Chap 3-4: Outline

Turing computability and uncomputability

- The concept of Turing machines
- The concept of Turing computability, *i.e.*, a function is computable by a Turing machine
- Examples of Turing uncomputable functions: the halting function, *et al.*

The intuitive notion of effective computability

- In Lecture 1, we gave a rough proof that the halting problem is not solvable by an algorithm.
- In this course, you will also learn that deciding if an first-order formula is satisfiable is not solvable by an algorithm.
- So what does it mean that a problem is solvable by an algorithm?
- Here a problem is not formally defined. Instead we consider a function from $\mathbf{Z}^+$ to $\mathbf{Z}^+$.
- We will define the notion that a function from $\mathbf{Z}^+$ to $\mathbf{Z}^+$ is computable by an algorithm, or effectively computable.

The intuitive notion of effective computability (2)

An intuitive definition

A function f from $\mathbf{Z}^+$ to $\mathbf{Z}^+$ is effectively computable if a list of instructions can be given that in principle make it possible to determine the value $f(n)$ for any argument n.

Some remarks

- The instructions must be performable by some mechanical device.
- We ignore practical limitations such as time and space limitations, and work with an idealized notion of computability that goes beyond what actual machines can do.
- We will prove that certain functions are uncomputable, even if practical limitation can be overcome.

The issue of notation system

- "determine the value $f(n)$ for any argument n"
- In fact, we are not given n, we are given a representation of n; likewise, we are not producing $f(n)$, we are producing a representation of $f(n)$
- In the course of human history, many systems of representation of numbers have been developed, *e.g.*, monadic, binary, decimal, and Roman
- Does the notation system make a difference in the definition of computability?

The issue of notation system (2)

- Computations can be harder in practice with some notations than with others
- But for all notation systems, there are explicit rules for translating among them
- Thus if a function is computable in one notation, it is also computable in another notation

Turing computability

- Despite all the explanation, the notion of effective computability remains an intuitive one, not a formal one.
- We now introduce the formal definition that a function is computable by a Turing machine.
- A specific kind of idealized machines for carrying out computations on positive integers in monadic notation

Alan Turing (1912-1954)

- An English mathematician, logician, cryptanalyst, and computer scientist
- The father of computer science and artificial intelligence
- FRS – Fellow of the Royal Society of London
- Turing Award: the Nobel Prize of computing
- Turing test: a test of a machine's ability to exhibit intelligent behavior equivalent to that of a human
- 2012: Then Alan Turing Centenary Conference

Alan Turing

- During World War II, Turing worked at Bletchley Park, Britain's codebreaking centre.
- His work saved thousands of British lives
- In 1945, Turing was awarded the OBE: the Most Excellent Order of the British Empire
- TV Movie: Britain's Greatest Codebreaker (2011), 62min
- Another TV movie: Code-Breakers: Bletchley Park's Lost Heroes (2011), 60 min

Turing machines – the tape

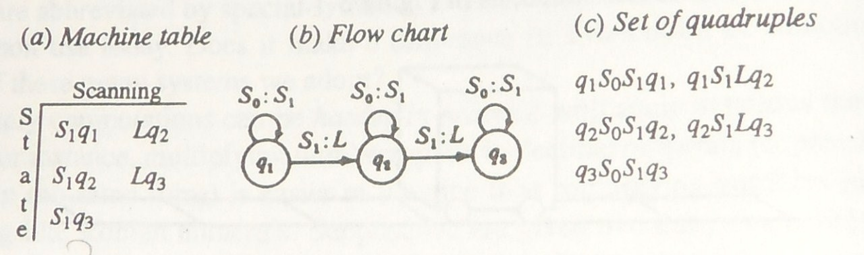
- A tape, marked into squares, unending in both directions
- Each square is either blank (denoted by $S_0/0/B$), or has a stroke printed on it (denoted by $S_1/1/|$)
- At each stage of the computation, all squares are blank with at most a finite number of exceptions

Turing machines – moves and states

- At each step of the computation, the machine is scanning one square of the tape
- It can erase a stroke or print a stroke in the scanned square
- It can move one square to the left or to the right
- It can also halt the computation
- At each step of the computation, it is in one of a finite number of internal states

Turing machines – the program

The program of the machine can be specified in different ways, *e.g.*, a machine table, a flow chart, or a set of quadruples



What does the machine T_1 do?

Turing machines – configurations

- One configuration for each stage of the computation, showing what's on the tape, the state of the machine, and which square is being scanned
- We denote a configuration by writing what's on the tape and writing the state under the symbol being scanned or as a subscript of the symbol being scanned, *e.g.*, $1_2100111$
- When writing out what's on the tape, for the infinitely many blanks on both ends, we include a finitely many, and omit the others
- Write out the configurations of T_1: ...

