Tutorial: Intro to Computational Complexity

Stathis Zachos

13th Panhellenic Logic Symposium

Overview

1. Complexity classes
2. Randomness
3. Interactivity, PCP
4. Counting
5. How to handle **NP**-complete problems
6. Approximation Algorithms
7. Search Complexity
8. Parameterized Complexity
9. Quantum Complexity

Complexity classes

- Computability theory: We are interested whether a problem is **computable or not**.
- Complexity theory: We consider only computable problems and we are interested for their solutions within given restricted resources, such as time for the computation, space (only intermediate working space is restricted and not space needed for the input and output), etc.
- These *restrictions* and other properties of the computation define complexity classes of computational problems.

Despite the constant and long-lasting effort of researchers, there are several **open questions** in the area of complexity theory. For example, there are problems in the class **NP** for which there is no polynomial-time algorithm, but have not been proven **NP**-complete so far.

Complexity classes (Hartmanis, Edmonds, Cook, Karp)

- 1965 Hartmanis
- 1965 – 1968 Edmonds: Graph problems
- 1971 Cook: **NP**-completeness ($\mathrm{CNF\text{-}SAT}$ with Cook reductions)
- 1972 Karp: Most of Edmonds' hard problems are **NP**-complete with Karp reductions

Complexity classes (What Karp didn't know in 1972)

- GRAPH ISOMORPHISM (the most well-known open problem): Given two graphs, are they isomorphic? (compare with SUBGRAPH ISOMORPHISM which is known to be **NP**-complete)
- LINEAR PROGRAMMING (it was open for many years): given a system of linear equations and inequalities and a linear objective function (either minimization or maximization), find a feasible optimum solution.
  - **Simplex method** (Dantzig): In worst case, exponential time is needed.
  - **Ellipsoid method** (Khachiyan): The first polynomial-time algorithm for linear programming. It was not of great practical interest.
  - **Karmarkar algorithm**: A polynomial-time algorithm that had pactical implications better than the Simplex method.
- PRIMALITY: Given an integer *n*, is *n* prime or not?
  Recently (2002 by Agrawal, Kayal, Saxena — AKS) it was proved that this problem, which was open for many years, is in the class **P**.

Basic definitions I I

Definition

The class TIME($t(n)$) (or DTIME($t(n)$)) contains problems that can be solved by a deterministic Turing machine in $t(n)$-time.

Definition

The class NTIME($t(n)$) contains problems that can be solved by a non-deterministic Turing machine in $t(n)$-time.

Definition

The class SPACE($s(n)$) (or DSPACE($s(n)$)) contains problems that can be solved by a deterministic Turing machine which uses additional $s(n)$ space.

Definition

The class NSPACE($s(n)$) contains problems that can be solved by a non-deterministic Turing machine which uses additional $s(n)$ space.

Basic definitions II

Based on the previous definitions, we define:

- $\mathbf{P} = \mathrm{PTIME} = \bigcup_{i\geq 1} \mathrm{DTIME}(n^i)$
- $\mathbf{NP} = \mathrm{NPTIME} = \bigcup_{i\geq 1} \mathrm{NTIME}(n^i)$
- $\mathbf{PSPACE} = \bigcup_{i\geq 1} \mathrm{DSPACE}(n^i)$
- $\mathbf{NPSPACE} = \bigcup_{i\geq 1} \mathrm{NSPACE}(n^i)$
- $\mathbf{L} = \mathrm{DSPACE}(\log n)$
- $\mathbf{NL} = \mathrm{NSPACE}(\log n)$
- $\mathbf{EXP} = \bigcup_{i\geq 1} \mathrm{DTIME}(2^{n^i})$
- $\mathbf{EXPSPACE} = \bigcup_{i\geq 1} \mathrm{DSPACE}(2^{n^i})$

Remark

A function f is called constructible if there exists a TM such that $\forall$ input x with $|x| = n$, it accepts the input in $O(n + f(n))$ time (time-constructible) or $O(f(n))$ working space (space-constructible).

Basic definitions III

If f is constructible, then:

- $\text{DSPACE}(f(n)) \subseteq \text{NSPACE}(f(n))$
- $\text{DTIME}(f(n)) \subseteq \text{NTIME}(f(n))$

since a deterministic Turing machine can be considered as a non-deterministic one with only one choice at each step.

- $\text{DTIME}(f(n)) \subseteq \text{DSPACE}(f(n))$
- $\text{NTIME}(f(n)) \subseteq \text{DSPACE}(f(n))$

since in $f(n)$ time, at most $f(n)$ space (positions on the machine tape) can be examined.

If $f(n) > \log n$ then:

- $\text{DSPACE}(f(n)) \subseteq \text{DTIME}(c^{f(n)})$
- $\text{NTIME}(f(n)) \subseteq \text{DTIME}(c^{f(n)})$
- $\text{NSPACE}(f(n)) \subseteq \text{DTIME}(k^{f(n)})$

Basic definitions IV

The following theorem is due to Savitch (1970):

Theorem

If $f(n) \geq \log n$ then $\mathrm{NSPACE}(f(n)) \subseteq \mathrm{DSPACE}(f^2(n))$.

- By Savitch's theorem we have that: $\mathbf{PSPACE} = \mathbf{NPSPACE}$.
- By the above relations we have the following hierarchy:

$$\mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} = \mathbf{NPSPACE}$$

- We know that $\mathbf{L} \neq \mathbf{PSPACE}$ and $\mathbf{NL} \neq \mathbf{PSPACE}$ (this is a corollary of the hierarchy theorem for space classes that we will mention below).
- The following remain open:

$$\mathbf{L} \supseteq \mathbf{NL} \supseteq \mathbf{P} \supseteq \mathbf{NP} \supseteq \mathbf{PSPACE}$$

Basic definitions V

The world so far is depicted in the following figure.

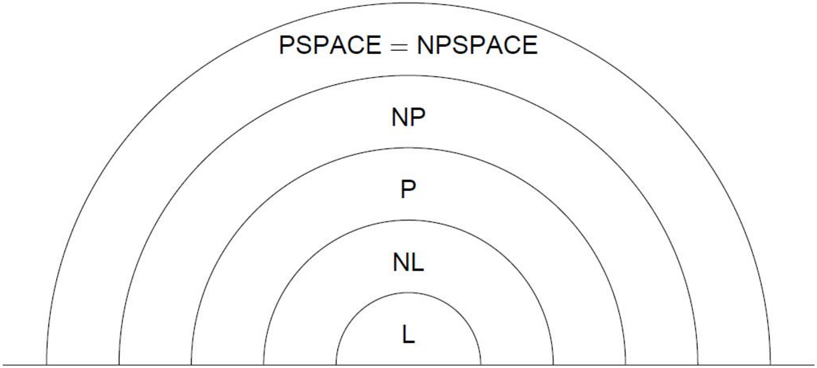


Figure: Complexity classes

Basic definitions VI

Notice that the aforementioned classes are classes of **decision** problems. We can also define complexity classes for Turing machines that compute **functions**. A typical example is the following.

Definition

FP = the set of functions that can be computed by deterministic Turing machines in *polynomial time*.

The class **FP** will be useful for the definition of reductions, since we need to include functions that can be computed "easily".

Another useful complexity class of functions is the next one.

Definition

FL = the set of functions that can be computed by deterministic Turing machines in *logarithmic space*.

Hierarchy theorems I

The following theorems hold for the model of the deterministic Turing machine with three tapes (we are always considering constructible functions t_1, t_2, s_1, s_2):

Theorem (Fürer, 1982)

Let $t_2(n) > n$. Then there is a language that is accepted in t_2 time, but not in t_1 time for every $t_1 = o(t_2(n))$.

Theorem (Hartmanis, Lewis, Stearns, 1965)

Let $s_2(n) > \log n$. Then there is a language that is accepted in s_2 space, but not in s_1 space for every $s_1 = o(s_2(n))$.

Technical proofs are ommited.

Analogous theorems hold for non-deterministic Turing machines as well. In fact the proofs are easier.

Hierarchy theorems II

We insist on constructible functions, since if we allow any function in the place of $t(n)$, $s(n)$, then we have pathological phenomena, such as the following.

Blum Complexity 1967

Theorem (Gap theorem)

There is a recursive function $t(n)$, such that $TIME(t(n)) = TIME(2^{t(n)})$.

Complementary complexity classes I

definition

Let L be a language. The complement of L is denoted and defined as

$$\overline{L} = \{x \mid x \notin L\}.$$

For a complexity class of languages $\mathcal{C}$, we define (using the complement of a language L):

$$\mathrm{co}\mathcal{C} = \{\overline{L} \mid L \in \mathcal{C}\}.$$

Example: The class **coNP** consists of the languages that are complements of languages in **NP**. A problem in **coNP** is $\overline{\mathrm{SAT}}$ or the closely related to it, TAUTOLOGY problem, i.e. given a propositional formula, determine whether it is a tautology.

Complementary complexity classes II

- It is interesting to see which complexity classes are **closed under complement**, i.e. for which classes $\mathcal{C}$, it holds that $\mathcal{C} = co\mathcal{C}$.

- In general, deterministic complexity classes (either time, or space) are closed under complement, i.e. DTIME($t(n)$) and DSPACE($s(n)$) are closed under complement.

- If we consider non-deterministic classes, the question is open in the case of time complexity. For example, we do not know whether **coNP** $\neq$ **NP**. In fact, the last inequality is related to the **P** versus **NP** problem, since obviously, if **coNP** $\neq$ **NP**, then **P** $\neq$ **NP**.

Complementary complexity classes III

Theorem (Immerman-Szelepcsényi)

The class $NSPACE(s(n))$ is closed under complement.

For $s(n) = n$ we have the class of problems that are solved by a Turing machine that uses linear space, also known as LBA (linearly bounded automaton), and so the above theorem answered a long-standing question, i.e. whether the class of LBA (or equivalently the class of context sensitive languages, as Kuroda showed in 1964) is closed under complement.

Reductions I

A reduction of polynomial time has to connect two problems via an "easy" computation. We are considering easy functions (and problems) that are computed in polynomial time.

We would like to have the following properties.

- If functions f and g are "easy", then their composition $f \circ g$ is "easy".
- If f is computable in $O(n^2)$ time, then it is considered to be easy.

Thus, we are consider easy problems (and functions), the problems (and functions) that can be computed in polynomial time (even in $O(n^{1000})$).

For these reasons, we define the Karp reduction as follows.

Definition (Karp reduction)

$$A \leq_m^P B : \quad \exists f \in \mathbf{FP}, \forall x (x \in A \iff f(x) \in B)$$

There are other useful reductions, such as *log-space* reductions, that use logarithmic space and are used for reductions between problems in "smaller" complexity classes, like **P**.

Reductions II

Definition (Log-space reduction)

$$A \leq_m^L B: \quad \exists f \in \mathbf{FL}, \forall x (x \in A \iff f(x) \in B)$$

It holds that $A \leq_m^L B \implies A \leq_m^P B$, but the converse is not true.

Another property that a reduction should have for various classes of languages, is the following.

Definition

We say that a class of languages C is **closed** under a reduction $\leq$ if

$$A \leq B \wedge B \in C \implies A \in C.$$

Some complexity classes closed under Karp reduction ($\leq_m^P$) are the following: **P**, **PSPACE**, **EXP**, **EXPSPACE** (see above for the definitions of these classes).

Reductions III

Definition (Hardness)

We say that A is C-hard with respect to $\leq$, if

$$\forall B \in C : B \leq A.$$

The notion of hardness gives a lower bound for the complexity of a problem, given that the problem A is at least as hard as any problem in the class C.

Definition (Completeness)

We say that A is C-complete with respect to $\leq$, if

$$A \text{ is } C\text{-hard with respect to } \leq \quad \wedge \quad A \in C.$$

Reductions IV

Below we give complete problems for some of the most important complexity classes.

- **NL**: the problem Reachability (log-space reductions).
- **P**: Circuit-Value and Linear Programming (again under log-space reductions).
- **NP**: 3SAT.
- **PSPACE**: QBF (Quantified Boolean Formula satisfiability problem).
- **EXP**: $n \times n$ Go.
- **EXPSPACE**: $\textsc{RegExp}(\cup, \cdot, *, {}^2)$, which is the problem of checking equivalence of regular expressions, that use operators $\cup$ (union), $\cdot$ (concatenation), $*$ (Kleene star) and ^{2}, where $\alpha^2 = \alpha \cdot \alpha$.

Parameters for defining complexity classes I

- Concrete Complexity: We consider a *specific* computational model, a *specific* problem and a *specific* algorithm for this problem in this model. In this way, we determine the exact complexity of the algorithm (constants are not important).
- Abstract or structural complexity: We consider complexity classes with various *computational parameters* and we compare these classes to each other (with respect to inclusion, separation etc). It is useful to find reductions and complete problems for these classes under these reductions.

Parameters for defining complexity classes II

We mention some parameters used to define complexity classes.

- **computational model**: Turing Machine (TM), Random Access Machine (RAM), Finite Automaton, Linearly Bounded Automaton (LBA), Parallel RAM (PRAM), monotone circuits.
- **method of operation/acceptance**: deterministic, non-deterministic, probabilistic, alternating, parallel.
- **kind of model/operation**: decider, acceptor, generator, transducer.
- **resources**: number of steps, number of comparisons, number of multiplications, time, storage space, number of processors, number of alternations in the computation tree, size of circuit, depth of circuit.
- **other tools**: randomness, oracles, interactivity, promise, operators.
- **bounds with respect to the size of the input**: for example $O(n^3)$ or polynomial time/space $(t(n), s(n))$ tradeoff, Probabilistic Checkable Proofs: $\mathrm{PCP}(r(n), q(n))$ (using $r(n)$ random bits and $q(n)$ queries to the proof).

Models of computation tree for TM I

- To study the behavior of Turing machines, we are going to encode the computation of a Turing machine using a computation tree.
- The computation starts at the **root** of the tree.
- We assume that if we have a non-deterministic choice at some point of the computation, then we have a **branching** in the tree.
- At the **leaves** of the computation tree of a TM we have the answers of the machine. Every path from the root of the tree to some leaf encodes a possible computation.
- Without loss of generality, we consider the computation tree to be binary, complete and full. All its leaves are at the same level.

