Computability Theory: Key Concepts

The general problem we want to confront is this: given a set or relation, when is there an algorithm for testing membership in that set? We can reduce this problem to another problem that, at first, appears to be much more restricted, namely, given a set *of natural numbers*, when is there an algorithm for testing membership in that set? We can effect this reduction by *coding* the given problem as a problem about numbers. A few examples will illustrate how this is done:

Example: We already know that there is an algorithm for testing whether an SC sentence is valid. Let us see how this problem can be coded as a problem about numbers. We first associate, with each simple symbol of our SC language, a numerical code, as follows:

1 is the code for "("

2 is the code for ")"

3 is the code for "∨"

4 is the code for "∧"

5 is the code for "→"

6 is the code for "↔"

7 is the code for "¬"

8 is the code for "A"

9 is the code for "B"

10 is the code for "C"

11 is the code for "D"

And so on. We can think of a sentence of the SC language as a finite sequence of symbols, so we can encode a sentence as a finite sequence of numbers. Moreover, as we shall see in a moment, it is possible to code a finite sequence of natural numbers as a single natural numbers. Putting

these two encodings together, we associate a code number with each sentence. What I have in mind by talking about coding is this: there is an algorithm which, given a an SC sentence as input, gives the code number of that sentence as output. Furthermore, there is an algorithm that, given a number as input, first determines whether the number is the code of an SC sentence; if it is, the algorithm gives you the sentence. Once we have such a coding, we see that the problem of determining whether a given SC sentence is valid reduces to the problem of testing whether a given natural number is the code of a valid SC sentence.

Coding a pair of natural numbers. Given natural numbers x and y, let pair(x,y) = ½(x^2 + 2xy + y^2 + x + 3y). pair is a bijection[1] from the set of ordered pairs of natural numbers to the set of natural numbers. There is an algorithm that, given x and y, givens you pair(x,y), and another algorithm that, given z, gives you the unique numbers x and y with pair(x,y) = z. Let's write x = 1st(z) and y = 2nd(z).

Coding a finite set of natural numbers. Where F is a finite set of natural numbers, let Code(F) = $\sum \{2^n: n \in F\}$. Code is a bijection from the set of finite sets of natural numbers to the set of

---

1 Recall that a *function* from a set A to a set B is a set f ⊆ A × B with the property that, for each element a of A, there is one and only one element b of B with <a,b> ∈ f. If <a,b> ∈ f, we write f(a) = b. A is the *domain* of the function, and the set of all elements b of B such that, for some a ∈ A, f(a) = bI is the *range.* If the range of f is all of B, f is said to be *surjective* or *onto*. If, for each a and a* in A, if f(a) = f(a*), then a = a, f is said to be *injective* or *one-one*. If f is both surjective and injective, f is said to be *bijective* or a *one-one correspondence.* If f is a bijection form A to B, the *inverse* of f, f^{-1}, {<b,a>: <a,b> ∈ f}, is bijection from B to A.

natural numbers. There is an mechanical procedure that, given F, will calculate Code(F). Moreover, there is a algorithm in the opposite direction, that, given a number n, computes the unique set F with Code(F) = n: write n in binary notation, and take F to be the set of places where 1s appear.

Coding a finite sequence. Given a finite sequence $<s_0, s_1, s_2,..., s_n>$, let its code be Code({pair(0,s_0), pair(1,s_1), pair(2,s_2),..., pair(n,s_n)}).

Example: We can form two infinite lists, one of which lists all the expressions of English in alphabetical order, and the other of which lists all the expressions of Russian in alphabetical order. There is a mechanical procedure by which, given an English expression, we can find the position of the expression on the list, and also a procedure by which, given a number n, we can find the nth expression on the list. Then the problem of determining, given an English expression and an Russian expression whether the latter is an acceptable translation of the former, is equivalent to the arithmetical problem of determining, for given m and n, whether pair(m,n) is an element of {pair(i,j): the ith English expression is an acceptable translation of the4 jth Russian expression.

