Computational Social Choice: Spring 2019

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

Plan for Today

A central technique used in COMSOC is the complexity-theoretic analysis of algorithmic problems arising in the context of social choice.

This will be a tutorial on *computational complexity theory*. Topics:

- Definition of complexity classes, in terms of time and space requirements of algorithms solving problems
- Hardness and completeness of a problem w.r.t. a complexity class
- Proving $\mathbf{NP}$-completeness
- Brief review of a few complexity classes beyond $\mathbf{NP}$

Our focus will be on *using* complexity theory to analyse problems, rather than on the 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 the subject of computability theory.
Here we deal with solvable problems, but ask how hard they are.
Some examples of such (computational) *problems:*

- Is $((p \to q) \to p) \to p$ a theorem (tautology) 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 $\leqslant n$, check whether it is a tautology!
- Find the shortest path between two given vertices on a given graph with up to n vertices! (or: is there a path $\leqslant K$?)

And we are only interested in *decision problems* (YES-NO questions).

Example

Problems will be defined like this:

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

Clearly, we can design an algorithm to solve any such problem.

But how efficiently?

Complexity Measures

Analysing the performance of an algorithm means measuring its usage of resources. There are different *resources* we might focus on:

- *Time:* How long will it take to run the algorithm?
- *Space:* How much memory do we need to do so?

There are different paradigms for aggregating individual measurements:

- *Worst-case analysis:* How much of the resource will the algorithm use in the worst case (across all problems of size n)?
- *Average-case analysis:* How much will it use on average?

But providing a formal average-case analysis that is theoretically sound is pretty difficult (*where do we get the input distribution from?*).

The complexity of a *problem* is defined as the complexity of the best *algorithm* that solves 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 returning, for any problem of size n, its *complexity* $f(n)$. This may be rather complicated (e.g., $f : n \mapsto 3n^2 - 2n + 101$).

Think of g as a function that is a "*good approximation*" of f and that is more convenient to talk about (e.g., $g : n \mapsto n^2$).

The *Big-O Notation* is a way of making this 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 $f(n) \leqslant c \cdot g(n)$ for all $n \geqslant n_0$.

Thus, from some n_0 onwards, f grows at most as fast as the reference function g, modulo some constant factor c (we don't care about).

Example: $3n^2 - 2n + 101$ is in $O(n^2)$.

Tractability and Intractability

A common distinction:

- *Tractable* problems can be solved in polynomial time, i.e., using algorithms that run in time $O(n^c)$ for some constant $c \in \mathbb{N}$.
- *Intractable* problems require algorithms that need exponential time, such as $O(2^n)$ or $O(1.1^n)$.

Remark: In practice, a polynomial algorithm running in $O(n^{1000})$ may do worse than an exponential algorithm running in $O(2^{\frac{n}{100}})$. However, experience suggests that such weird functions do not really come up in practice. And for very large n, the polynomial algorithm will do better.

Remark: There are examples for empirically successful algorithms for problems that are known not to be solvable in polynomial time. But, while such algorithms may be fast on most inputs likely to occur in practice, they cannot be efficient in all cases.

The Travelling Salesman Problem

The decision problem variant of a famous problem:

> TRAVELLING SALESMAN PROBLEM (TSP)
> **Input:** n cities; distance between each pair; $K \in \mathbb{N}$
> **Question:** Is there a route $\leqslant K$ 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.

Insight: 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.

- $\mathbf{TIME}(f(n))$ is the set of all decision problems that can be solved by an algorithm that runs in time $O(f(n))$.

  Example: $\textsc{Reachability} \in \mathbf{TIME}(n^2)$.

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

  Example: $\mathrm{TSP} \in \mathbf{SPACE}(n)$, because our brute-force algorithm only needs to keep in memory the route currently being tested and the route that has been the best so far.

These are sometimes called *deterministic* complexity classes (because the algorithms used to define them 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?"*. It might already be in that form originally (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. $M \models \varphi$?"*).

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

  Example: $\mathrm{TSP} \in \mathbf{NTIME}(n)$, as we can check in linear time whether a proposed route is short enough (just add the distances).

- $\mathbf{NSPACE}(f(n))$ is defined analogously.

So why are they called *nondeterministic* complexity classes?

Ways of Interpreting Nondeterminism

Original perspective (clarifying the name "nondeterministic"):

- 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 choice-point in the algorithm.

Equivalence to the verification-oriented perspective explained earlier:

- Asking all the "little oracles" along a computation path is equivalent to asking a "big initial oracle" once to guess a solution that can then be checked for correctness.

