Minimax Tutorial

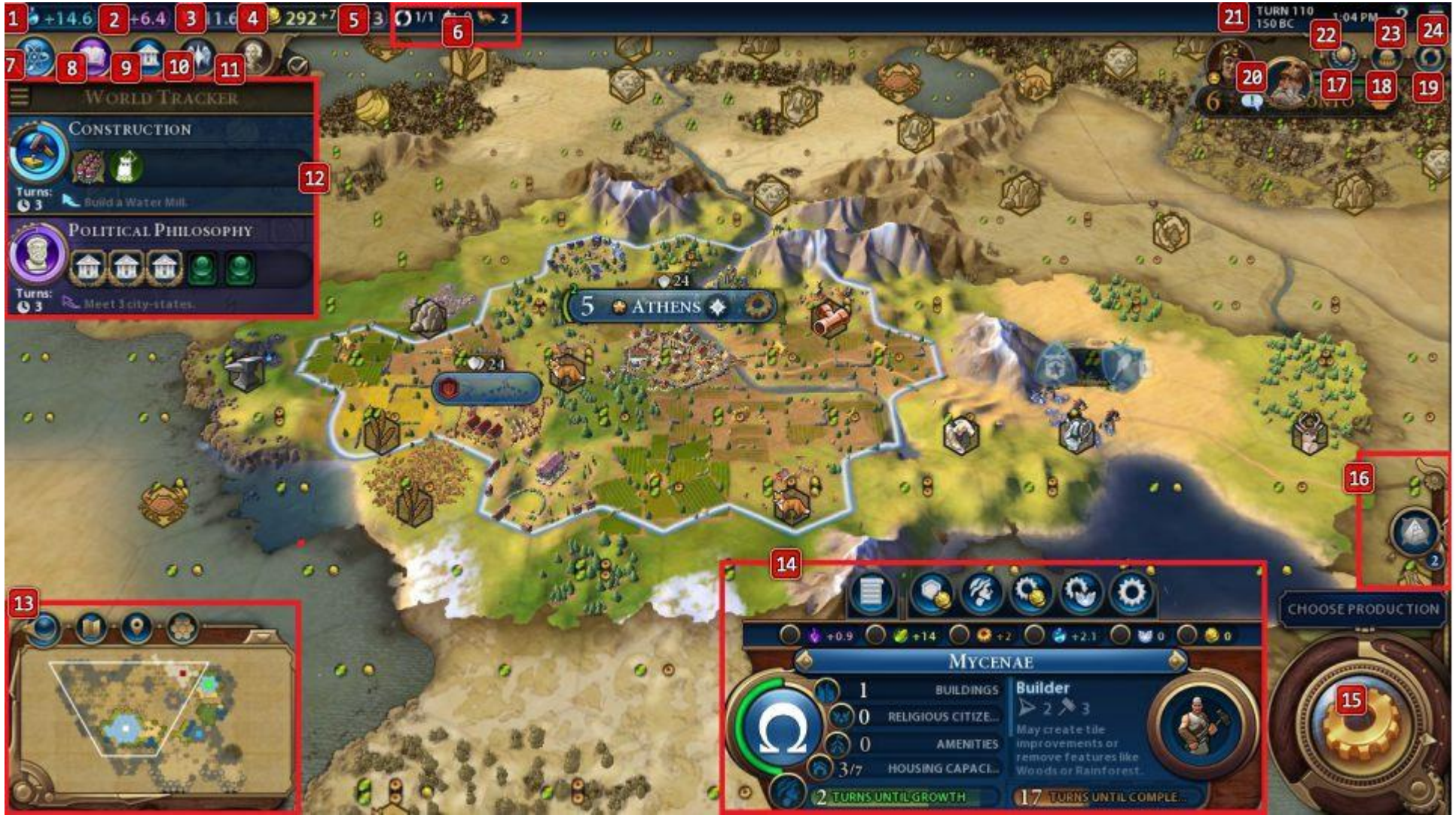
Civilization VI edition

Why Civilization VI (Civ VI)?

- It is a turn-based strategy game
- There can be multiple players
- Each player can make multiple moves every turn
- There are multiple victory conditions
- The game is too big to search all the way to any terminal

→ Perfect for our purpose

Civ VI interface example



Some notation we already know

* Before speeding things up*

S0 : The initial state

Player(s): Whose turn it is at state s

Actions(s): All available (legal) moves at state s

Result(s, a): Transition model (What state results from a)

Terminal-Test (s): Is the game over?

