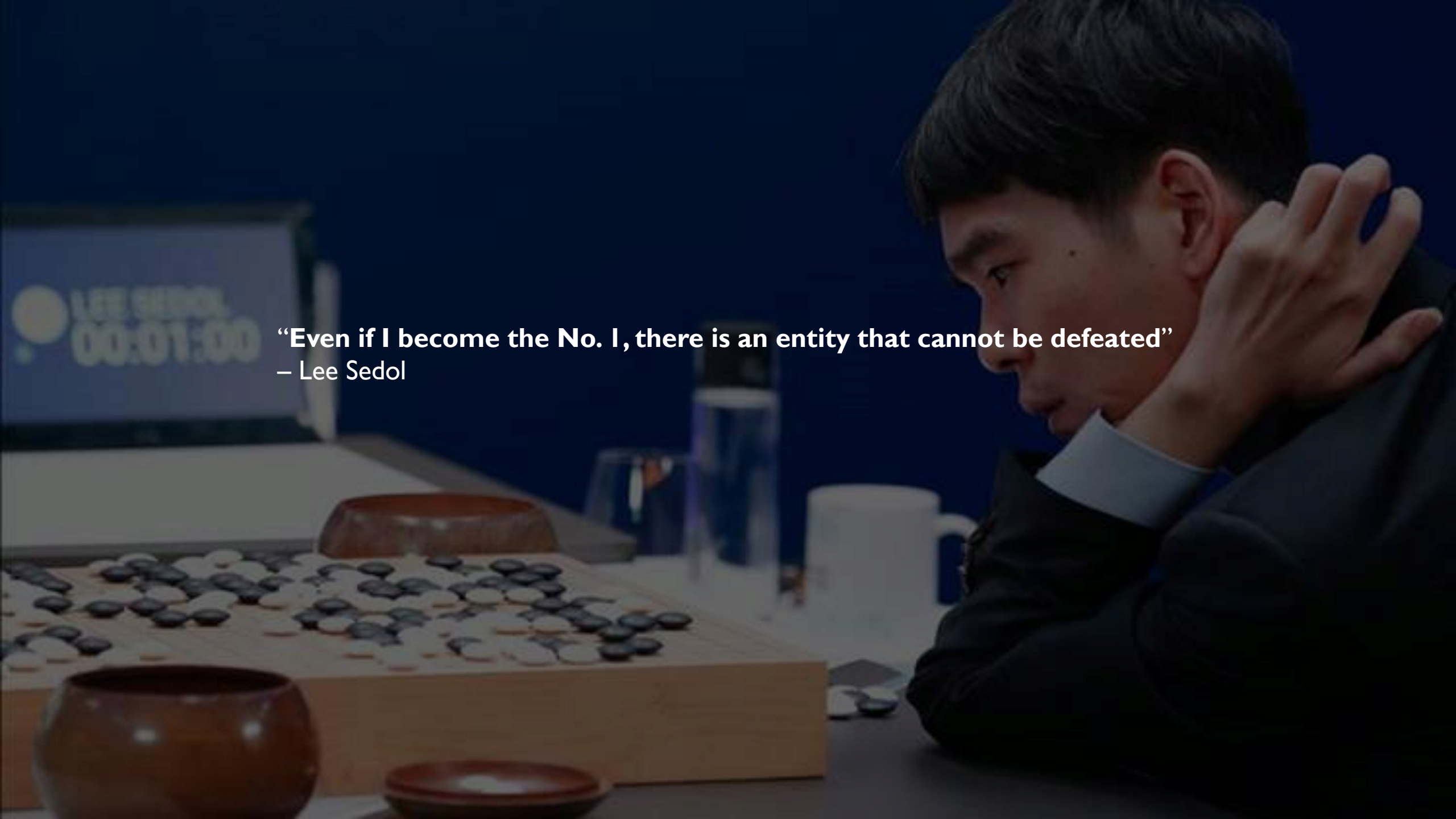


INTRODUCTION TO AI - HEURISTIC BASED AGENTS

Philip Mortimer

A photograph of Lee Sedol, a professional Go player, sitting at a table and playing Go. He is wearing a dark suit and is looking down at the board with a focused expression. His right hand is raised to his head, with fingers spread, suggesting deep concentration or a moment of stress. The Go board is filled with black and white stones. In the background, a laptop screen displays the text "LEE SEDOL 00:01:00".

“Even if I become the No. 1, there is an entity that cannot be defeated”
– Lee Sedol

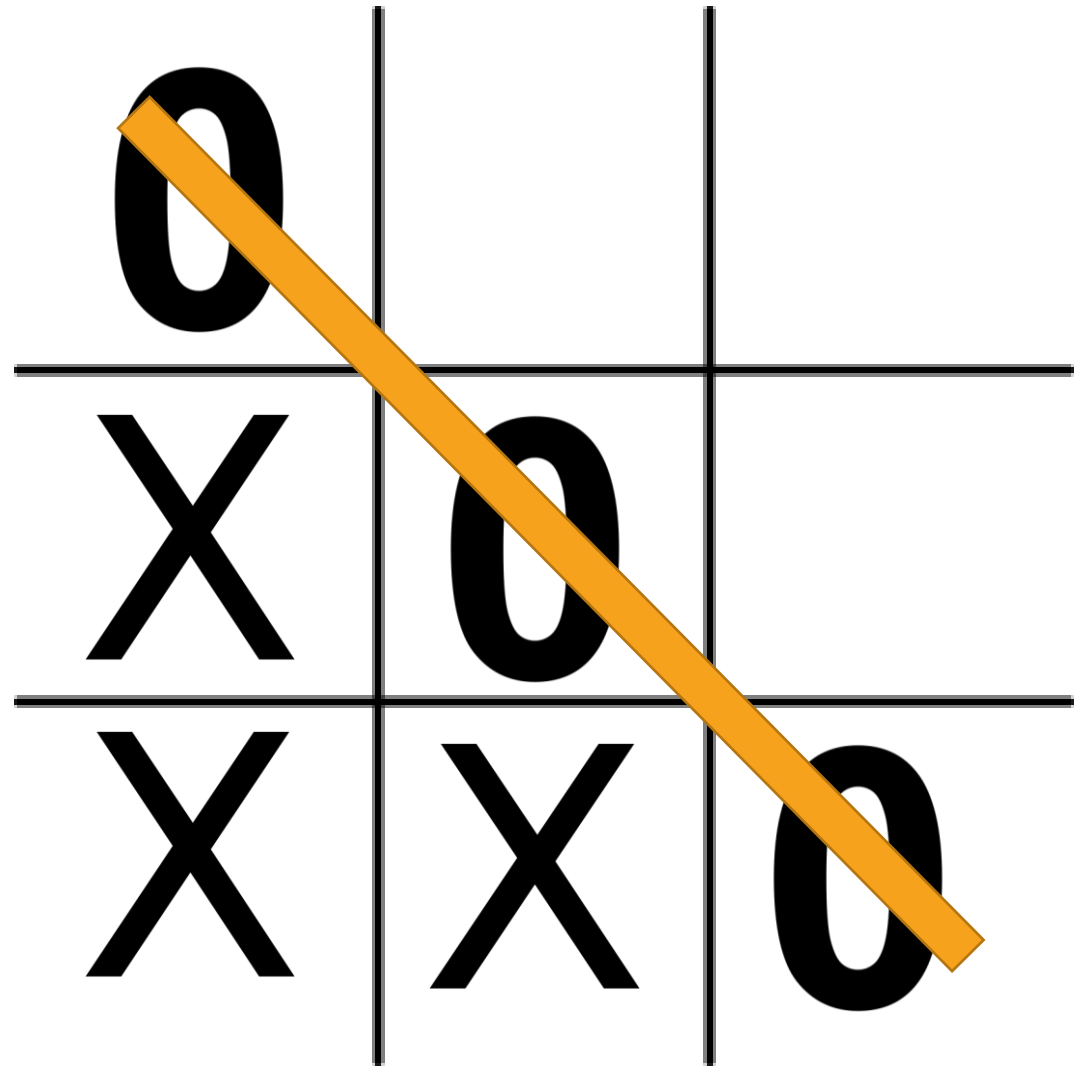
RECAP

- Discussed how to solve mazes using depth first search
- Introduced breadth first search as a way to solve mazes optimally
- Looked at tree structures as a way to represent mazes
- Dijkstra's Algorithm
- A* search
- Degrees of separation

TIC TAC TOE

- In Tic Tac Toe (also known as Noughts and Crosses!) two players take into in turn to either play a cross or a nought.
- The first player to have a horizontal, vertical or diagonal line of one symbol wins.
- If neither player manages a line, the game ends in a draw.

TIC TAC TOE



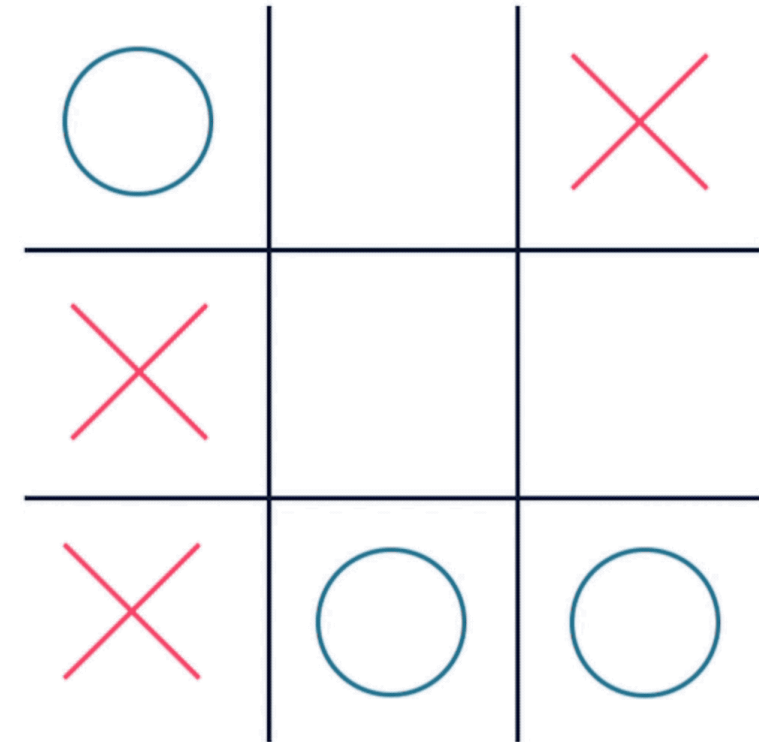
Noughts win!

TIC TAC TOE

Tic Tac Toe is a solved game, meaning perfect play is known. Why can Tic Tac Toe be solved?

There are a **finite** number of possible games of tic tac toe (around 255,168 games)

Let's design an AI agent that play tic tac toe perfectly



DESIGN STRATEGY

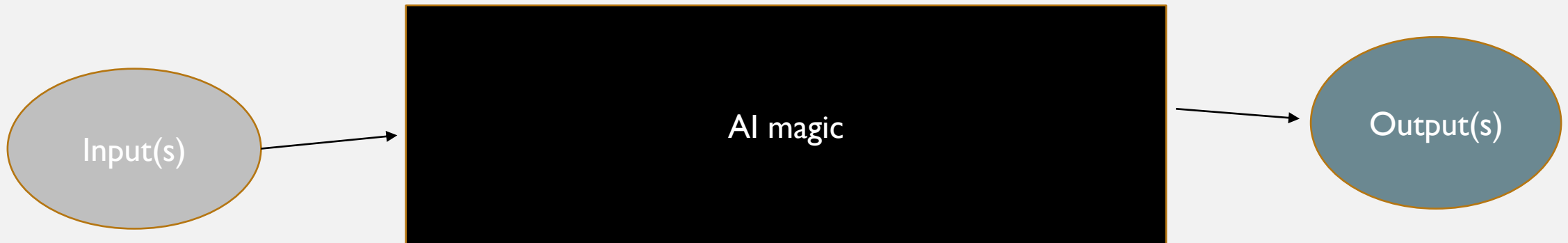
- When designing an AI system, it's important to design and plan everything **before** any coding starts
- When designing, we need to decompose a high level overview into small achievable tasks. This is known as decomposition
- Let's first adopt a black box, high level overview
- What would like our AI to do?
- This can be decomposed into thinking about our input and output.
- What is our input going to be?
- What is our output going to be?

DESIGN

What do we want our AI to do?

Input(s)

Output(s)

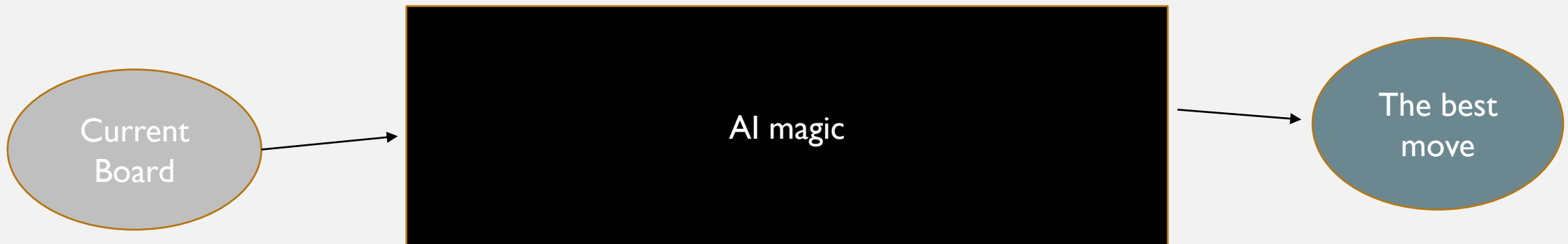


DESIGN

What do we want our AI to do? – Play perfect Tic Tac Toe

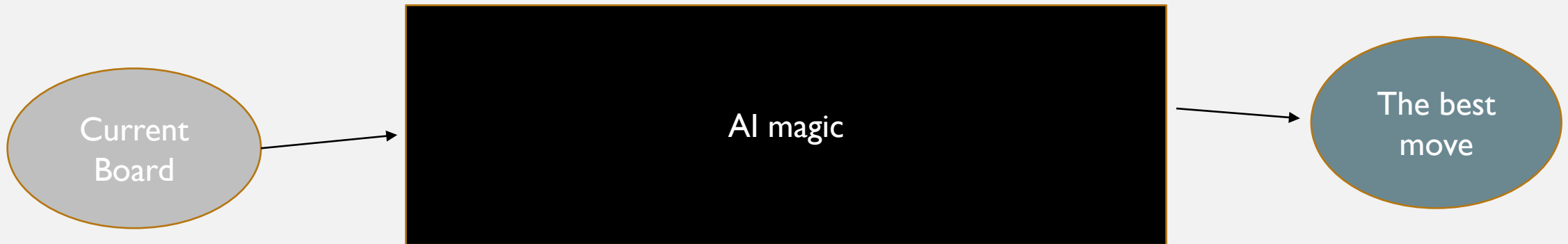
Input(s) – The Current Board state

Output(s) – The best move



DESIGN

- Once we know the input and output of the system, we can work on producing an actual implementation (i.e. the black box part).
- This idea of decomposition and producing precise system requirements are essential to creating an AI system.



