Computers: What can't they do!
Computers: What can't they do?

Part 5: Undecidability

April 23–24, 2012

CS 60: Principles of Computer Science

Vocabulary for Decision Problems

Decidable

- There is a program/algorithm/TM that always *gives* correct answers in finite time.
- Such a program is often called a *checker*

Undecidable

- The problem is not *decidable*.

Semidecidable

- There is a program/algorithm/TM that never *gives* a wrong answer.
- *And* if the answer is “yes,” this will be reported.

Manual Halt Checking

Do these functions halt (or stop or return) on the input `60`?

```
def P1(w):
  if (w == 0):
     return
  else:
     P1(w-1)
```

```
def P2(w):
  if (w == 0):
     return
  else:
     P2(w+1)
```

Building a Halt Checker

```
def HC(P, w):
  """Takes a program (function) P, and an input w,
     and returns True or False depending on whether
     P(w) would terminate if started."""
  ...put clever code here...
```

E.g., `HC(P1,60) == True` *and* `HC(P2,60) == False`.

Is it weird that our function takes *code* as input?

Is halt-checking at least *semidecidable*?

Is halt-checking obviously undecidable?

A Halt-Checker Would be Useful!

Goldbach's conjecture

From Wikipedia, the free encyclopedia

Goldbach's conjecture is one of the oldest unsolved problems in number theory and in all of mathematics. It states:

> Every even integer greater than 2 can be expressed as the sum of two primes.[1]

```
def goldbachCE(n):
  for i in range(2,n-1):
    if prime(i) and prime(n-i):
      goldbachCE(n+2)
  return n

print HC(goldbachCE, 4)
```

Halting is Undecidable

Suppose a Halt-Checker program exists:

```
def HC(P, w):
  """Takes a program (function) P, and an input w,
     and returns True or False depending on whether
     P(w) would terminate if started."""
  ...put clever code here...
```

Our plan: Proof by Contradiction.

- ✓ Construct a new program that uses this HC as a subroutine
- ✓ Show the new program cannot possibly exist.

Halting is Undecidable

```
def cant(P):
  #                KEY IDEA
  # We can write any code we want here
  # *and* we can call HC as a helper function
  # that is guaranteed to work correctly.
```

The Function `cant`

```
def cant(P):
  if HC(P, P):
    while True: pass   # infinite loop
  else:
    return 60
```

```
def lt5000(s):
  if len(s) < 5000:
    return len(s)
  else
    return lt5000(s + "++")
```

```
def lt5(s):
  if len(s) < 5:
    return len(s)
  else
    return lt5(s + "++")
```

What does `cant(lt5000)` *do?*

What does `cant(lt5)` *do?*

Halting is Undecidable

```
def cant(P):
  if HC(P, P):
     while True: pass   # infinite loop
  else:
     return 60
```

Does `cant(cant)` *go* into an infinite loop?

Does `cant(cant)` terminate?

Are there any *practical* reasons to run code
and *give* it its own program as input?

Beyond Basic Halting

Now that we know that Halting is not decidable, we can use this to show other problems are undecidable.

Reduction from Halting: Show that *if* we could solve some problem P, then we could use that solution to build a Halt Checker **HC**!

Hence, P is not decidable either.

The No-Input Halting Problem

Suppose we consider only programs that take no input (equivalently, TMs started on a blank tape). Can we determine whether `these` halt?

```
def NIHC(P):
  """Returns True if P() would halt, and
     returns False if not."""
  ...put clever code here...
```

Plan: Show that we can't write `NIHC`, by showing this helper function would let us write `HC`!

No-Input Halting is Undecidable

```
def HC(P, w):
    """Returns True if P(w) halts; False otherwise"""

    def Q():
        P(w)

    return NIHC(Q)
```

To verify:

if `NIHC` *always returns a correct answer, would* `HC` *always return a correct answer?*

TM-Style Writeup, for the Record

We show that a blank-tape-halt-checker can not exist by reduction from the Halting Problem.
Assume that a blank-tape-halt-checker BT(P) does exist. We build a program HC(P,w), which takes a program P and a string w as input, as follows:

1. Build a Turing Machine that takes no inputs. It first writes the string w to its blank tape, and then runs P on that tape. Call this no-input turing machine TM. Note that TM effectively runs P on w.
2. Call our blank-tape-halt-checker on this TM: BT(TM)
3. If BT(TM) reports that TM halts, halt and output "Yes"
4. If BT(TM) reports that TM does not halt, halt and output "No"

As long as BT exists, this constructed HC is a legitimate program. All of the steps are computable: writing a single, known string to a blank tape, running a turing machine's program, and conditional-checking. However, note that HC(P,w) is a halt checker!

- ✓ If P halts on w, HC(P,w) returns "Yes" (because TM will halt on no input, so BT(TM) returns "Yes")
- ✓ If P does not halt on w, HC(P,w) returns "No" (because TM will not halt on no input, so BT(TM) returns "No")

Since a halt-checker can not exist, we have reached a contradiction. Thus, our original assumption that the blank-tape-halt-checker exists was false. A blank-tape-halt-checker also can not exist.