Example: The core of the Star Wars nuclear defense system is envisaged to be a gigantic supercomputer that takes radar traces of attacking missiles as inputs and yields instructions to US missiles that will shoot the attacking missiles down as outputs. Now it isn't possible to assign a distinct numerical code to each possible trajectory of an incoming missile, because there are more possible trajectories than there are natural numbers. However, the apparatus we use to sense the incoming missiles has limited sensitivity, so that it will be unable to distinguish among trajectories that are very close together. If we group together trajectories that the instruments are

unable to distinguish, we find that there are only finitely many discernibility classes of trajectories, and these can be assigned numerical codes. The instructions the computer sends to the defensive missiles can likewise be given numerical codes, so that the computer's problem can be coded as a numerical problem.

We now present some key definitions. The definitions utilize colloquial notions that haven't been made mathematical precise, but they will nonetheless be precise enough to permit us to prove some theorems:

Definitions. A *partial function* from the natural numbers to the natural numbers is a subset f of $\mathbb{N} \times \mathbb{N}$ that meets the following condition:

$$(\forall x)(\forall y)(\forall z)((\langle x,z\rangle \in f \wedge \langle y,z\rangle \in f) \rightarrow x = y)$$

In other word, f is a function from a subset of $\mathbb{N}$ onto a subset of $\mathbb{N}$.

A partial function f is *total* iff its domain is all of f. Thus, in our usage, total functions are a kind of partial function.

A partial function f is *calculable* iff there is an algorithm such that, given n as input, the algorithm gives f(n) as output if n is in the domain of f; if f isn't in the domain of f, the algorithm gives no output.

A *decision procedure* for S is an algorithm that calculates the characteristic function of S; that is, if the input is a member of S, the output is 1, whereas if the input is a nonmember, the output is 0.

S is *decidable* if f there is a decision procedure for S.

A *proof procedure* for S is an algorithm that calculates a partial function whose domain includes S that assigns the value 1 to an input if and only if the input is an element of S.

An *enumeration procedure* for S is an algorithm that lists the elements of S. The function that takes a number n to the nth element on the list is a calculable partial function whose domain is an initial segment of the natural numbers and whose range is S.

S is *effectively enumerable* iff there is an enumeration procedure for S.

Theorem. There is an proof procedure for S if and only if S is effectively enumerable.

Proof: (⇒) A proof procedure for S calculates a partial function f with $Dom(f) \supseteq S$ and $f(n) = 1$ iff $n \in S$, for $n \in Dom(f)$. We want to find an enumeration procedure for f. Here's a proposal that doesn't work:

First, calculate f(0). If $f(0) = 1$, put 0 on the list.

Second, calculate f(1). If $f(1) = 1$, put 1 on the list.

Third, calculate f(2). If $f(2) = 1$, put 2 on the list.

Fourth, calculate f(3). If $f(3) = 1$, put 3 on the list.

And so on.

The trouble is that, when the algorithm gets to something that's not in the domain of f, it gets stuck. We want to modify the algorithm so that meeting up with a number that's not in the domain won't prevent it form going on to consider greater numbers. We might try this:

First, attempt to calculate f(0). If $0 \notin$ Dom(f), skip to step 2. if $0 \in$ Dom(f) and f(0) = 1, put 0 on the list.

Second, attempt to calculate f(1). If $1 \notin$ Dom(f), skip to step 3. If $1 \in$ Dom(f) and f(1) = 1, put 1 on the list.

Third, attempt to calculate f(2). If $2 \notin$ Dom(f), skip to step 4. If $2 \in$ Dom(f) and f(2) = 1, put 2 on the list.

Fourth, attempt to calculate f(3). If $3 \notin$ Dom(f), skip to step 5. If $3 \in$ Dom(f) and f(3) = 1, put 3 on the list.

And so on.

The trouble is that, in general, we won't have any test to tell us whether a number is in the domain of f. If n is in the domain of f, our algorithm for f will compute f(n), but if f isn't in the domain, the algorithm will typically keep running forever without giving an output. We want to arrange our procedure so that we never give up on trying to calculate f(n), but doing this won't prevent us from considering numbers greater than n. We accomplish this by a technique called *dovetailing* that weaves different computations together:

Step 1. Take one step in the attempt to compute f(0). If you succeed, see whether f(0) = 1. If it is, put 0 on the list.