Models of computation tree for TM II

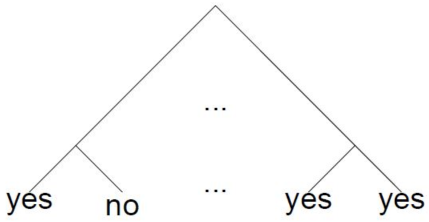


Figure: Computation tree model

An interesting case is when the length of the computation path from the root to the leaf is **polynomial** with respect to the input length (every path corresponds to an "easy", i.e. polynomial, computation).

By considering the above model, we are going to define some of the already known complexity classes and some new ones. More precisely, we are going to use quantifiers $(\exists, \forall)$ over paths. Since we always consider the restriction on the path length, we are going to write e.g. $\exists y$ instead of $\exists y : |y| \leq p(|x|)$, where y: a variable representing paths, x: a variable representing the input, p: a polynomial.

Models of computation tree for TM III

- For example, the class **NP** can also be defined as follows.

$$L \in \mathbf{NP} \iff \exists R \in \mathbf{P} \colon \begin{cases} x \in L \implies \exists y R(x,y) \\ x \notin L \implies \forall y \neg R(x,y) \end{cases}$$

  In other words, if $x \in L$ there exists at least one accepting computation, whereas if $x \notin L$ no computation is accepting.

- Similarly, the class **coNP** is also defined as follows.

$$L \in \mathbf{coNP} \iff \exists R \in \mathbf{P} \colon \begin{cases} x \in L \implies \forall y R(x,y) \\ x \notin L \implies \exists y \neg R(x,y) \end{cases}$$

Models of computation tree for TM IV

- The class **P** can also be defined as follows.

$$L \in \mathbf{P} \iff \exists R \in \mathbf{P}\colon \begin{cases} x \in L \implies \forall y R(x,y) \\ x \notin L \implies \forall y \neg R(x,y) \end{cases}$$

Note that the quantifiers that correspond to $x \in L$ and $x \notin L$ completely determine a complexity class. We introduce the following notation.

$$\mathbf{NP} = (\exists, \forall), \quad \mathbf{coNP} = (\forall, \exists), \quad \mathbf{P} = (\forall, \forall).$$

Overview

1. Complexity classes
2. Randomness
3. Interactivity, PCP
4. Counting
5. How to handle **NP**-complete problems
6. Approximation Algorithms
7. Search Complexity
8. Parameterized Complexity
9. Quantum Complexity

Randomness I

- Using the model of the **computation tree**, we are going to define complexity classes which are based on probabilities, determined by random choices.
- This approach is very useful from a practical point of view, since in many applications, it is sufficient to have an algorithm that makes some random choices and gives the correct result in most cases.
- A probabilistic algorithm is usually simpler and more efficient in practice than a deterministic one which solves the same problem. For example, simple probabilistic algorithms for checking whether a number is prime exist since the 1970s and are used in practice instead of more complex deterministic ones, like AKS.
- In the context of the computation tree model, we are going to assume that at every node of the tree, the choice is made randomly with probability $1/2$ for every child of the node. To show that the *overwhelming* majority of the computations give the correct result, we introduce a new quantifier, namely $\exists^+$.

Randomness II

- Using the quantifier $\exists^+$, we define the class **BPP** (Bounded error Probabilistic Polynomial):

Definition ($\mathbf{BPP} = (\exists^+, \exists^+)$)

$$L \in \mathbf{BPP} \iff \exists R \in \mathbf{P} : \begin{cases} x \in L \implies \exists^+ y R(x,y) \\ x \notin L \implies \exists^+ y \neg R(x,y) \end{cases}$$

Randomness III

- In other words, in a computation tree for the class **BPP**, the overwhelming majority of the leaves give the correct result. In the above definition, the exact definition of the overwhelming majority does not matter, but it has to be *bounded* above $1/2$. The majority percentage can be, for example, greater than $1/2 + \varepsilon$, $1/2 + 1/p(|x|)$, $2/3$, 99%, $1 - 2^{-p(|x|)}$ ($2^{-p(|x|)}$ is called a negligible amount). This flexibility in the choice of the bound is based on the fact that by running the algorithm polynomially many times, we can amplify the success probability as much as we want. **BPP** algorithms are called **Monte Carlo** or **two-sided error**, since, regardless of the result (yes or no), there exists a non-zero failure probability. Obviously, **BPP** is closed under complement.

Randomness IV

- Now let us consider algorithms that have one sided error. In this way we obtain the class **RP** (Randomized Polynomial):

Definition ($\mathbf{RP} = (\exists^+, \forall)$)

$$L \in \mathbf{RP} \iff \exists R \in \mathbf{P}: \begin{cases} x \in L \implies \exists^+ y R(x,y) \\ x \notin L \implies \forall y \neg R(x,y) \end{cases}$$

- In the case of **RP** the majority percentage is sufficient to be greater than $\frac{1}{p(|x|)}$ for some polynomial p.

Randomness V

In the case of this class, if the corresponding **RP** algorithm answers "yes" (i.e. the predicate R is true), we are certain that $x \in L$. On the contrary, if the algorithm answers "no", it might be wrong.

Obviously, it holds that **RP** $\subseteq$ **BPP**, **coRP** $\subseteq$ **BPP**, but we do not know whether **RP** $=$ **coRP**.

- Another useful class is the class that is defined as the intersection of **RP** and **coRP**, namely **ZPP** $=$ **RP** $\cap$ **coRP**. This class derives its name from Zero error Probabilistic Polynomial, since it is not hard to show that a problem is in **ZPP** if there exists a probabilistic algorithm that runs in expected polynomial time and always gives the correct answer. Indeed, if a problem is in **ZPP**, we have both an **RP** and a **coRP** algorithm for solving it, so it suffices to run these two algorithms repeatedly in parallel until one of them returns the value that is definitely correct. We may need to run the algorithms for ever, but with high probability, we will obtain a "decisive" answer after some finite number of runs (in fact, a polynomial number of times). Alternatively, we can say that a **ZPP** algorithm has three outputs: "yes", "no" (for the "decisive" answers) and "I don't know" (for the "indecisive" ones). **ZPP** algorithms are called **Las Vegas**.

Randomness VI

- Since probabilistic algorithms are widely used in practice, feasible problems are usually considered to be not only problems in **P**, but also problems in the probabilistic classes **BPP**, **RP** and **ZPP**.

  We do not know whether the classes defined above (**BPP**, **RP**, **ZPP**) have complete problems.

- If the error percentage is not bounded away from $1/2$, then we are certain that in the computation tree model, more than half of the computation paths give the correct answer. To express this fact we use the quantifier $\exists_{1/2}$. For unbounded two-sided error, we have the class **PP** (Probabilistic Polynomial):

Definition ($\mathbf{PP} = (\exists_{1/2}, \exists_{1/2})$)

$$L \in \mathbf{PP} \iff \exists R \in \mathbf{P}\colon \begin{cases} x \in L \implies \exists_{1/2} y R(x,y) \\ x \notin L \implies \exists_{1/2} y \neg R(x,y) \end{cases}$$

- Note that the threshold $1/2$ is not important. Any other threshold would define the same class.

Randomness VII

- Due to the lack of a bound (away from $1/2$) for the error probability, we cannot use the technique of polynomial repetition to amplify the correctness of a **PP** algorithm. Another indication that **PP** is a hard class is the following result.

Proposition

NP $\subseteq$ **PP**.

- Note that we have not considered the number of random bits used by a probabilistic algorithm, as a resource of the algorithm. In practice, every "random" bit we need, has a cost, since we obtain it by a *pseudorandom* bit generator.

- Finally, we mention the class **RL** (Randomized Logspace) which contains problems that have one-sided error algorithms which use logarithmic space and polynomial number of random bits (with respect to the input length).

Polynomial Hierarchy I

- We are going to define the polynomial hierarchy using the notion of the computation with an *oracle*.
- An algorithm uses an oracle for problem Π, if, during the computation, it can ask the oracle for an instance x of Π, whether $x \in \Pi$, and the oracle immediately answers either "yes" or "no". Regardless the hardness of Π, the algorithm does not need additional resources.

Polynomial Hierarchy II

Definition (Classes with oracles)

- $\mathcal{C}^{\Pi}$: the class of problems that can be solved by an algorithm that corresponds to class $\mathcal{C}$ and uses an oracle for the problem Π.
- $\mathcal{C}^{\mathcal{C}_o} = \bigcup_{\Pi \in \mathcal{C}_o} \mathcal{C}^{\Pi}$

For example, the class $\mathbf{P}^{\mathrm{SAT}}$ consists of the problems that can be solved by a deterministic polynomial-time algorithm that uses an oracle for SAT. Another description of this class is $\mathbf{P}^{\mathbf{NP}}$, since SAT is **NP**-complete.

Polynomial Hierarchy III

Definition

$(k \geq 0)$

- $\Sigma_0^p = \Pi_0^p = \Delta_0^p = \mathbf{P}$
- $\Sigma_{k+1}^p = \mathbf{NP}^{\Sigma_k^p}$, $\Pi_{k+1}^p = \mathrm{co}\Sigma_{k+1}^p$, $\Delta_{k+1}^p = \mathbf{P}^{\Sigma_k^p}$, $\Delta\Sigma_k^p = \Sigma_k^p \cap \Pi_k^p$
- Polynomial Hierarchy: $\mathbf{PH} = \bigcup_{k \in \mathbb{N}} \Sigma_k^p$

The following are true:

- $\Sigma_1^p = \mathbf{NP}$, $\Pi_1^p = \mathbf{coNP}$ and for all $k \geq 0$: $\Sigma_k^p \subseteq \Sigma_{k+1}^p$ and $\Pi_k^p \subseteq \Sigma_{k+1}^p$.
- Although these inclusions have not been proven to be strict (as in the case of the arithmetical hierarchy), we believe that the hierarchy is *strict*.
- If **PH** is not strict, then there is some k such that $\mathbf{PH} = \Sigma_k^p$, so we say that *the polynomial hierarchy collapses at the k-th level*.

Polynomial Hierarchy IV

$$\begin{tikzcd}
 & \mathit{PSPACE} & \\
\vdots & & \vdots \\
 & \Delta_3^p = \mathbf{P}^{\mathrm{NP}^{\mathrm{NP}}} \arrow[ul] \arrow[ur] & \\
\Pi_2^p = co\mathbf{NP}^{\mathrm{NP}} \arrow[ur] & & \Sigma_2^p = \mathrm{NP}^{\mathrm{NP}} \arrow[ul] \\
 & \Delta_2^p = \mathbf{P}^{\mathrm{NP}} \arrow[ul] \arrow[ur] & \\
\Pi_1^p = co\mathbf{NP} \arrow[ur] & & \Sigma_1^p = \mathrm{NP} \arrow[ul] \\
 & \Delta_0^p = \Sigma_0^p = \arrow[ul] \arrow[ur] & \\
 & = \Pi_0^p = \Delta_1^p = \mathbf{P} &
\end{tikzcd}$$

Figure: Polynomial hierarchy

Polynomial Hierarchy – Quantifier Alternation I

An alternative definition of the polynomial hierarchy can be given by using alternation of the quantifiers ($\exists$ and $\forall$). Note that the quantifiers are over objects the size of which is bounded by a polynomial p in the input size.

Proposition

$L \in \Sigma_k^p$ if there is a predicate R computable in polynomial time such that:

$$x \in L \iff \exists y_1 \forall y_2 \dots Q y_k \; R(x, y_1, y_2, \dots, y_k),$$

where $Q = \begin{cases} \exists, & k \text{ odd} \\ \forall, & k \text{ even} \end{cases}$
and the quantifiers are over objects the size of which is bounded by $p(|x|)$ for some polynomial p.

Polynomial Hierarchy – Quantifier Alternation II

Similarly, we define the class Π_k^p. In this case, the sequence of the quantifiers starts with $\forall$:

Proposition

$L \in \Pi_k^p$if there is a predicate R computable in polynomial time such that:

$$x \in L \iff \forall y_1 \exists y_2 \dots Q y_k \; R(x, y_1, y_2, \dots, y_k),$$

όπου $Q = \begin{cases} \forall, & k \text{ odd} \\ \exists, & k \text{ even} \end{cases}$
and the quantifiers are over objects the size of which is bounded by $p(|x|)$ for some polynomial p.

Polynomial Hierarchy – Alternating TM I

- The alternation of quantifiers in the polynomial hierarchy gives the motivation to define the *alternating* Turing machine.
- If we think of the tree representation of the computation of an **NP** Turing machine, the machine returns "yes", if there is at least one leaf that says "yes". We can assume that each node of the tree computes the disjunction ($\vee$) of its children's results and forwards it to its parent (a leaf just forwards its result to its parent) until the result reaches the root.
- A **coNP** Turing machine accepts if all leaves say "yes", so we can assume that each node forwards to its parent the conjunction ($\wedge$) of its children's results, until the correct result reaches the root.
- We say that the nodes in the computation tree of an **NP** machine are of type $\vee$, or $\exists$, or of existential type.
- We say that the nodes in the computation tree of a **coNP** machine are of type $\wedge$, or $\forall$, or of universal type.

Polynomial Hierarchy – Alternating TM II

- An alternating Turing machine is a Turing machine, the computation tree of which has internal nodes of type either $\vee$ or $\wedge$.
- The **number of type alternations** is important. The maximum **number of alternations** (on a path), which may be bounded, determines the computation power of such a machine.

For example, the computation tree of the following figure has a number of alternations equal to 2.

Polynomial Hierarchy – Alternating TM III

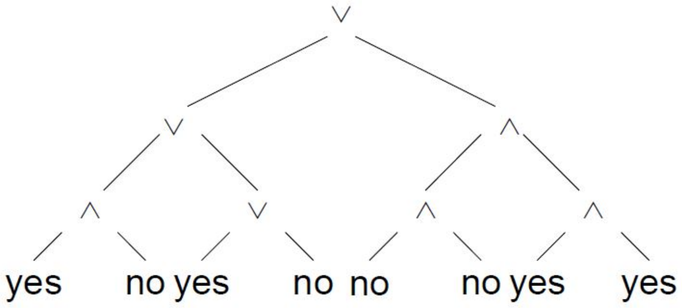


Figure: Computation tree with alternations

Polynomial Hierarchy – Alternating TM IV

We can show that the polynomial hierarchy is exactly the class of languages that are accepted by alternating Turing machines with a bounded number of alternations.

More precisely:

- $L \in \Sigma_k^p$ if L is accepted by a Turing machine that has at most k type alternations and starts with type $\vee$.
- $L \in \Pi_k^p$ if L is accepted by a Turing machine that has at most k type alternations and starts with type $\wedge$.

