Computational Social Choice: Spring 2009

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

Plan for Today

This will be a tutorial on Complexity Theory. Topics covered:

- Definition of complexity classes in terms of time and space requirements of algorithms solving problems
- Notion of hardness and completeness wrt. a complexity class
- Proving NP-completeness results

The focus will be on *using* complexity theory in other areas, rather than on learning about complexity theory itself.

Much of the material is taken from Papadimitriou's textbook, but can also be found in most other books on the topic.

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

Problems

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

- Is $((P \to Q) \to P) \to P$ a theorem of classical logic?
- What is the shortest path from here to the central station?

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

- 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! (*or:* is there a path $\leq K$?)

Finally, we will only be interested in *decision problems*, problems that require "yes" or "no" as an answer.

Example

Problems will be defined like this:

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

It is possible to solve this problem with an algorithm that has "quadratic complexity" — what does that mean?

Complexity Measures

First, 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 run the algorithm?
- *Space complexity:* How much memory do we need to do so?

Then, we can distinguish worst-case or average-case complexity:.

- *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?

But giving a formal average-case analysis that is theoretically sound is difficult (where will the input distribution come from?).

The complexity of a *problem* is the complexity of the best *algorithm* solving that problem.

The Big-O Notation

Take two functions $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$.

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.

Tractability and Intractability

Problems that permit polynomial time algorithms 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.

Deterministic Complexity Classes

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

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

  For example, Reachability $\in$ **TIME**(n^2).

- **SPACE**$(f(n))$ is the set of all decision problems that can be solved by an algorithm with memory requirements in $O(f(n))$.

  For example, 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.

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

Nondeterministic 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 *nondeterministic* complexity classes?

Ways of Interpreting Nondeterminism

- Think of an algorithm as being implemented on a *machine* that moves from one state (memory configuration) to the next. For a nondeterministic algorithm the state transition function is underspecified (more than one possible follow-up state).

  A machine is said to solve a problem using a nondeterministic algorithm iff there *exists* a run answering "yes".

- We can think of this as an *oracle* that tells us which is the best way to go at each choicepoint in the algorithm.

- This view is equivalent to interpreting nondeterminism as the ability to *check correctness* of a candidate solution: 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.

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.

Other Common Complexity Classes

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

Relationships between Complexity Classes

The following inclusions are known:

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

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.

Remarks: **PSPACE** = **NPSPACE** is Savitch's Theorem; **P** $\subset$ **EXPTIME** is a corollary of the Time Hierarchy Theorem; the other inclusions are easy.

Complements

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

  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.

- Clearly, **P** = **coP**. But nobody knows whether $\mathbf{NP} =^? \mathbf{coNP}$ (people tend to think not).

Polynomial-Time Reductions

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 process is "easy" (*polynomial*), then we can claim that problem B *is at least as hard* as problem A (as a B-solver can then solve any instance of A, and possibly a lot more).

So, to prove that problem B is at least as hard as problem A:

- Show how to translate any A-instance into a B-instance in polynomial time; and then
- show that the answer to the A-instance should be YES *iff* a B-solver will answer YES to the translated problem.

Hardness and Completeness

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

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

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 1 (Cook, 1971) *SAT is* **NP***-complete.*

The proof is difficult, and we shall not discuss it here.

Corollary 1 *Checking whether a given propositional formula is a tautology is* **coNP***-complete.*

S. Cook. *The Complexity of Theorem-Proving Procedures.* Proc. STOC-1971.

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 k-clauses
> **Question:** Is φ satisfiable?

(A k-clause is a disjunction of (at most) k literals.)

A variant of Cook's Theorem, again without proof (also difficult), shows that it does in fact not get any easier, as long as $k \geq 3$:

Theorem 2 *3SAT is* **NP***-complete (but 2SAT is in* **P***).*

But now that we have this result, we can get a lot of other results using reduction ...

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 (MAXkSAT)
> **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 3 *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. ✓

Independent Sets

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

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 4 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. ✓

Summary

We have covered the following topics:

- Definition of complexity classes: **P**, **NP**, **coNP**, **PSPACE**, ...
- Relationships between complexity classes
- Hardness and completeness wrt. a complexity class

Examples for **NP**-complete problems include:

- Logic: SAT, 3SAT, MAX2SAT (but not 2SAT)
- Graph Theory: INDEPENDENT SET

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.

You should be able to interpret complexity results, and to carry out simple reductions to prove **NP**-completeness results yourself.

Literature

Helpful textbooks include:

- C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, 1994.
- M. Sipser. *Introduction to the Theory of Computation*. Course Technology, 1996.

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

- M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman & Co., 1979.
- G. Ausiello *et al.* *Complexity and Approximation*. Springer, 1999. See also: `http://www.nada.kth.se/~viggo/wwwcompendium/`