Example: determining the parity

- Initial tape: 1^p, where $p \in \mathbf{Z}^+$, scanning first 1
- If p is even, ending tape: 0
- If p is odd, ending tape: 1, scanning this 1

Example: adding in monadic notation

- Initial tape: $1^p 0 1^q$, where $p, q \in \mathbf{Z}^+$, scanning first 1
- Ending tape: 1^{p+q}, scanning first 1

Example: doubling the number of strokes

- Initial tape: 1^p, scanning first 1
- Ending tape: 1^{2p}, scanning first 1
- General idea: Repeatedly writing 11 on the left and erasing 1 on the right
- At any stage of the computation: $1^{2(p-q)}01^q$, where $q \leq p$
- Refinement of the general idea:
  Repeat
      add 11 to LHS of $1^{2(p-q)}$, and erase 1 from RHS of 1^q
      if $q = 0$ then move to the beginning of 1^{2p}, and halt
          else move to the beginning of $1^{2(p-q)}01^q$

Our refined algorithm

1 move left

2 move left, write 11 leftward, and move right

5 find the second 0 on the right, and move left

7 write a 0, and move left

8 if the current symbol is 0, then move left, and goto step 11
else move left

9 find the first 0 on the left, move left

10 find the first 0 on the left, move right, and goto step 2

11 find the first 0 on the left, and move right

Converting this algorithm to a flow chart

Example: multiplying in monadic notation

- Initial tape: $1^p 0 1^q$, where $p, q \in \mathbf{Z}^+$, scanning first 1
- Ending tape: $1^{p \cdot q}$, scanning first 1
- At any stage of the computation: $1^r 0 0^{q \cdot (p-r-1)} 1^q$
- Our algorithm:
  Repeat
  erase 1 from LHS of 1^r
  if $r = 0$, then move right two squares,
  change all 0 to 1 until we see a 1, and halt
  else move 1^q q places to the right

How to move 1^q q places to the right

- this is similar to doubling the number of strokes
- Repeatedly erasing 1 on the left and writing 1 on the right
- At any stage of the computation: $0^r 1^{q-r} 0 1^r$, and when we erase the last 1, we write 1 in the next square
- This is left as an exercise
- You will get the sub-flow-chart of Figure 3-7 consisting of states 4 to 11

Our refined algorithm

1 erase 1, and move right

2 if the current symbol is 0, then move right, and goto step 15
else move right

3 move right until we see a 0, and move right

4 move right until we see a 1, and move right

5 move 1^q q places to the right

13 move left until we see a 1, and move left

14 move left until we see a 0, move right, and goto step 1

15 replace 0s with 1s until we see a 1, and move left

17 move left until we see a 0, and move right

Converting this algorithm to a flow chart

Turing-computable functions

Let f be a numerical function of k arguments. We say that a Turing machine T computes f if

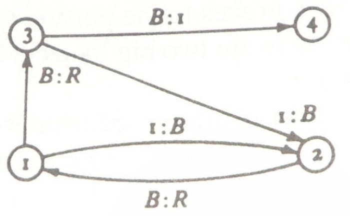
- Initial configuration: $1^{m_1}0\ldots01^{m_k}$, scanning first 1, in the lowest-numbered state
- Such a configuration is called a standard initial configuration
- If $f(m_1,\ldots,m_k)$ is defined, let the value be m, then ending configuration: 1^m, scanning first 1, in a halting state, *i.e.*, a state for which there is no instruction as what to do when scanning a symbol
- such a configuration is called a standard final configuration
- If $f(m_1,\ldots,m_k)$ is undefined, then the computation will never halt, or it will halt in a nonstandard final configuration

Examples

- Any Turing machine computes a certain k-ary function for each $k \in \mathbf{Z}^+$
- We have presented Turing machines computing the addition and multiplication functions
- Let T_2 be given by a single tuple: $q_1 1 1 q_2$
- What function does T_2 compute?
- T computes the identity function $id(m) = m$ for $m \in \mathbf{Z}^+$
- When $k \geq 2$, T computes the empty k-ary function, *i.e.*, the function which is undefined for all k-tuples

One more example

- Let T be given by



- What function does T_3 compute?
- for each k, T computes the k-ary function which assigns 1 to all k-tuples

Turing computability and effectively computability

Definition

A numerical function of k arguments is Turing computable if there is some Turing machine that computes it.

Clearly, any Turing computable function is effectively computable.

Turing's thesis

Any effectively computable function is Turing computable.

So the formal notion of Turing computability coincides with the intuitive notion of effective computability.

How can we be convinced of Turing's thesis?