Step 2. Take one step in the attempt to compute f(1). If you succeed, see whether f(1) = 1. If it is, put 1 on the list.

Step 3. Take another step in the attempt to compute f(0) (if you didn't already succeed in calculating f(0) at step 1). If you succeed in calculating f(0), see whether it's equal to 1. If it is, put 0 on the list.

Step 4. Perform one step in the attempt to calculate f(2). If you succeed and f(2) = 1, put 2 on the list.

Step 5. Take another step in the attempt to compute f(1) (if you didn't already succeed in calculating f(1) at step 2). If you succeed in calculating f(1) and it's equal to 1, put 1 on the list.

Step 6. Take another step in the attempt to compute f(0) (if you didn't already calculate f(0) at step 1 or step 3). If you succeed in calculating f(0) and it's equal to 1, put 0 on the list.

Step 7. Perform one step in the attempt to calculate f(3). If you succeed and f(3) = 1, put 3 on the list.

Step 8. Perform another step in the attempt to calculate f(2) (if you didn't already succeed in calculating f(2) at step 4). If you succeed in calculating f(2) and it's equal to 1, put 2 on the list.

Step 9. Perform another step in the attempt to calculate f(1) (if you didn't already succeed in calculating f(1) at step 2 or step 5). If you succeed in calculating f(1) and it's equal to 1, put 1 on the list.

Step 10. Perform another step in the attempt to calculate f(0) (if you haven't succeeded in calculating f(0)). If you succeed in calculating f(0) and it's equal to 1, put 0 on the list.

Step 11. Carry out one step in the attempt to calculate f(4). If you succeed and f(4) = 1, put 4 on the list.

Step 12. Perform another step in the attempt to calculate f(3) (if you didn't already succeed in calculating it). If you succeed in calculating f(3) and it's equal to 1, put 3 on the list.

Step 13. Perform another step in the attempt to calculate f(2) (if you haven't already calculated it successfully). If you succeed in calculating f(2) and it's equal to 1, put 2 on the list.

Step 14. Perform another step in the attempt to calculate f(1) (if you haven't calculated it already). If you succeed in calculating f(1) and it's equal to 1, put 1 on the list.

Step 15. Perform another step in the attempt to calculate f(0) (if you haven't calculated it already). If you succeed in calculating f(0) and it's equal to 1, put 0 on the list.

And so on.

(⇐) If we have an enumeration procedure for S, our proof procedure for S will e this:

Given n. Begin listing S. If and when n appears on the list, give the output 1.⊠

Theorem. A set is decidable if and only if it and its complement are both effectively enumerable.

Proof: (⇒) If S is decidable, then there is an algorithm for computing the characteristic function, χ_S, of S. This algorithm will also be a proof procedure for S. The algorithm that takes n to 1 - $\chi_S(n)$ will be a proof procedure for the complement of S.

(⇐) Given n, begin listing both S and its complement simultaneously. If n appears on the list for S, give the output 1. If n appears on the list for the complement, give the output 0.⊠

We've spoken about decidable and effectively enumerable sets, but we can also talk about decidable and effectively enumerable relations, and the same theorems will hold, with the same proofs. Similarly, we described "calculable partial function" for 1-place functions, but we can also talk about functions of more than one argument. We have:

> **Theorem.** A partial function of one argument is calculable if and only if, regarded as a binary relation, it is effectively enumerable.

Proof: (⇒) Suppose that f is a calculable partial function. Here is a proof procedure for f, thought of as a binary relation: given m and n, attempt to calculate f(m). If you get an output, check whether it's equal to n. If it is, give the output 1.

(⇐) Given an enumeration procedure for f, here is an algorithm for calculating f: given m, begin enumerating f. As a pair appears on the list, check whether its first component is equal to m. If it is, give the second component as output.⊠

> **Theorem.** A total function of one argument is calculable if and only if, regarded as a binary relation, it's decidable.