Parallelizable problems I

- To study parallel computations, we introduce a new computational model, the circuit.
- A circuit is a directed acyclic graph, which has a set of **input** nodes and an **output** node. We assume that the circuit takes as inputs truth values (the values 0 and 1) and each internal node corresponds to a logical function (or gate) with as many inputs as its incoming edges.
- If a circuit C has n 1-bit inputs: x_1, x_2, . . ., x_n, then for every $x \in \{0, 1\}^n$, it computes a unique value at the output, namely $C(x)$. If $C(x) = 1$, we say that the circuit C accepts the n-bit input: x.

Parallelizable problems II

Compared to the model of the Turing machine, a circuit can take only inputs of length exactly n, whereas a Turing machine (or an algorithm in general) takes inputs of arbitrary length.

Therefore, we consider a *circuit family* $\{C_1, C_2, \dots\}$, where every C_n has n input nodes. The language accepted by a circuit family C is the following

$$L(C) = \{x \mid C_{|x|}(x) = 1\}.$$

The problem here is that circuit families (unlike Turing machines) are not countable.

Parallelizable problems III

- To overcome the above difficulty, we restrict ourselves to uniform circuit families.
- For such families, there is an efficient algorithm such that, given n, it constructs the representation of circuit C_n of the family.
- One option is to consider **P**-uniform families, that use a polynomial-time algorithm for constructing the corresponding circuits.

However, circuits are usually used to define classes below the class **P**, so we are going to consider a more restricted notion of uniformity.

Definition

A circuit family C is DLOGTIME-uniform if there is a Turing machine (which has random access to the input tape) that answers the following questions in $O(\log n)$ time:

- Are nodes u and v connected in C_n?
- What type of gate corresponds to node u?

Parallelizable problems IV

- The *size* of a circuit is the number of nodes in the corresponding graph.
- The size measures the cost of constructing the circuit and it is usually considered to be at most polynomial with respect to the input length.
- However, the size of the circuit is not a good measure of the computation time of a circuit, since many logical gates can be computed in parallel.
- The gates that have to wait for intermediate results are those that are on some path connecting the input to the output.
- So, a more important measure of the computation time is the *depth* of a circuit, which is the length of the longest path from the input to the output node.

Parallelizable problems V

Also, the type of the logical gates used in a circuit is important. More precisely, we have the following types of logical gates.

1. Logical gates with bounded fan-in (bounded number of inputs), as well as unary gates $\neg$. It suffices to have binary gates $\wedge$ and $\vee$ (together with unary gates $\neg$).
2. Logical gates $\wedge$ and $\vee$ with unbounded fan-in, as well as unary gates $\neg$.
3. Threshold gates with unbounded number of inputs, as well as unary gates $\neg$. It suffices to have majority gates instead of general threshold gates. A majority gate returns 1 iff at least $r/2$ out of r inputs are 1.

Parallelizable problems VI

Now we can define the following classes.

Definition

$(k \geq 0)$:

1. $\mathbf{NC^k}$: the class of languages that are accepted by DLOGTIME-uniform circuit families of polynomial size and οικογένειες κυκλωμάτων $O(\log^k n)$ depth, using gates of bounded fan-in.
2. $\mathbf{AC^k}$: the class of languages that are accepted by DLOGTIME-uniform circuit families of polynomial size and οικογένειες κυκλωμάτων $O(\log^k n)$ depth, using gates of unbounded fan-in.
3. $\mathbf{TC^k}$: the class of languages that are accepted by DLOGTIME-uniform circuit families of polynomial size and οικογένειες κυκλωμάτων $O(\log^k n)$ depth, using threshold gates.
4. $\mathbf{SC^k}$: the class of languages that are accepted by DTM in polynomial time and $O(\log^k n)$ space.

Parallelizable problems VII

- Moreover, we define $\mathbf{NC} = \bigcup_{k \in \mathbb{N}} \mathbf{NC^k}$. The class **NC** is also called Nick's Class for Nicholas Pippenger, who was among the first that studied this kind of circuits. In fact, not only circuits, but also many other models of parallel computations (e.g. PRAM), can be used to define the class **NC**, which indicates that this is a robust class and closely related to parallelizable problems.
- "A" in $\mathbf{AC^k}$ comes from alternation, since it can be proved that class $\mathbf{AC^k}$, for $k \geq 1$, consists of exactly the languages that are accepted by an alternating Turing machine that uses $O(\log n)$ space and makes at most $O(\log^k n)$ alternations.
- "T" in $\mathbf{TC^k}$ comes from threshold.
- **SC**, Steve's Class for Steve Cook.

Parallelizable problems VIII

More precisely, the aforementioned classes are related to each other as follows.

Theorem

For every $k \geq 0$, $\mathbf{NC^k} \subseteq \mathbf{AC^k} \subseteq \mathbf{TC^k} \subseteq \mathbf{NC^{k+1}}$.

The next theorem also holds with respect to other known classes.

Theorem

$\mathbf{Regular} \subseteq \mathbf{NC^1} \subseteq \mathbf{L} = \mathbf{SC^1} \subseteq \mathbf{NL} \subseteq \mathbf{AC^1}$.
$\mathbf{Regular} \subset \mathbf{CF} \subset \mathbf{AC^1}$.

This means that the problem of checking whether a string belongs to a context free language is in the class $\mathbf{NC^2}$.

Overview

1 Complexity classes

2 Randomness

3 Interactivity, PCP

4 Counting

5 How to handle **NP**-complete problems

6 Approximation Algorithms

7 Search Complexity

8 Parameterized Complexity

9 Quantum Complexity

Interactivity I

Interactive Proofs (IP)

- Let us consider a prover (P) that tries to prove to someone else, who is called verifier (V), that a statement, like "$x \in L$", is true.
- The prover is omnipotent, in the sense that it is an algorithm with unlimited computational resources (time, space). On the contrary, the verifier is a probabilistic polynomial-time algorithm.
- The verifier and the prover take part in a communication protocol by exchanging messages. V either accepts or rejects the proof based on the messages it receives from P. The prover may not be honest and may want to convince the verifier that "$x \in L$" even for x for which "$x \notin L$" holds.
- The verifier can use, except for polynomial-time computations, the randomness it possess, against the omnipotent prover.

Interactivity II

Interactive Proofs (IP)

The class **IP** was defined by Goldwasser, Micali, Rackoff:

Definition

$L \in$ **IP**:

- $x \in L \implies$ there exists a prover P, such that the verifier V always accepts (i.e. we have probability of acceptance equal to 1).
- $x \notin L \implies$ for every prover P, the verifier V does not accept with high probability.

- Let us consider the **graph non-isomorphism problem**: "Given two graphs, are they not isomorphic?".
- This problem belongs to **coNP**.
- We are going to describe a protocol for the non-isomorphism graph problem, which shows that the problem belongs to **IP**.

Interactivity III

Interactive Proofs (IP)

1. At first, the verifier has the two graphs G_1 and G_2. He chooses at random one of them, let's say G_i, and computes a random graph, which is isomorphic to G_i, let's say graph H (this can be done by choosing a random permutation of the n vertices of G_i). V sends graph H to P, asking a j such that G_j is isomorphic to H.
2. The prover answers with a $j \in \{1, 2\}$.
3. The verifier accepts if $i = j$, otherwise it rejects.

- In the case of G_1, G_2 being not isomorphic, P, since it is omnipotent, finds the (unique) graph G_j that is isomorphic to H (sent by V), and gives the correct value j. So V accepts.
- If G_1, G_2 are isomorphic, P cannot conclude which graph G_j, H comes from. So, P cannot do anything better than sending a random $j \in \{1, 2\}$ to V. Hence, if the two graphs are isomorphic, V does not accept with probability $1/2$.

So the graph non-isomorphsm problem belongs to **IP**.

Interactivity IV

Interactive Proofs (IP)

In fact, every language in the polynomial hierarchy has such a protocol and belongs to **IP**.

The following, even stronger result has been proved.

Theorem (Shamir)

$\mathbf{IP} = \mathbf{PSPACE}$

Interactivity V

Interactive Proofs (IP)

- What happens when the verifier can interact with two or even more provers?
- If the provers communicate with each other, then we remain in the class **IP** (in practice, a prover, being omnipotent, can simulate any other prover).
- However, if the provers do not communicate with each other, then we obtain the stronger class **MIP** (Multi IP).
- It holds that **MIP** = **NEXP**.

Interactivity I
Arthur-Merlin Classes

- In the class **IP** the verifier keeps the random bits it uses “private” (hidden). Recall that in the prototcol for the graph non-isomorphism problem, this fact was crucial for the proof.
- It seems that if the verifier has to reveal its bits, we obtain a class smaller than IP.
- In this class of languages, the prover is called Merlin and the verifier Arthur (this description was given by Babai).
- In fact, we can assume that the messages sent by Arthur are even more restricted: he just sends the random bits to Merlin. Based on Merlin’s answers, Arthur decides whether he accepts or rejects.

Interactivity II

Arthur-Merlin Classes

We say that Arthur and Merlin play a game of k moves (every move corresponds to a message): if Arthur moves first, then the game is denoted by AM(k), whereas if Merlin moves first, it is denoted by MA(k).

For example, AM(1) = A, AM(2) = AM, AM(3) = AMA, MA(1) = M, MA(2) = MA, MA(3) = MAM.

Another difference with respect to the class **IP** is that we need to bound the probabilities away from 1/2 (again the exact value is not important).

Formally for the class **AM(k)**, we have the following.

Definition

$L \in \mathbf{AM(k)}$ if there is a game of k moves such that Arthur plays first and if:

- $x \in L \implies$ Arthur is convinced that $x \in L$ with probability greater than $2/3$.
- $x \notin L \implies$ Arthur is convinced that $x \in L$ with probability less than $1/3$.

Interactivity III

Arthur-Merlin Classes

By using generalized quantifiers, the classes can be written as follows (Zachos):

$$\mathbf{AM} = \mathbf{AM(2)} = (\exists^+\exists, \exists^+\forall), \quad \mathbf{MA} = \mathbf{MA(2)} = (\exists\exists^+, \forall\exists^+),$$

and for k even, if $\mathbf{AM(k)} = (\mathbf{Q}_1, \mathbf{Q}_2)$, where $\mathbf{Q}_1$, $\mathbf{Q}_2$ sequences of quantifiers:

$$\mathbf{AM(k{+}1)} = (\mathbf{Q}_1\exists^+, \mathbf{Q}_2\exists^+), \quad \mathbf{AM(k{+}2)} = (\mathbf{Q}_1\exists^+\exists, \mathbf{Q}_2\exists^+\forall).$$

The above description can be simplified as follows (Zachos):

$$\mathbf{AM} = \mathbf{AM(2)} = (\forall\exists, \exists^+\forall), \quad \mathbf{MA} = \mathbf{MA(2)} = (\exists\forall, \forall\exists^+),$$

and for k even, if $\mathbf{AM(k)} = (\mathbf{Q}_1, \mathbf{Q}_2)$, where $\mathbf{Q}_1$, $\mathbf{Q}_2$ sequences of quantifiers:

$$\mathbf{AM(k{+}1)} = (\mathbf{Q}_1\forall, \mathbf{Q}_2\exists^+), \quad \mathbf{AM(k{+}2)} = (\mathbf{Q}_1\forall\exists, \mathbf{Q}_2\exists^+\forall).$$

Interactivity IV

Arthur-Merlin Classes

Using properties of quantifiers, we obtain the following results:

Proposition

$\mathbf{MA} \subseteq \mathbf{AM}$.

Proposition

The hierarchy of Arthur-Merlin games collapses, i.e.

$$\mathbf{AM} = \mathbf{AM(k)} = \mathbf{MA(k+1)}, \quad \text{for all } k \geq 2.$$

Although, as already mentioned, the Arthur-Merlin class with polynomial number of interactive messages seems to be weaker than **IP** (because of the fact that the random bits are public), Goldwasser, Sipser proved that they are equivalent.

Interactivity I

Probabilistic Checable Proofs — PCP

In the interactive proofs, if we replace the prover with a simple proof, we have the class **PCP**. Let us assume that in **PCP**, the prover interacts with the verifier only at the beginning: he just writes a proof and send it to the verifier. Note that these proofs are checked probabilistically by V.

Formally:

Definition

$L \in \mathbf{PCP}$:

- $x \in L \implies$ there is a proof Π such that the verifier V always accepts (i.e. we have acceptance probability equal to 1).
- $x \notin L \implies$ for every proof Π, the verifier V does not accept with high probability.

This class seems much stronger than **IP**, since the verifier has to handle a static object (the proof) and not to communicate with a prover that can adjust to his questions.

It can be proved that $\mathbf{PCP} = \mathbf{MIP} (= \mathbf{NEXP})$.

Interactivity II

Probabilistic Checable Proofs — PCP

Therefore, we are going to consider restrictions of the class **PCP**.
We are going to consider two kinds of resources that are not limitless for the verifier:

- randomness (in the form of random bits)·
- bits of the proof that can be checked (queries to the proof).

Definition

The class **PCP(r(n), q(n))** consists of the languages $L \in$ **PCP** that the probabilistic polynomial-time verifier V uses $O(r(n))$ random bits and checks $O(q(n))$ bits in the proof.

For example, already known complexity classes can be defined as follows.

$$\mathbf{PCP} = \mathbf{PCP(poly(n),\ poly(n))},\ \mathbf{P} = \mathbf{PCP(0,0)},$$
$$\mathbf{NP} = \mathbf{PCP(0,\ poly(n))},\ \mathbf{coRP} = \mathbf{PCP(poly(n),\ 0)}.$$

Interactivity III

Probabilistic Checable Proofs — PCP

A very important result (Arora, Lund, Motwani, Sudan, Szegedy) is the following:

Theorem (PCP)

$\mathbf{NP} = \mathbf{PCP}(\log n, 1)$.

An application of the PCP theorem is in proofs of inapproximability results.

The basic tool in the proof of the PCP theorem is a technique (PCP encoding) that spreads a possible error that exists at some point of the proof over all parts of the proof. In this way, the verifier can find the error with high probability. This technique is based on error correcting codes.

Overview

1. Complexity classes
2. Randomness
3. Interactivity, PCP
4. Counting
5. How to handle **NP**-complete problems
6. Approximation Algorithms
7. Search Complexity
8. Parameterized Complexity
9. Quantum Complexity

Counting classes I

Classes of counting problems are defined based on the number of solutions that a problem has. These are classes of functions (like **FP**).

