CSE422: Artificial intelligence

Lab 03: Adversarial Search

Game playing using Minimax algorithm

Asif Shahriar
Lecturer, CSE, BRACU

WHY Adversarial Search

- So far we have seen TWO intelligent search techniques
- **Informed search (A-star search)**
  - Finds the optimal path to a **known** goal
- **Local search (Hill climbing, simulated annealing, genetic algorithm)**
  - Optimize a solution when the full search space is too large
  - No notion of path — only focuses on current state and neighbors
- Both search techniques assume:
  - The world **does not change** unless the agent acts
  - There is no **opponent** in the search space – agent is working alone
- In many real-world settings, the environment is **non-deterministic** and **competitive**
  - The outcome depends not only on your actions, but also on the opponent's actions
  - You are not just finding a path — you are **outsmarting an opponent**

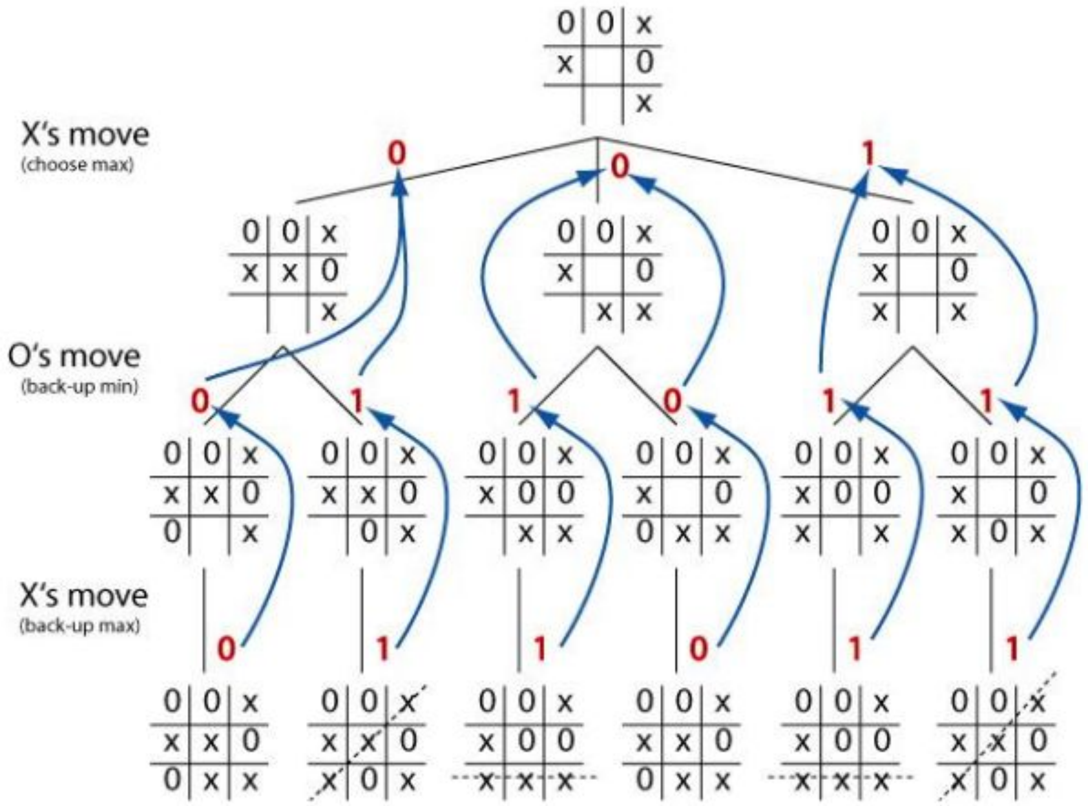
Let’s play a game - 1

- Suppose you and your friend both got caught doing something criminal and have been brought in for interrogation
- If both deny, both get 10 years sentence
- If both accept, both get 5 years sentence
- If one accepts and other deny, the one who denies goes free but the one who accepts get 10 years sentence
- What you gonna do?

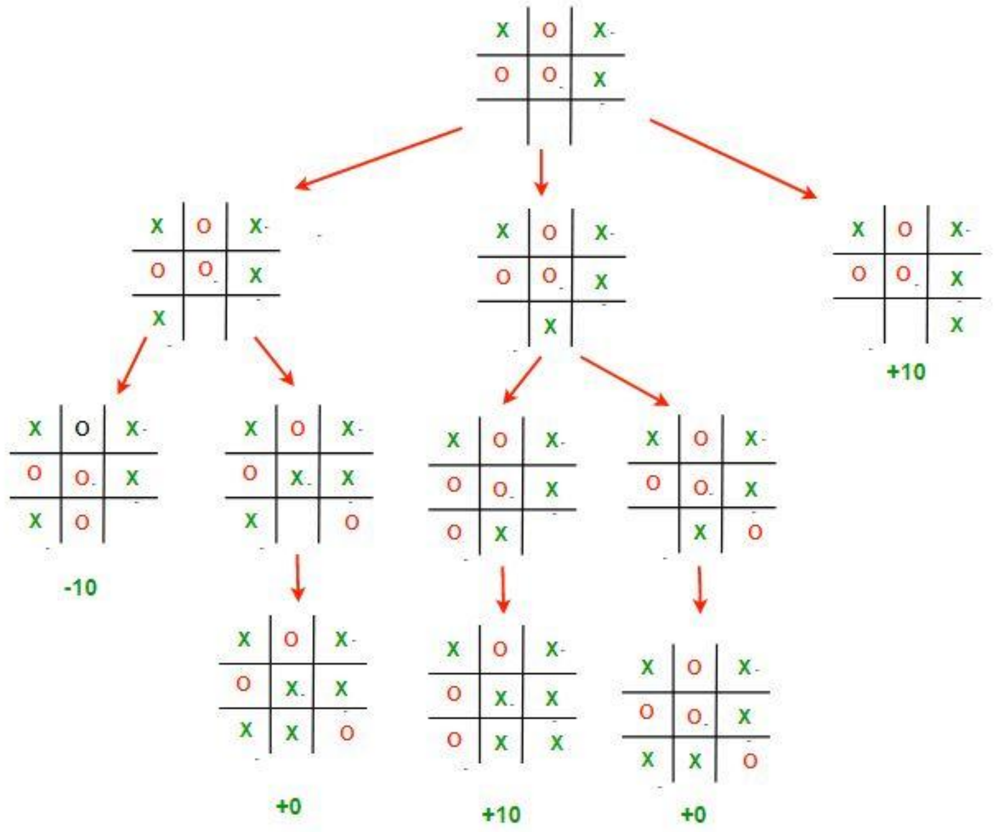
Let’s play a game - 2

- There are four coins and 2 players
- Each player can take ONE or TWO coins
- The one who takes the last coin, loses
- You can choose to go first or go second
- What you gonna do?

