Lecture 38: Even More Analysis
10:00 AM, Dec 5, 2018

Contents

1 Theory of Undecidability 1

2 Runtime of Hard Computational Problems 3
   2.1 Subsets ................................................................. 3
   2.2 Cryptography ............................................................. 3
   2.3 Hamiltonian Path ...................................................... 4

3 NP-Completeness 4
   3.1 Decision Problems .................................................... 4
   3.2 Reductions ............................................................... 5

1 Theory of Undecidability

We have seen that OCaml complains when the stack gets too big. Often that’s helpful—when your program is buggy and the recursion will never bottom out.

But it would be more convenient if the language could tell you that without running your code! For example, when grading your code, we want to automatically test your code on sample inputs. However, if it doesn’t terminate, this will break the grading process. OCaml can check if your program type-checks; it should also tell you if it will terminate.

Let’s consider the same task in Scheme—this is easier because an expression in Scheme can be represented as a Scheme data object.

For example, here is a buggy length procedure

(define mylength
  (lambda (L)
    (+ 1 (mylength L))))

What happens if we try to apply this procedure?

When this procedure is applied to a list, it will never terminate because all the recursive call does is add 1 to the previous call; we are not modifying the input so the recursion will never bottom out. We need to write a procedure that detects this.

Our goal is a procedure halt? that can tell us whether some expression evaluation terminates.

Conveniently, a procedure definition in Lisp can be represented as a Lisp data object (a list), e.g.
We want the halts? procedure to tell us whether the last procedure defined, if applied to a given actual argument, will terminate.

The spec

- **input**: a list $L$ of Scheme definitions, a scheme expression $E$
- **output**: Given the definitions in $L$, if the last procedure defined in $L$ is applied to $E$, will the computation terminate?

For example:

```scheme
> (halts? '((define mylength (lambda (L) (+ 1 (mylength L))))) '())
#false
```

How do we design the halts? procedure?

One strategy: analogous to Rackette, just start executing the computation. If it terminates, return #true. But what if the computation never terminates? How does the halts? procedure know when to stop executing and return #false?

The procedure cannot just blindly evaluate. It has to somehow analyze the structure of the programs to figure out if they will terminate.

For example, perhaps it could look at recursive procedures to verify that each recursive call operates on a “simpler” input than the original input. Maybe it checks whether cdr is used, or minus 1....

As hard as we think about it, we cannot come up with a set of rules that work correctly in all cases. Is that because we are not smart enough? This seems to be one of the most fundamental of computer science questions. Is computer science a failure?

In fact, there is a reason we cannot come up with such a set of rules. There is no such set of rules. Proving this was one of the originating results of computer science, the result that gave rise, I would argue, to computer science as a science.

Let’s prove why such a procedure does not exist.

Assume for a contradiction that there was a correct implementation of the above program that returns true or false depending on whether your our code terminates. Let’s write another procedure:

```scheme
(define (trouble
 (lambda (procdef)
 (if (halts? procdef procdef)
 (mylength '())
 #false))
```

The procedure trouble takes as input a procedure definition (represented as a list) and uses halt? to predict whether that procedure would terminate if given as input the definition itself. If
the program would terminate then trouble calls mylength, which would never terminate. On
the other hand, if the program would not terminate, then trouble just returns #false.

Let’s apply the procedure trouble to the list that gives the definition of trouble:

(trouble '(define (trouble
  (lambda (progdef)
    (if (halts? progdef progdef) (mylength '()) #false)))))

Assume it halts. That means the if-condition evaluated to false. The halts? procedure must
have predicted that the trouble procedure does not halt when given itself as input. That’s a
contradiction.

So, let’s assume it does not halt. That means the halts? procedure must have predicted that the
trouble procedure halts when given itself as input. That is also a contradiction.

Either way, we reach a contradiction. Therefore our assumption of the existence of halts? must be
false. This concludes the proof.

The problems halts? purports to solve is an undecidable problem.

