Artificial Intelligence

15-110 – Friday 11/19

Announcements

- **Quiz5** happening **Monday during lecture**

- **Check6-2** due **Tuesday at noon**
  - **Check6-1 revisions** also due Tuesday at noon!
  - When testing your Check6-2 work, don't forget to uncomment the tests at the bottom!

- No classes / office hours during Thanksgiving break (Wed 11/23-11/28).

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

Perception, Reason, and Action

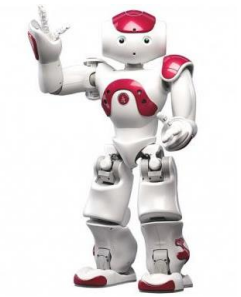
What is Artificial Intelligence?

Artificial Intelligence (AI) is a branch of computer science that studies techniques which 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. We call an AI application trained for a specific task an **agent**.

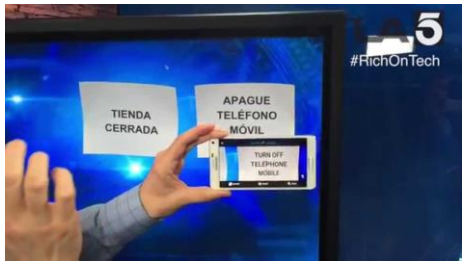
Examples of AI Agents

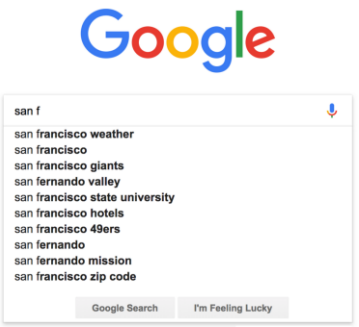


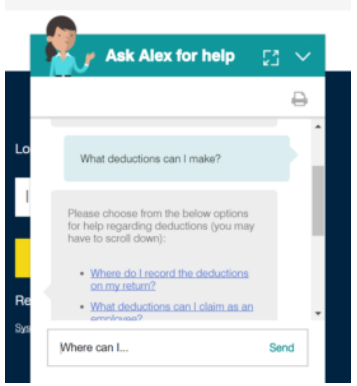
We've built AI agents that can play games, run robots, and win at Jeopardy.

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

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







Perception, Reason, and Action

An AI agent attempts to reach its goal 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'.

An agent's main task is 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 its 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 agent 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.

Reason: What Should be Done Next?

Second, the AI 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 often creates a **model representation** of the world based on the task it needs to solve and the data it has collected so far. It can then search through all the possible actions it can take to 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 long to decide whether or not to stop!

Action: Here's What to Do

Finally, the AI 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 AI agent 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 agent 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 describe three final ideas used by AI agents to support fast search- **game trees, minimax,** and **heuristics**.

