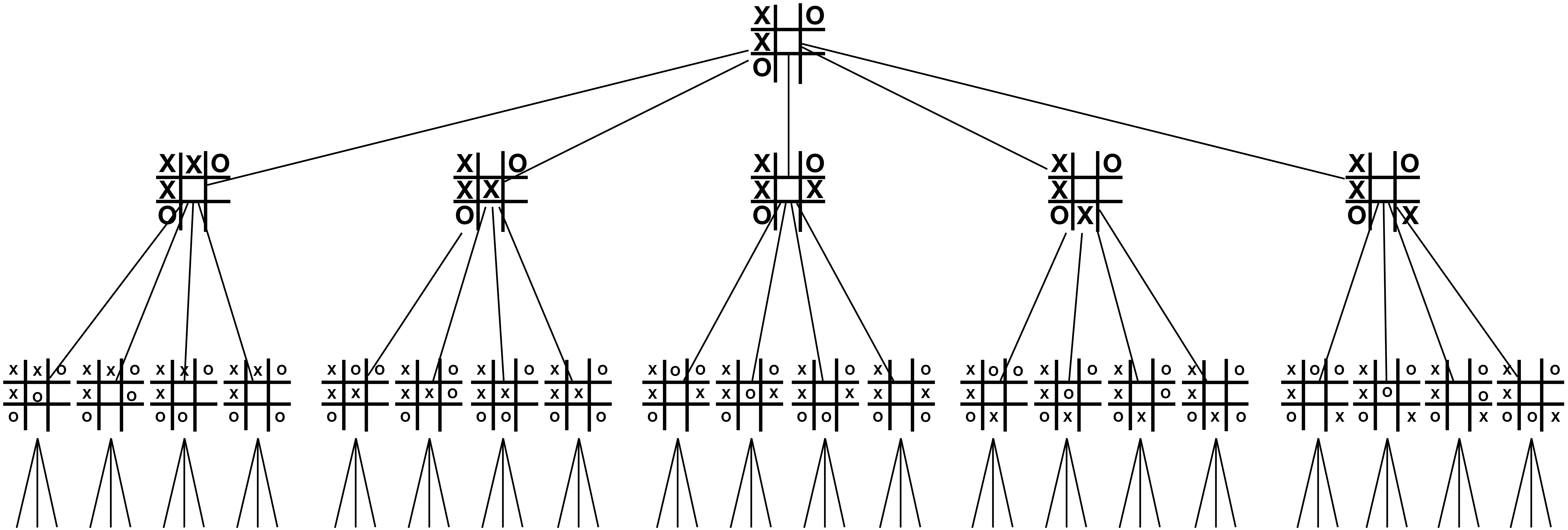
Implementation: `ttt/tt.py`

```python
def negamax(calls, d, psn, ptm): # 1/0/-1 win/draw/loss
  calls += 1
  psn_int = board_to_int(psn.brd)
  if psn.has_win(opponent(ptm)):
    return -1, calls  # previous move created win
  G = psn.legal_moves()
  if len(G) == 0:
    return 0, calls  # board full, no winner
  so_far = -1  # best score so far
  for cell in G:
    psn.brd[cell] = ptm
    nmx, c = negamax(0, d+1, psn, opponent(ptm))
    so_far, calls = max(so_far, -nmx), calls + c
    psn.brd[cell] = Cell.e  # reset brd to original
    # if so_far == 1: break  # improvement: return once win found
  return so_far, calls
```

Questions:

1. Why are we not checking whether `psm` has a win?
2. Could we do **alpha-beta** pruning with negamax?
3. Possible scores (1, 0, -1) are **known**; does this enable any **pruning**?

