CS 70 Discrete Mathematics and Probability Theory
Spring 2025 Rao

DIS 7B

1 Computability Intro

Note 12

Computability: The main focus is on the Halting problem, and programs that provably cannot exist.

The *Halting problem* is the problem of determining whether a program P run on input x ever halts, or whether it loops forever. It turns out that there does not exist any program that solves this problem.

Using this information, we can prove that other problems also cannot be solved by a computer program, through the use of *reductions*. The main idea is to show that if a given problem can be solved by a computer program `TestX`, then the Halting problem can also be solved by a computer program `TestHalt` that uses `TestX` as a subroutine.

The primary template we'll use for this course is as follows. Suppose we want to show that a program `TestX` does not exist, where `TestX(Q, y)` tries to determine whether a program Q on input y does some task $\mathscr{X}$ (i.e. it outputs "True" if $Q(y)$ does the task $\mathscr{X}$, and it outputs "False" if $Q(y)$ does not do the task $\mathscr{X}$). We can define `TestHalt` as follows (in pseudocode):

```
def TestHalt(P, x):
    def Q(y):
        run P(x)
        do 𝒳
    return TestX(Q, y)  # for some given y
```

Note that this template will be sufficient for our purposes in CS70, but more complex reductions will require more sophisticated programs—you'll learn more about this in classes like CS170 and CS172.

(a) Consider the reduction template given above. Let's break down what it's doing.

We follow an argument by contradiction—we assume that there is a program `TestX(Q, y)` that is able to determine whether another program Q on input y does some task $\mathscr{X}$.

There are two cases: either $P(x)$ halts, or it loops forever. We'd like to show that `TestHalt` as defined above returns the correct answer in both of these cases.

(i) Suppose $P(x)$ halts. What does `TestHalt` return, and why?

(ii) Suppose $P(x)$ loops forever. What does `TestHalt` return, and why?

(iii) What does this tell us about the existence of `TestX`? Briefly justify your answer.

Solution:

(a) (i) If $P(x)$ halts, then $Q(y)$ will finish executing $P(x)$, and eventually do the task $\mathscr{X}$. This means that `TestX` would return "True", since $Q(y)$ does eventually do $\mathscr{X}$.

(ii) If $P(x)$ loops forever, then $Q(y)$ will get stuck while executing $P(x)$, and will never get to doing the task $\mathscr{X}$. This means that `TestX` would return "False", since $Q(y)$ never does $\mathscr{X}$.

(iii) These answers returned by `TestHalt` exactly solve the Halting problem! However, we've already shown that the Halting problem cannot be solved by a computer program—this is a contradiction. As such, `TestX` cannot exist.

2 Hello World!

Note 12

Determine the computability of the following tasks. If it's not computable, write a reduction or self-reference proof. If it is, write the program. Throughout this problem, you are allowed to execute programs while surpressing their print statements.

(a) You want to determine whether a program P on input x prints "Hello World!". Is there a computer program that can perform this task? Justify your answer.

(b) You want to determine whether a program P prints "Hello World!" while or before running the kth line in the program. Is there a computer program that can perform this task? Justify your answer.

(c) You want to determine whether a program P prints "Hello World!" in the first k steps of its execution. Is there a computer program that can perform this task? Justify your answer.

Solution:

(a) Uncomputable. We will reduce `TestHalt` to `PrintsHW`(P,x).

```
TestHalt(P, x):
  P'(x):
    run P(x) while suppressing print statements
    print("Hello World!")

  return PrintsHW(P', x)
```

If `PrintsHW` exists, `TestHalt` must also exist by this reduction. Since `TestHalt` cannot exist, `PrintsHW` cannot exist.

(b) Uncomputable. We will reduce `TestHalt` to `PrintsHWByK`(P,x,k).

```
TestHalt(P, x):
    P'(x):
        run P(x) while suppressing print statements
        print("Hello World!")
```

```
    return PrintsHWByK(P', x, 2)
```

Here, we notice that `P'` has only two lines (or at most `len(P) + 1` lines, depending on how this is implemented), and we print "Hello World!" by the last line of `P'` if and only if `P(x)` halts.