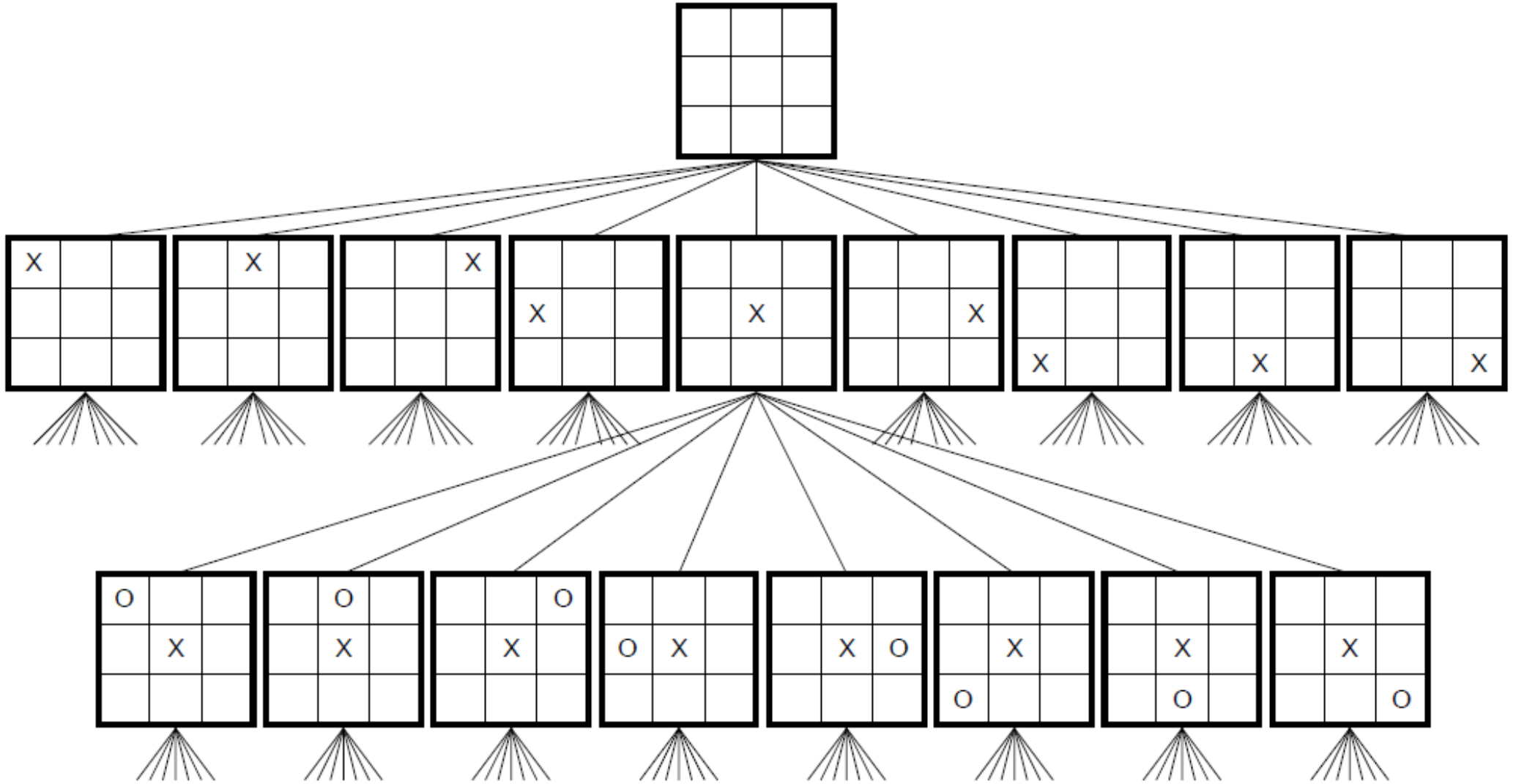
Game Trees and Minimax

Game Trees Represent Possible World States

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

A game tree is a tree where the nodes are **game states** and the edges are **actions** made by the agent or the opposing player. Game trees let the agent 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 agent wins, loses, or ties.

The **edges** between the root and the first set of children are the possible moves the agent can make. Then the next set of edges (from the first level of children 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 results of actions). There are 9 options for the first move, 8 for the second (for each of those nine states), 7 for the third, etc... that's **9!**, which is 362,880.