Proof: (⇒) Given that f is a calculable total function, here is a decision procedure. Given m and n, begin calculating f(m). If f(m) is equal to n, given the output 1. If f(m) is different from n, give the output 0.

(⇐) Any decidable total function will be an effectively enumerable partial function, and so calculable.⊠

Theorem. The union of two effectively enumerable sets is effectively enumerable.

Proof: Given enumeration procedures for A and B, here is a proof procedure for A ∪ B: given n, begin simultaneously listing A and B. If n appears on either list, give the output 1.⊠

Theorem. The intersection of two effectively enumerable sets is effectively enumerable.

Proof: Given enumeration procedures for A and B, here is a proof procedure for there intersection: given n, begin enumerating A If n appears on the list, then stop worrying about A and start listing B. If n appears, give the output 1⊠

Theorem. A set is effectively enumerable iff it's the domain of a calculable partial function.

Proof: (⇒) If A is effectively enumerable, then there is a proof procedure for A. By definition, that means that there is a calculable partial function f such that, for any n, n is in A if and only n is in the domain of f and f(n) = 1. Define a calculable function g by the following algorithm:

Being calculating f(n). If you get a value, check whether it's equal to 1. If it is, give the output 1

Then A is the domain of g.

(⇐) If f is a calculable total function, a proof procedure for the domain of f is the following:

Begin calculating f(n). If you get an output, give the output 1.⊠

Theorem. A set is effectively enumerable iff it's the range of a calculable partial function.

Proof: (⇒) If A is effectively enumerable, then it's the range of the partial function that takes n to the nth number on the list.

(⇐) If f is calculable, then, regarded as a binary relation, it's effectively enumerable. A algorithm for listing the range of f is the following:

> List f. Whenever an ordered pair appears, give it's second member as an output.⊠

> **Theorem.** A set is effectively enumerable iff it's either the empty set or the range of a calculable total function.

Proof: (⇒) Suppose that A is effectively enumerable and nonempty. If A is infinite, then the function that takes n to the nth element on the list is a calculable total function whose range is A. If A is finite, then it has the form $A = \{a_0, a_1, a_2,..., a_k\}$. Then A is the range of the function f, defined as follows:

> If $i = 0$, $f(i) = a_0$.
>
> If $i = 1$, $f(i) = a_1$.
>
> If $i = 2$, $f(i) = a_2$.
> .....
> If $i = k$, $f(i) = a_k$.
>
> If $i > k$, $f(i) = a_k$.

(⇐) If A is the empty set, it's enumerated by the lazy algorithm that never gives any output. If A is the range of a calculable total function, then it's the range of a calculable partial function.⊠

> **Theorem.** A set is effectively enumerable iff it's either finite or the range of a one-one calculable total function.

Proof: (→) If A is infinite and effectively enumerable, it can be listed without repetitions. Simply modify the listing procedure so that a number can only be added to the list if it hasn't been listed already. Then the function that takes n to the nth item on the list is a one-one calculable total function whose range is f.

(←) If A is finite, it's effectively enumerable. If A is the range of a one-one calculable total function, it's the range of a calculable partial function.⊠

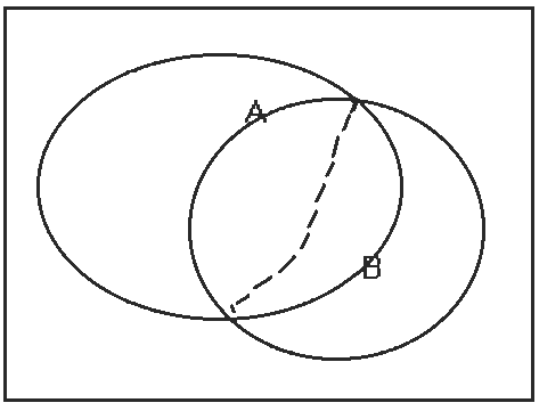
Theorem. If A and B are effectively enumerable sets, then there are effectively enumerable sets C and D with

