Adversarial Search

Stephen G. Ware

CSCI 4525 / 5525

Narrative Intelligence LAB

THE UNIVERSITY *of* NEW ORLEANS

Search So Far

Observable:	Fully
Agents:	Single
Deterministic:	Deterministic
Episodic:	Sequential
Static:	Static
Discrete:	Discrete

Adversarial Search

Observable:	Fully
Agents:	Multi
Deterministic:	Deterministic
Episodic:	Sequential
Static:	Static
Discrete:	Discrete

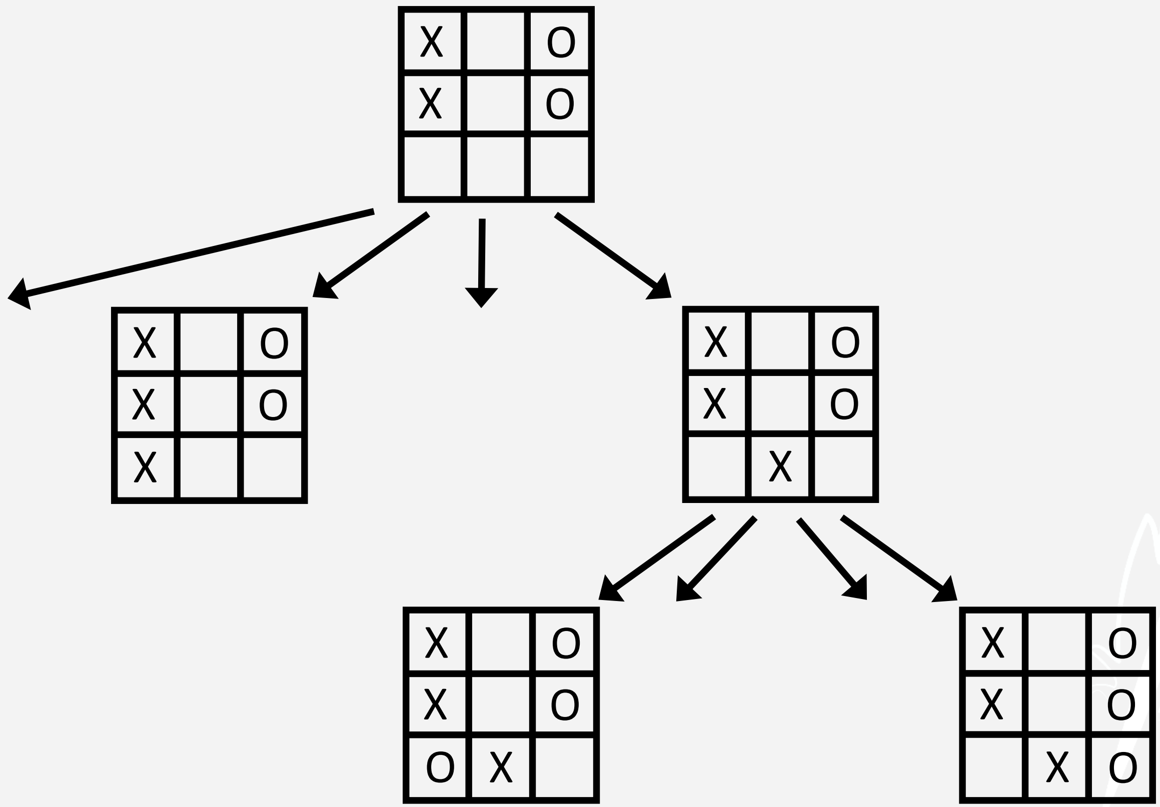
Lessons to Learn

- Dealing with other agents
- Coping with large search spaces
- Making the best of limited resources
- Designing heuristics

Game Theory Games

- Two perfectly rational players
- Each maximizes their own score
- Players take turns
- Moves are deterministic
- Game has a zero sum
- Perfect information is available to both players

Tic-Tac-Toe Game Tree



Utility

A **utility** function measures how desirable a given state is for an agent.

For Tic-Tac-Toe:

- 1 point if you won the game
- -1 point it you lost the game
- 0 points if the game is a tie

Utility in 2PZS Games

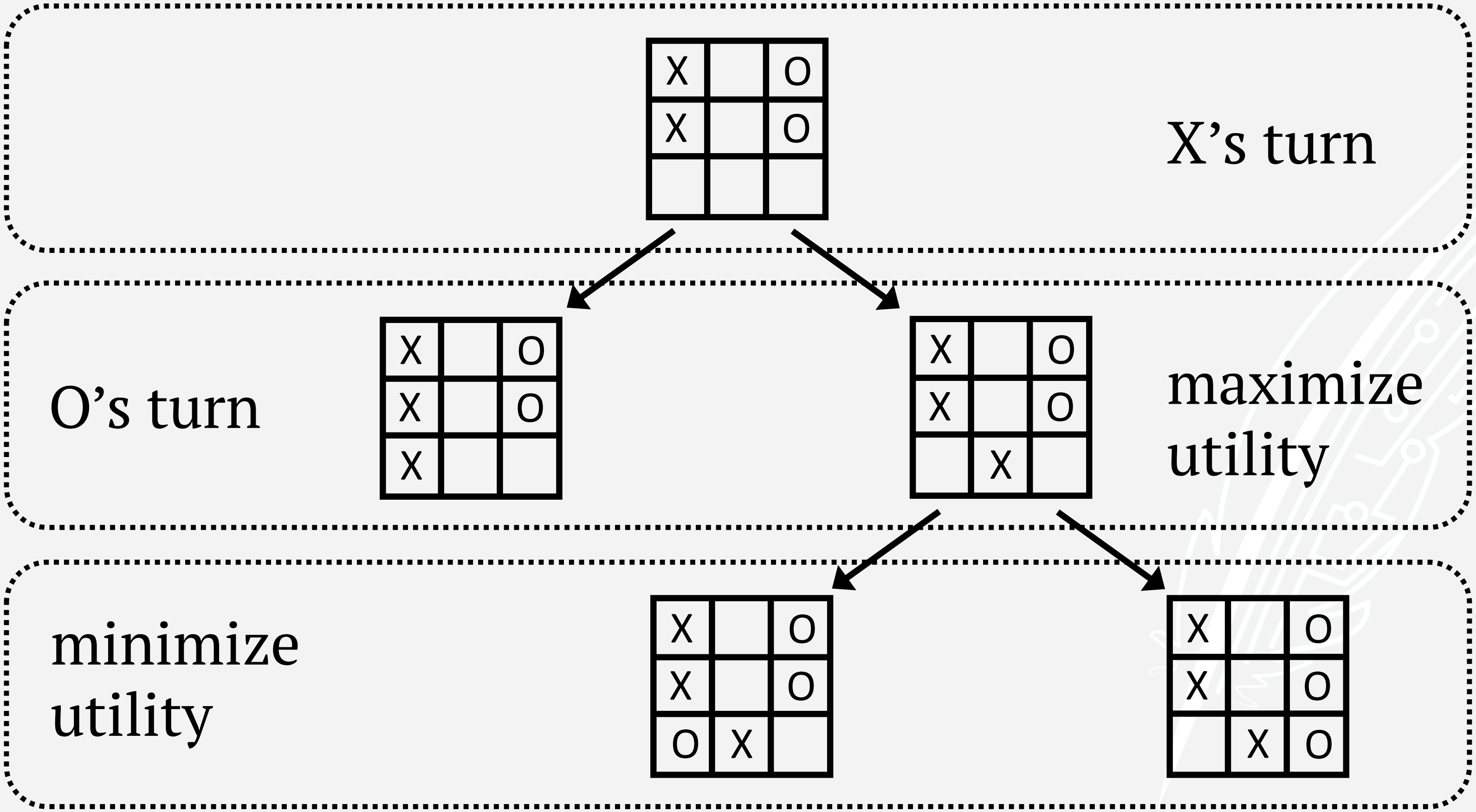
Generally, each agent has its own utility function which it tries to maximize.

In the case of 2-player, zero-sum games, we can use a single utility function for both agents.

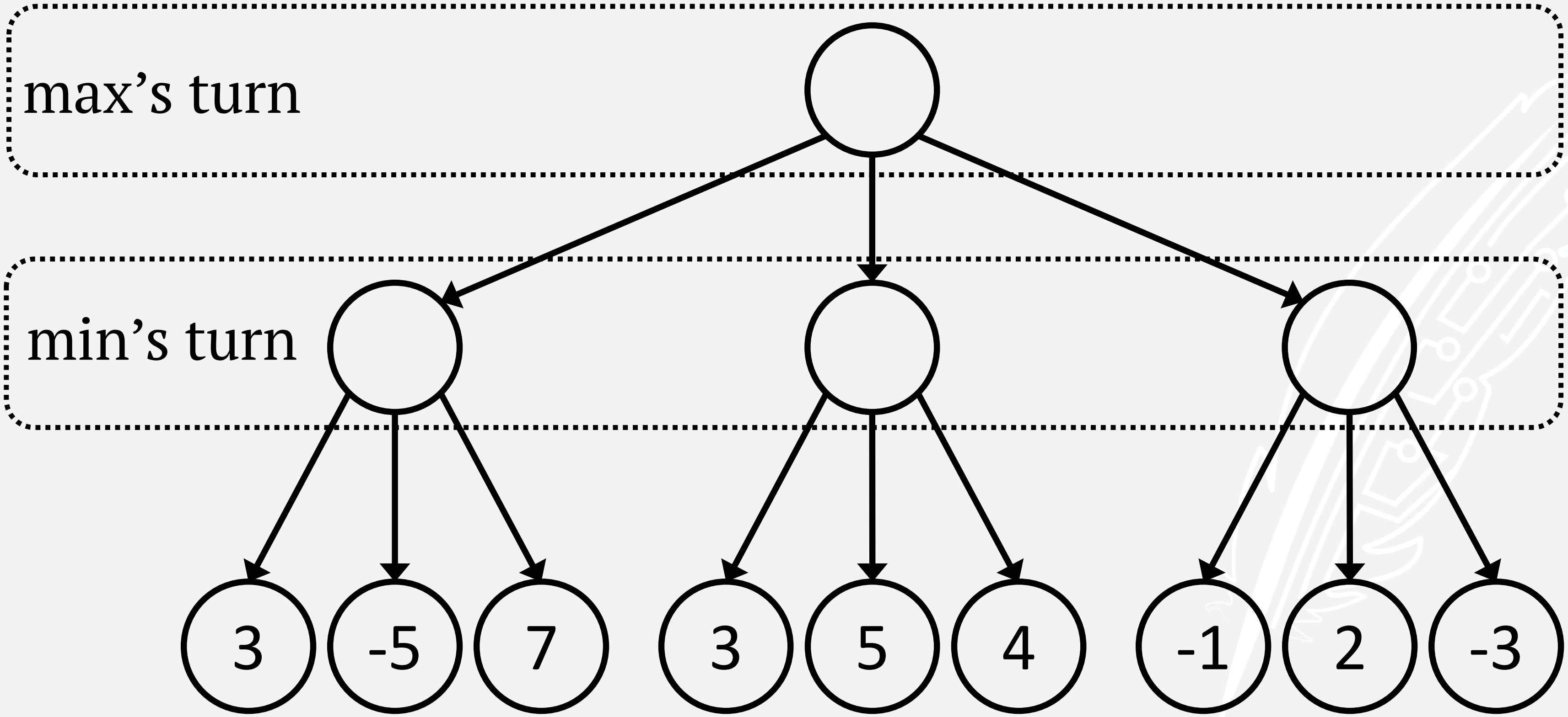
We name the two players:

- "max" tries to maximize the utility value
- "min" tries to minimize the utility value

Tic-Tac-Toe Game Tree

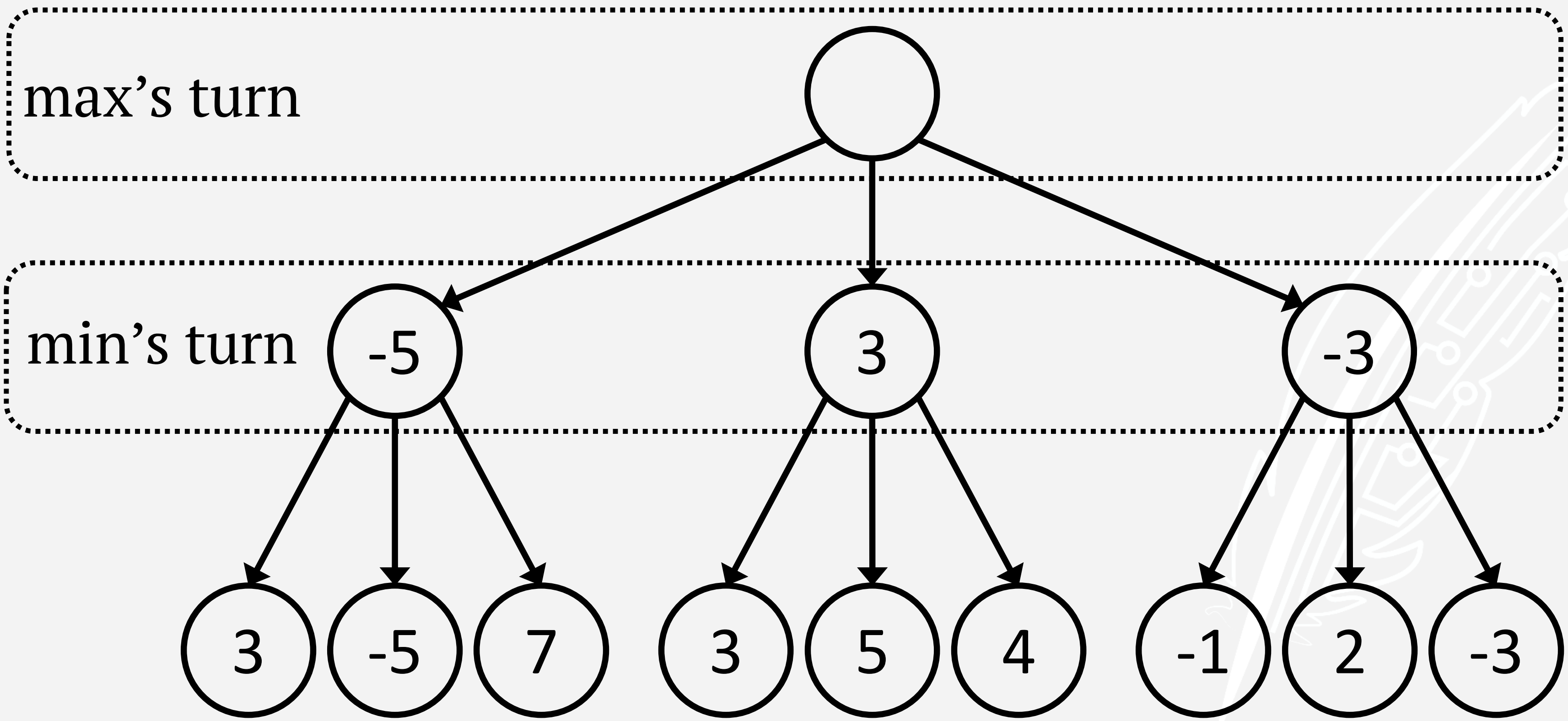


Game Trees in General

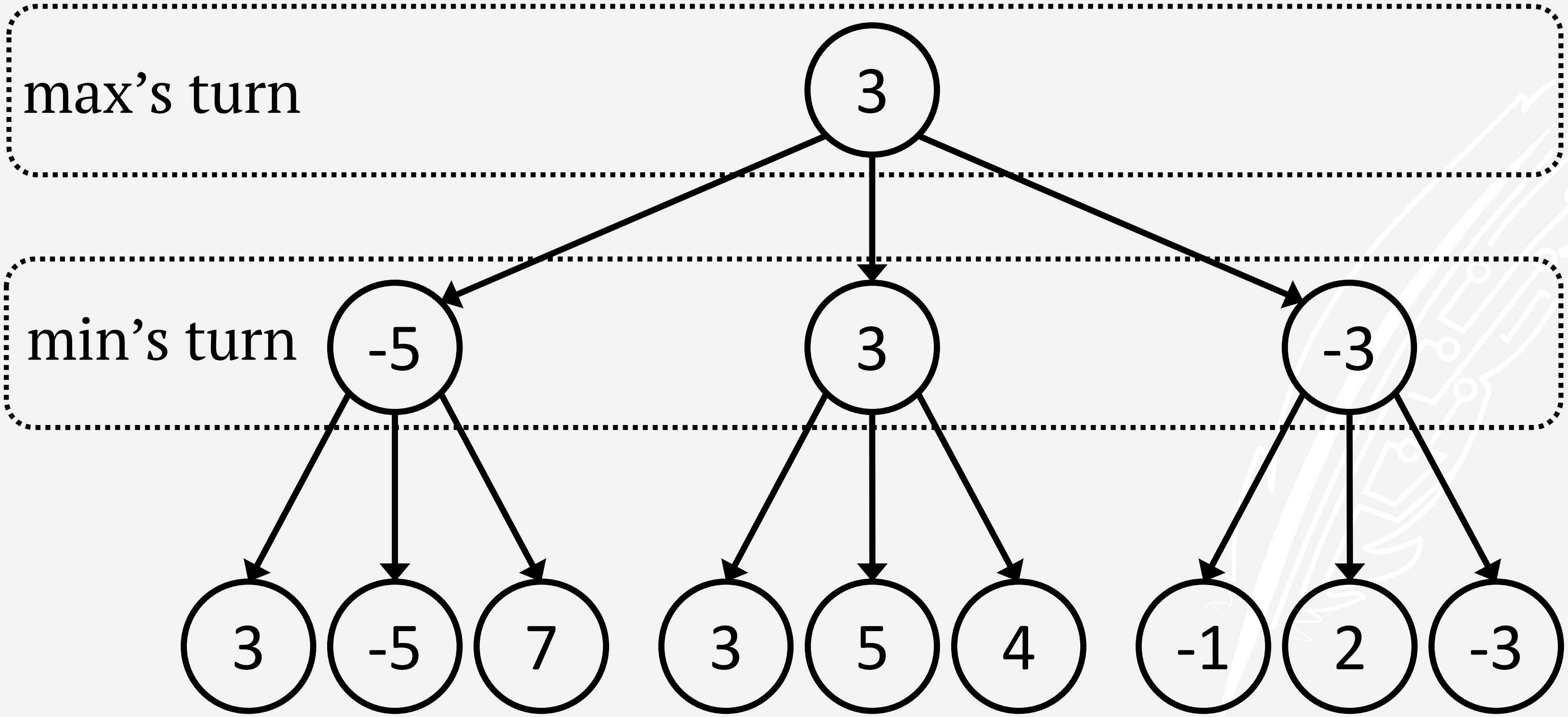


NIL

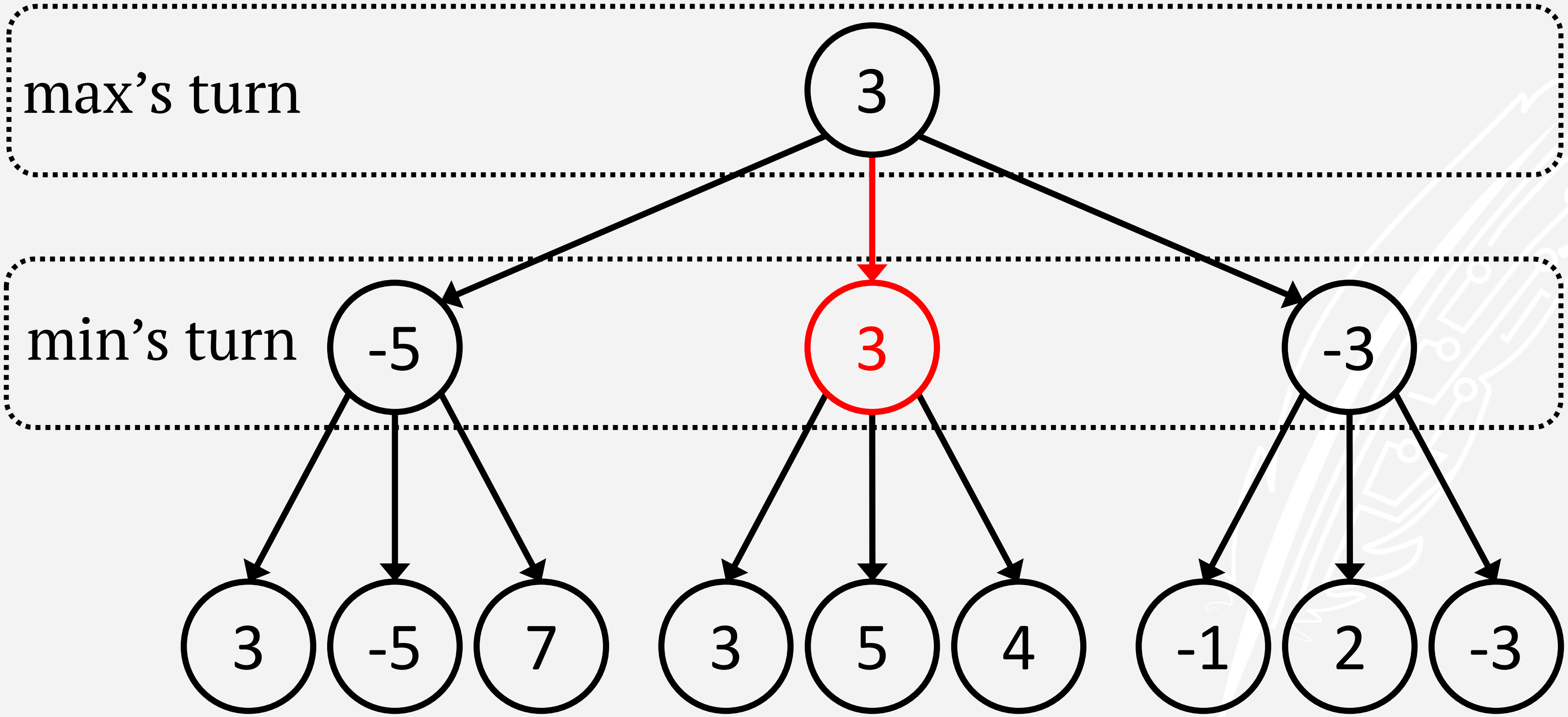
Game Trees in General



Game Trees in General



Max's Best Move



Minimax Search

To find the optimal strategy:

- Expand the tree using depth first search.
- Leaf nodes have a value equal to their utility.
- The value of a node at a max layer has the highest value of any of its children.
- The value of a node at a min layer has the lowest value of any of its children.