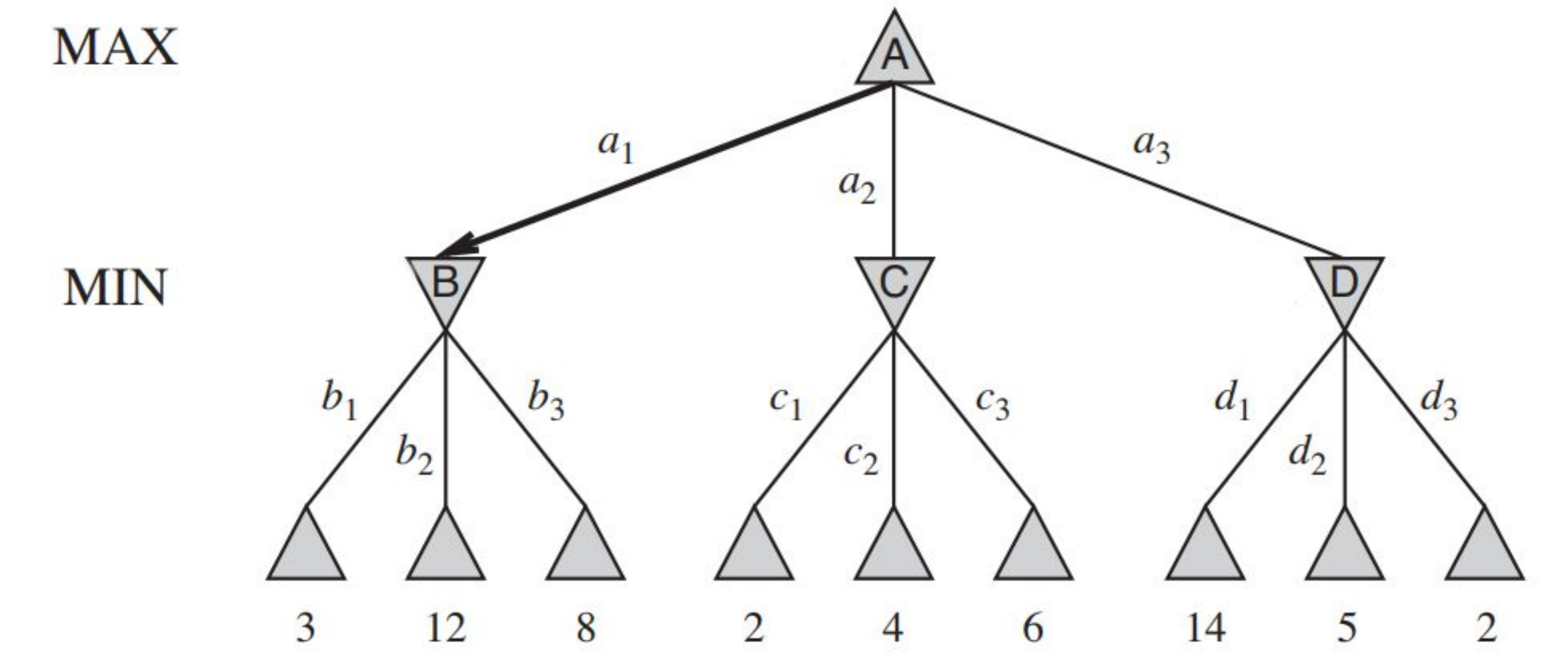
Let’s Play TIC-TAC-TOE



Let's Play TIC-TAC-TOE



Game Trees

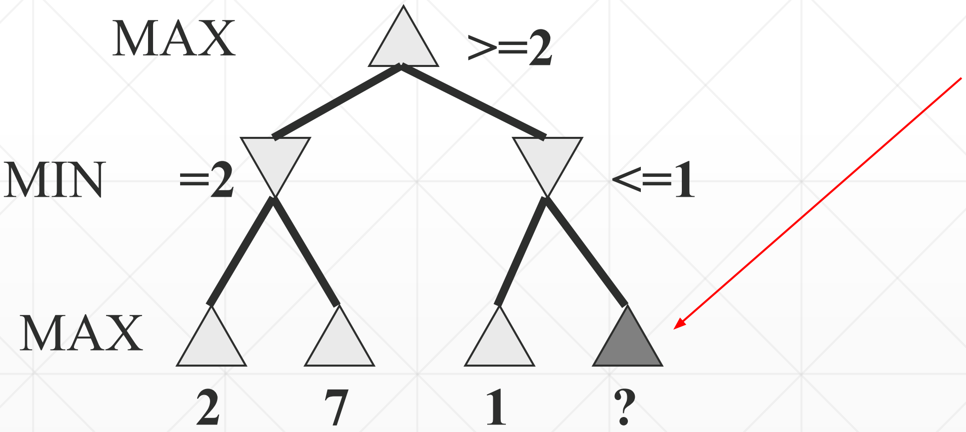


Minimax Algorithm

```
function minimax(position, depth, maximizingPlayer)
    if depth == 0 or game over in position
        return static evaluation of position

    if maximizingPlayer
        maxEval = -infinity
        for each child of position
            eval = minimax(child, depth - 1, FALSE)
            maxEval = max(maxEval, eval)
        return maxEval

    else
        minEval = +infinity
        for each child of position
            eval = minimax(child, depth - 1, TRUE)
            minEval = min(minEval, eval)
        return minEval


// initial call
minimax(currentPosition, 3, true)
```