Utility(s, p): Score of player p in (terminal) state s

Some notation we already know

* After speeding things up *

Depth d: Keep track of how deep in the tree we are

Cutoff-test: A function to determine if we cut off search

Evaluation function Eval: A method of evaluating non-terminal states

New notation for this version of minimax

Recall: A player goes multiple times in a row

→Int Move-Count: Keep track of how many moves a player has made in one turn

→Get-Move-Count(s, p): Get the number of allowed moves player p can make this turn

→Agent-Index: The index of current agent, used to keep track of whose agent's turn it currently is.

Some assumptions

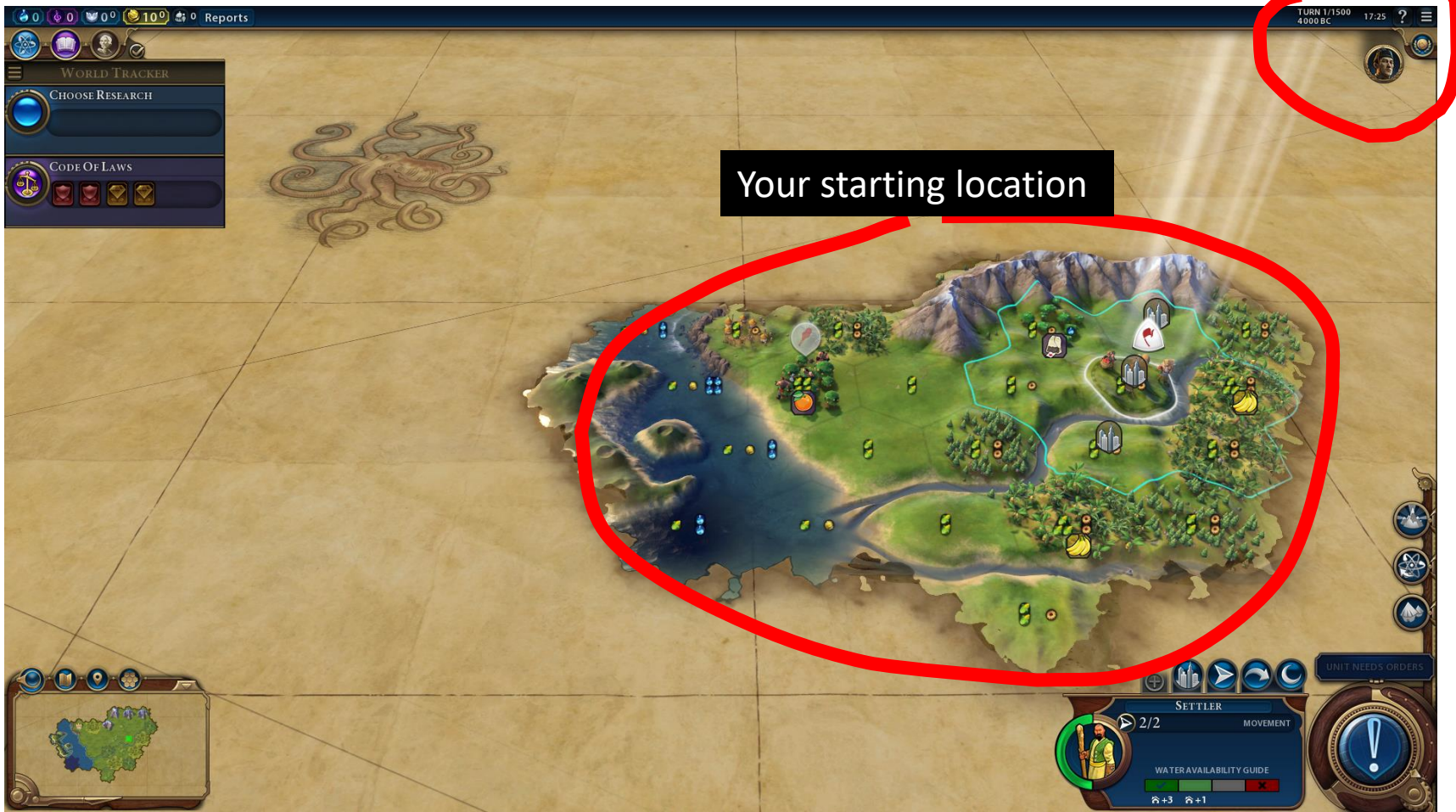
The number of moves every player can make at each state is known (either as global variables or embedded in the state s)

