Appendix

David C. Parkes and Sven Seuken

March 7, 2012

In this appendix we review some of the technical background necessary to understand the content in the main chapters of the book.

100.1 O()-Notation and Polynomial Worst-Case Time Complexity

In this section, we briefly review the big-O-notation for the analysis of asymptotic time complexity of an algorithm. Let x describe the size of the input to an algorithm. For example, this could be the number of words in a text, the number of entries in a table, or the number of variables in a linear equation. Let $T(x)$ denote the *worst-case* time complexity of an algorithm, which is the *maximum* amount of time taken on *any* input of size x. Equivalently, by a re-normalization into basic computational steps, each of which takes no more than some constant amount of time, $T(x)$ can also be defined as the maximum number of basic computational steps taken on any input of size x.

The time complexity of an algorithm is often expressed *asymptotically* using big O notation. For example, an algorithm whose time complexity is $T(x) = 2x^2 + 100x^{1/2}$ has asymptotic time complexity of $O(x^2)$; in this case, we can write $T(x) = O(x^2)$ or equivalently $T(x) \in O(x^2)$. In typical usage, big O notation hides constants and if $T(x)$ is the sum of several terms, then using big O notation we only keep the term with the largest growth rate. Here, we ignore the term $x^{1/2}$ because it is dominated by term x^2 as x gets large, and we ignore the factor 2 in front of x^2.

Definition 100.1 (Big O-Notation)**.** *Let x describe the size of the input to function $T(\cdot)$. We write:*

$$T(x) \in O(g(x)) \text{ or } T(x) = O(g(x))$$

if there exists a positive real number k and a real number x_0 such that:

$$\text{for all } x > x_0 : T(x) \leq k \cdot g(x).$$

The big-O-notation is used to describe the *asymptotic* time complexity because it focuses on the grow rate of the function when the input size becomes very large (formally, as $x \to \infty$). For these very large input sizes, the term with the largest growth rate will dominate.

Given this, an algorithm is *polynomial time* if the running time is upper bounded by a polynomial expression in the size of the input. Equivalently, we can say:

Definition 100.2 (Polynomial Time Complexity)**.** *An algorithm has worst-case polynomial time complexity if $T(x) \in O(x^k)$ for some constant k.*

Polynomial-time computation is the usual yard-stick for *efficient* computation in theoretical computer science. We normally say that an algorithm is efficient if it runs in worst-case polynomial time, and we say it is *inefficient* otherwise. Of course this hides extremely important practical details, including 1) the multiplicative factors as well as 2) the exponent k that describes the asymptotic growth rate. For example, even if an algorithm has time complexity $T(x) = 1,000,000 \cdot x^2$ we write $T(x) \in O(x^2)$, and say that the algorithm is efficient. Furthermore, if an algorithm has time complexity $T(x) = x^{500} \in O(x^{500})$, the algorithm formally still has a polynomial worst-case time complexity. But it may not be practicable to run the algorithm, even for medium-sized x, because the asymptotic growth rate is too high.

Nevertheless, this distinction between algorithms that have a worst-case polynomial time complexity and those that do not, has proven very useful in computer science. For most of the algorithms with worst-case polynomial time complexity that we study, the multiplicative factors and the exponents are relatively small. In contrast, algorithms with an exponential running time such that $T(x) = 2^x$ are usually infeasible to run, except for very small input sizes.

100.2 NP-Completeness

In this section we review some basic concepts from computational complexity theory. A *decision problem* is a problem with a Yes or No answer. Example decision problems studied in computer science are

- "given a Boolean formula, is there a satisfying truth assignment?"
- "given a graph, is there a way to color each vertex with one of k colors such that no adjacent vertices share the same color?"
- "given a set of items, each with some weight and value, and a bag with some capacity, is there a subset of items with total value greater than some target value, and with total weight that does not exceed the capacity?"

These correspond to the decision problems SATISFIABILITY, GRAPH COLORING and 0-1 KNAPSACK. Decision problems have a yes-or-no answer. Instances of a decision problem are either "yes"-instances (the answer is Yes) or "no"-instances (the answer is No.)

Computational complexity theory provides the following distinctions in regard to the complexity of different decision problems:

- A **polynomial decision problem** (the problem is in **P**) is one for which there exists a worst-case polynomial time algorithm that answers *Yes* on every "yes"-instance and *No* on every "no"-instance.
- A **non-deterministic polynomial decision problem** (the problem is in **NP**) is one for which there exists a worst-case polynomial time algorithm that can *verify* that the answer is *Yes* on every "yes"-instance, given a proof as input in addition to the instance; e.g., for GRAPH COLORING, the proof would be an assignment of a color to each vertex, and the algorithm would check that no pair of adjacent vertices have the same color.[1]
- An **NP-hard decision problem** (the problem is **NP**-hard) need not be in **NP**, but is at least as hard to solve as the hardest problems in **NP**.[2] In particular, a problem A is **NP**-hard if

[1] Note that (i) to be in **NP** a problem does not require an easy-to-verify proof for "no"-instances (when this exists, it is a **co-NP** problem); (ii) the size of the proof must be polynomial in the size of the input if it is to be checked in polynomial time; and (iii) there is an equivalent definition in terms of a non-deterministic theoretical computational machine (a Turing machine), and it is from this that the name **NP** derives.

[2] For example, the class **NP**-hard also contains non-decision problems such as *optimization problems* (e.g., find the

an algorithm to solve the problem in worst-case polynomial-time would imply an algorithm to solve any problem B that is in **NP** in worst-case polynomial time.

This would be achieved through a **reduction**, where an instance of B is encoded as an instance of A such that a solution to A gives a solution to B, and where the instance of A is at most polynomially larger than the instance of B.

- An **NP-complete** decision problem (the problem is **NP**-complete) is one for which the problem is both in **NP** and also **NP**-hard.

Intuitively, problems in **P** are "easy" to solve. But what about problems in **NP**? On the one hand, there is at least a fast way to verify that a "yes"-instance is indeed a yes instance. Certainly, every decision problem in **P** is also in **NP**. One can simply use the polynomial time algorithm that solves the decision problem as the polynomial-time verifier. But what about polynomial-time algorithms to solve the hard problems in **NP**? This leads us to the following:

Conjecture 1. $\mathbf{P} \neq \mathbf{NP}$.

This famous conjecture states that the class **P** is a strict subset of **NP**, which in particular requires that *no problem that is* **NP***-hard is in* **P**. To see this, notice that if a problem A that was **NP**-hard was in **P** then by the definition of **NP**-hard, all problems in **NP** could be solved by the polynomial-time algorithm for A.

Proving that "$\mathbf{P} \neq \mathbf{NP}$" is the principal unsolved problem in theoretical computer science today. This conjecture is widely believed for two reasons. First, a large number of different **NP**-complete problems can be reduced to each other, in the sense that if one can be solved in worst-case polynomial time, then all others can also be solved in worst-case polynomial time. Second, a great deal of effort has been dedicated to finding efficient algorithms for the various **NP**-complete problems, but none has been found so far.

Now remember our discussion of what constitutes an *efficient* algorithm and what constitutes an *inefficient* algorithm. We can now say, that for a problem that is in **P**, there exist efficient algorithms for solving it. And under the assumption that $\mathbf{P} \neq \mathbf{NP}$, we can say that for a problem that is **NP**-hard, there does not exist an efficient algorithm for solving it.

subset of items that maximizes total value is the "optimization variant" on the 0-1 KNAPSACK problem.)