There are all kinds of undecidable problems that seem to have nothing to do with programming. We
used the same ideas as in NP-Completeness; in fact, it came from undecidability. We used reduction.
We showed that if you could solve this new crazy problem, then you could solve the halts? problem.
So, there is a bunch of undecidable problems, and these we actually know you cannot solve.

2 Runtime of Hard Computational Problems

2.1 Subsets

Recall subset-sum. Here’s a slight variation of it.

Subset sum:

- input: sequence of numbers, \(w_1, w_2, \ldots, w_n\)
- output: subset of input numbers summing to zero

There are many variations of this problem — one variant asks if there is a subset that sums to zero,
another asks if the set can be partitioned into two groups whose sums are equal, etc. These are all
essentially the same problem. And algorithms that we have for solving this problem take about
\(O(2^n)\) time. There is a dynamic programming algorithm that runs in roughly \(O(W)\) time where
\(W = w_1 + w_2 + \ldots + w_n\). This is efficient if you are expecting small integers as input. Theoretically,
however, the traditional way to analyze algorithms is in terms of the length of the input. If we say
each number is written with \(b\) bits, then this algorithm takes \(O(2^b)\) - another exponential runtime.

We want to see if we can escape exponential dependence on input size. We have to ask: is there an
algorithm that runs in time \(O(n^3)\) or \(O(n^5)\) or even \(O(n^{10})\)? The theory of algorithms calls
an algorithm efficient if the running time is \(O(n^d)\) for some constant \(d\) where \(n\) is the input size
(measured, e.g., in bits). We call such an algorithm a polynomial-time algorithm.
2.2 Cryptography

Let’s think of another hard problem - cryptography. Let’s introduce traditional (symmetric-key) cryptography with an example. Say Alice and Bob agree on a secret key, \( k \), say an \( n \)-bit sequence such as 0010111010101010...

Later, once Alice and Bob are separated, they can still communicate privately over an insecure channel. Say Alice wants to send Bob a binary message \( m \) (called the plaintext). Alice encrypts the message using the key \( k \). This means applying an encryption procedure:

\[
\text{(define AES-encrypt (m k) ...)}
\]

The value \( c \) returned is called the ciphertext. Alice sends the ciphertext over the insecure channel. Bob receives the ciphertext, \( c \), and decrypts it, using the secret key, \( k \), with a decrypt procedure:

\[
\text{(define AES-decrypt (c k) ...)}
\]

As for implementing these procedures, there are some requirements for a secure encryption. First, it needs to be usable: decryption (with key) should take little time. There also needs to be secure against known-ciphertext attacks. Even if you know a plaintext and corresponding ciphertext, finding the key should take lots of time. As such, it should take a very long time to guess the key.

For practical cryptosystems, encryption takes linear time, and finding a key of size \( n \) should take roughly \( O(2^n) \). AES encrypts in linear time, but can it withstand a known-ciphertext attack?

**Known-ciphertext attack on AES**

- **input:** plaintext \( m \), ciphertext \( c \), key size \( n \)
- **output:** some key \( k \) for which \( m \) encrypts to \( c \)

Does the best known-ciphertext attack on AES take time \( O(2^n) \) (guessing all keys)? Or is there an attack that takes \( O(n^3) \) or \( O(n^{40}) \) time?

2.3 Hamiltonian Path

A Hamiltonian path in a graph is a path that visits every node exactly once.

**Hamiltonian path**

- **input:** a graph \( G \)
- **output:** a Hamiltonian path in \( G \)

We know an algorithm for finding a Hamiltonian path in an \( n \)-node graph takes roughly \( O(2^n) \) time, where \( n \) is the number of nodes. Is there an \( O(n^3) \) or even \( O(n^{40}) \) algorithm?
3 NP-Completeness

The theory we have to address the discussed problems is called the Theory of NP-Completeness, discussed by Michael Garey and David Johnson in “Computers and Interactivity: A Guide to the Theory of NP-Completeness.” This example comes from that book.

