Introduction to Logic in Computer Science: Autumn 2006

Ulle Endriss
Institute for Logic, Language and Computation
University of Amsterdam

Plan for Today

The next three lectures are going to be an introduction to the theory of *computational complexity*. Much of the material will be taken from Papadimitriou's textbook, although the same material can also be found in most other books on the topic.

Main issues to be covered today:

- Algorithms, problems, problem classes, complexity measures
- Big-O Notation (and variants) to describe complexity
- Definition of complexity classes, such as **P**, **NP**, **PSPACE**
- Relationships between different complexity classes

C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

Problems and Problem Classes

What can be computed at all is subject of computability (or recursion) theory. Here we deal with solvable problems, but ask how hard they are. Some examples for such *problems:*

- Find all prime numbers ≤ 500!
- Is $((P \rightarrow Q) \rightarrow P) \rightarrow P$ a theorem of classical logic?
- What is the shortest path from here to the central station?

Here we are not really interested in such specific problem instances, but rather in *classes of problems*, parametrised by their *size* $n \in \mathbb{N}$:

- Find all prime numbers $\leq n$!
- For a given formula of length $\leq n$, check whether it is a theorem of classical logic!
- Find the shortest path between two given vertices on a given graph with up to n vertices!

Decision Problems

Furthermore, we are only going to be interested in *decision problems*, problems that require "yes" or "no" as an answer.

But note that there are close connection between, say, optimisation problems and decision problems. For instance, instead of asking

> *"what is the shortest path from here to the station?"*

we may choose to ask

> *"is there a path $\leq K$ from here to the station (with, say, $K = 3km$)?"*

The two problems are not the same, but they are closely related. Standard complexity theory only deals with decision problems. Given another kind of problem, knowing the complexity of the corresponding decision problem can at least give us some pretty good indications regarding the original problem.

Graph Reachability

Let us look at a specific problem class and an algorithm for solving problems belonging to that class; and then analyse the complexity of that algorithm ...

REACHABILITY
Instance: Directed graph $G = (V, E)$ and two vertices $v, v' \in V$
Question: Is there a path leading from v to v'?

Example: Let $G = (V, E)$ with

- $V = \{1, 2, 3, 4, 5\}$
- $E = \{(1, 4), (2, 1), (2, 3), (3, 5), (4, 3), (5, 4)\}$

Is there a path leading from 1 to 5?

What would be a general algorithm for solving such a problem?

An Algorithm

Here's a generic algorithm for solving REACHABILITY.

Given: $G = (V, E)$ and $v, v' \in V$ (should find a path from v to v')

The algorithm uses two sets to maintain information:

- *FOC*: set of vertices in focus
- *VIS*: set of vertices already visited

Initially, set $FOC = VIS = \{v\}$. Then iterate:

- Choose a vertex $x \in FOC$ and remove it from FOC.
- For all edges $(x, y) \in E$ with $y \notin VIS$, add y to FOC and VIS.

Observe that this will terminate (*FOC* will eventually be empty).

Vertex v' is reachable from v *iff* $v' \in VIS$ in the end (and we could choose to interrupt the above loop as soon as v' is found).

What is the complexity of this algorithm (how long does it take)?

Complexity Measures

Our algorithm for REACHABILITY "has quadratic complexity"—what does that mean exactly?

- First of all, we have to specify the *resource* with respect to which we are analysing the complexity of an algorithm.
  - *Time complexity:* How long will it take to execute the algorithm?
  - *Space complexity:* How much memory do we need to do so?
- A second dimension of complexity is this:
  - *Worst-case analysis:* How much time/memory will the algorithm require in the worst case?
  - *Average-case analysis:* How much will it use on average?

  The latter is typically very difficult. Empirical studies using real-world data are often the only way.
- The complexity of a *problem* is the complexity of the best *algorithm* solving that problem.

The Big-O Notation

Let $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$ be two functions mapping natural numbers to natural numbers (if not, think of $f(n)$ as being short for $\max\{\lceil f(n) \rceil, 0\}$, etc.).

Think of f as computing, for any problem size n, the worst-case time complexity $f(n)$. This may be rather complicated a function.

Think of g as a function that may be a "good approximation" of f and that is more convenient when speaking about complexities. The Big-O Notation is a way of making the idea of a suitable approximation mathematically precise.

- We say that $f(n)$ is in $O(g(n))$ *iff* there exist an $n_0 \in \mathbb{N}$ and a $c \in \mathbb{R}^+$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

That is, from a certain n_0 onwards, the function f grows at most as fast as the reference function g, modulo some constant factor c about which we don't really care.

Examples

(1) Let $f(n) = \sqrt{n} + 100$. Then $f(n)$ is in $O(\sqrt{n})$.
Proof: Use $c = 2$ and $n_0 = 10000$.

(2) Let $f(n) = 5 \cdot n^2 + 20$. Then $f(n)$ is in $O(n^2)$.
Proof: Use $c = 6$ and $n_0 = 5$.

