Artificial Intelligence

15-110 – Monday 11/30

Learning Goals

- Recognize how AIs attempt to achieve **goals** by using a **perception, reasoning,** and **action** cycle
- Build **game decision trees** to represent the possible moves of a game
- Use the **minimax algorithm** to determine an AI's best next move in a game
- Design potential **heuristics** that can support 'good-enough' search for an AI

Perception, Reasoning, and Action

What is Artificial Intelligence?

Artificial Intelligence (AI) is a branch of computer science concerned with techniques that allow computers to do things that, when humans do them, are considered evidence of intelligence.

However, it's extremely hard to build a machine with **general intelligence-** that is, a machine that can do everything a human can do. We're still far away from this goal, as it includes many difficult tasks (visual and auditory perception, language understanding, reasoning, planning, and more).

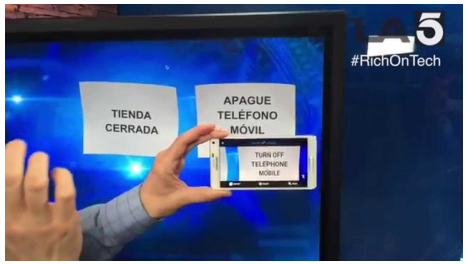
Most modern AI applications are **specialized**; they do one specific task, and they do it very well.

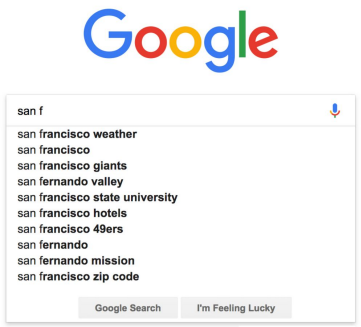
Examples of AIs

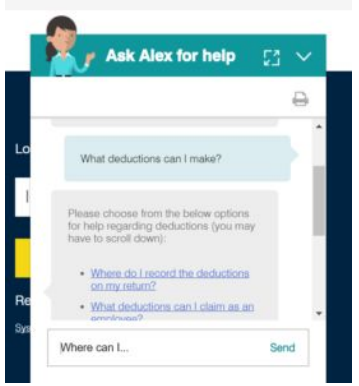
We've built AIs that can play games, run robots, and play Jeopardy.

AI is also used to translate text, predict what you'll type, and answer questions on websites.

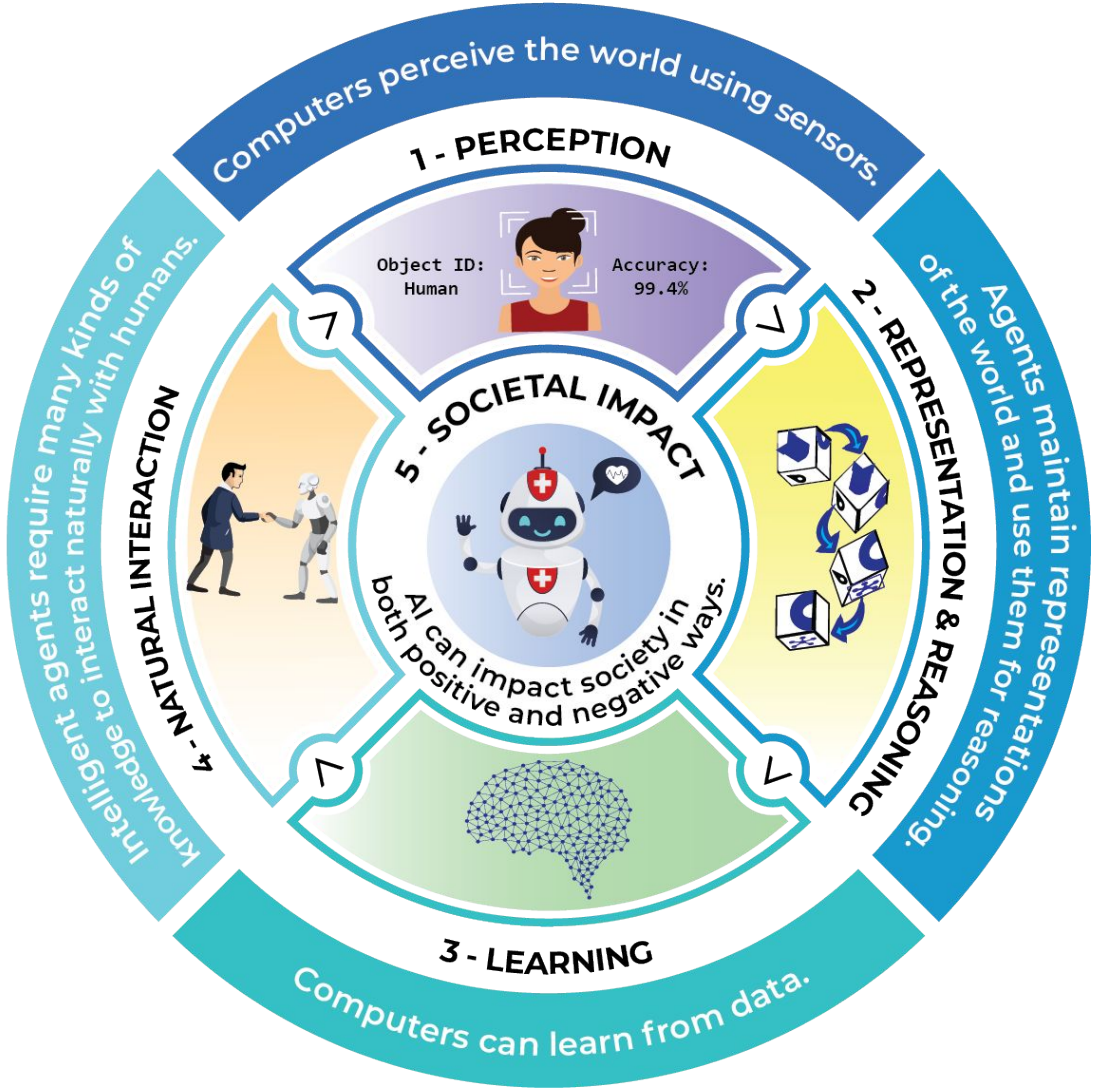
What do these AIs have in common? Each AI we build has a specific **goal**, the thing it is trying to do.







