CSCE 411
Design and Analysis of Algorithms

Set 12: Undecidability
Slides by Prof. Jennifer Welch
Spring 2014
Sources

Understanding Limits of Computing

- So far, we have studied how efficiently various problems can be solved.
- There has been no question as to whether it is possible to solve the problem.
- If we want to explore the boundary between what can and what cannot be computed, we need a model of computation.
Models of Computation

- Need a way to clearly and unambiguously specify how computation takes place
- Many different mathematical models have been proposed:
  - Turing Machines
  - Random Access Machines
  - ...
- They have all been found to be equivalent!
Church-Turing Thesis

- Conjecture: Anything we reasonably think of as an algorithm can be computed by a Turing Machine (specific formal model).

- So we might as well think in our favorite programming language, or in pseudocode.

- Frees us from the tedium of having to provide boring details
  - In principle, pseudocode descriptions can be converted into some appropriate formal model
There Exist Uncomputable Functions

- Consider all programs (in our favorite model) that compute functions from $\mathbb{N}$ to $\mathbb{N}$ ($\mathbb{N}$ is set of natural numbers).

- Show that the set of such functions cannot be enumerated (i.e., is uncountable).

- Show that the set of all programs can be enumerated (i.e., is countable).

- Thus there must be some functions that do not have a corresponding program.
Set of Functions is Uncountable

- Suppose in contradiction the set of functions from \( \mathbb{N} \) to \( \mathbb{N} \) is countable.
- Let the functions in the set be \( f_0, f_1, f_2, \ldots \).
- Define a function \( f^d \) (using "diagonalization") that should be in the set but is not equal to any of the \( f_i \)'s.
- If we can define such a function \( f^d \), this would be a contradiction, and thus the set of functions would be uncountable.
## Diagonalization

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_0$</td>
<td>4</td>
<td>14</td>
<td>34</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$f_1$</td>
<td>55</td>
<td>32</td>
<td>777</td>
<td>3</td>
<td>21</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>$f_2$</td>
<td>90</td>
<td>2</td>
<td>5</td>
<td>21</td>
<td>66</td>
<td>901</td>
<td>2</td>
</tr>
<tr>
<td>$f_3$</td>
<td>4</td>
<td>44</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>34</td>
<td>28</td>
</tr>
<tr>
<td>$f_4$</td>
<td>80</td>
<td>56</td>
<td>32</td>
<td>12</td>
<td>3</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>$f_5$</td>
<td>43</td>
<td>345</td>
<td>12</td>
<td>7</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$f_6$</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>12</td>
<td>15</td>
<td>18</td>
</tr>
</tbody>
</table>
Diagonalization

- Define the function: \( f^d(n) = f_n(n) + 1 \)

- In the example:
  - \( f^d(0) = 4 + 1 = 5 \), so \( f^d \neq f_0 \)
  - \( f^d(1) = 32 + 1 = 33 \), so \( f^d \neq f_1 \)
  - \( f^d(2) = 5 + 1 = 6 \), so \( f^d \neq f_2 \)
  - \( f^d(3) = 7 + 1 = 8 \), so \( f^d \neq f_3 \)
  - \( f^d(4) = 3 + 1 = 4 \), so \( f^d \neq f_4 \)
  - etc.

- So \( f^d \) is not in the list of all functions, contradiction.
Set of All Programs is Countable

- Fix your computational model (e.g., programming language).
- Every program is finite in length.
- For every integer n, there is a finite number of programs of length n.
- Enumerate programs of length 1, then programs of length 2, then programs of length 3, etc.
Uncomputable Functions

- Previous proof just showed there must exist uncomputable functions
- Did not exhibit any particular uncomputable function
- Maybe the functions that are uncomputable are uninteresting...
- But actually there are some VERY interesting functions (problems) that are uncomputable
The Function Halt

Consider this function, called Halt:
- input: code for a program P and an input X for P
- output: 1 if P terminates (halts) when executed on input X, and 0 if P doesn't terminate (goes into an infinite loop) when executed on input X

By the way, a compiler is a program that takes as input the code for another program.