Say your boss assigns you to come up with a fast algorithm for “determining whether or not any given set of specifications for a new bandersnatch component can be met and, if so, for constructing a design that meets them.”

3.1 Decision Problems

Using the theory of NP-completness, we first turn problems into equivalent decision problems. A decision problem is a computational problem in which the output is a boolean.

The decision problem for the subset search problem mentioned above is:

Subset sum (decision)

- **input**: sequence of numbers, \( w_1, w_2, \ldots, w_n \)
- **output**: is there a subset of input numbers summing to zero?

The two versions are essentially equivalent: the existence of an efficient algorithm for one implies the existence of an efficient algorithm for the other.

Let’s think about how we can solve the search problem using the decision problem. First, run the decision procedure on the entire thing to determine if there is a solution. If there is, run the decision procedure on all but the first number (tail). Depending on the output, we can determine whether or not the first number is or is not included in the subset. And we can repeat this to determine the solution. This is called self-reducability.

We can do the same thing for the Hamiltonian path problem, with a slight change:

Hamiltonian path (decision)

- **input**: a graph \( G \), start node \( s \)
- **output**: is there a Hamiltonian path in \( G \) starting at \( s \)?

You ask is there a solution starting at \( s \)? If so, you have to figure out where to go next. The neighbors of \( s \) are a possibility. Delete \( s \), get a new graph, and ask if it is feasible for the smaller graph. This is another self-reducability problem.

Finally, we can do the same this for the cryptography problem, with a slight change:

Known-ciphertext attack on AES (decision)

- **input**: plaintext \( m \), cipher text \( c \), key size \( n \), key prefix \( w \)
- **output**: is there a key \( k \) with prefix \( w \) for which \( m \) encrypts to \( c \)?

Now, you can use self-reducability to fill in longer and longer prefixes.
3.2 Reductions

A reduction from decision problem $A$ to decision problem $B$ is an efficient algorithm for the following:

- **input:** an input for decision problem $A$
- **output:** an input for decision problem $B$ such that correct outputs are the same (both “yes” or “no”)

Here’s an example - a reduction from the chosen-ciphertext attack problem to the Hamiltonian path problem is an efficient algorithm for the following problem:

- **input:** plaintext $m$, cipher text $c$, key size $n$, key prefix $w$
- **output:** a graph $G$ and a starting node $s$ such that there is an $n$-bit key whose prefix is $w$ that encrypts $m$ to $c$ if and only if $G$ has a Hamiltonian path starting at $s$.

We’re not going to prove today that such a reduction exists, but it does. And, from this, we can say that if there is an efficient algorithm for the Hamiltonian path problem, then there is an efficient algorithm for the chosen-ciphertext attack problem.

You can reduce chosen-ciphertext attack problems to the subset-sum problem. You can also reduce from integer factorization or graph isomorphism problems into subset-sum or Hamiltonian path problems. And there are hundreds of other reductions we can make, including from subset-sum to Hamiltonian-path.

Hundreds of problems have been defined that can all be reduced to each other. If we knew an efficient algorithm for any of them, we would know an efficient algorithm for every one of them. What do they have in common? For each, there is an efficient algorithm for checking if a solution is indeed valid.

The decision problem is of the form “does there exist an $x$ for which the check succeeds.” This characterizes a family of computational decision problems. It is called $\text{NP}$ (nondeterministic polynomial-time).

An $\text{NP}$-complete problem is a decision problem in $\text{NP}$ that every other problem in $\text{NP}$ can be reduced to.

Is there a polynomial-time algorithm for an $\text{NP}$-complete problem? We don’t know. This is the question of whether $\text{P}=\text{NP}$. Many people have tried very hard to prove or disprove $\text{P}=\text{NP}$. There is a million-dollar prize offered to the first person to resolve the question. Many crank solutions are submitted every year; all so far have been disproved. Our ignorance is somewhat embarrassing.