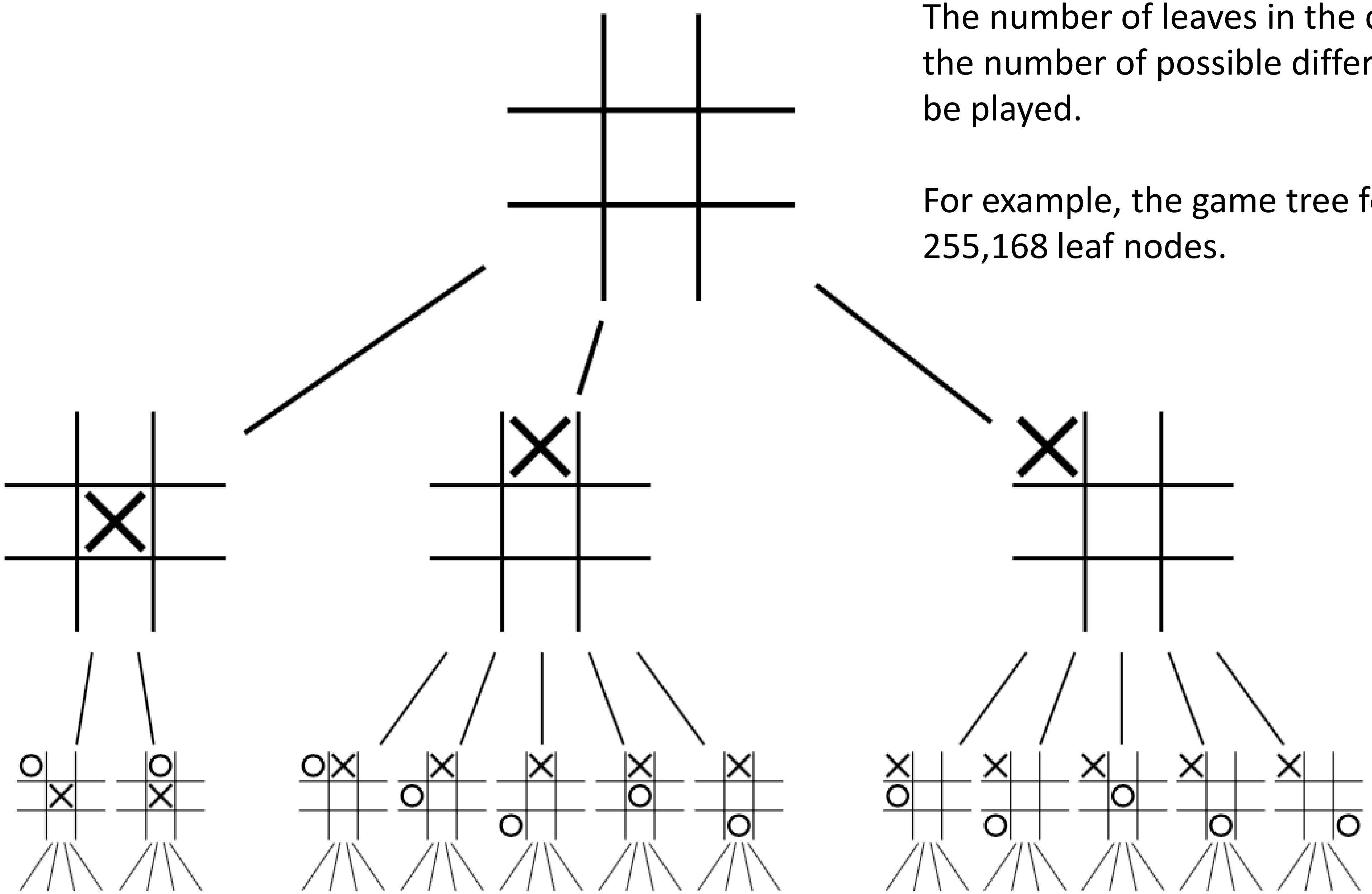
Game Trees

Game Tree

- A **game tree** is a tree, the nodes of which are positions in a game, and edges are moves.
- The **complete** game tree for a game is the game tree starting at the initial position and containing **all possible moves** from each position.



The number of leaves in the complete game tree is the number of possible different ways the game can be played.

