CSE 311: Foundations of Computing

Lecture 28: Undecidability, Reductions, and Turing Machines
Final Homework Assignment

• Due Wednesday, March 18 11:00 pm
• Submit in Gradescope no grinch
  – Worth > regular homework and < midterm

• For individual questions for me or the CSE 311 staff between now and then use the Ed discussion board.
  – Mark the “Private” checkbox near the bottom of the New Thread creation page
Review: Countability vs Uncountability

• To prove a set $A$ countable you must show
  – There exists a listing $x_1, x_2, x_3, \ldots$ such that every element of $A$ is in the list.

• To prove a set $B$ uncountable you must show
  – For every listing $x_1, x_2, x_3, \ldots$ there exists some element in $B$ that is not in the list.
  – The diagonalization proof shows how to describe a missing element $d$ in $B$ based on the listing $x_1, x_2, x_3, \ldots$.

*Important:* the proof produces a $d$ no matter what the listing is.
CODE(P) means “the code of the program P”

The Halting Problem

Given: - CODE(P) for any program P
      - input x

Output: true if P halts on input x
        false if P does not halt on input x

Theorem [Turing]: There is no program that solves the Halting Problem

Proof: By contradiction.

Assume that a program H solving the Halting program does exist. Then program D must exist
**Does \( D(\text{CODE}(D)) \) halt?**

**H** solves the halting problem implies that
\[
\text{H(\text{CODE}(D),x)} \text{ is true iff } D(x) \text{ halts, } \text{H(\text{CODE}(D),x)} \text{ is false iff not } D(x) \text{ halts}
\]

Suppose that \( D(\text{CODE}(D)) \) halts.

Then, by definition of **H** it must be that
\[
\text{H(\text{CODE}(D),\text{CODE}(D))}
\]

Which by the definition of **D** means \( D(\text{CODE}(D)) \) doesn’t halt.

Suppose that \( D(\text{CODE}(D)) \) doesn’t halt.

Then, by definition of **H** it must be that
\[
\text{H(\text{CODE}(D),\text{CODE}(D))}
\]

Which by the definition of **D** means \( D(\text{CODE}(D)) \) halts.

The ONLY assumption was the program \( H \) exists so that assumption must have been false.

Contradiction!
The Halting Problem isn’t the only hard problem

• Can use the fact that the Halting Problem is undecidable to show that other problems are undecidable

General method:
Prove that if there were a program deciding \( B \) then there would be a way to build a program deciding the Halting Problem.

“\( B \) decidable \( \rightarrow \) Halting Problem decidable”
Contrapositive:
“Halting Problem undecidable \( \rightarrow \) \( B \) undecidable”
Therefore \( B \) is undecidable
Last time: A CSE 141 assignment

Students should write a Java program that:
   – Prints “Hello” to the console
   – Eventually exits

Gradelt, Practicelt, etc. need to grade the students.

How do we write that grading program?

WE CAN’T: THIS IS IMPOSSIBLE!
Last Time: A related undecidable problem

• HelloWorldTesting Problem:
  – Input: CODE(Q) and x
  – Output:
    True if Q outputs “HELLO WORLD” on input x
    False if Q does not output “HELLO WORLD” on input x

• Theorem: The HelloWorldTesting Problem is undecidable.
• Proof idea: Show that if there is a program T to decide
  HelloWorldTesting then there is a program H to decide the
  Halting Problem for code(P) and x.
Last time: The HaltsNoInput Problem

- **Input**: $\text{CODE}(R)$ for program $R$
- **Output**: True if $R$ halts without reading input. False otherwise.

**Theorem**: HaltsNoInput is undecidable

General idea “hard-coding the input”:
- Show how to use $\text{CODE}(P)$ and $x$ to build $\text{CODE}(R)$ so $P$ halts on input $x \iff R$ halts without reading input.
Last time

- The impossibility of writing the CSE 141 grading program follows by combining the ideas from the undecidability of HaltsNoInput and HelloWorld.
More Reductions

- Can use undecidability of these problems to show that other problems are undecidable.

- For instance:

  \[ \text{EQUIV}(P, Q) : \begin{align*}
  \text{True} & \quad \text{if } P(x) \text{ and } Q(x) \text{ have the same I/O behavior for every input } x \\
  \text{False} & \quad \text{otherwise}
  \end{align*} \]
Rice’s theorem

Not every problem on programs is undecidable!

Which of these is decidable?

- Input `CODE(P)` and `x`
  Output: `true` if `P` prints “ERROR” on input `x` after less than 100 steps
  `false` otherwise

- Input `CODE(P)` and `x`
  Output: `true` if `P` prints “ERROR” on input `x` after more than 100 steps
  `false` otherwise

Rice’s Theorem (a.k.a. Compilers Suck Theorem - informal): Any “non-trivial” property of the input-output behavior of Java programs is undecidable.
CFGs are complicated

We know can answer almost any question about
• Regular Expressions, DFAs, NFAs, FSMs

But many problems about CFGs are undecidable!
• Is there any string that two CFGs both accept?
• Do two CFGs accept the same language?
• Does a CFG accept every string?
Computers and algorithms

• Does Java (or any programming language) cover all possible computation? Every possible algorithm?