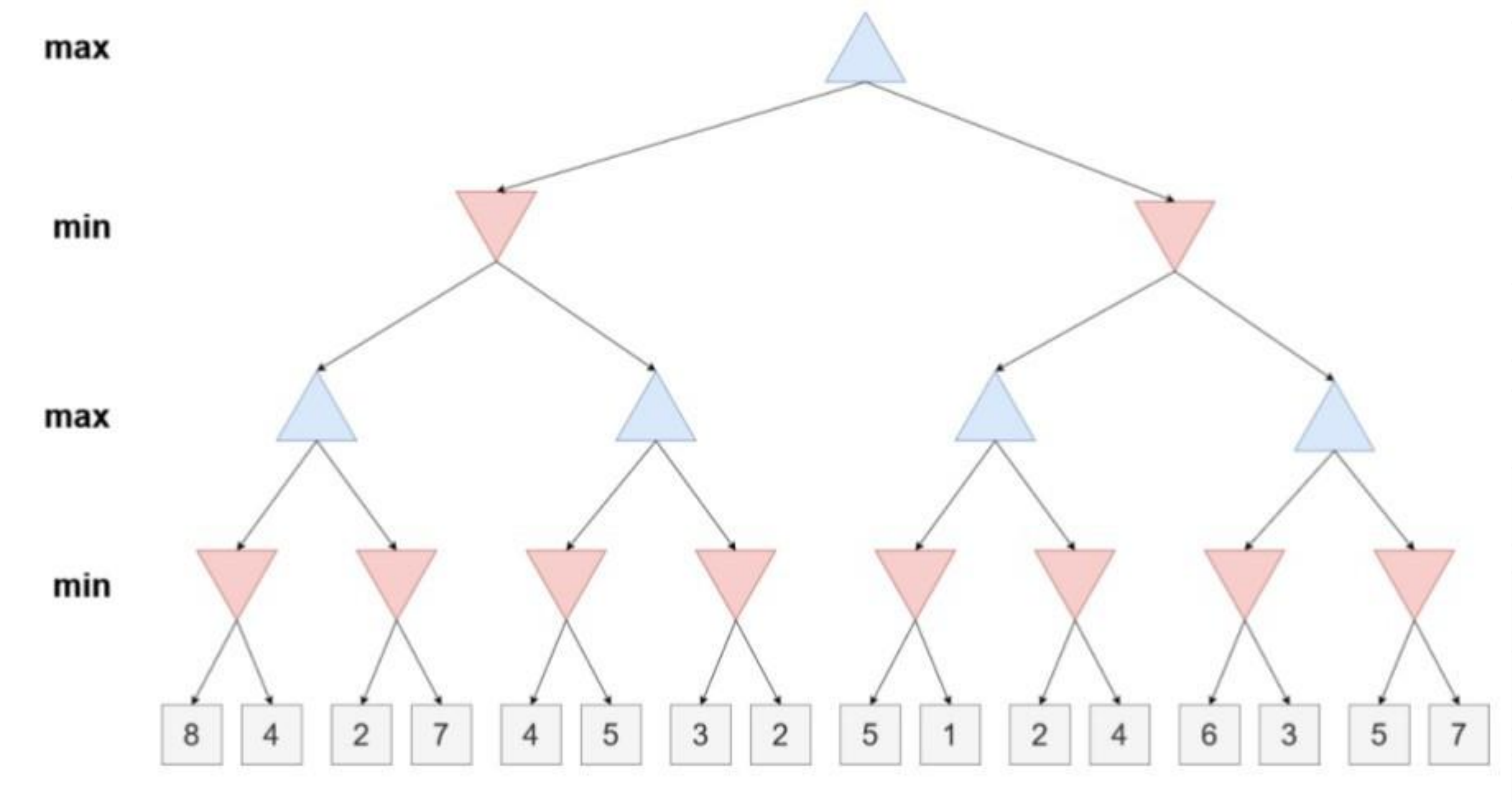
Alpha-beta Pruning

- Basic idea: “If you have an idea that is surely bad, don't take the time to see how truly awful it is.” -- Pat Winston
- In other words: We DO NOT need to check ALL moves