Note that the input X to P could be (the code for) P itself.
- in the compiler example, a compiler can be run on its own code
The Function Halt

- We can view Halt as a function from N to N:
  - P and X can be represented in ASCII, which is a string of bits.
  - This string of bits can also be interpreted as a natural number.
- The function Halt would be a useful diagnostic tool in debugging programs.
Halt is Uncomputable

- Suppose in contradiction there is a program $P_{\text{halt}}$ that computes Halt:
  - On input $(P,X)$, $P_{\text{halt}}$ returns 1 if $P$ halts on input $X$ and $P_{\text{halt}}$ returns 0 if $P$ does not halt on input $X$
- Use $P_{\text{halt}}$ as a subroutine in another program, $P_{\text{self}}$.
- Description of $P_{\text{self}}$:
  - Input: code for any program $P$
  - Constructs pair $(P,P)$ and calls $P_{\text{halt}}$ on $(P,P)$
  - Returns same answer as $P_{\text{halt}}$
$P_{self}$

The diagram shows a function $P_{self}$ that takes an input $P$ and outputs $(P,P)$, which is then processed by another function $P_{halt}$. The output of $P_{halt}$ is:

- 1 if $P$ halts on input $P$
- 0 if $P$ doesn't halt on input $P$
Halt is Uncomputable

- Now use $P_{\text{self}}$ as a subroutine inside another program $P_{\text{diag}}$.
- Description of $P_{\text{diag}}$:
  - input: code for any program $P$
  - call $P_{\text{self}}$ on input $P$
  - if $P_{\text{self}}$ returns 1 then go into an infinite loop
  - if $P_{\text{self}}$ returns 0 then output 0
- $P_{\text{diag}}$ on input $P$ does the opposite of what program $P$ does on input $P$
$P_{\text{diag}}$
Halt is Uncomputable

- Review behavior of $P_{\text{diag}}$ on input $P$:
  - If $P$ halts when executed on input $P$, then $P_{\text{diag}}$ goes into an infinite loop
  - If $P$ does not halt when executed on input $P$, then $P_{\text{diag}}$ halts (and outputs 0)

- What happens if $P_{\text{diag}}$ is given its own code as input? It either halts or doesn't.
  - If $P_{\text{diag}}$ halts when executed on input $P_{\text{diag}}$, then $P_{\text{diag}}$ goes into an infinite loop
  - If $P_{\text{diag}}$ doesn't halt when executed on input $P_{\text{diag}}$, then $P_{\text{diag}}$ halts
Halt is Uncomputable

- What went wrong?
- Our assumption that there is an algorithm (program) to compute Halt was incorrect.
- So there is no algorithm that can correctly determine if an arbitrary program halts on an arbitrary input.
Undecidability

The analog of an uncomputable function is an **undecidable set**.

The theory of what can and cannot be computed focuses on identifying sets of strings:

- an algorithm is required to "decide" if a given input string is in the set of interest
- similar to deciding if the input to some NP-complete problem is a YES or NO instance
Undecidability

- Recall that a (formal) language is a set of strings, assuming some encoding.
- Analogous to the function Halt is the set $H$ of all strings that encode a program $P$ and an input $X$ such that $P$ halts when executed on $X$.
- There is no algorithm that can correctly identify for every string whether it belongs to $H$ or not.
More Reductions

- For NP-completeness, we were concerned with (time) *complexity* of problems:
  - reduction from P1 to P2 had to be fast (polynomial time)
- Now we are concerned with *computability* of problems:
  - reduction from P1 to P2 just needs to be computable, don't care how slow it is
Many-One Reduction

all strings over $L_1$'s alphabet

all strings over $L_2$'s alphabet

$\text{f}$
Many-One Reduction

- YES instances map to YES instances
- NO instances map to NO instances
- computable (doesn't matter how slow)
- Notation: $L_1 \leq_m L_2$
- Think: $L_2$ is at least as hard to compute as $L_1$
Many-One Reduction Theorem

**Theorem:** If $L_1 \leq_m L_2$ and $L_2$ is computable, then $L_1$ is computable.

**Proof:** Let $f$ be the many-one reduction from $L_1$ to $L_2$. Let $A_2$ be an algorithm for $L_2$. Here is an algorithm $A_1$ for $L_1$.