Each player will use all their allowed number of moves (like how tic-tac-toe players don’t “skip” their turn)

There is no cooperation among players. In other words, it is you versus the world (all other players’ goal is to minimize your utility)

Tying abstract to concrete

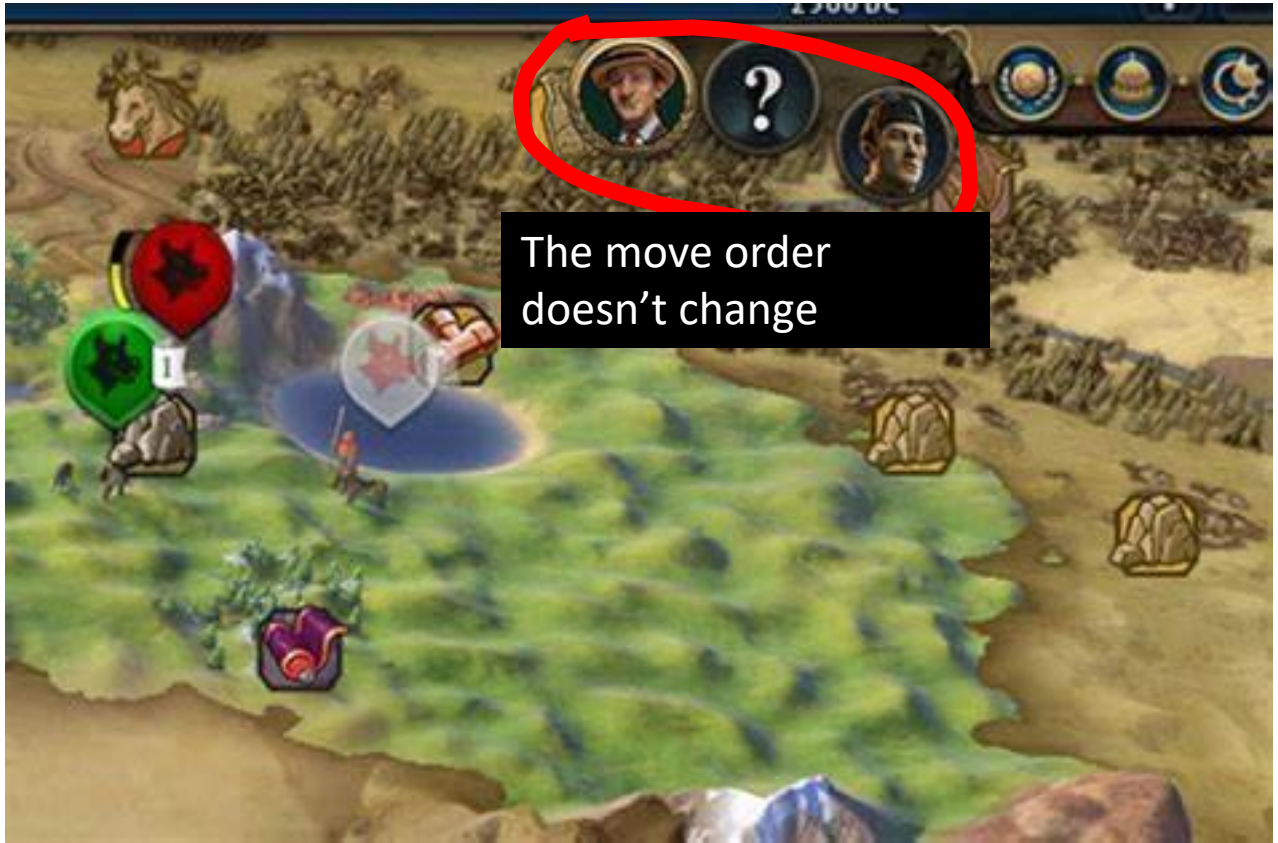
S0: World map with randomly generated resource locations.



