

Math 27800 / CS 27800, Winter 2024: Assignment 3 (Solution)

Denis Hirschfeldt, Duarte Maia

Due Friday, February 2nd

Note: This document is meant both as a solution to Homework 3, as well as a PSA. Notes like this one will occur in the document. If they were to be removed, this document would be a full score solution to Homework 3. The notes are supplementary, and serve to point out important details and to explain common mistakes I found while grading the homework.

I recommend that you take a look at these solutions and compare with your own. \square

Exercise 1.

- Show that every infinite binary tree has an infinite path.
- Show that there is a computable infinite binary tree with no computable path.
- Let T be a computable infinite binary tree. Show that if we could compute the Halting Problem, then we could compute an infinite path on T .
- Show that there is a computable tree T such that, if we could compute an infinite path on T , then we could compute a completion of **ZFC**.

Solution (1.a): Since T is infinite, then either it contains infinitely many binary strings starting with a 0, or it contains infinitely many binary strings starting with a 1. If the former occurs, set $\alpha_0 = 0$, otherwise set $\alpha_0 = 1$.

Then, iteratively repeat this procedure. (In the following, denote string concatenation by juxtaposition.) If we have constructed $\sigma = \alpha_0 \dots \alpha_n$ such that infinitely many elements of T are extensions of σ (and so in particular $\sigma \in T$), it must be the case they either infinitely many elements of T extend $\sigma 0$, in which case we set $\alpha_{n+1} = 0$, or infinitely many extend $\sigma 1$, in which case we set $\alpha_{n+1} = 1$.

The resulting infinite sequence α is a path on T . \blacksquare

Note: The first paragraph is technically unnecessary. \square

Note: To be faithful to the original statement, I have written the solution to 1.a. in terms of binary strings, but it would be equally acceptable (and perhaps more readable) to phrase things in terms of ‘going left or right’ and ‘left and right branches’. \square

Solution (1.b): Define a total computable function f acting on a binary string σ by the following algorithm:

```
function  $f(\sigma)$ :  
  for  $i = 0, \dots, \text{length}(\sigma)$   
    run  $\Phi_i(i)$  for  $\text{length}(\sigma)$  steps.  
  if it halted:
```

```

        if  $\sigma_i = \Phi_i(i)$ :
            output 0 and halt.
        endif
    endif
endfor
output 1.
endfunction

```

We observe that f is indeed total computable, because the algorithm always computes $f(\sigma)$ in at most (roughly) $\text{length}(\sigma)^2$ steps. Define T to be the set whose characteristic function is f , which makes T obviously computable. We need to verify three things: that T is a tree, that T is infinite, and that no path on T is computable.

To verify that T is a tree means: if $f(\sigma) = 1$, and if τ is a prefix of σ , then $f(\tau) = 1$. This is true, because the computation for $f(\tau)$ will have i ranging over an even smaller subset, and for each value of i we run the computation of $\Phi_i(i)$ for even less time, so the algorithm has ‘strictly less opportunity’ to output 0.

To verify that T is infinite, we explicitly define an infinite path. Define α_n by: If $\Phi_i(i) \downarrow = 0$, set $\alpha_n = 1$, otherwise set $\alpha_n = 0$. By construction, $f(\alpha_0 \dots \alpha_n) = 1$ for every n , so α is indeed a path.

Finally, we verify that no such path is computable. Indeed, every computable path α is equal to Φ_j for some natural number j . Let $N > j$ be some amount of steps which suffices to execute $\Phi_j(j)$. We claim that the initial sequence $\sigma = \alpha_0 \dots \alpha_N$ is not in T .

Indeed, if we execute $f(\sigma)$, when the loop reaches $i = j$, by definition of $N = \text{length}(\sigma)$ the instruction $\Phi_j(j)$ will finish executing in time, and its output will be precisely σ_j , hence the algorithm will output that no, σ is not in T . Thus, no computable path α exists in T . ■

Note: A very common error in this exercise was to ‘construct’ the tree by making an algorithm which would iteratively build the tree ‘up’ as a list of strings. At face value, this is insufficient: This only shows that the resulting tree is c.e, not computable.

On that note, I would like to emphasize: *If an exercise asks you to construct a certain computable set, or that a certain set is computable, you should clearly outline an algorithm for computing the characteristic function of this set!* An algorithm which iteratively builds a list containing its elements, for example, is not sufficient. □

Solution (1.c): First we show that, with access to an oracle for the Halting Problem, we can tell whether a subtree of a computable tree is finite or infinite.

Let T be a computable tree, say χ is its characteristic function, and $\sigma \in T$ a node. An essential observation is that the following two statements are equivalent:

- There are infinitely many strings in T extending σ ,

- There are strings of arbitrarily large length in T extending σ . ■

(Sketch: The first implies the second because for every finite N there are finitely many strings of length $\leq N$. The second implies the first because for any finite collection of strings there is a common bound to their length.)