Minimax Search

```
function min_max(Node n) {
      best = find_max(n);
      return the move that results in best utility;
}
```

Recursive Depth First Tree Search

```
function dfs(Node n) {
      for every child of n {
            child = next child of n;
            dfs(child);
      }
}
```

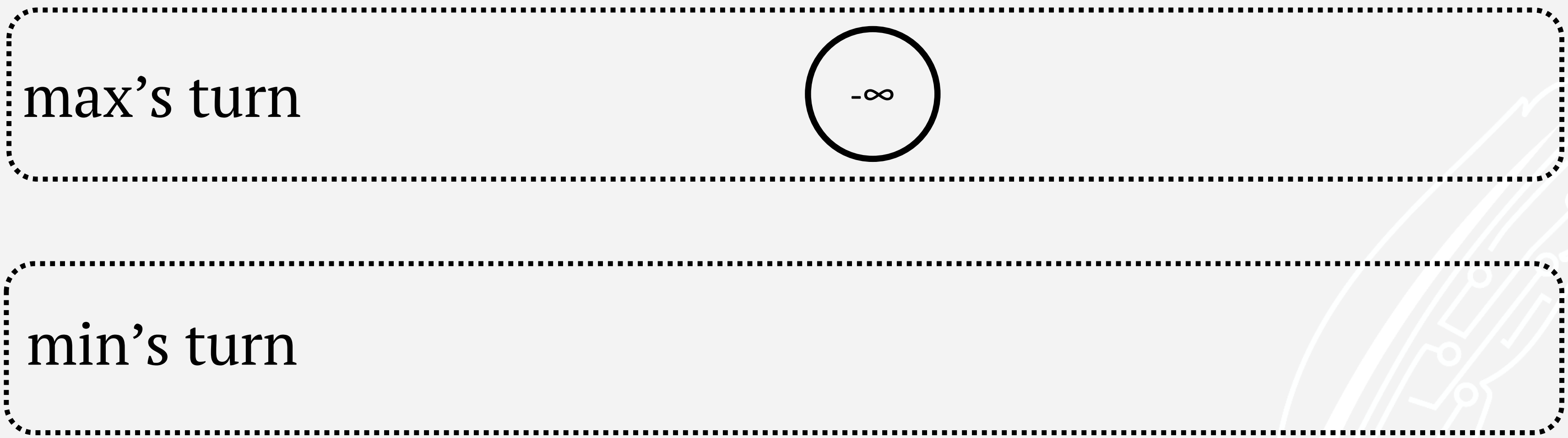
Find Max

```
function find_max(Node n) {
      if n is a leaf node, return utility(n);
      best = -∞;
      for every child of n {
            child = next child of n;
            child_value = find_min(child);
            best = max(best, child_value);
      }
      return best;
}
```

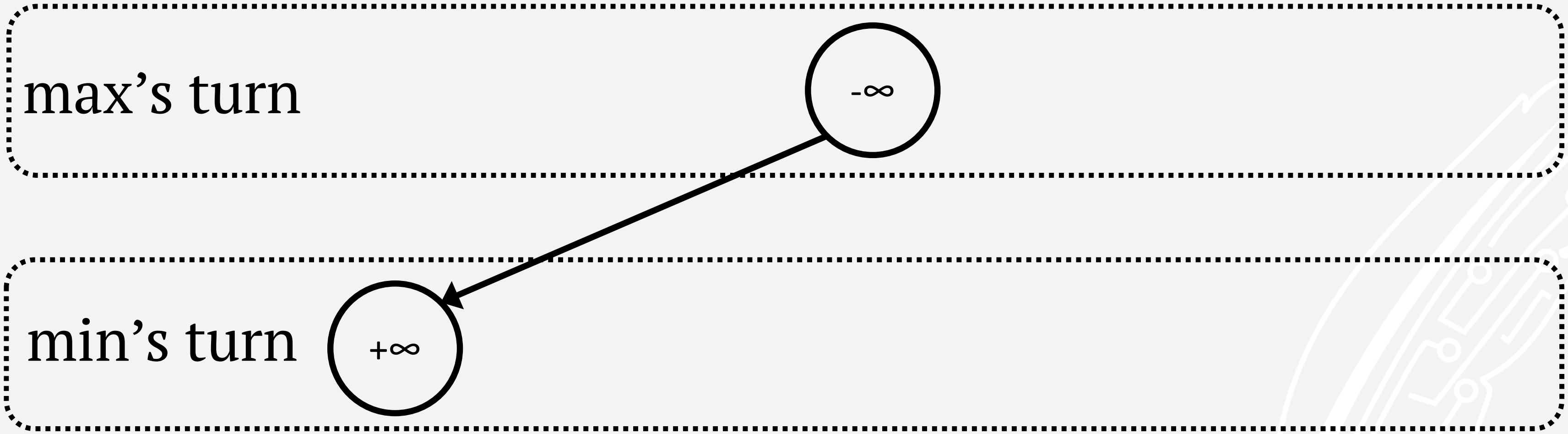
Find Min

```
function find_min(Node n) {
      if n is a leaf node, return utility(n);
      best = +∞;
      for every child of n {
            child = next child of n;
            child_value = find_max(child);
            best = min(best, child_value);
      }
      return best;
}
```