TIC TAC TOE

- Let's think about how we would go about playing tic tac toe perfectly? What should we do?
- Tic Tac Toe is an example of a system where both sides have all information available to them.
- Looking at this position, we know that Noughts has to play in the top right corner to avoid defeat.
- Therefore, we know that these two positions are equivalent.
- In essence, what we want to do is look at all of our possible moves for a given board. For each of those moves, we need to look at all available moves that our opponent has. Doing this many times will allow us to play perfectly.

		X
	O	X
		O
		X
	O	X

MINIMAX

- Tic Tac Toe is an example of an adversarial search.
- We already have learned about a range of search algorithms. We are going to use a variation of an algorithm we already have seen to play tic tac toe.
- First, we need to understand the game state of tic tac toe. A game can either be won, lost or drawn. For the sake of simplicity “won” = “won by crosses” and “lose” = “lost by crosses”.
- Every board state represents some game which will eventually be in one of these states. Thus, we can assign a value to each board based on this.

	0	
	0	
X	X	X

+1

0	X	X
X	0	0
X	0	X

0

X	X	
0	0	0
X		

-1

MINIMAX

- Lets give each of these three states some numerical representation.
- Let's say a "win" (for crosses) is = 1
- A "draw" = 0
- A loss = -1

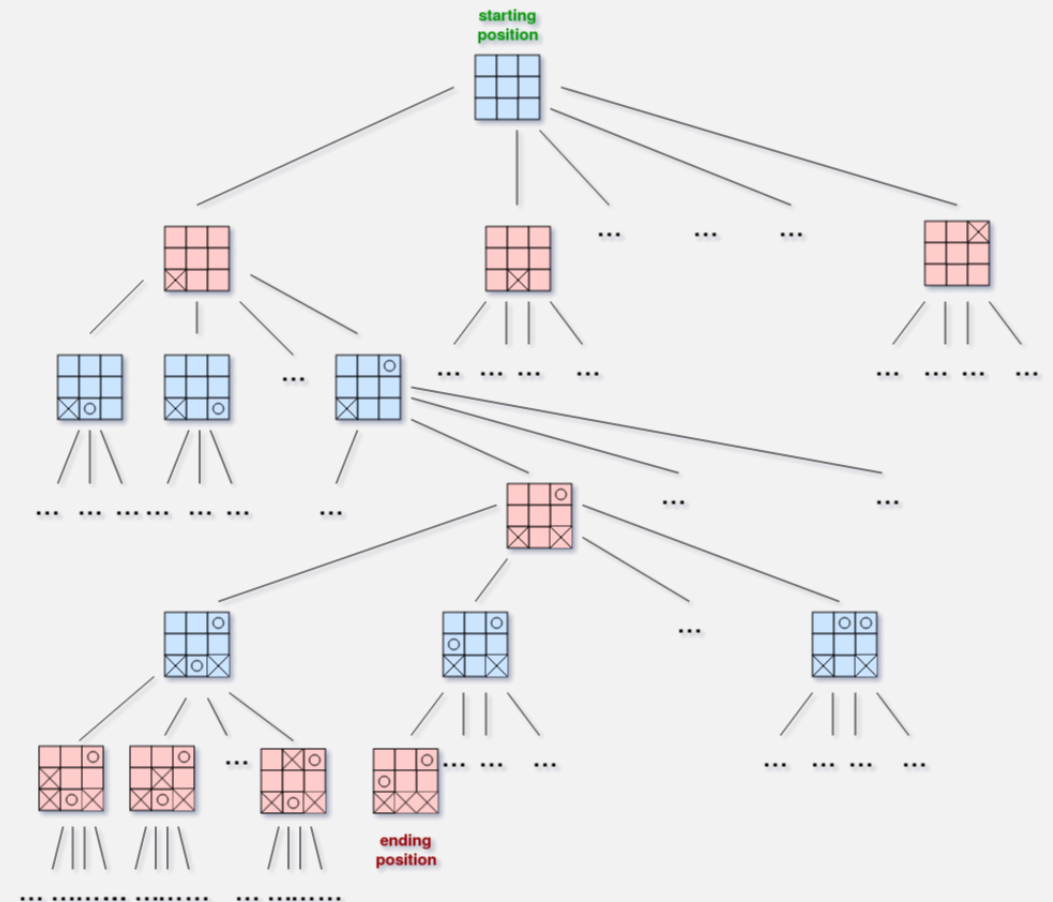
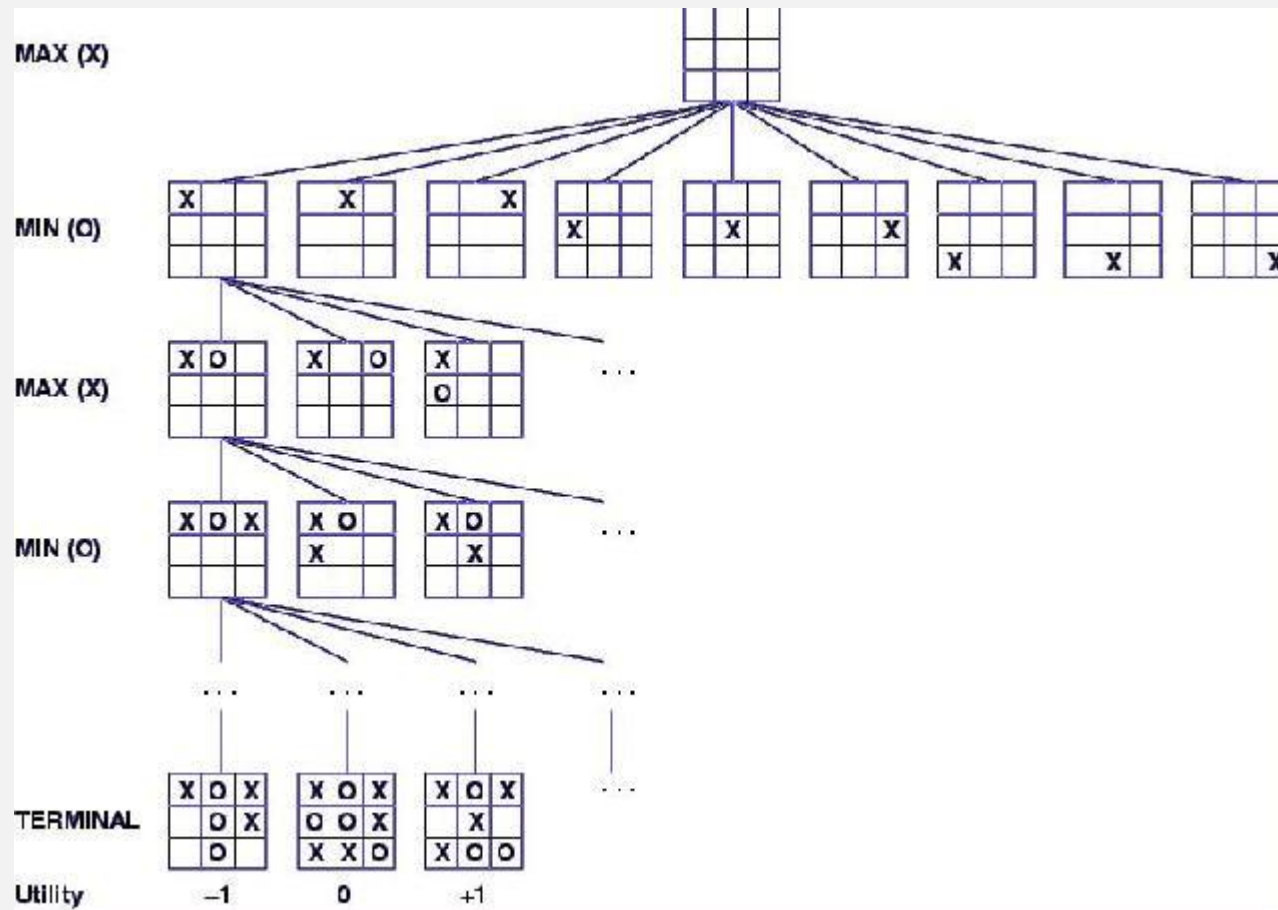
MINIMAX

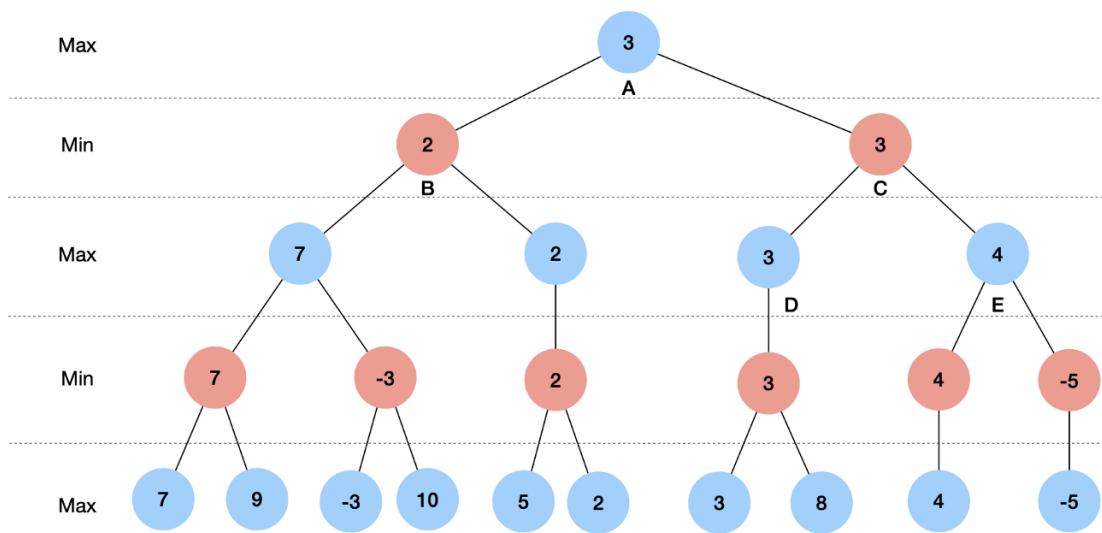
- Given this new encoding of board states, we know that each board actually has a value of 1, 0 or -1. This means that with **perfect play**, this is the worst possible outcome for the current player.
- So a score of 1, means that if crosses plays perfectly, they will win the game.
- This encoding systems helps us formalise what we want to achieve.
- When searching, if we are crosses, we want to **maximise** the value of the board.
- If we are noughts, we want to **minimise** the value of the board.

BOARD VALUE

- We know that certain positions are either 1, 0 or -1 because the game is over. (Three in a row has been achieved, or no more moves can be made). For these types of positions, we say that the game is in a **terminal state**.
- For non-terminal states, how can we calculate the score?
- The answer is an algorithm called minimax. Minimax is an example of a depth first search (DFS).

MINIMAX





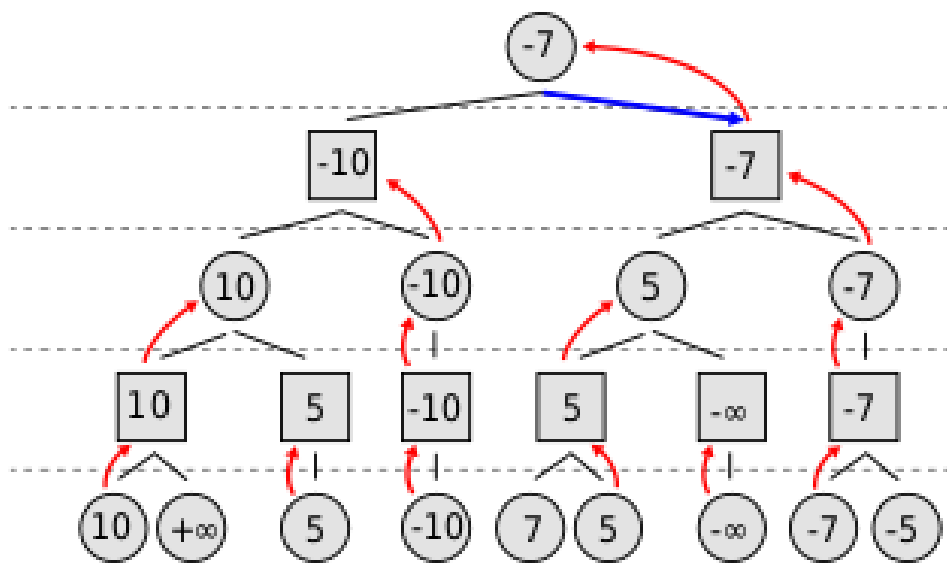
0 (max)

1 (min)

2 (max)

3 (min)

