CS301: Computability and Complexity Theory (CC)

Lecture 8: A definition of Information

Dumitru Bogdan

Faculty of Computer Science
University of Bucharest

November 24, 2023

Table of contents

Section 1

Previously on CS301

Computable functions

A TM computes a function by starting with the input to the function on the tape and halting with the output of the function on the tape

Definition

A function $f : \Sigma^* \to \Sigma^*$ is a **computable function** if some TM M, on every input w, halts with just $f(w)$ on its tape.

Example: All usual, arithmetic operations on integers are computable functions. For example, we can make a machine that takes input $< m, n >$ and returns $m + n$, the sum of m and n.

Mapping Reducibility

Next, we defined mapping reducibility. As usual, we represent computational problems by languages.

Definition

Language A is mapping reducible to language B, written $A \leq_m B$ if there is a computable function $f : \Sigma^* \to \Sigma^*$, where for every w

$$w \in A \iff f(w) \in B$$

the function f is called the **reduction** from A to B.

Mapping Reducibility

A mapping reduction of A to B provides a way to convert questions about membership testing in A to membership testing in B. To test whether $w \in A$, we use the reduction f to map w to $f(w)$ and test whether $f(w) \in B$. The term mapping reduction comes from the function or mapping that provides the means of doing the reduction.

If one problem is mapping reducible to a second, previously solved problem, we can thereby obtain a solution to the original problem. We capture this idea in next theorem

Theorem

If $A \leq_m B$ and B is decidable, then A is decidable.

Corollary

If $A \leq_m B$ and B is undecidable, then B is undecidable.

Section 2

Context setup

Context setup

Corresponding to Sipser 6.4 & 7.1

Context setup

- We revisit an important aspect of computability theory: **descriptive complexity**
- We start a new chapter: **time complexity**
- Even when a problem is decidable and thus computationally solvable in principle, it may not be solvable in practice if the solution requires an inordinate amount of time or memory
- We introduce computational complexity theory an investigation of the time, memory, or other resources required for solving computational problems. We begin with **time**
- First we introduce a way of measuring the time used to solve a problem. Then we show how to classify problems according to the amount of time required
- After that we discuss the possibility that certain decidable problems require enormous amounts of time, and how to determine when you are faced with such a problem

Section 3

A Definition of Information

Definition of Information

- The concepts algorithm and information are fundamental in computer science. While the Church–Turing thesis gives a universally applicable definition of algorithm, no equally comprehensive definition of information is known
- Instead of a single, universal definition of information, several definitions are used, depending upon the application. Next we discuss one way of defining information, using computability theory

Definition of Information

We start with an example. Consider the information content of the following two binary sequences

$$A = 01010101010101010101010101010101010101$$
$$B = 11100101101000111010100001110100110101111$$

Intuitively, sequence A contains little information because it is merely a repetition of the pattern 01 twenty times. In contrast, sequence B appears to contain more information.

Definition of Information

- We can use previous simple example to illustrate the idea behind the definition of information that we present
- We define the quantity of information contained in an object to be the size of that object's smallest representation or description
- By a description of an object, we mean a precise and unambiguous characterization of the object so that we may recreate it from the description alone
- Sequence A contains little information because it has a small description, whereas sequence B apparently contains more information because it seems to have no concise description

Definition of Information

- Why do we consider only the shortest description when determining an object's quantity of information?
- We may always describe an object, such as a string, by placing a copy of the object directly into the description
- We can obviously describe the preceding string B with a table that is 40 bits long containing a copy of B. This type of description is never shorter than the object itself and doesn't tell us anything about its information quantity
- However, a description that is significantly shorter than the object implies that the information contained within it can be compressed into a small volume, and so the amount of information can't be very large
- Hence **the size of the shortest description determines the amount of information**
- Next, we formalize this intuitive idea. First, we restrict our attention to objects that are binary strings. Second, we consider only descriptions that are themselves binary strings. By imposing this requirement, we may easily compare the length of the object with the length of its description