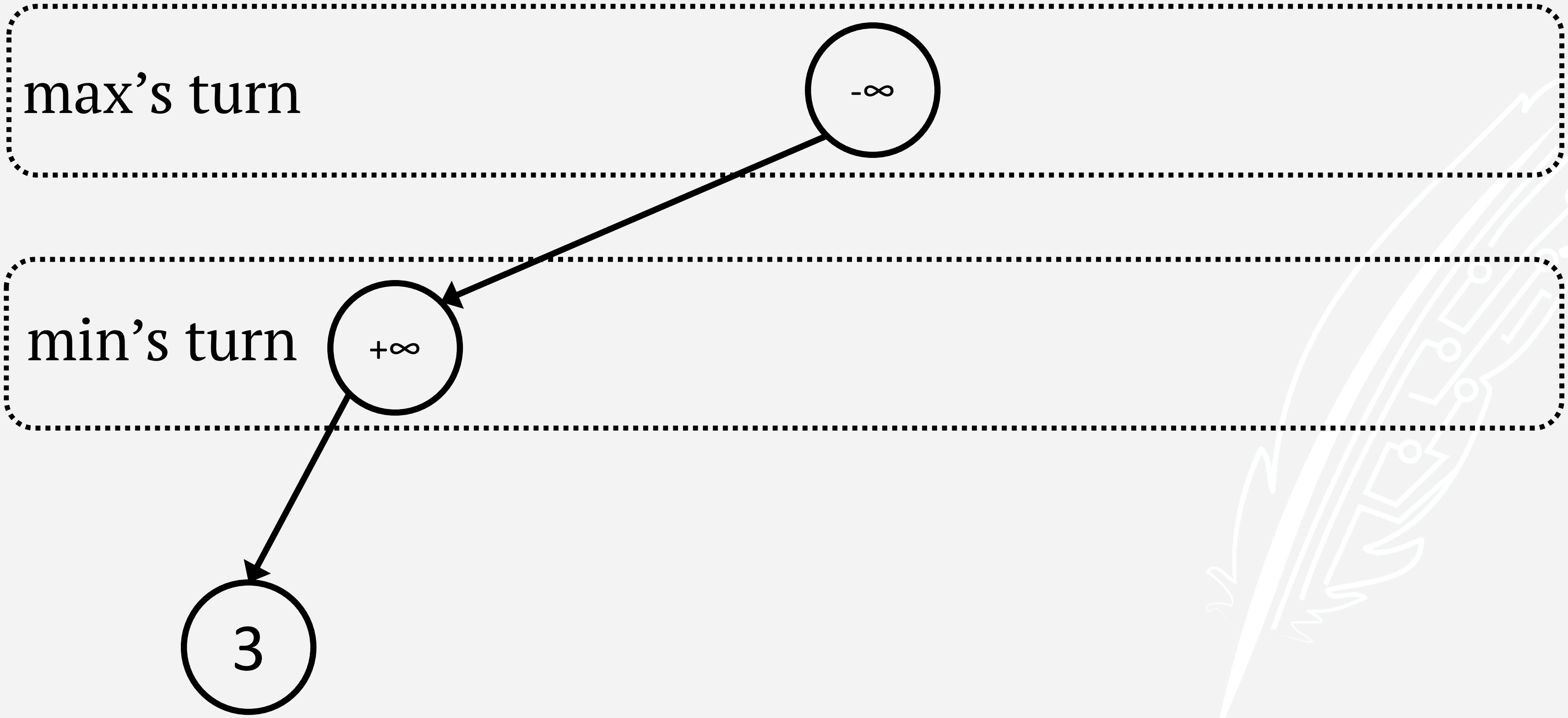
Minimax Example



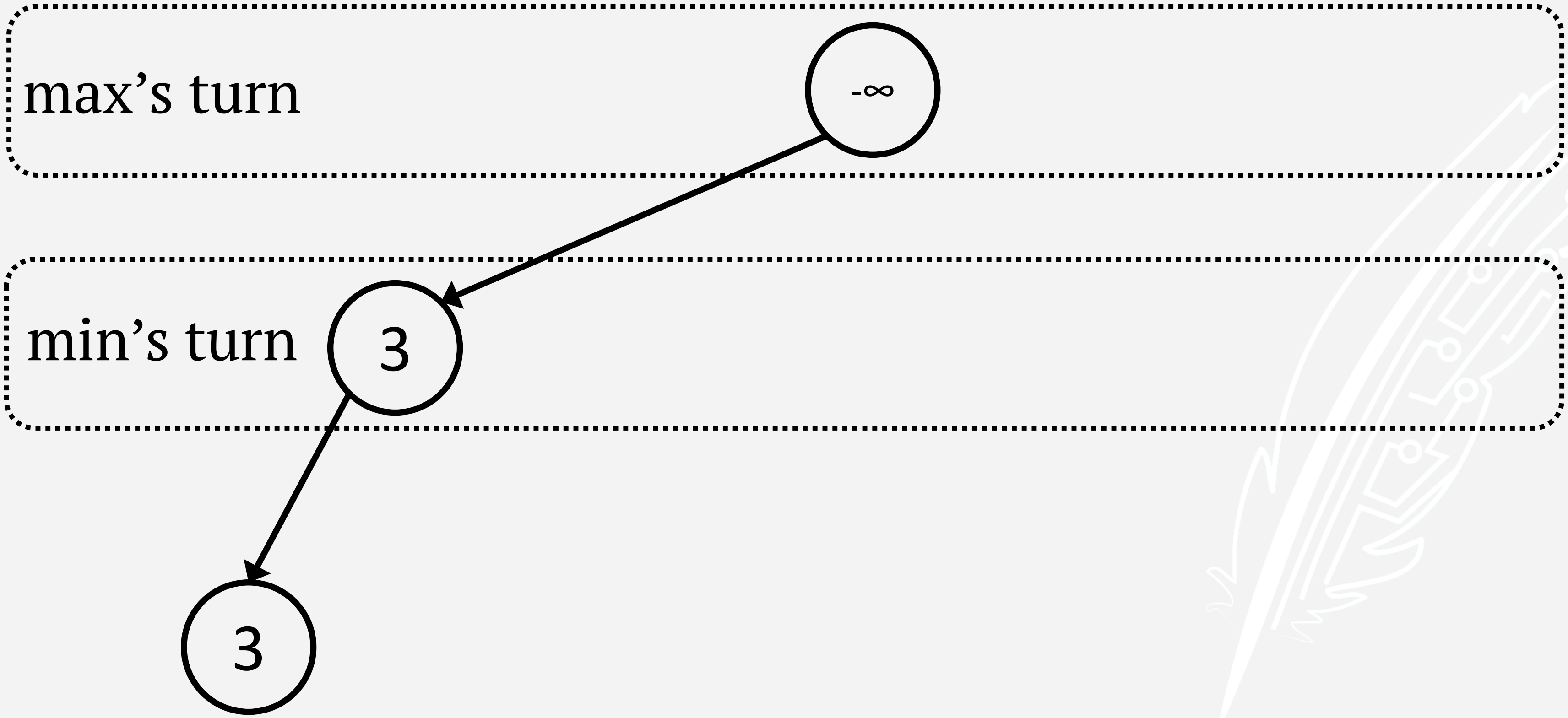
Minimax Example



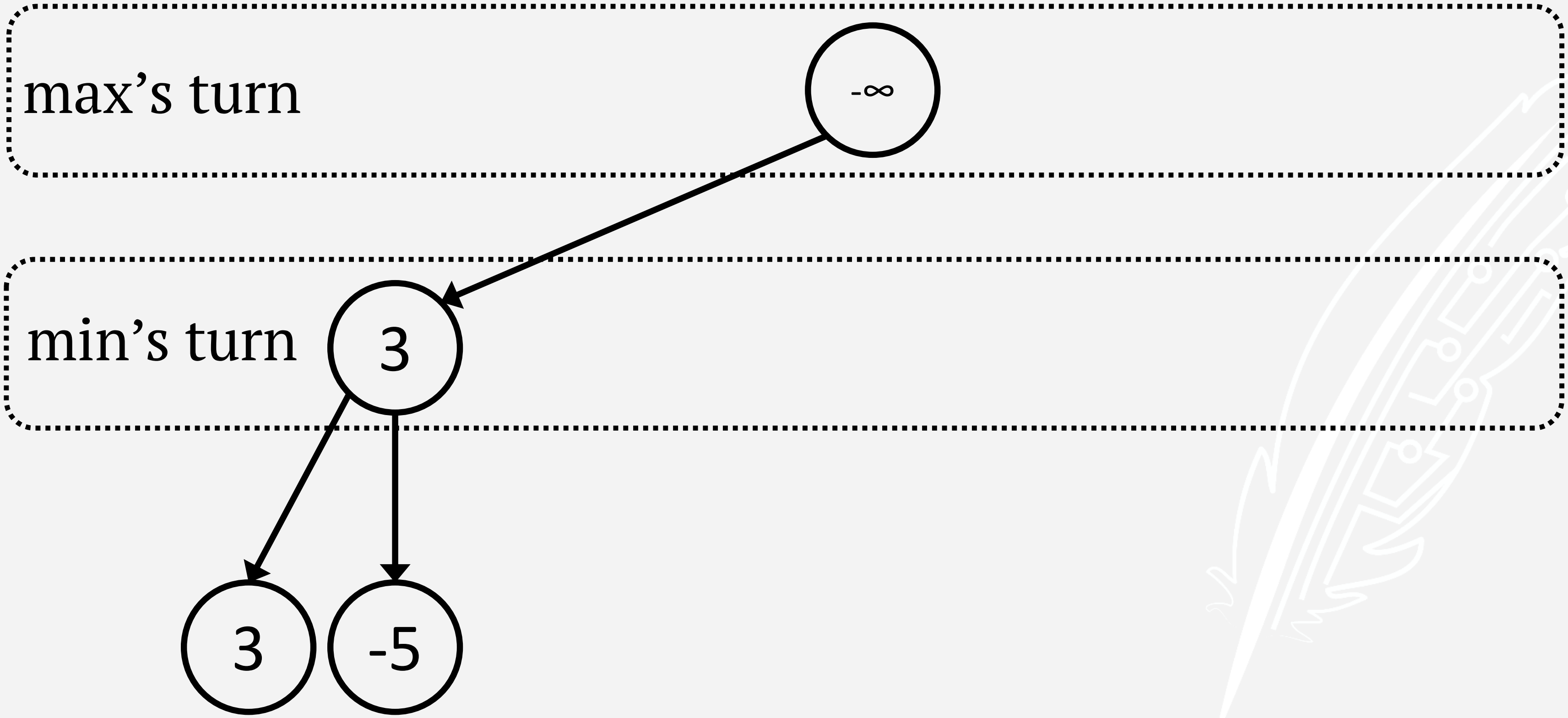
Minimax Example



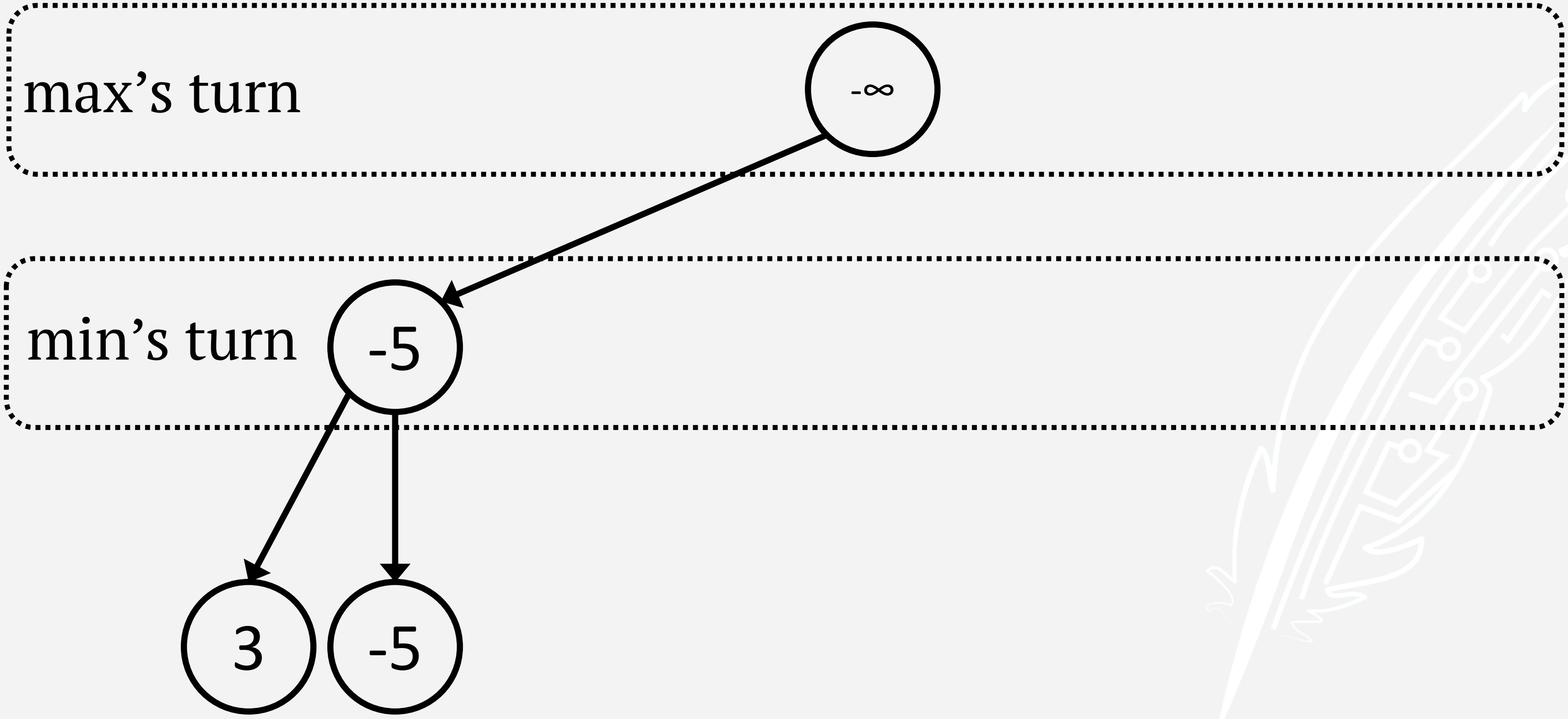
Minimax Example



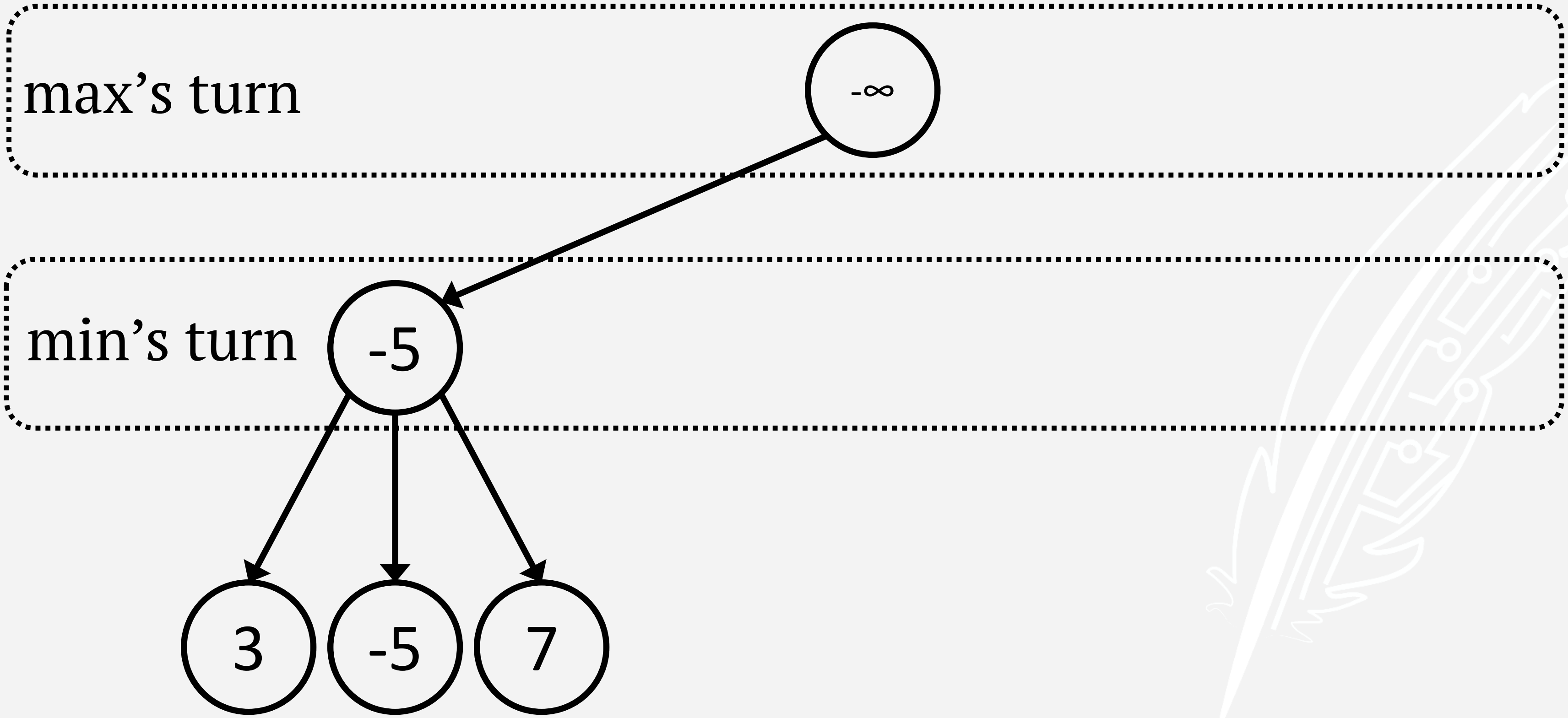
Minimax Example



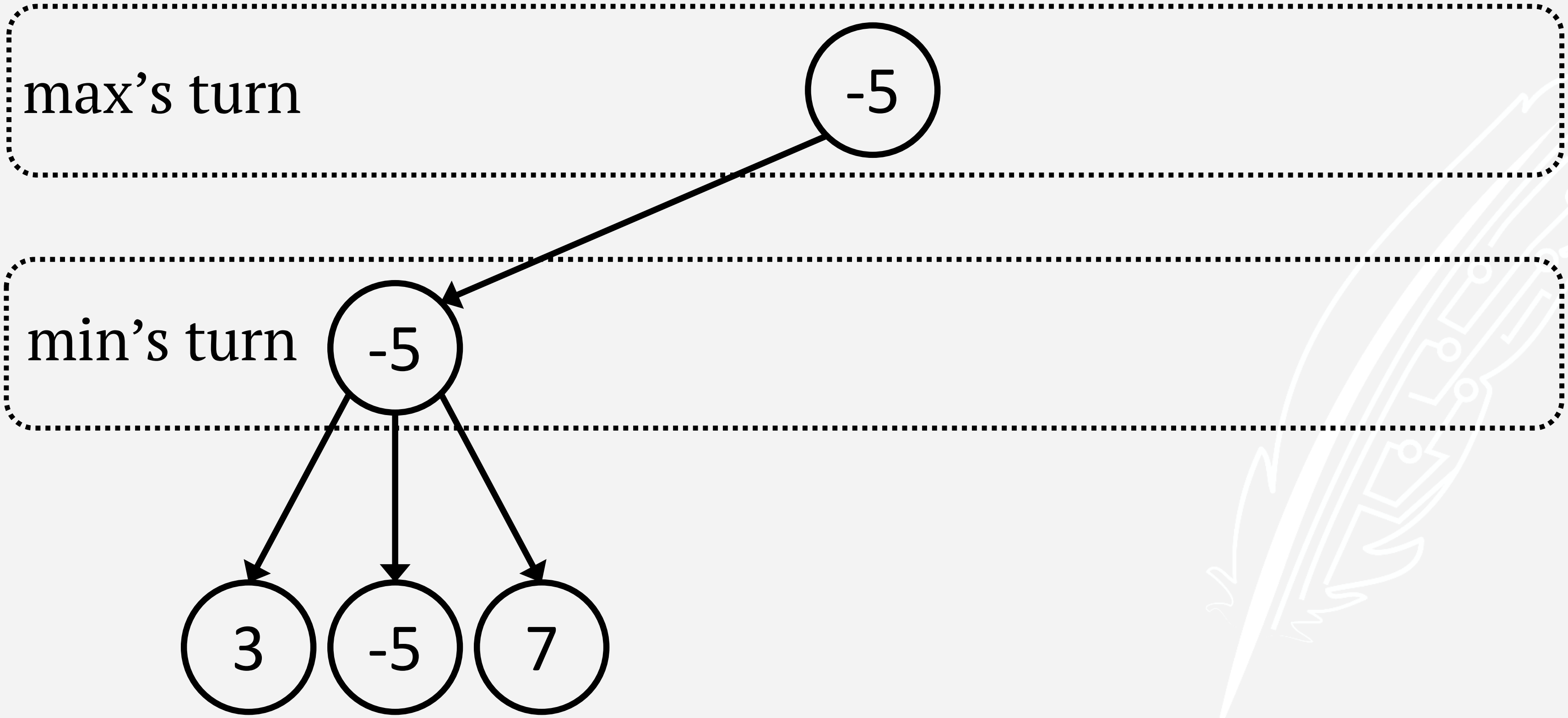
Minimax Example



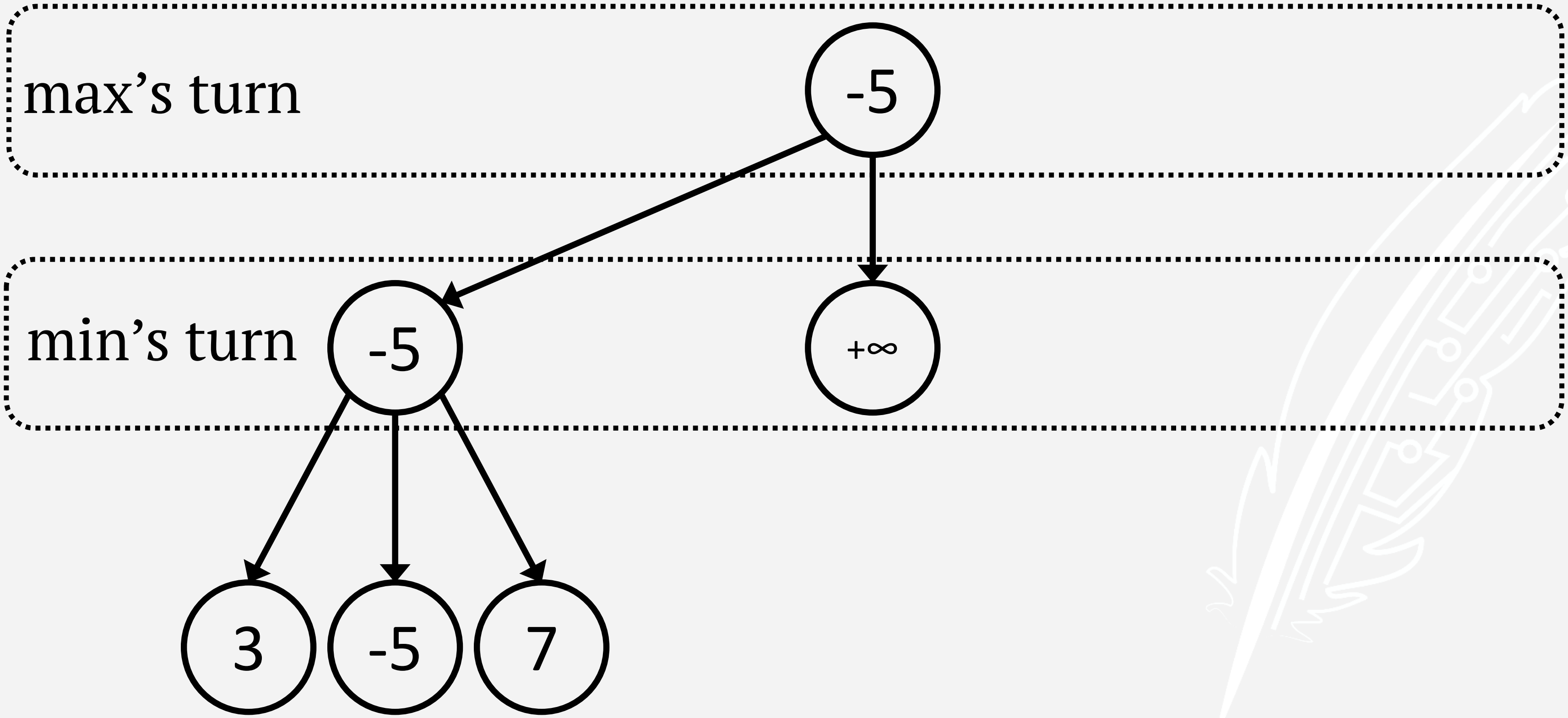
Minimax Example



Minimax Example

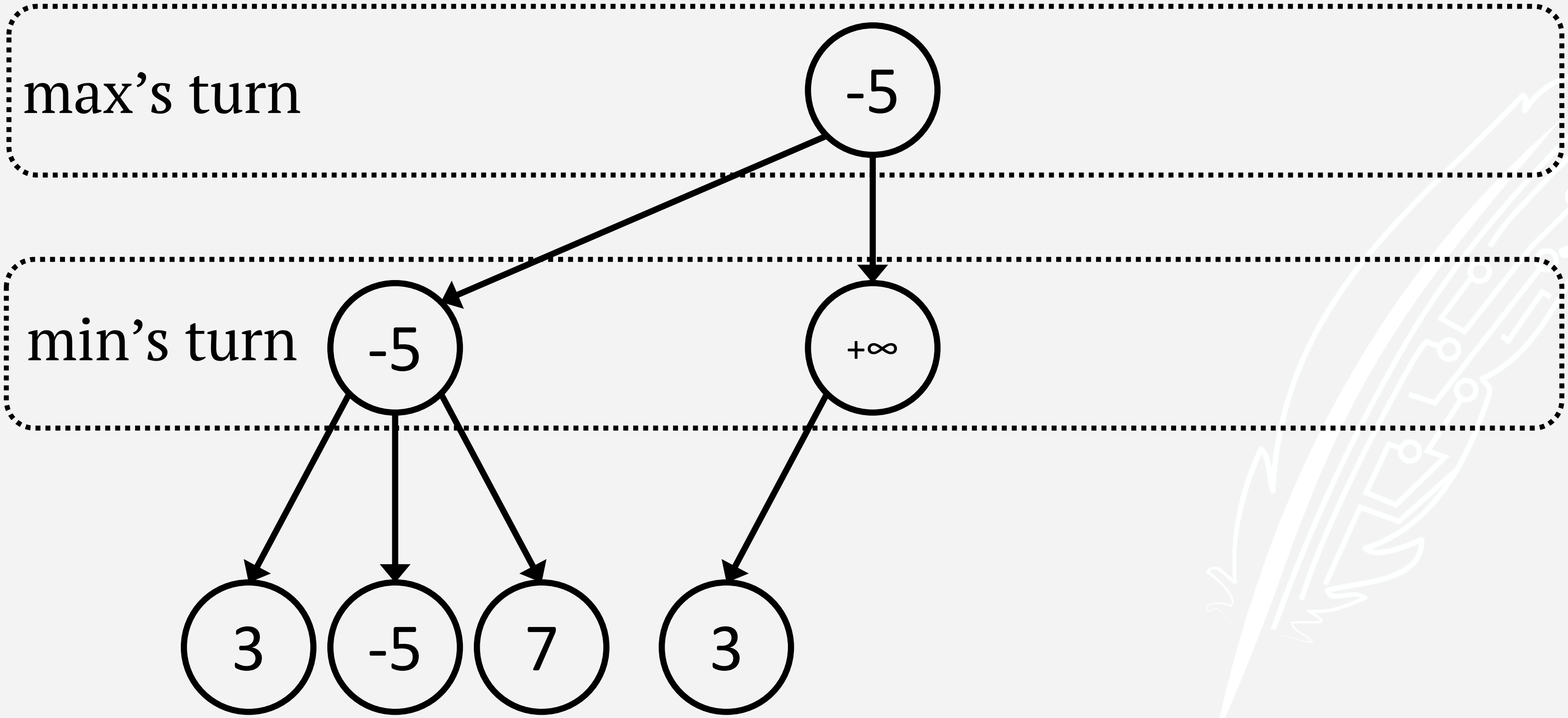


Minimax Example

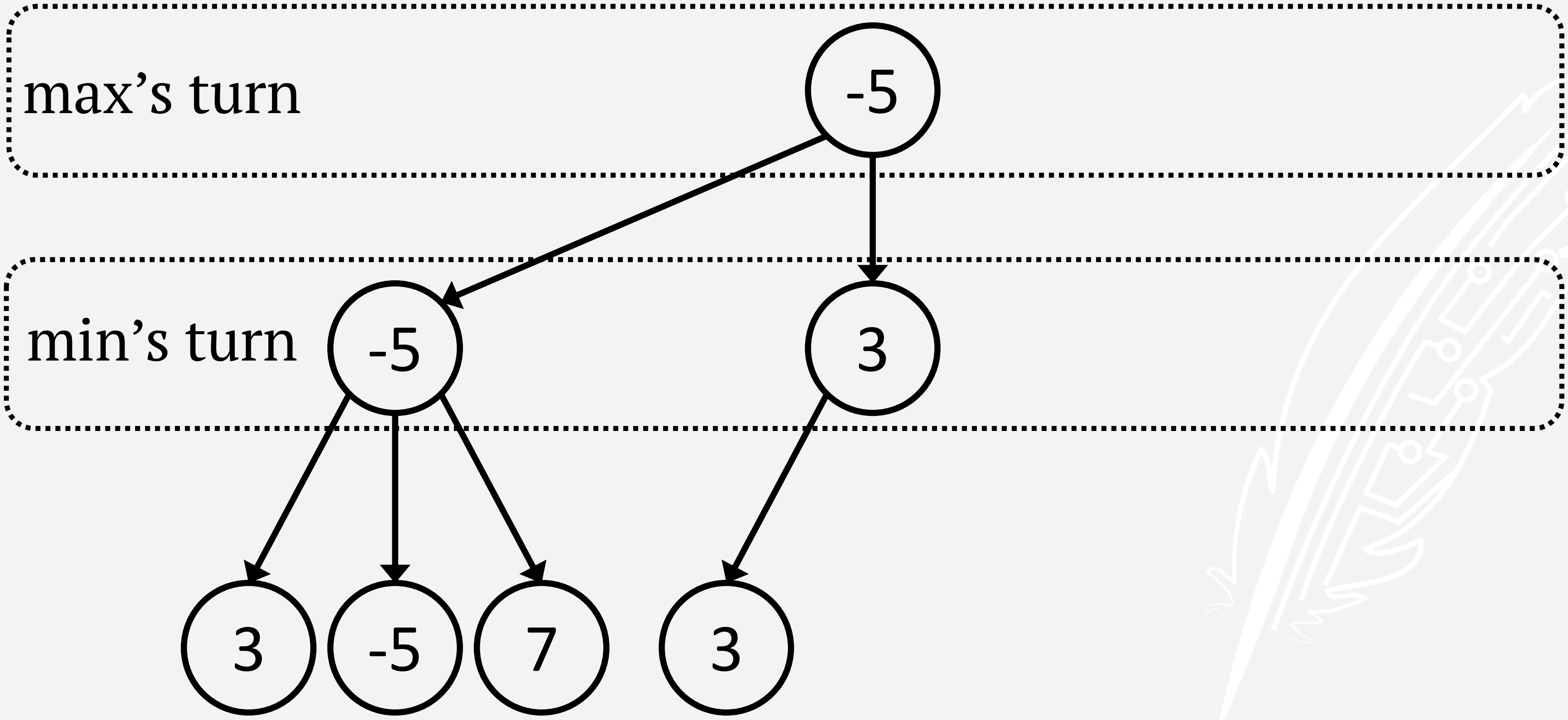


NIL

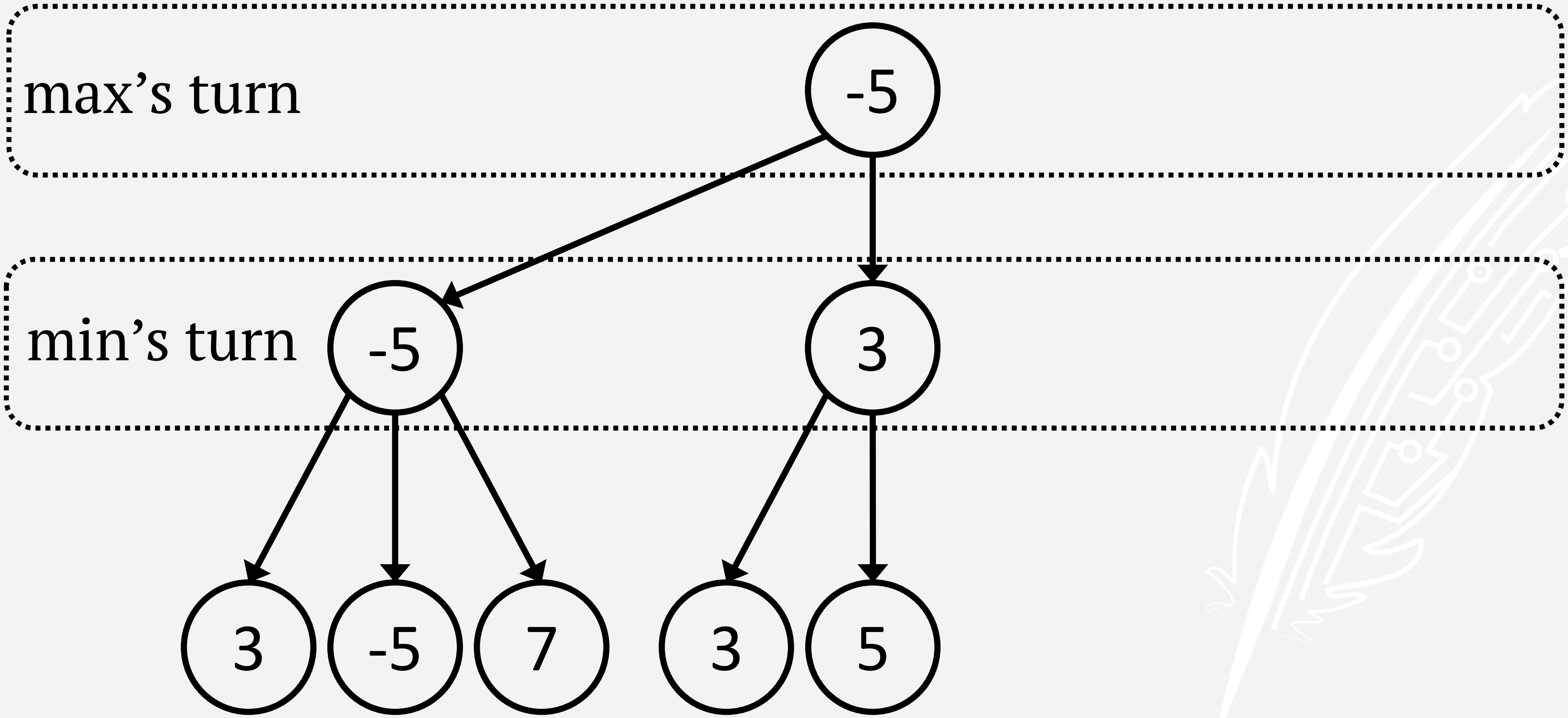
Minimax Example



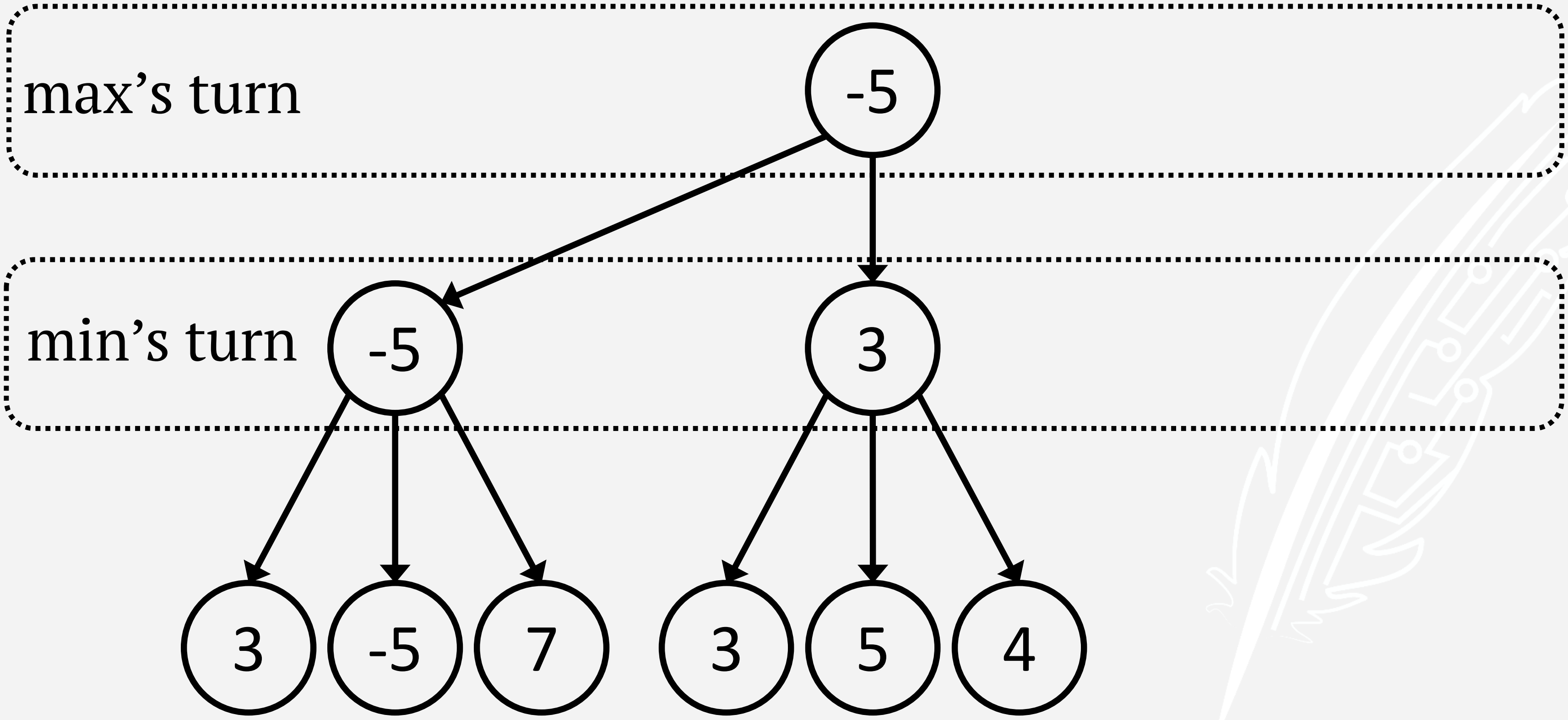