• There was a time when computers were people who did calculations on sheets paper to solve computational problems

• Computers as we known them arose from trying to understand everything these people could do.
Before Java

1930’s:

How can we formalize what algorithms are possible?

• Turing machines (Turing, Post)
  – basis of modern computers
• Lambda Calculus (Church)
  – basis for functional programming, LISP
• \( \mu \text{-recursive functions} \) (Kleene)
  – alternative functional programming basis
Turing machines

**Church-Turing Thesis:**
Any reasonable model of computation that includes all possible algorithms is equivalent in power to a Turing machine

**Evidence**
- Intuitive justification
- Huge numbers of models based on radically different ideas turned out to be equivalent to TMs
Turing machines

• Finite Control
  – Brain/CPU that has only a finite # of possible “states of mind”

• Recording medium
  – An unlimited supply of blank “scratch paper” on which to write & read symbols, each chosen from a finite set of possibilities
  – Input also supplied on the scratch paper

• Focus of attention
  – Finite control can only focus on a small portion of the recording medium at once
  – Focus of attention can only shift a small amount at a time
Turing machines

- **Recording medium**
  - An infinite read/write “tape” marked off into cells
  - Each cell can store one symbol or be “blank”
  - Tape is initially all blank except a few cells of the tape containing the input string
  - Read/write head can scan one cell of the tape - starts on input

- **In each step, a Turing machine**
  1. Reads the currently scanned cell
  2. Based on current state and scanned symbol
     i. Overwrites symbol in scanned cell
     ii. Moves read/write head left or right one cell
     iii. Changes to a new state

- Each Turing Machine is specified by its finite set of rules
# Turing machines

<table>
<thead>
<tr>
<th>State</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>s₁</td>
<td>(1, L, s₃)</td>
<td>(1, L, s₄)</td>
</tr>
<tr>
<td>s₂</td>
<td>(0, R, s₁)</td>
<td>(1, R, s₁)</td>
</tr>
<tr>
<td>s₃</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s₄</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Turing Machine Diagram](image-url)
UW CSE’s Steam-Powered Turing Machine

Original in Sieg Hall stairwell
Turing machines

Ideal Java/C programs:
- Just like the Java/C you’re used to programming with, except you never run out of memory
  - Constructor methods always succeed
  - `malloc` in C never fails

Equivalent to Turing machines except a lot easier to program:
- Turing machine definition is useful for breaking computation down into simplest steps
- We only care about high level so we use programs
Turing’s big idea part 1: Machines as data

Original Turing machine definition:

– A different “machine” $M$ for each task
– Each machine $M$ is defined by a finite set of possible operations on finite set of symbols
– So... $M$ has a finite description as a sequence of symbols, its “code”, which we denote $<M>$

You already are used to this idea with the notion of the program code or text but this was a new idea in Turing’s time.
Turing’s big idea part 2: A Universal TM

• A Turing machine interpreter $U$
  – On input $<M>$ and its input $x$,
    $U$ outputs the same thing as $M$ does on input $x$
  – At each step it decodes which operation $M$ would have performed and simulates it.

• One Turing machine is enough
  – Basis for modern stored-program computer
    Von Neumann studied Turing’s UTM design
Takeaway from undecidability

• You can’t rely on the idea of improved compilers and programming languages to eliminate major programming errors
  – truly safe languages can’t possibly do general computation

• Document your code
  – there is no way you can expect someone else to figure out what your program does with just your code; since in general it is provably impossible to do this!
We’ve come a long way!

• Propositional Logic.
• Boolean logic and circuits.
• Boolean algebra.
• Predicates, quantifiers and predicate logic.
• Inference rules and formal proofs for propositional and predicate logic.
• English proofs.
• Set theory.
• Modular arithmetic.
• Prime numbers.
• GCD, Euclid's algorithm and modular inverse
We’ve come a long way!

• Induction and Strong Induction.
• Recursively defined functions and sets.
• Structural induction.
• Regular expressions.
• Context-free grammars and languages.
• Relations and composition.
• Transitive-reflexive closure.
• Graph representation of relations and their closures.
We’ve come a long way!

- DFAs, NFAs and language recognition.
- Product construction for DFAs.
- Finite state machines with outputs at states.
- Minimization algorithm for finite state machines
- Conversion of regular expressions to NFAs.
- Subset construction to convert NFAs to DFAs.
- Equivalence of DFAs, NFAs, Regular Expressions
- Method to prove languages not accepted by DFAs.
- Cardinality, countability and diagonalization
- Undecidability: Halting problem and evaluating properties of programs.
What’s next? ...after the final homework...

• **Foundations II** (CSE 312)
  – Fundamentals of counting, discrete probability, applications of randomness to computing, statistical algorithms and analysis
  – Ideas critical for machine learning, algorithms

• **Data Abstractions** (CSE 332)
  – Data structures, a few key algorithms, parallelism
  – Brings programming and theory together
  – Makes heavy use of induction and recursive defns
Complexity Theory  (in CSE 431 and beyond)

Not just what can be computed at all...

How about what can be computed *efficiently*?

A rich, interesting, and important topic.
Thank you!

That's all Folks!