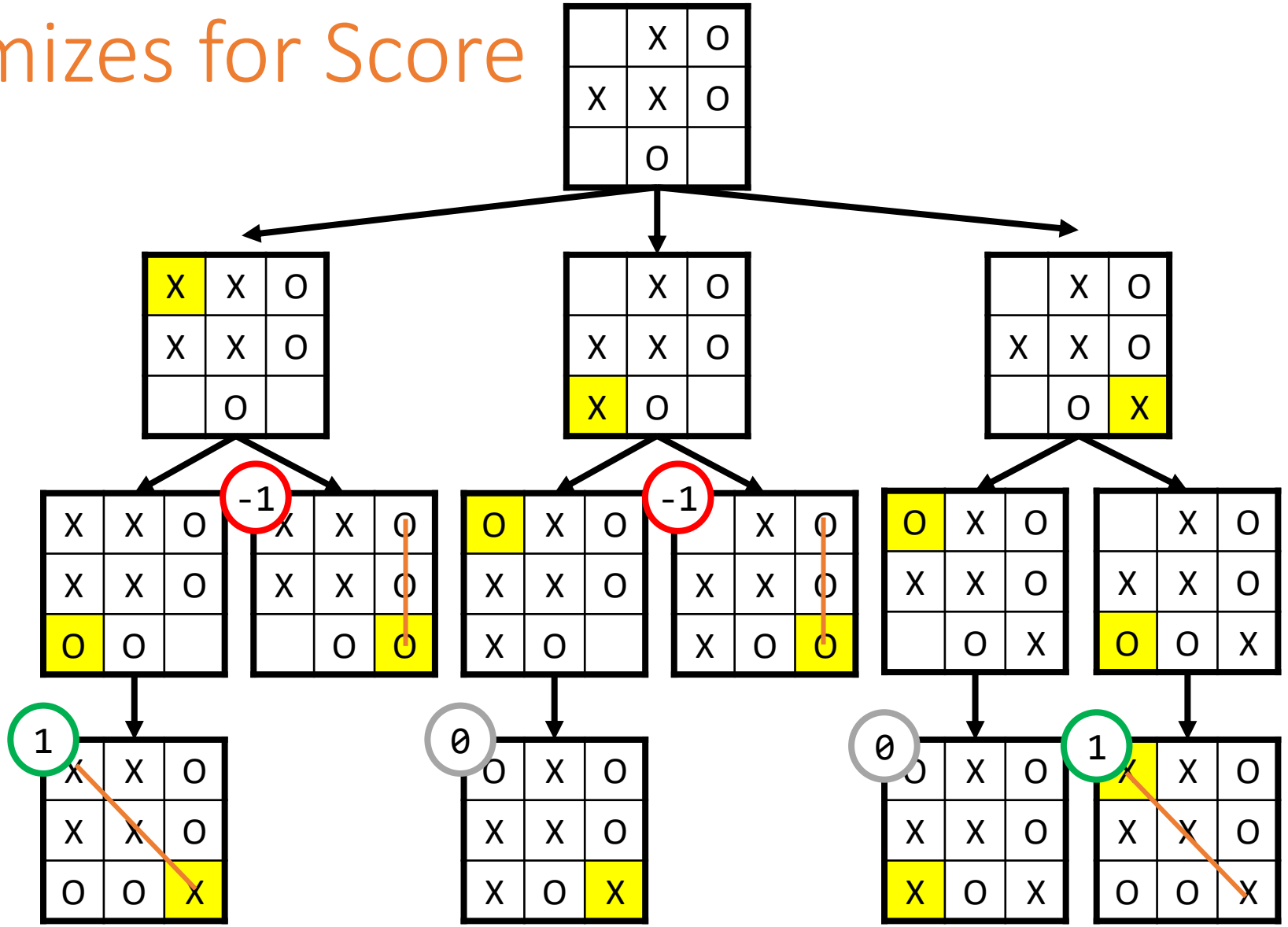
This number is a bit larger than the real set of possibilities (some games end early), but it's a good approximation.

How can the agent choose the best set of moves 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 agent.

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.