(3) Let $f(n) = 5 \cdot n^2 + 20$. Then $f(n)$ is also in $O(n^3)$, but this is not very interesting. We want complexity classes to be "sharp".

(4) Let $f(n) = 500 \cdot n^{200} + n^{17} + 1000$. Then $f(n)$ is in $O(2^n)$.
Proof: Use $c = 1$ and $n_0 = 3000$. In general, an *exponential* function always grows much faster than any *polynomial* function (we all know that; part of the homework will be to prove it formally). So $O(2^n)$ is not at all a sharp complexity class for f. A better choice would be $O(n^{200})$.

Variants of the Big-O Notation

The Big-O Notation is used to specify an upper bound (which is usually supposed to be "sharp", but it doesn't have to be).

The following notation is useful for specifying lower bounds:

$$\Omega(g(n)) = \{f : \mathbb{N} \to \mathbb{N} \mid \exists c \in \mathbb{R}^+ \; \exists n_o \in \mathbb{N} \; \forall n \geq n_0 : f(n) \geq c \cdot g(n)\}$$

A different way of putting it:

$$f(n) \in \Omega(g(n)) \text{ iff } g(n) \in O(f(n))$$

Finally, we also have this notation:

$$f(n) \in \Theta(g(n)) \text{ iff both } f(n) \in O(g(n)) \text{ and } f(n) \in \Omega(g(n))$$

Tractability and Intractability

Problems for which it is possible to devise a polynomial time algorithm are usually considered to be *tractable*. Problems that require exponential algorithms are considered *intractable*. Some remarks:

- Of course, a polynomial algorithm running in n^{1000} may behave a lot worse than an exponential algorithm running in $2^{\frac{n}{100}}$. However, experience suggests that such large factors do not actually come up for "real" problems. In any case, for very large n, the polynomial algorithm will always do better.
- It should also be noted that there *are* empirically successful algorithms for problems that are known not to be solvable in polynomial time. Such algorithms can never be efficient in the general case, but may perform very well on the problem instances that come up in practice.

The Travelling Salesman Problem

The decision problem variant of a famous problem:

Travelling Salesman Problem (TSP)
Instance: n cities; distance between each pair; $K \in \mathbb{N}$
Question: Is there a route $\leq K$ visiting each city exactly once?

A possible algorithm for TSP would be to enumerate all complete paths without repetitions and then to check whether one of them is short enough. The complexity of this algorithm is $O(n!)$.

Slightly better algorithms are known, but even the very best of these are still exponential (and *many* people tried). This suggests a fundamental problem: maybe an efficient solution is *impossible*?

Note that if someone guesses a potential solution path, then checking the correctness of that solution can be done in linear time. So *checking* a solution is a lot easier than *finding* one.

Complexity Classes

A complexity class is a set of *classes of decision problems* (or *languages*) with the same worst-case complexity.

- **TIME**$(f(n))$ is the set of all classes of decision problems that can be solved by an algorithm with a runtime of $O(f(n))$.

  For example, we have seen that REACHABILITY $\in$ **TIME**(n^2).
- **SPACE**$(f(n))$ is the set of all classes of decision problems that can be solved by an algorithm with memory requirements within $O(f(n))$.

  For instance, TSP $\in$ **SPACE**(n), because our brute-force algorithm only needs to store the route currently being tested and the route that is the best so far (together that's roughly twice the size of the input).

These are also called *deterministic* complexity classes (because the algorithms used are required to be deterministic).

Remarks

- The definitions on the previous slide are somewhat informal.
- To be absolutely precise, we should first fix a *machine model* with respect to which our algorithms are being executed. Usually, *Turing machines* are used for this.
- But once the definitions are clear, the next step is usually to understand that the precise machine model does in fact not affect the deeper ideas very much at all; so in this short introduction we can certainly do without Turing machines ...
- Another sense in which we are (and going to continue to be) somewhat informal is that functions $f : \mathbb{N} \to \mathbb{N}$ need to be restricted to *proper complexity functions* when used to define complexity classes. Broadly speaking, these are non-decreasing functions that are themselves computable within the time and space bounds they specify (see Papadimitriou for details).

Non-deterministic Complexity Classes

Remember that we said that checking whether a proposed solution is correct is different from finding one (it's easier).

We can think of a decision problem as being of the form *"is there an X with property P?"*. This may already be the chosen form (e.g. *"is there a route that is short enough?"*); or we can reformulate (e.g. *"is φ satisfiable?"* $\rightsquigarrow$ *"is there a model M s.t. $M \models \varphi$?"*).

- **NTIME**$(f(n))$ is the set of classes of decision problems for which a candidate solution can be checked in time $O(f(n))$.

  For instance, TSP $\in$ **NTIME**(n), because checking whether a given route is short enough is possible in linear time (just add up the distances and compare to K).
- Accordingly for **NSPACE**$(f(n))$.