Tic-tac-toe Game Tree



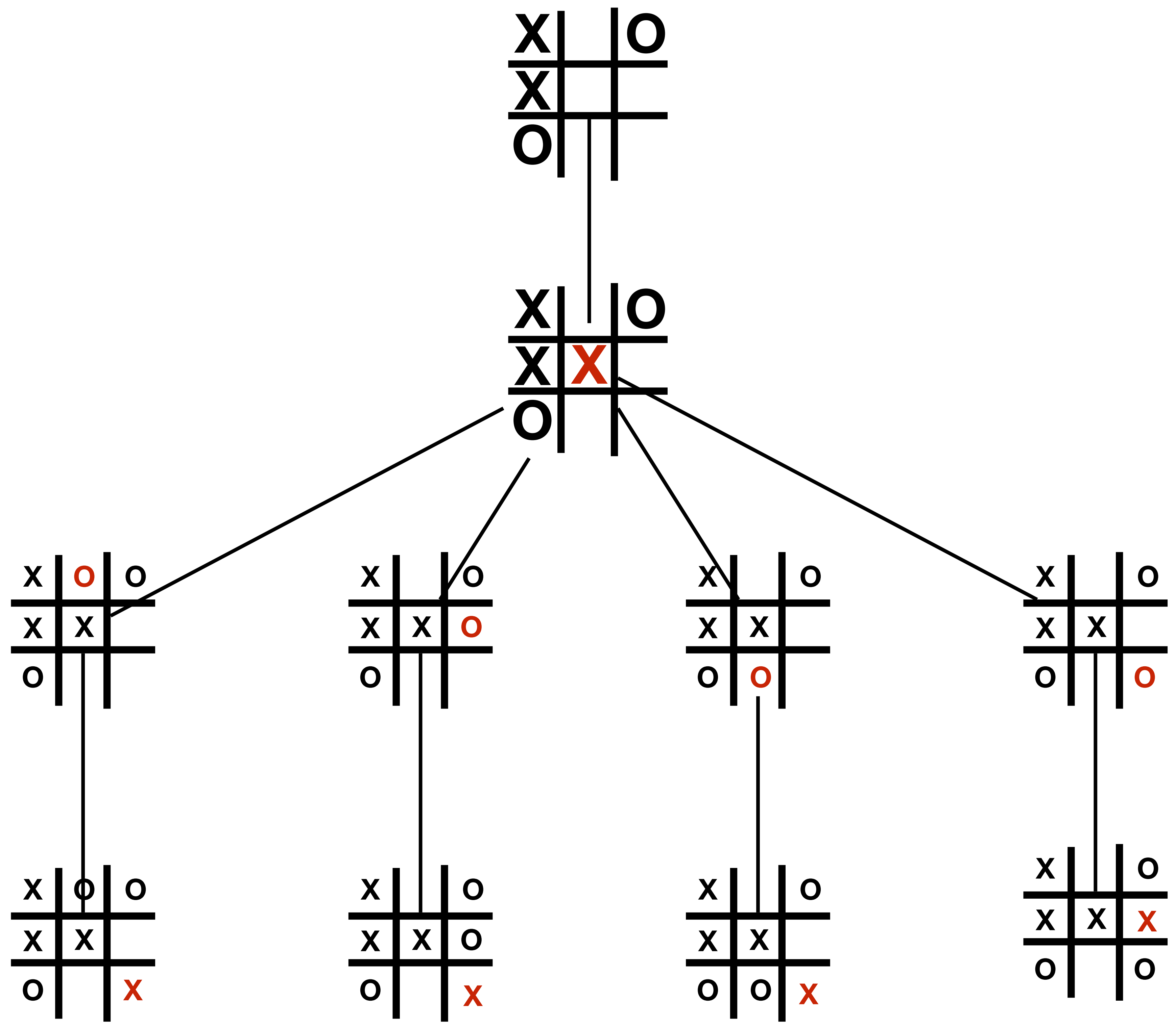
Questions:

1. **How many nodes** in the game tree rooted at this position?
   - Why do my calculations say "$\leq$"?
2. Do we need **all** of them to **prove** this position is a win for X? (**why or why not**?)

≤ 1 **node at root** $+$
≤ 5 **nodes at level 1** $+$
$\leq 5 \times 4$ **nodes at level 2** $+$
$\leq 5 \times 4 \times 3$ **nodes at level 3** $+$
$\leq 5 \times 4 \times 3 \times 2$ **nodes at level 4**$+$
$\leq 5 \times 4 \times 3 \times 2 \times 1$ **nodes at level 5** $= 326$

Tic-tac-toe Proof Tree

- A **proof tree** is the minimal subgraph of a game tree that proves a claim about a minimax value
- E.g.: claim that a position is a **win** for X
  - Each of X's positions contains **one child** (the winning move)
  - Each of O's positions contains **all children** (**why?**)



Summary

- **Minimax search** computes the **minimax value** for **every reachable state** in the game tree
    - Scores are all in terms of **Player 1** (the player-to-move of the root state)
    - When **P1** is player-to-move, **max** over recursive calls
    - When **P2** is player-to-move, **min** over recursive calls
- **Negamax search** computes the *same value*, but always in terms of **player-to-move's score**
    - **P2**'s scores are the **negative** of **P1**'s (because these are zero-sum games)
    - So negamax computes the max over the negative of recursive calls
    - Just one case instead of two nearly-identical cases
    - Every **leaf node** score is **player-to-move**'s score, not necessarily **P1**'s score
- **Proof tree** only contains enough of the game tree to demonstrate that one player can win
    - More generally, to prove the minimax value of the root
    - Only needs to contain a **single child** for each **winning player's** state (a winning move)
    - Still needs to contain **all children** for each of the **opponent's** states