Minimax Example



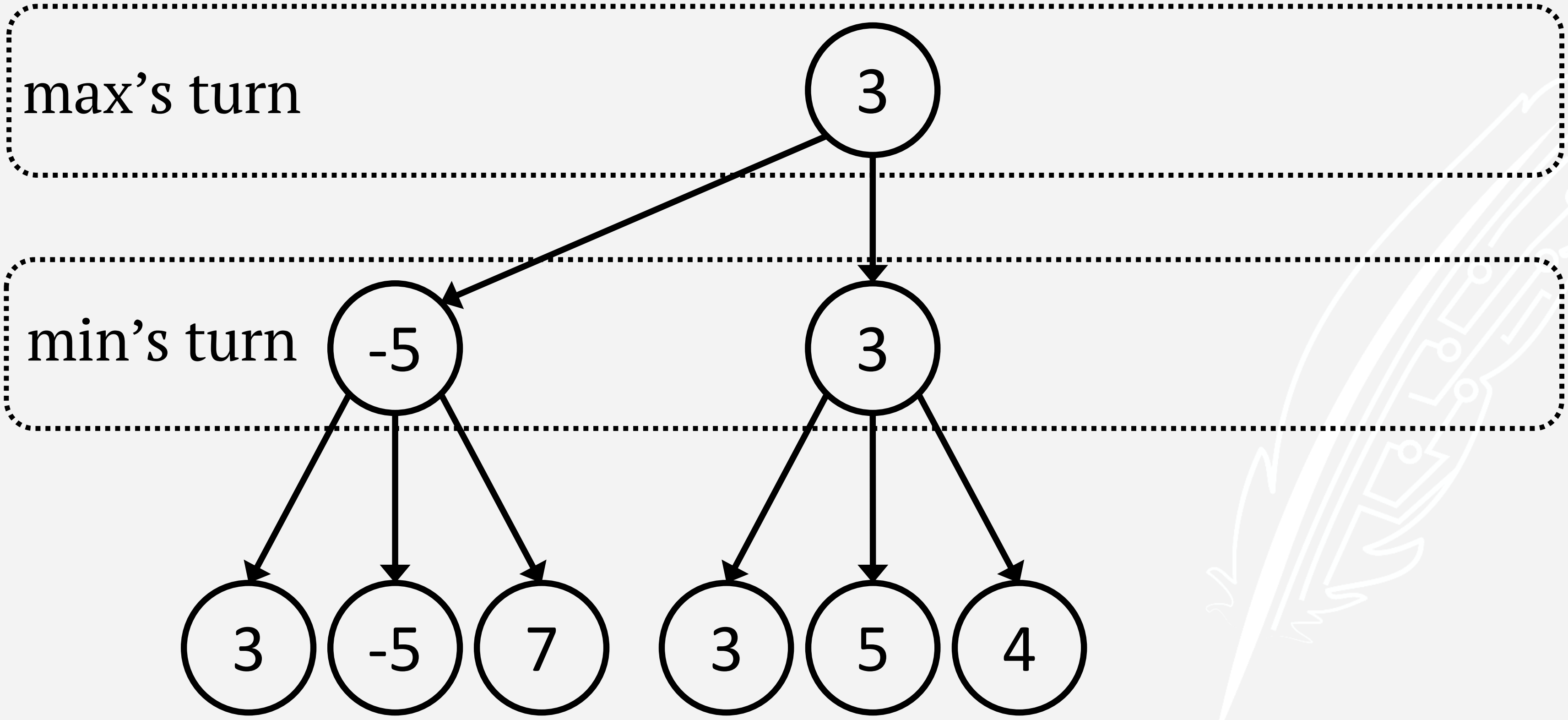
Minimax Example



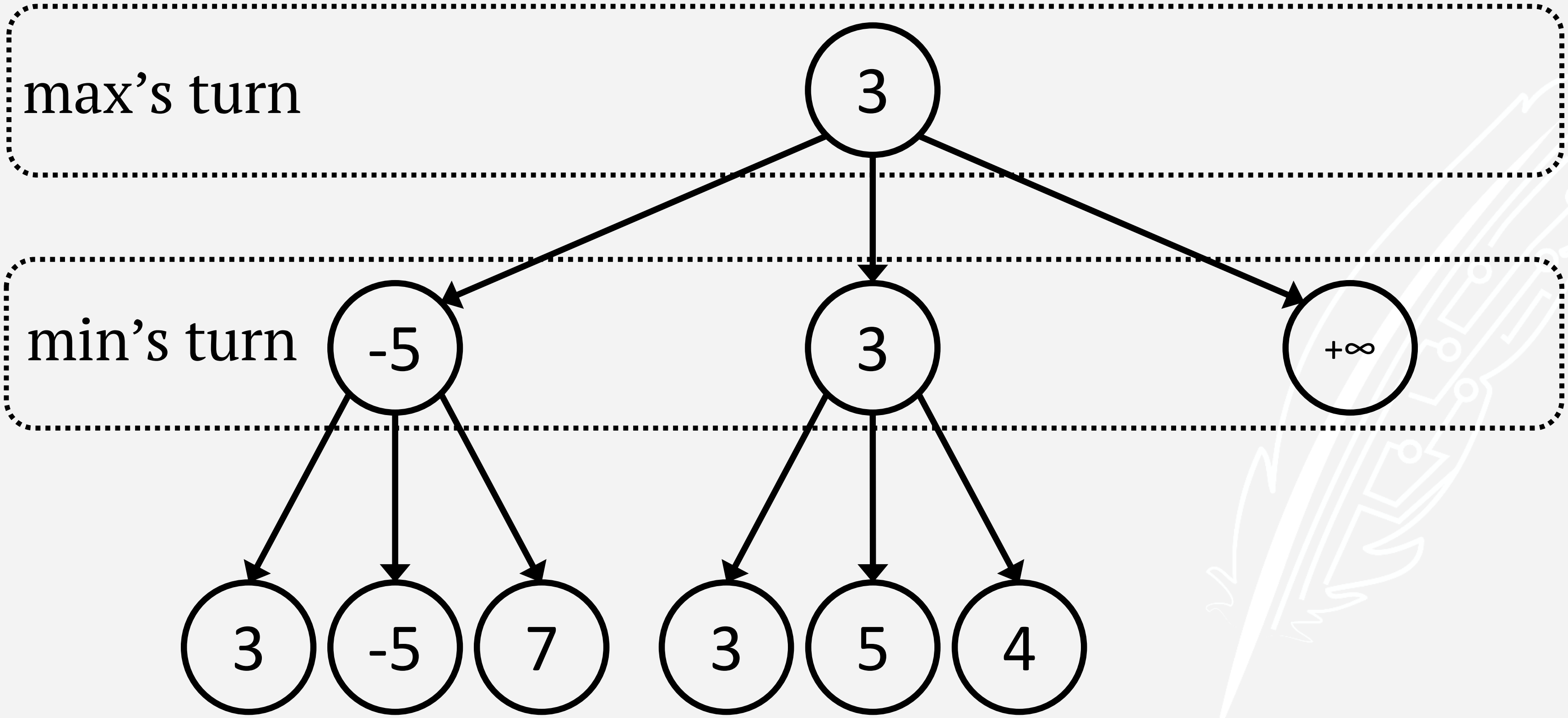
Minimax Example



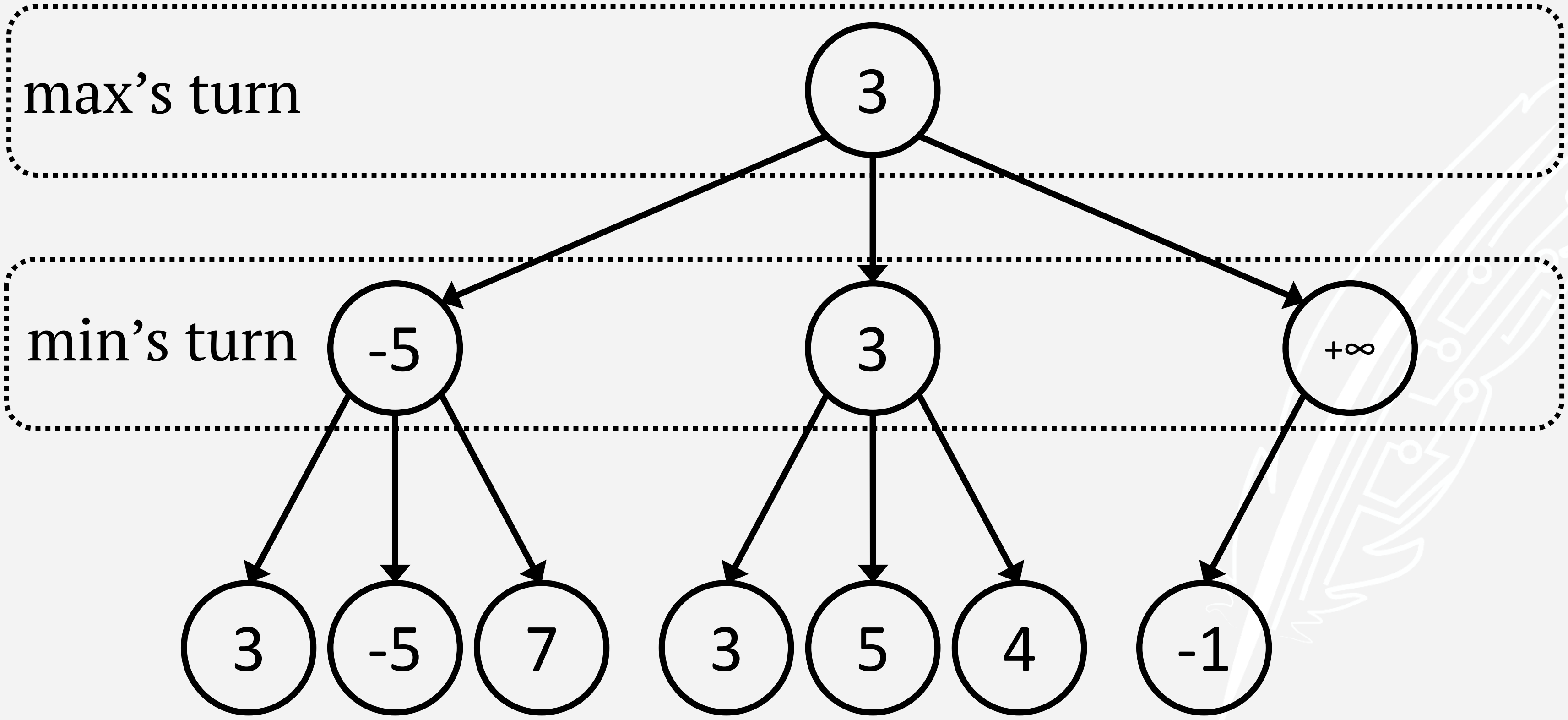
Minimax Example



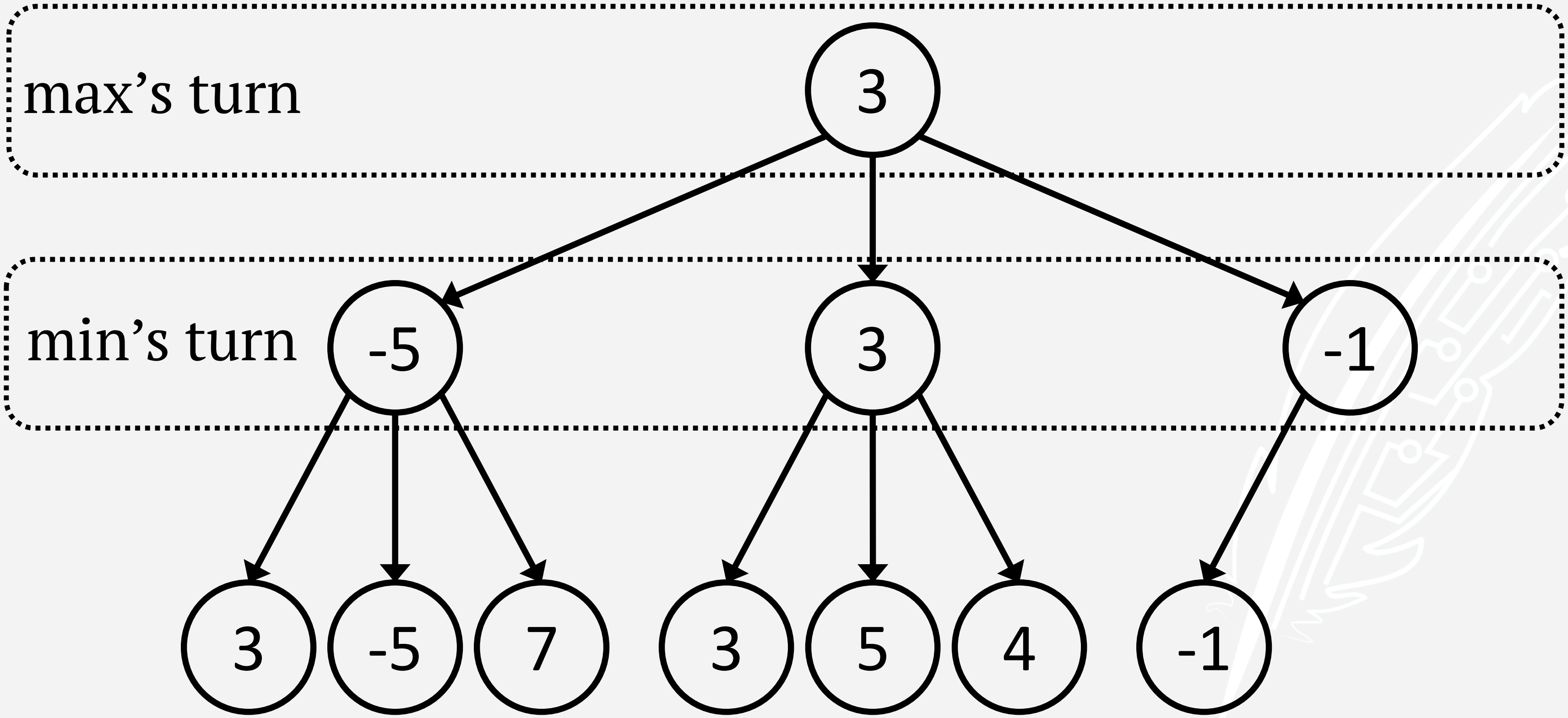
Minimax Example



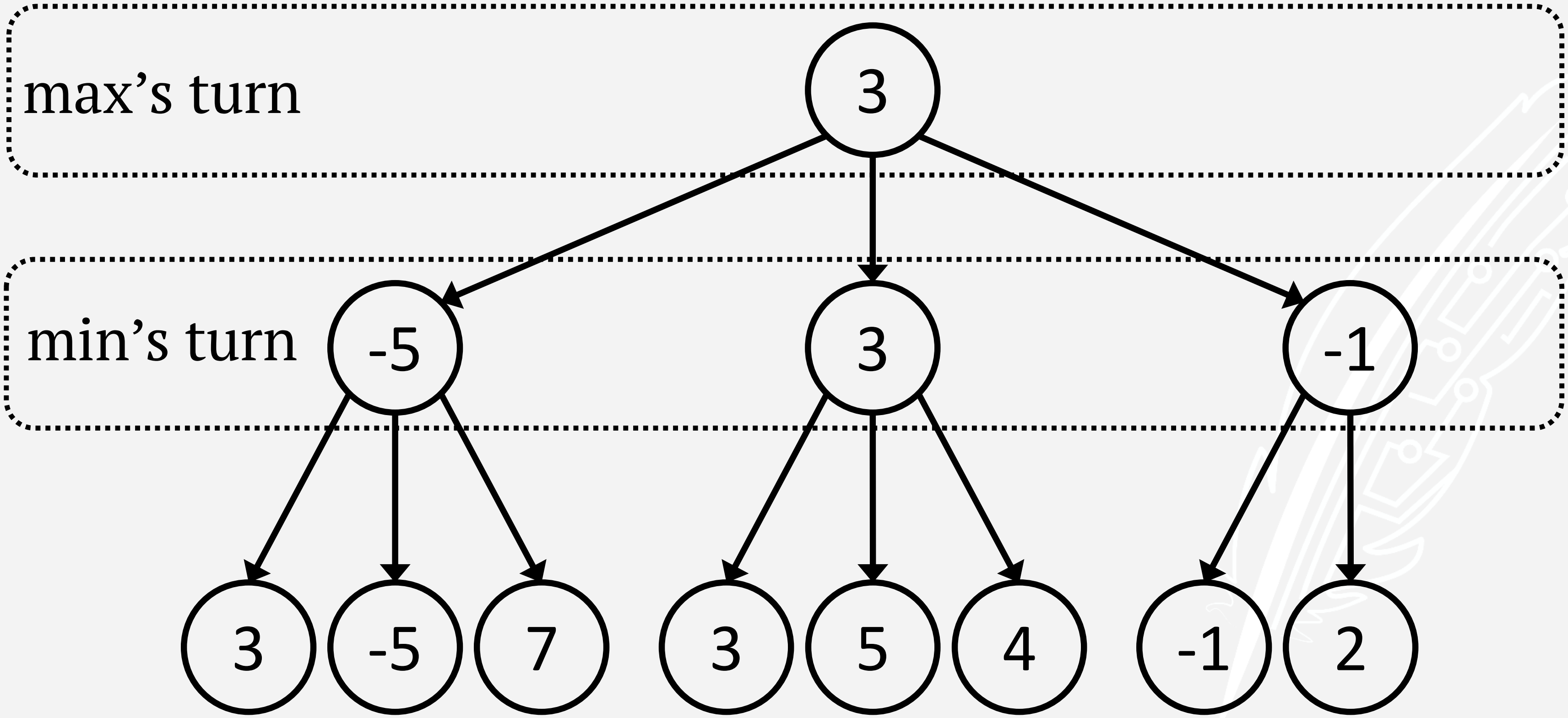
Minimax Example



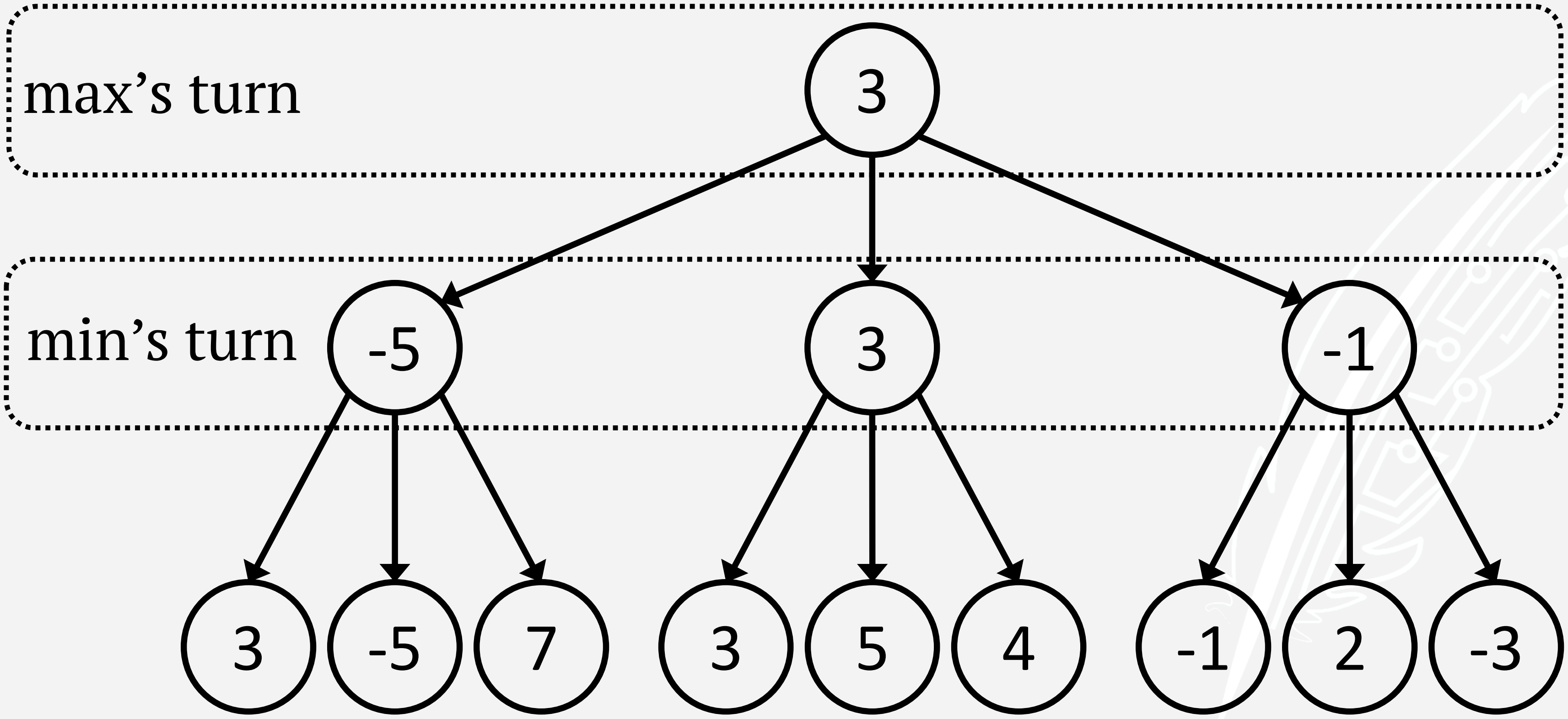
Minimax Example



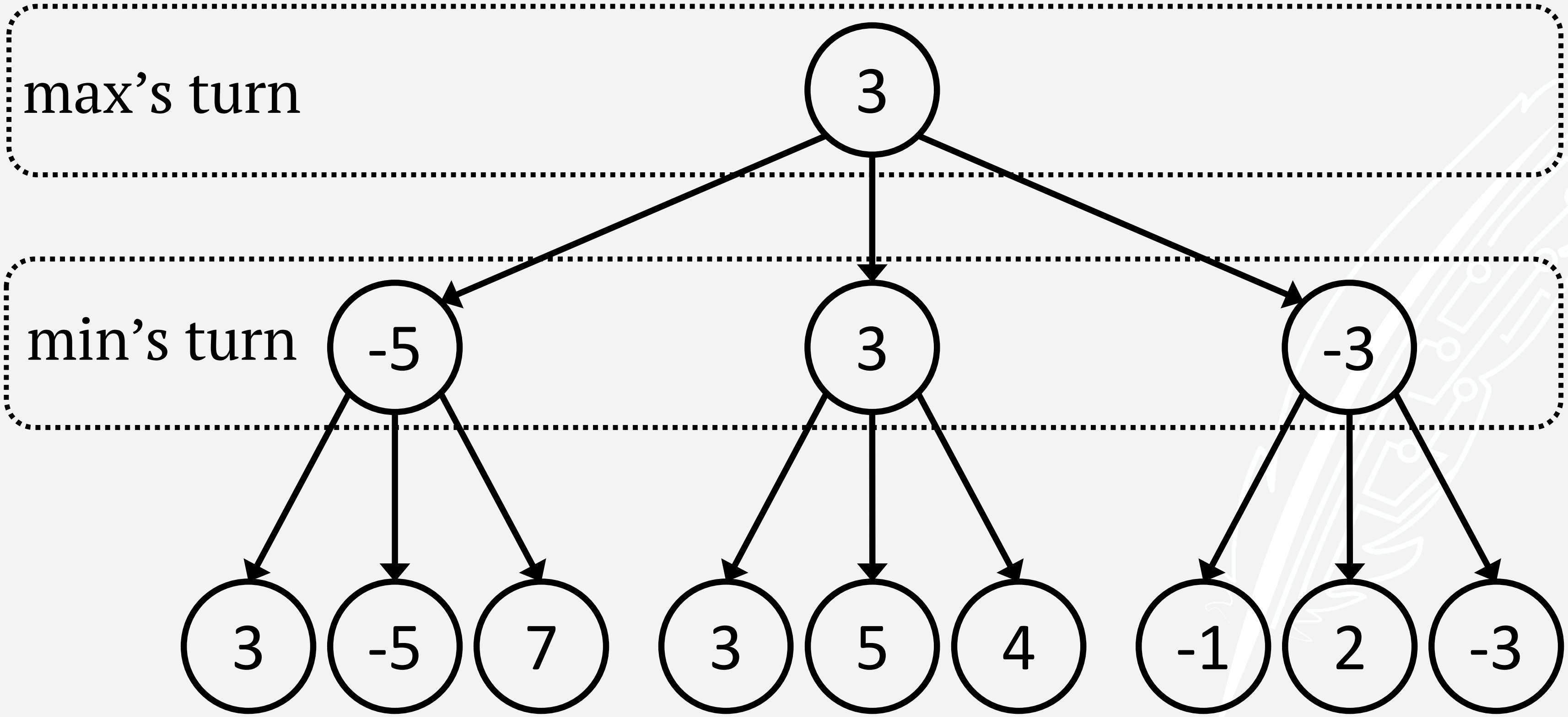
Minimax Example



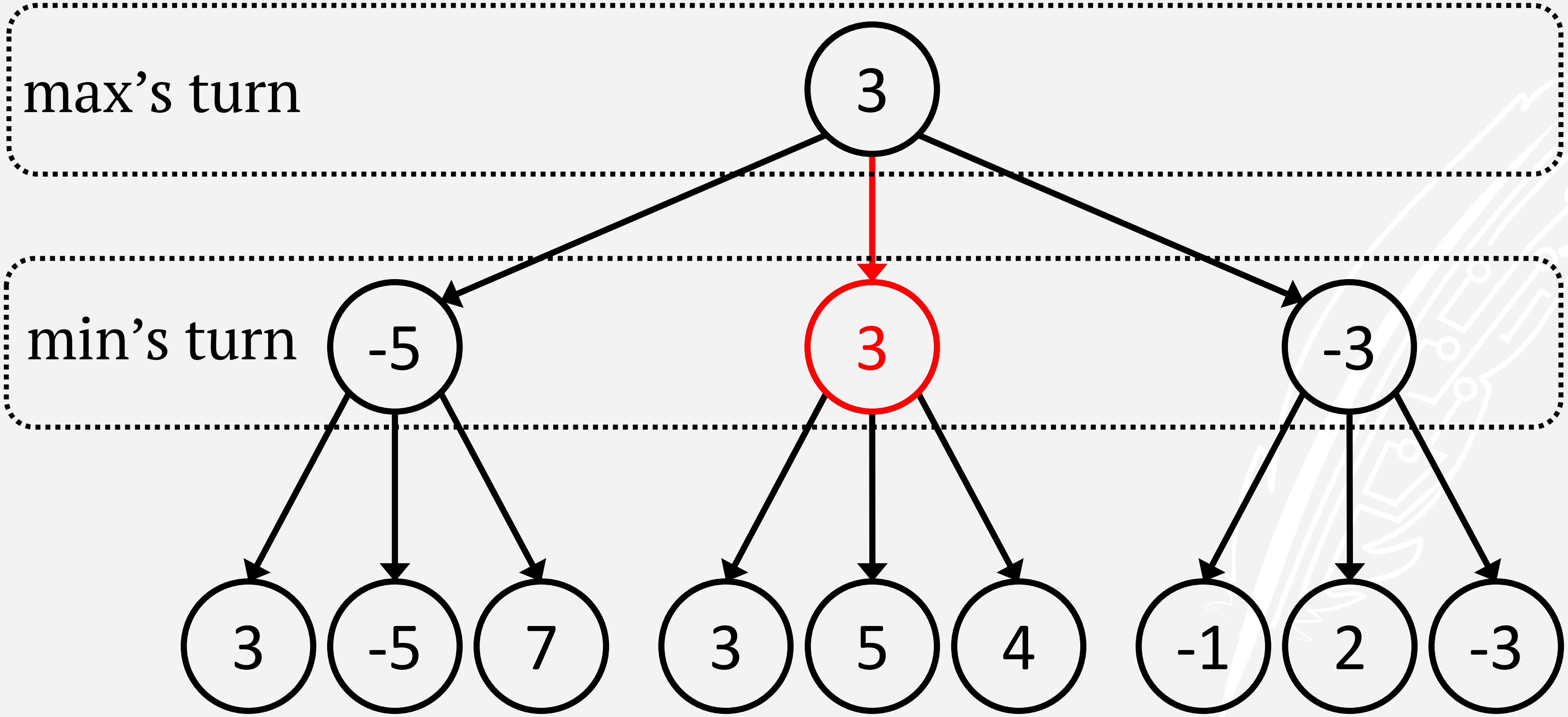
Minimax Example



Minimax Example



Minimax Example



Handling Large Trees

What if we don't have time to expand the entire game tree?

