Overview of Classical Computability Theory

Carl Mummert

October 25, 2016

Marshall University Combinatorics Seminar

Note

This is the first of two 50-minute talks on computability theory for the Marshall University Combinatorics Seminar in Fall 2016.

These talks are meant to be a general introduction to the area for colleagues who work in other areas of combinatorics.

The material presented is well known in the field. Although I have not included detailed references, no material here is original or due to me.

A selection of general references is provided at the end of the talk.

Introduction

The basic idea of classical computability theory is to study precise notions of computability.

These ideas can be studied for their own sake, as mathematical problems, using the same methods used in other areas of mathematics.

Ideas from computability theory can also be applied to other areas of mathematics. This will be the subject of the second part of this talk.

This part will introduce the fundamental ideas of computability theory that would be in an introductory course.

Algorithms

There have been many attempts to characterize the idea of an “algorithm”.

Motivational examples include:

- The Euclidean algorithm
- Long division
- The process of computing derivatives
- Newton’s method

To start, we just look at algorithms for functions from $\mathbb{N}$ to $\mathbb{N}$.

Later we can think about changing the domain and codomain.

Algorithms

A description of algorithms from Knuth:

- Finiteness: "An algorithm must always terminate after a finite number of steps"
- Definiteness: "Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case"
- Effectiveness: "all of the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time by a man using paper and pencil"

Basic parameters

We are interested in algorithms in the most general sense.

- Although each computation must end after some finite number of steps, we don’t require any particular bound on the number of steps.
- Although each computation must use only finitely much temporary storage space, we don’t require any particular bound on the amount of storage used.
- Although each program / algorithm must be finite, we don’t require any bound on the size of the instructions.

If we add constraints on space or running time, we enter the field of **computational complexity**.

Models of computation

A **model of computation** is a (theoretical) precise notion of computing.

Over time, numerous formal models of computation have been proposed for human-computable algorithms.

Many of these turned out to be equivalent to each other, while the remaining ones were weaker.

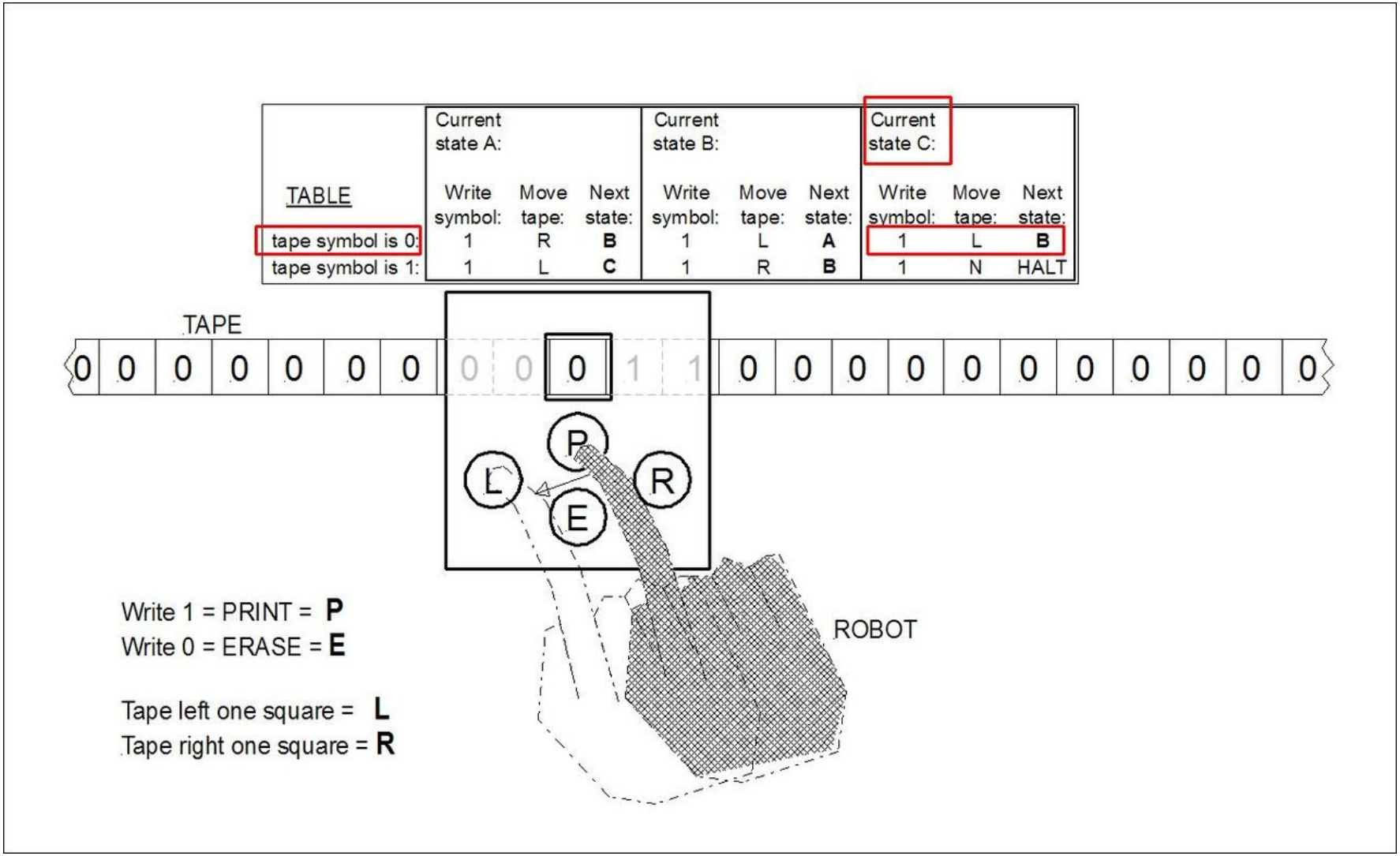
The notion of computability corresponding to the large collection the equivalent models is known as **Turing computability**.

