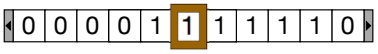
Eric Roberts
CS 54N

Handout #15
October 24, 2016

Uncomputable Functions

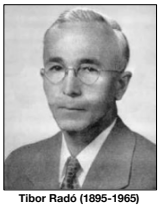
Uncomputable Functions



	0	1
1	1R2	1L0
2	0R3	1R2
3	1L3	1L1

Eric Roberts
CS 54N
October 24, 2016

The Busy Beaver Problem


Tibor Radó (1895-1965)

- Although it is possible to introduce the notion of undecidable problems using Turing's original argument involving a "universal" Turing machine, it is much easier to do so in the context of a more recent problem posed by Tibor Radó in the early 1960s:

  What is the largest finite number of **1**s that can be produced on blank tape using a Turing machine with *n* states?

- This problem is called the ***Busy Beaver Problem***.

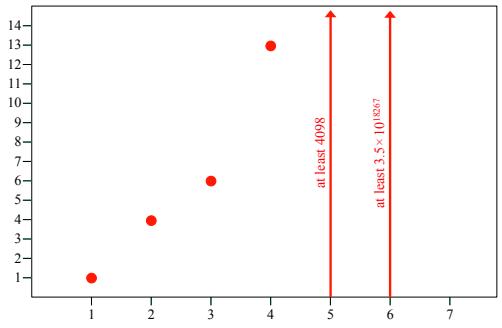
The Function BB(*n*)

- The Busy Beaver problem has a natural expression as a mathematical function. If *n* represents the number of states, let BB(*n*) represent the largest finite number of **1**s that can be written on blank tape by a machine of that size.
- For *very* small values on *n*, it is fairly easy to determine the value of the BB function:

  BB(1) = 1
  BB(2) = 4
  BB(3) = 6

- From there, the situation gets much harder. Proving that BB(4) = 13 was a Ph.D. thesis. No one is yet sure of the values for any higher number, although conjectures exist for BB(5) and BB(6).

Known Bounds for BB(*n*)



Computing BB(*n*)

- Given that BB(*n*) is a mathematical function, it makes sense to ask whether that function can be computed by a Turing machine. In other words, is there a machine M_{BB} that takes a number of **1**s representing *n* as input and writes out a number of **1**s representing BB(*n*) as output?
- It turns out that the answer is no. There is no Turing machine M_{BB} that computes the BB function. What's more, we'll be able to ***prove*** that such a function cannot be computed at all.
- The BB function is an example of an ***uncomputable function***, which is the profound new idea that Alan Turing and several of his contemporaries introduced to the mathematical world.

Observations about BB(*n*)

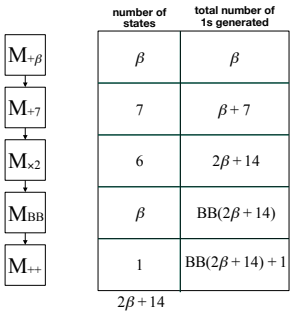
- In order to resolve the question about whether BB(*n*) can be computed by a Turing machine, it helps to make two observations about the BB function:

  1. BB(*n*) is a well-defined mathematical function. It does exist. For every number of states, the number of possible Turing machines is finite. There must be some machine that writes out at least as many **1**s as any other.
  2. BB(*n*) must be strictly increasing. With an extra state, it is always possible to write at least one more **1** than is possible with a Turing machine with fewer states.

Proof by Contradiction

- In seeking to prove that BB(n) is not computable by a Turing machine, the simplest approach is to employ a strategy called ***proof by contradiction***. In proof by contradiction, you start by assuming the opposite of what you wish to prove, and then show—typically by constructing a specific example—that doing so leads to an absurd conclusion or that violates one of the assumptions. If the steps in your construction are correct, the only questionable part of the process is the original assumption.
- Thus, to prove that BB(n) is not computable by a Turing machine, we start by assuming that it is. That means that we can assume the existence of a machine M_{BB} with β states that takes n as input and writes out BB(n) **1**s as output.
- The essence of the contradiction is to construct a machine with k states that writes out more than BB(k) **1**s.