A estimate of the Chess game tree states that it has about 35^{100} nodes.

That's over 2500000000000000000000000000000000000000000000 0000000000000000000000000000000000000000000000 0000000000000000000000000000000000000000000000 00000000000000000 nodes.

Alpha Beta Pruning

- We don't need to know the utility of each branch of the game tree, only the good moves.
- If we find a branch which is worse than the best outcome we have found so far, we don't need to know how much worse. Just ignore it.
- Don't bother considering a move if we know it will be worse than one we've already found.

Alpha Beta Pruning

- Keep track of two numbers during search:
- α, the highest utility found so far
- β, the lowest utility found so far

At max levels, don't bother with values $\geq \beta$.

At min levels, don't bother with values $\leq \alpha$.

Alpha Beta Pruning

- Keep track of two numbers during search:
- α, the highest utility found so far
- β, the lowest utility found so far

At max levels, don't bother with values $\geq \beta$.

(because min won't let that happen)

At min levels, don't bother with values $\leq \alpha$.

(because max won't let that happen)

Minimax Search + ABP

```
function min_max_ab(Node n) {
      best = find_max_ab(n, -∞, +∞);
      return the move that creates best utility;
}
```

Find Max + ABP

```
function find_max_ab(Node n, α, β) {
        if n is a leaf node, return utility(n);
        best = -∞;
        for every child of n {
                child = next child of n;
                child_value = find_min_ab(child, α, β);
                best = max(best, child_value);
                if best ≥ β, return best;
                α = max(α, best);
        }
        return best;
}
```

Find Min + ABP

```
function find_min_ab(Node n, α, β) {
        if n is a leaf node, return utility(n);
        best = +∞;
        for every child of n {
                child = next child of n;
                child_value = find_max_ab(child, α, β);
                best = min(best, child_value);
                if best ≤ α, return best;
                β = min(β, best);
        }
        return best;
}
```

