

CS 341 W22 Lecture 23

Undecidability

R. Peng, University of Waterloo



Recap & Plan

- So far in this course:
 - Algorithms, efficiency
 - Polynomial vs. exponential time
 - Bad news: some problems seem to only have exponential time algorithms (NP complete problems).
- Next: worse news, some problems have no algorithms
- Examples
 - Program equivalence: given two programs, do they output the same thing on all inputs?
 - Tiling: given finite set of tiles with colored edges, do they tile the plane.



Halting Problem

Given a computer program, does it halt?

We will show that the halting problem has no algorithm.

But first an intuitive reason: many famous open problems in math reduce to the halting problem.



Goldbach Conjecture, 1742

https://en.wikipedia.org/wiki/Goldbach%27s_conjecture:

every even number bigger than 2 can be written as the sum of two primes

As input to halting problem:

```
for(n = 4;;n+= 2)
    ok = False
    for i = 2...n/2
        if(IsPrime(i) & IsPrime(n - i))
            ok = True
    if (!Ok) Halt
```

This program terminates if and only if Goldbach's conjecture is false.



Collatz Conjecture

https://en.wikipedia.org/wiki/Collatz_conjecture

Does this program halt for all inputs n :

```
while(n > 1)
```

if n is even

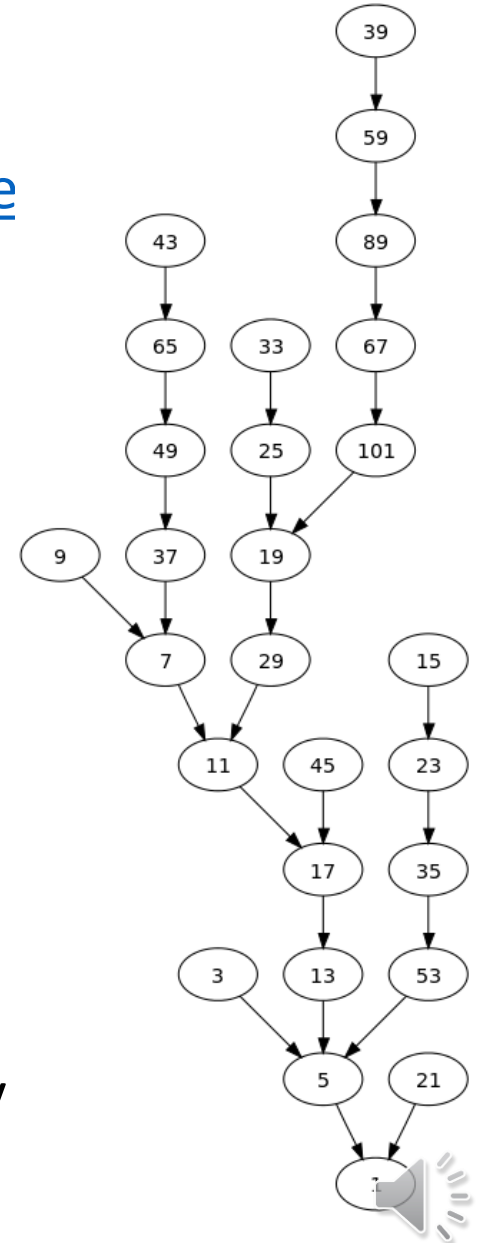
$$n \leftarrow n / 2$$

else n is odd

$$n \leftarrow n * 3 + 1$$

Paul Erdős: "Mathematics may not be ready for such problems."

Jeffrey Lagarias: "is an extraordinarily difficult problem, completely out of reach of present day mathematics."



Collatz Conjecture with Halt

```
for(;;++n)
    If(!Halt(Check(n)))
        exit()
```

This program does not halt iff
Collatz conjecture is true

```
Check(n)
    while(n > 1)
        if n is even
             $n \leftarrow n / 2$ 
        else \\n is odd
             $n \leftarrow n * 3 + 1$ 
```

Note: running this code directly may infinite loop for two reasons

- No such n exists
- $\text{Check}(n)$ doesn't terminate for some n



Undecidable Problems

Definition: a decision problem is **undecidable** if it has no algorithm

Definition (more general, not just for decision problems)
a problem is unsolvable if it has no algorithm

What is an algorithm? As discussed earlier in this course, any of the following are all equivalent:

- Java / C / Python programs
- Pseudocode
- Random Access Machine (RAM)
- Turing machine

We will study some undecidable problems in this lecture

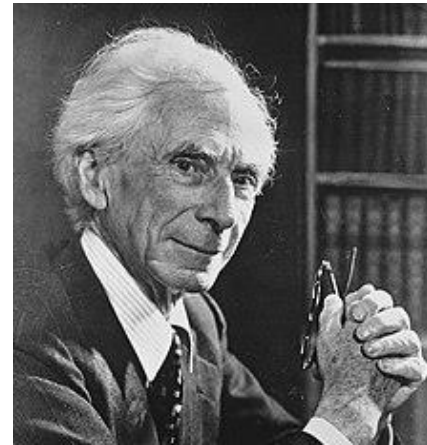


History of Undecidability

- Gottlob Frege: attempted to put all of math onto a firm foundation, formulate a set of axioms such that every true math statement could be derived using basic logic



https://en.wikipedia.org/wiki/Gottlob_Frege



https://en.wikipedia.org/wiki/Bertrand_Russell

- Russel's paradox: once a system is powerful enough, you get self-references that lead to contradictions.