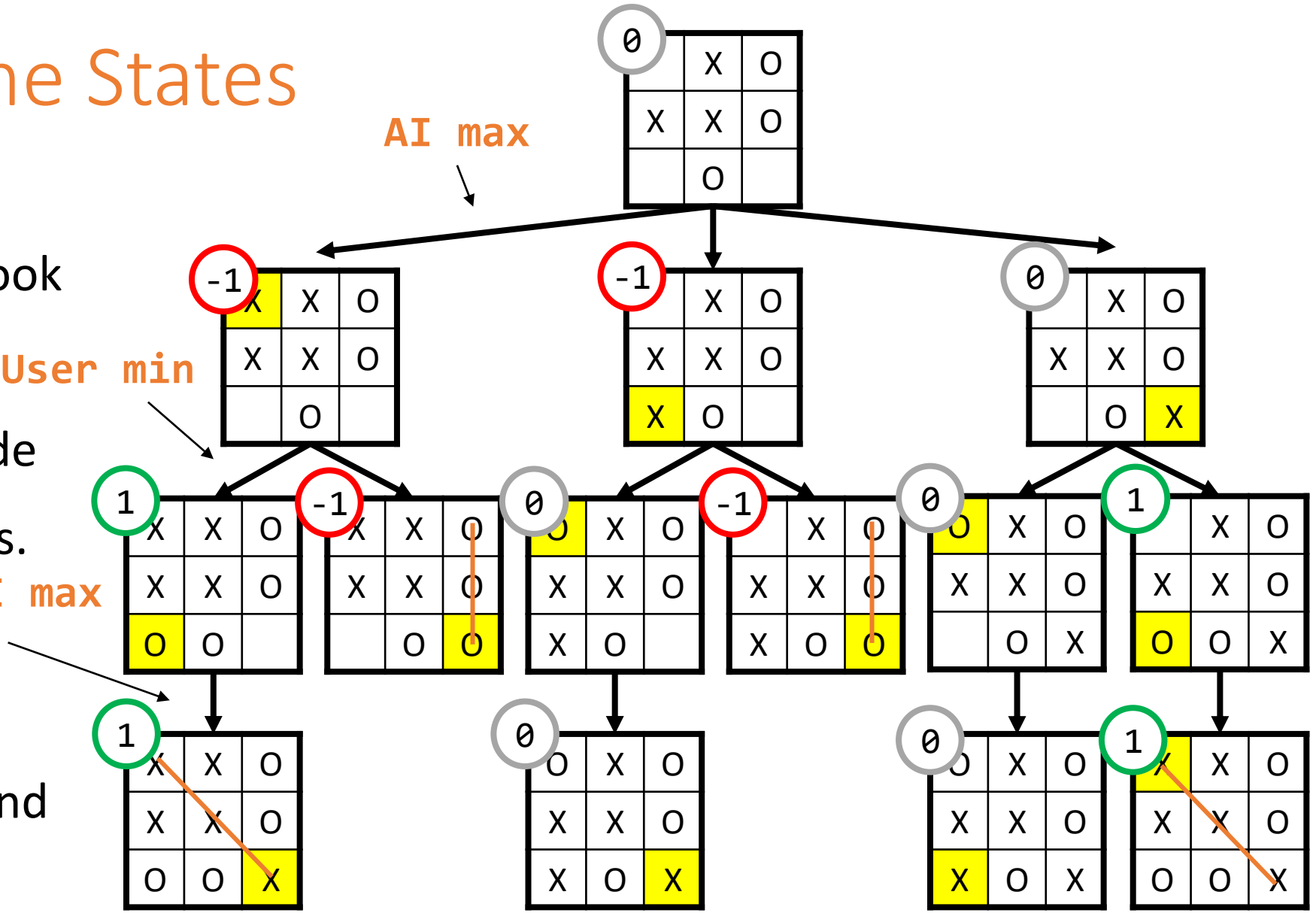
Scoring Game States

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

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

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

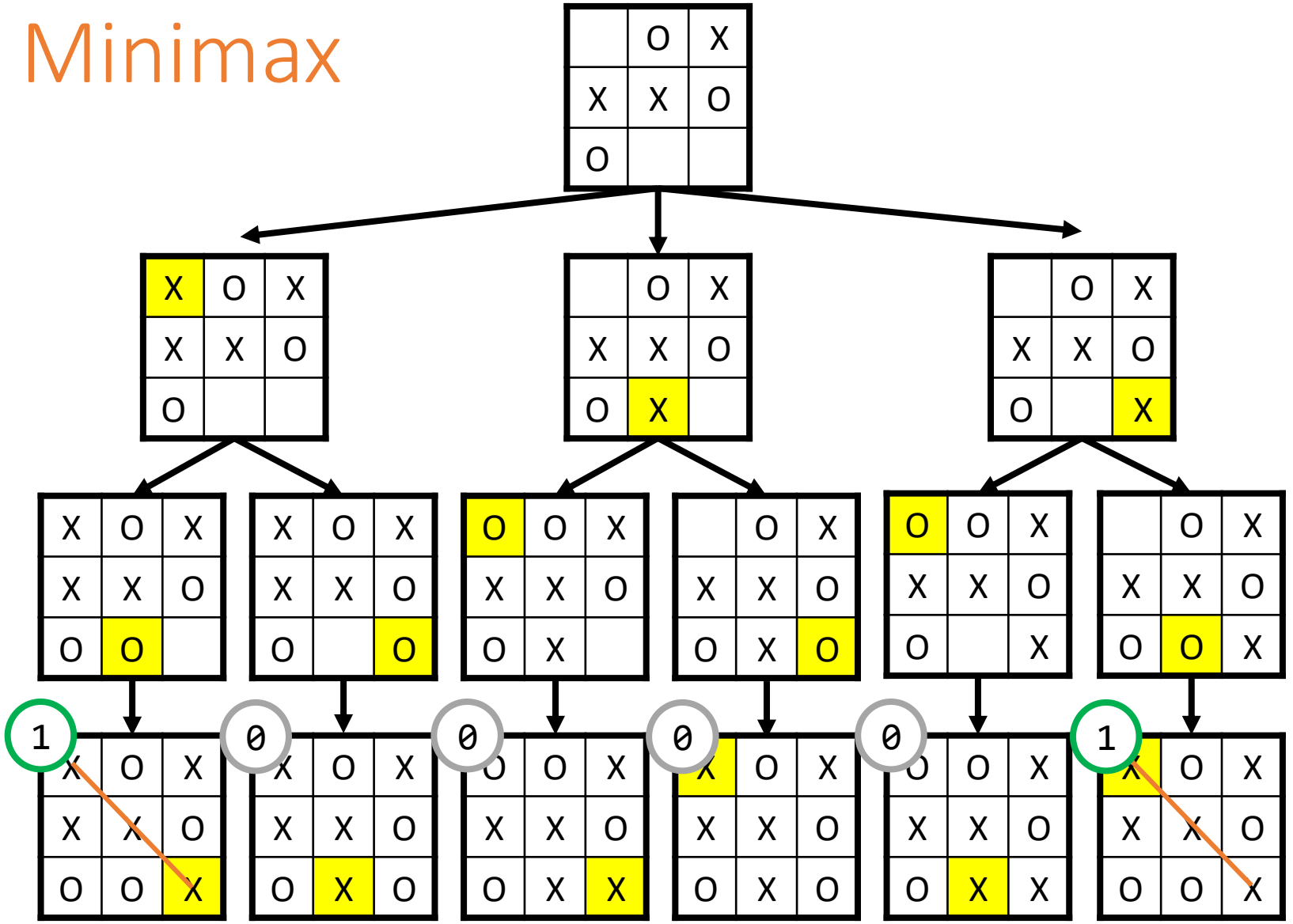
Start from the leaves and build up to the root.



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 agent.



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["contents"]) # 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 the agent's 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.

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**, which we discussed before in the Data Structures unit.