1500 turns! That's how you know we can't explore all the way to any terminal

Tying abstract to concrete

Player(s): Initialized at the beginning of the game, sequential (after you use all your moves, it's the next player's turn)



Tying abstract to concrete

Actions(s): For the sake of this tutorial, let’s just go with the fact that there are many actions a player can take, some of them are not available at a given state

Result(s, a):

+ Citizens move to another spot
+ You built a ship
+ Finished a research (Animal Husbandry)

Tying abstract to concrete

Terminal-Test(s):

- You win if:
  + Domination Victory: You defeat every other civilization on the map
  + Science Victory: Colonize Mars
  + Religious Victory: Be the predominate religion (>= 50%)
  + Culture Victory: Have the most visiting tourists among all civilizations
  + Score Victory: Last until the end and generate as many points as possible
- You lose if:
  + Any other player achieves one of the above before you.

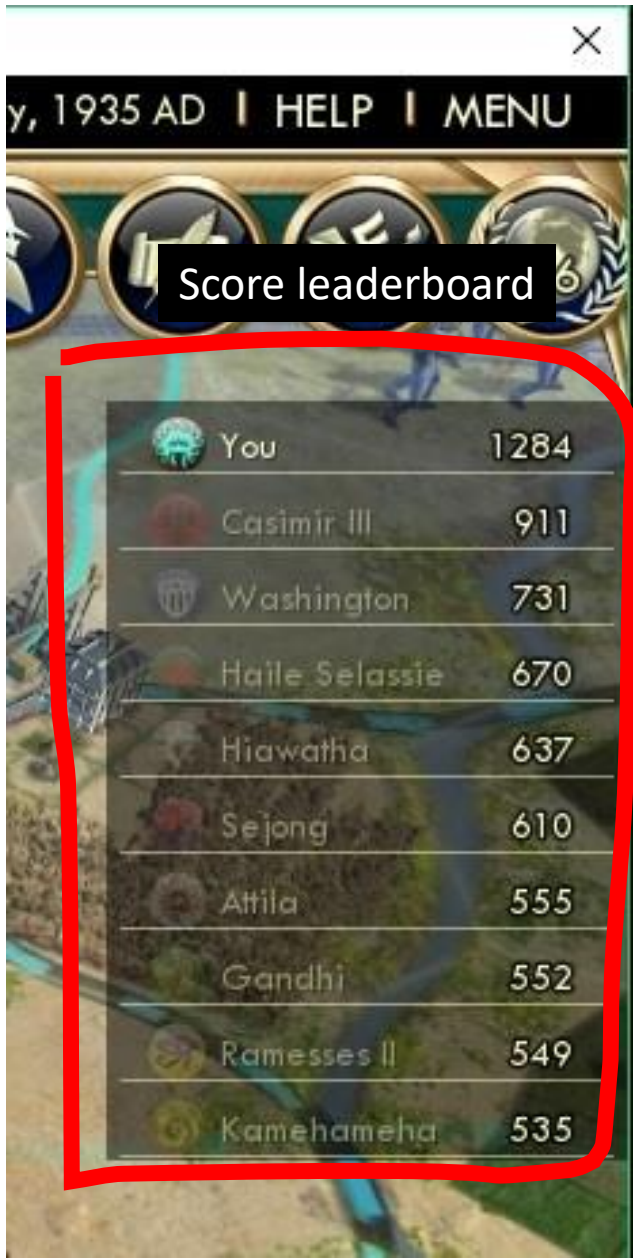
Tying abstract to concrete

Utility(s, p): The most obvious way to calculate utility is to get the score of player p at state s. However, how the game calculates player scores is not made explicit

Depth d: In practice, we get to pick the depth limit to explore. This implies how far we will look into the future

Cutoff-test: If we have reached the maximum depth allowed, stop searching

Evaluation function Eval: We can also calculate score at non-terminal states



Final notes before pseudocode

These are the things I tell myself constantly:

- Minimax is a function for one player, who assumes that all other players behave optimally. It's a simulation in one's head: If I do this, he will do this, she will do this, … It's essentially planning ahead of the actual game
- It is better to write multiple functions, instead of combining everything in one place, as it gets quite hard to work through recursion by hand this way
- Although we compare utilities among actions, what we ultimately want is the "best" action, so I should store the utility/action as a pair

Pseudocode: Built on the foundation of the original minimax code

```
# The “command center”, indicating who should go next
Minimax(s, agent_index, depth)
        if agent_index == number of agents then
                agent_index = 0
                depth += 1
        if agent_index == 0 then
                return Max(s, agent_index, depth)
        else
                return Min(s, agent_index, depth)
```