Steps in the Proof



But Wait . . . Why Can't You . . .

- Despite the proof by contradiction, the idea that BB(n) is uncomputable seems wrong. After all, we can simulate a Turing machine. Why isn't it possible to solve this problem using the following approach:
  1. Generate every Turing machine with n states. There is only a finite number of such machines.
  2. For each machine, run the Turing machine simulator and count the number of **1**s it generates.
  3. Keep track of the largest value so far and report that number at the end of the run.
- There is a problem here. Some of the machines go on forever, so there is no way to terminate the computation in step 2.
- If it were possible to tell whether a Turing machine would halt, it would be possible to compute the BB(n) function.

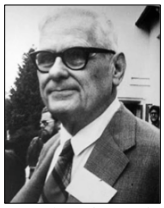
The Halting Problem in JavaScript

```
/*
 * File: Paradox.js
 * ----------------
 * This program uses the assumption that doesProgramHalt exists to
 * generate a paradox.
 */

function paradox() {
   if (doesProgramHalt("Paradox.js")) {
      Console.println("The program runs forever.");
      while (true) {
         /* Loop forever doing nothing */
      }
   } else {
      Console.println("The program halts.");
   }
}

/*
 * Reads the code stored in the named file and determines whether the first
 * function in that file halts, returning true or false accordingly.
 */

function doesProgramHalt(filename) {

}
```

The Church-Turing Thesis

- The question of what is computable by a Turing machine is important in a search for what is generally computable mostly because no one has ever found a more powerful model.
- Most computer scientists believe what has come to be known as the ***Church-Turing thesis***:

  No method of computation carried out by a mechanical process can be more powerful than a Turing machine.



Alonzo Church (1903-1995)

- This claim remains a conjecture, and it is not clear there is any way to prove it. At the same time, it has so far resisted all efforts to disprove it.

The Power of Computational Models

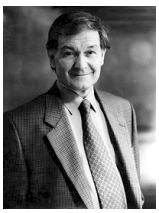
The mills of the gods grind slowly, but they grind exceedingly fine.
—Sextus Empiricus, 2nd century C.E.

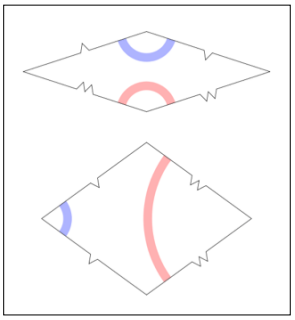
- When computer scientists talk about the power of a particular computational model, the focus is on what computations are *possible* using that model and not on questions of *efficiency.*
- Although simple models like the Turing machine run slowly, it seems that they can solve exactly the same set of problems that we can solve by any other computational model, even if that other model initially seems far more advanced.
- The usual strategy for proving that two computational models are equivalent is to come up with a strategy for transforming a arbitrary problem that uses the more advanced model into an equivalent problem that uses the less advanced one. This approach is called ***reduction***.

Tiling Problems

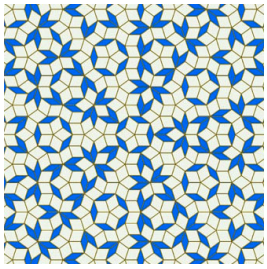
- The notion of undecidability comes up in many different contexts, most of which seem completely unrelated to the idea of Turing machines.
- One particularly interesting application appears if you try to answer the question of whether a set of tiles—constrained by rules that the colors on their edges have to match—can be positioned so that they completely fill the plane.
- Tiling problems gained some mathematical attention in the 1970s when Roger Penrose, Professor of Mathematics at Oxford developed a set of tiles that would tile the plane but only in a nonperiodic fashion.