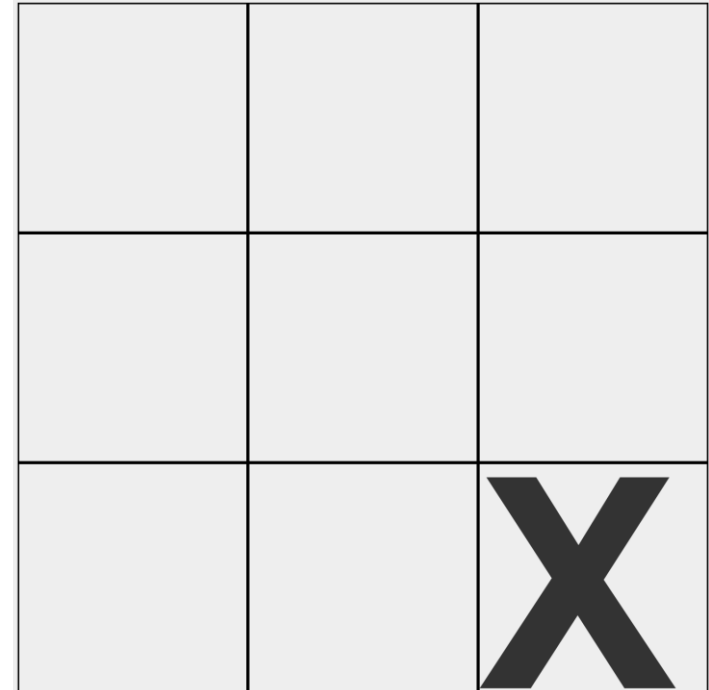
4 (max)



MINIMAX

MINIMAX

- The idea is we have two players. One agent is trying to **maximise** the score (crosses). Noughts is trying to **minimise** the score.
- From the tree we created, we can see how each agent in turn chooses which move to make. They do this by choosing the best move available to them at the time.
- The key idea is that for this current state, we can't actually know how the game will end and thus can't give the board a score.
- We can only score states where three in a row has been reached or the game has ended
- Thus we construct a tree of all possible moves from the current board and all possible moves from that.
- We only score terminal states, aka. Leaf nodes
- We then apply the minimax algorithm to calculate the score of the board



