Ways of Knowing: Computing

Joan Feigenbaum

http://www.cs.yale.edu/homes/jf/

Yale Univ.; New Haven; Aug. 19, 2025

Joan Feigenbaum: Biosketch

- Educational and professional
  - AB in Mathematics, Harvard, 1981
  - PhD in Computer Science, Stanford, 1986
  - AT&T (Bell) Laboratories, 1986 – 2000
  - Yale Computer Science Dept, 2000 – present
  - Grace Murray Hopper Prof. of CS, 2008 – present
  - CS Department Chair, 2014 – 2017
  - Amazon Scholar, 2018 – present

- Personal
  - Born in East New York, Brooklyn
  - Grew up and went to high school in Valley Stream, Long Island
  - NYC resident since finishing grad school in 1986
  - Married, one child (now 33 years old)

JF's Research Areas
(in chronological order)

- Algebraic graph theory
  - Product graphs
- Cryptographic complexity theory
  - Random-self-reducibility
- Authorization and trust management
- Massive-data-stream algorithmics
- Economics and computation
  - Incentive-compatible interdomain routing
- Accountability, anonymity, and privacy
- Computer Science and Law

Note Absence of AI and Physics

Edward Albert Feigenbaum is a computer scientist [known for his work] in the field of artificial intelligence. He was the winner, jointly with Raj Reddy, of the 1994 ACM Turing Award. He is often called the "father of expert systems."

Mitchell Jay Feigenbaum was an American mathematical physicist whose pioneering studies in chaos theory led to the discovery of the Feigenbaum constants. He was a winner of the MacArthur Award (1984), the Wolf Prize (1986), and the Heineman Prize (2008).

I am not related to either Ed or Mitchell. I have no distinguished relatives.

“Knowing” the Solution to a Problem

- Before computers, if you wanted to “know” the solution to a problem, you could
  - Figure it out
  - Look it up in a book
  - Ask an expert
  - …
- Once computers became commonplace, you had a new option: Run a computer program.
- In TradCS, a “problem” was a precise, well posed question that had a correct answer.

To “Know” a Solution Because You Ran a Computer Program, You’d Need:

- A correct procedure (or “algorithm”) that solves the problem “efficiently”
- A correct implementation of the algorithm

To “Know” a Solution Because You Ran a Computer Program, You’d Need:

- A correct procedure (or “algorithm”) that solves the problem “efficiently” [math]
- A correct implementation of the algorithm

To “Know” a Solution Because You Ran a Computer Program, You’d Need:

- A correct procedure (or “algorithm”) that solves the problem “efficiently” [math]
- A correct implementation of the algorithm

  [“formal methods” in software engineering]

Yale CS RoSE Group

“Rigorous Software Engineering”

https://rose.yale.edu/

Theory of Computation

- There are problems that computers can't solve. Canonical example is the "halting problem."
- Among problems that they can solve, some are easy ("efficiently solvable" or "tractable"), and some are hard ("intractable").

Theory of Computation

- There are problems that computers can't solve. Canonical example is the "halting problem."
- Among problems that they can solve, some are easy ("efficiently solvable" or "tractable"), and some are hard ("intractable").