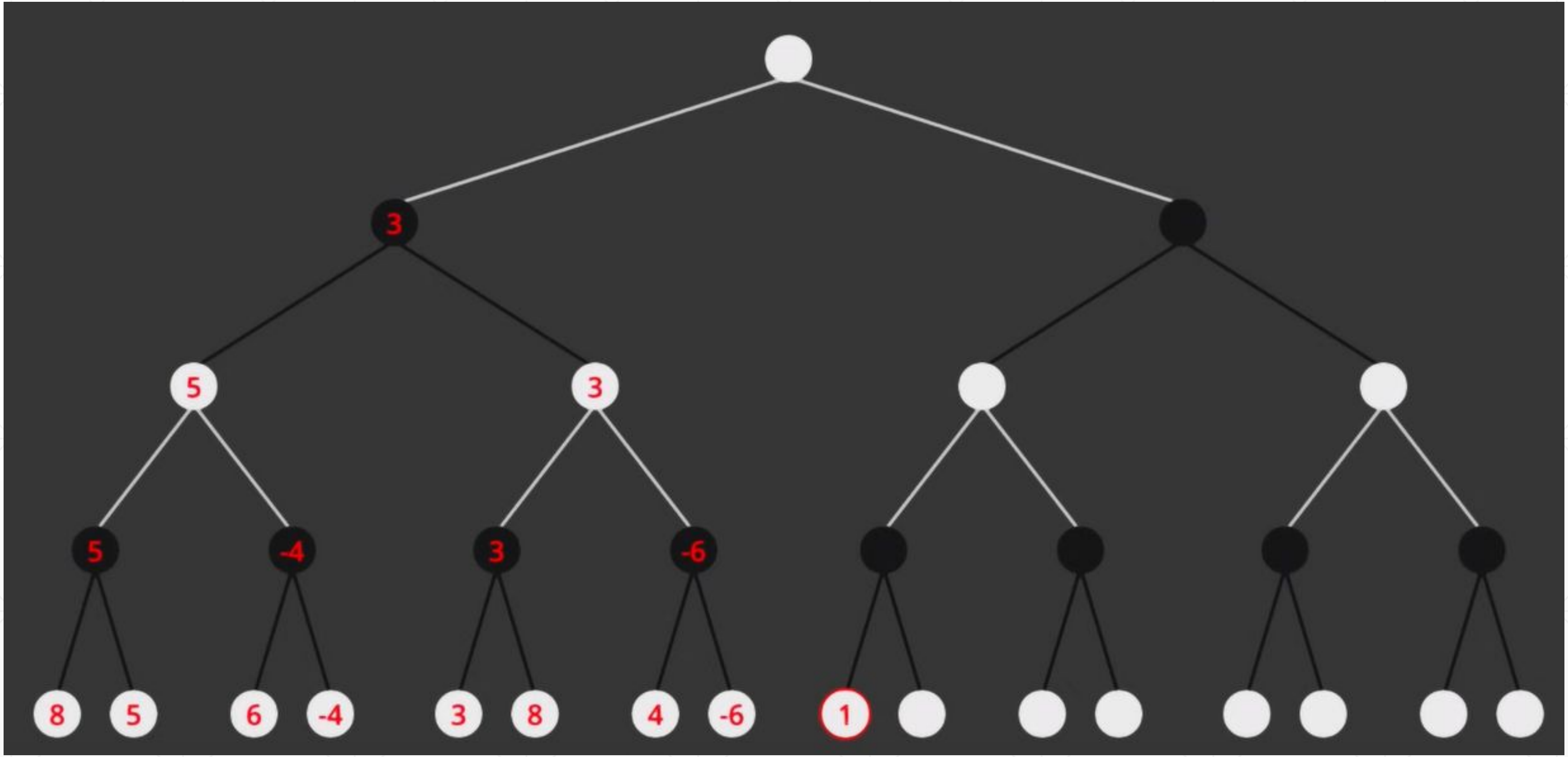


- We don’t need to compute the value at this node.
- No matter what it is, it can’t affect the value of the root node.