Minimal length

- Many types of description language can be used to define information. Selecting which language to use affects the characteristics of the definition. Our description language is based on algorithms
- One way to use algorithms to describe strings is to construct a TM that prints out the string when it is started on a blank tape and then represent that TM itself as a string
- A drawback to this approach is that a TM cannot represent a table of information concisely with its transition function. To represent a string of n bits, you might use n states and n rows in the transition function table
- Instead we describe a binary string x with a Turing machine M and a binary input w to M. The length of the description is the combined length of representing M and w
- We write this description with our usual notation for encoding several objects into a single binary string $< M, w >$
- We define the string $< M, w >$ to be $< M >, w$, where we simply concatenate the binary string w onto the end of the binary encoding of M. The encoding $< M >$ of M may be done in any standard way

Descriptive complexity

Definition

Let x be a binary string. The minimal description of x, written $d(x)$, is the shortest string $< M, w >$ where TM M on input w halts with x on its tape. If several such strings exist, select the lexicographically first among them. The **descriptive complexity** of x, written $K(x)$, is

$$K(x) = |d(x)|$$

In other words, $K(x)$ is the length of the minimal description of x. The definition of $K(x)$ is intended to capture our intuition for the amount of information in the string x. Descriptive complexity is also known as **Kolmogorov complexity**.

Next we establish some simple results about descriptive complexity.

Descriptive complexity

This theorem says that the descriptive complexity of a string is at most a fixed constant more than its length. The constant is a universal one, not dependent on the string

Theorem

$\exists c \forall x [K(x) \leq |x| + c]$

Proof.

To prove an upper bound on $K(x)$ as this theorem claims, we need only demonstrate some description of x that is no longer than the stated bound. Then the minimal description of x may be shorter than the demonstrated description, but not longer. Consider the following description of the string x. Let M be a TM that halts as soon as it is started. This machine computes the identity function, its output is the same as its input. A description of x is simply $< M > x$. Letting c be the length of $< M >$ completes the proof □

Descriptive complexity

Theorem

$\exists c \forall x [K(xx) \leq K(x) + c]$

Proof.

Consider the following TM M, which expects an input of the form $< N, w >$, where N is a TM and w is an input for it

$M =$ On input $< N, w >$, where N is a TM and w is a string:

1. Run N on w until it halts and produces an output string s
2. Outputs the string ss

A description of xx is $< M > d(x)$. Recall that $d(x)$ is a minimal description of x. The length of this description is $| < M > | + |d(x)|$, which is $c + K(x)$ where c is the length of $< M >$ $\square$

Descriptive complexity

Next we examine how the descriptive complexity of the concatenation xy of two strings x and y is related to their individual complexities.

Theorem

$\exists c \forall x, y [K(xy) \leq 2K(x) + K(y) + c]$

Descriptive complexity

Proof.

We construct a TM M that breaks its input w into two separate descriptions. The bits of the first description $d(x)$ are all doubled and terminated with string 01 before the second description $d(y)$ appears. Once both descriptions have been obtained, they are run to obtain the strings x and y and the output xy is produced. The length of this description of xy is clearly twice the complexity of x plus the complexity of y plus a fixed constant for describing M. This sum is

$$2K(x) + K(y) + c$$

and the proof is complete □

Descriptive complexity

- We have established some of the elementary properties of descriptive complexity and you have had a chance to develop some intuition, we discuss some features of the definitions
- Our definition of $K(x)$ has an optimality property among all possible ways of defining descriptive complexity with algorithms
- Let us consider a general **description language** to be any computable function $p : \Sigma^* \to \Sigma^*$ and define the minimal description of x with respect to p, written $d_p(x)$, to be the first string s where $p(s) = x$, in the standard string order
- Thus, s is lexicographically first among the shortest descriptions of x. Define $K_p(x) = |d_p(x)|$
- For example, consider a programming language such as Python (encoded into binary) as the description language. Then $d_{Python}(x)$ would be the minimal Python program that outputs x, and $K_{Python}(x)$ would be the length of the minimal program
- Next theorem shows that any description language of this type is not significantly more concise than the language of Turing machines and inputs that we originally defined

Descriptive complexity

Theorem

For any description language p, a fixed constant c exists that depends only on p, where

$$\forall x[K(x) \leq K_p(x) + c]$$

Proof idea: We illustrate the idea of this proof by using the Python example. Suppose that x has a short description w in Python. Let M be a TM that can interpret Python and use the Python program for x as M's input w. Then $< M, w >$ is a description of x that is only a fixed amount larger than the Python description of x. The extra length is for the Python interpreter M.