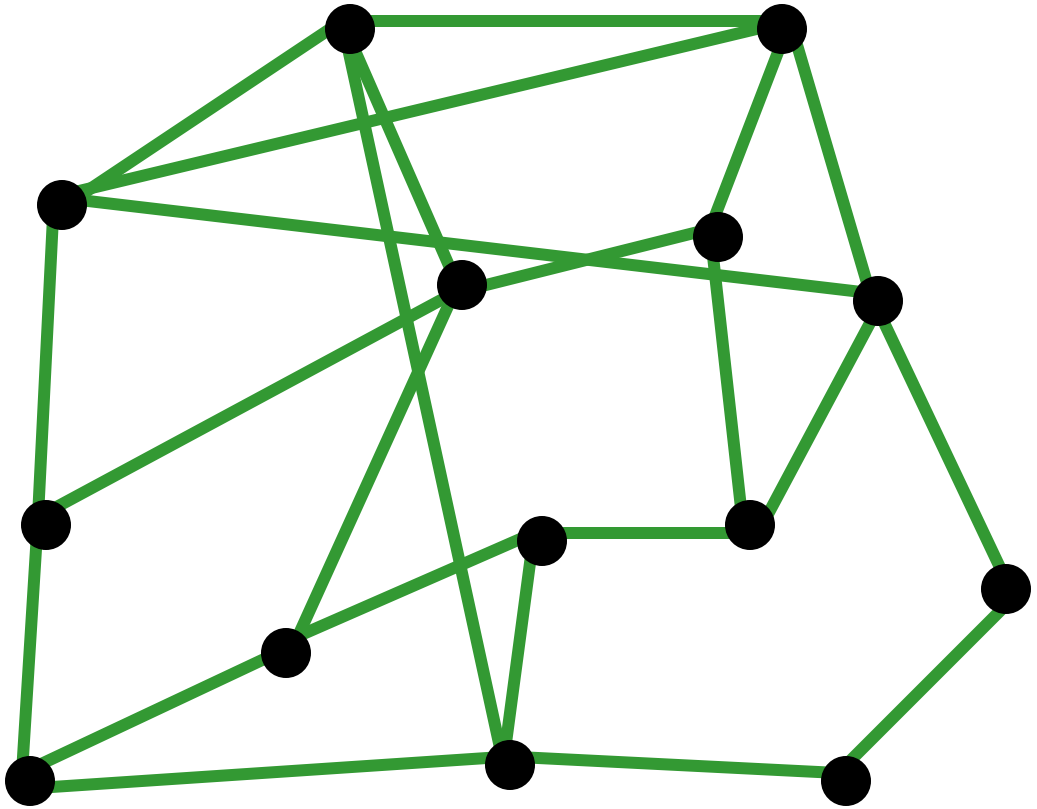
Separating the tractable from the intractable (esp., **proving lower bounds** on computational complexity) is the **core challenge in theoretical CS**.

Theory of Computation

- There are problems that computers can't solve. Canonical example is the "halting problem."
- Among problems that they can solve, some are easy ("efficiently solvable" or "tractable"), and some are hard ("intractable").
- Some computational problems are "equivalent" with respect to "hardness."
- There is a crucial and unresolved relationship between "finding solutions" and "verifying" them.