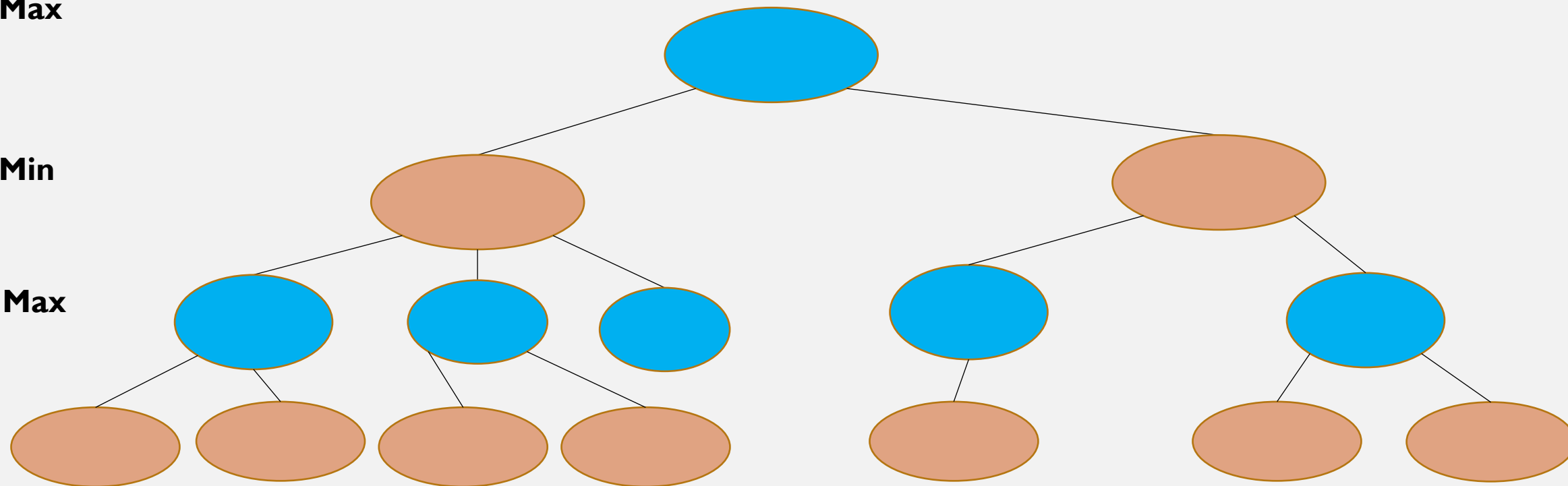
MINIMAX

Max

Min

Max

Min



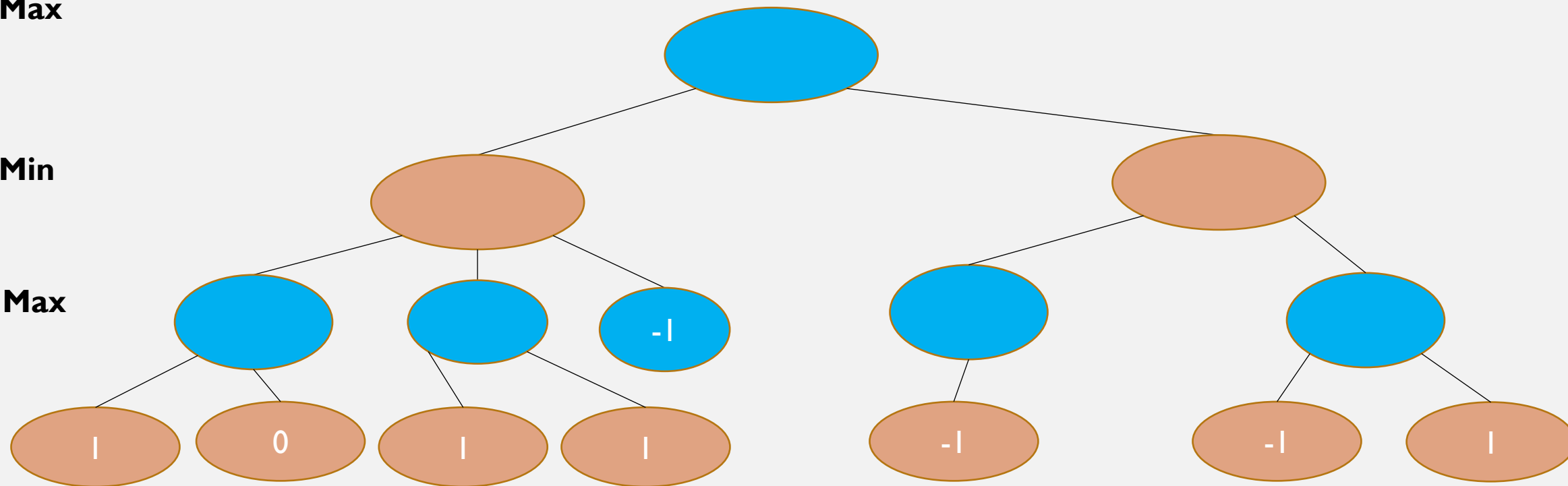
MINIMAX

Max

Min

Max

Min



Evaluate nodes at terminal state

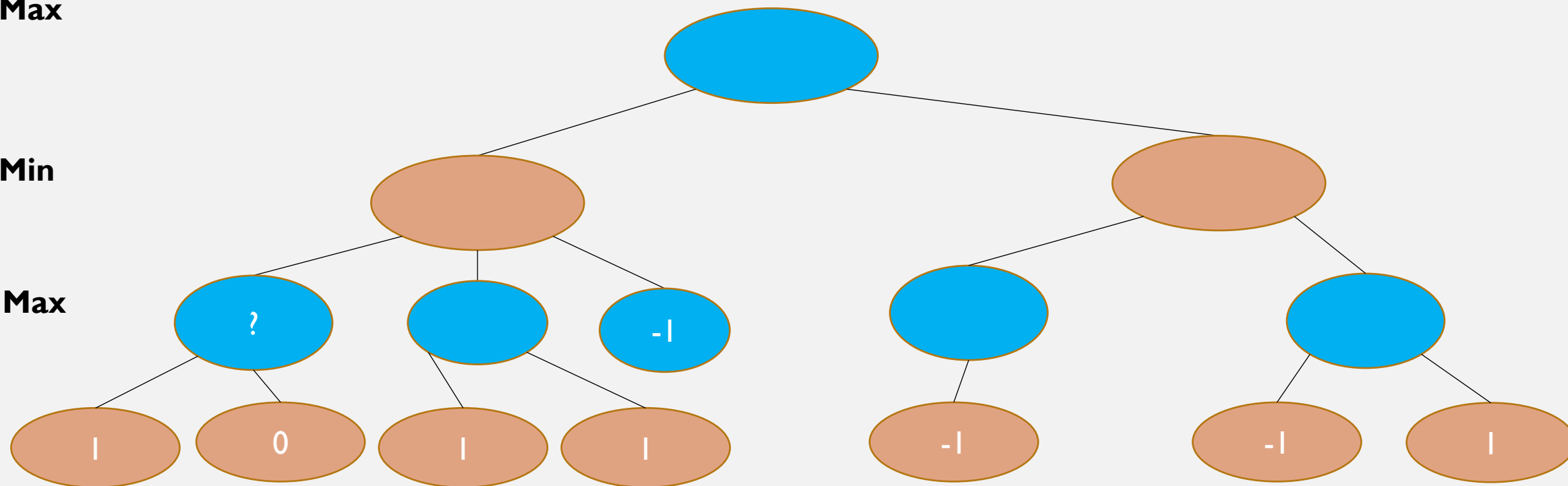
MINIMAX

Max

Min

Max

Min



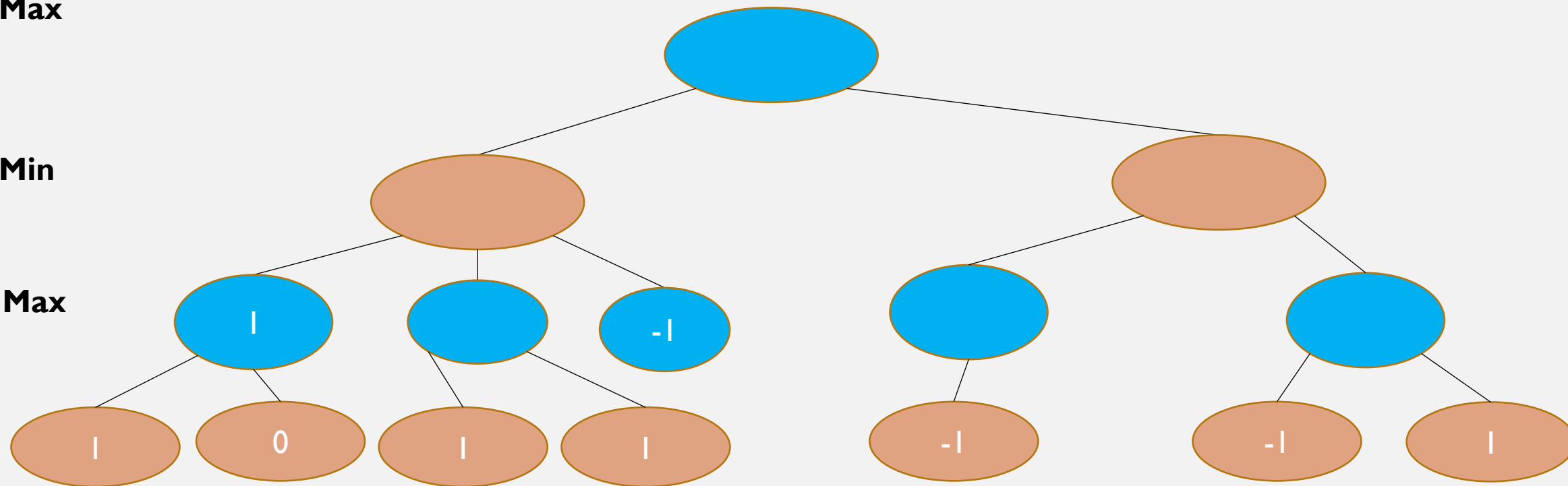
MINIMAX

Max

Min

Max

Min



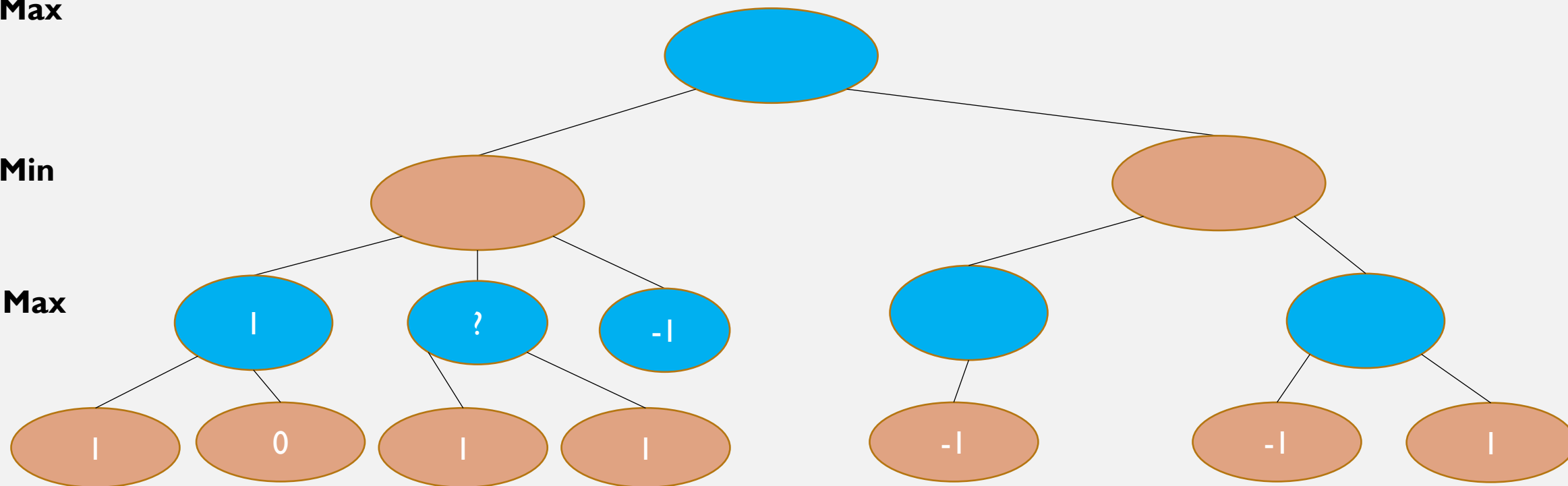
MINIMAX

Max

Min

Max

Min



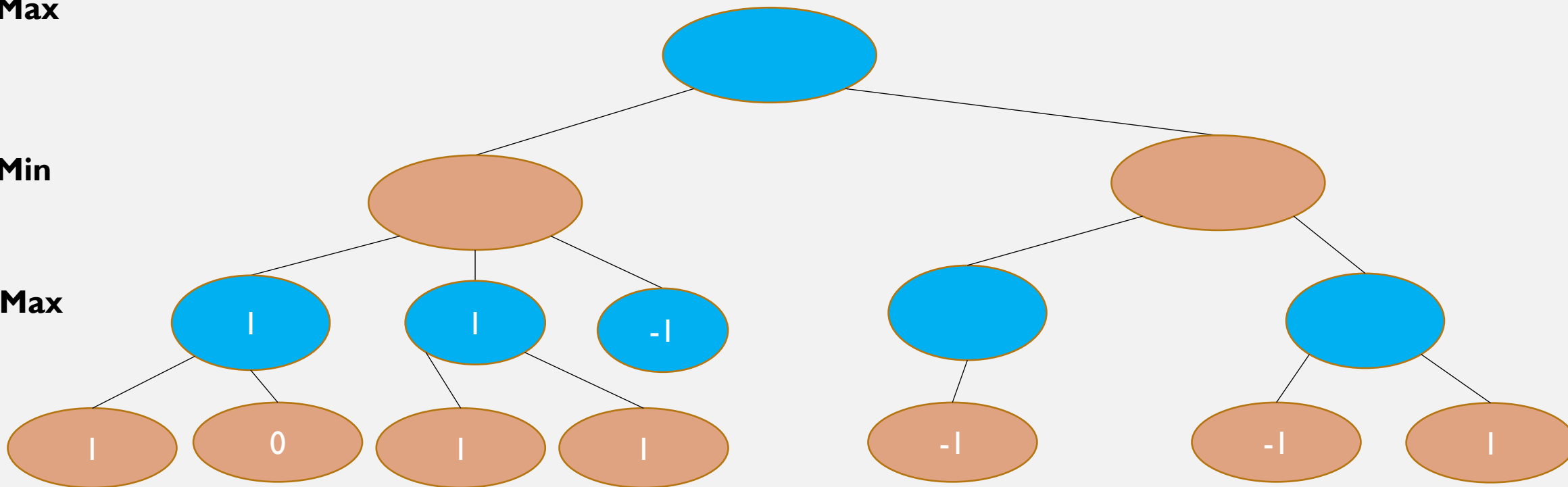
MINIMAX

Max

Min

Max

Min



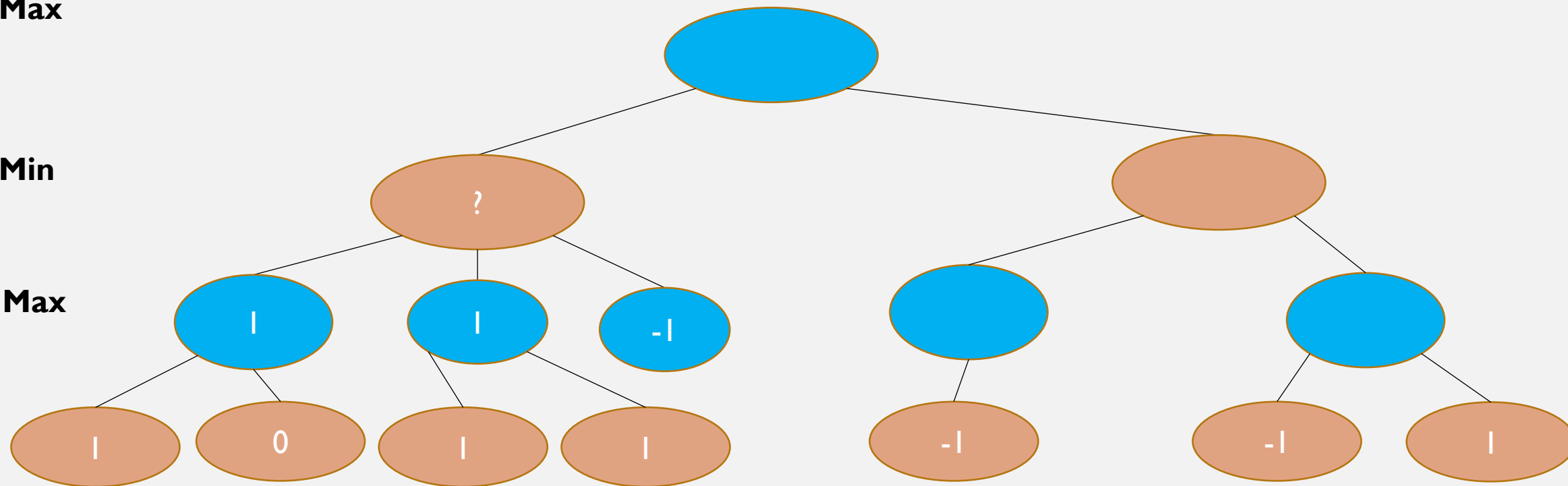
MINIMAX

Max

Min

Max

Min



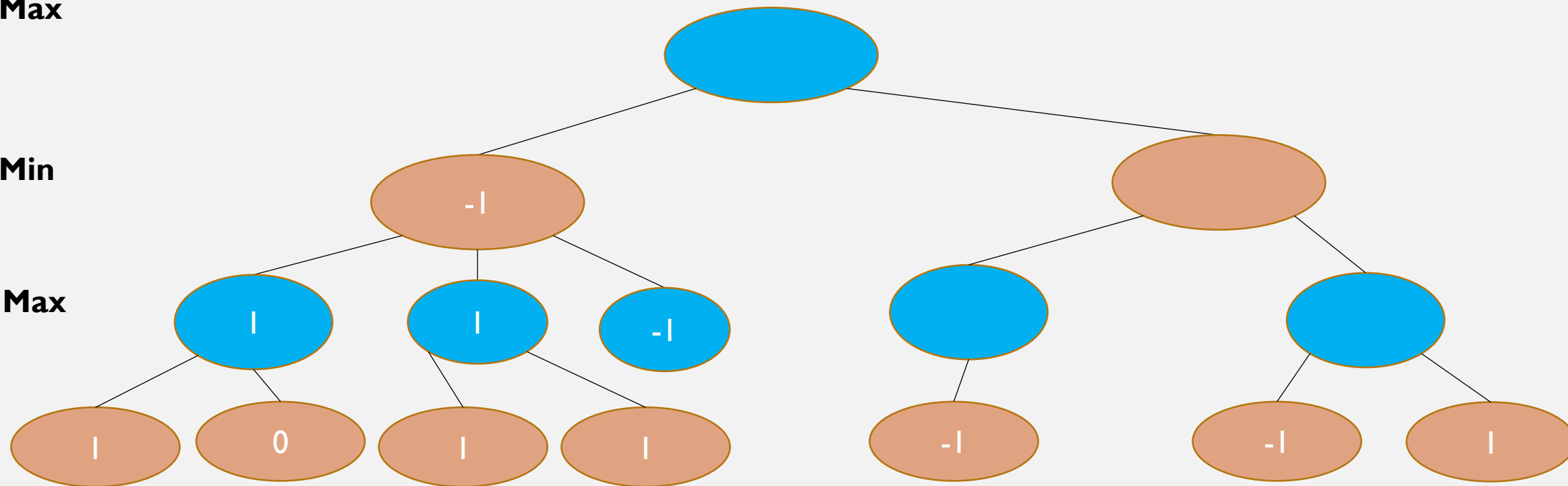
MINIMAX

Max

Min

Max

Min



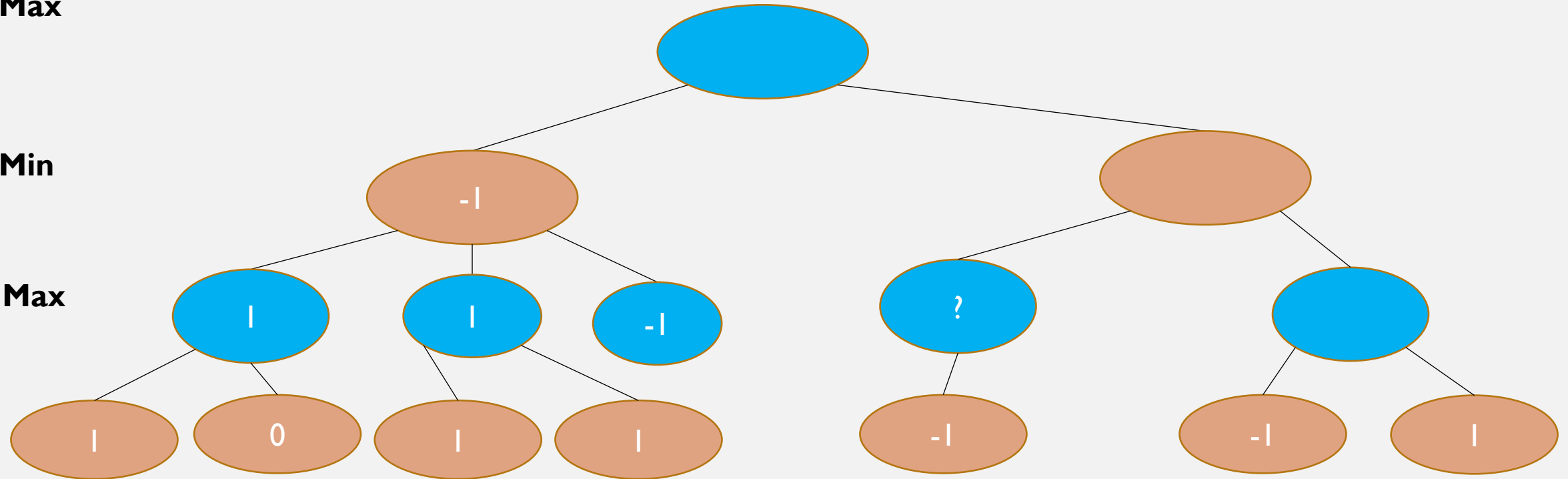
MINIMAX

Max

Min

Max

Min



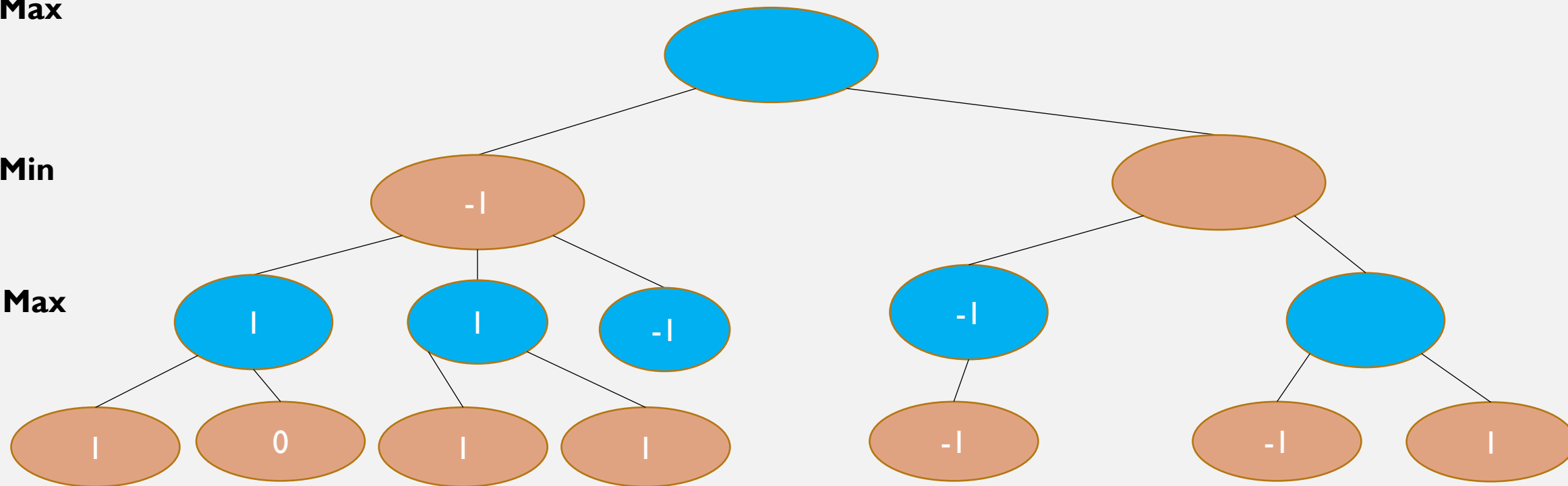
MINIMAX

Max

Min

Max

Min



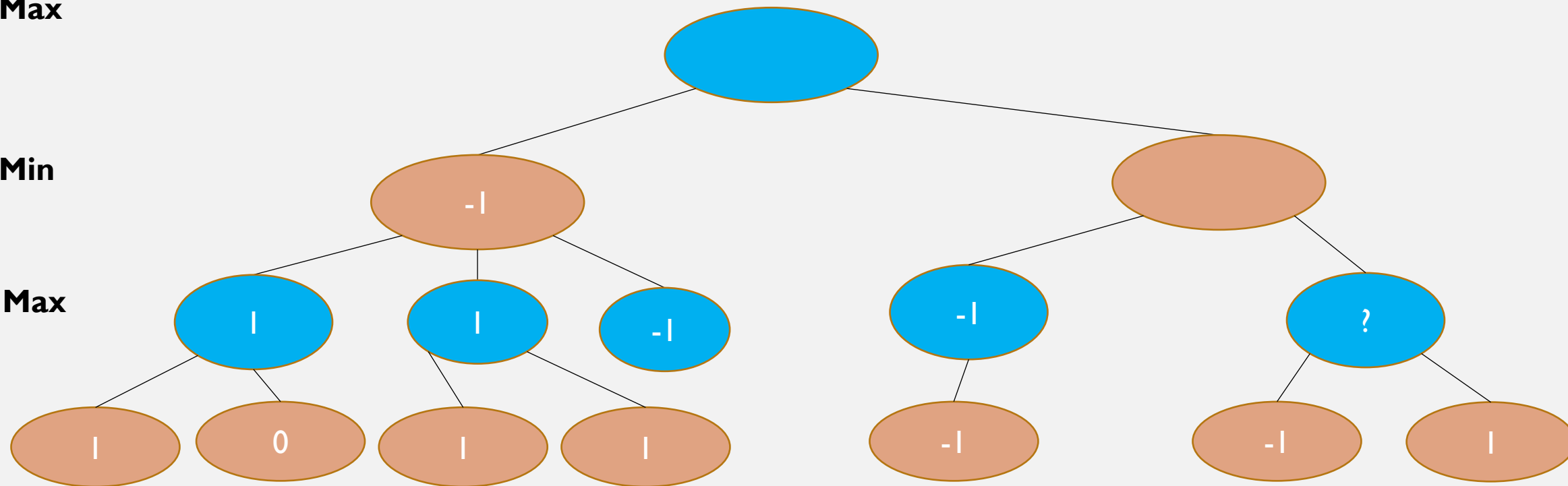
MINIMAX

Max

Min

Max

Min



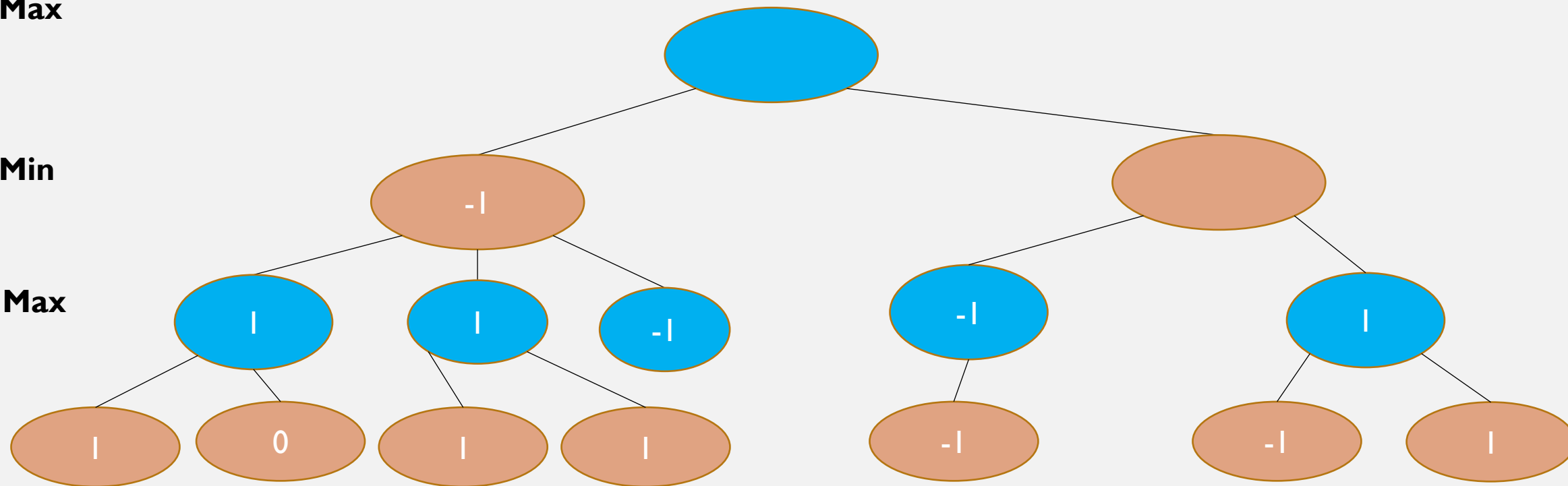
MINIMAX

Max

Min

Max

Min



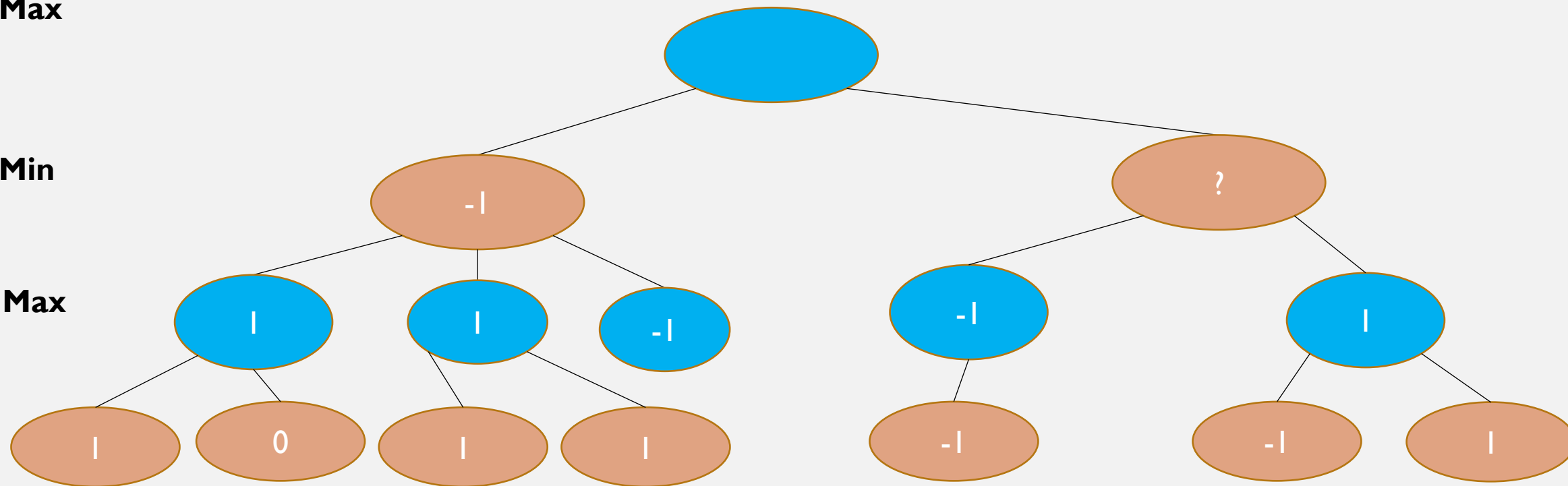
MINIMAX

Max

Min

Max

Min



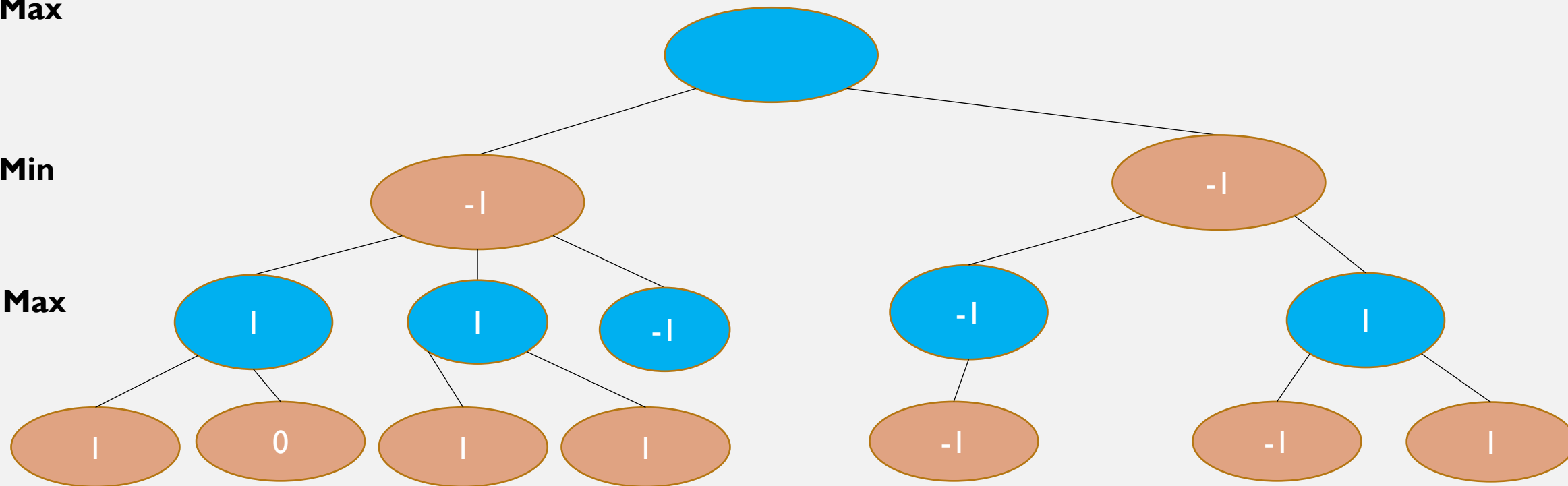
MINIMAX

Max

Min

Max

Min



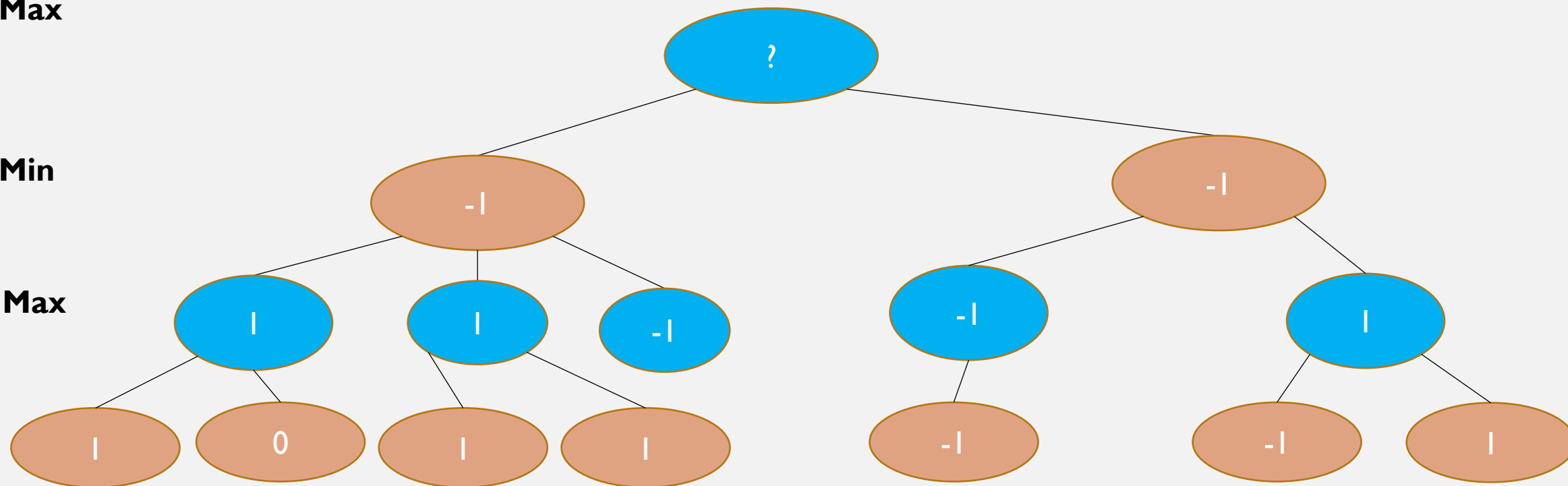
MINIMAX

Max

Min

Max

Min



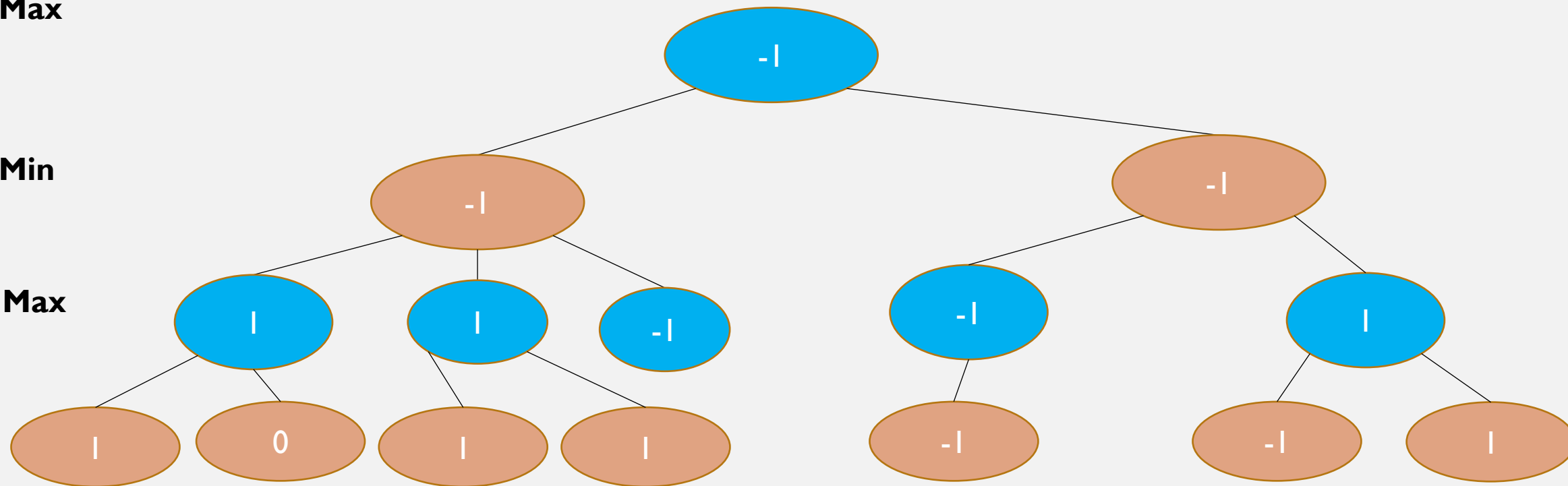
MINIMAX

Max

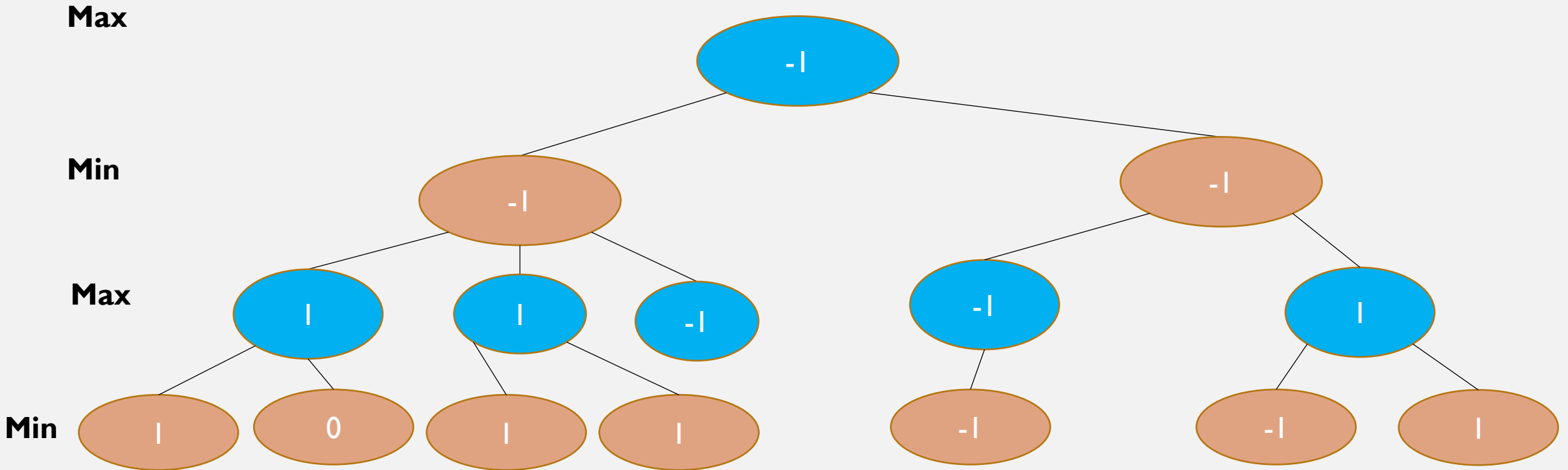
Min

Max

Min

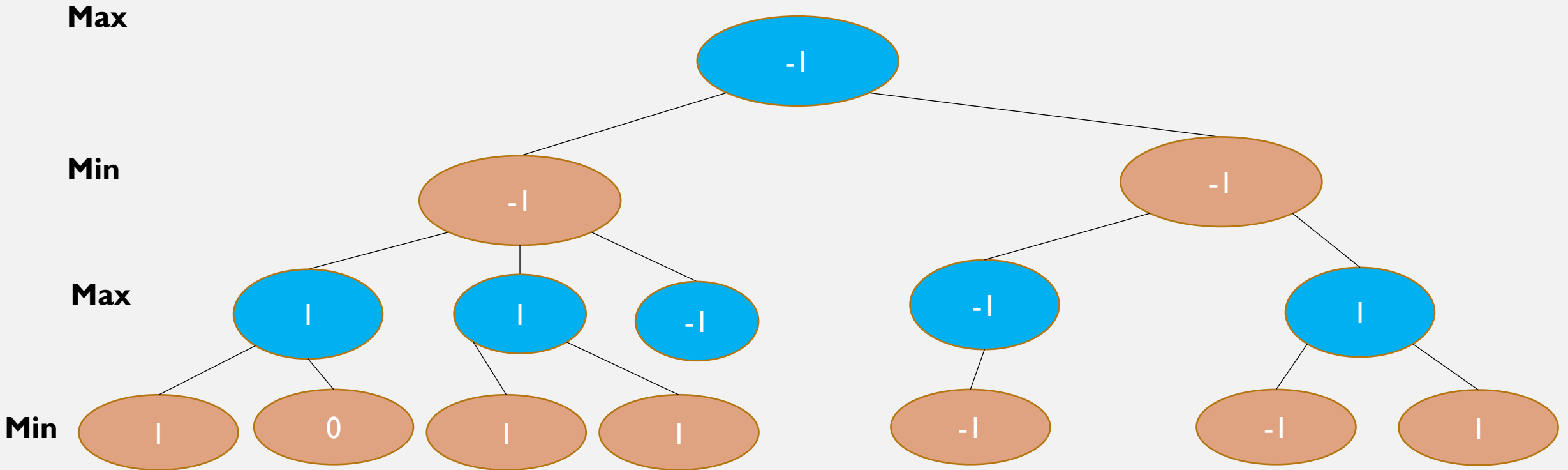


MINIMAX



- Thus, we can now say that the value of the current board state is **-|**
- This means that, with **perfect** play, crosses is guaranteed to lose. (I.e.), noughts are guaranteed to win.