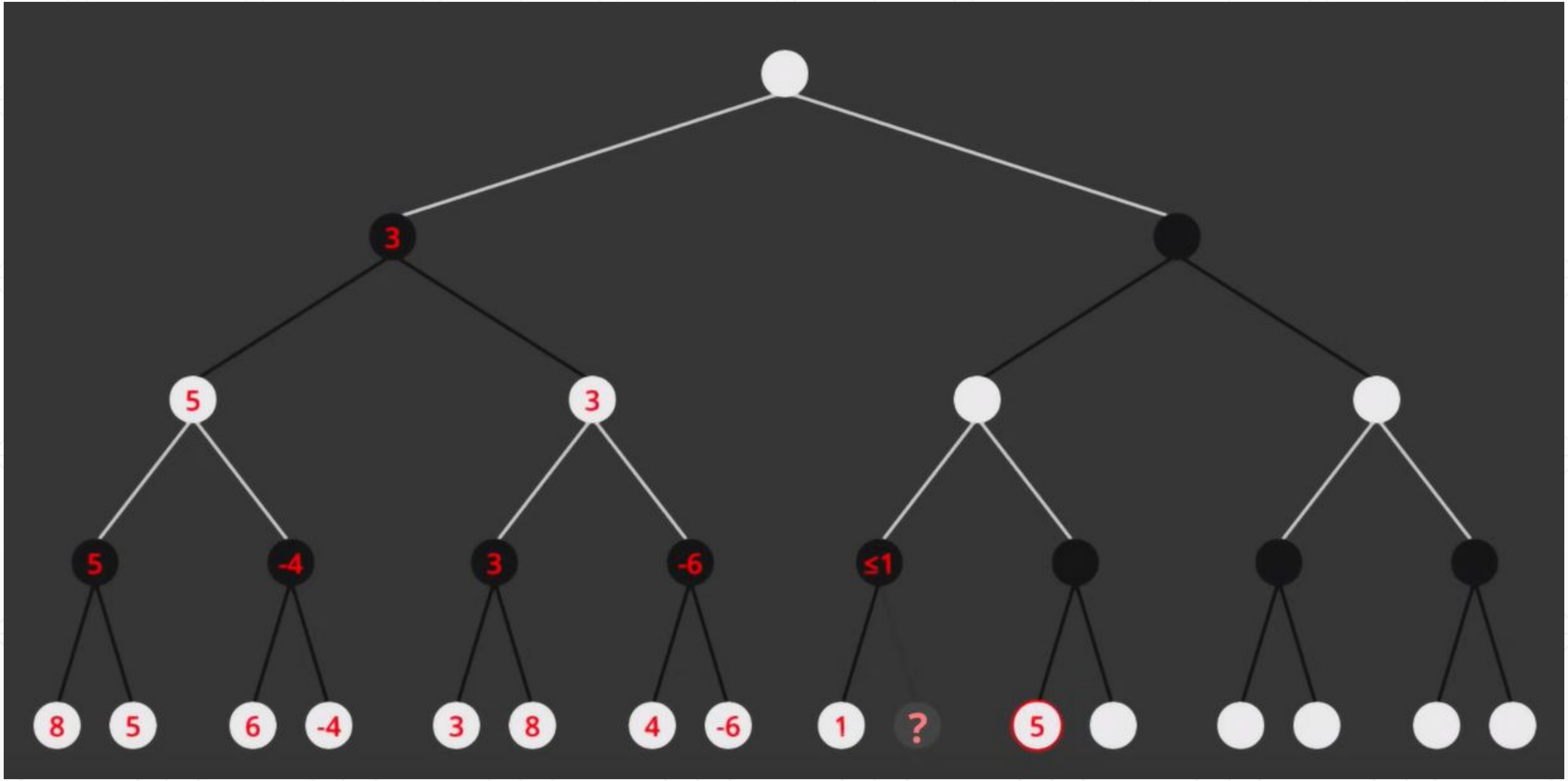
Alpha-beta Pruning Simulation - 3



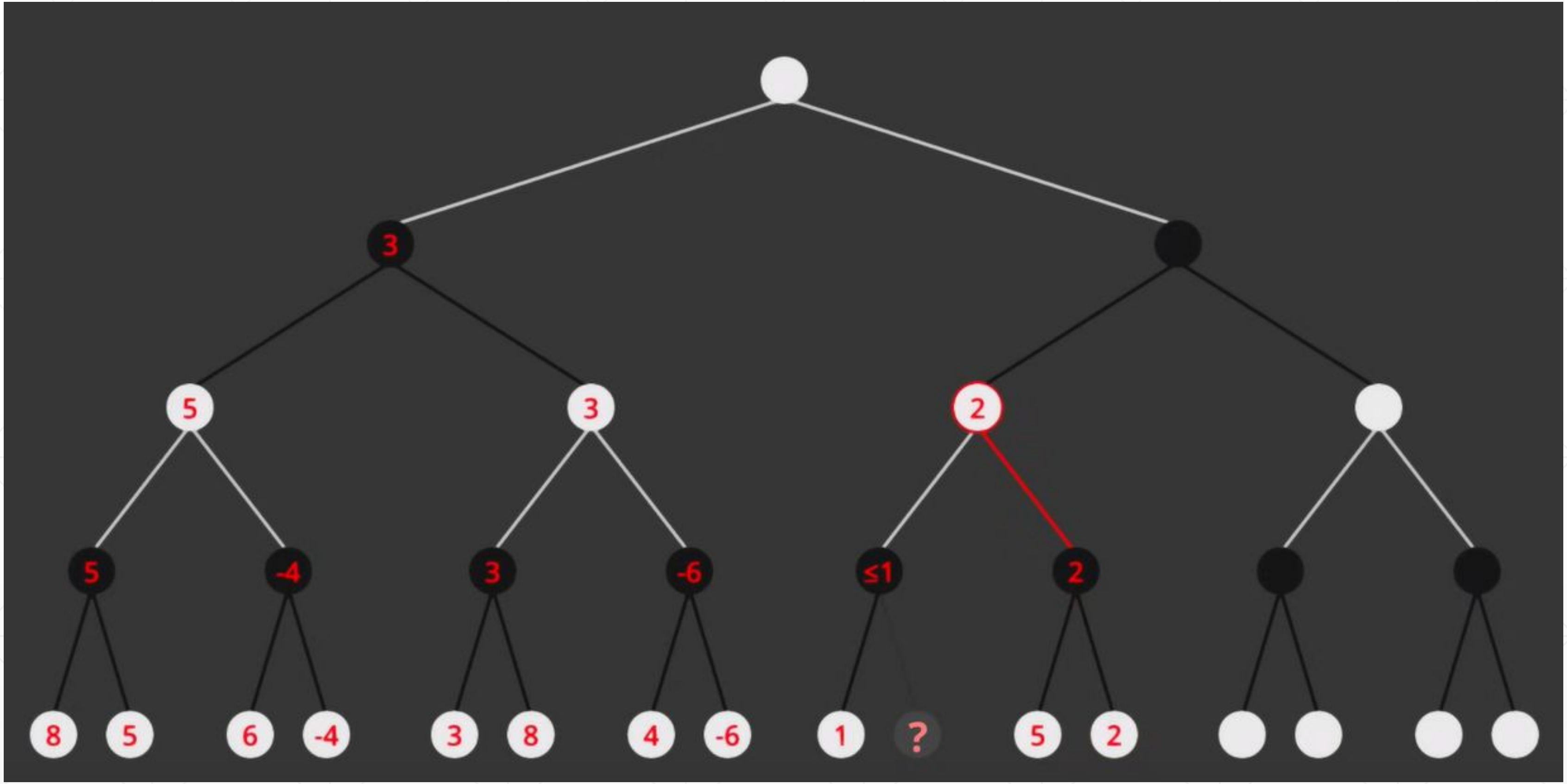
Alpha-beta Pruning



Alpha-beta Pruning



Alpha-beta Pruning

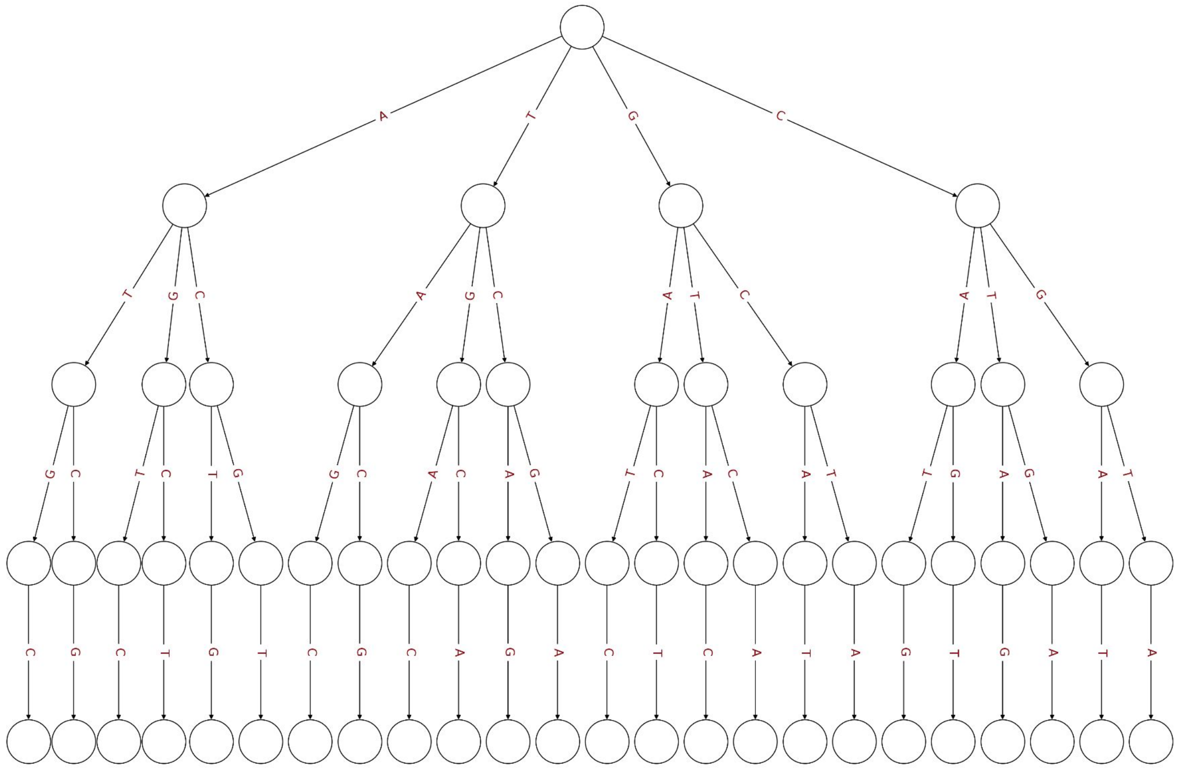


Alpha-beta Algorithm

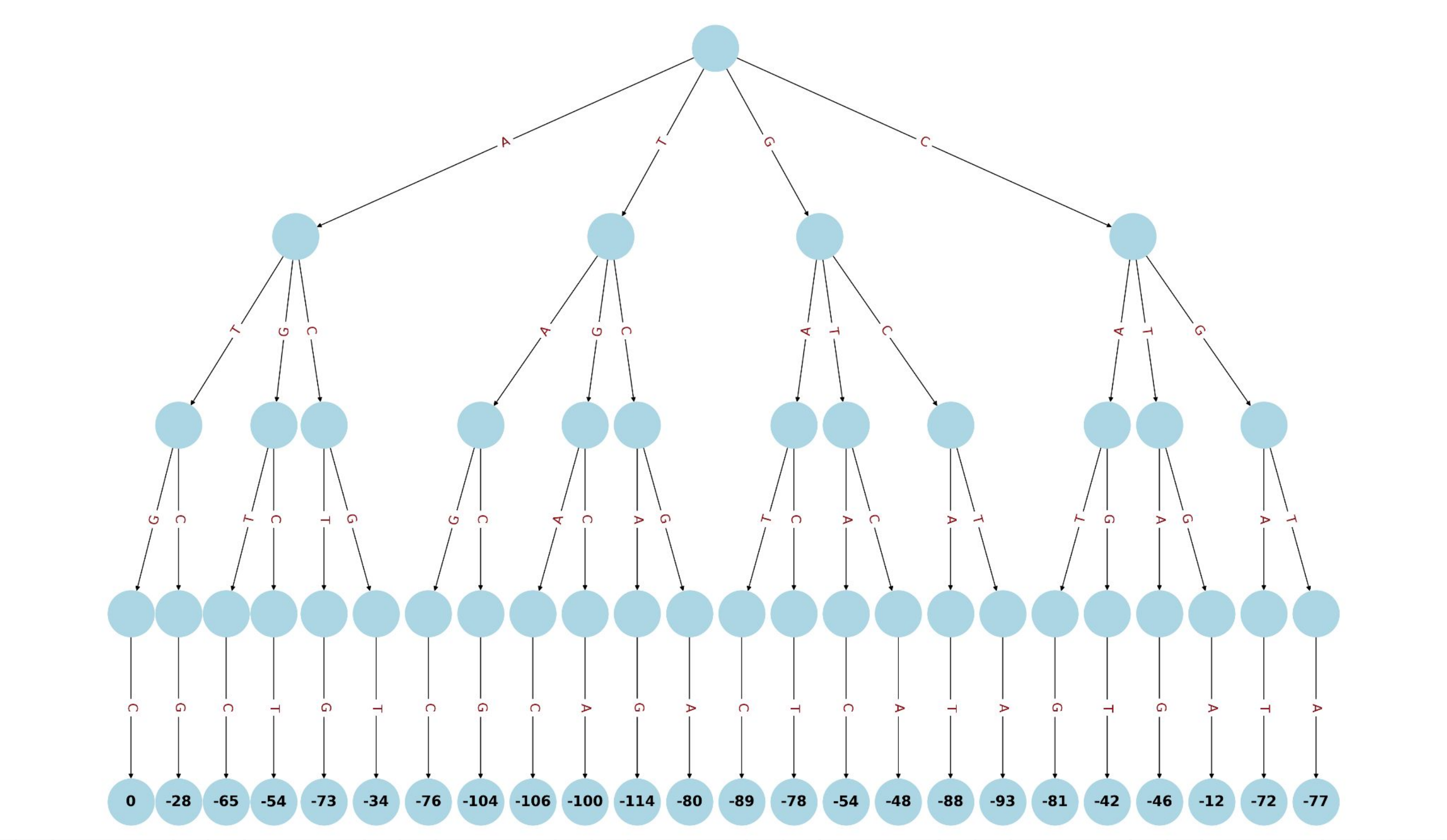
```
function minimax(position, depth, alpha, beta, maximizingPlayer)
    if depth == 0 or game over in position
        return static evaluation of position

    if maximizingPlayer
        maxEval = -infinity
        for each child of position
            eval = minimax(child, depth - 1, alpha, beta, false)
            maxEval = max(maxEval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha
                break
        return maxEval

    else
        minEval = +infinity
        for each child of position
            eval = minimax(child, depth - 1, alpha, beta, true)
            minEval = min(minEval, eval)
            beta = min(beta, eval)
            if beta <= alpha
                break
        return minEval
```

```
// initial call
minimax(currentPosition,3,-∞, +∞,true)
```