The following classes are interesting.

Definition

#P is the class of functions f such that there is a non-deterministic polynomial-time Turing machine (NPTM), the computation tree of which has exactly $f(x)$ accepting computation paths (on input x).

Definition

#L is the class of functions f such that there is a non-deterministic logarithmic-space Turing machine, the computation tree of which has exactly $f(x)$ accepting computation paths (on input x).

When we consider counting classes, useful reductions are the ones that preserve the number of solutions.

Counting classes II

A representative example of a **#P**-complete problem is $\#\textsc{Sat}$:
"Given a formula in conjunctive normal form, how many truth assignments are there that satisfy the formula?".

Obviously, $\varphi \in \mathrm{SAT}$ iff $\#\mathrm{SAT}(\varphi) \neq 0$.

Valiant showed that there are decision problems in **P** (e.g. the problem of deciding whether there exists a perfect matching in a graph) for which the corresponding counting problem (e.g. $\#\textsc{Perfect Matchings}$) is **#P**-complete.

Some results for these classes are the following.

$$\mathbf{FP} \subseteq \mathbf{\#P} \subseteq \mathbf{FPSPACE}, \quad \mathbf{P}^{\mathbf{PP}} = \mathbf{P}^{\mathbf{\#P}}, \quad \mathbf{FL} \subseteq \mathbf{\#L} \subseteq \mathbf{FNC}^{\mathbf{2}}.$$

Counting classes III

Theorem (Toda)

$\mathbf{PH} \subseteq \mathbf{P}^{\#\mathbf{P}}$.

The proof of Toda's theorem consists of proving two inclusions as shown next.

Counting classes IV

Lemma 1

$\mathbf{PH} \subseteq \mathbf{BPP}^{\oplus\mathbf{P}}$.

Proof:

1. $\oplus\mathbf{P}^{\oplus\mathbf{P}} = \oplus\mathbf{P}$
   (Papadimitriou-Zachos)
2. $\mathbf{BPP} \subseteq \mathbf{\Delta}_2^{\mathbf{P}} \subseteq \mathbf{PH}$
   (Sipser, Lautemann, Zachos)
3. $\mathbf{NP} \subseteq \mathbf{BPP} \Longrightarrow \mathbf{PH} \subseteq \mathbf{BPP}$
   (Zachos)
4. $\mathbf{NP} \subseteq \mathbf{RP}^{\oplus\mathbf{P}} \subseteq \mathbf{BPP}^{\oplus\mathbf{P}}$
   (Valiant-Vazirani) (trivial)
5. $\mathbf{NP}^{\oplus\mathbf{P}} \subseteq \mathbf{BPP}^{\oplus\mathbf{P}^{\oplus\mathbf{P}}} \Longrightarrow \mathbf{NP}^{\oplus\mathbf{P}} \subseteq \mathbf{BPP}^{\oplus\mathbf{P}}$
   (by 4. with oracle $\oplus\mathbf{P}$) (by 1.)
6. $\mathbf{NP}^{\oplus\mathbf{P}} \subseteq \mathbf{BPP}^{\oplus\mathbf{P}} \Longrightarrow \mathbf{PH}^{\oplus\mathbf{P}} \subseteq \mathbf{BPP}^{\oplus\mathbf{P}}$
   (by 3. with oracle $\oplus\mathbf{P}$)
7. $\mathbf{PH} \subseteq \mathbf{PH}^{\oplus\mathbf{P}} = \mathbf{BPP}^{\oplus\mathbf{P}}$
   (by 2. and 6.)

□

Counting classes V

Lemma 2.

$\mathbf{BPP}^{\oplus\mathbf{P}} \subseteq \mathbf{P}^{\#\mathbf{P}}$

Proof ommited.

Overview

1 Complexity classes

2 Randomness

3 Interactivity, PCP

4 Counting

5 How to handle **NP**-complete problems

6 Approximation Algorithms

7 Search Complexity

8 Parameterized Complexity

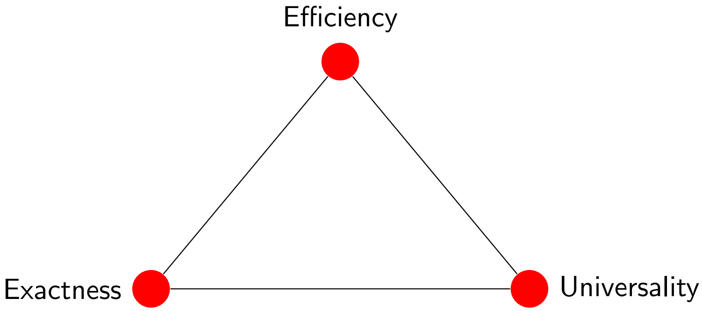
9 Quantum Complexity

How to handle **NP**-complete problems I

We know that the **NP**-hard problems cannot be solved:

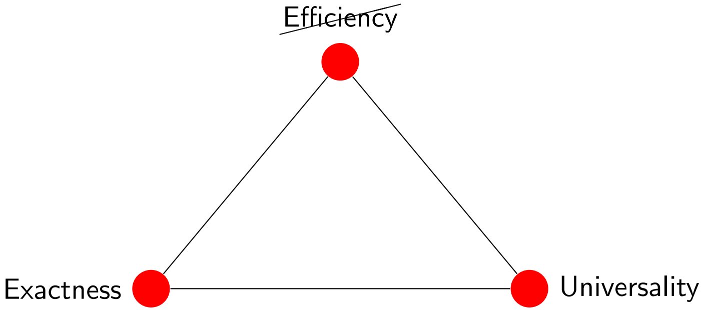
1. Exactly
2. For all instances
3. In polynomial time

How to handle **NP**-complete problems II



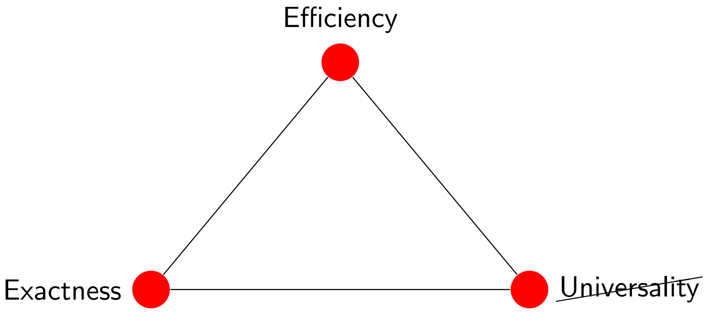
Mission impossible! (...unless $\mathbf{P} = \mathbf{NP}$)

How to handle **NP**-complete problems III



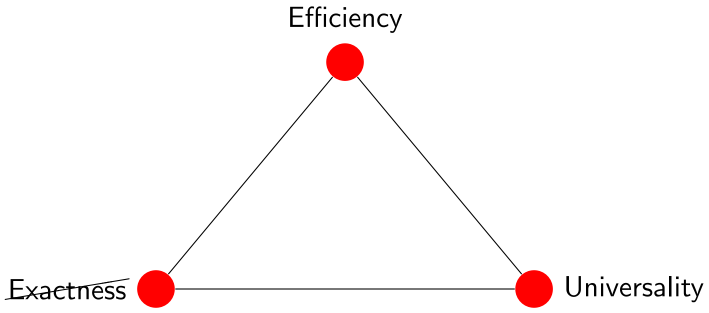
Better exponential-time algorithms
e.g. $O(1.2738^k + kn)$ for VERTEX COVER

How to handle **NP**-complete problems IV



Efficient computation for special cases of the problem (restricted class of instances) e.g. HornSAT

How to handle **NP**-complete problems V



Approximation algorithms!!!

How to handle **NP**-complete problems VI

- If we ignore condition (1), then we have approximation algorithms.
- If we ignore condition (2), then **we can find large subclasses of instances, for which the problem can be solved in polynomial time**, and we can decide whether an input belongs to this class of instances in polynomial time.
  - Pseudopolynomial, Strongly Polynomial
  - Parameterization (e.g. $\textsc{Vertex Cover}(n, k)$)
    Parameterized Complexity ($2^k n^c,\ n^k$ etc)
- If we ignore condition (3), then we classify superpolynomial-time solutions, for example:
  $1.003^n \leq 1.5^n \leq 2^n \leq 5^n \leq n! \leq n^n$
  $n^{\log\log n} \leq n^{\log n} \leq n^{\log^{13} n} \leq n^n$.

Overview

1 Complexity classes

2 Randomness

3 Interactivity, PCP

4 Counting

5 How to handle **NP**-complete problems

6 Approximation Algorithms

7 Search Complexity

8 Parameterized Complexity

9 Quantum Complexity

Approximation Algorithms I

An optimization problem is: (I, S, v, goal).

- I: instances of the problem.
- S: a function that maps every instance to its feasible solutions.
- v: the objective function that maps every feasible solution to a positive integer.
- goal: min or max, for minimization or maximization of the objective function, respectively.

The value of the objective function for the optimum solution on input x is denoted by $\mathrm{OPT}(x)$ and it is equal to $\mathrm{goal}\{v(y) \mid y \in S(x)\}$.

Approximation Algorithms II

Also, we define for every optimization problem, the *underlying* decision problem as follows:

Input: x (the input to the optimization problem) and a bound k.
Question: is $\mathrm{OPT}(x) \geq k$;
(for a maximization problem – analogously "is $\mathrm{OPT}(x) \leq k$?" for a minimization problem.)

Example

For the problem $\textsc{Max-Clique}$, the instance is a graph x, feasible solutions are all complete subgraphs of x (cliques), the objective function is the number of nodes in a clique and goal $=$ max.
The underlying decision problem is the well-known $\textsc{Clique}$.

Approximation Algorithms III

We define the following complexity classes of optimization problems.

Definition

NPO: The class of optimization problems, such that the underlying decision problem is in **NP** (under the condition that there are feasible solutions for every instance).

Definition

PO: The class of optimization problems, such that the underlying decision problem is in **P**.

Approximation Algorithms IV

Many optimization problems are **NP**-hard. Therefore, we look for approximation polynomial-time algorithms that solve such problems.

Definition

A polynomial-time algorithm M is ρ-approximation for a maximization problem if for every $x \in I$ returns a solution $M(x) \in S(x)$ such that:

$$\frac{v(M(x))}{\mathrm{OPT}(x)} \leq \rho.$$

Analogously, we define a ρ-approximation algorithm for minimization problems.

Approximation Algorithms V

The most known subclasses of **NPO**, except for **PO**, are the following:

- **poly-APX**: contains problems for which there is a $p(n)$-approximation algorithm for some polynomial p (where n is the length of the input: $n = |x|$).
- **log-APX**: contains problems for which there is a $\log n$-approximation algorithm (where n is the length of the input: $n = |x|$).
- **APX**: contains problems for which there is a ρ-approximation algorithm for some constant $\rho > 0$.

Approximation Algorithms VI

- **PTAS**: contains problems for which there is a *polynomial-time approximation scheme*, i.e. $(1+\varepsilon)$-approximation algorithm *for every* constant $\varepsilon > 0$.
- **FPTAS**: contains problems for which there is a *fully polynomial-time approximation scheme*, i.e. $(1+\varepsilon)$-approximation algorithm for every constant $\varepsilon > 0$, where the running time is also polynomial with respect to $1/\varepsilon$.
- **FPRAS**: contains problems for which there is a *fully polynomial-time randomized approximation scheme*, i.e. $(1+\varepsilon)$-approximation algorithm for every constant $\varepsilon > 0$ with high probability, where the running time is also polynomial with respect to $1/\varepsilon$.

Classes of Optimization Problems

$$\begin{tikzcd}
\textbf{NPO}\ (\texttt{TSP}) \\
\textbf{poly} - \textbf{APX}\ (\texttt{CLIQUE}) \arrow[u] \\
\textbf{log} - \textbf{APX}\ (\texttt{SET COVER}) \arrow[u] \\
\textbf{APX}\ (\texttt{VERTEX COVER}) \arrow[u] \\
\textbf{PTAS}\ (\texttt{BIN PACKING}) \arrow[u] \\
\textbf{FPRAS}\ (\texttt{\#DNF}) \arrow[u] \\
\textbf{FPTAS}\ (\texttt{0} - \texttt{1 KNAPSACK}) \arrow[u] \\
\textbf{PO}\ (\texttt{LP}) \arrow[u]
\end{tikzcd}$$

Overview

1 Complexity classes

2 Randomness

3 Interactivity, PCP

4 Counting

5 How to handle **NP**-complete problems

6 Approximation Algorithms

7 Search Complexity

8 Parameterized Complexity

9 Quantum Complexity

Search Complexity I

Papadimitriou, ...

Definition

FNP is the class of partial multi-valued functions that are computable by a non-deterministic polynomial-time Turing machine, such that the computation tree, on input x, has on its leaves either ? or the certificate y of the path that satisfies the corresponding predicate $R(x, y)$.

However, the non-deterministic model yields problems in the definition of function classes, because for a given input $x \in \Sigma^*$, there is no unique output string. The attempt to address these problems has led to the definition of the following classes.

- **NPMV**: *The class of partial multi-valued functions computable by a non-deterministic polynomial-time Turing machine, such that the computation tree, on input x, has on its leaves either ? or some of the possible answers of the Turing machine.*
- **NPSV**: *The class that includes single-valued* **NPMV** *functions.*

Search Complexity II

All the classes we are going to see in the rest of this section are subclasses of the class **TFNP**, where "T" declares that these functions are **total**, i.e. there always exists a solution. The existence of a solution for every such class is shown by *existential* proofs of some properties (usually graph-theoretic properties).

Definition

A search problem Π consists of a set of **instances**, and every instance I has a set $Sol(I)$ of **solutions**. Given an instance, the computation of a solution is required.
A search problem is *total* if $Sol(I) \neq \emptyset$, for every instance I.

Local Search Problems I

Definition

A problem Π belongs to the class **PLS** (Polynomial Local Search) if the size of every solution is polynomially bounded in the size of the input, and there are polynomial-time algorithms for the following:

1. Given a string I, check whether I is an instance of Π, and if yes, compute an initial solution that belongs to $Sol(I)$.
2. Given I, s, check whether $s \in Sol(I)$, and if yes, compute the cost of the solution $c_I(s)$.
3. Given I, s, check whether s is a *local* optimum solution, and if not, find a "better" solution $s' \in N_I(s)$, where $N_I(s)$ are the neighbors of the solution s for the instance I.