Therefore, the following algorithm will loop infinitely if there are infinitely many nodes below σ , and halt in finite time otherwise:

```
function  $g(\sigma)$ :
  for  $n = \text{length}(\sigma), \text{length}(\sigma) + 1, \dots$ :
    for  $\tau$  in 0,1-strings of length  $n$ :
      if  $\tau$  extends  $\sigma$  and  $\chi(\tau) = 1$ :
        goto [nextiteration]
      endif
    endfor
  halt execution and return 0
  [nextiteration]
endfor
endfunction
```

We can apply the s - m - n Theorem (we may need to add a dummy second argument to g) to obtain a computable function s such that $\phi_{s(\sigma)}(s(\sigma)) = g(\sigma)$, and so $s(\sigma)$ is in the Halting Problem iff there are infinitely many nodes below σ in T .

Thus, the following algorithm, which makes use of an oracle for the Halting Problem (which we call **hp**), will print out an infinite binary sequence of zeros and ones, which is itself an infinite path on T . This is achieved by encoding as an algorithm the proof of 1.a, and so the same argument therein proves that the output is an infinite path in T . In the sequence, we use juxtaposition to mean string concatenation, so e.g. $\sigma 1$ means ‘the binary string σ , plus a 1 at the end’.

```
begin procedure X:
   $\sigma \leftarrow$  (empty string)
  while True:
    if hp( $s(\sigma 0)$ ): //finitely many nodes on the left
       $\sigma \leftarrow \sigma 1$ 
      print(1)
    else:
       $\sigma \leftarrow \sigma 0$ 
      print(0)
    endif
  endwhile
end procedure.
```

We can turn this into a *bona fide* path α (i.e. a function $\mathbb{N} \rightarrow \{0, 1\}$) by the following method:

```
function  $\alpha(n)$ :
  run procedure X until  $n$  characters have been printed.
```

```

    (this will happen after  $n$  iterations of the loop)
    output this character.
endfunction

```

Solution (1.d): We construct a tree using an algorithm that is very similar to the solution of 1.b.

We take for granted, by the Church-Turing thesis, that we have an effective enumeration of all formulas in the language of set theory, say s_0, s_1 , etc. and likewise an effective enumeration of all finite sequences of such formulas, say p_0, p_1 , etc.

Let σ be a finite binary string. We identify it with the extension of **ZFC** obtained by adding to it the axioms: s_i for $i < \text{length}(\sigma)$ with $\sigma_i = 1$, and $\neg s_i$ for $i < \text{length}(\sigma)$ with $\sigma_i = 0$. By abuse of notation, let us call this extension **ZFC** + σ .

By the Church-Turing thesis, there is a computable function $c(i, \sigma)$ which checks whether p_i is a proof of **ZFC** + $\sigma \vdash \exists x (x \neq x)$.

That said, define:

```

function  $f(\sigma)$ :
    compute  $c(0, \sigma), \dots, c(\text{length}(\sigma), \sigma)$ 
    if a contradiction is found:
        output 0
    else:
        output 1
    endif
endfunction

```

It is clear that f is total and is the characteristic function of some set of binary strings T . We show that T is a tree (which is evidently computable), and that paths in T are in correspondence with completions of **ZFC**.

To show that T is a tree: Suppose that $f(\sigma) = 1$, and that τ is a prefix of σ such that $f(\tau) = 0$. Then, there is some $i < \text{length}(\tau)$ such that $c(i, \tau) = 1$. However, by definition of c we easily obtain that $c(i, \sigma) = 1$. But since $i < \text{length}(\tau) \leq \text{length}(\sigma)$, this contradicts the assumption that $f(\sigma) = 1$.

Now, let α be an infinite binary string. To it, we may correspond an extension of **ZFC**, let us call it **ZFC** + α , defined by $\bigcup_n \text{ZFC} + (\alpha_0 \dots \alpha_n)$. Evidently, **ZFC** + α is always complete, and every complete consistent extension is of this form, so the question is for which α is **ZFC** + α consistent.

First, suppose that **ZFC** + α is consistent. Then, for every finite initial segment σ of α we have **ZFC** + σ is consistent, and by definition of f it is evident that $f(\sigma) = 1$. Thus, α is a path on T .

Finally, suppose that **ZFC** + α is inconsistent. Then, by compactness there is a proof p_n of a contradiction that uses a finite fragment of **ZFC** + α , say **ZFC** + $\alpha_0 \dots \alpha_m$. Let $N = \max(n, m)$. Then, by definition of f it is easy to verify that $f(\alpha_0 \dots \alpha_m) = 0$, and consequently α is not a path on T .

In conclusion, we have built a computable tree T whose paths are in one-to-one effective correspondence with completions of **ZFC**, and so given such a path we could construct a completion of **ZFC**. ■

Note: Notice that in the second paragraph, we introduced an *effective* enumeration of the formulas and of the proofs, by resorting to the Church-Turing thesis. Some students used instead an argument like the following: There are countably many symbols, and a formula is a finite sequence of symbols, so there are countably many formulas, and we let s_0, s_1 , etc. be an enumeration of the formulas. This argument is invalid.