Turing computability

Turing computability is given by

- Turing machines
- Systems for defining functions by recursion (studied by Church, Kleene, and others)
- Idealized versions of familiar programming languages (Java, C, Python, etc)
- Some very simple programming languages (WHILE)
- Idealized simple CPUs (register machines)

Turing machines



https://en.wikipedia.org/wiki/File:Busy_Beaver_1.JPG

The WHILE language

The language has variables $v_1, v_2, \ldots$ which can hold arbitrary natural numbers.

Programs are built from a small list of basic commands:

- LET $v_i = 0$
- LET $v_i = v_j$
- LET $v_i = v_j + 1$
- IF $v_i = v_j$ THEN $\{\ldots\}$ ELSE $\{\ldots\}$
- WHILE $(v_i > 0)$ $\{\ldots\}$ ENDWHILE

To compute a function, we put its inputs into v_1, v_2, ..., and wait for the program to halt. If it halts, the value left in v_1 is the output.

Subtracting 1 in WHILE

v_1 holds the input n, with $n > 0$

All other variables start at 0.

1. LET $v_2 = v_2 + 1$
2. WHILE ($v_2 > 0$)
   - 2.1 LET $v_3 = v_3 + 1$
   - 2.2 IF ($v_3 = v_1$) THEN LET $v_2 = 0$ ELSE LET $v_4 = v_4 + 1$
3. ENDWHILE
4. LET $v_1 = v_4$

When the program ends, v_1 holds the value $n - 1$.

Computable functions

Many functions are Turing computable:

- Multiplication, exponentiation, etc.
- $f(n) =$ the nth prime
- $f(n, m) =$ the exponent of the nth prime in the prime factorization of m

Some computable functions are **partial**: some inputs do not lead to an output value.

- $f(n) =$ the first step when the Collatz iteration of n reaches 1, undefined if the iteration never reaches 1

The Church–Turing thesis

> ***Church–Turing thesis****: Any function from* $\mathbb{N}$ *to* $\mathbb{N}$ *that is algorithmically computable by a human being is a Turing computable function in the formal sense.*

There is no general agreement about whether this is:

- An informal statement that is not suitable for proof
- A definition of computability
- A theorem that can be proved

Computable sets

So far I have mentioned computability of **functions** only.

We say that a set B of natural numbers is **computable** if its characteristic function 1_B is a (total) computable function.

This means there is an algorithm to decide whether arbitrary numbers are members of the set.

A set C is **computably enumerable** if it is the range of a computable function.

Coding programs as numbers

In any particular formal model of computation, there will be only countably many programs.

We can represent these as natural numbers, for example by using prime powers to encode the binary form of the program as a giant natural number.

A natural number representing a program is called a **index** of the program.

The function computed by index e is written φ_e.

Universal programs

In each model of computation, there will be a particular **universal computable function** $U(e, n)$ so that

$$U(e, n) = \varphi_e(n).$$

If $\varphi_e(n)$ doesn't halt for some n, then $U(e, n)$ also doesn't halt.