So why are they called *non-deterministic* complexity classes?

Ways of Interpreting Non-determinism

- Think of an algorithm as being implemented on a machine that moves from one state to the next (a state is characterised by the machine's current memory configuration).

  For a *non-deterministic* algorithm the state transition function would be underspecified, and there could be more than one possible follow-up state.
- A machine is said to solve a problem using a non-deterministic algorithm iff there *exists* a run answering "yes".

  For comparison, with a deterministic machine model, there is just on possible run for each input (answering "yes" or "no").
- A common interpretation of non-determinism is that whenever there is a choicepoint in an algorithm, an *oracle* tells us which is the best computation path to pursue (think of a search tree with the oracle telling us which branch to follow).

Ways of Interpreting Non-determinism (cont.)

- The "oracle interpretation" is equivalent to our earlier interpretation based on the ability to *check* a candidate solution using the given time/space resources:

  All the "little oracles" along a computation path can be packed together into one "big initial oracle" to guess a solution; then all that remains to be done is to check its correctness.

- The "checking interpretation" is probably the best way of understanding non-deterministic complexity classes.
- The story about the oracle is important to understand where the **N** in the names is coming from.

P and NP

The two most important complexity classes:

$$\mathbf{P} = \bigcup_{k>1} \mathbf{TIME}(n^k)$$

$$\mathbf{NP} = \bigcup_{k>1} \mathbf{NTIME}(n^k)$$

From our discussion so far, you know that this means that:

- **P** is the class of problems that can be *solved* in polynomial time by a deterministic algorithm; and
- **NP** is the class of problems for which a proposed solution can be *verified* in polynomial time (or that *could* be solved by a non-deterministic algorithm in polynomial time ... if such a thing actually existed outside of mathematics).

Further Common Complexity Classes

We are also going to discuss the following complexity classes:

$$\mathbf{PSPACE} = \bigcup_{k>1} \mathbf{SPACE}(n^k)$$

$$\mathbf{NPSPACE} = \bigcup_{k>1} \mathbf{NSPACE}(n^k)$$

$$\mathbf{EXPTIME} = \bigcup_{k>1} \mathbf{TIME}(2^{(n^k)})$$

Relationships between Complexity Classes

Now that we have defined a whole range of complexity classes, we would like to understand how they relate to each other ...

The following result is obvious, given that a deterministic algorithm is just a special case of a non-deterministic one:

Proposition 1 *Let $\mathcal{C}$ be any deterministic complexity class and let $\mathbf{N}\mathcal{C}$ be the corresponding non-deterministic class. Then $\mathcal{C} \subseteq \mathbf{N}\mathcal{C}$.*

For instance, we have $\mathbf{P} \subseteq \mathbf{NP}$.

Relating Time and Space

The next result is also obvious, given that even a non-deterministic algorithm can fill up at most as many "space units" as it has "time units" at its disposal:

Proposition 2 *We have* $\mathbf{NTIME}(f(n)) \subseteq \mathbf{SPACE}(f(n))$ *for any function* $f : \mathbb{N} \to \mathbb{N}$.

As a consequence, we have $\mathbf{NP} \subseteq \mathbf{PSPACE}$.

Proposition 3 *We have* $\mathbf{PSPACE} \subseteq \mathbf{EXPTIME}$.

Proof sketch: Take any deterministic algorithm requiring a polynomially bound number of memory cells. WLOG assume each cell stores either a 0 or a 1 at any one time. During a run of the algorithm, each "memory configuration" can come up at most once (otherwise we'd enter an infinite loop). That is, if $f(n)$ cells are used, the algorithm must terminate after at most $2^{f(n)}$ steps (= the number of possible memory configurations). ✓

References

More details on the material covered today can be found in Papadimitriou's textbook.

- Read Chapter 1 for an introduction to algorithms, a definition of the Big-O Notation, and a discussion of the appropriateness of the general approach (why worst-case complexity?; and why equate polynomial complexity with tractability?).
- Chapter 2 gives a through definition of complexity classes such as $\mathbf{SPACE}(f(n))$ and $\mathbf{NTIME}(f(n))$. Note that we have taken some shortcuts here; in particular we have kept the bit on Turing machines informal.
- The rest of today's material is covered in Chapter 7.

C.H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

Summary

Topics covered today:

- Basic concepts discussed: decision problems, problem classes, algorithms, worst-case vs. average-case complexity, time vs. space complexity, (in)tractability and (super-)polynomial complexity, ...
- Big-O Notation: $O(f(n))$, as well as $\Omega(f(n))$ and $\Theta(f(n))$
- Complexity classes: grouping of problems of comparable hardness
- Relationships between different complexity classes

Topics to be covered next:

- More on relationships between complexity classes
- Complements of complexity classes (e.g. **coNP**)
- Completeness wrt. a complexity class: what are the "most difficult" problems within a given class? (e.g. **NP**-completeness)
- Reductions between problems to obtain complexity results