When doing computability theory, cardinality is rarely a valid argument to use, because knowing that there is an enumeration of a set does not let us use that enumeration in an algorithm; otherwise, it would be very easy to solve the halting problem (its complement is countable, after all!)

Be careful to distinguish an enumeration from an ‘effective enumeration’, or equivalently a computable enumeration. □

Note: As written, problem 1.d is trivially true. Any finite tree T will satisfy what T asks: if we could compute an infinite path on T , we vacuously could compute a completion of **ZFC**. This is would obviously not be accepted as a solution, though to my surprise no one even bothered trying.

A slightly closer to intended interpretation of the problem might be to demand that T is infinite, at least assuming that **ZFC** is consistent. Since this was not asked for in the problem, there was no penalization for failing to do so. The above solution proves something slightly stronger, which is that there is a ‘nice’ correspondence between consistent completions of **ZFC** and paths in T . From this, we obtain that if **ZFC** is consistent, there is a completion of it, which furnishes a path in T and hence T is infinite. □

Note: The reader will have noticed similarities between problem 1.b and problem 1.d. This is no coincidence. In both cases, the problem would be much easier to solve if the trees were requested to be co-c.e. (i.e. if $\mathbb{N} \setminus T$ is requested to be c.e.). Such trees are closely related to something called Π_1^0 classes, and a study of them may be found in any modern book on computability theory. The solutions to problems 1.b and 1.d are both applications of a general trick, which allows us to turn any co-c.e. tree T into a slightly larger computable tree T' with no more infinite paths than T (and so exactly the same infinite paths). □

Exercise 2. Show that there are computably inseparable c.e. sets A and B .

Solution: Following the hint, set

$$\begin{aligned} A &= \{n \in \mathbb{N} \mid \phi_n(n) \downarrow = 0\}, \\ B &= \{n \in \mathbb{N} \mid \phi_n(n) \downarrow > 0\}. \end{aligned} \tag{1}$$

Both A and B are evidently c.e., as A (resp. B) is the domain of the following computable function: Given $n \in \mathbb{N}$, compute $\phi_n(n)$, and if it is zero (resp. nonzero), return 0, otherwise loop forever.

Now, suppose for the sake of contradiction that there is a computable set C which separates A and B as in the problem statement. Let ϕ_c be the characteristic function of C . Note that ϕ_c is a total function, and hence $\phi_c(c)$ is well-defined.

If $\phi_c(c) = 0$, then $c \in A$ but $c \notin C$, which contradicts the assumption that $A \subseteq C$.

If $\phi_c(c) = 1$, then $c \in B$ but $c \in C$, which contradicts the assumption that $C \cap B = \emptyset$.

In either case we have a contradiction, and thus C may not exist. Hence, A and B are computably inseparable. ■

Exercise 3. Let A and B be c.e. sets. For each of the following sets, must the set be c.e.?: $A \cup B$, $A \cap B$, $A \setminus B$.

Solution: Since A and B are c.e., each of them is the domain of some partial computable function, say resp. ϕ_a and ϕ_b .

- ($A \cup B$ is c.e.) Consider the following algorithm: Given x , execute the Turing Machines for $\phi_a(x)$ and $\phi_b(x)$ in parallel. If either of them ever halts, halt execution and output 0.

The resulting partial computable function will evidently have domain $A \cup B$.

- ($A \cap B$ is c.e.) Consider the following algorithm: Given x , compute $\phi_a(x)$, and once the execution is done, output $\phi_b(x)$.

The resulting partial computable function will evidently have domain $A \cap B$.

- ($A \setminus B$ may not be c.e.) Consider $A = \mathbb{N}$ and let B be the Halting Problem. Both are known to be c.e., but B is known not to be computable. We know from class that if both B and its complement are c.e. then B is computable, and so we obtain that $\mathbb{N} \setminus B = A \setminus B$ is not c.e. in this scenario. ■

Note: This exercise is an example of a scenario where knowing the two different facets of the definition of c.e. comes in useful. Indeed, to show that $A \cup B$ is c.e. it is actually quite natural to consider the characterization of c.e. via ‘image of a total function’ or ‘there is an algorithm which prints out its elements’; for example, if there is an algorithm which prints out all the elements of A and one which prints out the elements of B , we might consider running both algorithms in parallel and outputting the results of both to the same screen.

This solution is perfectly reasonable and natural, but requires a little bit of care in handling the (trivial) cases where either A or B is empty, which is why I chose to use the ‘domain of a partial computable function’ characterization instead.

On the other hand, to show that $A \cap B$ is c.e., the domain characterization leads to an extremely simple and natural solution, while the ‘algorithm that

prints out all elements' characterization leads to a much more convoluted solution. It's still doable, mind you: If $A = \text{im}(f)$ and $B = \text{im}(g)$, one may iterate over the pairs $i, j \in \mathbb{N}^2$, check whether $f(i) = g(j)$, and print out the result if this is the case. But the other solution is much cleaner. \square