For example, the game tree for tic-tac-toe has 255,168 leaf nodes.

https://en.wikipedia.org/wiki/Game_tree

Example: Nim

- Start with 15 pebbles
- To make a move you pick up 1, 2, or 3 pebbles
- Whoever picks up the last pebble loses
- e.g.,
    1. You pick up 3 (12 left)
    2. I pick up 3 (9 left)
    3. You pick up 2 (7 left)
    4. I pick up 2 (5 left)
    5. You pick up 1 (4 left)
    6. I pick up 3 (1 left)
    7. You pick 1 (0 left)
    8. I win!

Example: Nim

- Nim has a quite clear winning strategy!
- Consider Nim with 5 pebbles.
- If it’s my turn and there are 5 pebbles left then I lose:
  - if I take 1, you take 3, and there is 1 left;
  - if I take 2, you take 2;
  - if I take 3 you take 1.
- Similarly: 9…5, 13...9, etc.
- The pattern:
  - if it’s my turn and #pebbles mod 4 $\cong$ 1 then you have a winning strategy: so if you can, always leave the number of pebbles congruent to 1 mod 4.

Example: Nim

- Nim is special: a quick calculation tells who will win if players play optimally.
- For a game like chess (or Go or Connect 4 or …) you can’t tell (in constant time) just by looking at a board who will win.
- So, we must use computation to search possible future states.
- Build a game tree where nodes are states and edges are moves.
- Each row is labelled with the player whose turn it is.

Minimax (for 2-player games)

Minimax is a standard AI strategy for (2-player) games that are

- **alternating**: players take turns
- **deterministic**: each move has a well-defined outcome (there is no randomness)
- **perfect-information**: each player knows the complete state of the game (there is no hidden information)
- **zero-sum**: if I win you lose, and vice versa — what’s good for me is bad for you (but draws are allowed)

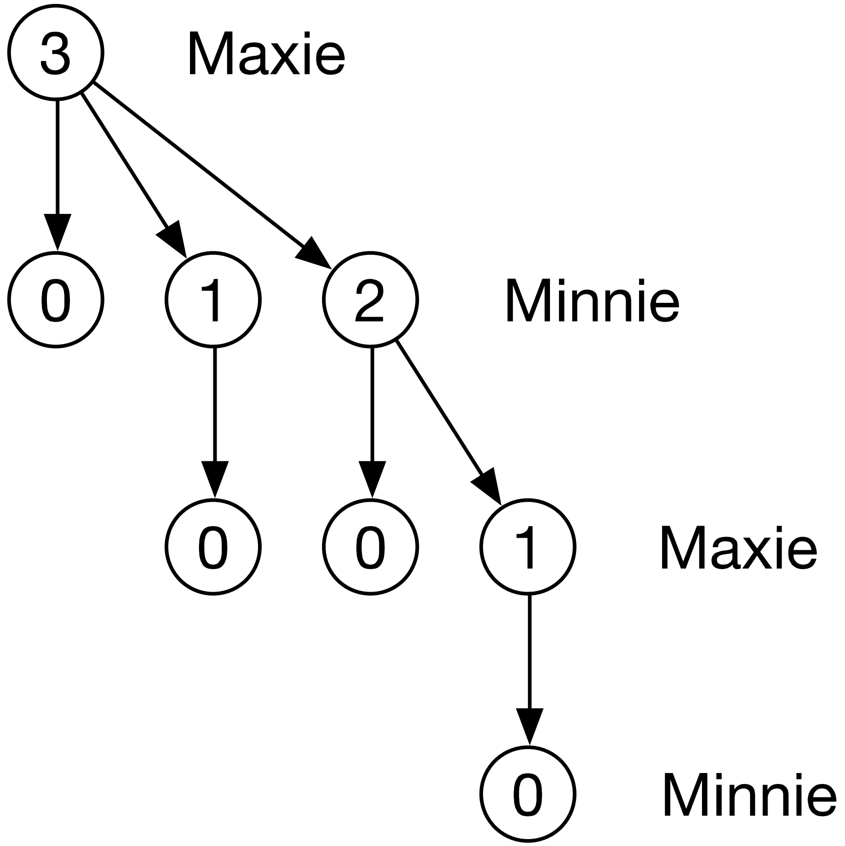
Minimax

A **minimax algorithm** is a recursive algorithm choosing the next move in a two-player game.