Five Big Ideas in AI (from AI4K12.org)



1 - Perception

2 - Representation & Reasoning

3 - Learning

4 - Natural Interaction

5 - Societal Impact

Perception, Reasoning, and Action

Intelligent agents attempt to reach their goals by cycling through three steps: **perceive** information, **reason** about it, then **act** on it.

This is similar to how humans and animals work. We constantly take in information from our senses, process it, and decide what to do (consciously or unconsciously) based on that 'data'.

The agent’s main task will be to determine a **series of actions** that can be taken to accomplish its goal.

Perception: What Data Can Be Gathered?

First, the agent needs to **perceive** information about the state of the problem it's solving.

This can range from data inputted directly by the user to contextual information about other actions the user has taken. For example, an autocomplete AI might use data both about what the user is currently typing *and* about what they've typed before.

Agents that interact with the real world can perceive information through **sensors**, pieces of hardware that collect data and send it to the agent.

Reasoning: What Should be Done Next?

Second, the agent needs to **reason** about the data it has collected, to decide what should be done next to move closer to the goal.

Reasoning uses **algorithms**, as we've discussed this whole semester. The agent may search through all the possible actions it can take or try to create a model of the world based on the data to better inform its decision.

A general goal of reasoning is to make decisions **quickly**, so that tasks can be accomplished efficiently. You don't want a self-driving car to take too long to decide whether or not to stop!

Action: Here's What to Do

Finally, the agent needs to **act**, to produce a change in the state of the problem. All actions should lead the agent closer to its goal.

Actions don't need to reach the goal *immediately*, and often can't. As long as some progress is made, the agent can continue cycling through perceiving, reasoning, and acting until the goal is reached.

Agents that interface with the real world (robots) use **actuators** to make changes. This can be complicated (moving a robot arm) or simple (turning up the heat on the thermostat).

Example: IBM Watson

IBM's Watson was designed to play (and win!) the game Jeopardy. Its **goal** was to answer Jeopardy problems with a question. How did it work?

Watson **perceived** the questions by receiving them as text, then breaking them down into keywords using natural language processing.

It used that information to search documents in its database, looking for the most relevant information. With that information, Watson used **reasoning** to determine how confident it was that the answer it found was correct.

If Watson decided to answer, it would **act** by organizing the information into a sentence, then pressing the buzzer with a robotic 'finger'.

Search Supports Artificial Intelligence

In Watson (and many other artificial intelligence applications), the key to being able to perceive and act quickly lies in **fast search algorithms**.

Being able to search quickly makes it possible for an AI to look through hundreds of thousands of possible actions to find which action will work best. This is what makes it possible for Watson to find a correct answer so quickly, or for a self-driving car to identify when it needs to stop immediately.

We've discussed many data structures and algorithms to support search already. We'll now introduce three final ideas used by AIs to support fast search- **game trees, minimax,** and **heuristics**.