P and NP

The two most important complexity classes:

$$\mathbf{P} = \bigcup_{k \in \mathbb{N}} \mathbf{TIME}(n^k)$$

$$\mathbf{NP} = \bigcup_{k \in \mathbb{N}} \mathbf{NTIME}(n^k)$$

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

- $\mathbf{P}$ is the class of problems that can be *solved* in polynomial time by a deterministic algorithm.
- $\mathbf{NP}$ is the class of problems for which a proposed solution can be *verified* in polynomial time.

Other Common Complexity Classes

$$\mathbf{PSPACE} = \bigcup_{k \in \mathbb{N}} \mathbf{SPACE}(n^k)$$

$$\mathbf{NPSPACE} = \bigcup_{k \in \mathbb{N}} \mathbf{NSPACE}(n^k)$$

$$\mathbf{EXP} = \bigcup_{k \in \mathbb{N}} \mathbf{TIME}(2^{(n^k)})$$

Relationships between Complexity Classes

The following inclusions are known:

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

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

One of the main research topics *within* complexity theory is to uncover such relationships. As this is not our topic today, just very briefly:

- $\mathbf{PSPACE} = \mathbf{NPSPACE}$ is known as Savitch's Theorem.
- $\mathbf{P} \subset \mathbf{EXP}$ is a corollary of the Time Hierarchy Theorem.
- All other inclusions are easy.

Complements

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

  Example: SAT is the problem of checking whether a given formula of propositional logic is satisfiable. Its complement UNSAT is the problem of checking whether a given formula is *not* satisfiable (equivalent to checking whether its negation is a tautology).

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

  Example: $\mathbf{coNP}$ is the class of problems for which a negative answer can be verified in polynomial time.

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

Polynomial-Time Reductions

Problem A *reduces* to problem B if we can translate any input for A into an input for 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-input into a B-input in poly-time
- and show that the answer given the A-input should be YES iff a B-solver will answer YES to the translated input.

Hardness and Completeness

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

- A problem P is *$\mathcal{C}$-hard* if every problem $P' \in \mathcal{C}$ is polynomial-time reducible to P. Thus, 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* if P is $\mathcal{C}$-hard and furthermore $P \in \mathcal{C}$. Thus, these are the hardest problems in $\mathcal{C}$, and only those.

The Cook-Levin Theorem

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

SATISFIABILITY (SAT)
Input: Propositional formula φ
Question: Is φ satisfiable?

The *size* of an instance of SAT is the length of φ. SAT can be solved in exponential time (by trying all possible truth assignments), but no (deterministic) polynomial algorithm is known. We state w/o proof:

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

Remark: A similar result was proved by L. Levin around the same time.

Corollary 2 *Deciding whether* φ *is a tautology is* **coNP**-*complete.*

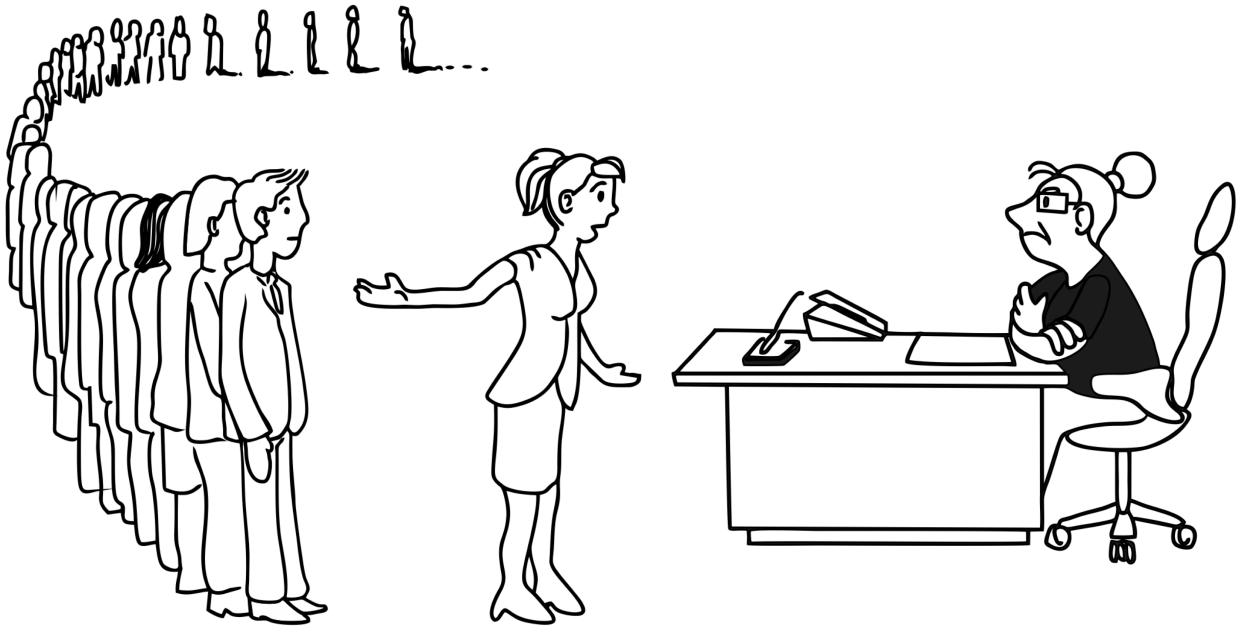
S. Cook. *The Complexity of Theorem-Proving Procedures*. Proc. STOC-1971.