- A value is associated with each final state of the game (leaf of the game tree). It indicates how good it would be for me to finish with that position.
- Then for any state whose children all have values, if it is my move I will choose the move with the best value, so I give this state the **maximum** value from its children.
- If it is my opponent’s move, I assume they will choose the move with the worst value (for me), so I give this state the **minimum** value.
- This principle propagates recursively up the tree, until we reach the root. I then choose the child of the root with the best value.

Example: Nim

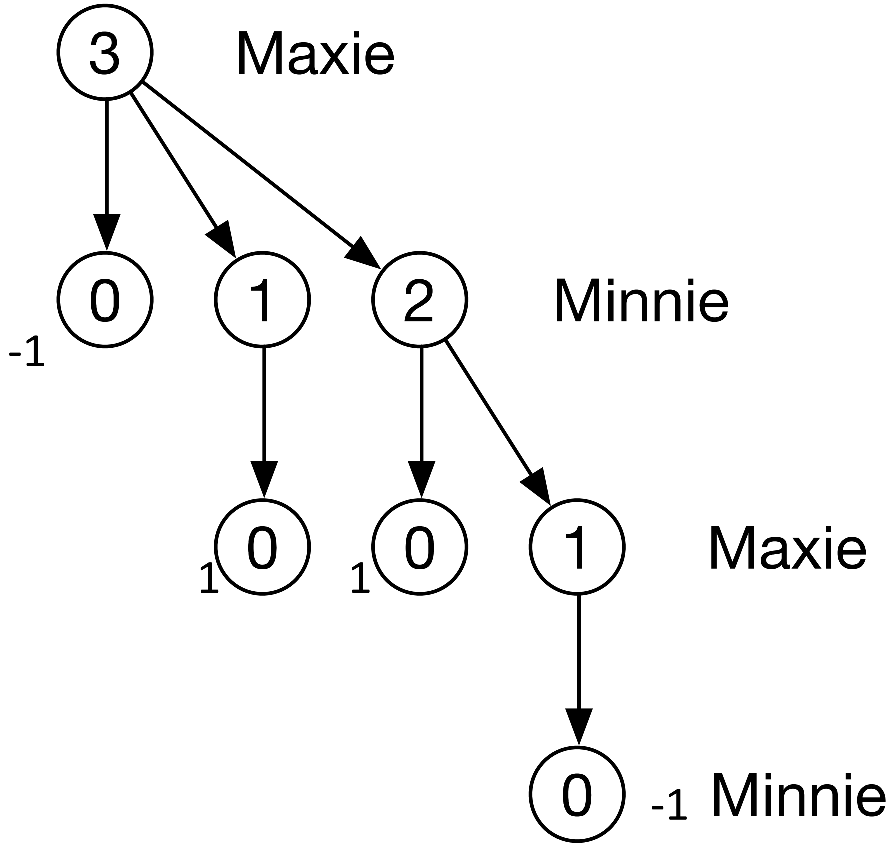
- Starting with 3 pebbles:



Example: Nim

- Next assign each leaf a value, saying who wins.

- Maxie wins if the value is 1.
- Minnie wins if the value is -1.