- **input:** $x$
- **compute** $f(x)$
- **run** $A_2$ on input $f(x)$
- **return** whatever $A_2$ returns
Implication

- If there is no algorithm for $L_1$, then there is no algorithm for $L_2$.
- In other words, if $L_1$ is undecidable, then $L_2$ is also undecidable.
- Pay attention to the direction!
Example of a Reduction

- Consider the language $L_{NE}$ consisting of all strings that encode a program that halts (does not go into an infinite loop) on at least one input.

- Use a reduction to show that $L_{NE}$ is not decidable:
  - Show some known undecidable language $\leq_m L_{NE}$.
  - Our only choice for the known undecidable language is $H$ (the language corresponding to the halting problem).
  - So show $H \leq_m L_{NE}$. 
Example of a Reduction

Given an arbitrary $H$ input (encoding of a program $P$ and an input $X$ for $P$), compute an $L_{NE}$ input (encoding of a program $P'$) such that $P$ halts on input $X$ if and only if $P'$ halts on at least one input.

Construction consists of writing code to describe $P'$.

What should $P'$ do? It's allowed to use $P$ and $X$.
Example of a Reduction

- The code for P' does this:
  - input X':
  - ignore X'
  - call program P on input X
  - if P halts on input X then return whatever P returns

- How does P' behave?
  - If P halts on X, then P' halts on every input
  - If P does not halt on X, then P' does not halt on any input
Example of a Reduction

- Thus if $(P,X)$ is a YES input for $H$ (meaning $P$ halts on input $X$), then $P'$ is a YES input for $L_{NE}$ (meaning $P'$ halts on at least one input).

- Similarly, if $(P,X)$ is a NO input for $H$ (meaning $P$ does not halt on input $X$), then $P'$ is a NO input for $L_{NE}$ (meaning $P'$ does not halt on even one input).

- Since $H$ is undecidable, and we showed $H \leq_m L_{NE}, L_{NE}$ is also undecidable.
Generalizing Such Reductions

- There is a way to generalize the reduction we just did, to show that lots of other languages that describe properties of programs are also undecidable.

- Focus just on programs that accept languages (sets of strings):
  - I.e., programs that say YES or NO about their inputs
  - Ex: a compiler tells you YES or NO whether its input is syntactically correct
Properties About Programs

- Define a property about programs to be a set of strings that encode some programs.
  - The "property" corresponds to whatever it is that all the programs have in common

Example:
- Program terminates in 10 steps on input y
- Program never goes into an infinite loop
- Program accepts a finite number of strings
- Program contains 15 variables
- Program accepts 0 or more inputs
Functional Properties

A property about programs is called functional if it just refers to the language accepted by the program and is not about the specific code of the program.

- Program terminates in 10 steps on input y not functional
- Program never goes into an infinite loop functional
- Program accepts a finite number of strings functional
- Program contains 15 variables not functional
- Program accepts 0 or more inputs functional
Nontrivial Properties

A functional property about programs is **nontrivial** if some programs have the property and some do not.

- Program never goes into an infinite loop **nontrivial**
- Program accepts a finite number of strings **nontrivial**
- Program accepts 0 or more inputs **trivial**
Rice's Theorem

- Every nontrivial (functional) property about programs is undecidable.
- The proof is a generalization of the reduction shown earlier.
- Very powerful and useful theorem:
  - To show that some property is undecidable, only need to show that is nontrivial and functional, then appeal to Rice's Theorem
Applying Rice's Theorem

Consider the property "program accepts a finite number of strings".

This property is functional:
- it is about the language accepted by the program and not the details of the code of the program

This property is nontrivial:
- Some programs accept a finite number of strings (for instance, the program that accepts no input)
- some accept an infinite number (for instance, the program that accepts every input)

By Rice's theorem, the property is undecidable.
Implications of Undecidable Program Property

- It is not possible to design an algorithm (write a program) that can analyze any input program and decide whether the input program satisfies the property!

- Essentially all you can do is simulate the input program and see how it behaves
  - but this leaves you vulnerable to an infinite loop