Proving NP-Completeness

Next, we will prove several other problems to be $\mathbf{NP}$-complete as well.

Why is this useful? $\mathbf{NP}$-completeness doesn't mean that no poly-time algorithm exists (as we don't know for sure that $\mathbf{P} \neq \mathbf{NP}$).

But: All $\mathbf{NP}$-complete problems are equally hard: You find a poly-time algorithm for one of them, you automatically get one for *all* of them.



"I can't find an efficient algorithm, but neither can all these famous people."

Picture credits: Stefan Szeider

Variants of Satisfiability

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

k-SATISFIABILITY (kSAT)
Input: Conjunction φ of k-clauses
Question: Is φ satisfiable?

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

A variant of the Cook-Levin Theorem (again without proof) shows that it does in fact not get any easier, as long as $k \geqslant 3$:

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

Remark: To see that 2SAT *is polynomial*, write clauses as implications, create graph with vertices = literals and edges = implications, and use REACHABILITY to check you cannot reach $\neg x$ from x and x from $\neg x$.

Theorem 3 in hand, we can get lots of other results via reductions ...

Maximum Number of Satisfiable Clauses

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

MAXIMUM k-SATISFIABILITY (MAXkSAT)
Input: Set S of k-clauses and $K \in \mathbb{N}$
Question: Is there a satisfiable $S' \subseteq S$ such that $|S'| \geqslant K$?

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

Theorem 4 MAX2SAT *is* $\mathbf{NP}$*-complete.*

Proof: MAX2SAT is clearly *in* $\mathbf{NP}$*:* if you guess an $S' \subseteq S$ with $|S'| \geqslant K$ and a model M, then I can easily check whether $M \models S'$. ✓

Next we show $\mathbf{NP}$*-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 input for $3\textsc{Sat}$, construct an input for $\textsc{Max2Sat}$:

> 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 constructed input are (simultaneously) satisfiable. ✓

Independent Sets

Many conceptually simple problems that are **NP**-complete can be formulated as problems in graph theory. Here is an 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
Input: Undirected graph $G = (V, E)$ and $K \in \mathbb{N}$
Question: Does G have an independent set I with $|I| \geqslant K$?

Theorem 5 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. ✓

Computing with Oracles

Imagine you have access to an $\mathbf{NP}$-*oracle:* a machine that can solve NP-complete problems (such as SAT) in a single time step.

Some complexity classes that are important for COMSOC:

- $\Delta_2^p = \mathbf{P}^{\mathbf{NP}}$: problems that can be decided in polynomial time by a machine with access to an $\mathbf{NP}$-oracle
- $\Theta_2^p = \mathbf{P}_{||}^{\mathbf{NP}} = \mathbf{P}^{\mathbf{NP}}[\log]$: same, but all oracle queries need to be placed in parallel (equivalent: only log-many oracle queries)
- $\Sigma_2^p = \mathbf{NP}^{\mathbf{NP}}$: problems for which a positive instance can be verified in polynomial time with access to an $\mathbf{NP}$-oracle

  Example: SAT for quantified boolean formulas $\exists\vec{x}.\forall\vec{y}.\varphi$ is Σ_2^p-complete
- $\Pi_2^p = \mathbf{coNP}^{\mathbf{NP}}$: complement of Σ_2^P

  Example: SAT for quantified boolean formulas $\forall\vec{x}.\exists\vec{y}.\varphi$ is Π_2^p-complete

Σ_2^p and Π_2^p form the second level of the *polynomial hierarchy*.

Summary

We have covered the following topics:

- Definition of complexity classes: **P**, **NP**, **coNP**, **PSPACE**, ...
- Relationships between complexity classes
- Hardness and completeness w.r.t. 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

To quickly look look up a definition, try the *Complexity Zoo:*

`http://complexityzoo.uwaterloo.ca/`

Helpful *textbooks* include:

- S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.
- C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 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/`