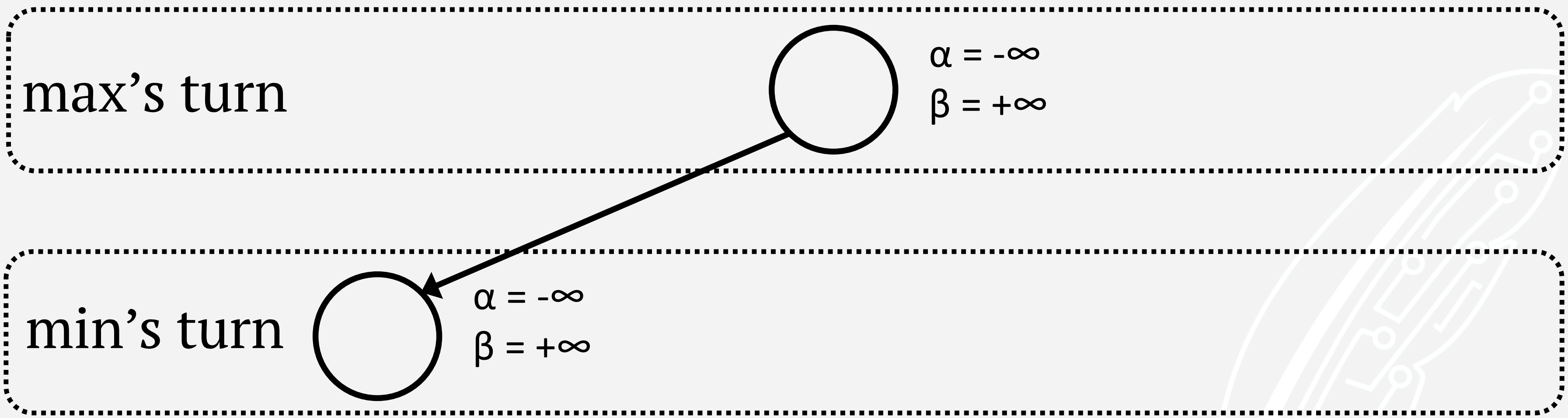
Alpha Prune Example

max's turn

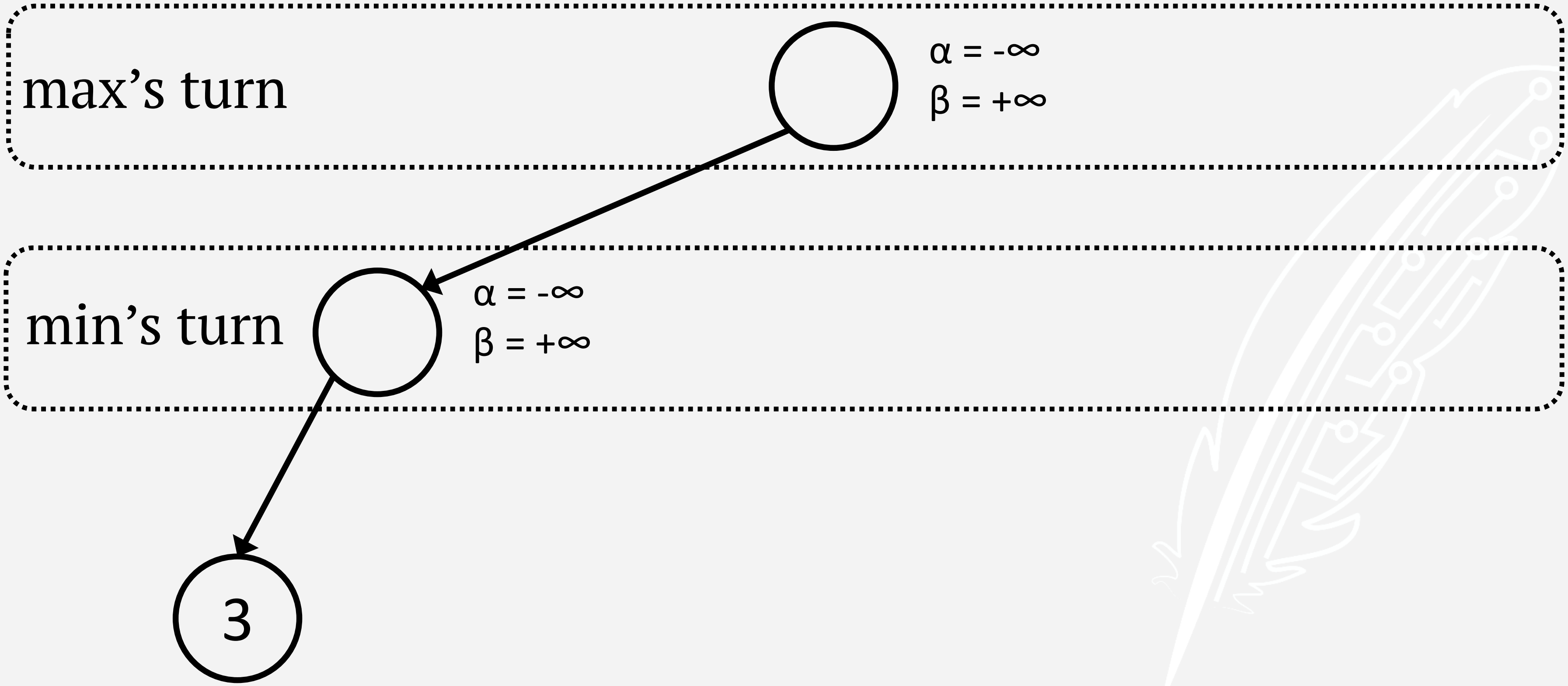
$\alpha = -\infty$
$\beta = +\infty$

min's turn

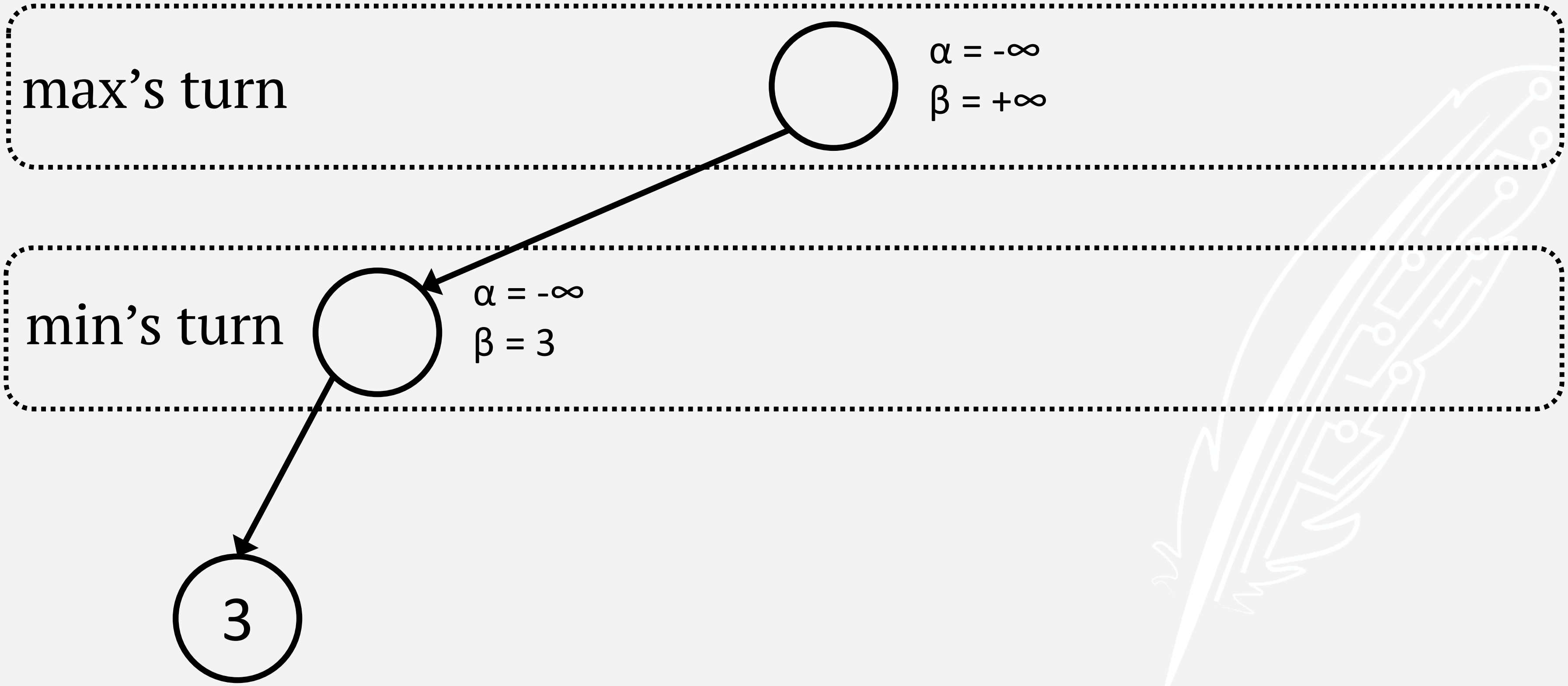
Alpha Prune Example



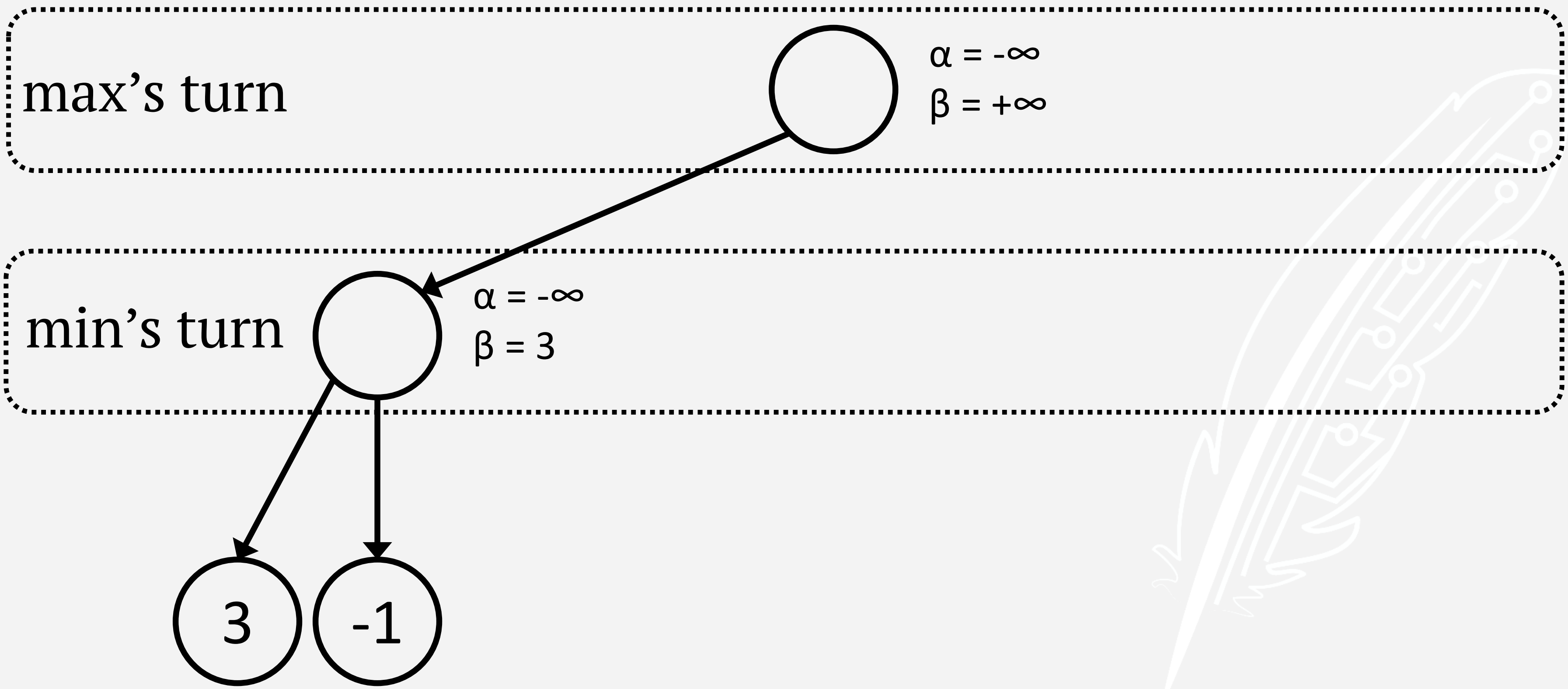
Alpha Prune Example



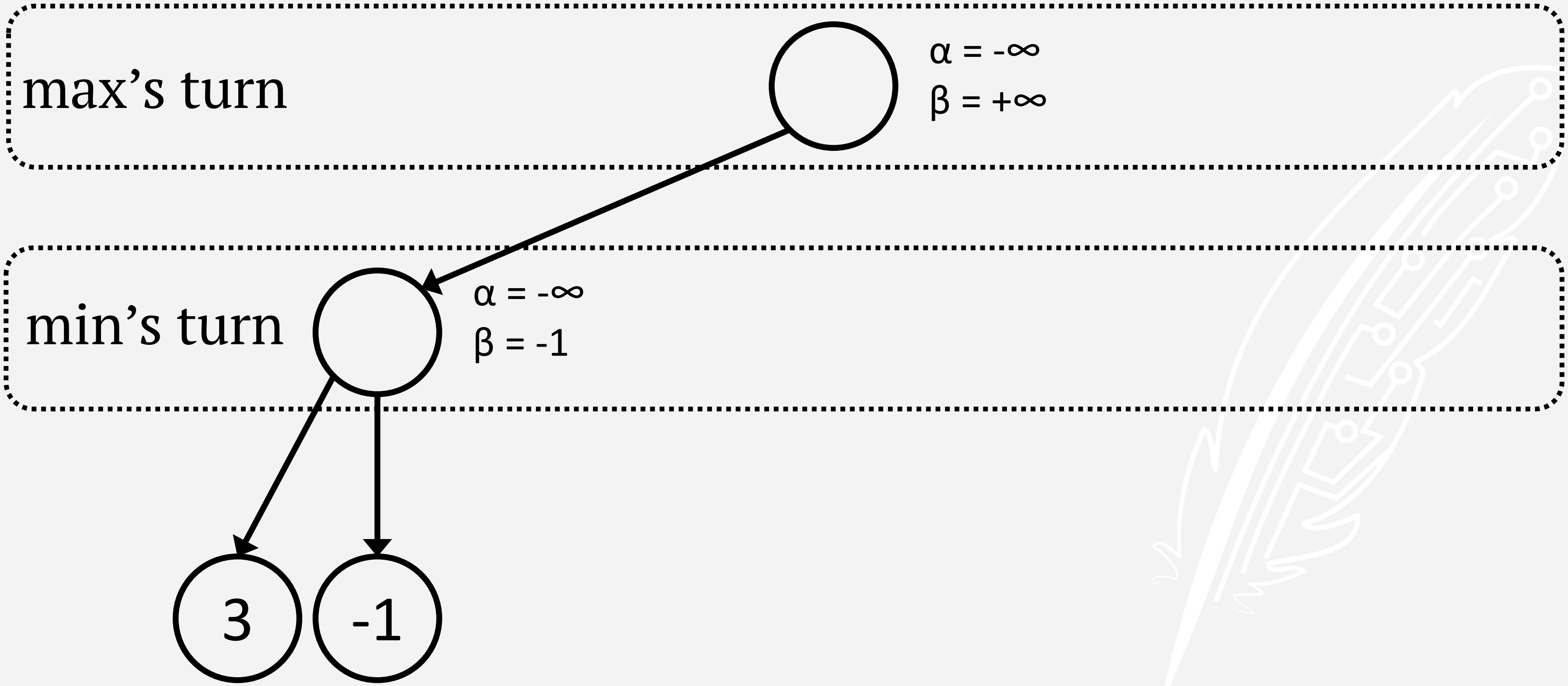
Alpha Prune Example



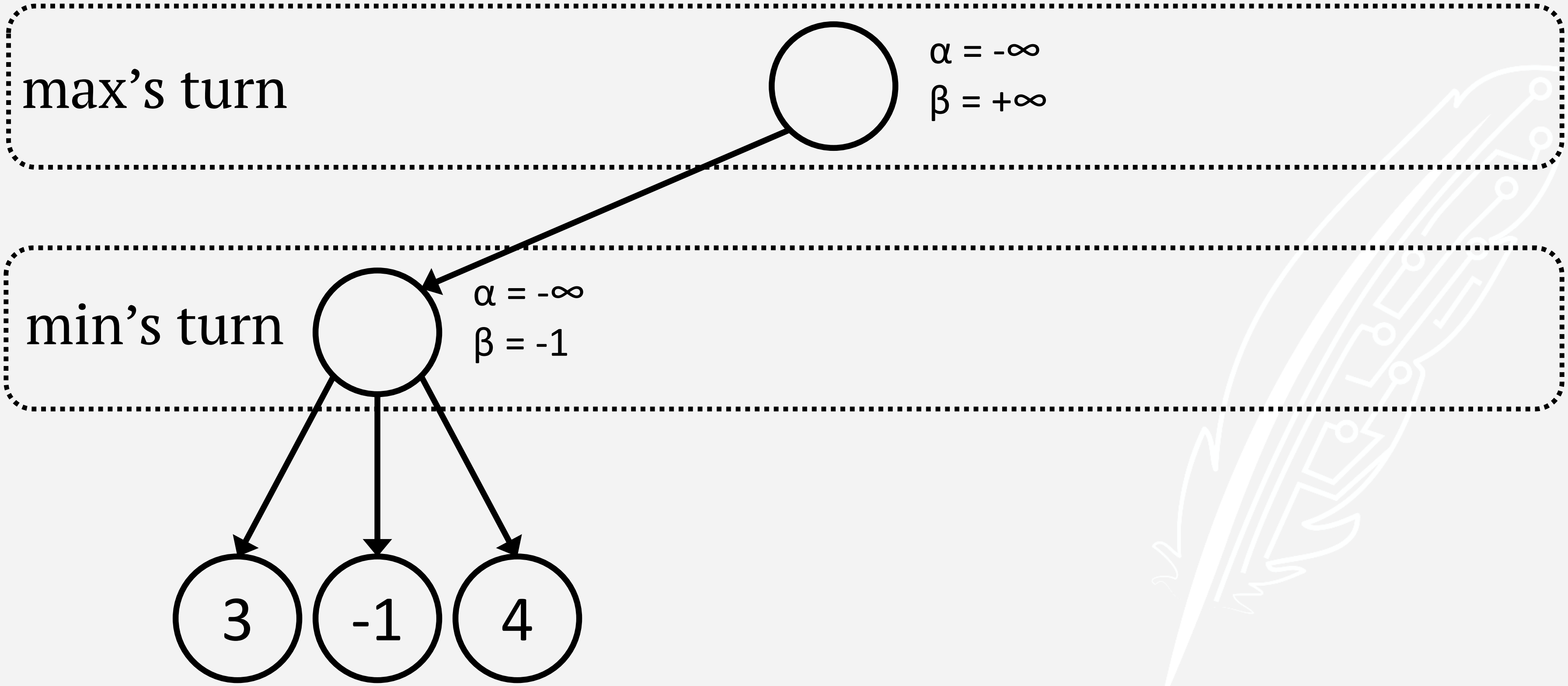
Alpha Prune Example



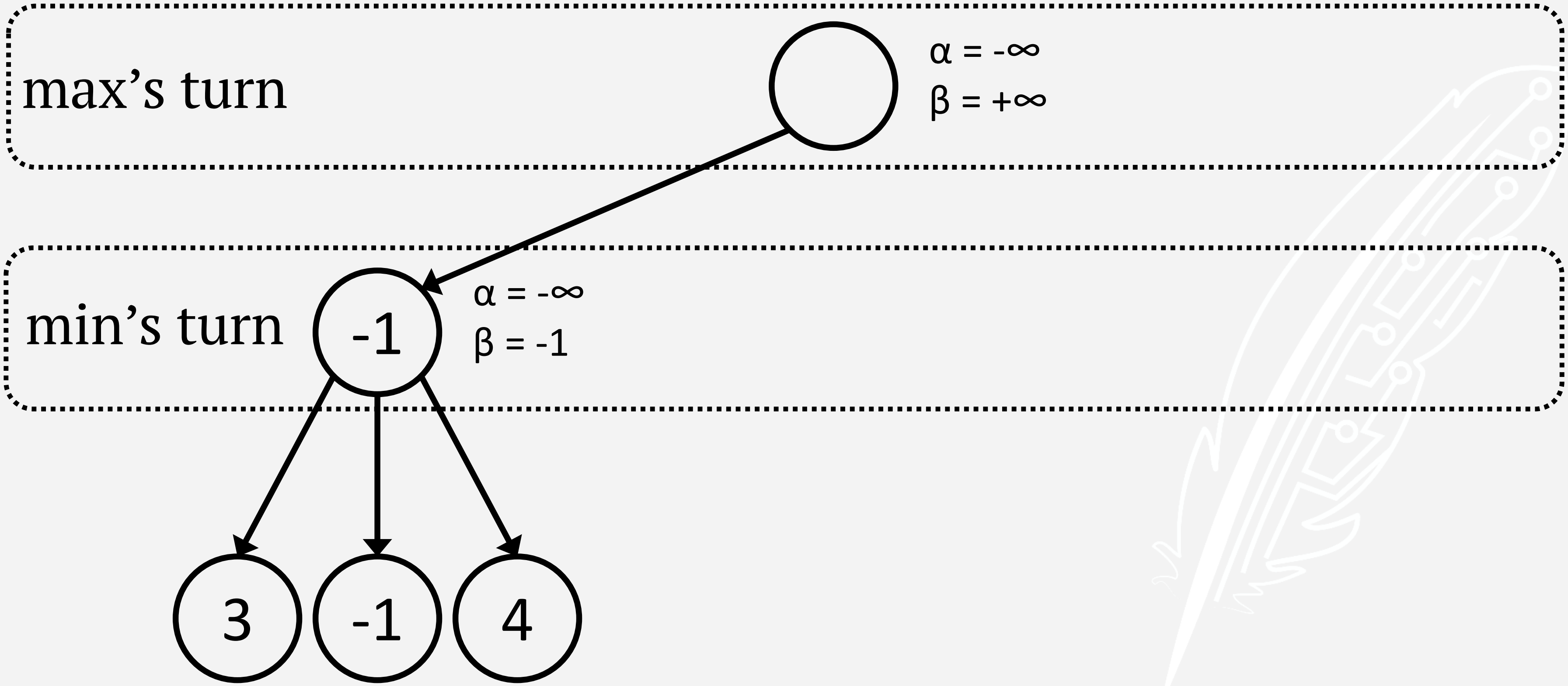
Alpha Prune Example



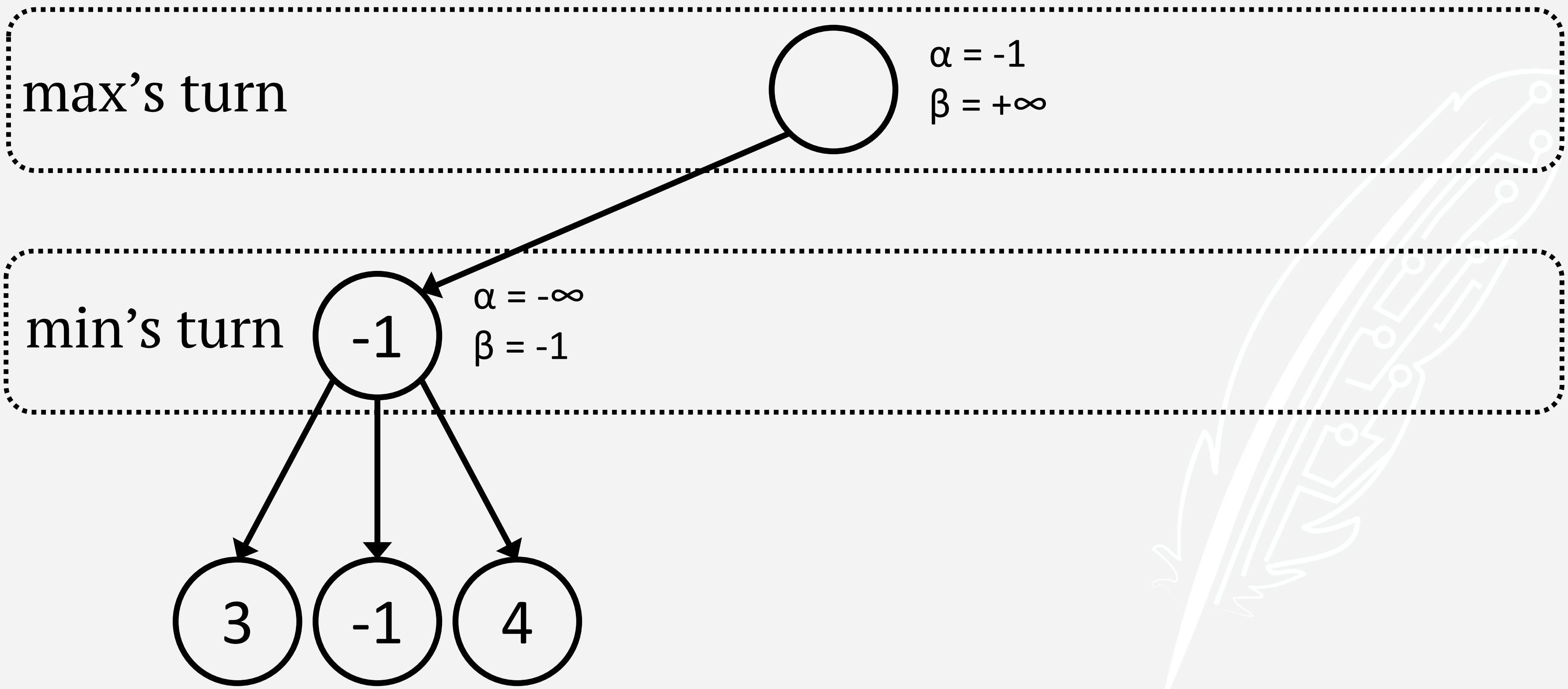
Alpha Prune Example



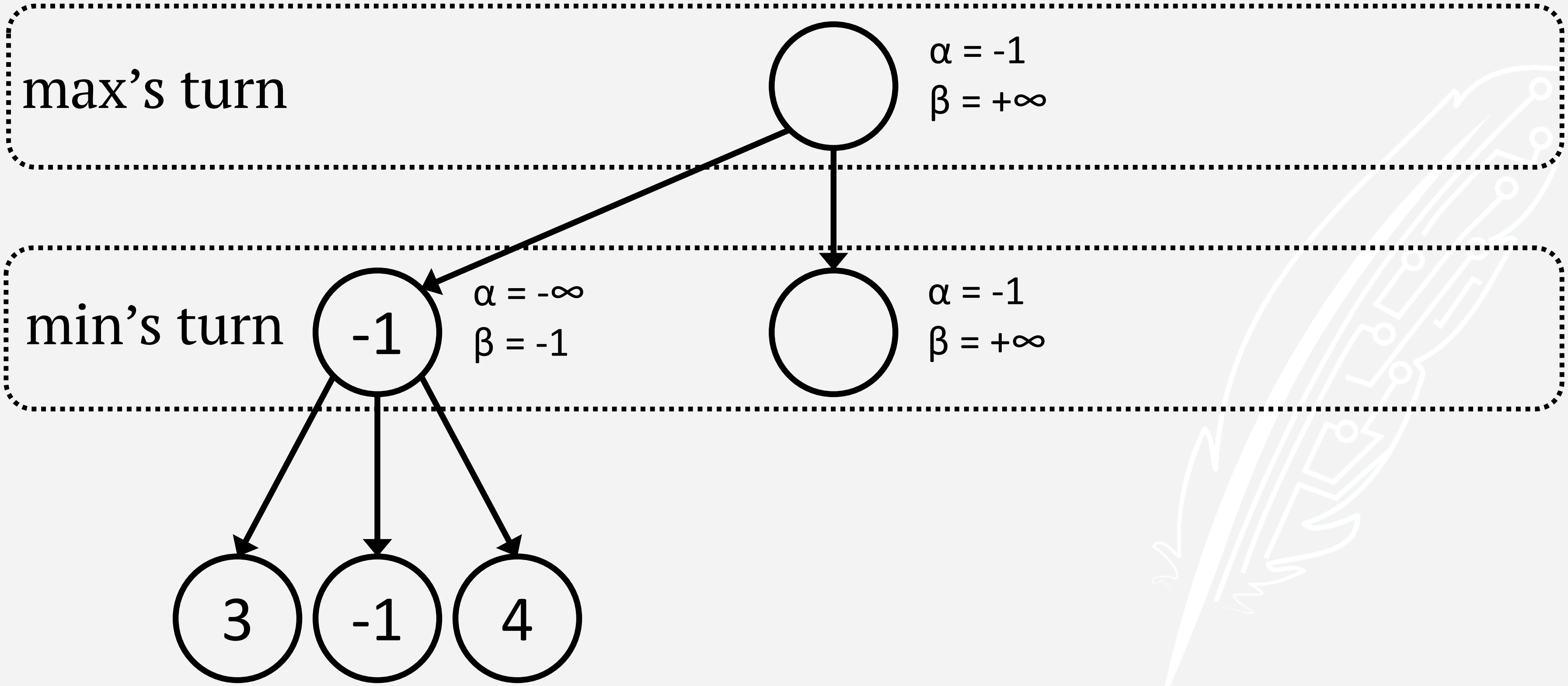
Alpha Prune Example



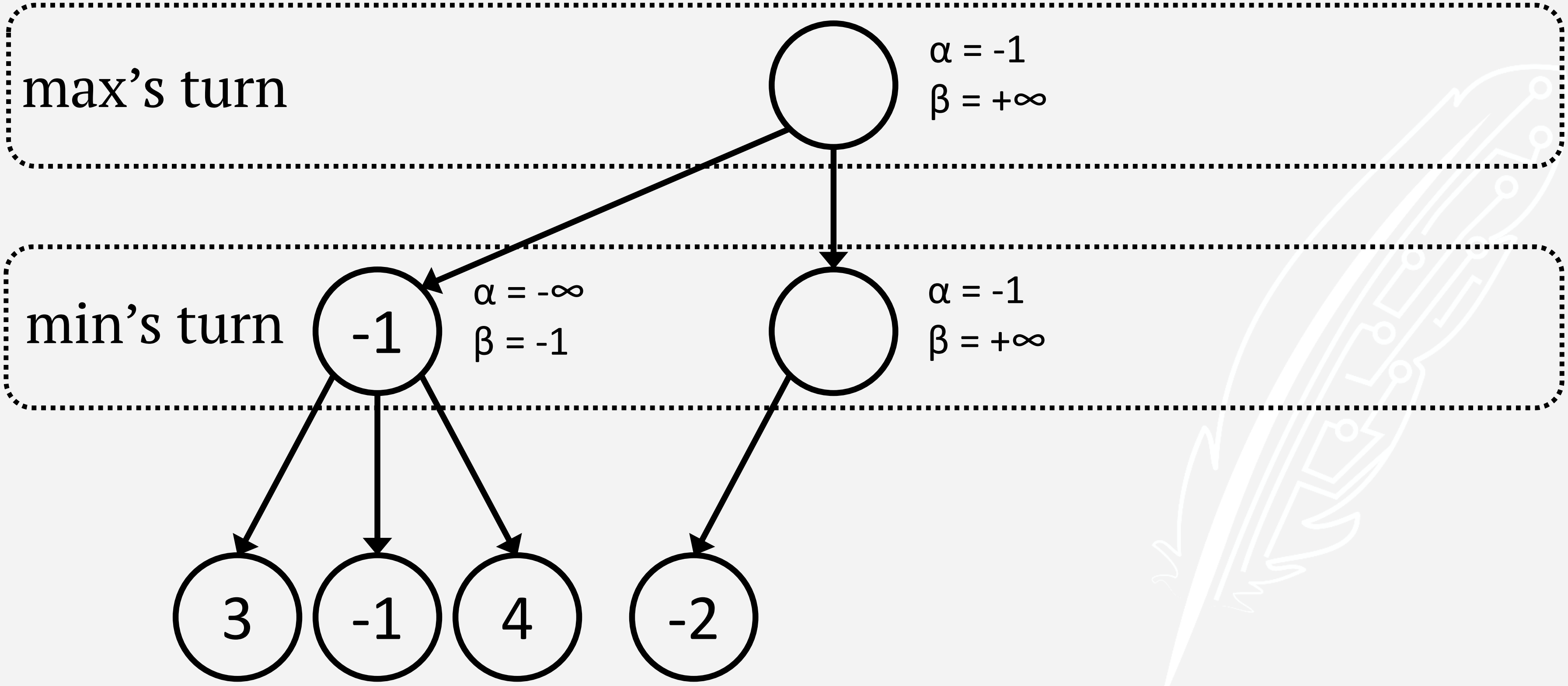
Alpha Prune Example



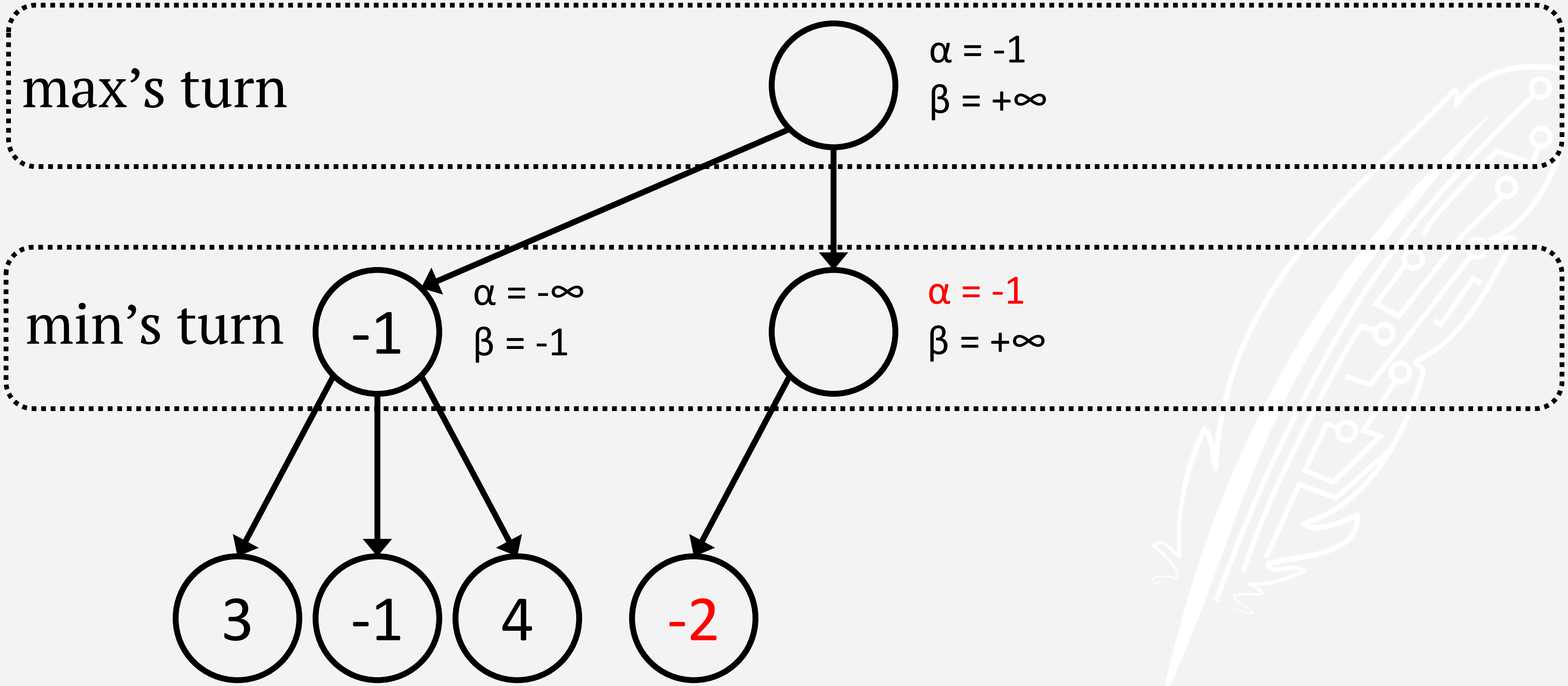
Alpha Prune Example



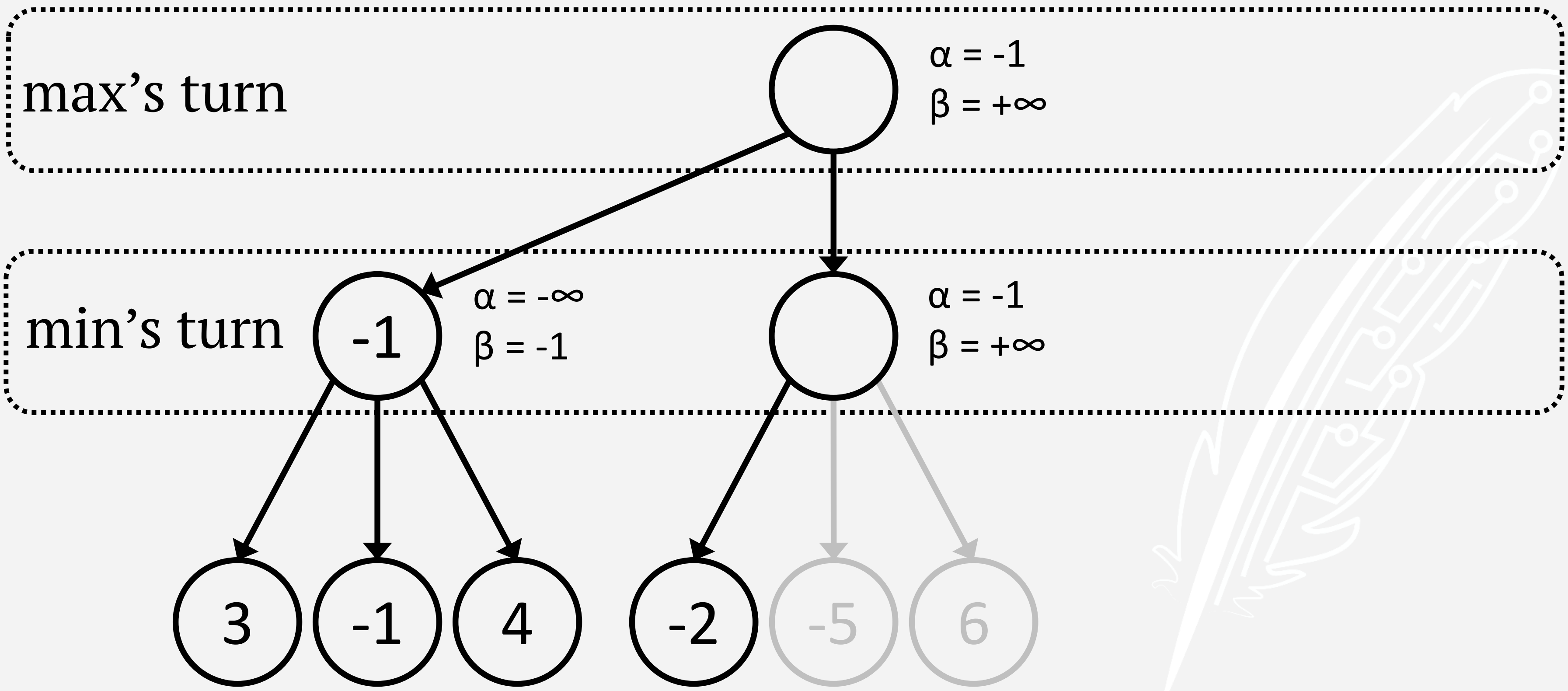
Alpha Prune Example



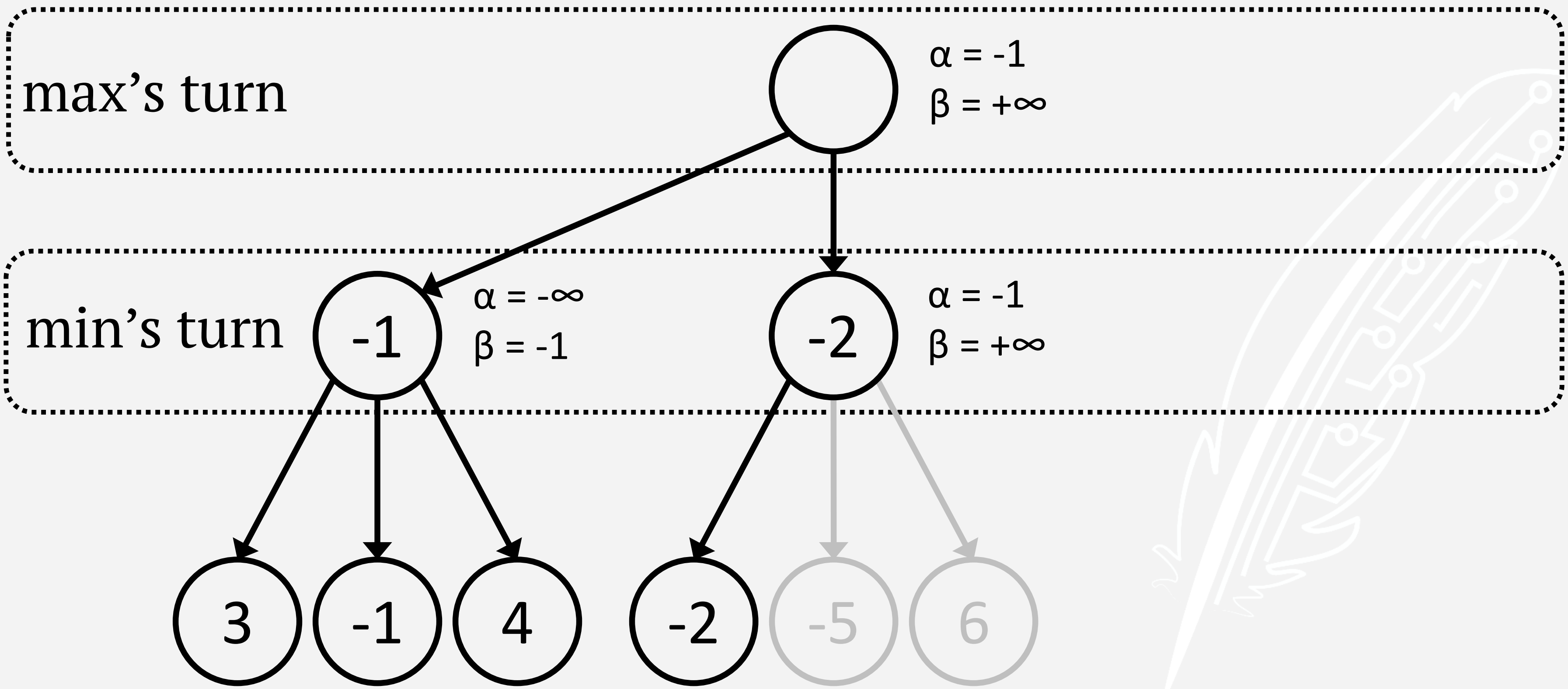
Alpha Prune Example



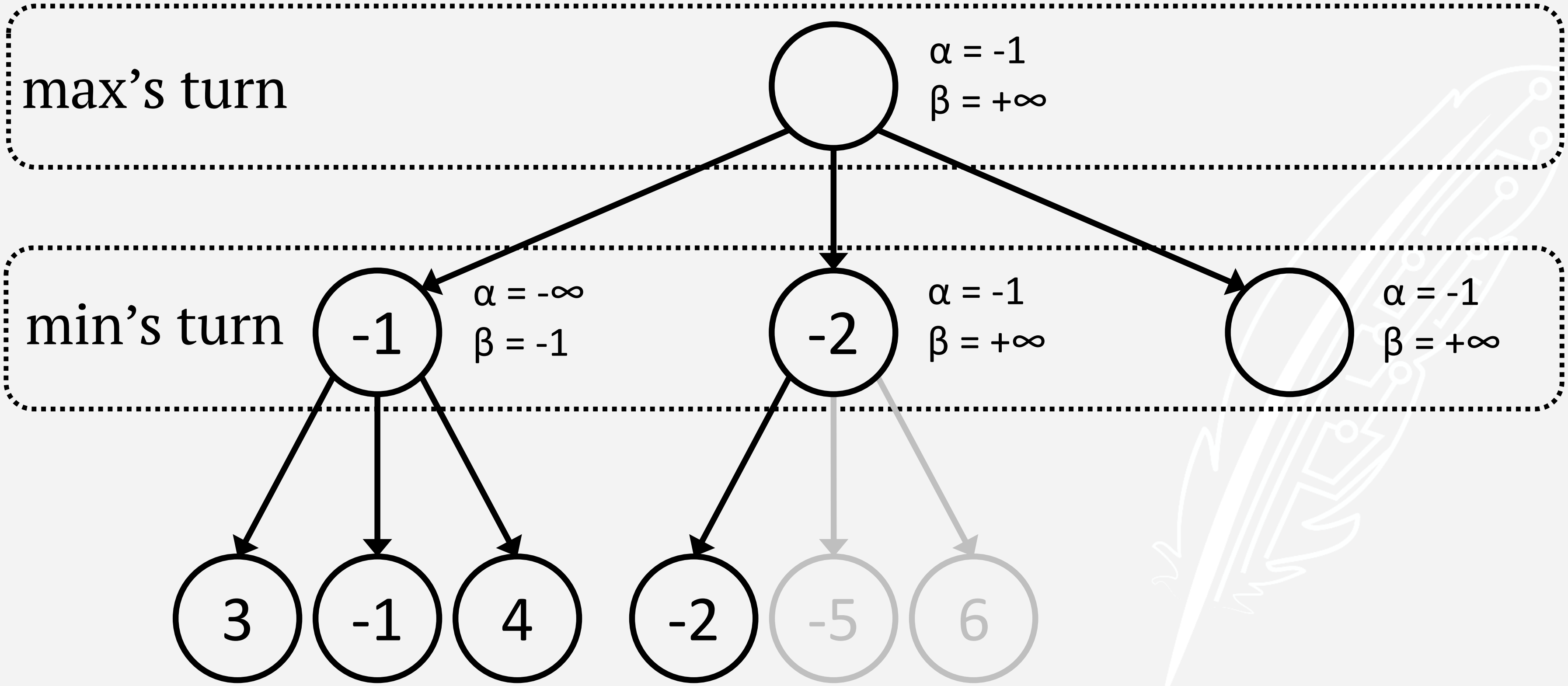
Alpha Prune Example



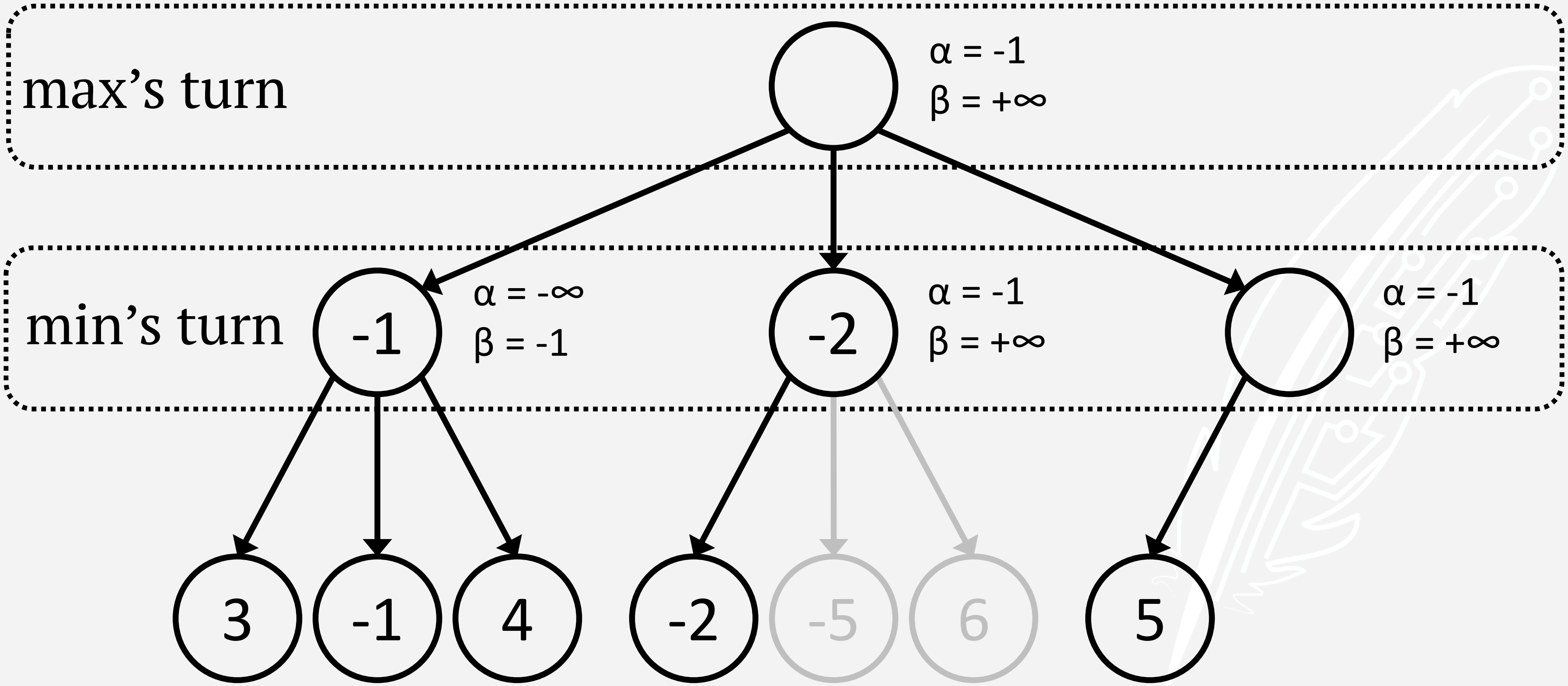
Alpha Prune Example



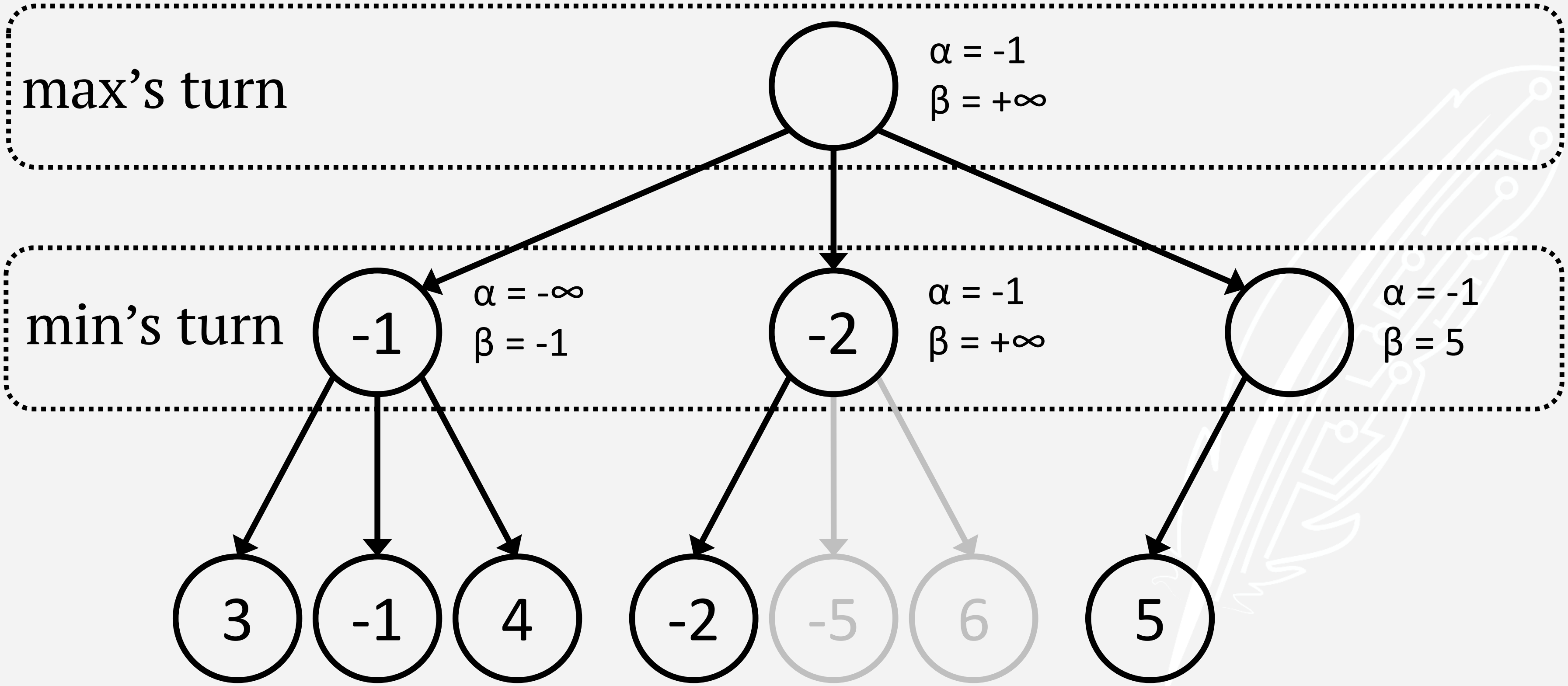
Alpha Prune Example



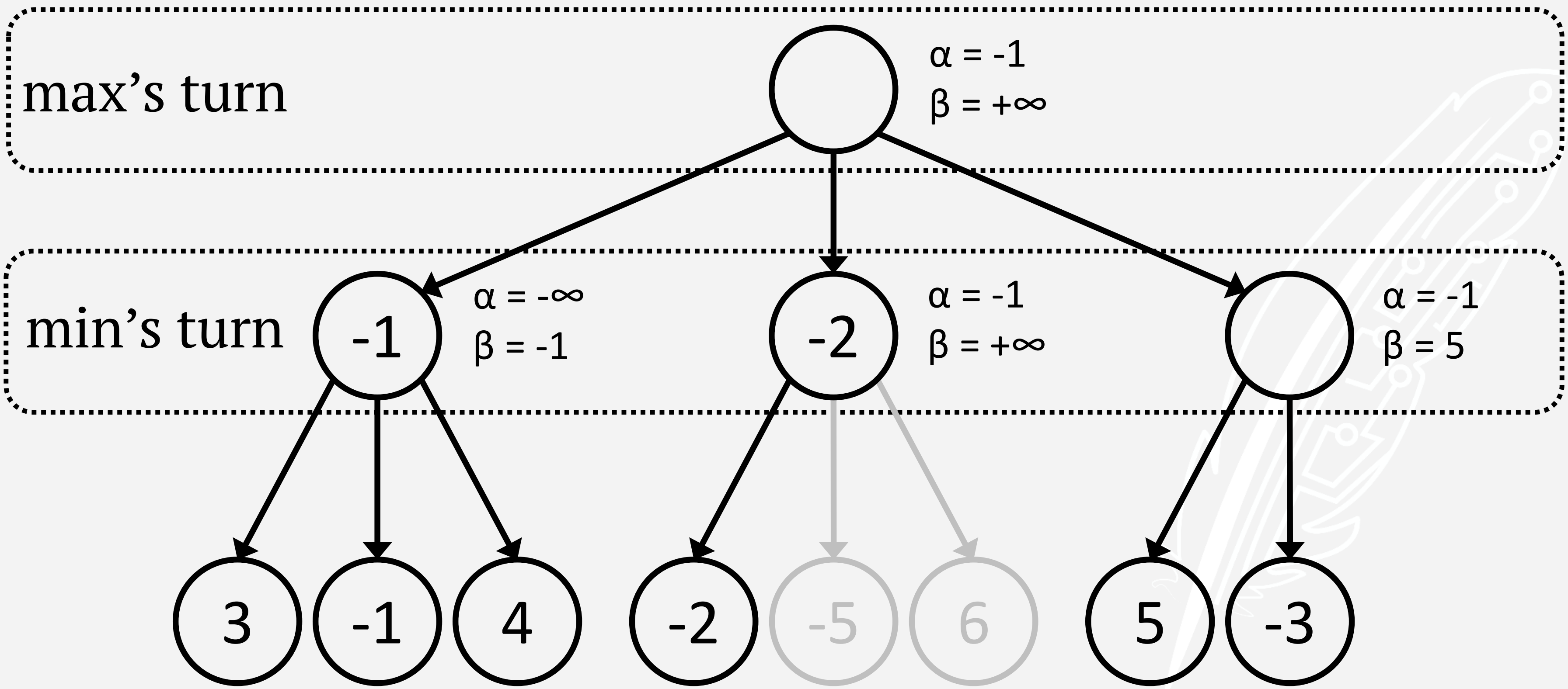
Alpha Prune Example



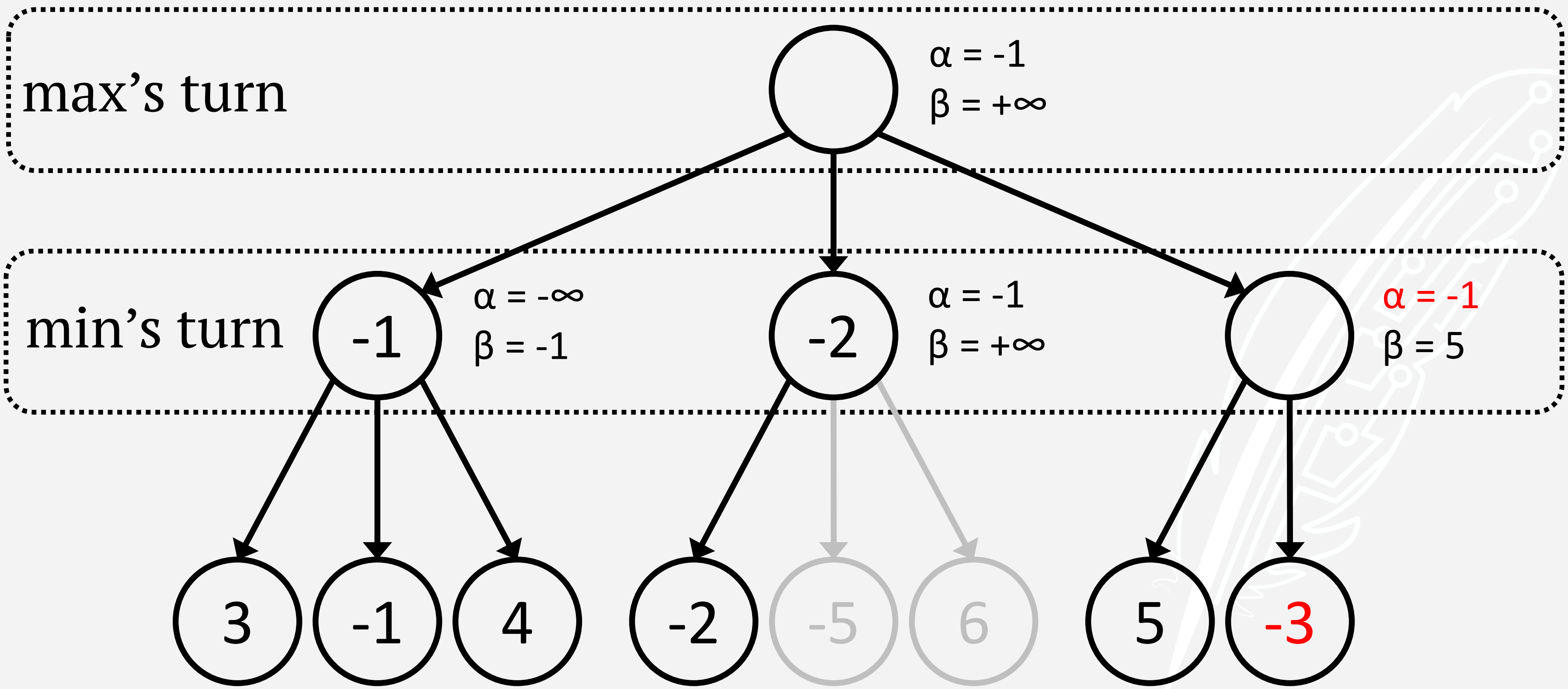
Alpha Prune Example



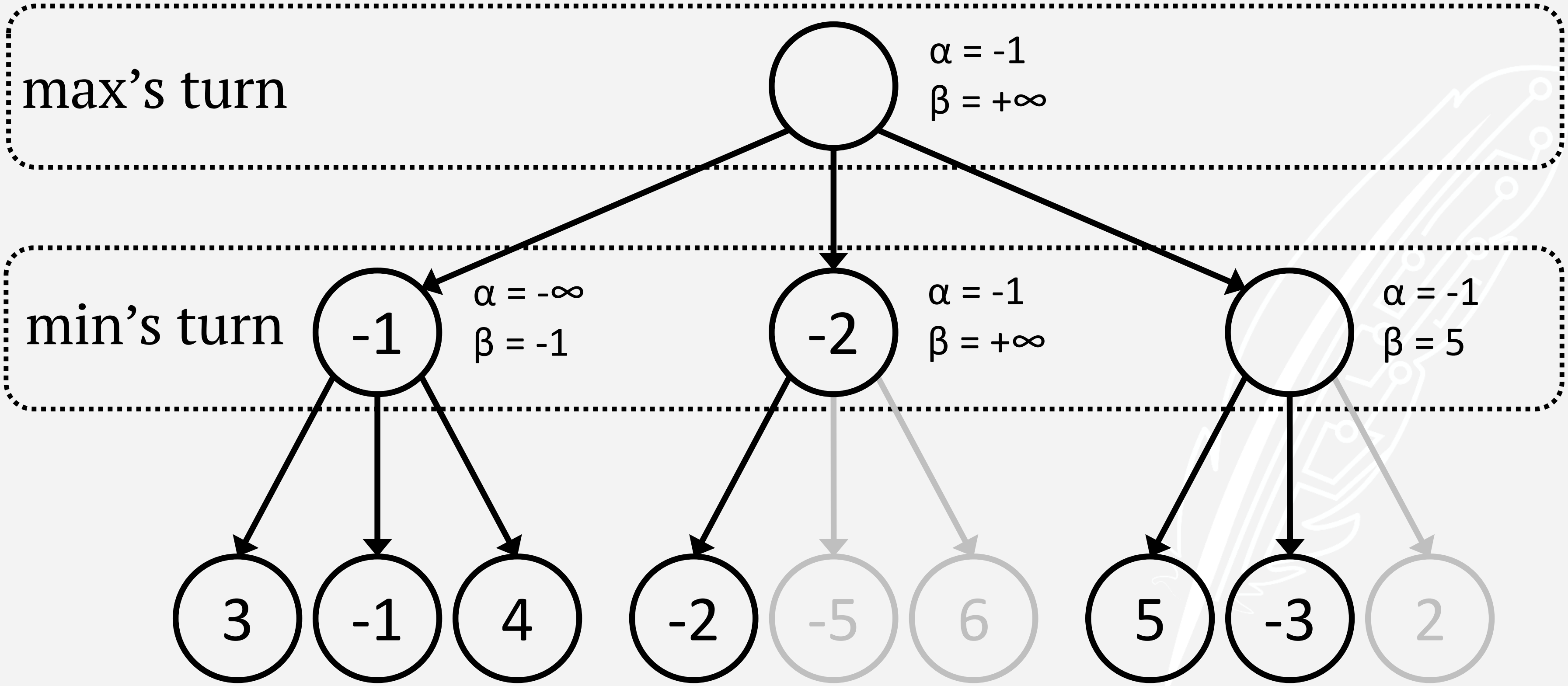
Alpha Prune Example



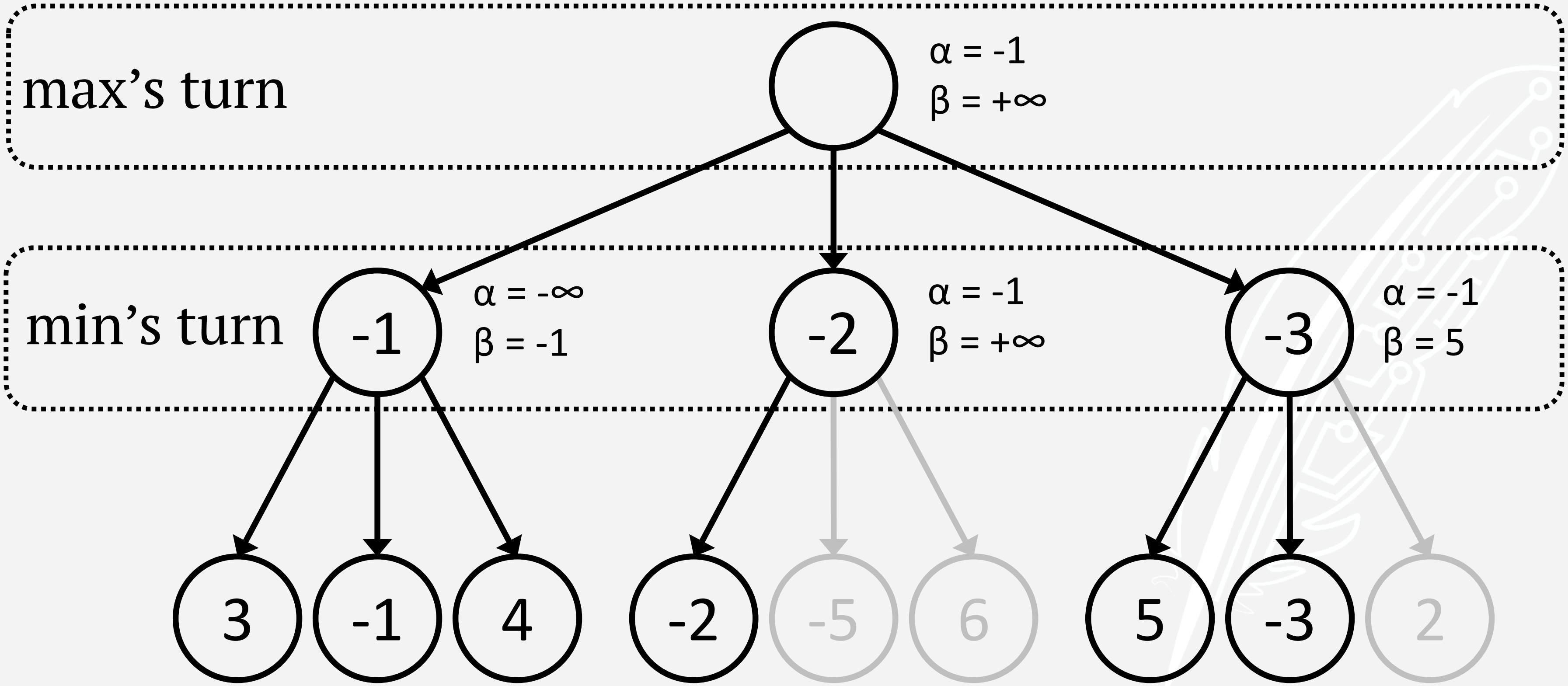
Alpha Prune Example



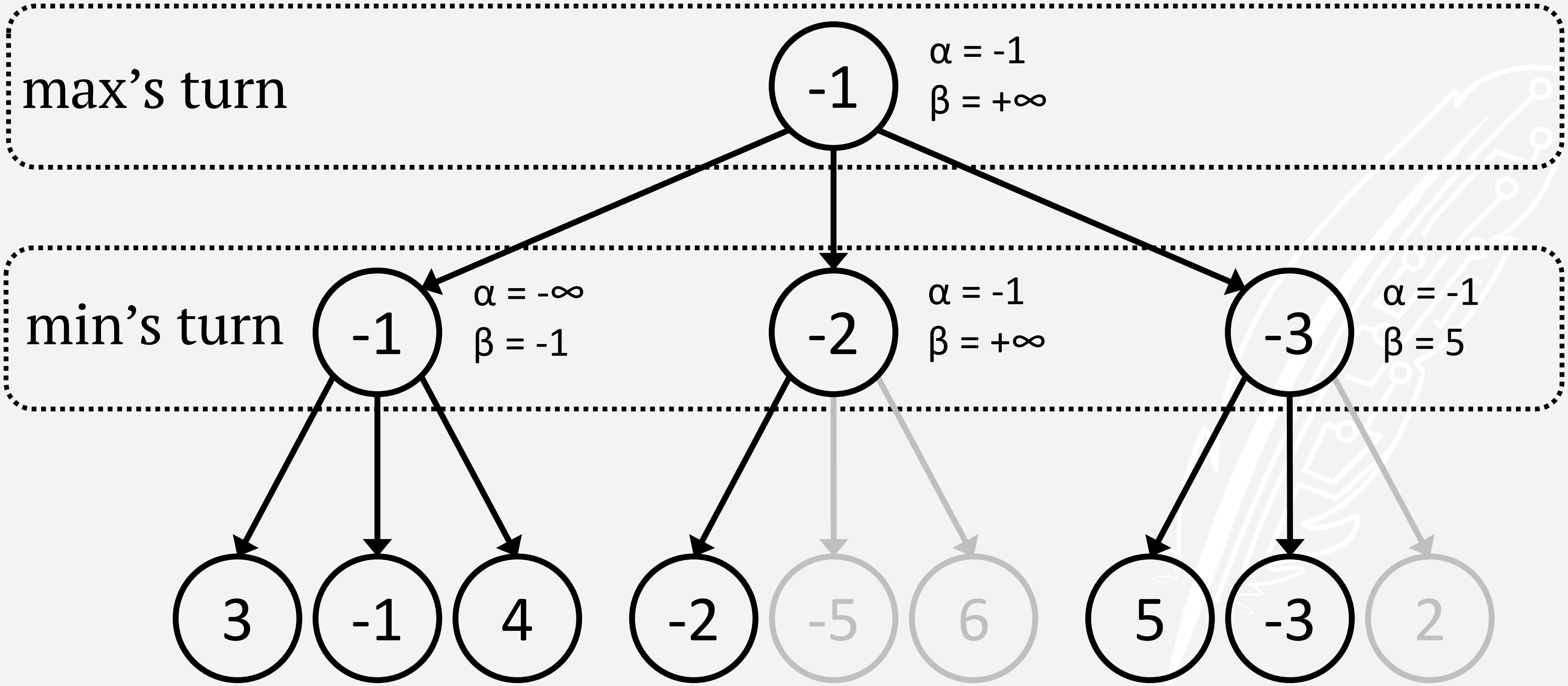
Alpha Prune Example



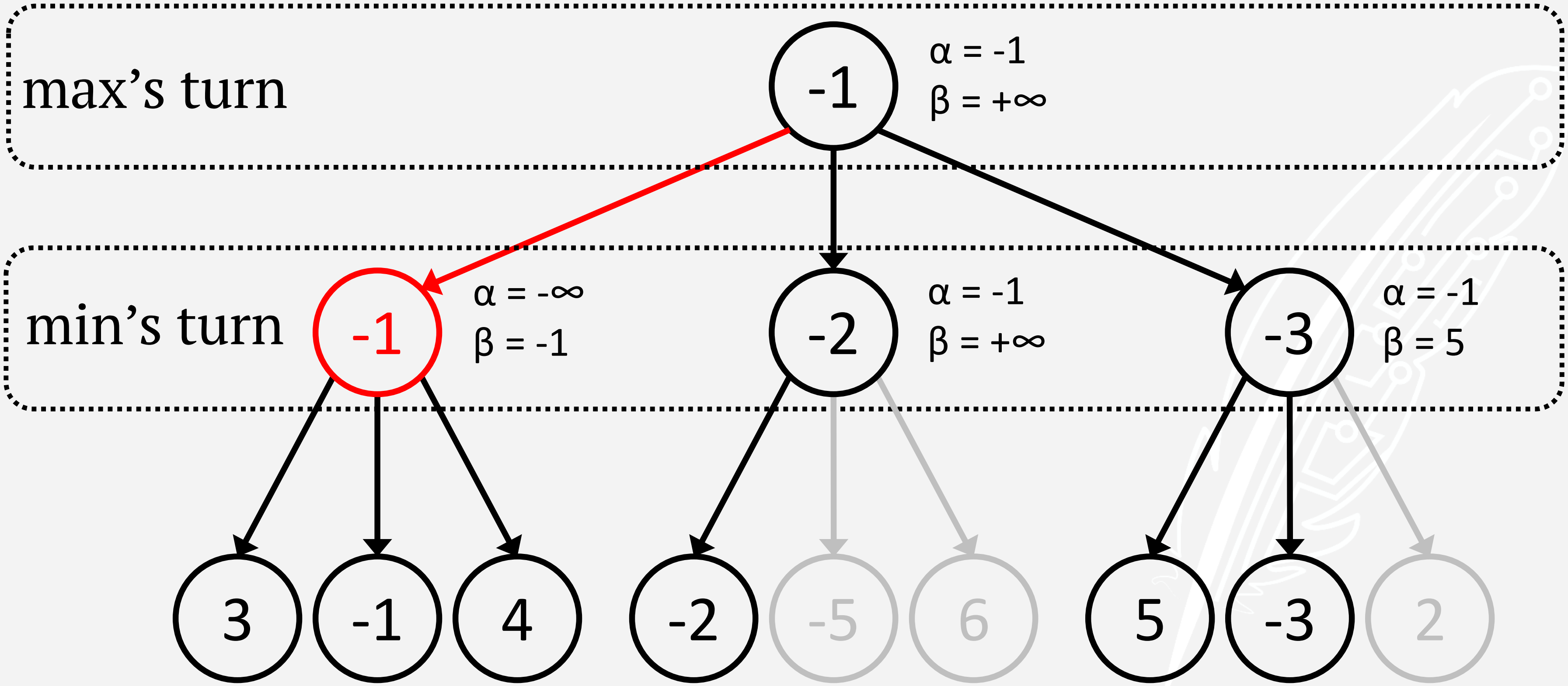
Alpha Prune Example



Alpha Prune Example