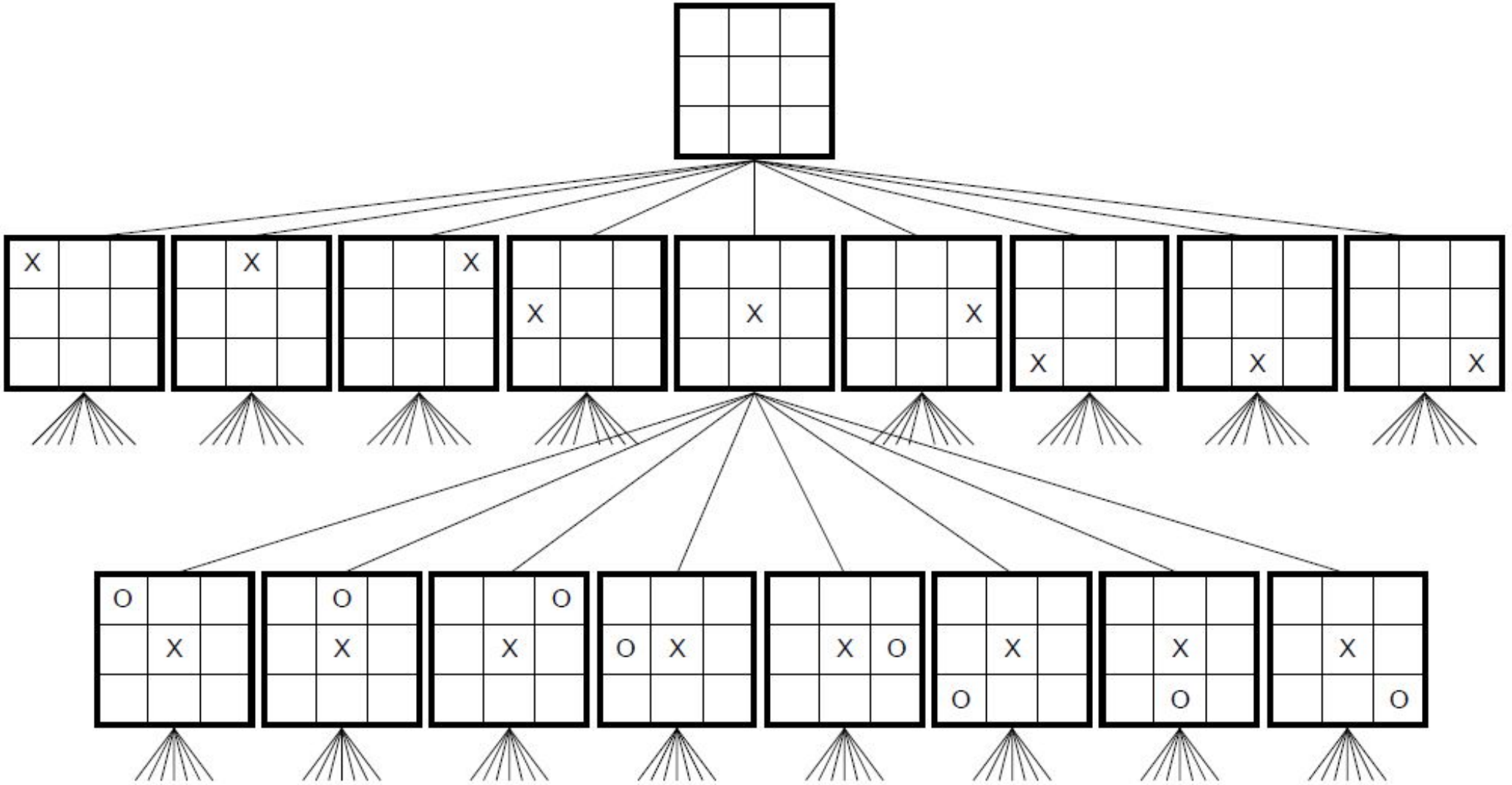
Game Trees and Minimax

Game Trees Use Heuristics To Choose Moves

To search data about possible actions and results quickly, an AI first needs to organize that data in a sensible way. Let's focus on a simple example: a two-player game between an AI and a human.

A game tree is a tree where the nodes are **game states**, and the edges are **actions** made by the AI or the opposing player. Game trees let the AI represent all the possible outcomes of a game.

For example, the game tree for Tic-Tac-Toe looks like this...



Full board here: https://xkcd.com/832/

Reading a Game Tree

The **root** of a game tree is the current state of the game. That can be the start state (as in the previous example), or it can be a game state after some moves have been made.

The **leaves** of the tree are the final states of the game, when the AI wins, loses, or ties.

The edges between the root and the first set of children are the possible moves the AI can make. Then the next set of edges (from the first level to the second) are the moves the opponent can make. These alternate all the way down the tree.

Game Trees are Big

How many possible outcomes are there in a game of Tic-Tac-Toe?

Let's assume that all nine positions are filled. That means the **depth** of the tree is 10 (there are nine moves, so the root + 9 levels). There are 9 options for the first move, 8 for the second, 7 for the third, etc... that's **9!**, which is 362,880.

