Is it Going? Exploring applications for Turing's Halting Problem

Lauren Leung / laurenle@seas.upenn.edu

18 November 2016

The Halting Problem is a computability theory problem that attempts to determine whether a computer program will halt and return for a given input, or whether it will run forever. Although the Halting Problem is frequently associated with computer scientist Alan Turing, who proved the problems undecidability in a 1936 paper, the mathematician Alonzo Church proved the problems undecidability months prior using Lambda calculus.

Computability problems can be classified as decidable, undecidable-but-recognizable, or unrecognizable. A decidable problem can always be verified or debunked, such as the statement "there is a blonde student sitting in DRL 3C2 right now:" quickly scanning the finite number of students sitting in the room will tell the observer whether the statement is true or false. A recognizable-but-undecidable problem is one that can always be verified as true for a given input, but may not always produce a clear negative. An example of this problem is the statement "at some point, there will be a MATH 320 student with blonde hair." If a student looks around and sees someone in class has blonde hair, she can immediately report that the statement is

true. If there is no one in the classroom with blonde hair, she may report that the statement is true as soon as a blonde student joins the class, or she may continue to await a blonde MATH 320 student indefinitely (assuming that the course listing remains until the end of the time). The Halting Problem belongs to such a class of recognizable but undecidable problems.

The Halting Problem can be extended to a number of applied scenarios. There are many circumstances in which a programmer might like to know whether a program will halt. Many iterative algorithms depend on a user-supplied upper bound on maximum iterations, but if the programmer knew with certainty that the algorithm would halt on its own, these algorithms could obtain more precise answers. The application of the halting problem is visible in the Busy Beaver Problem, which determines the program that outputs the most data and halts for a fixed program size n, and is undecidable. To demonstrate the Busy Beaver Problem formally, it is easiest to do so using Turing Machines, an abstract computational model that has the same computational capabilities as any computer or programming language.

I have included code for the Busy Beaver problem at the end of this paper. Using a module for Turing Machine objects I found at Python-Course, I wrote the logic for a bounded Busy Beaver solver. A Turing Machine consists of several important components: a set of states, an initial state, a final state, and transition functions that map a *(state, read-value)* pair to a *(state, write-value, direction)* tuple. I assumed that the initial state and final state were fixed, and used combinatorics to generate a list of all possible transition functions. After the possible transition functions are generated, I simulate each Turing machine until it has exceeded a user-specified number

of iterations, and check whether the number of ones written exceeds the previous maximum. The program is very slow, since the number of possible transition functions grows rapidly according to the number of states; future implementations may take advantage of a faster language like C.

The pseudocode for this implementation is as follows:

- check whether user inputs n and max_steps are positive
- given states s numbered [0..n-1], where the initial_state=0 and final_state=(n - 1), generate possible mappings from states [0..n-2] to states [0..n-1]
- for each mapping t:
  - create a turing machine tm using s, t
  - simulate tm on a tape of zeros
  - if the number of iterations surpasses max_steps, return -1. else, return the number of ones written to the tape.
- return the maximum number of ones generated by the for loop

How might the halting problem benefit this program? The Busy Beaver problem requires that a solution eventually halts and does not write an infinite stream of data. Building Turing Machines in such a random fashion means many of these machines may never reach their final state, and time is wasted when the program must iterate through such a large number of garbage Turing Machines. A solution to the Halting Problem could greatly reduce execution time, and would also reduce the error introduced by a user-selected max_steps value. Instead, the programmer must decide

whether to risk the program running forever or potentially generating an erroneous response. This brute-force method can only be optimized within limitations: While it may be easy to manually detect whether a randomly constructed two-state Turing Machine will halt, the Halting Problem indicates that any universal heuristic falls apart as the number of states increase.

By nature of its undecidable classification, an analyst may not always know whether a given program will halt, but developments in computability theory have made it possible for some cases. Turing himself published a paper on evaluating for-loops for termination, and recent pattern-matching algorithms demonstrate great potential in this strategy. Ranking functions that pattern a variable's transformation over time are not only used to prove termination, but also help solve the difficult problem of determining how many times a section of code will be reached during execution. Unfortunately for our busy beavers, the utility of these tools is relegated to certain edge scenarios, and determining whether a program will halt frequently requires the same divinity as predicting a blonde future classmate.

```
#!/usr/bin/python

import sys
from itertools import *
from turing_machine import TuringMachine

directions = "LRN"
readable = "01 "
writable = "1"
blank = " "
nop = "N"

# generate ordered list of possible transitions for n-state turing machine
def generate_transitions(states):
  # assume last state to be final state
  possible_transition_inputs = list(product(states[0:-1], readable))
  possible_transition_outputs = list(product(states, writable, directions))

  possible_transition_funcs = [
    zip(possible_transition_inputs, item) for item in product(
      possible_transition_outputs,
      repeat=len(list(possible_transition_inputs))
    )
  ]

  return possible_transition_funcs
```

```
# for a given turing machine, return the number of ones in its final tape
def get_ones(tm, max_steps):
  i = 0
  while not tm.final():
    if i >= max_steps:
      return -1
    else:
      i += 1
    tm.step()
  tape = tm.get_tape()
  return tape.count('1')

def busy_beaver(n, max_steps):
  states = list(map(str, xrange(n)))
  possible_transition_funcs = generate_transitions(states)
  max_ones = -1

  for t in possible_transition_funcs:
    tm = TuringMachine("0" * max_steps,
      initial_state = '0',
      final_states = {states[n - 1]},
      transition_function=dict(t)
    )
    ones = get_ones(tm, max_steps)
    if ones > max_ones:
      print t, ones
      max_ones = ones
```

```
  print(max_ones)
  return max_ones

# Parse commandline input and pass to busy beaver method
def main():
  n = int(sys.argv[1])
  max_steps = int(sys.argv[2])

  print 'Received arguments', str(n), str(max_steps)

  if n < 1 or max_steps < 1:
    print 'Error: n and max_steps must both be positive'
  else:
    max_ones = busy_beaver(n + 1, max_steps)
    print 'Busy beaver complete, got max value of', str(max_ones)

# Executable section of code
main()
```