Descriptive complexity

Proof.

Take any description language p and consider the following TM M
$M =$ On input w:

1. Output $p(w)$

Then $< M > d_p(x)$ is a description of x whose length is at most a fixed constant greater than $K_p(x)$. The constant is the length of $< M >$. □

Descriptive complexity

- Previous theorem shows that a string's minimal description is never much longer than the string itself
- Of course for some strings, the minimal description may be much shorter if the information in the string appears sparsely or redundantly
- Do some strings lack short descriptions? In other words, is the minimal description of some strings actually as long as the string itself?
- We show that such strings exist. These strings can't be described any more concisely than simply writing them out explicitly.

Descriptive complexity

Definition

Let x be a string. Say that x is **c-compressible** if

$$K(c) \leq |x| - c$$

If x is not c-compressible, we say that x is **incompressible by** c
If x is incompressible by 1, we say that x is **incompressible**

In other words, if x has a description that is c bits shorter than its length, x is c-compressible. If not, x is incompressible by c. Finally, if x doesn't have any description shorter than itself, x is incompressible.

Next, we show that incompressible strings exist

Descriptive complexity

Theorem

Incompressible strings of every length exist

Proof idea: The number of strings of length n is greater than the number of descriptions of length less than n. Each description describes at most one string. Therefore, some string of length n is not described by any description of length less than n. That string is incompressible.

Proof.

The number of binary strings of length n is 2^n. Each description is a binary string, so the number of descriptions of length less than n is at most the sum of the number of strings of each length up to $n-1$

$$\sum_{0 \leq i \leq n-1} 2^i = 2^n - 1$$

Therefore, at least one string of length n is incompressible □

Descriptive complexity

- Incompressible strings have many properties that we would expect to find in randomly chosen strings.
- For example, we can show that any incompressible string of length n has roughly an equal number of 0s and 1s,
- The length of its longest run of 0s is approximately $log_2 n$, as we would expect to find in a random string of that length.
- Proving such statements would take us too far afield into combinatorics and probability (i.e. there is a relation between fields)
- Is Kolakoski sequence compressible?

Section 4

Measuring Complexity

Example I

Let us take $A = \{0^k 1^k | k \geq 0\}$. Obviously, A is a decidable language. How much time does a single-tape TM need to decide A? We examine the following single-tape TM M_1 for A. We give the TM description at a low level, including the actual head motion on the tape so that we can count the number of steps that M_1 uses when it runs

M_1 = On input string w:

1. Scan across the tape and reject if a 0 is found to the right of a 1
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*

We will analyze the algorithm for TM M_1 deciding A to determine how much time it uses. First, we introduce some terminology and notation for this purpose.

Measuring Complexity

- The number of steps that an algorithm uses on a particular input may depend on several parameters. For instance, if the input is a graph, the number of steps may depend on the number of nodes, the number of edges, and the maximum degree of the graph, or some combination of these and/or other factors
- For simplicity, we compute the running time of an algorithm purely as a function of the length of the string representing the input and don't consider any other parameters
- In **worst-case analysis**, the form we consider here, we consider the longest running time of all inputs of a particular length. In **average-case analysis**, we consider the average of all the running times of inputs of a particular length

Measuring Complexity

Definition

Let M be a deterministic TM that halts on all inputs. The **running time** or **time complexity** of M is the function $f : N \to N$, where $f(n)$ is the maximum number of steps that M uses on any input of length n. If $f(n)$ is the running time of M, we say that M runs in time $f(n)$ and that M is an $f(n)$ time TM. Customarily we use n to represent the length of the input.