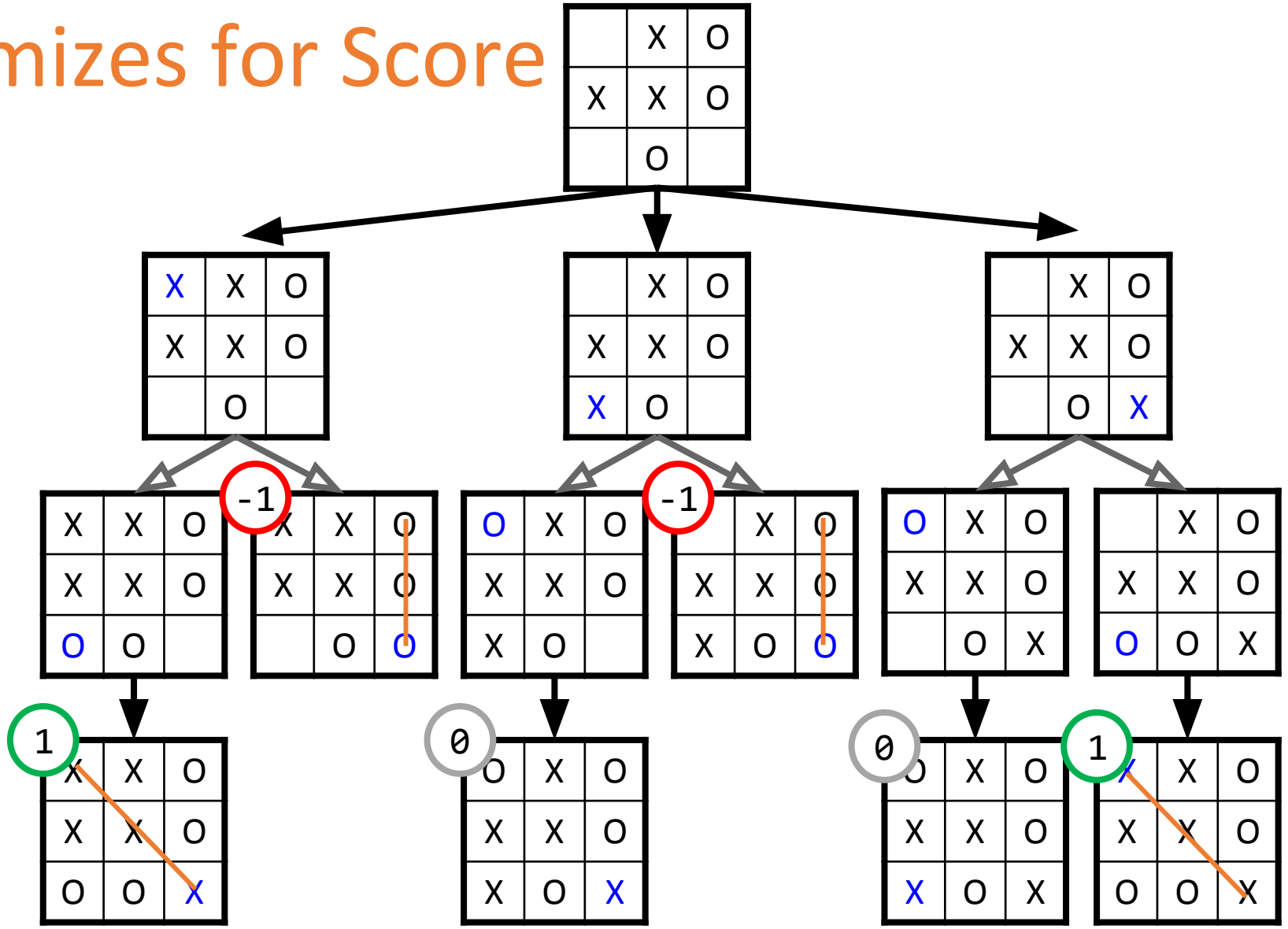
This number is a bit larger than the real set of possibilities (some games end early), but it's a good approximation.

How can the AI choose the best move to make out of all these options?

Minimax Optimizes for Score

The **minimax** algorithm can be used to maximize the final 'score' of a game for an AI.

In Tic-Tac-Toe, we'll say that the score is 1 if the computer wins, 0 if there's a tie, and -1 if the human wins.