MINIMAX



- Notice, how we only evaluate nodes at the bottom of the tree initially. This makes minimax a **depth-first search**.
- However, notice that we explore that entire tree, thus guaranteeing us optimality. This also makes it an instance of **breadth-first search**.

```
int boardValue(Board board){
    if(crossWin(board)) return 1;
    else if(draw(board)) return 0;
    else return -1;
}
```

MINIMAX

```
int boardValueMaxPlayer(Board board){
    if (isTerminalState(board)){
        return boardValue(board);
    }
    int bestValue = Integer.MIN_VALUE;
    for(Move move : allAvailableMoves(board)){
        Board boardAfterMove = makeMove(board, move);
        int value = boardValueMinPlayer(boardAfterMove);
        bestValue = Math.max(value, bestValue);
    }
    return bestValue;
}
```

```
int boardValueMinPlayer(Board board){
    if (isTerminalState(board)){
        return boardValue(board);
    }
    int bestValue = Integer.MAX_VALUE;
    for(Move move : allAvailableMoves(board)){
        Board boardAfterMove = makeMove(board, move);
        int value = boardValueMaxPlayer(boardAfterMove);
        bestValue = Math.min(value, bestValue);
    }
    return bestValue;
}
```

```
Move getBestMove(Board board){
    Move bestMove = null;
    int bestValue = isCrossTurn(board)? Integer.MIN_VALUE : Integer.MAX_VALUE;
    int currVal = 0;