- Because the exact running time of an algorithm often is a complex expression, we usually just estimate it
- In one convenient form of estimation, called **asymptotic analysis**, we seek to understand the running time of the algorithm when it is run on large inputs
- We do so by considering only the highest order term of the expression for the running time of the algorithm, disregarding both the coefficient of that term and any lower order terms, because the highest order term dominates the other terms on large inputs

Measuring Complexity

- For example, the function $f(n) = 6n^3 + 2n^2 + 20n + 45$ has four terms and the highest order term is $6n^3$
- Disregarding the coefficient 6, we say that f is asymptotically at most n^3
- The **asymptotic notation** or big-O notation for describing this relationship is $f(n) = O(n^3)$. Formalizing we get:

Definition

Let f and g be functions $f, g : N \to R^+$. Say that $f(n) = O(g(n))$ if positive integers c and n_0 exist such that for every integer $n \geq n_0$,

$$f(n) \leq cg(n)$$

When $f(n) = O(g(n))$, we say that $g(n)$ is an **upper bound** for $f(n)$, or more precisely, that $g(n)$ is an **asymptotic upper bound** for $f(n)$, to emphasize that we are suppressing constant factors.

Measuring Complexity

- Intuitively, $f(n) = O(g(n))$ means that f is less than or equal to g if we disregard differences up to a constant factor
- You may think of O as representing a suppressed constant
- In practice, most functions f that you are likely to encounter have an obvious highest order term h. In that case, write $f(n) = O(g(n))$, where g is h without its coefficient

Example I

Let $f_1(n)$ be the function $5n^3 + 2n^2 + 22n + 6$. Then, selecting the highest order term $5n^3$ and disregarding its coefficient 5 gives $f_1(n) = O(n^3)$.

Let's verify that this result satisfies the formal definition. We do so by letting $c = 6$ and $n_0 = 10$. Then, $5n^3 + 2n^2 + 22n + 6 \leq 6n^3$ for every $n \geq 10$.

In addition, $f_1(n) = O(n^4)$ because n^4 is larger than n^3 and so is still an asymptotic upper bound on f_1. However, $f_1(n)$ is not $O(n^2)$. Regardless of the values we assign to c and n_0, the definition remains unsatisfied in this case.

Example II

The big-O interacts with logarithms in a particular way. Usually when we use logarithms, we must specify the base, as in $x = log_2 n$. The base 2 here indicates that this equality is equivalent to the equality $2x = n$.

Changing the value of the base b changes the value of $log_b n$ by a constant factor, owing to the identity $log_b n = log_2 n / log_2 b$. Thus, when we write $f(n) = O(logn)$, specifying the base is no longer necessary because we are suppressing constant factors anyway.

Let $f_2(n) = 3nlog_2 n + 5nlog_2 log_2 n + 2$. In this case, we have $f_2(n) = O(nlogn)$ because $log\ n$ dominates $loglog\ n$.

Measuring Complexity

- Big-O notation also appears in arithmetic expressions such as the expression $f(n) = O(n^2) + O(n)$. Because the $O(n^2)$ term dominates the $O(n)$ term, that expression is equivalent to $f(n) = O(n^2)$
- When the O symbol occurs in an exponent, as in the expression $f(n) = 2^{O(n)}$, the same idea applies. This expression represents an upper bound of 2^{cn} for some constant c
- The expression $f(n) = 2^{O(logn)}$ represents an upper bound of n^c for some c
- The expression $n^{O(1)}$ represents the same bound in a different way because the expression $O(1)$ represents a value that is never more than a fixed constant
- Frequently, we derive bounds of the form n^c for c greater than 0. Such bounds are called **polynomial bounds**
- Bounds of the form $2^{(n^\delta)}$ are called **exponential bounds** when δ is a real number greater than 0

Measuring Complexity

Big-O notation has a companion called **small-o notation**. Big-O notation says that one function is asymptotically *no more* than another. To say that one function is asymptotically *less than* another, we use small-o notation.

Definition

Let f and g be functions $f, g : N \to R^+$. Sat that $f(n) = o(g(n))$ if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

in other words, $f(n) = o(g(n))$ means that for any real number $c > 0$, a numbr n_0 exists, where $f(n) < cg(n)$ for all $n \geq n_0$

The difference between the big-O and small-o notations is analogous to the difference between $\leq$ and $<$.