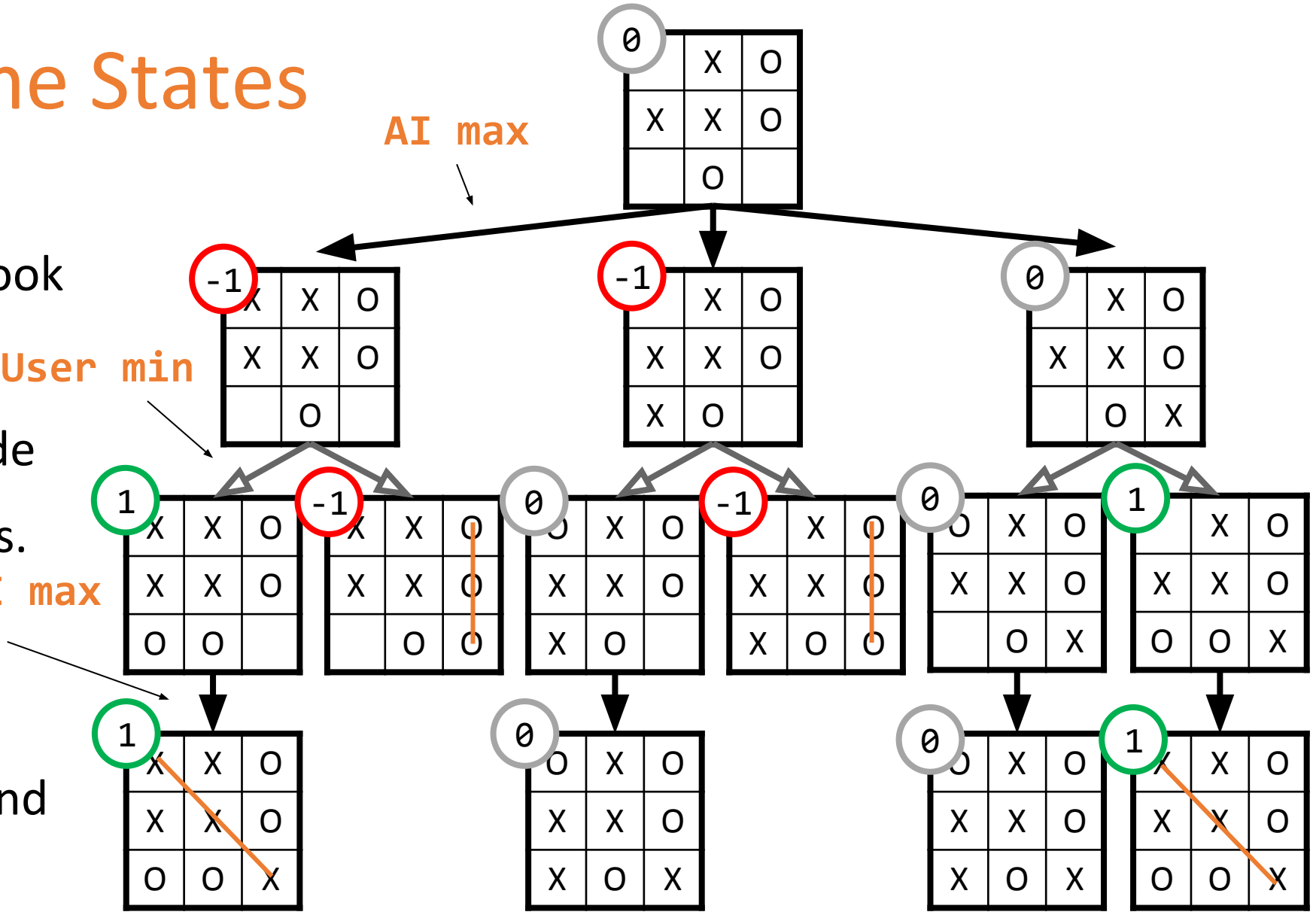
Scoring Game States

How do we score the intermediate states? Look at the scores of the state's children.

If the next move is made by the AI, take the **maximum** of the scores.

If it's made by the opponent, take the **minimum**.

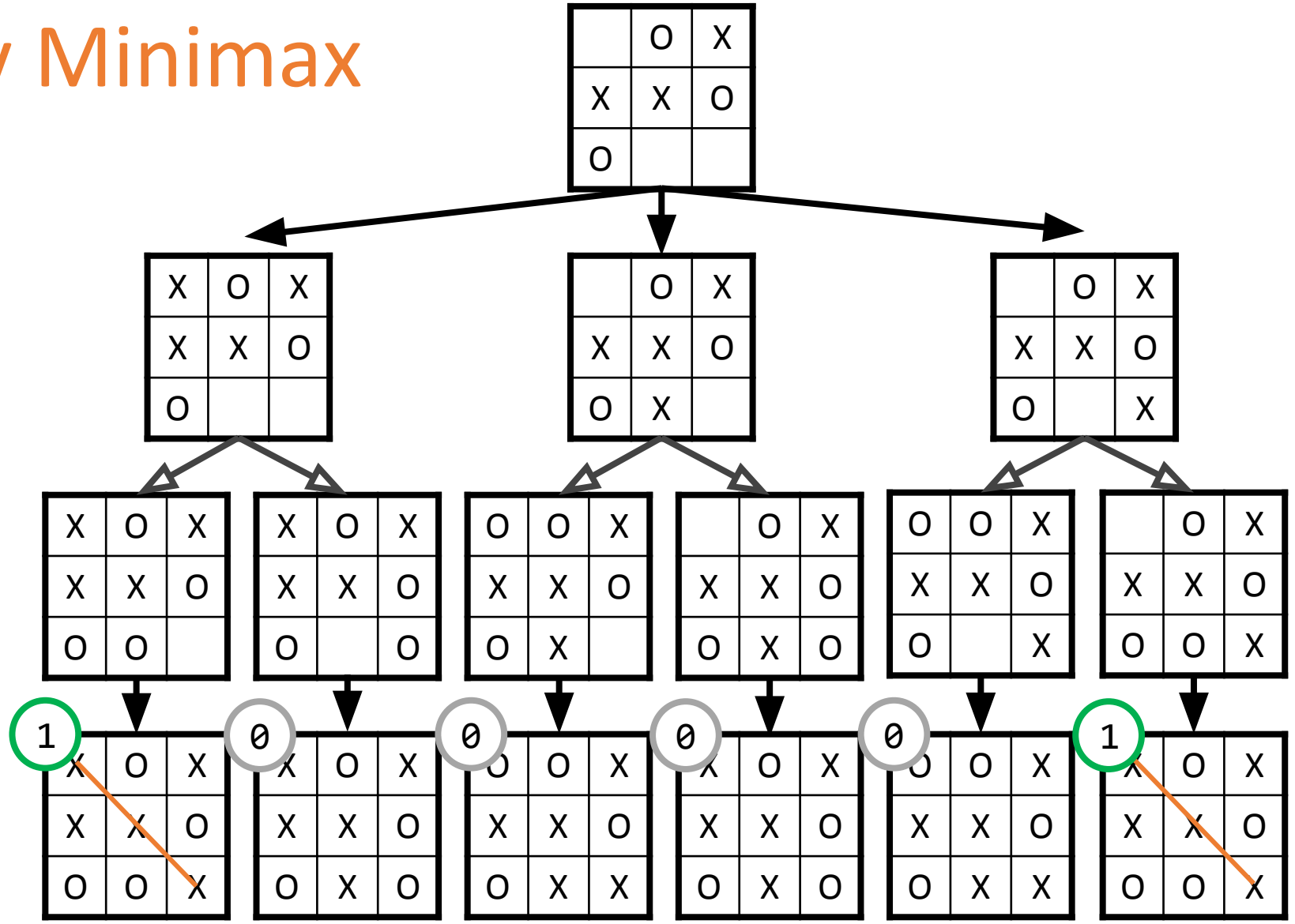
Start from the leaves and build up to the root.



