3P17/4P17 COMPUTABILITY 2014-2015 Lecture Handout 12

Polynomial equivalence and reasonable encodings

One of the most important principles in this course is that different Turing machine models, and even different models of computation, are equivalent, meaning the same languages can be recognised/decided and the same functions can be computed.

There is an analogous principle in complexity, the notion of **polynomial equivalence** between models.

Before we speak about polynomial equivalence between models, we consider polynomial equivalence between encoding schemes.

For non-negative integers, we gave a specific encoding scheme; that of the minimal bit representation. For other objects, e.g., graphs, we have left it somewhat undefined, but stipulated an encoding scheme be "reasonable".

What constitutes a reasonable encoding scheme? Essentially, reasonable encoding schemes in a model of computation (e.g., Turing machines) can be converted between each other in polynomial time.

That is, if object O, e.g., a graph, is encoded in reasonable encoding scheme 1 as the string $\langle O \rangle_1$, and that same object is encoded in reasonable encoding scheme 2 as the string $\langle O \rangle_2$, then there should be a Turing machine M that can read $\langle O \rangle_1$ and convert it to $\langle O \rangle_2$, and it should be able to do so in time that is polynomial in $|\langle O \rangle_1|$

Note that this implies that the space complexity changes by a polynomial factor as well.

We have said that reasonable encoding schemes are polynomially equivalent (in time, and therefore, in space), but we need to give at least one example of a reasonable model in order to give a proper definition. For this, you may pick any one of the common encoding schemes.

Before we go further, we give an example of an *unreasonable* encoding scheme: A ***unary*** encoding of an integer is an encoding involving only a single symbol. For example, the number 8 is encoded as $********$. In binary, the number 8 is `1000`.

So here we have three encoding schemes for the same object, one in decimal, one in unary, and one in binary. They required 1, 8 and 4 symbols respectively.

In general, the non-negative integer N is encoded with $\lfloor \log_{10} N \rfloor + 1 \approx \log_{10} N$ symbols in decimal, with N symbols in unary, and $\lfloor \log_2 N \rfloor + 1 \approx \log_2 N$ in binary.

Now $\frac{\log_2 N}{\log_{10} N} = \log_2 10 = 3.32$ to 2 d.p. so the binary representation is less than 4 times longer than the decimal.

On the other hand, $\frac{N}{\log_{10} N} = \frac{10^x}{x}$, where $x = \log_{10} N$.

Thus, converting from decimal unary takes up *exponentially* more space (in x, the number of symbols using base-10), whereas converting from decimal to binary only takes about 3.32 times as much, *regardless* of what N.

So because time complexity $\geq$ space complexity, then converting a decimal (or binary) representation to unary must also take exponential time (in the number of symbols used by the decimal representation) so definitely no polynomial time equivalence in these two encoding schemes.

What is the time complexity of converting decimal to binary? If N has n digits in decimal form, then a conversion can be done in time polynomial in n, so these two encoding schemes really are polynomially equivalent.

Hence, for the purposes of encoding non-negative integers, we can say any encoding polynomially equivalent to binary is reasonable.

Recall, in M_{+1}, the input was a binary string $\langle N \rangle$ and the output was a binary string $\langle N+1 \rangle$.

If we wanted another version of M_{+1} which operated in decimal, then we could always convert it to binary in polynomial time, simulate the binary-based M_{+1}, then convert the result back to decimal in polynomial time. Thus, we will not incur more than polynomial time penalties. Hence, if the binary-based machine was polynomial time, we could always create a polynomial time decimal based machine.

What about other objects like graphs? An undirected graph G is a set of vertices and edges between them, e.g., $G = (\{1,2,3,4\}, \{(1,2),(1,3),(2,3),(1,4)\})$. This the conventional representation of graphs in mathematics. We could also represent them using an *adjacency list*: $\{(1 \mid 2,3,4),(2 \mid 1,3),(3 \mid 1,2),(4 \mid 1)\}$, or and adjacency matrix, which in this case would be a 4×4 square matrix A where $A_{ij} = 1$ if vertices i and j have a edge between them, and $A_{ij} = 0$ if they don't.

All of these are different ways of representing the same structure, i.e., they are different encodings.

It is not too hard to see that we can convert one form into another in a polynomial number of steps. Hence, if one of them is reasonable, they all are.

All these representations/encodings can be used by Turing machines with the necessary alphabet (it would have to include, for example, $\{, \}$, $(,)$), but even if a Turing machine only had alphabet $\{\mathtt{0}, \mathtt{1}\}$, you could encode those symbols (brackets etc) into binary numbers (ultimately, that is what a real computer has to do).

So we have said that if any of these representations/encodings of a graph is reasonable then they all are. We will also demand, in the case of graphs, that the space complexity of a representation is polynomial in the number of nodes. This is clearly the case for these three.

Polynomial equivalence and reasonable (deterministic) computational models

We have already shown that for two deterministic models of Turing machine - the single tape and multitape - that one can simulate the other in polynomial time.

As we have stated, reasonable models of computation (Turing machines, lambda calculus, random access machines, etc) can recognise/decide the same languages, and compute the same functions. In fact, they can simulate each other.

Now we can restrict this notion of "reasonableness" to account for time complexity: we say that reasonable ***deterministic*** models of computation are polynomially equivalent, by which we mean they can simulate each other incurring only polynomial factor increase in the running time.

Note that other reasonable models of computation will run up against many of the same issues that we have seen with Turing machines. Without actually introducing another model (which we don't have time for), the discussion necessarily remains somewhat abstract. That said, consider that a model of computation will involve taking an input, performing a sequence of steps, and giving some output. So a time complexity can always be defined in this way. Objects such as graphs or integers will always needed to be encoded in some form, so we will come up against the notion of reasonable encodings again.

To make things slightly more concrete, remember that in all of these models - Turing machines and otherwise - all of them can be precisely specified and simulated by a human using pen and paper.

Observe one important point: we have qualified the polynomial-equivalence definition of "reasonable" with them being deterministic. As we have seen a deterministic TM can simulate a non-deterministic one in time $2^{O(f(n))}$ if the non-deterministic one runs in time $f(n)$. This is emphatically not polynomial, and it is widely believed that it cannot be simulated in polynomial time. Consequently, even though NDTMs have the same

computational power (recognise/decide same languages, compute same functions) as DTMs, they are not reasonable models of computation.