Every problem in **PLS** admits an algorithm of local search: We use the first algorithm to obtain an initial solution, and then we apply the third algorithm repeatedly, until we reach a local optimum solution. Since the feasible solutions are exponentially many, this process is not necessarily completed in polynomial time.

Local Search Problems II

Example

- Let the problem MAXCUT, where an undirected graph $G(V, E)$ and a weight $w_e \geq 0$ for every edge is given. The feasible solutions correspond to partitions $(S, \overline{S})$ of the set of vertices, and the objective function to the maximization of the total weight of the edges that belong to the partition.
- Two solutions are neighboring if we can transition from one to the other by moving one vertex from the partition to its complement.
- We start from an arbitrary partition $(S, \overline{S})$, and as long as there is a better neighboring solution, we transition to it.
- The algorithm terminates when there is no better neighboring solution, i.e. when we reach a local optimum solution.

Note that the number of feasible solutions for MAXCUT is exponential in the size of the input.

The structure of the problem induces a graph G, where the vertices are feasible solutions, and two vertices are connected through an edge, if we can transition from one to the other by a simple change.

Local Search Problems III

Note that every problem in the class **PLS** belongs to **TFNP**, because every directed acyclic graph (DAG) has a sink, or equivalently, every finite set of numbers has a minimal element.

Definition

A reduction from a search problem Π_1 to a problem Π_2 consists of two polynomial-time algorithms:

1. An algorithm A that maps instances $x \in \Pi_1$ to instances $A(x) \in \Pi_2$.
2. An algorithm B that maps solutions y of Π_2 on input $A(x)$ to solutions $B(y)$ of Π_1 on input x.

In the case of **PLS**, the solutions that are mapped by B are the local optima.

The next problems are **PLS**-complete:

- `MAXCUT`
- `TSP`
- `MAXSAT`
- `PURE NASH EQUILIBRIUM` in congestion games.

Nash equilibrium I

- Let (S, f) be a game of *n players*, where S_i is the set of *strategies* of palyer i, $S = S_1 \times S_2 \times \cdots \times S_n$ the set of *strategy profiles* of the players, and $f(x) = (f_1(x), \ldots, f_n(x))$ the *utility function* computed on $x \in S$.
- Let x_i be the strategy profile of player i and x_{-i} the strategy profiles of all the other players except for i.
- Given that every player $i \in \{1, \ldots, n\}$ chooses strategy x_i, the strategy profiles chosen are described by the vector $x = (x_1, \ldots, x_n)$ and the utility of each player is computed by the utility function $f_i(x)$.

Definition (Nash equilibrium)

A strategy profile $x^* \in S$ is a Nash equilibrium, if no player can do better by deviating from his strategy, if the strategies of the others remain unchanged. In other words:

$$\forall i, x_i \in S_i : f_i(x_i^*, x_{-i}^*) \geq f_i(x_i, x_{-i}^*)$$

Note that in the case that the above inequality is strict for all players and their strategies, we have the definition of a *strict* Nash equilibrium. Analogously, if a player can change his strategy and preserve (but not necessarily increase) his utility, we say that we have a *weak* Nash equilibrium.

Nash equilibrium II

The following topological theorem played an important role in the proof of Nash that there is a (mixed) Nash equilibrium in every finite game.

Definition

Let $S \subset \mathcal{R}^n$ be a convex and compact (closed and bounded) space. For every continuous function $f : S \rightarrow S$, there is an element x_0 such that $f(x_0) = x_0$ (fixed point).

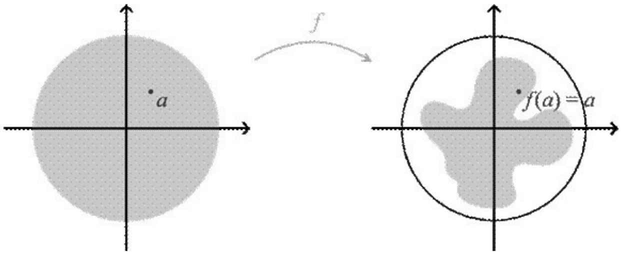


Figure: Fixed Point Theorem

Nash equilibrium III

Now, we are going to discuss Sperner's Lemma, which is the combinatorial analogue of the Fixed Point Theorem. For simplicity, we are going to present its version in the case of two dimensions. Let's assume we have the rectangle ABCD and a triangulation on it, as shown in the next figure. In addition, we assume that the vertices are colored according to the following rules:

1. The vertices on the side AB must have color 1 (yellow), those on sides BC and CD color 2 (blue) and those on side DA color 3 (red).
2. The vertices that are not on the borders of ABCD can have any color.

Nash equilibrium IV

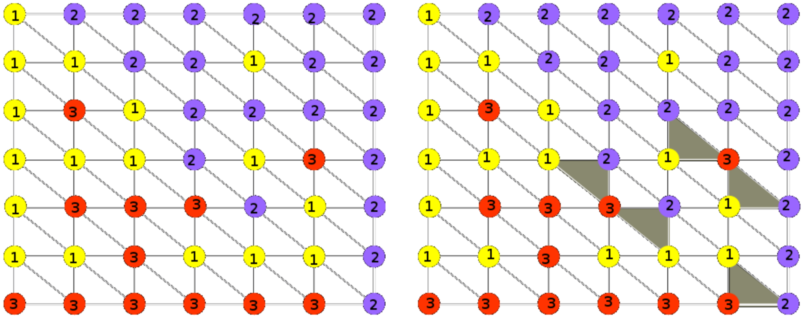


Figure: A valid coloring of an arbitrary triangulation.

Nash equilibrium V

Lemma (2D-Sperner)

In every triangulation with a *valid* coloring (as described above), there is a tri-chromatic triangle, i.e. a triangle, every vertex of which has a different color. It holds, also, that the number of tri-chromatic triangles is odd.

Proof: Given a valid coloring, we can always construct an "artificial" tri-chromatic triangle, outside the rectangle, on its bottom left side. Starting from this triangle, we are going to define a walk through the triangles, that will have a tri-chromatic triangle as its final destination. Let KL be the edge of the artificial triangle (the starting point) from which we can enter the surface ABC. Let k and l the colors of the vertices that define this edge, yellow and red, respectively. The rule according to which we walk through the triangles is the following: *Whenever we find edge with colors 1 and 3 in the current triangle, we traverse it, only if the color 1 is on our left*. We can easily verify that this walk ends when we reach a tri-chromatic triangle.

Nash equilibrium VI

Since the number of triangles is finite, our walk has to stop, i.e. has to find a tri-chromatic triangle. Otherwise, the walk has to go into an infinite loop, which cannot happen because of the rule we defined.

Since we have proved that tri-chromatic triangles exist in ABCD, we can show that their number is odd in a similar way. More precisely, starting from a tri-chromatic triangle and following the reverse walk with respect the one that we described above, we can show that we will find another tri-chromatic triangle. We conclude that tri-chromatic triangles come in pairs. Since one of them is the artificial triangle, we conclude that their number is odd. □

Nash equilibrium VII

Brouwer's Fixed Point Theorem can be proven by using the above lemma. More precisely, we consider a weak version of the Fixed Point Theorem, i.e. that there is fixed point x_0 (in the weak sense) such that $|f(x_0) - x_0| \leq \epsilon$, for $\epsilon > 0$. In this case, if we consider a Sperner triangulation on the domain of f, we require that the diameter of the triangles is ϵ, and we color the vertices with three colors based on the direction of $f(x) - x$ at these points, then we can see (using some imagination) that every tri-chromatic triangle corresponds to a weak fixed point. Finally, we can pass from the weak to the strong form of the Fixed Point Theorem using the fact that the space is compact and taking the limit when ϵ approaches zero. Below we can see an example in the case of two dimensions:

Nash equilibrium VIII

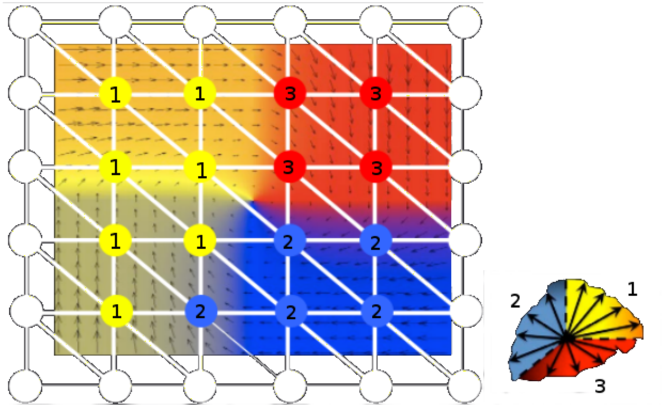


Figure: Finding the fixed point using Sperner's lemma in two dimensions.

Nash equilibrium IX

Definition (NASH)

The input consists of a number n of players, the sets of strategies of every player, S_i, and the utility function $f(x)$. An approximation parameter $\epsilon > 0$ is also given in the input.
The NASH or $\epsilon -$ NASH problem requires the computation of an equilibrium, where no player can increase her utility more than ϵ by changing her strategy unilaterally.

Definition (SPERNER)

Let the input be a grid such that it consists of $2^n \times 2^n$ vertices and the vertices on its borders have a valid coloring (as the one we described before). We consider that the color of each interior vertex is given by a circuit C, which, on input the coordinates x and y of a vertex, returns one of the three colors.
The SPERNER problem is the problem of finding and returning a tri-chromatic triangle.

Nash equilibrium X

By the previous analysis about the relationship among Nash, Fixed Point Theorem and Sperner, one can be convinced, at least intuitively, that $\epsilon - \texttt{NASH}$ problem is reducible to the corresponding $\epsilon - \texttt{BROUWER}$, which in turn is reducible to $\texttt{SPERNER}$. The reductions are based on techniques that are similar to those used to connect the existential arguments for these notions.

In particular, we can see that $\texttt{NASH}$ can be solved by solving $\epsilon - \texttt{BROUWER}$. Moreover, the (almost) fixed points of $\epsilon - \texttt{BROUWER}$ can be found by finding tri-chromatic triangles of the corresponding $\texttt{SPERNER}$ problem, if we consider a grid of appropriate size depending on the approximation parameter ϵ.

Nash equilibrium XI

Before we define the class **PPAD**, recall the proof of the fact that there exist tri-chromatic triangles in the Sperner grid.
We can consider an ancillary graph, the vertices of which correspond to triangles and an edge connects two vertices iff they represent two triangles that we can move from one to the other using the rule we defined for the walk.
The resulting graph depicts the complexity of the search problem and consists of isolated nodes, simple paths and cycles as shown in the following figure:

Nash equilibrium XII

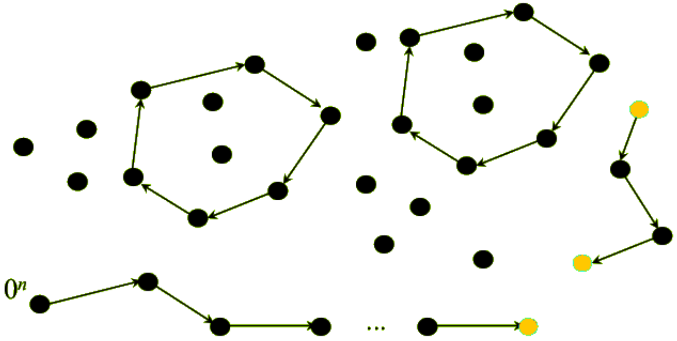


Figure: The search graph of the SPERNER problem.

Nash equilibrium XIII

Definition

A problem Π belongs to the class **PPAD** if each of its solutions has polynomial length with respect to the input length and there are polynomial-time algorithms for the following:

1. Given a string I, check whether I is an instance of Π, and if yes, compute an initial solution $s_0 \in Sol(I)$.
2. Given I, s, check whether $s \in S(I)$, and if yes, return a solution $pred(s) \in Sol(I)$, such that $pred(s_0) = s_0$.
3. Given I, s, check whether $s \in S(I)$, and if yes, return a solution $succ(s) \in Sol(I)$, such that $succ(s_0) \neq s_0$ and $pred(succ(s_0)) = s_0$.

Nash equilibrium XIV

Like in the case of **PLS**, the definition of **PPAD** induces a directed graph $G(Sol(I), E)$, the vertices of which represent feasible solutions and $E = \{(u, v) : \; u \neq v, succ(u) = v, pred(v) = u\}$.

The process described in the definition of the class allows us to find a solution s, except for s_0, that has $indegree(s) + outdegree(s) = 1$, by following the path in G starting from s_0. Since it holds that $indegree(s_0) + outdegree(s_0) = 1$, there exists at least a solution $s \neq s_0$, because of the well-known lemma:

> *"Every graph has an even number of nodes of odd degree."*

Nash equilibrium XV

Note that the structure of the graph induced by the definition of **PPAD** and the functions $succ(u)$ and $pred(u)$ is exactly the same as that of the search graph of the `SPERNER` problem. This is not a coincidence, since `SPERNER` $\in$ **PPAD**. Furthermore, because of the reductions we described, it holds that `BROUWER` $\in$ **PPAD** and `NASH` $\in$ **PPAD**.
In their recent paper "The Complexity of Computing a Nash Equilibrium", Daskalakis, Goldberg and Papadimitriou proved that the `NASH` problem with three players is **PPAD**- complete.

The following problems are **PPAD**-complete:

- `END OF THE LINE`
- `SPERNER`
- `NASH`

Other classes I

Using several lemmas, we can define other classes as well:

- **PPADS**: Similarly to **PPAD**, but in this case we search for a sink, i.e. a solution with *indegree* $= 1$ and *outdegree* $= 0$.
- **PPA**: The analogue of **PPAD**, but with an undirected graph (there is a 'neighborhood' function instead of functions *succ* and *pred*).
- **PPP**: In this class, a function f is defined on the set of solutions: Our goal is to find either a solution that is mapped to the initial solution, or two solutions y and y', such that $f(x, y) = f(x, y')$. Such solutions always exist because of the *Polynomial Pigeonhole Principle*.
- **CLS**: The analogue of **PLS** for continuous spaces and functions (Continuous Local Search). The class **CLS** contains the problems of searching an approximate local optimum of a continuous function, using an oracle f, where f is also a continuous function.