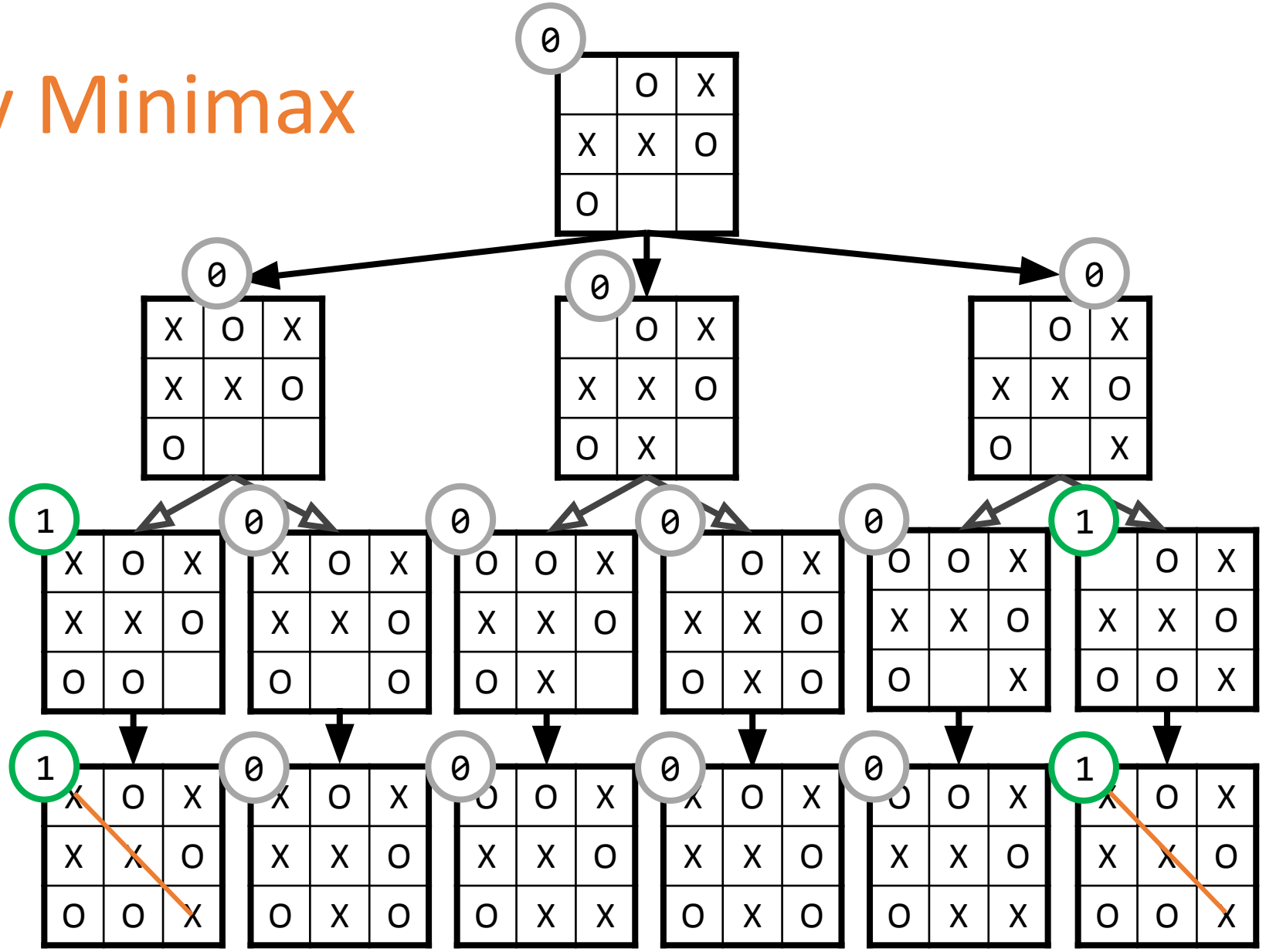
Activity: Apply Minimax

You do: given the tree to the right, apply minimax to find the score of the root node.