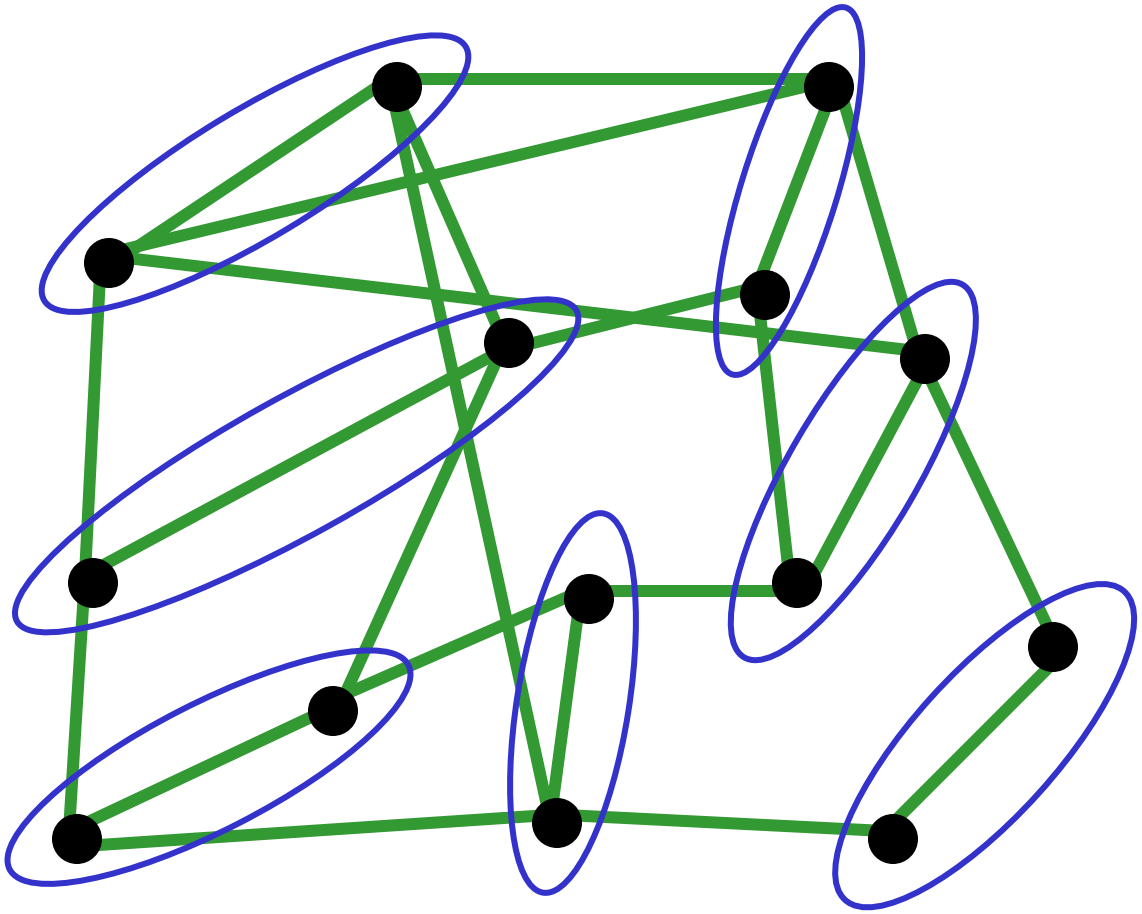
Efficiently Solvable

Nontrivial Example: Matching



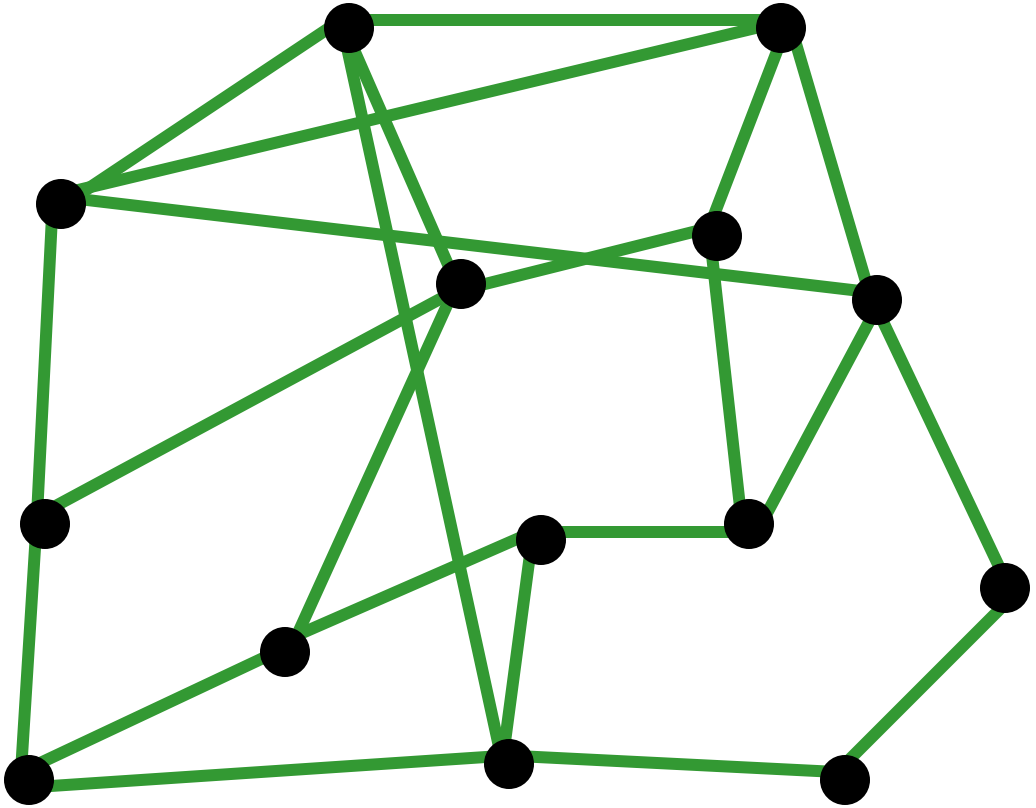
Efficiently Solvable

Nontrivial Example: Matching