When all players have moved, reset index to 0, implying that the player (MAX) will go next. Also increment depth

Pseudocode

In slide 14, we didn't specify this parameter. Therefore, it will be initialized to 0

```
# Function for the player – MAX
Max(s, agent_index, depth, move_count = 0):
        if Terminal-Test(s) then return Utility(s, agent_index)
        if Cutoff-test(s, d) then return Eval(s)
        if move_count == maximum number of moves allowed then
                return MAX a in Actions(s) (Minimax(Result(s, a), agent_index + 1, depth)[0], action)
        else
                return MAX a in Actions(s) (Max(Result(s, a), agent_index, depth, move_count + 1)[0], action)
```

If player MAX has used all of their allowed moves, it is no longer their turn. Therefore, we need to give the turn to the next opponent

If MAX still has moves to make, we let him stay in control. Note how we increment move_count, but not agent_index

Pseudocode

```
# Function for other players (your opponents) – MIN
Min(s, agent_index, depth, move_count = 0):
        if Terminal-Test(s) then return Utility(s, agent_index)
        if Cutoff-test(s, d) then return Eval(s)
        if move_count == maximum number of moves allowed then
                return MIN a in Actions(s) (Minimax(Result(s, a), agent_index + 1, depth)[0], action)
        else
                return MIN a in Actions(s) (Min(Result(s, a), agent_index, depth, move_count + 1)[0], action)
```

In slide 14, we didn't specify this parameter. Therefore, it will be initialized to 0

If one of the MIN players has used all of their allowed moves, it is no longer their turn. Therefore, we need to give the turn to the next opponent

If MIN still has moves to make, we let him stay in control. Note how we increment move_count, but not agent_index

Example run of pseudocode

- 3 players – MAX, MIN1, MIN2. Each player has 3 moves. Depth limit is 1
- Procedure:

+ Start at Minimax. It is MAX's turn

+ Go to Max. Terminal and Cutoff tests not met. MAX isn't done moving (0 < 3), so they will get to move again

+ Go to Max. Terminal and Cutoff tests not met. MAX isn't done moving (1 < 3), so they will get to move again

+ Go to Max. Terminal and Cutoff tests not met. MAX isn't done moving (2 < 3), so they will get to move again

+ Go to Max. Terminal and Cutoff tests not met. MAX is done moving (3 = 3), so we give the right to another player

+ Go back to Minimax. It is MIN1's turn

+ Go to Min. Terminal and Cutoff tests not met. MIN1 isn't done moving (0 < 3), so they will get to move again

+ Go to Min. Terminal and Cutoff tests not met. MIN1 isn't done moving (1 < 3), so they will get to move again

+ Go to Min. Terminal and Cutoff tests not met. MIN1 isn't done moving (2 < 3), so they will get to move again

+ Go to Min. Terminal and Cutoff tests not met. MIN1 is done moving (3 = 3), so we give the right to another player

+ Go back to Minimax. It is MIN2's turn

Example run of pseudocode

+ Go to Min. Terminal and Cutoff tests not met. MIN2 isn't done moving (0 < 3), so they will get to move again

+ Go to Min. Terminal and Cutoff tests not met. MIN2 isn't done moving (1 < 3), so they will get to move again

+ Go to Min. Terminal and Cutoff tests not met. MIN2 isn't done moving (2 < 3), so they will get to move again

+ Go to Min. Terminal and Cutoff tests not met. MIN2 is done moving (3 = 3), so we give the right to another player

+ Go back to Minimax. agent_index == number of agents (3 = 3), so we reset the index back to 0, while also increment depth to 1. Therefore, it is MAX's turn

+ Go to Max. Cutoff test met (depth = 1, which is depth limit), return evaluation function at that state.

In essence:

Command center → Let MAX make 3 moves → Command center → Let MIN1 make 3 moves → Command center → Let MIN2 make 3 moves → Command center → Cutoff

What I hope the pseudocode achieves

The pseudocode can be generalized to a game of multiple players (not just MAX and MIN), in which each player can make multiple moves (instead of just one move each turn)

That’s it

Thank you for giving me the opportunity to brush up on my Minimax knowledge. If anything on this PowerPoint is poorly conveyed by me, I am more than happy to meet in-office and explain my idea and reasoning.