Alpha Prune Example



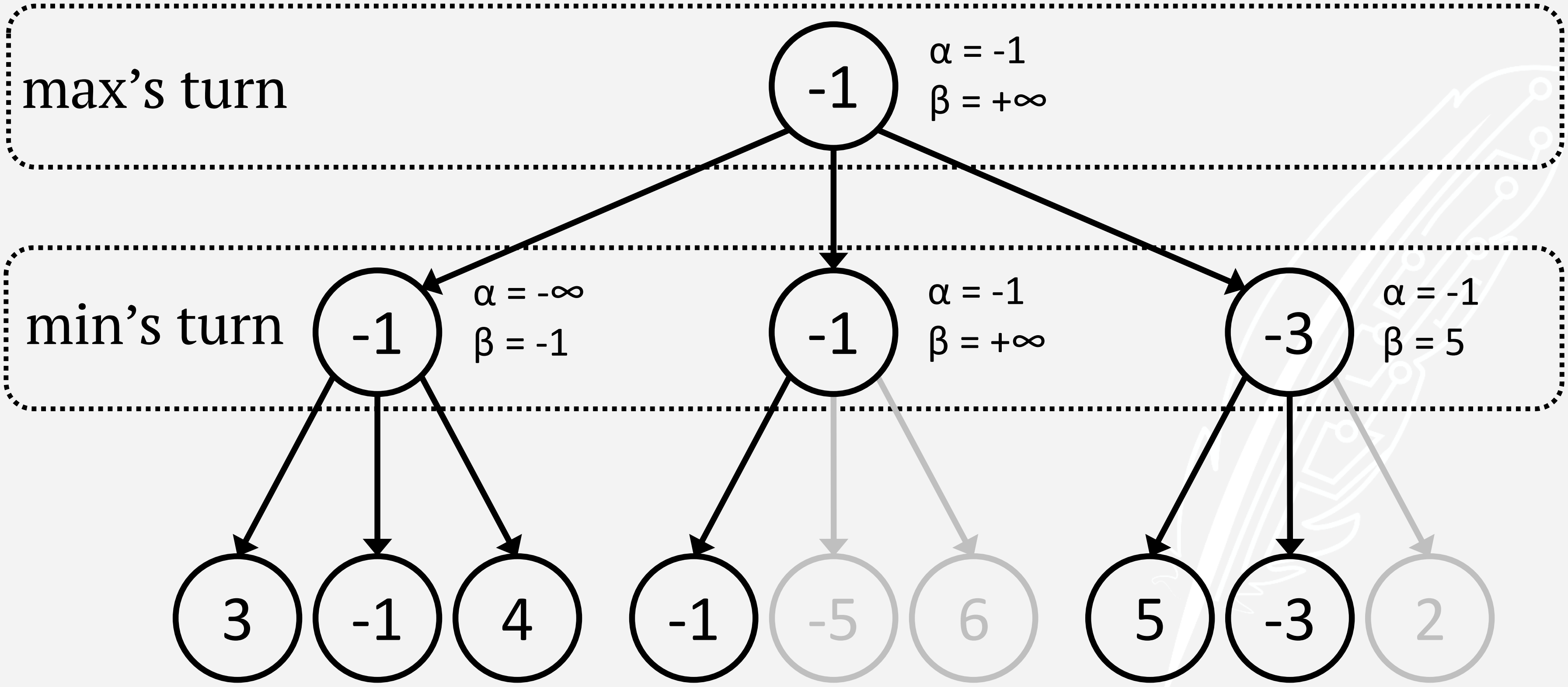
Alpha Beta Pruning

Returns exactly the same results as minimax, but can expand significantly fewer branches.

Pruned Branches

We do not know the exact value of a branch which has been pruned… only that it is high enough or low enough that we shouldn't bother with it.

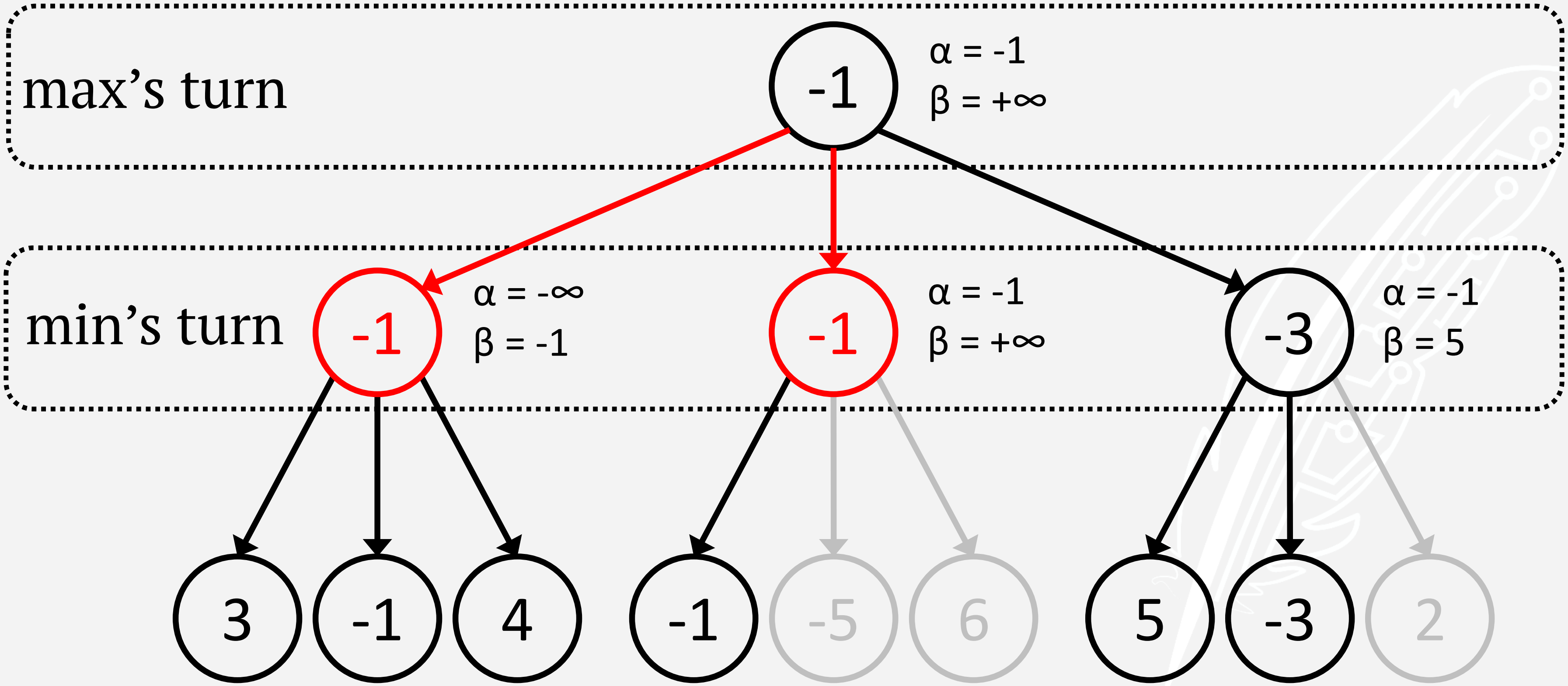
Alpha Prune Example



NIL

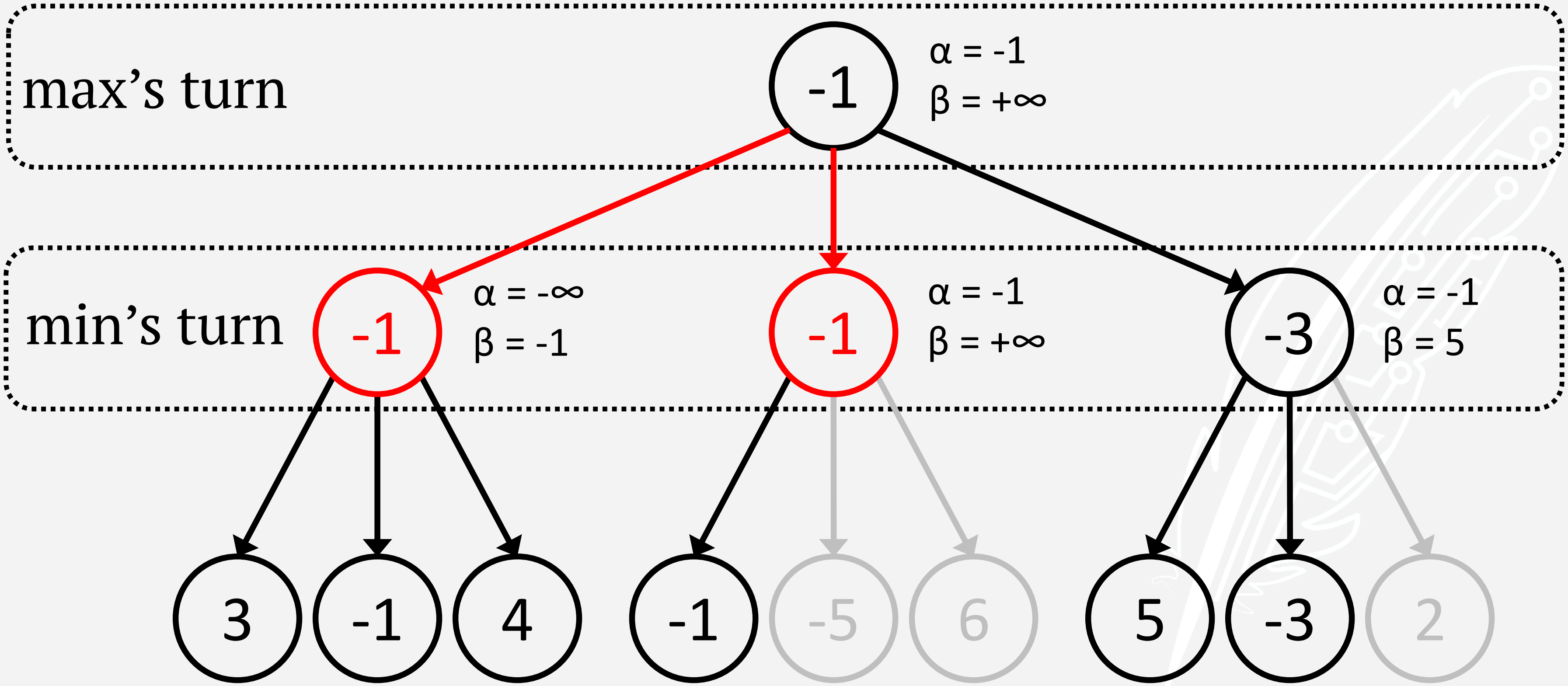
Alpha Prune Example

Does it matter which one we choose?



Alpha Prune Example

Does it matter which one we choose? Yes! Don't choose the pruned branch.



Limited Time and Space

It is often impractical to go all the way to a leaf.

Imperfect Decision Making

- Design a utility function which can evaluate the state of a game in progress (e.g. chess material).
- Impose an arbitrary cut off past which we treat nodes as leaf nodes even if they are not.
- No longer guaranteed to find the best strategy.

Forward Pruning

- **Beam Search** limits the number of children that a given parent can have.
- Example: Consider only the 5 best children of a node, and don't bother with the rest.
- Dangerous because we might prune the best or worst branch.

Iterative Deepening

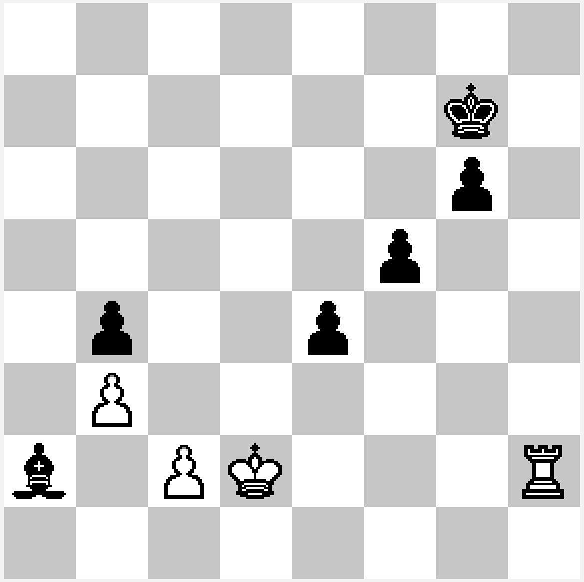
- If we run out of time or memory before minimax search is finished, we can't trust the answer it gives us.
- One solution is to use iterative deepening.