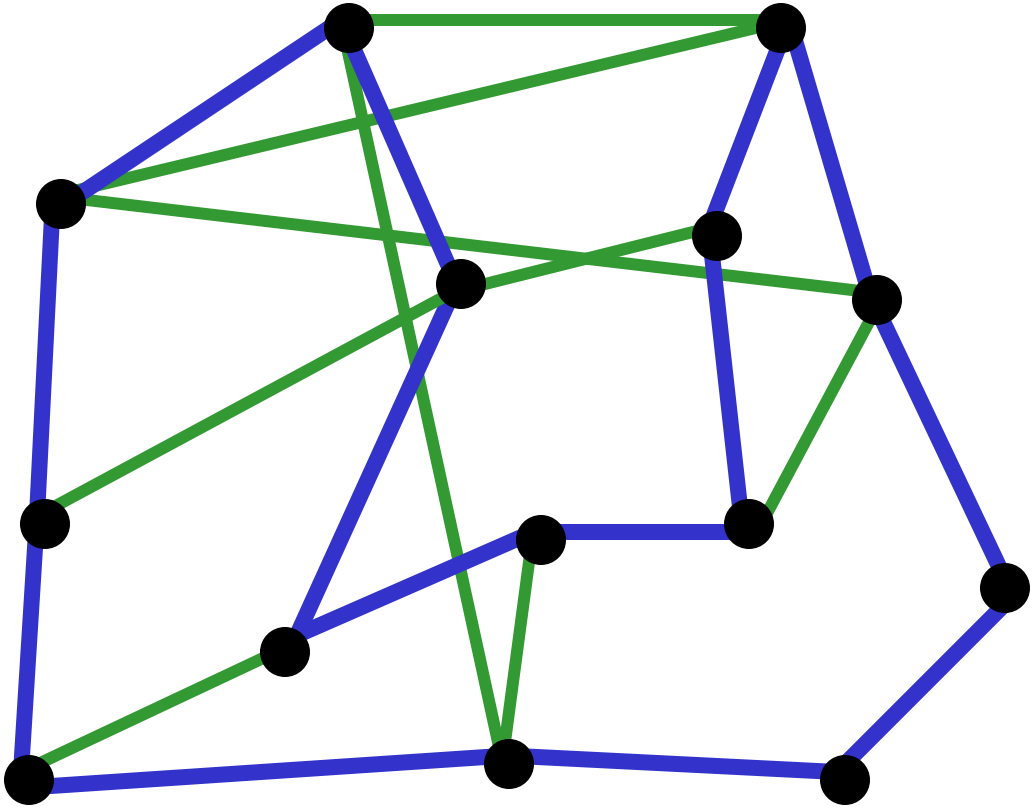
Efficiently Verifiable

Trivial Example: Hamiltonian Cycle (HC)



Efficiently Verifiable

Trivial Example: Hamiltonian Cycle (HC)



- Is it Easier to Verify a Solution than to Find one?