Other classes II

$$\begin{tikzcd}
 & & \mathbf{NPMV} & \\
 & & \mathbf{FNP} \arrow[u] & \\
 & & \mathbf{TFNP} \arrow[u] & \\
\mathbf{NPSV} \arrow[uuurr] & \mathbf{PPA} \arrow[ur] & \mathbf{PPP} \arrow[u] & \mathbf{PLS} \arrow[ul] \\
 & & \mathbf{PPADS} \arrow[ul] \arrow[u] & \\
 & & \mathbf{PPAD} \arrow[u] & \\
 & & & \cdot \arrow[ul] \arrow[uuu] \\
 & & & \mathbf{CLS} \arrow[u] \\
 & & \mathbf{FP} \arrow[ur] \arrow[uuu] \arrow[uuuuull] &
\end{tikzcd}$$

Overview

1 Complexity classes

2 Randomness

3 Interactivity, PCP

4 Counting

5 How to handle **NP**-complete problems

6 Approximation Algorithms

7 Search Complexity

8 Parameterized Complexity

9 Quantum Complexity

Parameterized Complexity I

Definition

A *parameterization* of Σ^* is a *recursive* function $k : \Sigma^* \to \mathbb{N}$.
A *parameterized problem* is an ordered pair (L, k), where $L \subseteq \Sigma^*$ and k is a parameterization of Σ^*.
An algorithm A is an FPT-algorithm (Fixed Parameter Tractable) with respect to parameter k, if there is a *computable* function f and a polynomial p, such that for every $x \in \Sigma^*$, the algorithm A decides the problem in $O(f(k(x)) \cdot p(|x|))$ time.

Definition

We define **FPT** to be the class of parameterized problems that can be solved by an FPT-algorithm.

Parameterized Complexity II

The next step, analogously to classical Complexity Theory, is to connect these problems using reductions.

Definition

Let (L, k), (L', k') be parameterized problems. (L, k) reduces to (L', k') with an *FPT*-reduction (symb. $L \leq_{FPT} L'$) if there is algorithm R such that:

1. For every $x \in \Sigma^*$, $x \in L \Leftrightarrow R(x) \in L'$
2. R is computable by an *FPT*-algorithm.
3. $k' = g(k)$, where $g : \mathbb{N} \to \mathbb{N}$ is a computable function.

If $A \leq_{FPT} B$ and $B \leq_{FPT} A$, then we say that A, B are *FPT*-equivalent (symb. $A \equiv_{FPT} B$).

Example

Let the problem `pSAT`, where we are given a propositional formula ϕ, and a parameter k, that represents the number of variables in ϕ. Is ϕ satisfiable?

Parameterized Complexity III

We are going to define the parameterized analogue of the class **NP**:

Definition

Let (L, k) be a parameterized problem. (L, k) belongs to the class **paraNP** if there exists a computable function $f : \mathbb{N} \to \mathbb{N}$, a polynomial p and a *non-deterministic* algorithm, which, for every $x \in \Sigma^*$, decides the problem in $O(f(k(x)) \cdot p(|x|))$ time.

We say that a parameterized problem is trivial, if $L = \emptyset$ or $L = \Sigma^*$, and we define the i-th slice of the problem (L, k) as the problem:

$$(L, k)_i = \{x \in L | k(x) = i\}$$

The following characterization of the class **paraNP** holds:

Theorem

Let $(L, k) \in$ **paraNP** be a non-trivial parameterized problem. Then the finite union of slices of (L, k) is **NP**-complete *iff* (L, k) is **paraNP**-complete.

Parameterized Complexity IV

We are going to define the parameterized analogue of the class **EXP**:

Definition

Let (L, k) be a parameterized problem. (L, k) belongs to the class **XP** if there exists a computable function f, and an algorithm A, such that, for every $x \in \Sigma^*$,decides the problem in $O(|x|^{f(k(x))})$ time.

It holds that **FPT** $\subset$ **XP**, whereas the relationship of **paraNP** with **XP** is unknown.

Definition

A non-deterministic Turing machine M is called *k-restricted* if there is a computable function $g : \mathbb{N} \rightarrow \mathbb{N}$ and a polynomial p such that M needs $f(k(x)) \cdot p(|x|)$ computational steps, and at most $g(k(x)) \cdot \log |x|$ of them are non-deterministic.

Definition

We define the class **W[P]** as the class that contains all the parameterized problems (L, k), which are decidable by k-restricted Turing machines.

Parameterized Complexity V

Most **NP**-complete problems parameterized so that they do not belong to **FPT**, are in **W**[**P**].
The following inclusions hold: $\mathbf{FPT} \subseteq \mathbf{W[P]} \subseteq \mathbf{XP} \cap \mathbf{paraNP}$.

$$\begin{tikzcd}
\mathbf{paraNP} & & \mathbf{XP} \\
& \cdot \arrow[ul] \arrow[ur] & \\
& \mathbf{W[P]} \arrow[u] & \mathbf{AH} \arrow[uu] \\
& \mathbf{WH} \arrow[u] \arrow[ur] & \\
& \mathbf{FPT} \arrow[u] &
\end{tikzcd}$$

Overview

1 Complexity classes

2 Randomness

3 Interactivity, PCP

4 Counting

5 How to handle **NP**-complete problems

6 Approximation Algorithms

7 Search Complexity

8 Parameterized Complexity

9 Quantum Complexity

Quantum Complexity — Computational Models I

The most widely used model for quantum computations and quantum algorithms is the quantum circuit and specifically, **uniform** families of quantum circuits (in the sense that there is a polynomial-time algorithm that returns their description). These circuits are similar to logic circuits, which implement Boolean functions (recall that every Boolean function can be computed by a logic circuit that uses only the logic gates NOT and AND).

The gates of a quantum circuit can be constructed as combinations of the quantum gates CNOT, H, T.

Quantum Complexity — Computational Models II

These gates are linear unitary transformations: A linear transformation T, on a complex vector space, is unitary if $T^{-1} = T^*$, i.e. if its inverse is equal to its adjoint or conjugate transpose. Unitary transformations preserve the Euclidean Norm, i.e. they are isometries.

$$\mathrm{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix},$$

$$H = \frac{1}{\sqrt{2}} \cdot \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix},$$

$$T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i \cdot \frac{\pi}{4}} \end{pmatrix},$$

where $i^2 = -1$.

Quantum Complexity — Computational Models III

- The inputs and outputs of quantum algorithms are vectors.
- Every logic circuit can be transformed into a quantum circuit: the latter are a generalization of the former.
- Apart from the quantum circuit, other models have been proposed, such as quantum Turing machines.
- Quantum circuits are mostly used in the definitions of time quantum complexity classes, whereas quantum Turing machines are mostly used in the definitions of space quantum complexity classes.
- Regarding computability, conventional computers are equivalent to quantum ones (Church-Turing). However, we have strong evidence that quantum computers are more efficient.

Quantum Complexity Classes I

- First, we will define classes that are quantum analogues of classical classes: **BQP** corresponds to **BPP** $\supseteq$ **P**, and **QMA** to **MA** $\supseteq$ **NP**.
- The main difference between quantum and classical classes is the following:
  - To describe a quantum state we need exponentially large classical information with respect to the size of the quantum system it describes.
  - To describe a classical state we need linearly large classical information with respect to the size of the classical state it describes.

  This means that a quantum state is a superposition of classical states. For example, a quantum state of size n requires 2^n complex numbers to be described, whereas a classical state of size n just requires n bits.

Definition (**BQP**)

For a language L, it holds that $L \in$ **BQP** if there is a family of *quantum* circuits $\{Q_i\}$ of polynomial time in i , such that for every $x \in \Sigma^*$:

- $x \in L \Rightarrow \Pr\left[\text{το } Q_{|x|} \text{ αποδέχεται το } x\right] \geq \frac{2}{3}$
- $x \notin L \Rightarrow \Pr\left[\text{το } Q_{|x|} \text{ αποδέχεται το } x\right] \leq \frac{1}{3}$

Quantum Complexity Classes II

The class **BQP** represents the problems that can be efficiently solved by using quantum computations.

Quantum Complexity Classes III

Definition (QMA)

For a language L, it holds that $L \in \mathbf{QMA}$ if there is a family of *quantum* circuits $\{Q_i\}$ of polynomial time in i , such that for every $x \in \Sigma^*$:

- $x \in L \Rightarrow$ $\exists$ a quantum state K of polynomial length in $|x|$: $\Pr\left[Q_{|x|} \text{ accepts } (x, K)\right] \geq \frac{2}{3}$
- $x \notin L \Rightarrow$ $\forall$ quantum state K of polynomial length in $|x|$: $\Pr\left[Q_{|x|} \text{ accepts } (x, K)\right] \leq \frac{1}{3}$

The class **QMA** represents the problems that can be efficiently *verified* by using quantum computations. It holds that $\mathbf{BQP} \subseteq \mathbf{QMA}$.

Quantum Complexity Classes IV

Definition (QCMA)

For a language L, it holds that $L \in$ **QCMA** if there is a family of *quantum* circuits $\{Q_i\}$ of polynomial time in i , such that for every $x \in \Sigma^*$:

- $x \in L \Rightarrow$ $\exists$ a classical state C of polynomial length in $|x|$: $\Pr\left[Q_{|x|} \text{ accepts } (x, C)\right] \geq \frac{2}{3}$
- $x \notin L \Rightarrow$ $\forall$ classical state C of polynomial length in $|x|$: $\Pr\left[Q_{|x|} \text{ accepts } (x, C)\right] \leq \frac{1}{3}$

The class **QCMA** can be seen also as **CMQA**, or **MQA**.

Quantum Complexity Classes V

Definition ($\mathbf{QCMA_1}$)

For a language L, it holds that $L \in \mathbf{QCMA_1}$ if there is a family of *quantum* circuits $\{Q_i\}$ of polynomial time in i , such that for every $x \in \Sigma^*$:

- $x \in L \Rightarrow$ $\exists$ a classical state C of polynomial length in $|x|$:
  $\Pr\left[Q_{|x|} \text{ accepts } (x, C)\right] = 1$
- $x \notin L \Rightarrow$ $\forall$ classical state C of polynomial length in $|x|$:
  $\Pr\left[Q_{|x|} \text{ accepts } (x, C)\right] \le \frac{1}{3}$

It can be proven that $\mathbf{QCMA_1} = \mathbf{QCMA}$.

Quantum Complexity Classes VI

Definition (PQP)

For a language L, it holds that $L \in \mathbf{PQP}$ if there is a family of *quantum* circuits $\{Q_i\}$ of polynomial time in i , such that for every $x \in \Sigma^*$:

- $x \in L \Rightarrow \Pr\left[Q_{|x|} \text{ accepts } x\right] > \frac{1}{2}$
- $x \notin L \Rightarrow \Pr\left[Q_{|x|} \text{ accepts } x\right] \leq \frac{1}{2}$

It holds that $\mathbf{PP} \subseteq \mathbf{PQP}$.

Definition (QIP)

For a language L, it holds that $L \in \mathbf{QIP}$ if there is a family of *quantum* circuits $\{Q_i\}$ of polynomial time in i , such that for every $x \in \Sigma^*$:

- $x \in L \Rightarrow (\exists P) \Pr\left[P \text{ convinces } V_{|x|} \text{ to accept } x\right] \geq \frac{2}{3}$
- $x \notin L \Rightarrow (\forall P) \Pr\left[P \text{ convinces } V_{|x|} \text{ to accept } x\right] \leq \frac{1}{3}$

It holds that $\mathbf{IP} \subseteq \mathbf{QIP}$.

Quantum Complexity Classes VII

Definition (BQPSPACE)

For a language L, it holds that $L \in$ **BQPSPACE** if there is a polynomial space quantum Turing machine M, such that for every $x \in \Sigma^*$:

- $x \in L \Rightarrow \Pr[M \text{ accepts } x] \geq \frac{2}{3}$
- $x \notin L \Rightarrow \Pr[M \text{ accepts } x] \leq \frac{1}{3}$

Definition (PQPSPACE)

For a language L, it holds that $L \in$ **PQPSPACE** if there is a polynomial space quantum Turing machine M, such that for every $x \in \Sigma^*$:

- $x \in L \Rightarrow \Pr[M \text{ accepts } x] > \frac{1}{2}$
- $x \notin L \Rightarrow \Pr[M \text{ accepts } x] \leq \frac{1}{2}$

It holds that **PSPACE** $=$ **BPPSPACE** $\subseteq$ **BQPSPACE** $\subseteq$ **PQPSPACE**.

Quantum Complexity Classes VII

Inclusions are depicted in the following diagram.

$$\begin{tikzcd}
 & \mathbf{QIP} = \mathbf{PSPACE} & \\
 & \mathbf{PP} = \mathbf{PQP} \arrow[u] & \\
 & \mathbf{QMA} \arrow[u] & \\
 & \mathbf{QCMA} \arrow[u] & \\
\mathbf{BQP} \arrow[ur] & & \mathbf{MA} \arrow[ul] \\
\mathbf{BPP} \arrow[u] \arrow[urr] & & \mathbf{NP} \arrow[u] \\
 & \mathbf{P} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Quantum Complexity Classes – Fundamental Results I

Theorem

- $\mathbf{BQP}^{\mathbf{BQP}} = \mathbf{BQP}$
- There exists an oracle A, such that $\mathbf{BQP}^A \not\subseteq \mathbf{BPP}^A$.
- There exists an oracle A, such that $\mathbf{NP}^A \not\subseteq \mathbf{BQP}^A$.
- $\mathbf{QMA} \subseteq \mathbf{PP} = \mathbf{PQP}$
- $\mathbf{PSPACE} = \mathbf{QIP} = \mathbf{BQPSPACE} = \mathbf{PQPSPACE}$

Theorem (Grover)

There is a quantum algorithm that computes the position of (let's say a unique) object s, that belongs to a list of size $N \in \mathbb{N}$, in $O(\sqrt{N})$ steps. The efficiency of this algorithm can be proven to be optimal.

Definition (`FACTORING`)

Input: A natural number n.
Output: The factorization of n into a product of powers of prime numbers.

Quantum Complexity Classes – Fundamental Results II

Definition (`DISCRETE LOGARITHM`)

Input: Two elements a, b of a group $(G, \cdot)$. Recall that $a^0 = e$, where e is the identity element of $(G, \cdot)$, and $a^m = a^{m-1}a$, $\forall m \in \mathbb{N}^{\geq 1}$.
Output: A natural number c such that $a^c = b$, if such a number exists, otherwise an indication that such a c does not exist.

Theorem

$\texttt{FACTORING} \in \mathbf{BQP}$, $\texttt{DISCRETE LOGARITHM} \in \mathbf{BQP}$ (for the corresponding decision problems).

Both problems belong to **NP**, but we do not know whether they lie in **P**.

Quantum Complexity Classes – Fundamental Results III

Definition (`GROUP NON-MEMBERSHIP`)

Input: A subgroup $(H, \cdot)$ of a group $(G, \cdot)$, and an element $g \in G$ (the subgroup H is known by its generators).
Question: Does it hold that $g \notin H$?

Theorem

`GROUP NON-MEMBERSHIP` $\in$ **QMA**.

We do not know whether `GROUP NON-MEMBERSHIP` $\in$ **NP**.

Analogously, the problem `2-LOCAL HAMILTONIAN` is **QMA**-complete.
There are other problems that have been characterized as **QMA**-complete.

Quantum Complexity Classes – Open Problems I

- $\mathbf{NP} \overset{?}{\subseteq} \mathbf{BQP}$, $\mathbf{BQP} \overset{?}{\subseteq} \mathbf{NP}$.
- $\mathbf{QMA} \overset{?}{\subseteq} \mathbf{QCMA}$. Is there an oracle A, such that $\mathbf{QMA}^A \not\subseteq \mathbf{QCMA}^A$?
- $\mathbf{BQP} \overset{?}{\subseteq} \mathbf{BPP}$, $\mathbf{BQP} \overset{?}{\subseteq} \mathbf{P}$.
- $\mathbf{BQP} \overset{?}{\subseteq} \mathbf{PH}$. Recall that $\mathbf{BPP} \subseteq \mathbf{PH}$. Is there an oracle A, such that $\mathbf{BQP}^A \not\subseteq \mathbf{PH}^A$?
- Are there $\#\mathbf{P}$-complete problem of quantum nature?
- `GRAPH ISOMORPHISM` $\overset{?}{\in} \mathbf{BQP}$? Recall that `GRAPH ISOMORPHISM` $\in \mathbf{NP}$, but we do not know whether `GRAPH ISOMORPHISM` $\in \mathbf{P}$.
- `GRAPH NON-ISOMORPHISM` $\overset{?}{\in} \mathbf{QMA}$? We know that `GRAPH ISOMORPHISM` $\in \mathbf{coNP} \cap \mathbf{AM}$.

Counting Complexity

Stathis Zachos

Overview

1. Counting problems: The class #P
2. Approximation of counting problems
3. Counting problems with easy decision version: the class #PE
4. Counting problems with easy decision and self-reducibility: the class TotP

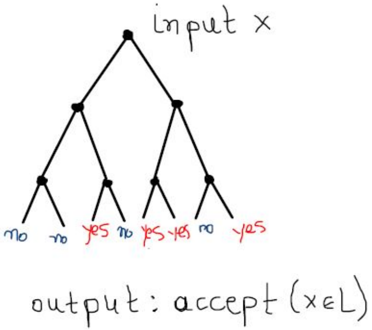
Overview

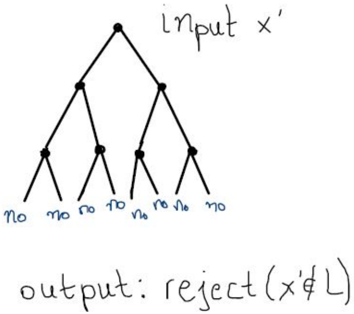
1. Counting problems: The class #P
2. Approximation of counting problems
3. Counting problems with easy decision version: the class #PE
4. Counting problems with easy decision and self-reducibility: the class TotP

The class NP

- **NP**: $L \in$ NP iff there is a non-deterministic polynomial-time Turing machine (NPTM) M s.t. for every $x \in \Sigma^*$:

$$x \in L \Leftrightarrow M(x) \text{ has an accepting computation path}$$





Why counting?

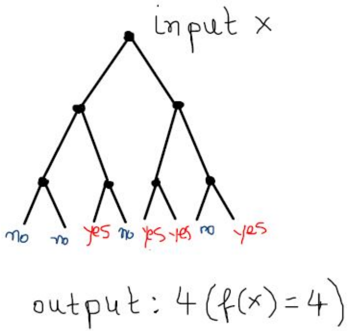
Why not? But also many interesting problems from different areas can be expressed as counting problems:

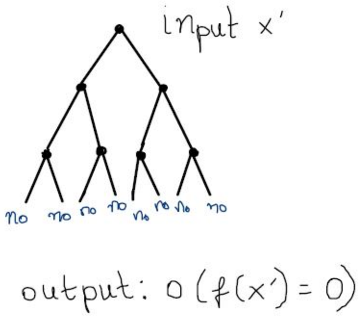
- Computing the partition function in **statistical physics**.
- Computing the volume of a convex body in **computational geometry**.
- Computing the permanent in **linear algebra**.
- Computing the social cost of a given mixed Nash equilibrium in selfish games in **algorithmic game theory**.
- There is a framework for **optimization under uncertainty** which requires counting approximate solutions for the corresponding decision problem.

The class #P (Valiant 1979)

- An NPTM outputs 'yes' or 'no' and we want to compute the number of accepting computation paths.
- **#P**: $f \in \#\mathrm{P}$ iff there is an NPTM M s.t. for every $x \in \Sigma^*$:

$$f(x) = \#(\text{accepting paths of } M \text{ on input } x).$$





- Every decision problem in NP has a counting version in #P.

- Counting the number of accepting paths is at least as hard as deciding if an accepting path exists.

- Accepting paths correspond to solutions of computational problems.

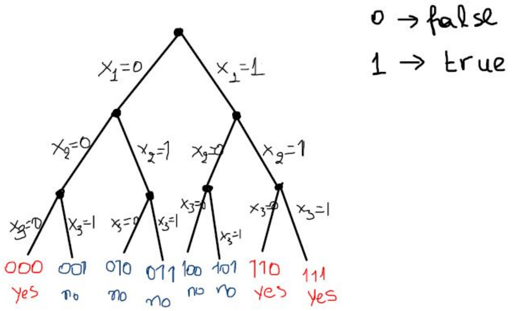
Example (1)

Sat
Input: A propositional formula ϕ in conjunctive normal form.
Output: Accept if ϕ is satisfiable. Otherwise, reject.

#Sat
Input: A propositional formula ϕ in conjunctive normal form.
Output: The number of satisfying assignments of ϕ.

Example (1)

- Let $\phi = (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_3) \wedge (\neg x_1 \vee x_2)$ be the input.



- The output of $\textsc{Sat}$ is "accept", or $\phi \in \textsc{Sat}$.
- The output of $\#\textsc{Sat}$ is 3, or $\#\textsc{Sat}(\phi) = 3$.
- $\textsc{Sat} \in \mathsf{NP}$ and $\#\textsc{Sat} \in \#\mathsf{P}$.

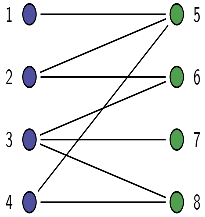
Example (1)

- Cook's Theorem 1971: $\textsc{Sat}$ is NP-complete.
- $\#\textsc{Sat}$ is #P-complete (under Karp reductions).

Example (2)

2-Coloring: Is a graph 2-colorable?

⋆ Equivalently, is the graph bipartite?



#2-Colorings: How many 2-colorings does a graph have?

⋆ Every connected bipartite graph has exactly two 2-colorings (swap the colors).

Example (2)

- 2-COLORING is in P.
- #2-COLORINGS is in FP.

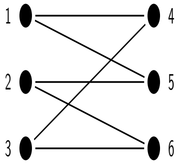
Example (3)

Bipartite Perfect Matching: Does a bipartite graph G have a perfect matching?

⋆ It is reducible to the Max-Flow problem.

#Bipartite Perfect Matchings: How many perfect matchings does a bipartite graph have?

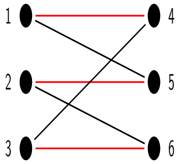
⋆ We write the **biadjacency matrix** A of G, i.e. the left vertices correspond to rows and the right vertices correspond to columns.



$$\Longrightarrow \quad A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 4 & 5 & 6 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix} \end{array}$$

Example (3)

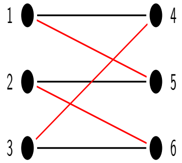
Every perfect matching in G corresponds to a combination of 1's in A such that exactly one 1 appears in every row and every column. For example,



$$\Downarrow$$

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 4 & 5 & 6 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix} \end{array}$$

Example (3)



$$\Downarrow$$

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 4 & 5 & 6 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix} \end{array}$$

Example (3)

#Bipartite Perfect Matchings or Permanent: How many perfect matchings does a bipartite graph have?

⋆ Equivalently, compute the permanent of $A = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}$:

$$\textsc{Permanent}(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i,\sigma(i)}.$$

Note that

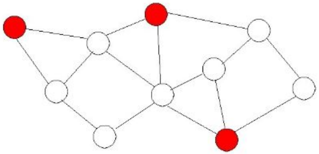
$$\det(A) = \sum_{\sigma \in S_n} \left(\operatorname{sgn}(\sigma) \prod_{i=1}^{n} a_{i,\sigma(i)} \right).$$

Example (3)

- Bipartite Perfect Matching is in P.
- Valiant 1979: Permanent is #P-complete (under Cook reductions).

Example (4)

INDEPENDENT SET: Is there an independent set (of any size) in a graph?



#INDEPENDENT SETS: How many independent sets (of any size) are there in a graph?

Example (4)

- INDEPENDENT SET is trivial.
- #INDEPENDENT SETS is #P-complete (under Cook reductions).

Problem	**Decision version**	**Counting version**
SAT	NP-complete	#P-complete (under Karp)
2-COLORING	P	FP
BIPARTITE PERFECT MATCHING	P	#P-complete (under Cook)
INDEPENDENT SET	trivial	#P-complete (under Cook)

Reductions between counting functions

- **Cook** (poly-time Turing)

$$f \leqslant_T^p g : \ f \in \mathsf{FP}^g$$

- **Karp** / parsimonious (poly-time many one)

$$f \leqslant_m^p g : \ \exists h \in \mathsf{FP}, \ \forall x \ f(x) = g(h(x))$$

Reductions between counting functions

- **Cook** (poly-time Turing)

$$f \leqslant_T^p g : \; f \in \mathrm{FP}^g$$

- **Karp** / parsimonious (poly-time many one)

$$f \leqslant_m^p g : \; \exists h \in \mathrm{FP}, \; \forall x \; f(x) = g(h(x))$$

- PERMANENT is #P-complete under Cook reductions.
  - If PERMANENT were #P-complete under Karp reductions, then P = NP.
- #SAT is #P-complete under Karp reductions (thus also Cook reductions).

NP-completeness versus #P-completeness

- Is a decision problem corresponding to a #P-complete problem under Cook reductions, NP-complete?

NP-completeness versus #P-completeness

- Is a decision problem corresponding to a #P-complete problem under Cook reductions, NP-complete?
  - We just answered NOT ALWAYS!
  - Thus #P-completeness (under Cook reductions) $\not\Rightarrow$ NP-completeness.
  - *Under Karp reductions*: Every #P-complete has an NP-complete decision version.
- What about the converse?

NP-completeness versus #P-completeness

- Is a decision problem corresponding to a #P-complete problem under Cook reductions, NP-complete?
  - We just answered NOT ALWAYS!
  - Thus #P-completeness (under Cook reductions) $\not\Rightarrow$ NP-completeness.
  - *Under Karp reductions*: Every #P-complete has an NP-complete decision version.

- What about the converse?

- Conjecture: Every NP-complete problem has a #P-complete counting version under Cook reductions.
  - We believe that NP-completeness $\Rightarrow$ #P-completeness (under Cook reductions).
  - *Under Karp reductions*: It has been proven that there is an NP-complete problem that its counting version is not #P-complete unless P = NP.

Some basic inclusions

- $\mathsf{FP} \subseteq \#\mathsf{P} \subseteq \mathsf{FPSPACE}$.
- $\mathsf{NP} \subseteq \mathsf{P}^{\#\mathsf{P}[1]}$.
- If $\mathsf{FP} = \#\mathsf{P}$, then $\mathsf{P} = \mathsf{NP}$.
- **Toda's Theorem:** $\mathsf{PH} \subseteq \mathsf{P}^{\#\mathsf{P}[1]}$.

Overview

1 Counting problems: The class #P

2 Approximation of counting problems

3 Counting problems with easy decision version: the class #PE

4 Counting problems with easy decision and self-reducibility: the class TotP

Exact and efficient counting

Exact and efficient counting is rare:

- #2-COLORINGS.
- #PERFECT MATCHINGS in planar graphs.
- #SPANNING TREES in general graphs.

- The last two problems are reducible to the computation of the determinant.
- Of course, all these problems have a decision version in P!

Approximate counting

Definition

A fully polynomial randomized approximation scheme (fpras) for a counting problem $f : \Sigma^* \to \mathbb{N}$ is a randomized algorithm that takes as input an instance $x \in \Sigma^*$, an error tolerance $0 < \varepsilon < 1$, and $0 < \delta < 1$, and outputs a number $\widehat{f(x)} \in \mathbb{N}$ such that

$$\Pr[(1-\varepsilon)f(x) \leq \widehat{f(x)} \leq (1+\varepsilon)f(x)] \ \geq 1-\delta.$$

The algorithm must run in time polynomial in $|x|$, $1/\varepsilon$ and $\log(1/\delta)$.

- For example, given $\varepsilon = 0.1$, we would have

$$0.9 \leq \frac{\widehat{f(x)}}{f(x)} \leq 1.1$$

  with high probability.

- Given $|x|$, ε can be an inverse polynomial of $|x|$, and δ can be inversely exponential in $|x|$ (negligibly small).

All or nothing theorem

Theorem

For any self-reducible counting problem, if there exists a polynomial-time randomized algorithm that solves the problem within a polynomial factor, then there exists an fpras for it.

All or nothing theorem

Theorem

For any self-reducible counting problem, if there exists a polynomial-time randomized algorithm that solves the problem within a polynomial factor, then there exists an fpras for it.

For self-reducible counting problems,

1. the distinction between polynomial, logarithmic, constant and $(1 \pm \varepsilon)$-approximation is irrelevant,
2. "approximability" means "there is an fpras".

Approximable counting problems

Some examples of counting problems that admit an fpras are the following:

- #DNF: count the satisfying assignments of a formula in disjunctive normal form.
- #BIPARTITE PERFECT MATCHINGS: count the perfect matchings in a bipartite graph.
- #NFA: count the strings of length n that are accepted by an NFA M, where the input is 1^n and an encoding of M.