Note that the first action is taken by the AI.



Answer: Apply Minimax



Minimax Algorithm

```
# Need to use a general tree- "children" instead of "left" and "right"
def minimax(tree, isMyTurn):
    if len(tree["children"]) == 0:
        return score(tree["value"]) # base case: score of the leaf
    else:
        results = [] # recursive case: get scores of all children
        for child in tree["children"]:
            # switch whose turn it will be for the children
            results.append(minimax(child, not isMyTurn))
        if isMyTurn == True:
            return max(results) # my turn? maximize!
        else:
            return min(results) # opponent's turn? minimize!

def score(state):
    ??? # this depends on your goal
```

Complexity of Minimax

How efficient is minimax? It needs to visit **every node** of the tree, so if the tree has n nodes, it runs in O(n) time.

However, complete game trees are **huge;** more complex games have much larger trees. For example, in Chess, there's an average of 35 possible next moves per turn, with an average of 100 turns per game. That means there are 35^{100} possible states to check – way too many!!

We'll need a way to **constrain** the size of the game tree. We'll do that using **heuristics**.

Heuristics

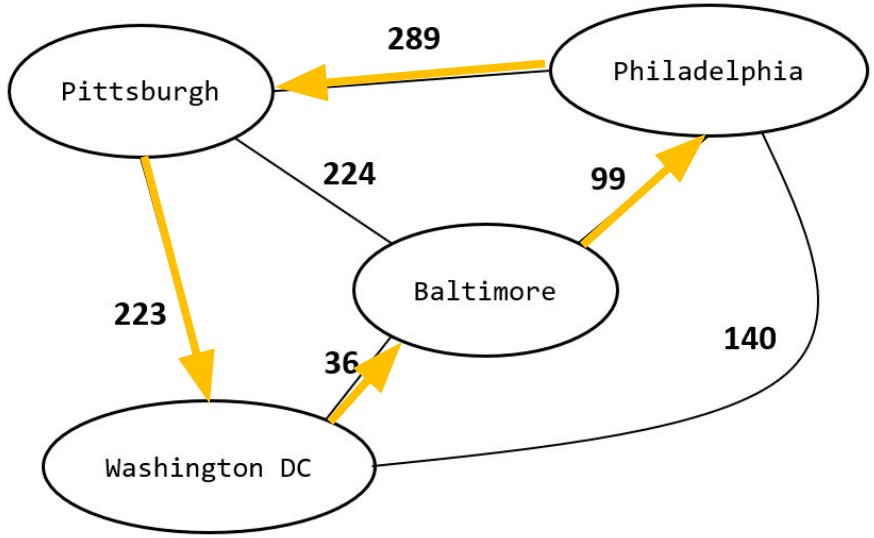
Heuristics Provide Approximate Answers

A **heuristic** is a technique used by an algorithm to find a **good-enough solution** to a problem. Heuristics are typically used because they're **faster** than brute-force algorithms, and because they often achieve good results.

Example: we can create a search heuristic for a graph search that ranks possible next steps. The AI can then try the highest-ranked next step instead of looking at all possible options, and save a lot of time.

Heuristics Example: Travelling Salesperson

Think back to the Travelling Salesperson problem. A heuristic for this problem would be to rank paths based on their length. The algorithm can always choose the next city to visit by trying the shorter paths first.