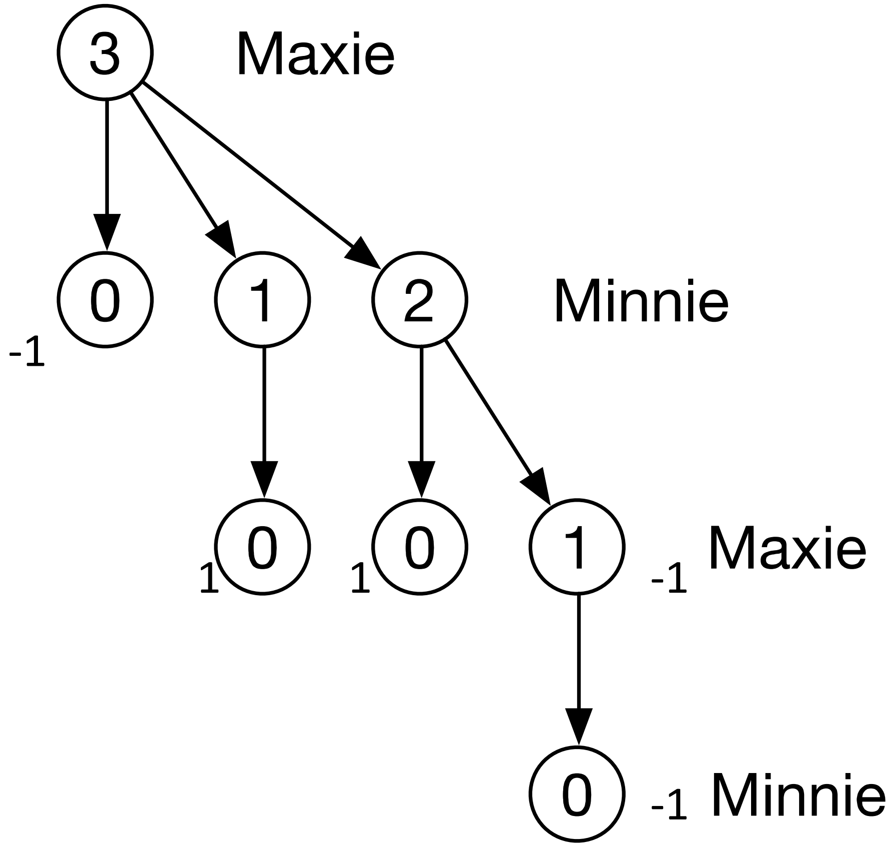


Example: Nim

- Then propagate the labels up the tree.

- If it's Maxie's turn the value is the maximum of the children (because Maxie will choose the maximising move).
- If it's Minnie's it is the minimum.

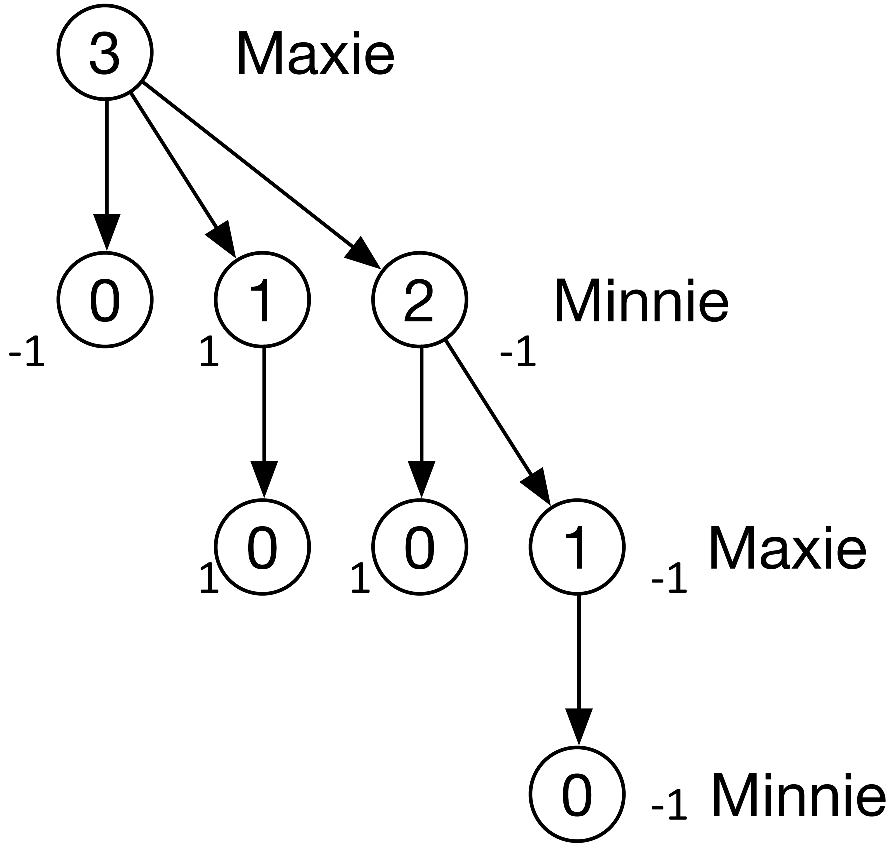
- First level.



Example: Nim

- Next level.