Designing Heuristics

Reminder: Heuristics Are Good Enough

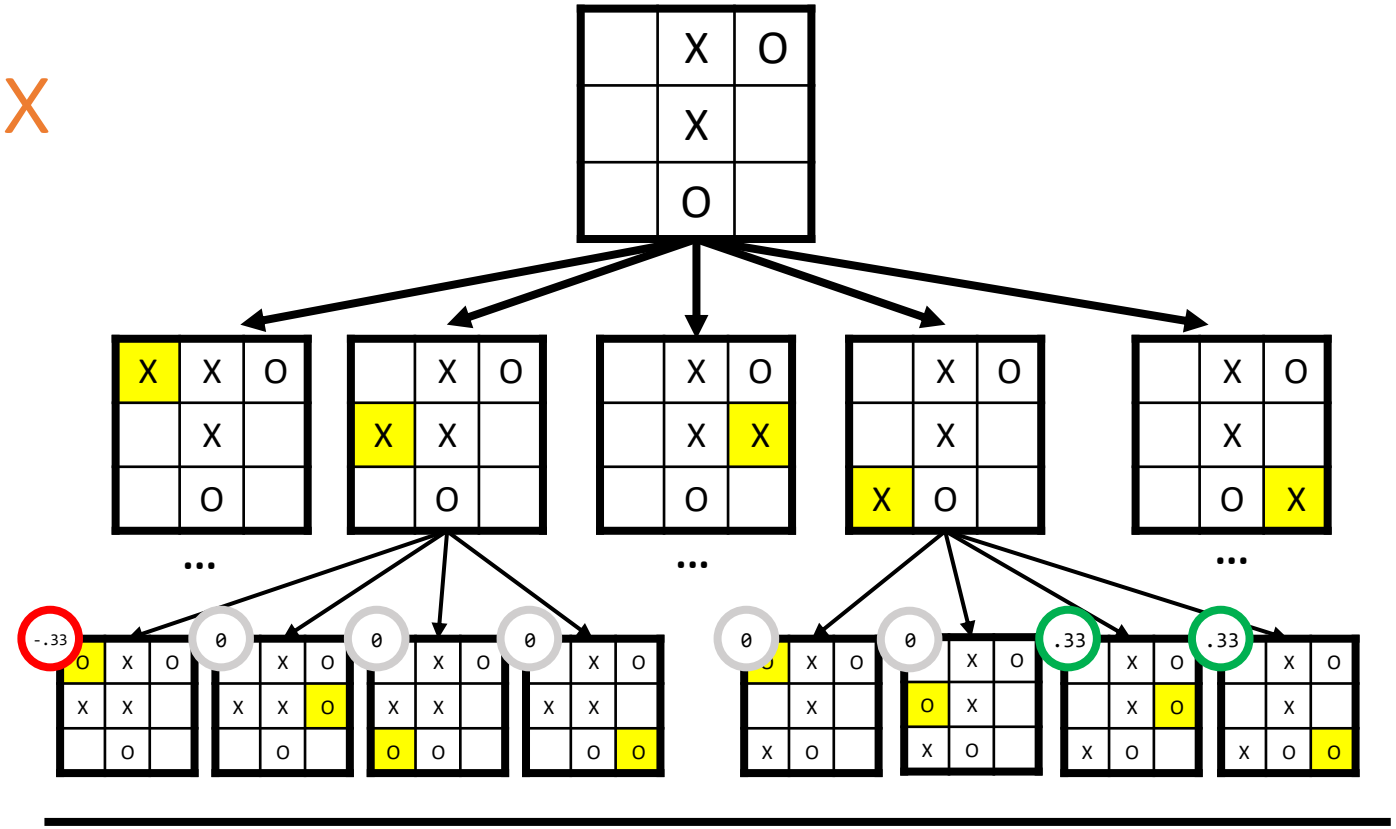
Recall that a **heuristic** is a search technique used by an algorithm to find a **good-enough solution** to a problem. Heuristics may not find the best answer to a problem, but they often achieve good results.

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 agent 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:

(number of possible X wins - number of possible O wins)

---

total number of non-tie results

Design Heuristics to Score Possibilities

The heuristic lets us **score** the possible choices so that we could compare them directly. This is how we handled heuristics with the Travelling Salesperson problem previously as well.

This approach only works if we **design the heuristic well**. The score that the algorithm assigns must be a good representation of the probability that the state is the best choice to make.

How can we design heuristics well? Try to map all the information contained in the state to a number- the larger, the better!

Activity: Heuristics for Chess

Example: Let's consider Chess again. Could we design a heuristic to support an AI for Chess?

Yes – we just need to use information about the state of the board to decide the score of each possibility!

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

How **valuable** are 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 AI agents 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 AI agents 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

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-f21-feedback