- Chess example: First look 2 moves ahead. If that finishes, look 4 moves ahead. If that finishes, look 6 moves ahead. Keep increasing this value until we run our of time or space.

(Make sure not to trust the last, incomplete search!)

Move Ordering

- The sooner we find very high and very low moves, the more nodes can be pruned.
- Consider moves in an order that is likely to discover high and low values sooner.
- Chess example: always consider moves which capture a piece first. These are likely to lead to more extreme utility values and thus improve pruning.

Horizon Problem



Material

White: 7

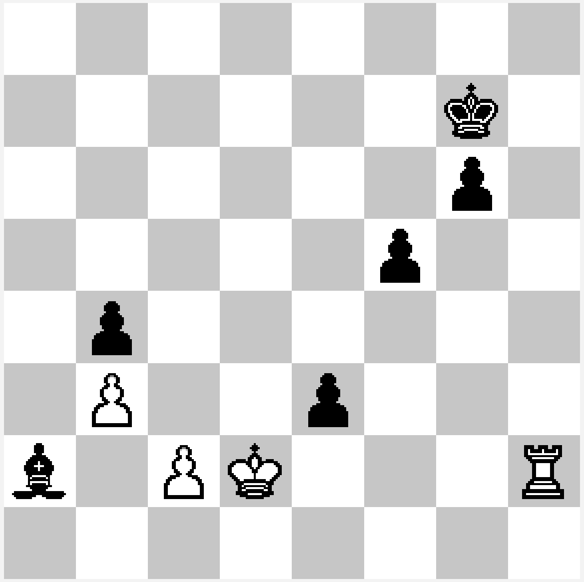
Black: 7

Black to move.

Black looks 2 moves ahead.

If the bishop is captured, black's score will drop to 4.

Horizon Problem

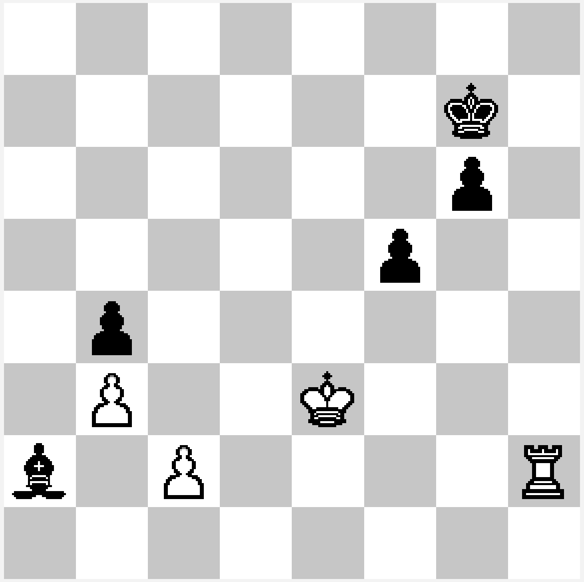


Material

White: 7

Black: 7

Horizon Problem



Material

White: 7

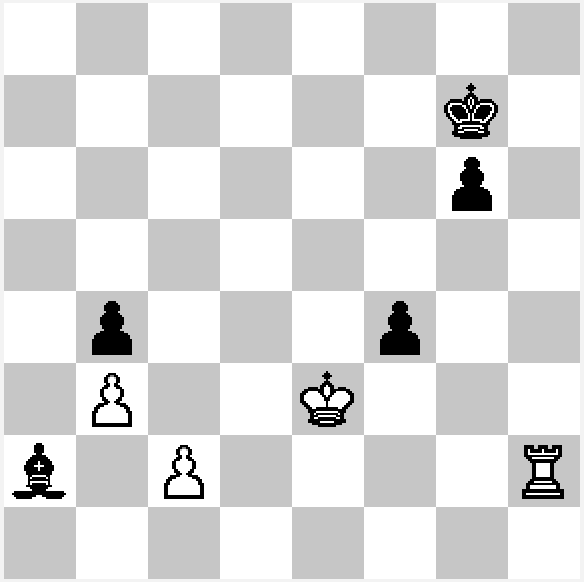
Black: 6

Black to move.

Black looks 2 moves ahead.

If the bishop is captured, black's score will drop to 3.

Horizon Problem

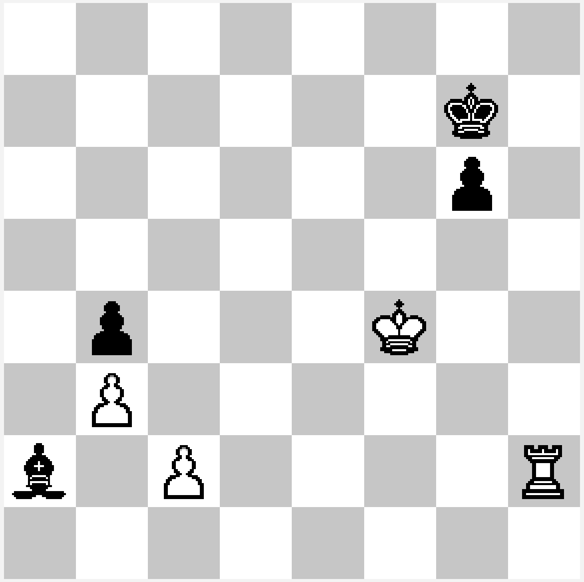


Material

White: 7

Black: 6

Horizon Problem



Material

White: 7

Black: 5

Quiescence

- When search is arbitrarily cut off, if the next move is really good or really bad, we won't see it.
- Only cut off the search if we have reached the max depth limit AND the game is in a relative state of equilibrium (e.g. next moves are not big ones).
- Chess example: Cut off search if we have reached the max depth AND the previous move was not a capture.