$C \subseteq A$

$D \subseteq B$

$C \cap D = \varnothing$

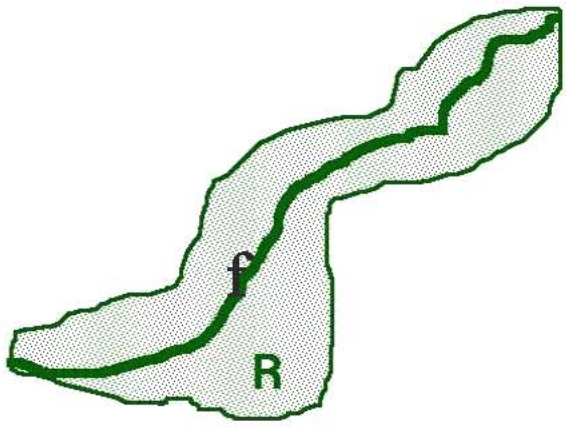
$C \cup D = A \cup B$



Proof: Here are enumeration procedures for C and D: begin simultaneously to list A and B. If a number n appears on the list for A at a stage at which it has not yet appeared on the list for B, put n on the list for C. If n appears on the list for B at a stage at which it has not yet appeared on the list for A, put n on the list for D. If n appears on both lists at the same stage, put n on the list for A. ⊠

Theorem. If R is an effectively enumerable relation such that $(\forall x)(\exists y)R(x,y)$, then there is a calculable total function f such that $R(x,f(x))$, for every x.

Proof: Here is an algorithm for calculating f: given n, begin enumerating the ordered pairs in R until you come to one whose first component is equal to n. The first time you encounter such a pair, give its second component as output.⊠



If f is a calculable partial function (of one argument, say), then there is a computer program that calculates f. That is, there is a program that, given a number n as an input, will calculate for a while, then give the output f(n), then halt, if n is in the domain of f. If n isn't in the domain of f, the program will keep running forever, without giving any output. (We'll look at this a little later on in more detail.) We can write arrange all the possible program in alphabetical order (or something like it), so that, for each calculable partial function f, there is a number m such that the mth machine calculates f. Given m and n, we can write out the mth program, then calculate what output, if any, the program gives on the output n. The *halting problem* is this: given m and n, to determine whether the mth program halts when it's given the input n. There is a proof procedure for the halting problem, consisting in just carrying out the computation. There is, however, no decision procedure.

Theorem. There is no decision procedure for the halting problem.

Proof: If there were such a decision procedure, then the following recipe would compute a calculable total function – call it f:

Given m. If the mth machine yields the output k on input m, give the output k+1. If the mth machine doesn't halt on input m, give the output 0.

Because f is calculable, there is a machine that calculates f; let's say it's the jth machine. Because f is total, the jth machine yields an output on every input. In particular, the jth machine yields an output on the input j, and we have:

1 + the output of the jth machine on input j

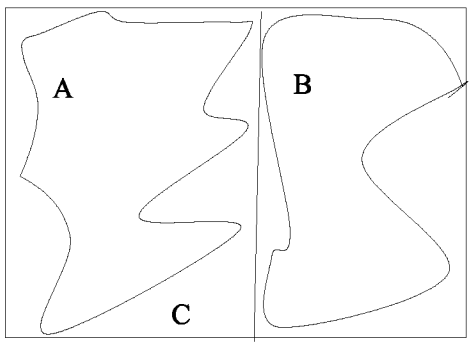
= f(j) [by the way f was defined]

= the output of the jth machine on input j [by the way j was chozen]

Contradiction.⊠

Theorem. There are disjoint, effectively enumerable sets A and B such that there isn't any decidable set that includes A and in disjoint from B.

Proof: Let A = {m: the mth machine gives output 0 on input m}. Let B = {m: the mth machine gives output 1 on input m}. Then A and B are disjoint and effectively enumerable. Pretend there were a decidable set C that included A and was disjoint from B. Since C is decidable, its characteristic function is calculable. Let's say the kth machine calculates the characteristic function of the complement of C.



If k is in C, then $\chi_C(k) = 1$, and so the kth machine yields output 1 on input k, which means that k is in B. But that's impossible, since B is disjoint from C.