Name: "Quiz"

To Show: All-Input Halting is undecidable, by reduction from Halting.
Suppose we have a solution

```
def AIHC(P):
  """Returns True if P(x) halts for *every* input x;
     returns False otherwise."""
  ...put clever code here...
```

Show a contradiction, that we could use `AIHC` to write a Halt Checker:

```
def HC(P, w):
  """Returns True if P(w) halts;
     returns False otherwise"""
  def Q(x):        # Goal: Q halts on all inputs iff P(w) halts




  return AIHC(Q) # Don't run Q; just check it!
```

NAME:

"QUIZ **EXTRA**"

To show: CS5 auto-grading is undecidable, by reduction from Halting.

Suppose we have a solution

```
def EQ(P, SampleSolution):
  """Returns True if P and SampleSolution do the same thing
     for all inputs; returns False otherwise."""
  ...put clever code here...
```

Show a contradiction, that we could use `EQ` to write a Halt Checker:

```
def HC(P, w):
  """Returns True if P(w) halts;
     returns False otherwise"""
  def Q1(x):



  def Q2(y):



  return EQ(Q1, Q2)       # Goal: Q1, Q2 same iff P(w) halts
```

Rice's Theorem

Theorem

No nontrivial property of a program's input/output behavior is decidable.

Full Employment Theoerem for Compiler Writers

Theorem

There is no perfect size-optimizing compiler.

Proof.

Any program that infinite loops without output could be identified, as it would reduce to a single loop instruction:

```
L1:   jmp L1
```

□

Perfect Garbage Collection is Undecidable

Java, Python, Haskell, Scheme, *etc.*, all rely on *garbage collection* to *deallocate* unused memory.

- ✓ At any point during *execution*, a *piece of data* is *live* it it will be used in the future, and *otherwise dead* or *garbage*.
- ✓ A *garbage collector detects* and *deallocates garbage*.
- ✓ *Perfect* garbage collection is undecidable.

Kolmogorov Complexity

Fix a programming language. For each natural number $n > 0$, define

$k(n) :=$ length of the smallest program that prints the number n

In Python, we know that

$$k(n) \leqslant \lceil \log_{10} n \rceil + 8$$

Note:

$$k(10000000000000000000000000000...00000)$$
$$<<$$
$$k(17011768420006872822488857 7...85601)$$

$k(n)$ is Uncomputable!

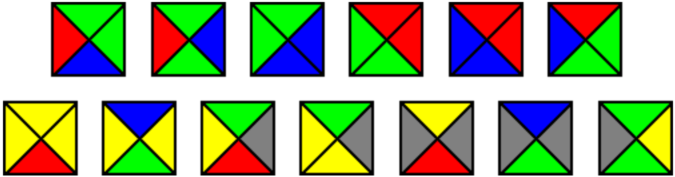
```
# Suppose this function exists, always works, and
# is, say, 42,000 characters long.
def k(n):
  """Returns the length of the shortest Python program
     that prints the number n"""
  ...put clever code here...

def ouch():
  x = 0
  while k(x) < 50000:
     x += 1
  return x
```

What's wrong?

Tiling

Given a set of template tiles, can you cover the plane with them?

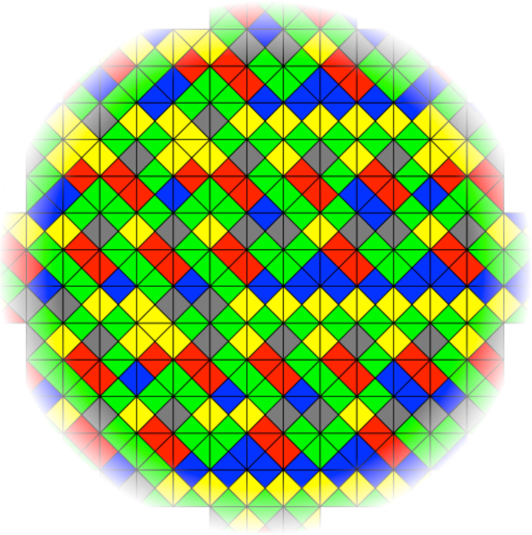
- ✓ Constraints: *edge colors must match, no rotations*



For this set: yes, but not periodically (Proved in 1996.)

For an arbitrary set of tiles: undecidable.

Tesselation



http://en.wikipedia.org/wiki/File:Wang_tesselation.svg

You Never Know What Will Be Useful!

Wang Tiles for Image and Texture Generation

Michael F. Cohen[1] Jonathan Shade[2,3] Stefan Hiller[4] Oliver Deussen[4]

[1]Microsoft Research [2]WildTangent [3]University of Washington [4]Dresden University of Technology

Abstract

We present a simple stochastic system for non-periodically tiling the plane with a small set of Wang Tiles. The tiles may be filled with texture, patterns, or geometry that when assembled create a continuous representation. The primary advantage of using Wang Tiles is that once the tiles are filled, large expanses of non-periodic texture (or patterns or geometry) can be created as needed very efficiently at runtime.

means to overcome this problem is to create (or capture) a small example of complexity and then reuse this example many times. Unfortunately, when the same example is used many times in a periodic fashion, the repetition is often apparent and distracting.

We present a new stochastic algorithm to non-periodically tile the plane with a small set of Wang Tiles [Wang 1961; Wang 1965]. This allows Wang Tiles to share the efficiency of reusing example tiles to create large expanses of complex texture, patterns, or pre-lighted geometry at runtime, while avoiding the obvious visual artifacts of

SIGGRAPH 2003

Wang Tiles, Applied

Wang Tiles, Applied

artifacts from regular tiling:

vs. more natural result from aperiodic Wang tilings: