Introduction to
Logic in Computer Science: Autumn 2007

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

Computational Complexity

This part of the course will 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.

- Algorithms, problems, problem classes, complexity measures
- Big-O Notation (and variants) to describe complexity
- Definition of complexity classes, e.g. **P**, **NP**, **coNP**, **PSPACE**
- Relationships between different complexity classes
- Completeness with respect to a complexity class
- Examples for **NP**- and **PSPACE**-complete problems
- Proving complexity results by means of reduction

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 connections 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 "has quadratic complexity"—but what does that mean?

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?

Then we might be interested in a worst-case or an average-case analysis.

- *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?

Giving a formal average-case analysis that is theoretically sound is typically very difficult (where does the input distribution come from?). 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. 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:

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

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.

Summary and Further Reading

Topics covered so far:

- 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 $\Theta(f(n))$ to define the complexity of a problem/algorithm

More details 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?).

Complexity Classes

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

- $\mathbf{TIME}(f(n))$ 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 $\in \mathbf{TIME}(n^2)$.
- $\mathbf{SPACE}(f(n))$ 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 $\in \mathbf{SPACE}(n)$, 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.

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?"* $\leadsto$ *"is there a model M s.t. $M \models \varphi$?"*).

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

  For instance, TSP $\in \mathbf{NTIME}(n)$, 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 $\mathbf{NSPACE}(f(n))$.

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:

$$\begin{aligned}\mathbf{P} &= \bigcup_{k>1} \mathbf{TIME}(n^k)\\ \mathbf{NP} &= \bigcup_{k>1} \mathbf{NTIME}(n^k)\end{aligned}$$

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:

$$\begin{aligned}\mathbf{PSPACE} &= \bigcup_{k>1} \mathbf{SPACE}(n^k)\\ \mathbf{NPSPACE} &= \bigcup_{k>1} \mathbf{NSPACE}(n^k)\\ \mathbf{EXPTIME} &= \bigcup_{k>1} \mathbf{TIME}(2^{(n^k)})\end{aligned}$$

Summary and Further Reading

This is where we are now:

- We have given definitions of several *complexity classes*. Examples include **SPACE**$(f(n))$ and **NP**, amongst others. These are groups of problems of comparable hardness.
- Note that we have taken some shortcuts here; in particular we have kept the bit on *Turing machines* informal.
- Chapter 2 of Papadimitriou's textbook gives a more thorough definition of complexity classes (recommended in particular if you have not come across Turing machines before).

What next?

The remaining lectures on computational complexity will basically be concerned with three important topics:

- Establishing *relationships* between different complexity classes (e.g. one does or does not include the other, etc.).
- Providing a definition of what are the most difficult problems within a given complexity class (theory of *completeness*).
- Examples for how to prove whether a given problem belongs to or is complete with respect to a given complexity class.

  We usually show completeness for one problem only and then *reduce* any other problems to that reference problem.

Remark: Any results stated in the sequel assume that all functions $f : \mathbb{N} \to \mathbb{N}$ involved are "proper complexity functions".

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 $\mathbf{NP} \subseteq \mathbf{PSPACE}$.

Relating Space and Time

This one is a bit more tricky:

Proposition 3 *We have* **PSPACE** $\subseteq$ **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). ✓

The Time Hierarchy Theorem

Not all complexity classes are the same (we weren't sure before!):

Theorem 1 *We have* $\mathbf{TIME}(f(n)) \subset \mathbf{TIME}((f(2n+1))^3)$ *for any function* $f : \mathbb{N} \to \mathbb{N}$ *with* $f(n) \geq n$.

Proof idea: Define a *time-bound halting problem* as follows:

$$H_f = \{M; x \mid \text{machine } M \text{ accepts } x \text{ after} \leq f(|x|) \text{ steps}\}$$

Then prove the following lemmas:

(1) $H_f \in \mathbf{TIME}((f(n))^3)$: Roughly, each step can be simulated in quadratic time; so the whole thing takes cubic time (see book).

(2) $H_f \notin \mathbf{TIME}(f(\lfloor \frac{n}{2} \rfloor))$
Proof by contradiction: Suppose there exists a machine M_{H_f} deciding H_f in $f(\lfloor \frac{n}{2} \rfloor)$. Define machine D_f such that

$D_f(M)$ says "yes" *iff* $M_{H_f}(M; M)$ says "no" (else "no")

Then wonder what the output for $D_f(D_f)$ would be ... ✓

P and EXPTIME

Recall the theorem: $\mathbf{TIME}(f(n)) \subset \mathbf{TIME}((f(2n+1))^3)$.

Corollary 1 *We get* **P** $\subset$ **EXPTIME**.

Proof: Apply the Time Hierarchy Theorem to $f(n) = 2^n$, yielding $\mathbf{TIME}(2^n) \subset \mathbf{TIME}(2^{(2n+1)\cdot 3})$. The result then follows from $\mathbf{P} \subseteq \mathbf{TIME}(2^n)$ and $\mathbf{TIME}(2^{(2n+1)\cdot 3}) \subseteq \mathbf{EXPTIME}$. ✓

Savitch's Theorem

Savitch's Theorem is a non-trivial result relating deterministic and non-deterministic polynomial space:

Theorem 2 (Savitch) $\mathbf{NSPACE}(f(n)) \subseteq \mathbf{SPACE}(f^2(n))$ *for any function* $f : \mathbb{N} \to \mathbb{N}$ *with* $f(n) \geq \log n$.

Proof idea: First show that Reachability $\in \mathbf{SPACE}(\log^2 n)$:

- Let Path(x, y, t) be the problem: is there a path $\leq t$ from x to y?
  Algorithm: Recursively try all nodes z and check if both Path$(x, z, t/2)$ and Path$(z, y, t/2)$ hold.
  Analysis: Recursion depth is at most $\log t$. Storing one node takes $\log n$ space (binary representation). The claim then follows from Reachability = Path(x, y, n).

Now take any problem $\in \mathbf{NSPACE}(f(n))$. The machine solving it can have at most $c^{f(n)}$ states. Solve Reachability for the corresponding graph of $\leq c^{f(n)}$ nodes $\Rightarrow$ problem $\in \mathbf{SPACE}(f^2(n))$. ✓

Consequences

Recall Savitch's Theorem: $\mathbf{NSPACE}(f(n)) \subseteq \mathbf{SPACE}(f^2(n))$.

Corollary 2 *We get* **PSPACE** = **NPSPACE***.*

Compare this with the situation for *time:* we don't know whether $\mathbf{P} = \mathbf{NP}$, but strongly suspect *not*, namely $\mathbf{P} \subset \mathbf{NP}$.

Summary of Complexity Class Relationships

This is what we know so far:

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} = \mathbf{NPSPACE} \subseteq \mathbf{EXPTIME}$$
$$\mathbf{P} \subset \mathbf{EXPTIME}$$

Hence, one of the $\subseteq$'s above must actually be strict, but we don't know which. Most experts believe they are probably all strict. In the case of $\mathbf{P} \subset^? \mathbf{NP}$, the answer is worth \$1.000.000.

Complements

- Let P be a class of decision problems. The *complement* $\overline{P}$ of P is the set of all instances that are *not* correct solutions for P.

  If you think of a class of problems as a *language*, then this is the usual notion of complementation.

- Example: Sat is the problem of checking whether a given formula of propositional logic is satisfiable. The complement of Sat is checking whether a given formula is *not* satisfiable (which is equivalent to checking whether its negation is valid).

- For any complexity class $\mathcal{C}$, we define $\mathbf{co}\mathcal{C} = \{\overline{P} \mid P \in \mathcal{C}\}$.

- Example: **coNP** is the class of problems for which a negative answer can be verified in polynomial time.

Problems and Languages

Sometimes a small shift in terminology can make things clearer . . .

A class of problems can be understood as a *language* over some given alphabet: each problem instance can be encoded as a word over the alphabet, and the language is the set of all words corresponding to problem instances with a positive answer.

For instance, for Reachability, we can describe any graph $G = (V, E)$ together with two chosen vertices as a string $\subseteq \{0, 1\}^*$. The language corresponding to Reachability would then be the set of words encoding an instance of this class of problems for which the correct answer would be "yes".

The complement of a language L (class of problems) with respect to an alphabet Σ is then easily defined as $\overline{L} = \Sigma^* \setminus L$.

A complexity class $\mathcal{C}$ is said to be *closed* under complementation *iff* $L \in \mathcal{C}$ implies $\overline{L} \in \mathcal{C}$ (similar for union and intersection).

Results for Complements

The following result follows from the fact that a deterministic algorithm for a given problem can be turned into a deterministic algorithm for the complement of that problem by simply swapping "yes" and "no" in the output.

Proposition 4 *$\mathcal{C} = \mathbf{co}\mathcal{C}$ for any deterministic complexity class $\mathcal{C}$.*

For example, we have $\mathbf{P} = \mathbf{coP}$. But nobody knows whether $\mathbf{NP} \stackrel{?}{=} \mathbf{coNP}$ (people tend to think not).

The Immerman-Szelepscényi Theorem

An important theorem that we just state here without proof:

Theorem 3 (Immerman-Szelepscényi) *If $f(n) \geq \log n$, then $\mathbf{NSPACE}(f(n)) = \mathbf{coNSPACE}(f(n))$.*

That is, non-deterministic space is closed under complementation. An example would be $\mathbf{NPSPACE} = \mathbf{coNPSPACE}$.

Summary and Further Reading

We have established several results relating different complexity classes:

- Non-deterministic is more powerful than deterministic: $\mathcal{C} \subseteq \mathbf{N}\mathcal{C}$
- Space is more powerful than time: $\mathbf{NTIME}(f(n)) \subseteq \mathbf{SPACE}(f(n))$
- . . . unless we have *much* more time: $\mathbf{PSPACE} \subseteq \mathbf{EXPTIME}$
- The Time Hierarchy Theorem confirms that the hierarchy of time complexity classes does not collapse, particularly $\mathbf{P} \subset \mathbf{EXPTIME}$.
- Savitch's Theorem shows that non-determinism does not help further for polynomial space: $\mathbf{PSPACE} = \mathbf{NPSPACE}$
- The complement of a problem is as hard as the original for deterministic algorithms: $\mathcal{C} = \mathbf{co}\mathcal{C}$ for deterministic classes $\mathcal{C}$
- By the Immerman-Szelepscényi Theorem, also non-det. space is closed under complements: $\mathbf{NSPACE}(f(n)) = \mathbf{coNSPACE}(f(n))$

This material is covered in Chapter 7 of Papadimitriou's book.

Reductions

To compare the hardness of different classes of problems, we introduce the concept of reduction.

Problem A *reduces* to problem B if we can translate any instance of A into an instance of B that we can then feed into a solver for B to obtain an answer to our original question (of type A).

If the translation (reduction) process itself is not too complex, then we can rightfully claim that problem B *is at least as hard* as problem A. This is because a B-solver can solve any instance of A, and maybe many more problems.

There are several ways of making this notion precise . . .

Polynomial-Time Reductions

The important bit is that it has to be possible that the reduction be computed in *polynomial time* (this makes sense if we are interested in complexity classes **P** and above). Two variants:

- *Karp reduction* (aka. *many-one reduction*): Reduce an A-instance to a B-instance in polynomial time and then solve it using the B-solver (one call at the very end of the reduction).
- *Turing reduction* (aka. *Cook reduction*): Solve an A-instance in polynomial time using a polynomial number of B-oracles.

We'll usually gloss over the details (as is common practice in the literature) and not explicitly state which type of reduction we use.

Remark: Papadimitriou uses logarithmic-space rather than polynomial-time reductions in his book . . . more powerful as he explains, but also less standard.

Hardness and Completeness

Let $\mathcal{C}$ be a complexity class.

- A language L is $\mathcal{C}$-*hard* iff any $L' \in \mathcal{C}$ is polynomial-time reducible to L. That is, the $\mathcal{C}$-hard problems include the very hardest problems inside of $\mathcal{C}$, and even harder ones.
- A language L is $\mathcal{C}$-*complete* iff L is $\mathcal{C}$-hard and $L \in \mathcal{C}$. That is, these are the hardest problems in $\mathcal{C}$, and only those.

Example: As we shall see later, Sat is an example for an **NP**-complete problem. That means, it is as hard as any other problem in **NP**; any other such problem can be reduced to Sat.

What next?

Now that we have a basic understanding of what complexity classes are and how they relate to each other, we will identify specific problems that are complete wrt. interesting complexity classes:

- The original **NP**-complete problem: the satisfiability problem for propositional logic
- Variants of the satisfiability problem, some of which are also **NP**-complete, and some of which are not
- **NP**-complete problems in graph theory
- An example for a **PSPACE**-complete problem: satisfiability for quantified boolean formulas

For each complexity class we will have to prove completeness (of some representative problem) *once* from first principles; all subsequent results rely on reductions.

Cook's Theorem

The first decision problem ever to be shown to be **NP**-complete is the satisfiability problem for propositional logic.

Satisfiability (Sat)
Instance: Propositional formula φ
Question: Is φ satisfiable?

The *size* of an instance of Sat is the length of φ. Clearly, Sat can be solved in *exponential* time (by trying all possible models), but no (deterministic) *polynomial* algorithm is known.

Theorem 4 (Cook) Sat *is* **NP**-*complete.*

Proof: **NP**-membership is easy to show: if someone guesses a satisfying assignment of truth values to propositional letters (model), then we can check its correctness in polynomial time. ✓

The proof of **NP**-hardness is sketched on the following slides . . .

Proof of NP-Hardness (1)

We need to show that *any* problem in **NP** can be reduced to SAT. By definition, for any problem in **NP** there is a non-deterministic algorithm that accepts or rejects all instances of the problem in polynomial time. We now sketch how to encode this algorithm as a propositional formula.

Suppose the algorithm runs in time $\leq n^k$. So it will use at most n^k memory cells. Hence, there *exists* (non-determinism!) a table of size $n^k \times n^k$ for each possible run of the algorithm, with each table cell holding an element belonging to some (small) alphabet.

Introduce a propositional variable x_{ijs} for each row i, column j, symbol s, saying whether or not cell (i,j) is holding symbol s. The number of these variables is polynomial in n.

If we can fully specify all constraints these x_{ijk} have to meet to represent a well-formed computation table for an accepting run as a formula (of polynomial size), then we are done, as that formula will be satisfiable iff there is an accepting run.

Proof of NP-Hardness (2)

Follows Sipser (1997). Each table cell must hold exactly one symbol:

$$\varphi_{cell} \quad = \quad \bigwedge_{1\leq i,j\leq n^k} \left[\bigvee_{s\in S} x_{ijs} \;\wedge \bigwedge_{s,s'\in S} \neg(x_{ijs} \wedge x_{ijs'}) \right]$$

We can also encode the initial configuration as a conjunction φ_{start}, and the characteristics of an accepting configuration as a formula φ_{accept}.

To encode what is a legal move from one configuration to the next, we need to be more specific about our machine model. As any algorithm can be implemented on a Turing machine, we may assume that the contents of memory cell (i,j) will only depend on the content of its predecessor and that cell's neighbours: $(i-1,j-1)$, $(i-1,j)$, $(i-1,j+1)$.

We can write a formula $\varphi_{move(i,j)}$ specifying all possible moves of this sort for (i,j). Then $\varphi_{move} = \bigwedge_{1\leq i,j\leq n^k} \varphi_{move(i,j)}$ describes *all* legal moves. Finally, $\varphi_{cell} \wedge \varphi_{start} \wedge \varphi_{accept} \wedge \varphi_{move}$ encodes the algorithm. ✓

M. Sipser. *Introduction to the Theory of Computation.* Thomson, 1997.

Checking Tautologies

This is a simple corollary to Cook's Theorem:

Corollary 3 *Checking whether a given propositional formula is a tautology is a* **coNP***-complete decision problem.*

Proof: A formula φ is a tautology iff $\neg\varphi$ is satisfiable. Hence, tautology checking is the complementary problem of satisfiability checking, and the claim follows from Cook's Theorem. ✓

Variants of Satisfiability

If we restrict the structure of propositional formulas, then there's a chance that the satisfiability problem will become easier.

k-SATISFIABILITY (kSAT)
Instance: Conjunction φ of disjunctions of k literals each
Question: Is φ satisfiable?

Alternative formulation: check whether a given set S of k-clauses is satisfiable.

By close inspection of the proof for SAT, it is possible to show (see Papadimitriou's book for details):

Theorem 5 3SAT *is* **NP***-complete.*

Only once we go down from 3 to 2, we get a positive result . . .

2SAT is in P

Theorem 6 2Sat *is in* **P**.

Proof sketch: Recall that Reachability $\in$ **P**. We are going to reduce 2Sat to Reachability.

Let φ be a formula in CNF with clauses of length 2 and let P be the set of propositional variables in φ.

Build a graph $G = (V, E)$ with $V = P \cup \{\neg p \mid p \in P\}$ and $(x, y) \in E$ iff a there is a clause in φ that is equivalent to $x \rightarrow y$.

Observe that φ is *unsatisfiable* iff there is a $p \in P$ such that there is both a path from p to $\neg p$ and from $\neg p$ to p in G.

This condition can be tested by running our algorithm for Reachability several times. ✓

Counting Clauses

If not all clauses of a given formula in CNF can be satisfied simultaneously, what is the maximum number of clauses that can?

Maximum k-Satisfiability (Max$k$Sat)
Instance: Set S of k-clauses and $K \in \mathbb{N}$
Question: Is there a satisfiable $S' \subseteq S$ such that $|S'| \geq K$?

For this kind of problem, we cross the border between **P** and **NP** already for $k = 2$ (rather than $k = 3$, as before):

Theorem 7 Max2Sat *is* **NP**-*complete.*

Proof sketch: Max2Sat is clearly in **NP**: if someone guesses an $S' \subseteq S$ and a model with $|S'| \geq K$, we can check whether S' is true in that model in polynomial time. ✓

Next we show **NP**-hardness by reducing 3Sat to Max2Sat ...

Reduction from 3SAT to MAX2SAT

Consider the following 10 clauses:

$$(x),\ (y),\ (z),\ (w),$$
$$(\neg x \vee \neg y),\ (\neg y \vee \neg z),\ (\neg z \vee \neg x),$$
$$(x \vee \neg w),\ (y \vee \neg w),\ (z \vee \neg w)$$

Observe: any model satisfying $(x \vee y \vee z)$ can be extended to satisfy (at most) 7 of them; all other models satisfy at most 6 of them.

Given an instance of 3Sat, construct an instance of Max2Sat: For each clause $C_i = (x_i \vee y_i \vee z_i)$ in φ, write down these 10 clauses with a new w_i. If the input has n clauses, set $K = 7n$.

Then φ is satisfiable iff (at least) K of the 2-clauses in the new problem are satisfiable. ✓

Satisfiability for Horn Clauses

Another restriction of the satisfiability problem:

Horn Satisfiability (HornSat)
Instance: Conjunction φ of Horn clauses
Question: is φ satisfiable?

Recall that Prolog is built around Horn clauses.

Theorem 8 HornSat *is in* **P**.

Proof sketch: The following (polynomial) algorithm does the job. Distinguish negative Horn clauses: $(\neg x_1 \vee \cdots \vee \neg x_k)$; and implications (including atoms: $x_1 = \mathit{true}$): $(x_1 \wedge \cdots \wedge x_\ell \rightarrow y)$.

Initially, make all variables *false*. Iterate: if $(x_1 \wedge \cdots \wedge x_\ell \rightarrow y)$ not satisfied, then make y true $\rightsquigarrow$ *minimal* model M for implications.

Full formula can only be satisfiable if M satisfies all negative clauses as well. For if not, we'd get $\{x_1, \ldots, x_k\} \subseteq M$ and we'd have to make one of the x_i *false*, thereby contradicting minimality. ✓

Independent Sets

Many conceptually simple problems that are **NP**-complete can be formulated as problems in graph theory. Example:

Let $G = (V, E)$ be an *undirected* graph. An *independent set* is a set $I \subseteq V$ such that there are no edges between any of the vertices in I.

Independent Set
Instance: Undirected graph $G = (V, E)$ and $K \in \mathbb{N}$
Question: Does G have an independent set I with $|I| \geq K$?

Theorem 9 Independent Set *is* **NP**-*complete.*

Proof sketch: **NP**-membership: easy ✓
NP-hardness: by reduction from 3Sat with n clauses —
Given a conjunction φ of 3-clauses, construct a graph $G = (V, E)$.
V is the set of occurrences of literals in φ. Edges: make a "triangle" for each 3-clause, and connect complementary literals. Set $K = n$.
Then φ is satisfiable iff there is an independent set of size K. ✓

More Graph-theoretic Problems

All of the following are also **NP**-complete problems. They each take an undirected graph $G = (V, E)$ and an integer K as input.