Self-Reference

- This statement is false
- Russel's paradox:
 - Let S be the set of all sets that don't contain themselves.
 - Is S in S ? Either YES or NO give contradictions

Analogous ideas allow us to prove that there are undecidable problems.

If algorithms were powerful enough to decide all problems then we could decide something self-referential, and this gives a contradiction



History of Undecidability

https://en.wikipedia.org/wiki/David_Hilbert

In 1928, David Hilbert asked

- Is mathematics **complete**: every statement can be proved or disproved.
- Is mathematics **consistent**: no valid sequence of proof steps give a contradiction.
- Is mathematics **decidable**: is there an algorithm that determines the truth of any statement.

“Entscheidungsproblem” (= decision problem)



History of Undecidability

- In 1931, Gödel showed that every sufficiently powerful mathematical system is either inconsistent or incomplete.
- In 1931, Alan Turing showed that there is no algorithm to determine the truth of any mathematical statement.



https://en.wikipedia.org/wiki/Kurt_G%C3%B6del



https://en.wikipedia.org/wiki/Alan_Turing



Summary of Lecture 23 Part 1

Concept, intuition, and history of undecidability

What you should know from Lecture 23 Part 1:

- Definition of undecidable.

Next: proving problems are undecidable.



Proving Undecidability

As with NP-completeness, the first undecidability proof is ~~hard~~ different, but the following proofs are by reductions.

The Halting Problem. Given an algorithm/program A and input w , does A halt on w ?

Theorem. The Halting Problem is undecidable



Proof via Self Reference

- Suppose there is a program H for the halting problem.
Input: a program A and input w
Output: YES/NO, does A halt on w
- Make a new program H' :
Input: a program B
If $(H(B, B) == \text{True})$ loop forever, else HALT
- Does $H'(H')$ halt?
 - If yes, $H(H', H')$ returns true, then $H'(H')$ loops forever, contradiction
 - If no, then $H(H', H')$ returns false, but then $H'(H')$ halts, another contradiction!
- So our assumption of H existing is wrong.



Undecidability Proofs

To show other problems are undecidable: use reductions

Recall **reductions**: problem X **reduces to** problem Y, written $X \leq Y$, if an algorithm for Y can be used to make an algorithm for X. Think of this as “X is easier than Y”.

i.e. if Y has an algorithm, so does X

Contrapositive: if X is undecidable, so is Y.



Halt-no-input

Given program A with no input, does A halt?

Theorem: Halt-no-input is undecidable

Proof: show that Halting problem $\leq$ Halt-no-input

Suppose we have P that decides Halt-no-input, then we build an algorithm that decides the halting problem.

Let input to halting problem be B, w. Then we build B' which takes no input, but has w hard coded, followed by B. Then P(B') returns true if and only if B halts on w.



Program Verification

Given a program and specification of the inputs and corresponding outputs, does the program meet the specifications

Theorem: Program Verification is undecidable

Proof: will show that $\text{Halt-no-input} \leq \text{Program Verification}$

Assume we have algorithm V that decides program verification. Then we use it to solve $\text{Halt-no-input}(A)$ by:

- Create A' that reads & discards input, runs A , and output 1.
- Return $V(A', \text{spec: for every input, output is 1})$

Note that A' outputs 1 on all inputs iff A halts



Program Equivalence

Given two programs, do they do the same thing on all inputs?

Theorem: Program Equivalence is undecidable

Proof: Show that $\text{Halt-no-input} \leq \text{Program Equivalence}$

Construct A' as in the spec verification proof

Construct B that reads & ignores input, and outputs 1.

A' and B are equivalent iff A halts.



More Complexity Classes

Are there interesting classes between P and decidable?

$P \subseteq NP \subseteq \text{PSPACE} \subseteq \text{EXP} \subsetneq \text{Decidable} \subsetneq \text{All Decision Problems}$

 ↑ ↑

 Poly(n) space $2^{\text{poly}(n)}$ time

- Know: $P \neq \text{EXP}$
- Open: is $P = \text{PSPACE}$?



Summary of Lecture 23

Undecidability:

- Definition
- Examples of undecidable problems
- Proving undecidability using reductions

What you should know from Lecture 23:

- Definition of undecidable
- How to prove a problem to be undecidable using Turing reduction ($\leq$)