    for(Move move: allAvailableMoves(board)){
        if(isCrossTurn(board)){
            currVal = boardValueMinPlayer(makeMove(board, move));
            if(currVal >= bestValue){
                bestValue = currVal;
                bestMove = move;
            }
        }else{
            currVal = boardValueMax(makeMove(board, move));
            if(currVal <= bestValue){
                bestValue = currVal;
                bestMove = move;
            }
        }
    }
    return bestMove;
}
```

MINIMAX

- This algorithm explores all of the possible game states that could occur after the current one
- This guarantees that the best possible move is found
- Using this, we have now created a perfect tic tac toe playing AI. It will never lose at the game!
- We've now completed our design



DESIGN LIMITATIONS

- This works great for noughts and crosses
- Each level of the tree is exponentially larger than the previous level. Tic Tac toe has a tree depth limit of 9. This means that our computer (which can execute billions of instructions per second) can handle this.
- Tic Tac Toe has 255,168 possible games and thus is very easy to compute. Chess on the other hand is believed to have more legal possible games than particles in the Universe. There are believed to be 10^{120} games of chess that are possible.

ALPHA-BETA PRUNING

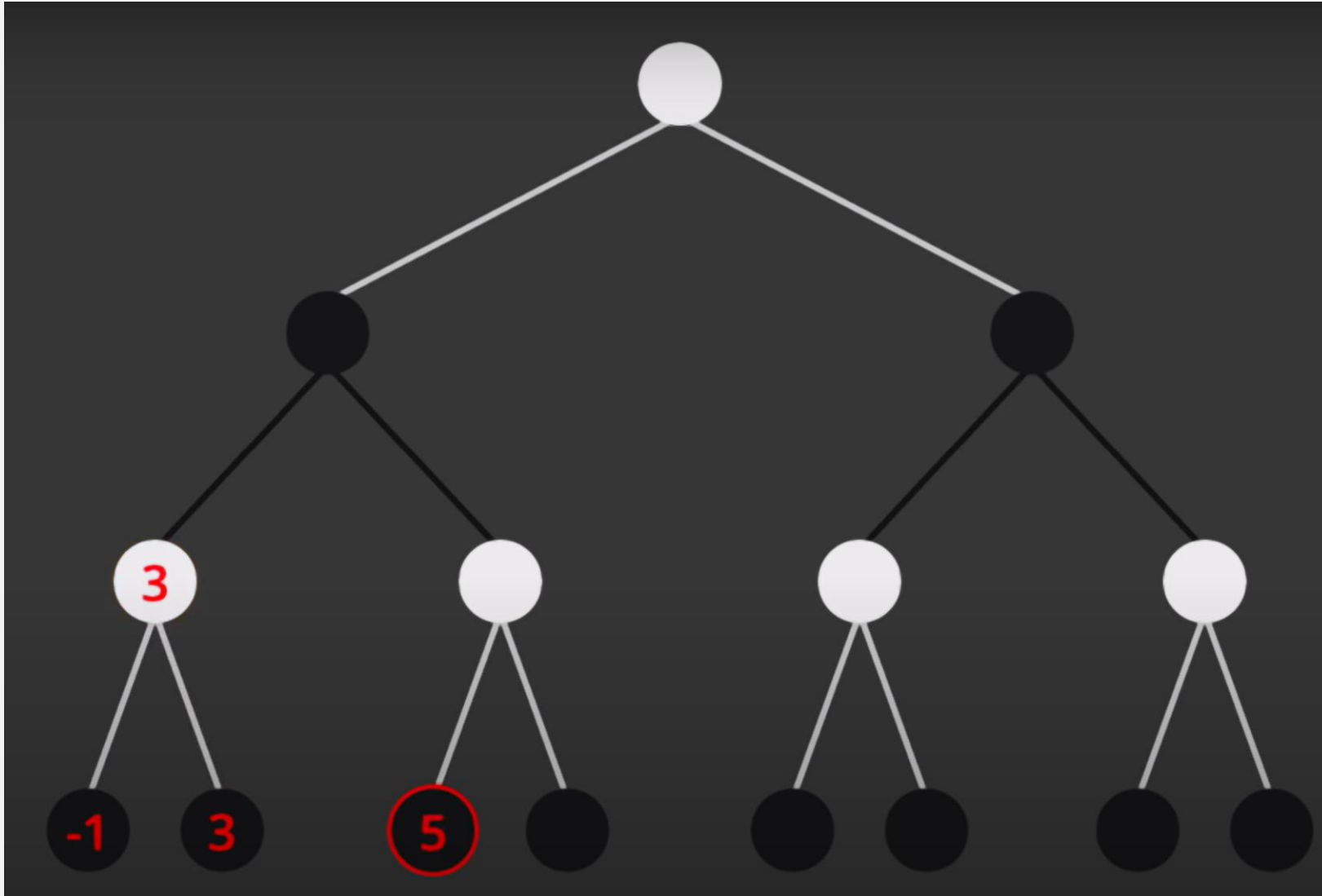
Credit: Sebastian Lague
<https://youtu.be/l-hh5IncgDI>