The main argument is to accumulate examples of effectively computable functions which are shown to be Turing computable

- So far we have only shown that the following functions are computable: addition, multiplication, the identity function, the empty function, and the constant function
- There are many other arithmetic operations such as exponentiation
- According to Turing's thesis, they must be Turing computable
- But designing a TM for multiplication is already difficult
- We will prove they are Turing computable using a less direct approach

There exist Turing uncomputable functions

Proof

- The set of functions from $\mathbf{Z}^+$ to $\mathbf{Z}^+$ is not enumerable
- The set of Turing machines is enumerable, since a TM is a finite string from an enumerable alphabet
- Thus the set of Turing computable functions is enumerable

But can we give an explicit example of a Turing uncomputable function?

Encoding Turing machines

- The quadruple representation of T_3:
  $q_1 S_0 R q_3, q_1 S_1 S_0 q_2, q_2 S_0 R q_1, q_3 S_0 S_1 q_4, q_3 S_1 S_0 q_2$.
- We take the lowest-numbered state as the initial state, and we assume the highest numbered state is the halting state (if not, we can always add an additional state)
- We can assume that for any non-halting state q_i, there is a quadruple beginning $q_i S_j$ (if not, we can always add $q_i S_j S_j q_k$, where q_k is the halting state)
  $q_1 S_0 R q_3, q_1 S_1 S_0 q_2, q_2 S_0 R q_1, q_2 S_1 S_1 q_4, q_3 S_0 S_1 q_4, q_3 S_1 S_0 q_2$.
- We can omit the first two elements in each quadruple,
  $R q_3, S_0 q_2, R q_1, S_1 q_4, S_1 q_4, S_0 q_2$
- We represent q_i by i, S_0 by 1, S_1 by 2, L by 3, R by 4,
  $4, 3, 1, 2, 4, 1, 2, 4, 2, 4, 1, 2$
- We can get a single positive integer by encoding this sequence

Encoding Turing machines (2)

- The encoding is an injection, but not a bijection
- But we can get a bijection from the injection
- Thus we can list the TMs as $M_1, M_2, M_3, \ldots$
- We can list the Turing computable unary functions as $f_1, f_2, f_3, \ldots$, where f_i is the unary function computed by M_i

The diagonal function d is not Turing computable

- The diagonal function d is defined as follows:
  $$d(n) = \begin{cases} 2 & \text{if } f_n(n) = 1 \\ 1 & \text{otherwise} \end{cases}$$
- Then d is different from every f_n
- So d is not Turing computble

The diagonal function d is not Turing computable

- The diagonal function d is defined as follows:
  $d(n) = \begin{cases} 2 & \text{if } f_n(n) = 1 \\ 1 & \text{otherwise} \end{cases}$
- Then d is different from every f_n
- So d is not Turing computable
- By Turing's thesis, d is not effectively computable

Why is d not effectively computable

- Given n, it is routine to compute M_n
  - we find the nth integer p which encodes a Turing machine
  - we decode p and get the quadruples of M_n
  - although this process might not be feasible in practice, it is routine in principle

Why is d not effectively computable (2)

- It is routine to follow the instructions of M_n on n
- Later in this course, we will prove that any Turing computable function is computable by a TM such that if it halts, it halts in a standard configuration
- If M_n does halt on n, it is routine to check if the output is 1, so it is routine to compute $d(n)$
- If M_n does not halt on n, we should set $d(n) = 1$

Why is d not effectively computable (3)

- But is it routine to decide if M_n halts on n?
- Nobody has been able to give such a routine
- We conclude that the halting problem is not effectively computable, otherwise d is effectively computable
- Next, we give a formal proof that the halting problem is not Turing computable.

The halting function h

$h(m, n) = 1$ if M_m halts on n, and 2 otherwise

Theorem

The halting function h is not Turing computable.

Proof:

- we need a copying machine C
  - initial tape: 1^p, scanning the first 1
  - ending tape: 1^p01^p, scanning the first 1
- we need a dithering machine D
  - initial tape: 1^p, scanning the first 1
  - if $p > 1$, halts, else never halts

The halting function h (2)

- Suppose we have a machine H that computes h
- We join C and H to get G
- We now join G and D to get M
- Let m be the coding for M
- on m, M halts on m iff M does not halt on m
- Thus h is not Turing computable

By Turing's thesis, the halting function is not effectively computable.

Exercise 1

Prove that there is no algorithm which can decide if a program with a given input ever prints the digit 1

Exercise 2

Ex 4.2

- Is there a Turing machine that, started anywhere on the tape, will eventually halt iff the tape originally was completely blank?
- If so, sketch the design of such a machine;
- if not, briefly explain why not.