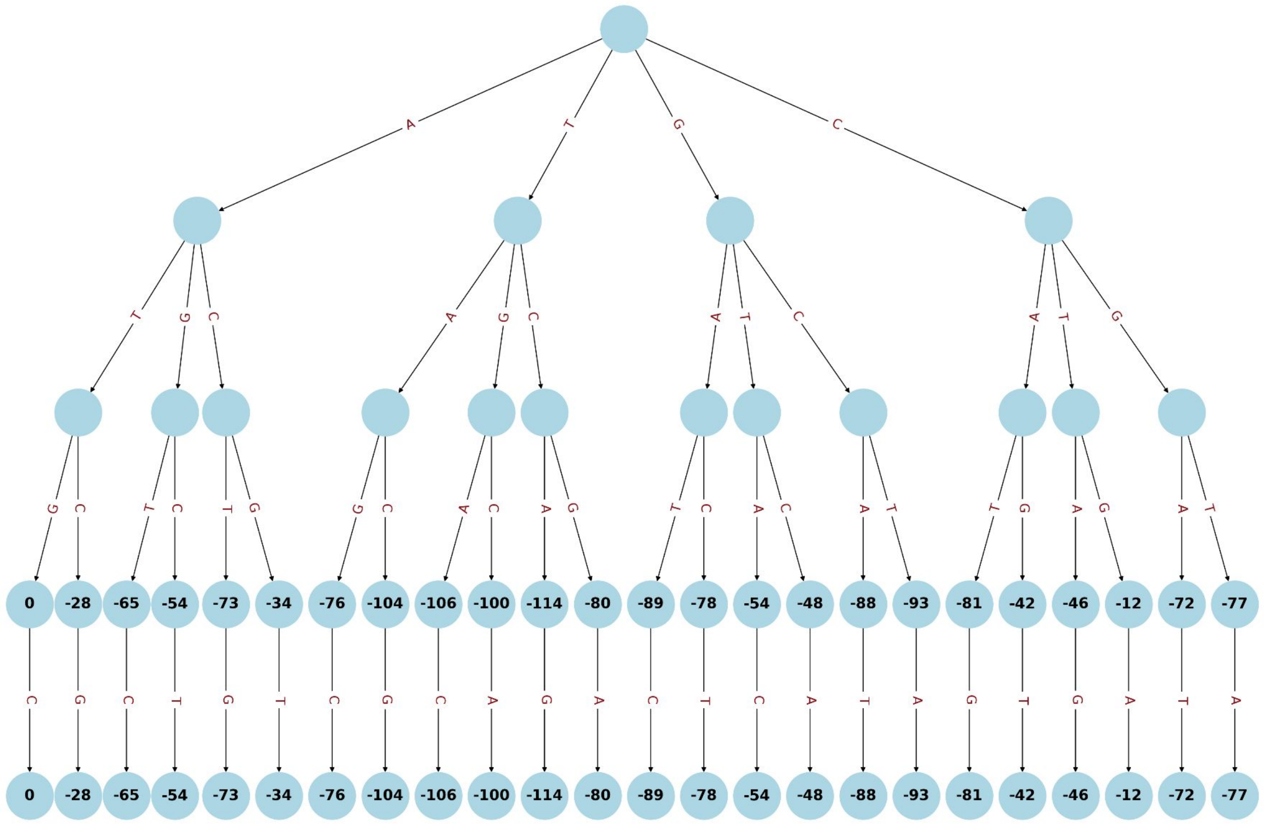
Nucleotide Tree



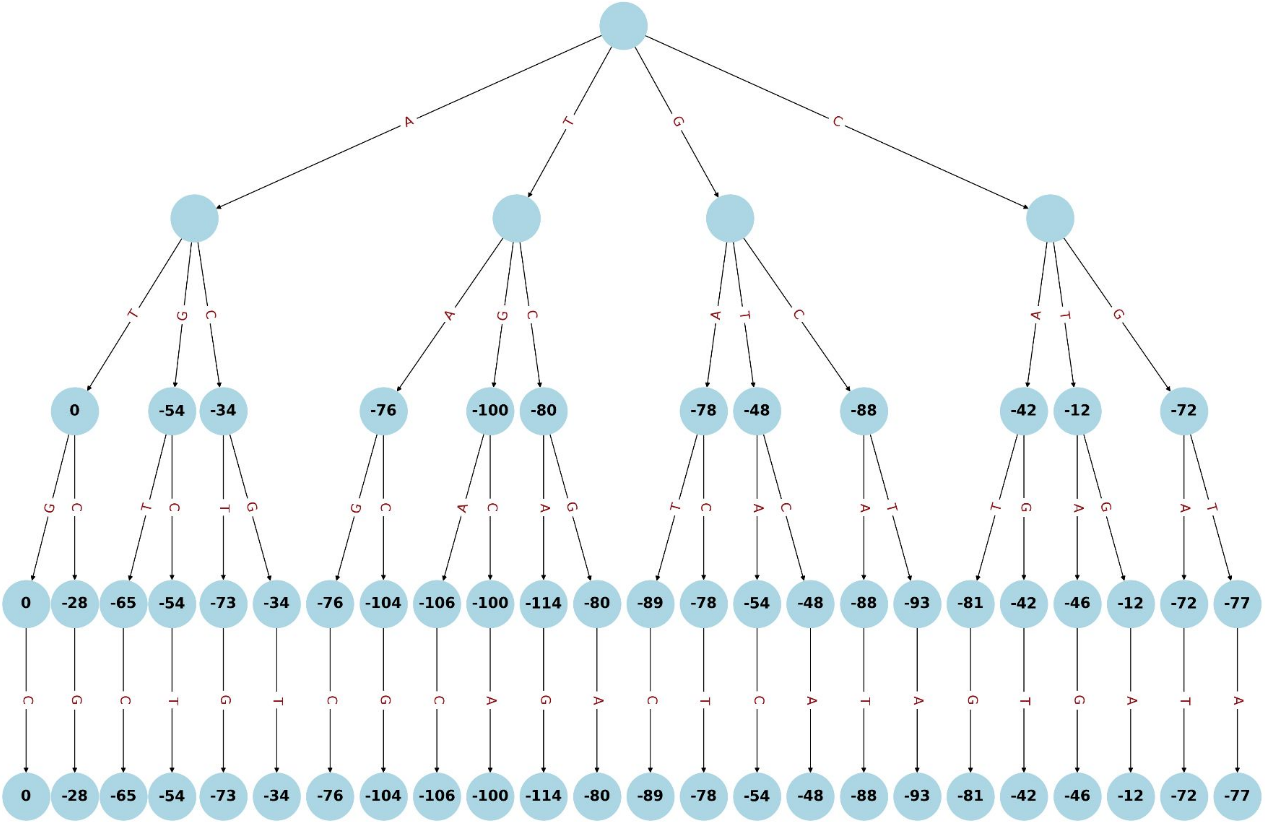
Nucleotide Tree



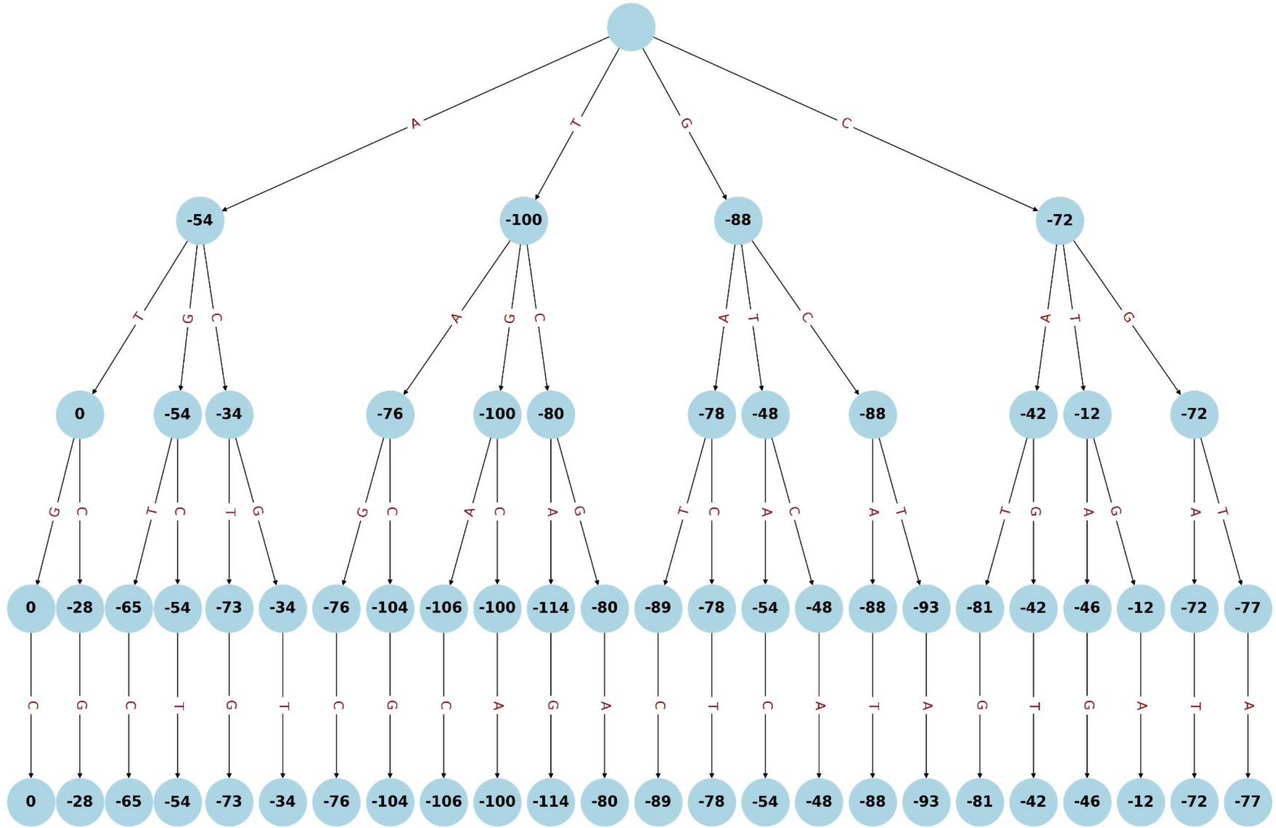