Max

Min

Max

Min



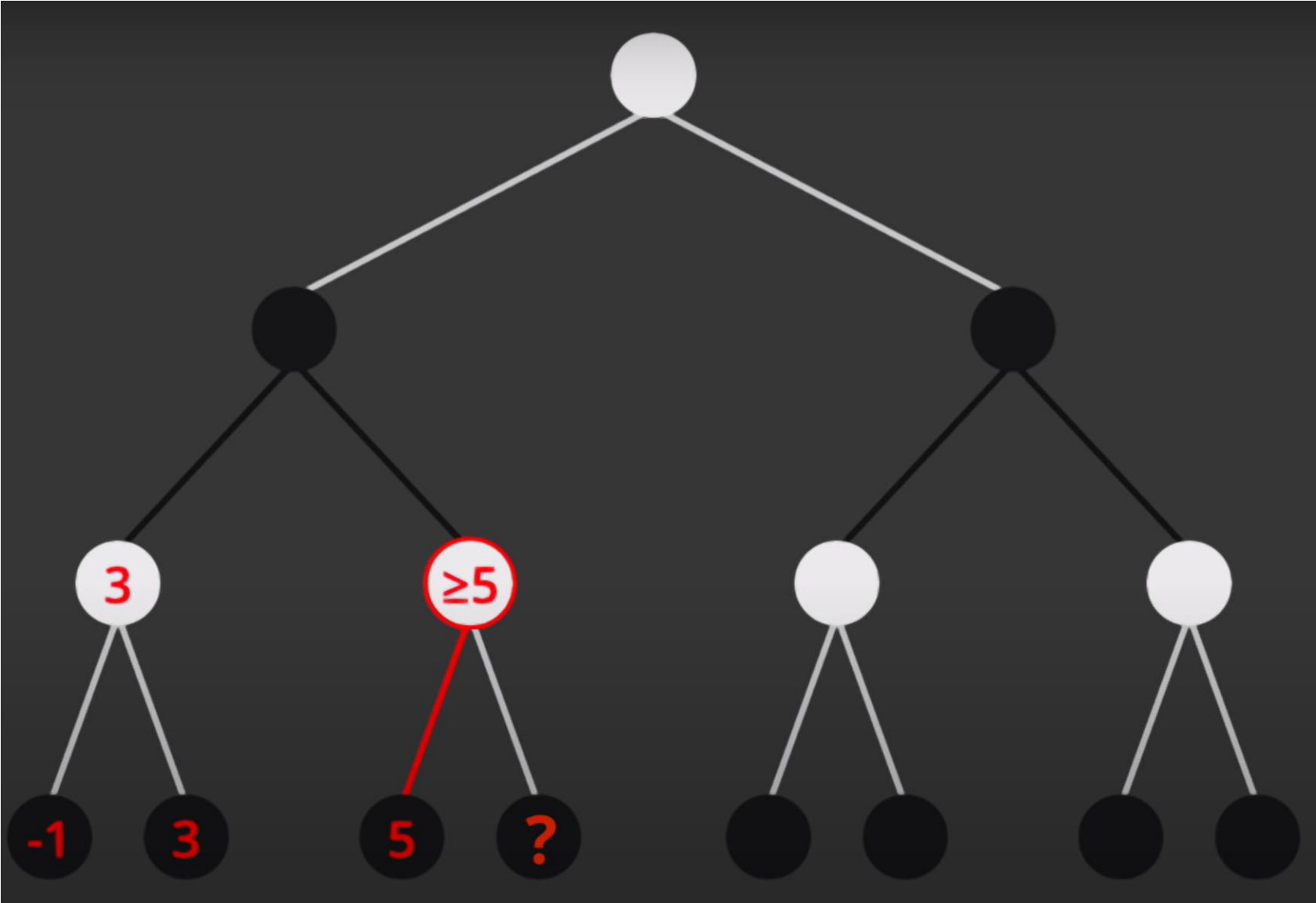
ALPHA-BETA PRUNING

Max

Min

Max

Min



ALPHA-BETA PRUNING

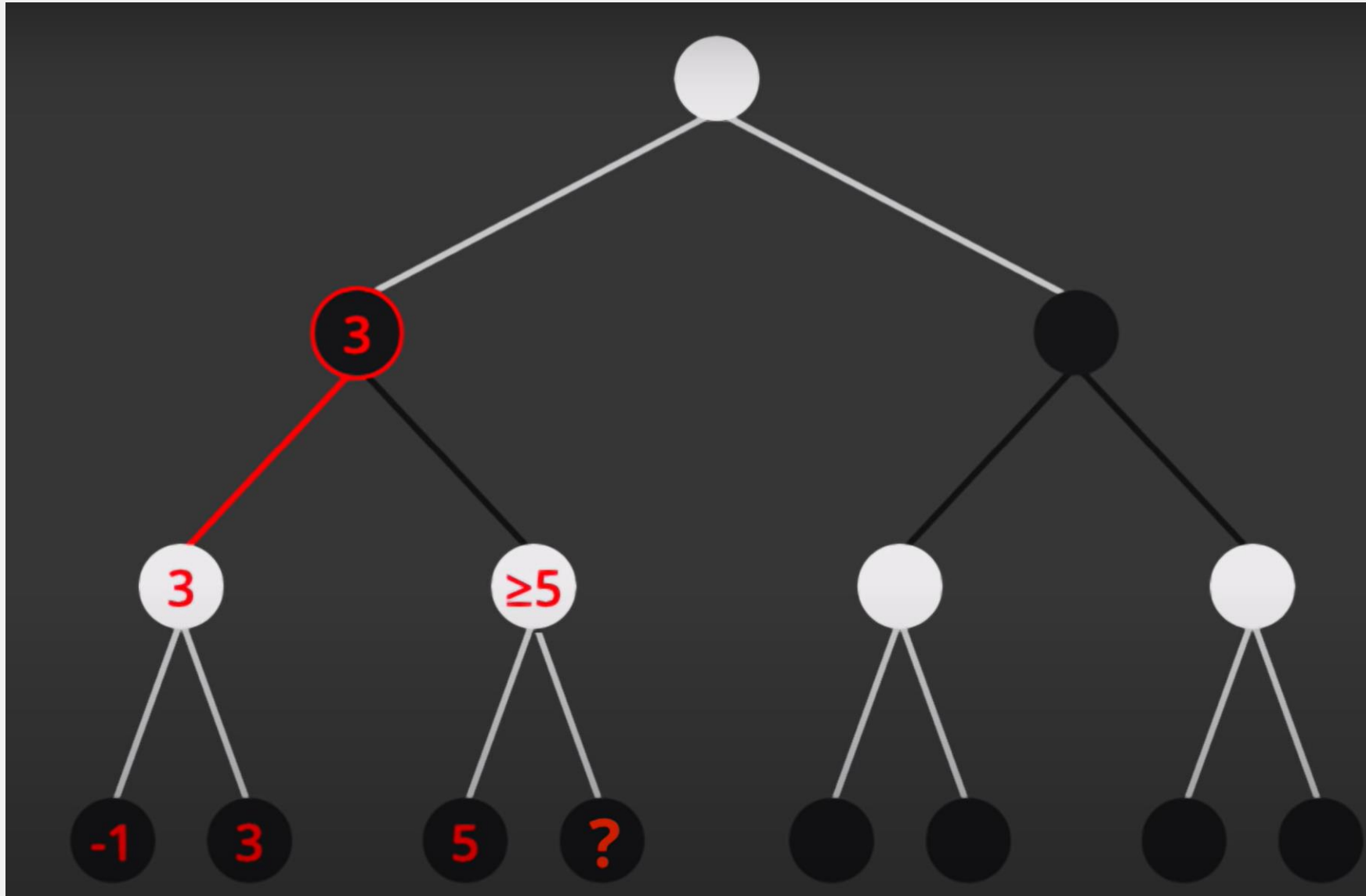
Credit: Sebastian Lague
<https://youtu.be/l-hh5IncgDI>

Max

Min

Max

Min



ALPHA-BETA PRUNING

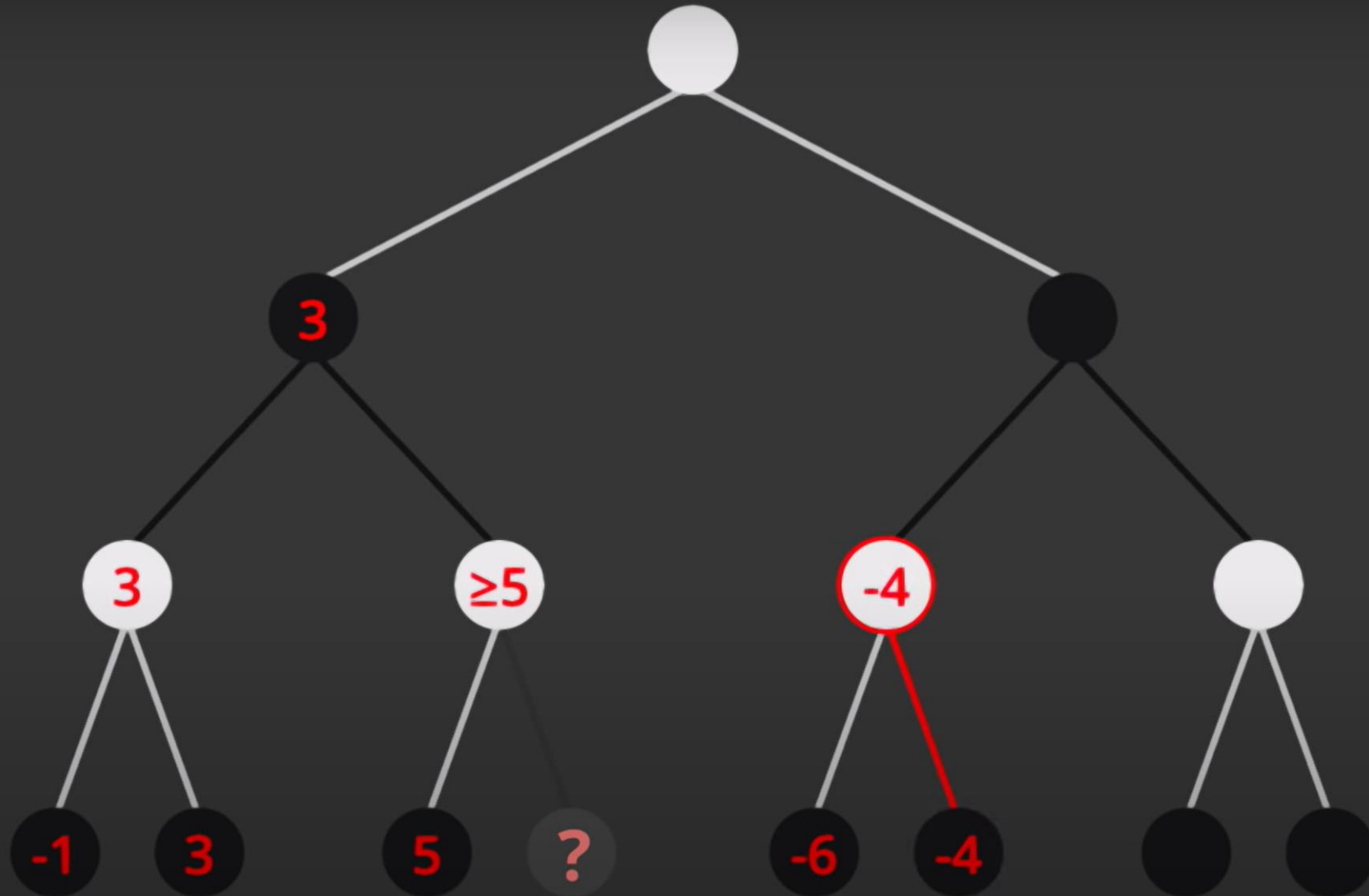
Credit: Sebastian Lague
<https://youtu.be/l-hh5IncgDI>