Alternatively, we can reduce `PrintsHW`(P,x) from part (a) to this program `PrintsHWByK`(P,x,k):

```
PrintsHW(P, x):
  for i in range(len(P)):
    if PrintsHWByK(P, x, i):
      return true
  return false
```

Note that we technically need to iterate through all the lines here, since there may be large jumps within the code of `P`; this means that we may for example jump from line 1 to line 100 and back to line 2 to print "Hello World!", but `PrintsHWByK`$(P,x,100)$ will return false, since we first reach line 100 without printing "Hello World!".

(c) Computable. You can simply run the program until k steps are executed. If P has printed "Hello World!" by then, return true. Else, return false.

The reason that part (b) is uncomputable while part (c) is computable is that it's not possible to determine if we ever execute a specific line because this depends on the logic of the program, but the number of computer instructions can be counted.

3 Undecided?

Let us think of a computer as a machine which can be in any of n states $\{s_1,\ldots,s_n\}$. The state of a 10 bit computer might for instance be specified by a bit string of length 10, making for a total of 2^{10} states that this computer could be in at any given point in time. An algorithm $\mathscr{A}$ then is a list of k instructions $(i_0,i_1,\ldots,i_{k-1})$, where each i_ℓ is a function of a state c that returns another state u and a number j describing the next instruction to be run. Executing $\mathscr{A}(x)$ means computing

$$(c_1,j_1)=i_0(x), \qquad (c_2,j_2)=i_{j_1}(c_1), \qquad (c_3,j_3)=i_{j_2}(c_2), \qquad \ldots$$

until $j_\ell \geq k$ for some ℓ, at which point the algorithm halts and returns $s_{\ell-1}$.

(a) How many iterations can an algorithm of k instructions perform on an n-state machine (at most) without repeating any computation?

(b) Show that if the algorithm is still running after $nk+1$ iterations, it will loop forever.

(c) Give an algorithm that decides whether an algorithm $\mathscr{A}$ halts on input x or not. Does your contruction contradict the undecidability of the halting problem?

Solution:

(a) Each of the k instruction can be called on at most n different states, therefore there are at most nk distinct computations that can be performed during any execution.

(b) Since $nk+1 > nk$, by the Pigeonhole Principle, $\mathscr{A}$ must repeat a computation $i_m(s_t)$ for some $(m,t) \in \{1,\ldots,n\} \times \{0,\ldots,k-1\}$. But we know that when $i_m(s_t)$ is performed the second time, its consective computations will be precisely the same that followed the first evaluation of $i_m(s_t)$. In particular, we will see $i_m(s_t)$ a third time, and hence a fourth, fifth time etc.

(c) From our solution to part (b) it follows that we only need to check whether after $nk+1$ iterations, $\mathscr{A}(x)$ is still running or not. If it is, $\mathscr{A}(x)$ does not halt, otherwise it does. This does not contradict the undecidability of the halting problem, since it only states the inability to decide whether an *arbitrary* algorithm halts. Here we only proved the decidability for algorithms that can be run on an n-state machine, of which there are only finitely many!

4 Code Reachability

Note 12

Consider triplets (M,x,L) where

- `M` is a Java program
- `x` is some input
- `L` is an integer

and the question of: if we execute $M(x)$, do we ever hit line L?

Prove this problem is undecidable.

Solution: Suppose we had a procedure that could decide the above; call it `Reachable(M, x, L)`. Consider the following example of a program deciding whether $P(x)$ halts:

```
def Halt(P, x):
    def M(t):
        run P(x)   # line 1 of M
        return     # line 2 of M
    return Reachable(M, 0, 2)
```

Program M reaches line 2 if and only if $P(x)$ halted. Thus, we have implemented a solution to the halting problem — contradiction.