Penrose Tilings

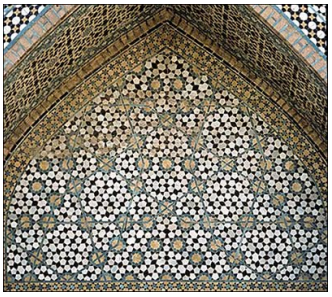




Penrose Tilings



Nonperiodic Islamic Tilings



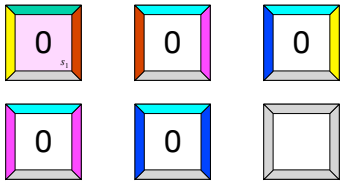
Mosaic from the Darb-i Imam shrine, Isfahan, Iran, 15c

Uncomputability of Tiling

- One of the surprising mathematical results of the 20th century is that the question of whether a set of tiles will cover the plane is uncomputable.
- The strategy for proving this result uses tiles to simulate a Turing machine, configuring the tiles in such a way that the tiles cover the complete plane only if that Turing machine runs forever starting on blank tape. Since the halting problem is undecidable, the tiling problem must be undecidable also.
- In the remaining time, I will sketch the proof of a slightly simpler result, in which the set of tiles contains a designated ***start tile*** that must occur somewhere in the pattern:

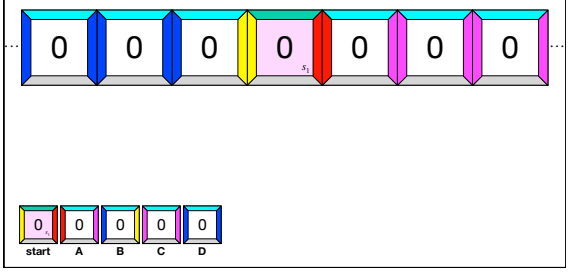
Simulating a Turing Machine in Tiles

- Suppose that your set starts with the following six tile types:

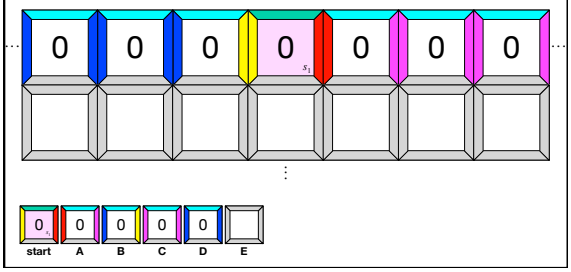


- How far could you get if you start by placing the start tile?

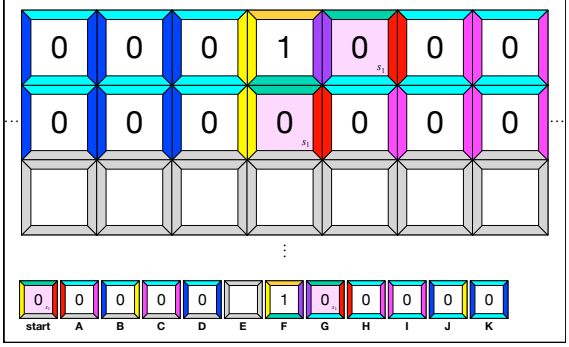
Getting Started



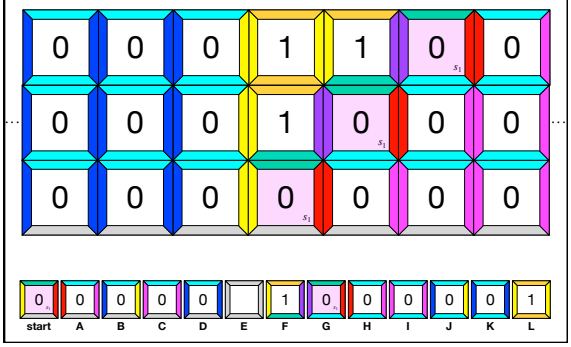
Getting Started



Simulating a **1R1** Instruction



Completing the Tiling



Removing the Origin Constraint

- Requiring a special start tile makes it easier to prove that tiling is uncomputable.
- You can eliminate the start tile by embedding the Turing machine simulation inside a Sierpinski triangle in which the computation is replicated at different scales.