Nucleotide Tree



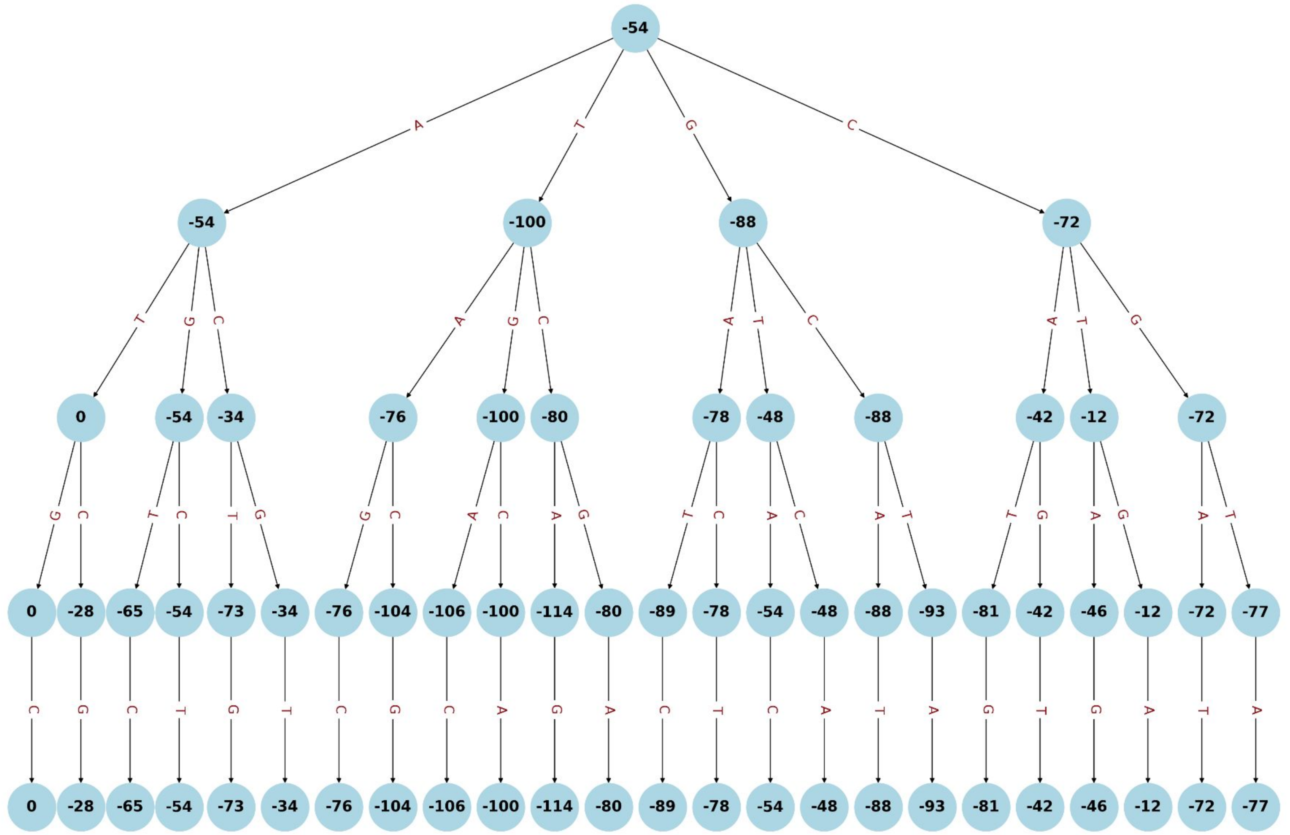
Nucleotide Tree



Nucleotide Tree



Nucleotide Tree



Nucleotide Tree - alpha beta pruned