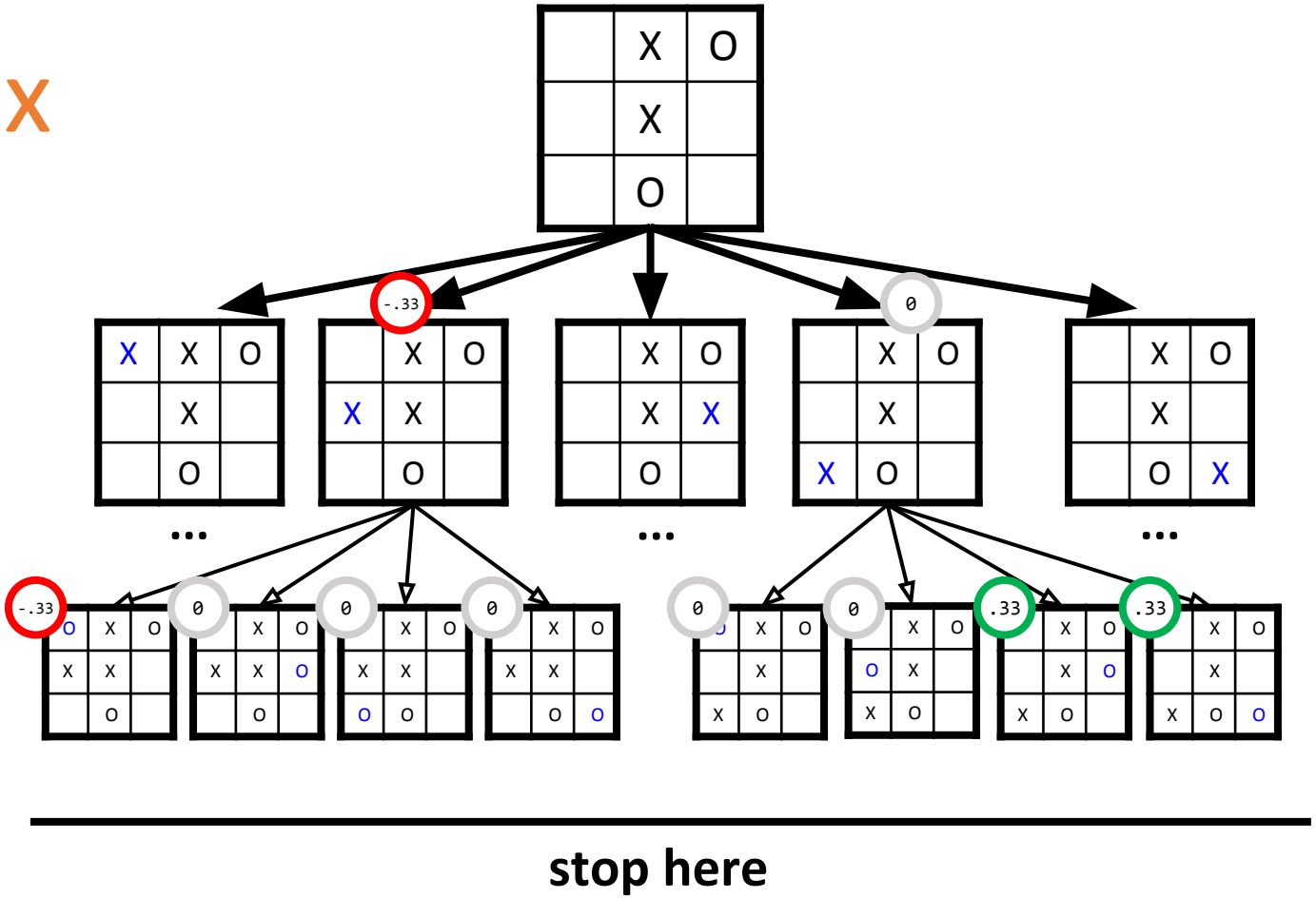
Heuristics are fast, but they also have drawbacks. If we use the Travelling Salesperson heuristic, we lose **optimality;** the path we find will be good, but it might not be the best possible path.

Heuristics in Minimax

The main flaw in minimax is the size of the game tree. We can address this by having the computer move down a set number of levels in the game tree, then stop, even if it has not reached an end state.

For states that are not leaves, use a heuristic to **score** the state based on the current setup of the game. Then the AI can use minimax to find the next-best move based on the heuristic scores.

If the heuristic is well-designed, its score should approximate the real result, and minimax should still produce a good result!



Heuristic: the number of possible X wins - number of possible O wins, divided by total number of non-ties

Design Heuristics to Measure Game State

In general, when you design a heuristic, you should aim to represent the **current state of the game**, and how likely it is that the AI will be able to win.

Example: how could we design a good heuristic for chess? Use information about the state of the board!

How **many** pieces does the computer have left? What about the user?

How **valuable** is each of those pieces? The queen should get a higher rating than a pawn.

You do: other suggestions?

Sidebar: Game AIs

Algorithms like minimax and the use of heuristics have made it possible for AIs to beat world champions at games like Chess, Go, and Poker.

Why did it take 19 years to get from Chess to Go? Go has many more next moves than Chess, so it needed more advanced algorithms (including Monte Carlo randomization and machine learning!).

These AIs will keep improving as computers grow more powerful and we design better algorithms.

DeepBlue beat chess grandmaster Garry Kasparov in 1997

AlphaGo beat 9-dan ranked Go champion Lee Sedol in 2016

Unit 4 Review

On Wednesday, we'll review Unit 4 in lecture.

To request specific topics for review, please fill out this form:
https://forms.gle/gyFKzAzJmBrxwUGu9

Learning Goals

- Recognize how AIs attempt to achieve **goals** by using a **perception, reason,** and **action** cycle
- Build **game decision trees** to represent the possible moves of a game
- Use the **minimax algorithm** to determine an AI's best next move in a game
- Design potential **heuristics** that can support 'good-enough' search for an AI
- **Feedback:** https://bit.ly/110-feedback