- Clique: Is there a $V' \subseteq V$ with $|V'| \geq K$ such that any two vertices in V' are joined by an edge in E?
- Vertex Cover: Is there a $V' \subseteq V$ with $|V'| \leq K$ such that for any edge $(x, y) \in E$ either $x \in V'$ or $y \in V'$?
- Hamilton Path: Does G have a Hamilton path (that is, a path visiting every vertex)? (no K needed as input)
- Travelling Salesman Problem: Is there a path $\leq K$ visiting each vertex once? (additional input: distances)
- MaxCut: Is there a $V' \subseteq V$ such that the number of edges between nodes in V' and in $V \setminus V'$ exceeds K?

Quantified Boolean Formulas

The canonical example for a **PSPACE**-complete problem is the satisfiability problem for *quantified boolean formulas.*

A quantified boolean formula (QBF) is a propositional formula φ preceded by either $\forall x$ or $\exists x$ for each propositional variable in φ:

$$\forall x.\exists y.\forall z.[(x \rightarrow y) \vee z]$$

The semantics is exactly what you would expect it to be ...

This gives rise to a new decision problem:

Quantified Satisfiability (QSat)
Instance: Quantified boolean formula φ
Question: Is φ satisfiable (true)?

The size of a QSat instance is the length of φ.

PSPACE-Completeness

Theorem 10 QSat *is* **PSPACE**-*complete.*

Proof idea: **PSPACE**-membership: Recursively go through the formula in the obvious manner. The recursion depth is given by the number of propositional variables, and at each state we have to remember what "satisfiability requirements" we still need to meet (polynomial space). ✓

PSPACE-hardness: We need to chow that any problem in **PSPACE** can be encoded as a QBF. Use the same idea as for Cook's Theorem: encode the computation table as a formula.
Problem: the number of rows in the table might be exponential.
Solution: Use the idea from the proof of Savitch's Theorem and write a QBF expressing that you can go from configuration c_1 to configuration c_2 in t steps if there's another configuration c such that you can go from c_1 to c and from c to c_2 in $\lceil \frac{t}{2} \rceil$ steps each ...

Remark

The satisfiability and the validity problem coincide for QBFs. This matches up nicely with **PSPACE** = **coPSPACE**.

For comparison, in propositional logic these are distinct problems and we don't know whether **NP** $=^?$ **coNP**.

Summary

We have seen a range of **NP**-complete problems:

- Logic: Sat, 3Sat, Max2Sat, but *not* 2Sat and HornSat
- Graph Theory: Independent Set and others

Recall that the **P**-**NP** borderline is widely considered to represent the move from tractable to intractable problems, so developing a feel for what sort of problems are **NP**-complete is important to understand what can and what cannot be computed in practice.

In logic, in particular, we also encounter problems that are a lot harder: the **PSPACE**-completeness result for QSat is an example.

Remember: Complexity proofs need not be difficult. Once completeness has been established for a reference problem, further results can be obtained by reduction.

Further Reading

All material on reductions and completeness results can be found in Papadimitriou's textbook:

- Cook's Theorem is proved in Chapter 8, and further **NP**-completeness results are established in Chapter 9.
- The **PSPACE**-completeness proof for quantified boolean formulas may be found in Chapter 19.

For large collections of **NP**-complete problems, the books by Garey and Johnson (1979) and Ausiello *et al.* (1999) are useful references.

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

M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, 1979.

G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation.* Springer-Verlag, 1999.

Conclusion

- Topics covered: definition of complexity classes; relationships between complexity classes; definition of hardness and completeness wrt. a class; proving complexity results directly and by reduction
- Really understanding complexity is difficult and requires some dedication (we have glossed over some of the technical details).
- But most logicians and computer scientists are only *users* of complexity theory: a proof by reduction need not be that difficult, but is still very useful.
- For every problem you study, every algorithm you propose, every logic you invent . . . it's always a good idea to ask yourself how complex these are: this puts your work into perspective.