The program U is essentially just an interpreter: it decodes the instructions in e and then simulates them with input n.

This is certainly something a human could do, and it is doable in all Turing complete models of computation.

Universal programs and partiality

The existence of universal programs forces our systems to have some partial computable functions.

Consider a program that does the following:

Given input e, return $U(e, e) + 1$.

This program will have some index k.

But then $\varphi_k(k)$ cannot halt, or else

$$\varphi_k(k) = U(k, k) = \varphi_k(k) + 1.$$

The **Halting problem** is the set

$$K = \{e : U(e, e) \text{ halts}\}.$$

Halting problem

The **Halting problem** is the set $K = \{e : U(e, e) \text{ halts}\}$.

This set is not computable. If it was, we could make a program that does the following:

> *On input e, first determine whether $e \in K$. If so, then go into an infinite loop. If $e \notin K$ then immediately halt and return* 0.

That program would have some index k.

But then we have that $U(k, k)$ halts if and only if $U(k, k)$ doesn't halt, a contradiction.

Turing used this argument in 1936 to solve the *Entscheidungsproblem* posed by Hilbert.

Uncomputable problems

There are many known undecidable problems:

- Given a finite presentation of a finite group, tell whether the group is trivial.
- Given a finite set of $n \times n$ integer matrices for some n, determine whether some product of copies of the matrices is zero. (Uncomputable even when restricted to pairs of 15×15 integer matrices.)
- Given a system of multivariate polynomial equations over $\mathbb{Q}$, determine whether there is a solution in $\mathbb{Q}$. (Hilbert's 10th problem.)

Oracle computation

So far we have only looked at programs that take natural numbers as inputs.

We can modify our notion of computability to allow for computable functions to take an arbitrary **set** of natural numbers as an input.

Turing called these sets “oracles”, because we assume we already have some way to tell whether numbers are in the input set.

Oracles in WHILE

We can do this with a new instruction in our WHILE programming language:

- IF $v_n \in A$ THEN ... ELSE ...

To run the program, we now need to have a set A in mind when we begin.

When we encounter a new instruction, we check whether the number stored in v_n is in A, and proceed accordingly.

The set A is not part of the program, and the same program can be run with any set of numbers as the input set A.

Oracle computation

We write φ_e^A for the computable function with index e and input set A.

We say that a set B is **computable relative to** A if the characteristic function of B is computable using an oracle for A.

We write $B \leq_T A$ for this relationship, which is called **Turing reducibility**.

The $\leq_T$ relation is reflexive and transitive.

Universal oracle machines

There is a universal function $U^A(e, n)$ for programs with an oracle set A.

The universal function is the same for all sets – the set is not part of the program.

Ordinary computability is the special case where the set A is computable.

Relativized halting problem

For each set A of natural numbers, we can define the set

$$K^A = \{e : U^A(e, e) \text{ halts}\}.$$

For each set $A \subseteq \mathbb{N}$, we have $A \leq_T K^A$, and $K^A \not\leq_T A$.

So, by iterating the map $A \mapsto K^A$ starting with a computable set, we can produce a sequence of sets each of which is "more uncomputable" than the previous one.

Turing degrees

We define $A \equiv_T B$ if $A \leq_T B$ and $B \leq_T A$.
We call this relation **Turing equivalence**.

The equivalence classes are called **Turing degrees**.

The set of these equivalence classes, $\mathcal{D}$, is also partially ordered by $\leq_T$.

Many structural properties of $(\mathcal{D}, \leq_T)$ are known:

- There are minimal noncomputable Turing degrees.
- There is a set of $2^{\aleph_0}$ pairwise incomparable Turing degrees.
- Every countable partially ordered set embeds into $(\mathcal{D}, \leq_T)$.

Research directions

There are many research directions that flow from these concepts.

- Study the structure of the Turing degrees, both globally and locally
- Study the structure of computably enumerable sets, and the Turing degrees of such sets
- Apply computability methods to study other areas of mathematics

The second part of this talk will focus on the third item.

References

Classical computability theory

- Herbert B. Enderton, *Computability Theory: An Introduction to Recursion Theory*, Academic Press, 2010.
- Richard Soare, *Computably Enumerable Sets and Degrees*, Springer, 1987.

Turing's seminal paper

- Alan M. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem", *Proc. London Mathematical Society* 42, 1937, 230–265.

On the Church–Turing thesis

- Richard Soare, "Computability and Recursion", *Bull. Symbolic Logic* 2, 1996, 284–321.