Max

Min

Max

Min



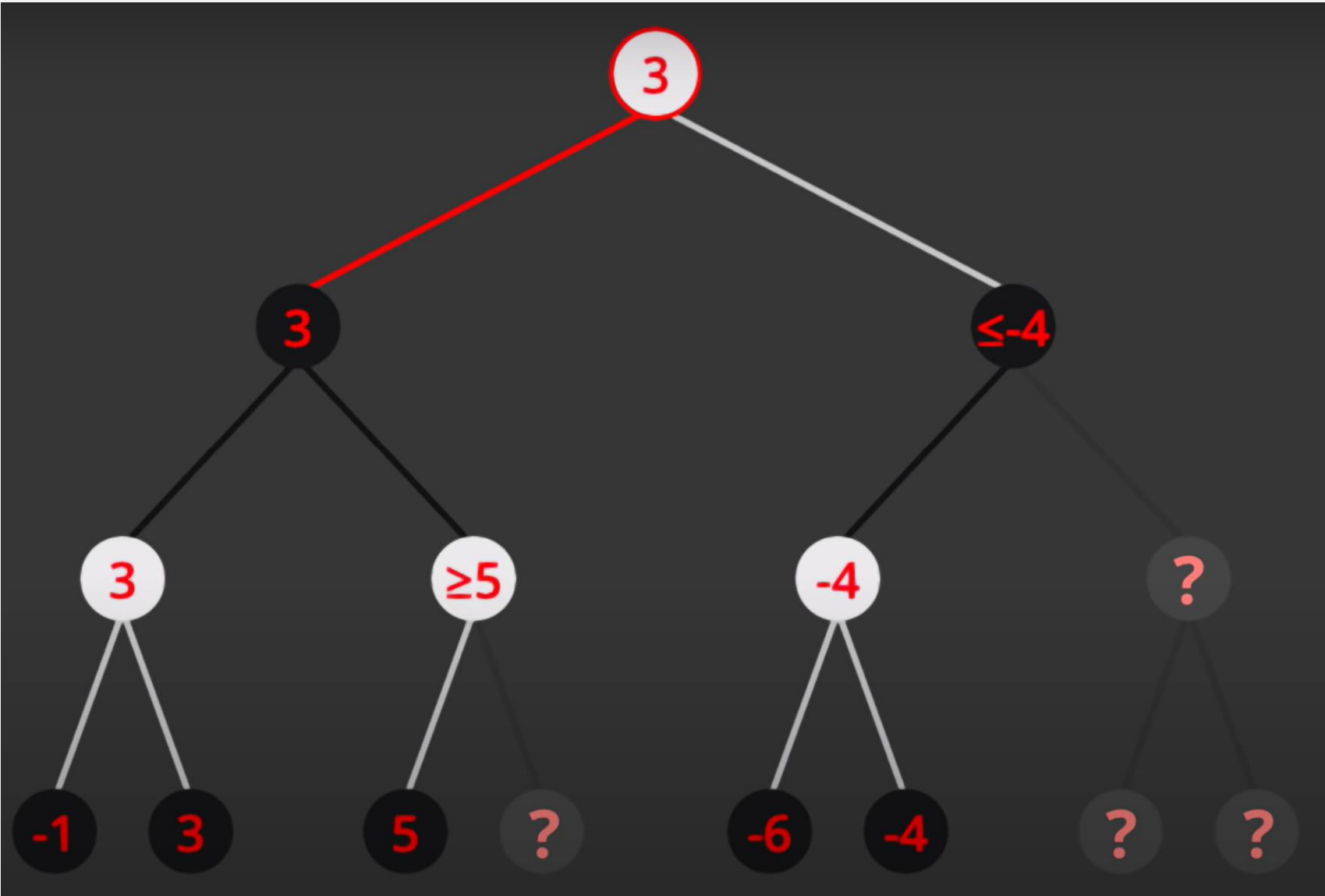
ALPHA-BETA PRUNING

Max

Min

Max

Min

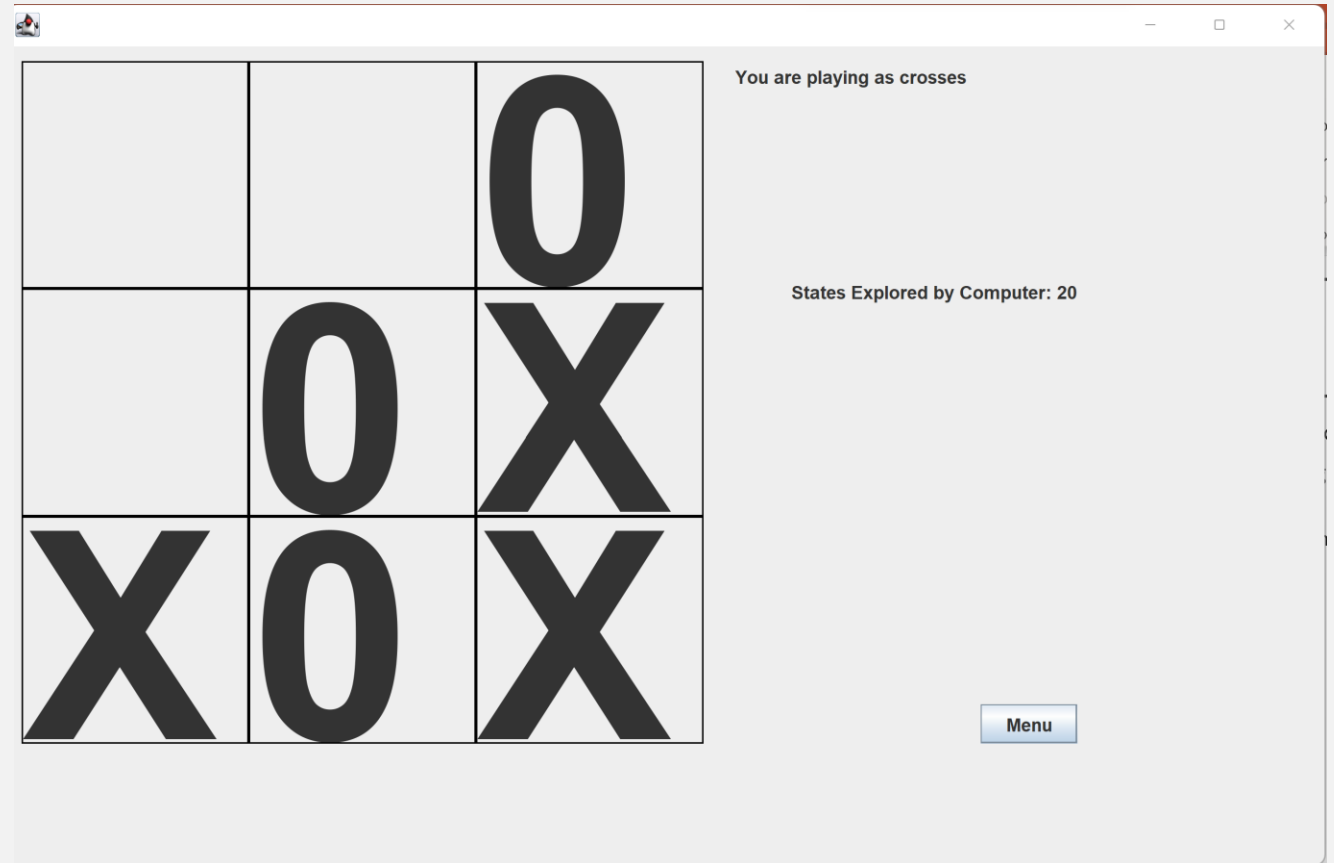


ALPHA-BETA PRUNING

- This gives us the exact same result as using Minimax.
- In the worst case, it is as slow (but no slower than Minimax). The algorithm is said to be $O(b^m)$, where b is the branching factor (number of legal moves at available at each point) and m is the maximum tree depth possible.
- But, it can run in $O(b^{\frac{m}{2}})$ using Alpha-beta pruning in the best case scenario.
- This holds when the best moves are evaluated first. This means that better algorithms will look at moves that are more likely to be good first.
- Even if we look at moves in a random order, we will be able to prune a good number of leaves / branches and massively speed up our algorithm
- For example, in Tic Tac Toe, a minimax algorithm may look at 550,000 possible moves. AB pruning only needs to look at 28,000 moves!

TIC TAC TOE

- I implemented minimax and minimax + alpha-beta pruning in Tic Tac Toe.
- Code walkthrough! – To view source code, follow the following link:
<https://github.com/philipmortimer/AI-Course/tree/main/Programs/Part%202/NoughtsAndCrossesDemo>
- Let's fix any technical issues + look through source code together.
- Let's play tic tac toe!



STATES EXPLORED (ROUGH)

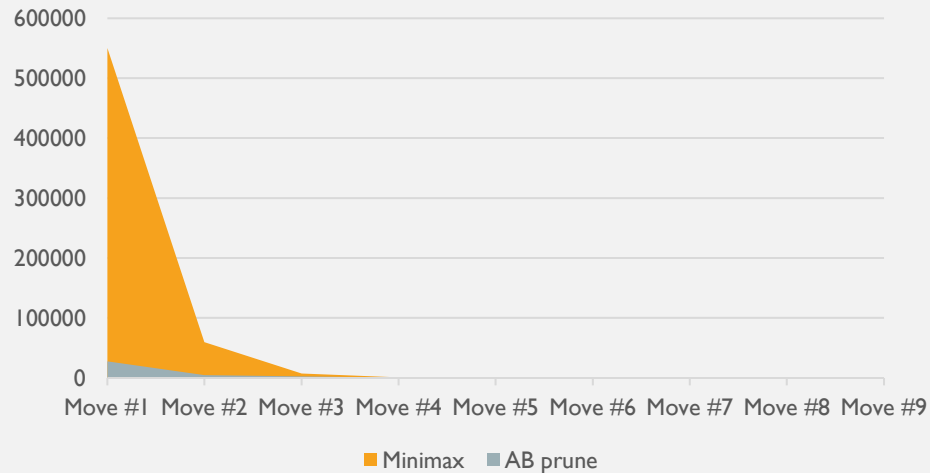
	Move #1	Move #2	Move #3	Move #4	Move #5	Move #6	Move #7	Move #8	Move #9
Minimax	549,945	59,704	7,331	934	197	46	13	4	1
AB prune	27,565	4,676	2,346	525	96	43	13	4	1

Note, that this AB number could be significantly smaller (my implementation is not particularly efficient).

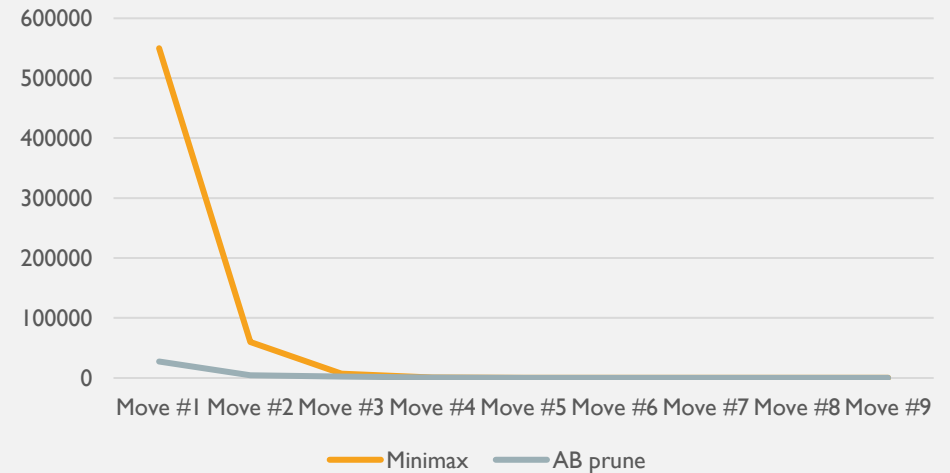
STATES EXPLORED (ROUGH)

	Move #1	Move #2	Move #3	Move #4	Move #5	Move #6	Move #7	Move #8	Move #9
Minimax	549,945	59,704	7,331	934	197	46	13	4	1
AB prune	27,565	4,676	2,346	525	96	43	13	4	1

States Explored

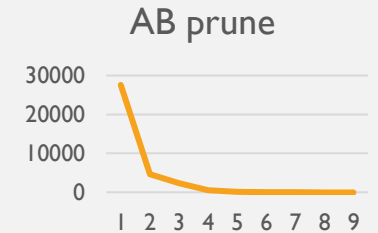


States Explored



DEPTH LIMITED MINIMAX

- Even with Alpha-Beta pruning, the number of states grows exponentially. This means that games that have more than 9 possible moves quickly become impossible to solve.
- Thus we typically limit how many moves ahead into the future our system is allowed to search. For example, in chess, we might say that we are only allowed to look a maximum of 7 seven moves into the future. A computer may run the AB calculation in a fraction of a second. But if we look 12 moves into the future, we may wait **years** for the result.
- For tic tac toe, we said we would only evaluate board states that are terminal. If we limit our search depth, we can't guarantee that our board state is terminal.
- Thus, we need a **heuristic** to estimate which player is winning. For example, we may want to see which player currently has more pieces (in chess) to see whether white or black is winning. We use this heuristic value on the tree instead.



DEPTH LIMITED MINIMAX

- Top AI systems use a depth limited minimax search that implements Alpha-beta pruning. Implementation of a depth limit is fairly easy, we can simply amend our current pseudocode slightly.

```
int boardValueMaxPlayer(Board board, int depth){
    if (isTerminalState(board) || depth == 0){
        return boardValue(board);
    }
    int bestValue = Integer.MIN_VALUE;
    for(Move move : allAvailableMoves(board)){
        Board boardAfterMove = makeMove(board, move);
        int value = boardValueMinPlayer(boardAfterMove, depth - 1);
        bestValue = Math.max(value, bestValue);
    }
    return bestValue;
}
```

```
int boardValueMinPlayer(Board board, int depth){
    if (isTerminalState(board) || depth == 0){
        return boardValue(board);
    }
    int bestValue = Integer.MAX_VALUE;
    for(Move move : allAvailableMoves(board)){
        Board boardAfterMove = makeMove(board, move);
        int value = boardValueMaxPlayer(boardAfterMove, depth - 1);
        bestValue = Math.min(value, bestValue);
    }
    return bestValue;
}
```

ALPHA-BETA PRUNING

- Let's formalise our reasoning behind AB pruning into code.
- At every call in the tree, we will have values to keep track of: alpha and beta.
- Alpha represents the value that the maximising player is guaranteed to achieve for the current search level or above.
- Beta represents the value that the minimising player is guaranteed to achieve for the current level or above.
- If it's the maximisers turn, they update the alpha value if they discover a move that leads to a higher score than the current alpha.
- If it's the maximisers turn, and alpha exceeds or is equal to beta, pruning occurs. This is because the minimising player will never choose a branch higher than beta (as they are already guaranteed beta as a worst-case).
- Similar logic applies to the minimiser turn.

AB PRUNING

Credit: GeeksForGeeks
<https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/>

```
function minimax(node, depth, isMaximizingPlayer, alpha, beta):
```

```
    if node is a leaf node :  
        return value of the node
```

```
    if isMaximizingPlayer :  
        bestVal = -INFINITY  
        for each child node :  
            value = minimax(node, depth+1, false, alpha, beta)  
            bestVal = max( bestVal, value)  
            alpha = max( alpha, bestVal)  
            if beta <= alpha:  
                break  
        return bestVal
```

```
    else :  
        bestVal = +INFINITY  
        for each child node :  
            value = minimax(node, depth+1, true, alpha, beta)  
            bestVal = min( bestVal, value)  
            beta = min( beta, bestVal)  
            if beta <= alpha:  
                break  
        return bestVal
```

```
// Calling the function for the first time.  
minimax(0, 0, true, -INFINITY, +INFINITY)
```

MINIMAX

- We have now formalised an algorithm that can be applied to a whole range of adversarial search problems (including two player board games).
- We look at all possible moves up to a specific amount of time in the future. We then apply some heuristic that estimates the value of the board state at this point. We then use the Minimax algorithm (a depth-first search algorithm) to determine what the best move is. Alpha-beta pruning is used to search this space more efficiently.
- With small modifications, this can be applied to other domains outside of just two player games where both sides have all (or some) information accessible.

ITERATIVE DEEPENING

- The problem with depth-limited minimax is that we must manually select how deep our computer should search.
- Different computers will be more / less powerful and hence different search depths will be needed.
- Often, we actually want to limit our search by **time**
- We can adapt our minimax algorithm so that it only searches for a given period of time
- This is widely used in games like chess, where time limits are implemented
- To achieve this, we use an algorithm known as iterative deepening

ITERATIVE DEEPENING

- The idea of iterative deepening is simple. We run a depth-limited alpha-beta pruned minimax search at increasing depths until we run out of time.
- This means we search at depth 1, then 2, then 3 and so on until we've used up all available time
- In chess this translates to looking one move into the future then 2, then 3. When we run out of time, we return the result from the last complete search.
- Isn't this really inefficient? Aren't we just repeating work unnecessarily (e.g., we discard **all** search results other than the last completed one).
- It turns out that it has the same runtime complexity as minimax: $O(b^m)$
- This is because there are far more nodes on the last layer of the tree than at all the previous levels combined. Essentially, a search at depth 2 is trivially small compared to depth 3. Thus, if our computer is capable of searching to depth 3, depth 2 and depth 1 searches are negligible.

ITERATIVE DEEPENING

- Remember, the key to an efficient algorithm is move ordering. The best moves should be expanded first in order to maximise alpha-beta pruning efficiency
- Iterative deepening will give us the best moves at depth of 1. Often, this will be the best (or one of the best moves) at depth 2 as well. Hence, if we store the result from our previous depth search and expand this first, we are likely to cut-off a much larger portion of the search tree.
- Hence, iterative deepening is often a far more efficient (and faster) than minimax + alpha beta pruning. Essentially iterative deepening allows us to look further ahead into the future, for the same computation cost. This is why it is used in the top chess engines.

SUMMARY

- Minimax is an algorithm that allows us to perform an adversarial search
- Discussed how it can be applied to board games to achieve perfect / high-level play.
- Alpha-beta pruning is an optimisation that drastically speeds up the minimax algorithm whilst still producing the same result. It is fastest when moves are ordered from best to worst (although even a random ordering will lead to massive speedups)
- Discussed how depth-limits are needed to for most board games as search space is incredibly large.
- Heuristics are used to estimate a given board state at nodes at the specified depth-limit.
- Iterative deepening allows AI agents to search for a given period of time, and allows for deeper searches than using a fixed depth in the same amount of time.

QUESTIONS



FURTHER READING

- Minimax: <https://towardsdatascience.com/how-a-chess-playing-computer-thinks-about-its-next-move-8f028bd0e7b1>
- Minimax + AB-pruning + depth limit: <https://www.youtube.com/watch?v=I-hh5IncgDI>
<https://www.cs.cornell.edu/courses/cs312/2002sp/lectures/rec21.htm>
- Iterative deepening: https://www.chessprogramming.org/Iterative_Deepening
- Heuristics (evaluation function): <https://www.chessprogramming.org/Evaluation>