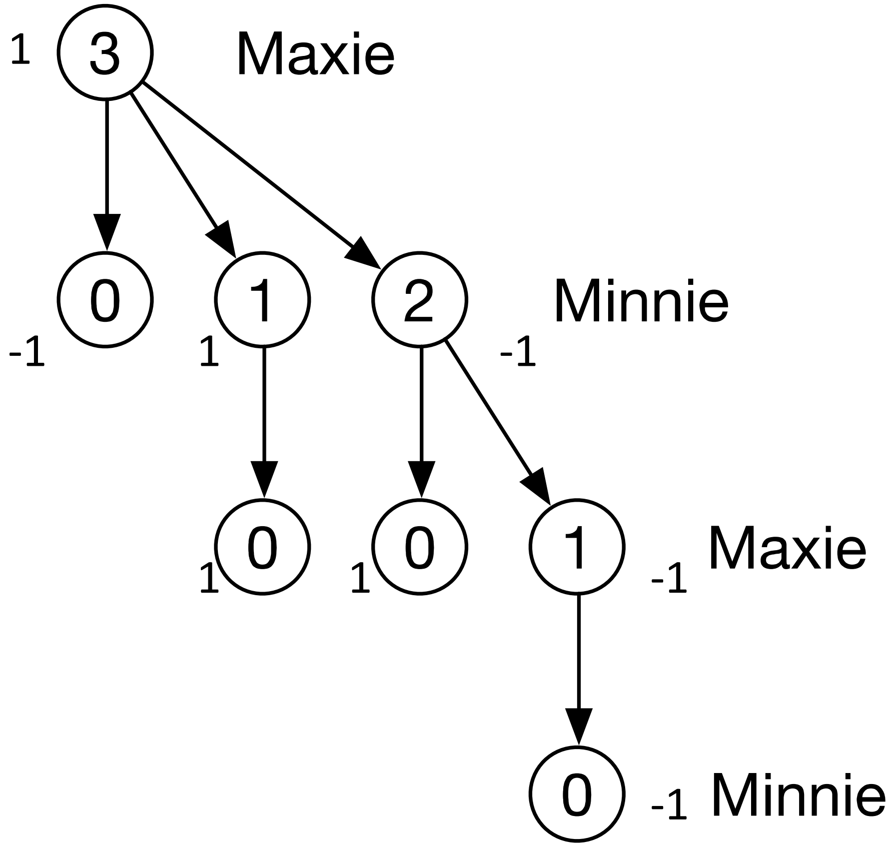
- For example, on the rightmost subtree if there are two pebbles left then Minnie should take 1, leaving 1, rather than 2, leaving 0.



Example: Nim

- Next level.

- Maxie should take 2, leaving 1, rather than taking 3 or 1.



Minimax algorithm

- **Minimax** computes the value of a game state assuming both players will play optimally.
- It does not account for things like
  - “that chess board is chaotic, so it will be easy to make a mistake”
- For a game with a bigger search space than Nim we **can’t draw out the whole tree**!
- Instead, a **heuristic** must **approximate** the value of non-final states.
- Nim has a perfect heuristic: number of pebbles congruent to 1 mod 4, i.e.   # pebbles \`mod\` 4 = 1.

Heuristic

https://en.wikipedia.org/wiki/Heuristic_(computer_science)

- A **heuristic** is a technique designed for solving a problem more quickly when classic methods are too slow, or for finding an approximate solution when classic methods fail to find any exact solution.
- A **heuristic function** is a function that *ranks alternatives* in search at each branching step based on available information to decide which branch to follow.

Trade-off criteria for deciding a heuristic

- **Optimality:** When several solutions exist for a given problem, does the heuristic guarantee that the best solution will be found? Is it actually necessary to find the best solution?
- **Completeness:** When several solutions exist for a given problem, can the heuristic find them all? Do we actually need all solutions? Many heuristics are only meant to find one solution.
- **Accuracy and precision:** Can the heuristic provide a confidence interval for the purported solution? Is the error bar on the solution unreasonably large?
- **Execution time**: Is this the best known heuristic for solving this type of problem? Some heuristics converge faster than others. Some heuristics are only marginally quicker than classic methods.

Heuristics for games

- A heuristic for a game is an assignment of a value to the state of an unfinished game, indicating how well the game is going for you.
- Many games (e.g. most team sports) have a running score. Are these always a good heuristic?
- For chess, a good heuristic would include which pieces are left, where they are positioned, etc.
- So designing a heuristic requires a strong understanding of the game, and much experimentation.

Minimax algorithm

The overall algorithm is:

1. explore the game tree up to a certain depth (**lookahead**)
2. use the heuristic to give a value to all leaves, where the game has finished *or* the lookahead has been reached
3. assign any node whose children all have values the *maximum* or *minimum* value of its children, depending on whose turn it is
4. recursively propagate values up to the children of the root, then choose the child of the root with the best value