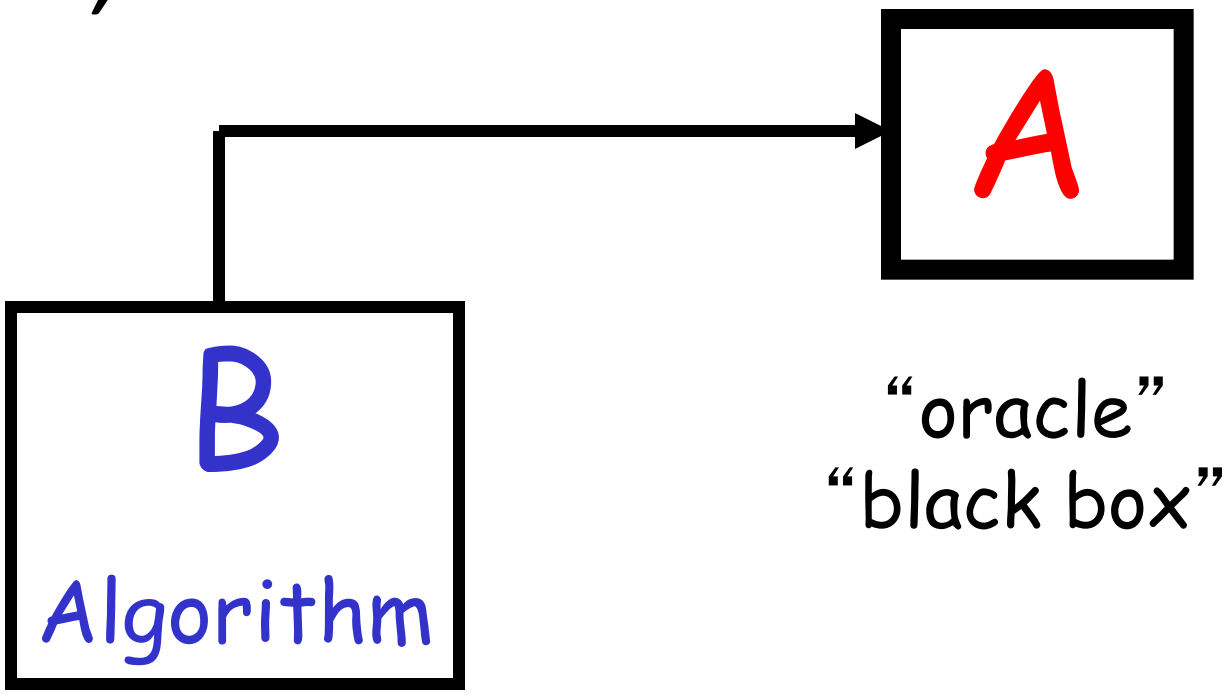
- Fundamental Conjecture of Computational Complexity:

$$P \neq NP$$

Efficient Reduction of B to A

If A is "Easy,"

then B is, too.



Conceptual Breakthrough: NP-Completeness

- The Cook-Levin Theorem (1971):

  SAT is NP-complete

  $\exists\, x_1, x_2, x_3, x_4\ (x_1 \vee \neg x_2 \vee x_4) \wedge (x_2 \vee \neg x_3 \vee \neg x_4)$

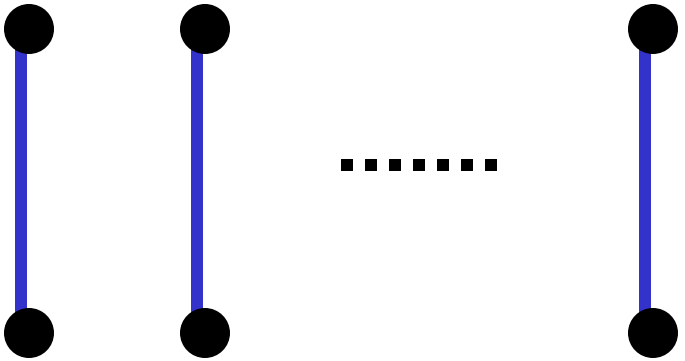
- NP-complete problems
  - The "hardest" problems in NP
  - An equivalence class under efficient reductions