Approximable counting problems

Some examples of counting problems that admit an fpras are the following:

- $\#\mathrm{DNF}$: count the satisfying assignments of a formula in disjunctive normal form.
- $\#\textsc{Bipartite Perfect Matchings}$: count the perfect matchings in a bipartite graph.
- $\#\mathrm{NFA}$: count the strings of length n that are accepted by an NFA M, where the input is 1^n and an encoding of M.

Most known approximable counting problems have a decision version in P.

There are approximable counting problems with decision version in RP.

Inaproximable counting problems

If any of the following counting problems has an fpras then RP = NP.

- #SAT.
- #2SAT.
- #MONOTONE SAT.
- #INDEPENDENT SETS.

Inaproximable counting problems

If any of the following counting problems has an fpras then RP $=$ NP.

- $\#$SAT.
- $\#$2SAT.
- $\#$MONOTONE SAT.
- $\#$INDEPENDENT SETS.

NP-complete problems have inapproximable counting versions (unless RP $=$ NP).

But there are also problems in P with inapproximable counting versions (unless RP $=$ NP).

Overview

1 Counting problems: The class #P

2 Approximation of counting problems

3 Counting problems with easy decision version: the class #PE

4 Counting problems with easy decision and self-reducibility: the class TotP

Decision version of a counting problem

Given a counting problem, what is its decision version?

- Given $\#\textsc{Sat}$, its decision version is the problem of deciding "Is $\#\textsc{Sat}(\phi) > 0$?", where ϕ is some CNF formula.
  In other words, this is the NP problem $\textsc{Sat}$.
- Given $f \in \#\mathsf{P}$, its decision version is the problem of deciding whether $f(x) > 0$ on any input $x \in \Sigma^*$.
- For any $f \in \#\mathsf{P}$, its decision version is in NP.

The class #PE (#PEasy)

Definition (Pagourtzis 2001)

A function $f : \Sigma^* \to \mathbb{N}$ belongs to #PE if it is in #P and has a decision version in P.

Overview

1. Counting problems: The class #P
2. Approximation of counting problems
3. Counting problems with easy decision version: the class #PE
4. Counting problems with easy decision and self-reducibility: the class TotP

The class TotP (1)

We are interested in a subclass of $\#$PE, which is called TotP.

Definition (Kiayias, Pagourtzis, Sharma & Zachos 2001)

$f \in$ TotP iff there is an NPTM M s.t. for every $x \in \Sigma^*$:

$$f(x) = \#(\text{all paths of } M \text{ on input } x) - 1.$$

The class TotP (1)

We are interested in a subclass of #PE, which is called TotP.

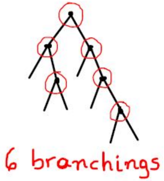
Definition (Kiayias, Pagourtzis, Sharma & Zachos 2001)

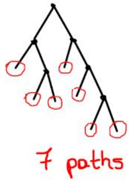
$f \in$ TotP iff there is an NPTM M s.t. for every $x \in \Sigma^*$:

$$f(x) = \#(\text{all paths of } M \text{ on input } x) - 1.$$

Note: We can consider that M has a binary computation tree. Then

$$f(x) = \#\ (\text{all branchings of M on input } x).$$



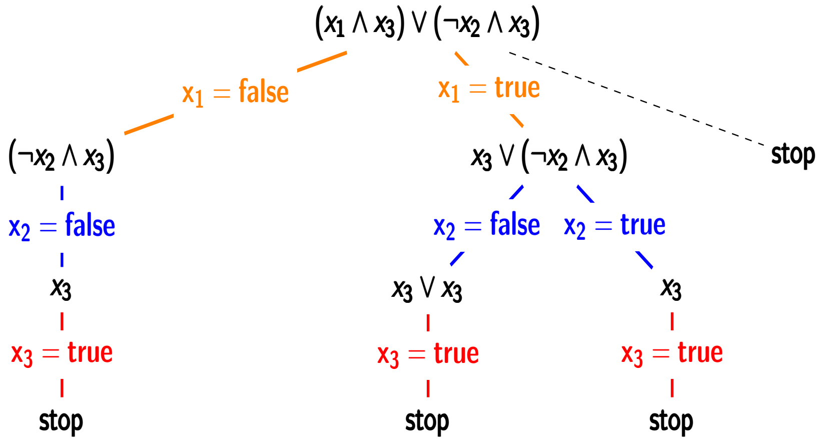


The class TotP (2)

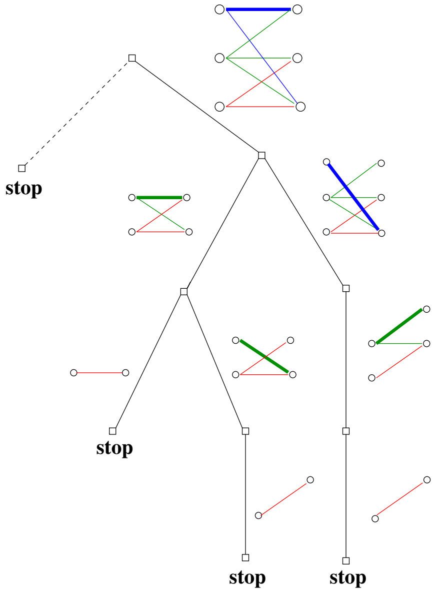
Proposition (Pagourtzis & Zachos 2006)

TotP is the Karp closure of self-reducible #P functions with decision version in P.

self-reducibility & easy decision $\Rightarrow$ membership in TotP (1)



self-reducibility & easy decision ⇒ membership in TotP (2)



Non-Uniform Complexity

Stathis Zachos

Non-Uniform Circuit Families

- We can define non-uniform circuit families, without requiring the existence of an algorithm contructing C_n given n.

Definition

Let $T : \mathbb{N} \to \mathbb{N}$ a constructible complexity function. A language L is in the class $\mathbf{P}_{/\mathbf{poly}}$ if it can be decided by a family of circuits of polynomial size.

Non-Uniform Circuit Families

- The computation of a TM deciding a language in **P**, on an input x, can be encoded as a circuit of polynomial size, hence **P** is a subset of $\mathbf{P}_{/\mathbf{poly}}$.

Non-Uniform Circuit Families

- The computation of a TM deciding a language in **P**, on an input x, can be encoded as a circuit of polynomial size, hence **P** is a subset of $\mathbf{P}_{/\mathbf{poly}}$.
- But, every unary language belongs to $\mathbf{P}_{/\mathbf{poly}}$ (*exercise*). Let's consider this language, which is closely related to Halting Problem:

$$U_H = \{1^n \mid n \text{ encodes a pair } (M, x) \text{ such that } M(x) \downarrow\}$$

Non-Uniform Circuit Families

- The computation of a TM deciding a language in **P**, on an input x, can be encoded as a circuit of polynomial size, hence **P** is a subset of $\mathbf{P}_{/\mathbf{poly}}$.
- But, every unary language belongs to $\mathbf{P}_{/\mathbf{poly}}$ (*exercise*). Let's consider this language, which is closely related to Halting Problem:
  $$U_H = \{1^n \mid n \text{ encodes a pair } (M, x) \text{ such that } M(x) \downarrow\}$$
- U_H is a unary language, so it belongs to $\mathbf{P}_{/\mathbf{poly}}$, but it is *not* decidable (we can reduce Halting Problem to it). So, the inclusion is *proper*:

Theorem

$\mathbf{P} \subsetneqq \mathbf{P}_{/\mathbf{poly}}$

Other Properties of $\mathbf{P}_{/\mathbf{poly}}$

Theorem (Karp-Lipton)

If $\mathbf{NP} \subseteq \mathbf{P}_{/\mathbf{poly}}$, *then* $\mathbf{PH} = \Sigma_2^p$

Theorem (Meyer)

If $\mathbf{EXP} \subseteq \mathbf{P}_{/\mathbf{poly}}$, *then* $\mathbf{EXP} = \Sigma_2^p$

Theorem

$\mathbf{BPP} \subsetneq \mathbf{P}_{/\mathbf{poly}}$

Lower Bounds

- $\mathbf{P}_{/\mathbf{poly}}$ is closely related to the **P** *vs* **NP** problem, since if one can find a language in **NP** which is not in $\mathbf{P}_{/\mathbf{poly}}$, then $\mathbf{P} \neq \mathbf{NP}$.

- This point of view commenced a research program during the 80s, trying to construct explicitely such a language, and since $\mathbf{P}_{/\mathbf{poly}}$ is a wide class, researchers also studied the bounded-depth subclasses of $\mathbf{P}_{/\mathbf{poly}}$, such as (non-uniform) **NC**, **AC**, **ACC** variants.

Lower Bounds

- Such an example is $\mathbf{ACC}^0[m]$, the non-uniform analogue of $\mathbf{AC}^0$, with the extra use of MOD-counting gates (gates that output 0 iff the modulo sum of all inputs x_i equals to 0 ($\sum x_i \mod m = 0$).

- The most developed techniques for proving lower bounds for these subclasses were the Random restriction method and the Polynomial method.

Random restriction method

- By using the Random restriction method, Furst, Saxe, Sipser, Ajtai proved the following:

Theorem

$\mathit{PARITY} \notin \mathbf{AC}^0$.

- Håstad improved the aforementioned result by showing that circuits of depth d need $2^{\Omega(n^{1/(d-1)})}$ size to compute *PARITY*.

Polynomial Method

- Razborov and Smolensky showed that every circuit of bounded depth that computes a language in $\mathbf{ACC}^0[m]$ can be probabilistically represented by a low-degree polynomial in $\mathbb{F}_2$.
- On the other side, they proved the existence of functions that cannot be represented by low-degree polynomials, even with high probability, hence they cannot by computed by $\mathbf{ACC}^0[m]$ circuits.

Theorem (Razborov-Smolensky)

For distinct primes p and q, MOD_p function is not in $\mathbf{ACC}^0[q]$.

Monotone Circuits

- Another line of research was the restricion of circuit families to *monotone* circuits, that is circuits without NOT gates (inverters). For monotone circuit families, we have the following lower bound for the Clique problem:

Theorem (Razborov-Andreev-Alon-Boppana)

There is an $\varepsilon > 0$, such that for all $k \leq n^{1/4}$ the k-clique problem cannot be computed by monotone circuits of size $2^{\varepsilon\sqrt{k}}$.

Lower Bounds

- Recently, new significant circuit lower bounds for **NEXP** were proven. The proof techniques are related to the Circuit Satisfiability problem:

  Given a circuit C_n, is there an $x \in \{0,1\}^n$ such that $C_n(x) = 1$?

- The obvious algorithm is to (brute-force) check all 2^n possible inputs of length n, and in most cases is the best algorithm we know.

- Any improvement to such algoriths will lead to new lower bounds for **NEXP**. Such an example is the following theorem:

Lower Bounds

Theorem

Let $s(n)$ be a superpolynomial function. If Circuit Satisfiability problem for circuits of n inputs and $poly(n)$ size can be solved in time $2^n \cdot poly(n)/s(n)$, then $\mathbf{NEXP} \not\subseteq \mathbf{P}_{/\mathbf{poly}}$.

- The above theorem holds for any "natural" subclass of $\mathbf{P}_{/\mathbf{poly}}$, eg $\mathbf{ACC}^0$.
- Recent advances in $\mathbf{ACC}^0$-SAT algorithms combined with the $\mathbf{ACC}^0$-version of the above theorem, proved new (and *unconditional*) lower bounds for $\mathbf{ACC}^0$:

Theorem

$$\mathbf{NEXP} \not\subseteq \mathbf{ACC}^0$$

where $\mathbf{ACC}^0 = \bigcup_{(m_1,\ldots,m_l)} \mathbf{ACC}^0[m_1,\ldots,m_l]$

Descriptive Complexity

Stathis Zachos

Descriptive Complexity

What is the type of logic that is needed to express the problems of a complexity class?

Instances are encoded by finite relational structures

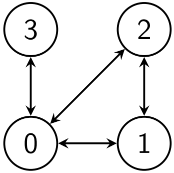
The input to any computational problem can be seen as a finite relational structure.

Let $\tau = \langle P^{a_1}, R^{a_2}, Q^{a_3}, \ldots \rangle$. A structure over τ looks like:

$$\mathcal{A} = \langle A, P^{\mathcal{A}}, R^{\mathcal{A}}, Q^{\mathcal{A}}, \ldots \rangle.$$

Graphs as finite relational structures

Graph G



$$\text{Vocabulary } \tau = \langle E^2 \rangle$$

$$\mathcal{G} = \langle V, E \rangle, \text{ where}$$

$$V = \{0,1,2,3\}, E = \{(0,1),(1,0),\ldots\}$$

Then $\mathcal{G} \models (\forall x, y)\Big[\neg E(x,x) \wedge \big(E(x,y) \leftrightarrow E(y,x)\big)\Big]$ says that graph G is irreflexive and symmetric.

Fagin's theorem

Theorem (Fagin 1973)

$\exists$SO= NP over finite structures: For any language L, $L \in$ NP iff it is definable by an existential second-order sentence.

Examples

- Does G have a 3-coloring?

$$G \in \text{3-}\textsc{Coloring} \text{ iff}$$

$$\mathcal{G} \models (\exists R)(\exists B)(\exists G)(\forall x)\Big[\big(R(x) \vee B(x) \vee G(x)\big) \wedge \\ (\forall y)\Big(E(x,y) \to \neg(R(x) \wedge R(y)) \wedge \neg(B(x) \wedge B(y)) \wedge \neg(G(x) \wedge G(y))\Big)\Big]$$

- Does G contain a clique (of any size)?

$$G \in \textsc{Clique} \quad \text{iff} \quad \mathcal{G} \models \exists X \forall x \forall y \big(X(x) \wedge X(y) \wedge x \neq y\big) \to E(x,y).$$

An overview

Over **all** *finite structures:*

- $\exists$SO$=$ NP: Existential second-order logic captures NP. (Fagin 1974)

Over finite **ordered** *structures:*

- FO(LFP) $=$ P: First-order logic with least fixed point captures P. (Immerman – Vardi 1982)
- FO(PFP) $=$ PSPACE: First-order logic with partial fixed point captures PSPACE. (Vardi 1982)
- FO(TC) $=$ NL: First-order logic with the transitive closure operator captures NL. (Immerman 1986)

P versus NP

Conjecture (Gurevich)

There is no logic that captures P over the class of all finite structures.

- This a very strong conjecture, since there is such a logic for NP (by Fagin's theorem).
- So proving this comjecture would imply that P $\neq$ NP.