So k isn't in C, and so $\chi_C(k) = 0$, that is, the kth machine gives output 0 on input k. But that means that k is in A, which is a subset of C. Contradiction.⊠

Theorem. There is a calculable partial function that can't be extended to a calculable total function.

Proof: Using A and B from the last theorem, define a calculable partial function g by:

$g(n) = 1$ if $n \in A$

$= 0$ in $n \in B$

Suppose, for *reductio ad absurdum*, that there were a calculable total function h that extended g. Then the function that takes an input n to the maximum of h(n) and 1 would be the characteristic function of a decidable set that included A and was disjoint from B.⊠

Something important to remember is that effective enumerability and decidability are properties of *sets*. Whether a set is decidable doesn't depend on how the set is named, and it doesn't depend on our epistemic state. Often, a set can be named in many different ways. S might be {n: n has property P} and it might also be {n: n has property Q}, and it might turn out that we have an algorithm for answering all question of form "Does _____ have property P?" (where the blank if filled in with an Arabic numeral[2]) but no algorithm for answering all questions of the form "Does _____ have property Q?" In such a case, S would count as decidable. A set S is decidable iff there is some property[3] P such that S is the set of numbers that

---

2 S to be decidable, we don't have to be able to answer questions like, "Does the number of fish in Lake Anza have property P?"

3 Here I am using the notion of property "pleonastically," so that to say that Traveler has the property of horseness is just another way of saying that Traveler is a horse. We could express the same idea without getting tangled in the metaphysics by talking about predicates. S is decidable iff there is some predicate ϕ such that S = {n: ϕ(n)} and such

have property P and such that there is an algorithm for answering questions of the form "Does _____ have property P?" The fact that there is some other property Q such that S is the set of numbers that have property Q and such that there is no known algorithm for answering questions of the form "Does _____ have property Q?" doesn't spoil the decidability of S.

To take an example, let D = {numbers n: there is a string of n or more successive 7s in the decimal expansion of π}. D is clearly effectively enumerable. The enumeration procedure is simply to start grinding out the decimal expansion of π and to add n to the list when you come across a string of n 7s. Is D also decidable? No one knows how to answer questions of the form "If _____ is D?" No one knows whether 1000 is in D, or whether 1,000,000 is in D. As far as anyone knows, every number could be in D. Nonetheless, D is decidable. If it happens to be the case that every number is in D, the a decision proacedure for D is the following:

No matter what the input, give the output 1.

If not every number is in D, then there is a number k such that D = {n: $n \leq k$}. In that case, a decision procedure for D is this:

Give the output 1 is the input is $\leq$ k. If the input is > k, give the output 0.

One way or another, there is a decision procedure for D.

The same goes for functions: a partial function is a set of ordered pairs, and whether it's calculable doesn't depend on how the function is names. To take an example, the Continuum

---

that there is an algorithm for determining the truth values of sentences obtained from the open sentence $\phi(x)$ by replacing free occurrences of "x" by a numeral. It doesn't matter if there is another predicate ψ such that S = (n:ψ(n)} for which there is no such algorithm.

Hypothesis is the most famous unproved conjecture in set theory.[4] Not only hasn't anyone ever been able either to prove or to refute the Continuum Hypothesis, but it's known that the hypothesis can't be either proven or refuted on the basis of the currently accepted axioms of set theory. Now consider the function c, defined as follows:

$f(n) = n+1$ if the Continuum Hypothesis is true

$= n$ it the Continuum Hypothesis is not true

It is not possible, on the basis of the currently accepted of set theory, to determine any of the values of the function c. Nonetheless c is calculable. Either c is the successor function, which is calculable, or c is the identity map, which is calculable.

We've managed to identify some general structural properties of the set of effectively enumerable sets, but we haven't yet attempted to say precisely which the effectively enumerable sets are. We are going to wind up identifying the effectively enumerable sets as those that are named by especially simple formulas of the language of arithmetic. So what we need to do now is to introduce the language of arithmetic.

---

4 To make the point we're making here, it doesn't matter what the hypothesis says.