- 10k's naturally occurring problems (including HC) in diverse fields have been shown to be NP-complete

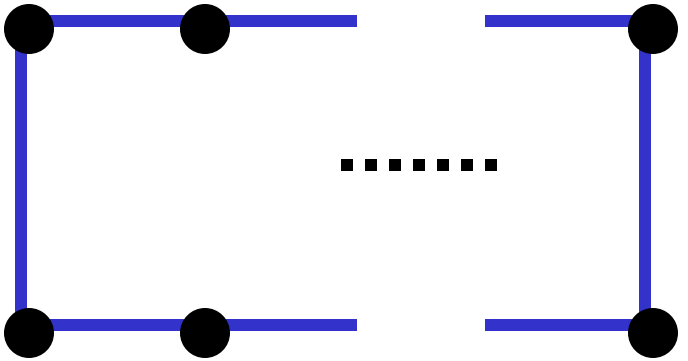
  Math, CS, Engineering, Economics, Physical Sciences, Geography, Politics…

Clarified Observed Differences

Matching:



HC:



Fundamentally Different

That was about TradCS.
AI and ML Change the Story.

- Now the computer is supposed to “learn” how to compute a solution.

What is Machine Learning (ML)? (1)

Moritz Hardt
Max Planck Institute
for Intelligent Systems

A **learning algorithm** is, loosely speaking, any algorithm that takes historical instances (so-called **training data**) of a decision problem as input and produces a decision rule or **classifier** that is then used on future instances of the problem. An attribute of the data is often called a **feature**, and the set of all available attributes defines the **feature space** or **representation** of the data.

What is Machine Learning (ML)? (2)

- **Training** phase
  - Input: A (usually large) set of **classified examples**
  - Example: $(f_1, f_2, \ldots, f_d, c)$; f_i is a **feature**, and c is a **class**.
  - Output: A **model** or **classifier** $P(\cdot, \cdot, \ldots, \cdot)$
- **Prediction** phase
  - Input: An **unclassified example** $(f_1, f_2, \ldots, f_d)$
  - Output: The **predicted class** $c \leftarrow P(f_1, f_2, \ldots, f_d)$
- Use cases abound: Medical diagnoses, fraud detection, investment outcomes, spam detection, credit worthiness, ad effectiveness, criminal justice (sentencing or bail), *etc.*
- Note the close relationship with **Big Data.**

That was about TradCS.
AI and ML Change the Story.

- Now the computer is supposed to “learn” how to compute a solution.
- In general, the model won’t be a perfectly “correct” program. It’s the result of crunching a very large set of training data down to a compact representation.
- The model can still be useful in the context of a real-world application, because an incorrect answer can be detected and discarded.

Gen AI Blows the Story Away!

- "Generative AI is a type of artificial intelligence that creates new content, such as text, images, music, and videos, based on patterns learned from existing data. **Unlike traditional AI that analyzes or categorizes data, generative AI produces novel and original content.** It learns from vast datasets and uses algorithms to **mimic human creativity**, generating new material based on the patterns it has learned."

Gen AI Blows the Story Away!

- "Generative AI is a type of artificial intelligence that creates new content, such as text, images, music, and videos, based on patterns learned from existing data. **Unlike traditional AI that analyzes or categorizes data, generative AI produces novel and original content.** It learns from vast datasets and uses algorithms to **mimic human creativity**, generating new material based on the patterns it has learned."

- Technically, an LLM (*e.g.*, ChatGPT) does something like "maximum-likelihood, next-token generation."
- Practically speaking, what real-world problem does that solve?

When you query an LLM, do you have in mind a “well posed question that has a correct answer”?

After receiving the LLM output, what do you “know”?

DISCUSSION